A first course in randomized algorithms

Nick Harvey
University of British Columbia

April 17, 2025

Contents

Preface

I was always enthusiastic about algorithms. The standard undergraduate curriculum covers many algorithms, most of which are clever, many of which are practical, and some of which are both. When I first saw *randomized algorithms* I was dazzled. They seemed like a collection of magic tricks with astonishing levels of simplicity and elegance.

In modern computer science, probabilistic ideas are pervasive. For example, randomization is essential for preserving secrecy and privacy. Sampling methods are crucial for efficiently analyzing enormous data sets, as is done by streaming algorithms. Coordinating and organizing agents in a distributed system often relies on randomization. Quantum algorithms are invariably randomized because every quantum measurement has probabilistic outcomes.

In this book we will see numerous examples of randomized algorithms and the mathematical techniques for their analysis. I hope that these fundamental ideas will be empowering to students across the spectrum of computer science.

Part I

Basics

Chapter 1

Introduction

You usually don't know whether an obsession is a great quest or a great folly until it's over.

Shankar Vedantam

1.1 An introductory example

Randomization often allows us to design simple and surprising algorithms. Let us begin with a simple example that demonstrates this point. The problem is to determine: *are all entries of a vector zero?* In more detail, let $x = (x_1, x_2, \ldots, x_n)$ be a vector of n bits, with n even. Suppose we know that there are two cases.

All-zero case. All entries of x are 0.

Half-zero case. Exactly half the entries in x are 0 and half are 1. However, we do not know in advance *which* bits are 0.

We would like to design an algorithm that examines some of the bits and decides which of these two cases holds.

This problem can be solved by a trivial ***deterministic*** algorithm, meaning one without randomness. It examines all n bits in turn, and returns "Yes" if they are all zero; otherwise it returns "No". This algorithm is guaranteed to be correct for every possible input.

---

function IsZero(bit vector $x = (x_1, \ldots, x_n)$)
    for $i = 1, \ldots, n$
        if $x_i = 1$ **then return** "No"
    return "Yes"
end function

Question 1.1.1. Can you improve this algorithm so that it examines fewer than n bits?

Answer.

Yes, the algorithm can be improved if $n > 2$. The algorithm simply examines the first $(n/2) + 1$ bits (or any $(n/2) + 1$ bits) instead of examining all of them.

Problem to Ponder 1.1.2. Is there a deterministic algorithm that uses $o(n)$ bits?

1.1.1 A randomized algorithm

There is an intriguing way to apply randomization to this problem: simply pick an index uniformly at random, and examine the corresponding bit of the input vector. Whereas the deterministic algorithm examined $\Theta(n)$ bits, this algorithm only examines a *single* bit.

Algorithm 1.1 A randomized algorithm to test if all entries of a vector are zero. The notation $[n]$ means the set $\{1, \ldots, n\}$.

```
1: function IsZero(bit vector x = (x_1, ..., x_n))
2:     Pick J ∈ [n] uniformly at random
3:     if x_J = 1 then return "No" else return "Yes"
4: end function
```

Observe that line 2 requires us to pick a uniformly random number. At this point it may not be clear how that step can be implemented. We will discuss methods for sampling random numbers in the next chapter; for now we will focus on the correctness of this algorithm.

Whereas the deterministic algorithm is always correct, the randomized algorithm can sometimes make an error. Let us now clarify what could mean.

- *Random inputs.* In several contexts it is common to assume that the *input data* is drawn randomly. For example, this viewpoint is typically used in statistics, machine learning, and the average-case analysis of algorithms. When it is claimed that the algorithm is unlikely to make an error, the claim is that the algorithm is correct *for most* inputs. The analysis is typically a statement of the form

  $$\Pr_{\text{input } x}[\,\text{algorithm makes an error on input } x\,] \text{ is small.}$$

  Here the subscript "input x" indicates that the only source of randomness is the randomly chosen input.

- *Worst-case analysis.* In theoretical computer science, and in this book, it is common to adopt the viewpoint of "worst-case analysis", which means that the analysis must hold *for all* inputs. For deterministic algorithms, this would usually mean that the algorithm is efficient and correct for all inputs. In contrast, for randomized algorithms, the analysis is typically a statement of the form

  $$\text{for every input } x,\ \Pr_{\text{algorithm's random choices}}[\,\text{algorithm makes an error on input } x\,] \text{ is small.}$$

  Here the subscript emphasizes that the only source of randomness is the internal random choices of the algorithm. In particular, the input x is not random.

Throughout this book we will almost always adopt the viewpoint of *worst-case analysis*, and will not assume that the input data is random. Thus, when we consider the probability that the algorithm makes an error, this is with respect to the internal random choices of the algorithm, namely the choice of J on line 2.

We analyze the probability of making an error by separately considering two cases.

Case 1. The first case is that the input x is all-zeros. It is clear that x_J will always equal 0, so the algorithm will always return "Yes", which is the correct output. Thus, there is zero probability of making an error.

Case 2. The second case is that the input x is half-zeros. In this case the correct output is "No", so the algorithm makes an error if and only if it chooses an index J for which $x_J = 0$. So we must analyze $\Pr[\, x_J = 0 \,]$.

Let us reiterate that "$x_J = 0$" is indeed a random event. Although the vector x is *not* random, the index J *is* random. Thus, the bit x_J is a random variable, and "$x_J = 0$" is an event whose probability we can discuss. In contrast, x_1 is *not* random, so discussing $\Pr[\, x_1 = 0 \,]$ is technically valid, but a bit silly.

Question 1.1.3. In the all-zeros case, what is $\Pr[\, x_1 = 0 \,]$?

Answer.

This probability is 1.

Question 1.1.4. In the half-zeros case, what can be said about $\Pr[\, x_1 = 0 \,]$?

Answer.

Depending on the vector x, this probability might be 0 or might be 1. In fact, $\Pr[\, x_1 = 0 \,] = 1 - x_1$.

Continuing the analysis of Case 2, let $Z = \{\, i \in [n] \,:\, x_i = 0 \,\}$ be the indices of the zero entries in x. Then the probability of error is

$$\begin{aligned}
\Pr[\, \text{algorithm makes an error} \,] &= \Pr[\, x_J = 0 \,] \\
&= \Pr[\, J \in Z \,] \\
&= \frac{|Z|}{n} && \text{(by (A.3.3), since } J \text{ is uniformly distributed)} \\
&= \frac{n/2}{n} && \text{(due to the half-zero case)} \\
&= \frac{1}{2}.
\end{aligned}$$

To conclude, we have shown that

$$\begin{aligned}
\text{if } x \text{ is all-zeros} &\;\Rightarrow\; \Pr_{\text{algorithm's random choices}}[\, \text{algorithm makes an error on input } x \,] = 0 \\
\text{if } x \text{ is half-zeros} &\;\Rightarrow\; \Pr_{\text{algorithm's random choices}}[\, \text{algorithm makes an error on input } x \,] = 1/2.
\end{aligned}$$

The randomized algorithm is significantly faster than the deterministic algorithm, since it only examines a single bit. However, this comes at the cost of making an error with probability at most $1/2$. An error probability of $1/2$ might seem fairly large, but we will discuss below how to decrease it.

Broader context. Although the specific problem that we have studied might seem contrived, it is actually quite fundamental. It is known as the Deutsch-Jozsa problem, and it has been studied since the 1980s. The reason that this problem is interesting is that quantum computers can solve it by examining the vector x just once, while suffering *no error at all*. This is better than can be achieved with deterministic or randomized algorithms.

1.2 Different types of error

In theoretical computer science, problems with Yes/No outputs are called decision problems. Sometimes we say that these problems have Boolean outputs, or True/False outputs. The problem considered in the previous section, deciding whether a given vector is all zeros, is an example of a decision problem. It is useful to consider the various outcomes that can occur in randomized algorithms for decision problems.

Correct Output \ Algorithm's Output	Yes	No
Yes	true positive	false negative
No	false positive	true negative

Figure 1.1: The possible outcomes and their conventional names.

These sorts of tables are commonly used in statistical tests or medical diagnostic tests, and there is a lot of jargon relating to them.

- A ***false positive*** is when the correct output is No, but the algorithm outputs Yes. In statistical hypothesis testing, this is also called a "Type I error".
- A ***false negative*** is when the correct output is Yes, but the algorithm outputs No. In statistical hypothesis testing, this is also called a "Type II error".

In the current terminology of machine learning, the table itself is called a confusion matrix.

There are some simple relationships between the probabilities of the outcomes of the algorithm. If we assume that the algorithm always terminates, then it must either output Yes or No. Thus, we have

$$\text{for every input } x \text{ whose correct output is Yes,}$$
$$\Pr[\underbrace{\text{algorithm outputs True on input } x}_{\text{"true positive"}}] + \Pr[\underbrace{\text{algorithm outputs False on input } x}_{\text{"false negative"}}] \;=\; 1.$$

Similarly,

$$\text{for every input } x \text{ whose correct output is No,}$$
$$\Pr[\underbrace{\text{algorithm outputs True on input } x}_{\text{"false positive"}}] + \Pr[\underbrace{\text{algorithm outputs False on input } x}_{\text{"true negative"}}] \;=\; 1.$$

Question 1.2.1. Consider Algorithm 1.1 above. What are the false positive and false negative probabilities?

Answer.

The false positive probability is $1/2$. The false negative probability is 0.

In general, the true/false positive/negative probabilities are not fixed values, because they typically depend on the input x. Thus, it is a bit informal to say "the false positive probability". Nevertheless, in Algorithm 1.1 it just so happens that these probabilities do not depend on the input, because we have assumed the all-zero or half-zero cases.

Problem to Ponder 1.2.2. Find an algorithm and two inputs that have differing false positive probabilities.

1.2.1 Some types of randomized algorithms

Depending on the probabilities of the false positives and negatives, we might judge a randomized algorithm to be "good". In this section we define a few common criteria that can inform our judgements.

We will ignore the algorithm's runtime, or any other resource usage, and just focus on its probability of correctness. We will also assume that the algorithm cannot run forever: for all inputs, it must always terminate with a Yes/No output.

It is a very fortunate situation if an algorithm has either false positive probability or false negative probability equal to 0. Such an algorithm is said to have ***one-sided error***. The reason why this is useful will be explained in Section 1.3.

We will now describe three types of randomized algorithms that frequently arise. The first type is an ***RP-type algorithm***.

Correct Output \ Algorithm's Output	Yes	No
Yes	$\geq 1/2$ true positive	$\leq 1/2$ false negative
No	0 false positive	1 true negative

Figure 1.2: Probabilities of outcomes for an **RP-type algorithm**.

References: (Motwani and Raghavan, 1995, Sections 1.2 and 1.5.2), (Sipser, 2012, Definition 10.10), Wikipedia

The interpretation of this table is as follows. For *every* input whose correct output is No, the algorithm must always output No. Since the algorithm has no false positives, it has ***one-sided error***. For *every* input whose correct output is Yes, the algorithm must output Yes with probability at least $1/2$, where the probability is over the internal randomization of the algorithm. Once again, let us emphasize that there are no assumptions regarding any distribution on the inputs.

The constant $1/2$ in the definition of RP-type algorithms is not crucial. It would make very little difference if we changed the bound on the false negatives from $1/2$ to any other constant strictly between 0 and 1. This is due to ***amplification***, which is an important idea thas is used heavily throughout the book. Further discussion of amplification is in Section 1.3 below.

Question 1.2.3. Suppose we required instead that true positives have probability $\leq 1/2$ and false negatives had probability $\geq 1/2$. Would this definition make sense too?

Answer.

This would not be a useful definition. The algorithm that ignores the input and always returns No would satisfy this definition.

A ***coRP-type algorithm*** is analogous to an RP-type algorithm except that it makes no errors if the correct output is Yes. Since the algorithm has no false negatives, it also has ***one-sided error***.

Correct Output \ Algorithm's Output	Yes	No
Yes	1 true positive	0 false negative
No	$\leq 1/2$ false positive	$\geq 1/2$ true negative

Figure 1.3: Probabilities of outcomes for a **coRP-type algorithm**.

Of course, it is not always the case that an algorithm will have one-sided error. Those with both false positives and false negatives are said to have ***two-sided error***. For such algorithms, a useful definition is a ***BPP-type algorithm***.

Correct Output \ Algorithm's Output	Yes	No
Yes	$\geq 2/3$ true positive	$\leq 1/3$ false negative
No	$\leq 1/3$ false positive	$\geq 2/3$ true negative

Figure 1.4: Probabilities of outcomes for a **BPP-type algorithm**.

References: (Motwani and Raghavan, 1995, Section 1.5.2), (Sipser, 2012, Definition 10.4), Wikipedia

It would make very little difference if we changed the constant $2/3$ in this table to any other constant strictly between $1/2$ and 1. This statement is not proven until Section 10.2 because it requires more sophisticated techniques than the results of Section 1.3.

Question 1.2.4. Suppose that an algorithm has no false positives and no false negatives, i.e., "zero-sided error". Does that mean that it is a deterministic algorithm?

Answer.

No. It could be the case that the algorithm makes no errors, but yet its runtime is a random quantity. For a silly example, an algorithm could test graph connectivity by flipping a coin. If heads, it performs breadth-first search; if tails, it performs exhaustive search. It is always correct, but its runtime is random.

Question 1.2.5. Consider our example of testing if a bit vector is zero. Is this an RP-type algorithm, a coRP-type algorithm, or a BPP-type algorithm?

Answer.

It is a coRP-type algorithm because we have observed in Question 1.2.1 that the false positive probability is $1/2$ and the false negative probability is 0.

1.3 Probability amplification, for one-sided error

A nice feature of algorithms with one-sided error is that it's easy to amplify their probability of success. Suppose that $\mathcal{B}$ is an algorithm with $\Pr[\text{false positive}] \leq 1/2$ and $\Pr[\text{false negative}] = 0$ (a coRP-type algorithm). We design an "amplified" algorithm $\mathcal{A}$ by performing repeated trials of $\mathcal{B}$. If any trial outputs "No" then that is definitely the correct output (because there are no false negatives). If all trials output "Yes", then most likely that is the correct output.

Algorithm 1.2 The algorithm $\mathcal{A}$ for amplifying success of algorithm $\mathcal{B}$. The parameter ℓ determines the number of trials.

```
function AMPLIFY(algorithm B, input x, integer ℓ)
    for i = 1, ..., ℓ
        Run B(x) (with new, independent randomness for its internal random choices)
        if B returns No then return No                ▷ This is definitely correct
    return Yes                  ▷ B's outputs were all Yes, so this is probably correct
end function
```


Theorem 1.3.1. Assume that $\mathcal{B}$ is a coRP-type algorithm, meaning that it satisfies

$$\Pr[\text{false positive}] \leq 1/2 \qquad \text{and} \qquad \Pr[\text{false negative}] = 0.$$

Then $\mathcal{A}$ satisfies

$$\Pr[\text{false positive}] \leq 1/2^{\ell} \qquad \text{and} \qquad \Pr[\text{false negative}] = 0.$$

Proof. There are two cases.

Case 1: The correct answer is Yes. Since $\mathcal{B}$ has no false negatives it will always output Yes. From the pseudocode it is clear that $\mathcal{A}$ will output Yes too. This shows that $\mathcal{A}$ has no false negatives.

Case 2: The correct answer is No. $\mathcal{A}$ will only make a mistake if *all* executions of $\mathcal{B}$ make a mistake.

$$\begin{aligned}
&\Pr[\mathcal{A} \text{ incorrectly outputs Yes}] \\
&\quad = \Pr[\text{in all } \ell \text{ trials, } \mathcal{B} \text{ incorrectly outputs Yes}] \\
&\quad = \prod_{i=1}^{\ell} \Pr\left[\text{in } i^{\text{th}} \text{ trial, } \mathcal{B} \text{ incorrectly outputs Yes}\right] && \text{(the trials are independent)} \\
&\quad \leq \prod_{i=1}^{\ell} (1/2) && \text{(since } \mathcal{B} \text{ is a coRP-type algorithm)} \\
&\quad = 1/2^{\ell}. \quad \square
\end{aligned}$$

Question 1.3.2. Suppose we modified the hypotheses of the theorem by assuming that algorithm $\mathcal{B}$ has $\Pr[\text{false positive}] \leq 0.9$. What can we say about the false positives of algorithm $\mathcal{A}$?

Answer.

For algorithm $\mathcal{A}$, the same proof would show that $\Pr[\text{false positive}] \leq 0.9^{\ell}$. This does not decrease as fast as $1/2^{\ell}$ but it still decreases exponentially fast, which is good.

Amplification for the zero-testing problem. Recall that Algorithm 1.1 above is a coRP-type algorithm, so it satisfies the hypotheses of Theorem 1.3.1. For any $k > 0$, if we run the amplification algorithm with $\ell = \lceil \lg(k) \rceil$, then it examines only ℓ entries of the bit vector x, and the failure probability decreases to at most $1/2^{\ell} \leq 1/k$.

1.4 Exercises

Exercise 1.1. Suppose that we want to solve a certain decision problem. We know a RP-type algorithm $\mathcal{A}$ for this problem with runtime $f(n)$. Describe a BPP-type algorithm for this problem with runtime $O(f(n))$.

Exercise 1.2. Suppose that algorithm $\mathcal{B}$ has

$$\Pr[\text{false positive}] \leq 0.99 \qquad \text{and} \qquad \Pr[\text{false negative}] = 0.$$

Let δ be any constant satisfying $0 < \delta < 1$. Prove that we can choose $\ell = O(\log(1/\delta))$ so that the algorithm $\textsc{Amplify}(\mathcal{B}, x, \ell)$ satisfies

$$\Pr[\text{false positive}] \leq \delta \qquad \text{and} \qquad \Pr[\text{false negative}] = 0.$$

Exercise 1.3. Let A be an unsorted array of n distinct numbers, where n is a multiple of 4. Let us say that an element is "in the middle half" if there are at least $n/4$ elements less than it, and at least $n/4$ elements greater than it.

Part I. Describe a randomized algorithm that runs in $O(1)$ time and outputs an element of A. The probability that the output is *not* in the middle half should be at most 0.5. You may assume that you can generate a random number in $O(1)$ time.

In a later chapter, we will see that this probability can be decreased to 0.01.

Part II. Describe a randomized algorithm that runs in $O(n)$ time and that either outputs an element of A in the middle half, or outputs "fail". It should fail with probability at most 0.5.

Part III. Describe a randomized algorithm that runs in $O(n)$ time and that either outputs an element of A in the middle half, or outputs "fail". It should fail with probability at most 0.01.

Exercise 1.4. The all-zeros or half-zeros assumptions in Section 1.1 might seem contrived. Consider the following example.

Define the following two functions.

- $\textsc{Parity}(y)$. Given an integer y, returns 1 if its binary representation has an odd number of 1 bits, and otherwise returns 0.
- $\textsc{BitwiseAnd}(y,z)$. Given two integers y and z, it returns the integer obtained from the bitwise Boolean And of their binary representations.

Part I. Write a program that does the following. For every integer $y \in [\![32]\!]$, create the bitvector x where $x_i = \textsc{Parity}(\textsc{BitwiseAnd}(y, i))$ for $i \in [\![32]\!]$, then output the number of zeros in x.

Part II. Let k be an arbitrary integer and $y \in [\![2^k]\!]$. Define the bitvector x by

$$x_i = \textsc{Parity}(\textsc{BitwiseAnd}(y, i)) \quad \forall i \in [\![2^k]\!].$$

Give a formula for the number of zeros in x, and prove that it is correct.

Chapter 2

Sampling numbers

It is quite common for computer programs to invoke a function that generates random numbers. The source of this randomness might be a hardware device that relies on physical properties, such as heat, or it might be a pseudo-random number generator that is completely deterministic and only seems random. Either way, there may be various sorts of random numbers of interest. For example, one might want to generate an unbiased random bit, a biased random bit, a uniform integer from a certain range, a uniform real number from a certain range, etc.

In this chapter, we will assume that we are given a basic random number generator, then show how we can build upon that to generate other random values. Figure 2.1 illustrates the different methods and the subroutines that they use.

2.1 Uniform random variables

Perhaps the simplest random variable to generate is an unbiased random bit. This is also called a fair coin, a uniform $\{0, 1\}$ RV, or a Bernoulli RV with parameter $1/2$. In this section we will make one key assumption:

> *there is a function* UnbiasedBit *that returns a random bit that is unbiased, and independent from all previous bits.*

Using UnbiasedBit as a subroutine, we will show how to generate various types of random variables.

2.1.1 Uniform ℓ-bit integers

Suppose we want to generate a uniform ℓ-bit integer. We can do so by constructing its binary representation to be a sequence of ℓ independent, unbiased bits.

```
function UniformLBitInteger(int ℓ)
    for i = 1, ..., ℓ
        X_i ← UnbiasedBit()
    return ⟨X_ℓ ... X_1⟩          ▷ The integer whose binary representation is X_ℓ, ..., X_1.
end function
```

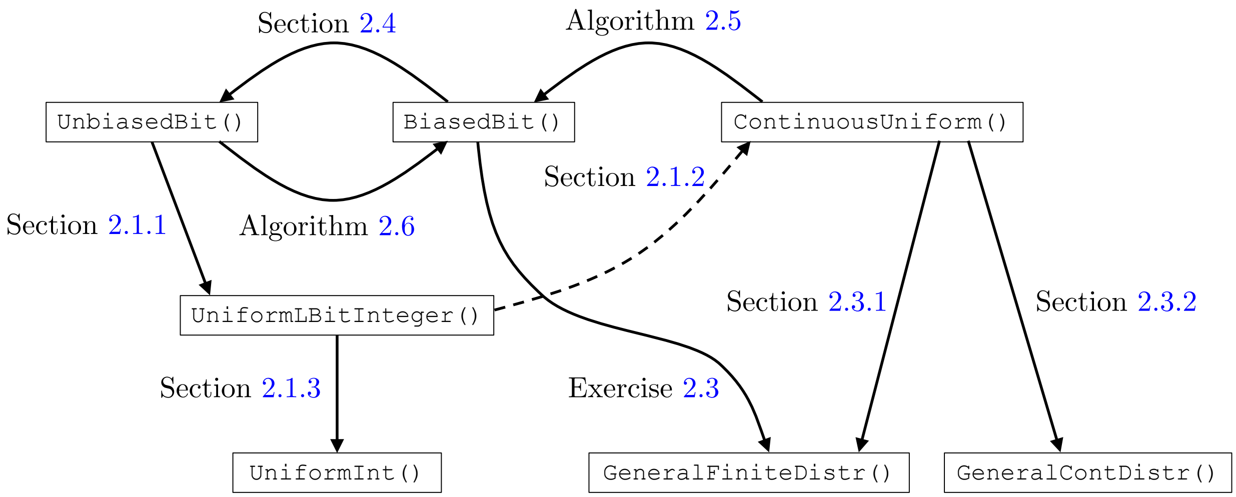


Figure 2.1: Several algorithms for sampling random numbers. Each arrow indicates that the algorithm at the head can be implemented using the algorithm at the tail. The dotted line indicates that the algorithm for ContinuousUniform can only *approximately* be implemented using UniformLBitInteger.

The output of this algorithm is an ℓ-bit integer, which we claim is uniformly distributed. To see this, note that there are 2^ℓ different ℓ-bit integers, so each of them should be produced with probability $2^{-\ell}$ (see Section A.3.3). This is easy to check. For any particular integer $\langle x_\ell \ldots x_1 \rangle$, we have

$$\begin{aligned}
&\Pr[\,\text{output is } \langle x_\ell \ldots x_1 \rangle\,] \\
&\quad= \Pr[\,X_1 = x_1 \wedge X_2 = x_2 \wedge \cdots \wedge X_\ell = x_\ell\,] \\
&\quad= \Pr[\,X_1 = x_1\,] \cdot \Pr[\,X_2 = x_2\,] \cdots \Pr[\,X_\ell = x_\ell\,] && (\textsc{UnbiasedBit} \text{ returns independent bits}) \\
&\quad= \frac{1}{2} \cdot \frac{1}{2} \cdots \frac{1}{2} && (\textsc{UnbiasedBit} \text{ returns unbiased bits}) \\
&\quad= 2^{-\ell}.
\end{aligned}$$

2.1.2 Continuous random variables

A real number in the interval $[0, 1]$ would typically take an infinite number of bits to represent, so we cannot expect to do so exactly on a computer. If we use fixed-point arithmetic instead of floating point arithmetic, we could use ℓ bits of precision and approximately represent the number as an integer multiple of $2^{-\ell}$. Using this idea, we can approximate continuous uniform random variables as follows.

Algorithm 2.1 Generating a uniform random variable on $[0, 1]$.

global ℓ ▷ Number of bits of precision
function ContinuousUniform
    $X \leftarrow$ UniformLBitInteger(ℓ)
    return $X \cdot 2^{-\ell}$
end function

Naturally the output of ContinuousUniform only approximately satisfies the properties of a continuous uniform RV. For example, an actual uniform RV has probability 0 of taking any fixed value (see Fact A.3.16), whereas ContinuousUniform outputs each multiple of $2^{-\ell}$ in $[0,1)$ with probability $2^{-\ell}$. Of course this probability can be made arbitrarily small by increasing ℓ, so ContinuousUniform is still useful in many applications.

2.1.3 Uniform on any finite set

Suppose that we want to generate a random variable that is uniformly distributed on any finite set. Actually, it suffices to generate a random variable that is uniform on

$$[\![n]\!] \;=\; \{0, 1, \ldots, n-1\}$$

because any finite set has a one-to-one correspondence with such a set. If n is power of two, then $[\![n]\!]$ is precisely the set of $\lg(n)$-bit integers, from which UniformLBitInteger can generate a uniformly random value. So we focus our attention on the case when n is not a power of two.

An imperfect approach. In theory, there is a straightforward solution based on *continuous* uniform random variables. This approach is commonly used in the libraries of modern programming languages, for example in Python.

Algorithm 2.2 Generating a uniform random variable on $[\![n]\!]$ using a continuous uniform random variable.

```
function UniformInt(int n)
    X ← ContinuousUniform()
    return ⌊n · X⌋
end function
```

Question 2.1.1. Why is the output uniformly distributed on $[\![n]\!]$?

Answer.

The output is i if and only if $i/n \leq X < (i+1)/n$. It follows from Fact A.3.17 that this happens with probability $1/n$.

Unfortunately, this approach is slightly flawed in practice. The problem is that the output of ContinuousUniform must be discrete, so it cannot be perfectly uniform on $[0,1]$.

Question 2.1.2. Suppose that we implement ContinuousUniform by choosing only $\ell = 2$ bits of precision. What is the distribution on the outputs of UniformInt(3)?

Answer.

The possible outputs of ContinuousUniform are $\{0, \frac{1}{4}, \frac{2}{4}, \frac{3}{4}\}$, each occurring with probability $1/4$. So the output of UniformInt(3) is 0 with probability $1/2$, whereas 1 or 2 each have probability $1/4$.

A perfectly uniform approach. There is another approach that can avoid the flaw and give a perfectly uniform random value in $[\![n]\!]$. The idea is not hard: sample from a set that is slightly larger than $[\![n]\!]$, and just discard any values that don't land in $[\![n]\!]$.

A nice way to do this is to enlarge the set by adding "dummy values" until its size is a power of two. We can then sample from the larger set using UniformLBitInteger from Section 2.1.1. This concept of ***rounding up to a power of two*** is useful in many algorithms, and it is explained in Definition A.2.3.

The larger set with the dummies will have exactly 2^ℓ items, where $\ell = \lceil \lg n \rceil$. We sample uniformly from this set by generating a uniform ℓ-bit integer. With some probability we might sample a dummy element — what should we do then? A natural idea is to retry and draw a new sample. Pseudocode implementing this idea is as follows.

Algorithm 2.3 Generating a uniform random variable on $[\![n]\!]$.

function UniformInt(int n)
  $\ell \leftarrow \lceil \lg n \rceil$
  repeat
    $U \leftarrow$ UniformLBitInteger(ℓ) ▷ Initially U is uniform on $[\![2^\ell]\!]$
  until $U \in [\![n]\!]$
  return U ▷ Now U is uniform on $[\![n]\!]$
end function

The reason why this works is quite intuitive. Suppose that U is uniformly distributed on $[\![2^\ell]\!]$. If we condition on the event $U \in [\![n]\!]$, it stands to reason that U is now uniformly distributed on $[\![n]\!]$. This is indeed true, and will be proven below. In Algorithm 2.3, the loop terminates when $U \in [\![n]\!]$, so the output is conditioned on that event. This idea is a very basic form of more general technique that we now introduce.

Rejection sampling

Rejection sampling is a technique to sample from a desired distribution using samples from another distribution, and rejecting those that are unwanted. It can be used in very general settings, such as non-uniform distributions or continuous distributions. We focus on the simple setting of uniform distributions on finite sets.

References: (Grimmett and Stirzaker, 2001, Example 4.11.5), Wikipedia.

Algorithm 2.4 Given sets $R \subseteq S$, use uniform samples on S to generate uniform samples on R.

function RejectionSample(set R, set S)
  repeat
    Let U be a RV that is uniform on S
  until $U \in R$
  return U ▷ Now U is uniform on R
end function

First we argue that RejectionSample generates uniform samples on R. Since the loop continues until $U \in R$, intuitively the output of the algorithm should be distributed like U conditioned on $U \in R$. (That is true, but proving it formally takes a bit of work — see Problem to Ponder 2.1.4.) The next claim shows that this is indeed the uniform distribution on R.

Claim 2.1.3. Let R and S be finite non-empty sets with $R \subseteq S$. Suppose that U is uniform on S. If we condition on the event $U \in R$, then U becomes uniform on R. In other words,

$$\Pr[\, U = x \mid U \in R \,] = \frac{1}{|R|} \qquad \forall x \in R.$$

Proof. Fix any $x \in R$. Observe that $U = x$ implies $U \in R$, so the event "$U = x \wedge U \in R$" is the same as the event "$U = x$".

$$\begin{aligned}
\Pr[\,U = x \mid U \in R\,] &= \frac{\Pr[\,U = x \wedge U \in R\,]}{\Pr[\,U \in R\,]} && \text{(definition of conditional probability)}\\
&= \frac{\Pr[\,U = x\,]}{\Pr[\,U \in R\,]} && \text{(as observed above)}\\
&= \frac{1/|S|}{\sum_{r \in R} 1/|S|} && \text{(by (A.3.2) and (A.3.3))}\\
&= \frac{1}{|R|}. \quad \square
\end{aligned}$$

Problem to Ponder 2.1.4. Let Θ be the output of the algorithm. Prove more formally that Θ is uniformly distributed on R.

Next we analyze the runtime of this algorithm. Each iteration generates U randomly, and returns if $U \in R$. We can think of each iteration as a random trial that succeeds if $U \in R$. The algorithm continues until a trial succeeds. The number of iterations needed is a random variable, say X, having a ***geometric distribution***. Recall that X has a parameter p, which is the probability that each trial succeeds; so

$$p \;=\; \Pr[\,U \in R\,] \;=\; \frac{|R|}{|S|}.$$

We now recall a basic fact about geometric random variables.

Fact A.3.21. Let X be a geometric random variable with parameter p. Then $\mathrm{E}[\,X\,] = 1/p$.

This discussion proves the following claim.

Claim 2.1.5. The expected number of iterations of RejectionSample is $1/p = |S|/|R|$.

Interview Question 2.1.6. The techniques used in this section are related to the problem of sampling from a list while avoiding certain entries. See, e.g., this problem.

Analysis of UniformInt

Now that we have analyzed the general Rejection Sampling method, we can analyze our method for uniformly sampling from a finite set. Recall that UniformInt(n) samples from a larger set with dummy elements, which has total size 2^ℓ where $\ell = \lceil \lg n \rceil$. Thus UniformInt$(n)$ is equivalent to RejectionSample$([\![n]\!], [\![2^\ell]\!])$

Claim 2.1.7. UniformInt(n) is uniformly distributed on $[\![n]\!]$.

Proof. This follows immediately from Claim 2.1.3. $\square$

Claim 2.1.8. UniformInt performs less than 2 iterations in expectation.

Proof. Since we chose $\ell = \lceil \lg n \rceil$, inequality (A.2.6) tells us that $\frac{1}{2} \cdot 2^\ell < n \leq 2^\ell$. Rearranging the first inequality, we get

$$\frac{|S|}{|R|} \;=\; \frac{2^\ell}{n} \;<\; 2.$$

The result follows from Claim 2.1.5. $\square$

2.2 Biased coin from unbiased coin

This chapter has assumed so far that we have a function UnbiasedBit that produces unbiased random bits. In many algorithms we need to generate *biased* random bits, i.e., those for which the probabilities of 0 and 1 are different. These are also called biased coins, or general Bernoulli RVs.

Let's say that we want a bit that is 1 with probability b. This can be done using the imperfect approach based on continuous uniform random variables, analogous to Section 2.1.3. We generate a continuous random variable X that is uniform on the interval $[0,1]$, then output 1 if $X \leq b$. This event has probability b due to (A.3.4).

Algorithm 2.5 Generating a random bit that is 1 with probability b.

function BiasedBit(real b)
  $X \leftarrow$ ContinuousUniform()
  if $X \leq b$ **then return** 1 **else return** 0
end function

As in Section 2.1.3, there is the flaw that a practical implementation would use a floating-point or fixed-point number with a limited amount of precision. A consequence is that the probability is not completely accurate: if X and $1/3$ are stored with limited precision, then $\Pr[\, X \leq 1/3 \,]$ will not be exactly $1/3$.

Another drawback is wastefulness. We could generate an unbiased coin by BiasedBit$(1/2)$, but this would presumably take many random bits to generate the continuous uniform RV, whereas an unbiased coin flip should only need a single random bit!

A nice idea is to use a *just-in-time* strategy to generate the random bits of X, while comparing them to the binary representation of b. Let us illustrate with an example, taking $b = 1/3$. The binary representation of b is

$$b \;=\; \langle 0.b_1 b_2 b_3 \ldots \rangle \;=\; \langle 0.01010101\ldots \rangle$$

since $1/3 = 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots$. Similarly, let us represent X in binary as $\langle 0.X_1 X_2 X_3 \ldots \rangle$.

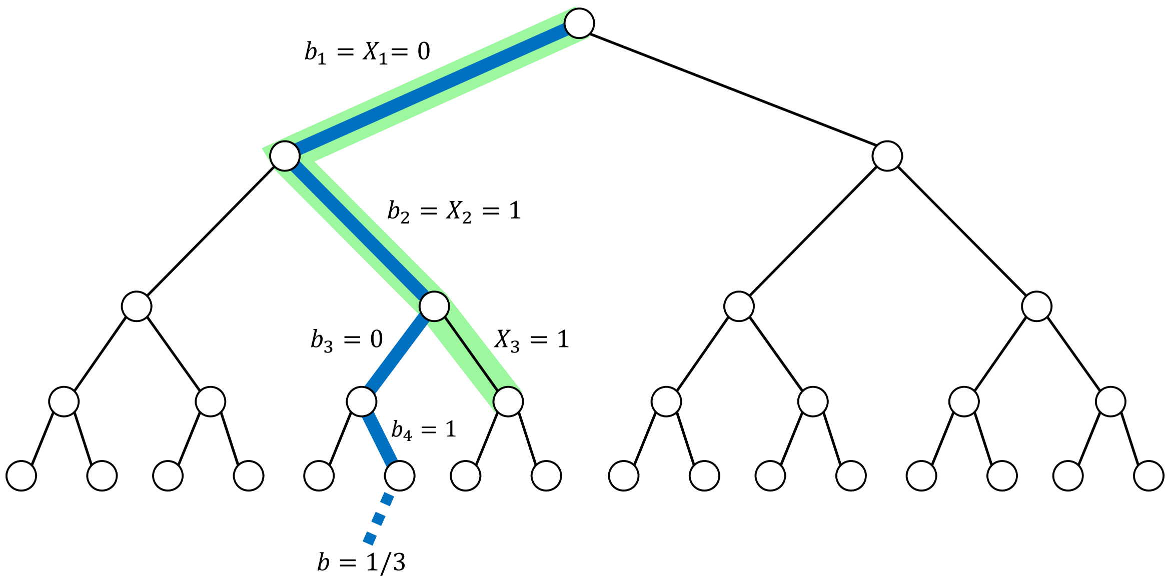
Suppose we randomly generate the first bit, X_1. If $X_1 = 1$ then we already know that $X \geq 1/2$, which definitely implies that $X > b$ so we can immediately return 0. On the other hand if $X_1 = 0$ then we should keep generating more bits of X until it becomes clear whether $X < b$ or $X > b$.

This strategy is implemented in Algorithm 2.6 and illustrated in Figure 2.2.

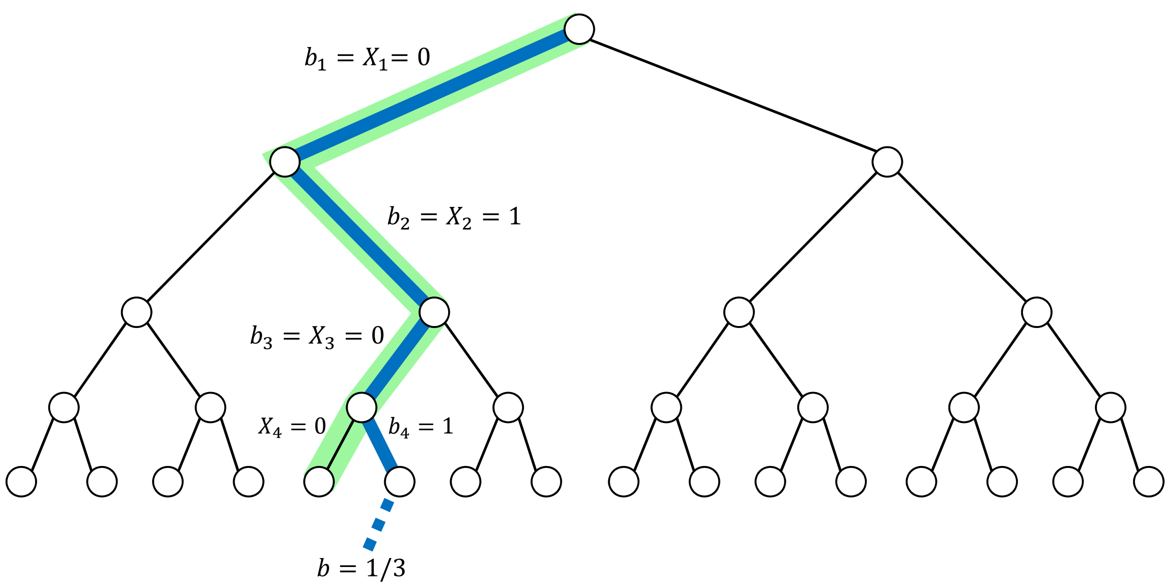
Algorithm 2.6 Generating a random bit that is 1 with probability b.

function BiasedBit(real b)
  Let $\langle 0.b_1 b_2 b_3 \ldots \rangle$ be the binary representation of b.
  for $i = 1, 2, \ldots$ **do** ▷ The first $i-1$ bits of X match b
    Let $X_i \leftarrow$ UnbiasedBit()
    if $X_i > b_i$ **then return** 0 ▷ We now know that $X > b$
    if $X_i < b_i$ **then return** 1 ▷ We now know that $X < b$
  end for
end function

Question 2.2.1. In expectation, how many iterations of the for loop are required?



(a)



(b)

Figure 2.2: In Algorithm 2.6, it is useful to think of the bits of b as describing a path (shown in blue) in a binary tree towards the value of b. A 0 bit means to follow the left child, and a 1 bit means to follow the right child. The algorithm generates bits of X which also describe a path (shown in green) in this binary tree. The algorithm continues so long as X's path matches b's path. (a) If X's path branches to the right of b's path then X lies in the interval $(b, 1]$, so the output is 0. (b) If X's path branches to the left of b's path then X lies in the interval $[0, b)$, so the output is 1.

Answer.

The function terminates at the first iteration i for which X_i and b_i disagree (either $X_i = 1 \wedge b_i = 0$, or $X_i = 0 \wedge b_i = 1$). For each i, this holds with probability exactly $1/2$. So the number of iterations required is a geometric random variable with expectation 2.

References: (Cormen et al., 2001, Exercise C.2-6).

2.3 General distributions

2.3.1 Finite distributions

Above we have shown how to sample from a uniform distribution on any finite set. Suppose instead that we want to sample from an arbitrary distribution with *non-uniform* probabilities on a finite set. It suffices to consider sampling from $[k]$, because any finite set has a one-to-one correspondence with such a set. This is called a categorical distribution. More concretely, let $p_1, \ldots, p_k$ be probabilities satisfying $\sum_{i=1}^k p_i = 1$, and suppose we want to generate a random variable X satisfying

$$\Pr[X = i] = p_i.$$

The following approach is very natural. First, partition the unit interval $[0,1]$ into k segments, where the i^{th} segment has length p_i. As illustrated in Figure 2.3, the right endpoint of the i^{th} segment has the value $q_i = \sum_{1 \le j \le i} p_j$.

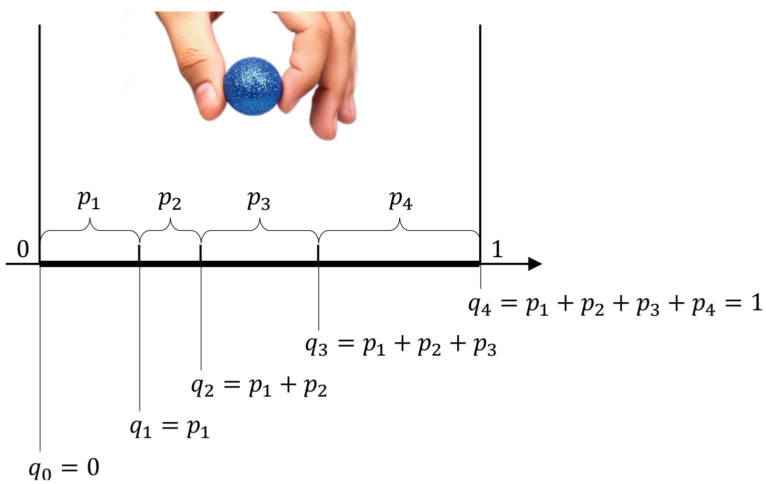


Figure 2.3: The interval $[0,1]$ is divided into segments of lengths $p_1, \ldots, p_k$. The boundaries between the segments are the q_i values.

We can imagine dropping a ball randomly so that its location is uniform on the interval $[0,1]$. If the ball lands on segment i then we output the value i. It follows from (A.3.4) that

$$\Pr[\text{output is } i] \;=\; q_i - q_{i-1} \;=\; p_i.$$

Algorithm 2.7 gives pseudocode for this approach. If we generate many samples from the same categorical distribution we would not want to repeatedly compute the q_i values, so this pseudocode is written as a data structure.

Algorithm 2.7 A data structure for sampling from a categorical distribution.

class CategoricalSampler:

  array $q[0..k]$

  ▷ The constructor. p is the array of probabilities.

  Constructor (reals $p[1..k]$)

    Set $q_0 \leftarrow 0$

    for $i = 1, \ldots, k$

      Set $q_i \leftarrow \sum_{1 \leq j \leq i} p_j$ ▷ The right endpoint of the i^{th} segment.

  ▷ Generate a sample from the distribution

  GenerateSample ()

    Let $Y \leftarrow$ ContinuousUniform()

    Find i such that Y lies in the interval $[q_{i-1}, q_i)$

    return i

Question 2.3.1. For an efficient implementation of this data structure, how much time is needed for the constructor? How much time is needed to generate each sample?

Answer.

The preprocessing can be implemented in $O(k)$ time since $q_i = q_{i-1} + p_i$. Each sample can be generated in $O(\log k)$ time using binary search.

You might have noticed that the values q_i above are exactly the values of the ***cumulative distribution function***, or ***CDF***. That is, if $\Pr[X = i] = p_i$, then the CDF of X is the function $F : \mathbb{R} \to [0, 1]$ where

$$\begin{aligned} F(x) &= \Pr[X \leq x] & & & \text{for all } x \in \mathbb{R}, \text{ and} \\ F(i) &= \Pr[X \leq i] &= \sum_{1 \leq j \leq i} p_j &= q_i & \text{for } i \in \{0, 1, \ldots, k\}. \end{aligned}$$

We can reinterpret this algorithm using the CDF. Essentially, all we have to do is rotate our point of view by 90 degrees counterclockwise. Instead of dropping a ball from above, imagine now throwing a ball from the left at a height that is uniform on the interval $[0, 1]$. This is shown in Figure 2.4 (b). Let i be the x-coordinate of the point at which the ball hits the CDF. The algorithm then outputs the value i. As before,

$$\Pr[\text{output is } i] = F(i) - F(i-1) = p_i.$$

Algorithm 2.8 Generating a categorical random variable using the CDF.

function CategoricalSampler(CDF F)

  Let $Y \leftarrow$ ContinuousUniform()

  return i such that $Y \in [F(i-1), F(i))$

end function

References: (Anderson et al., 2017, Example 5.29), (Grimmett and Stirzaker, 2001, Theorem 4.11.1(b)).

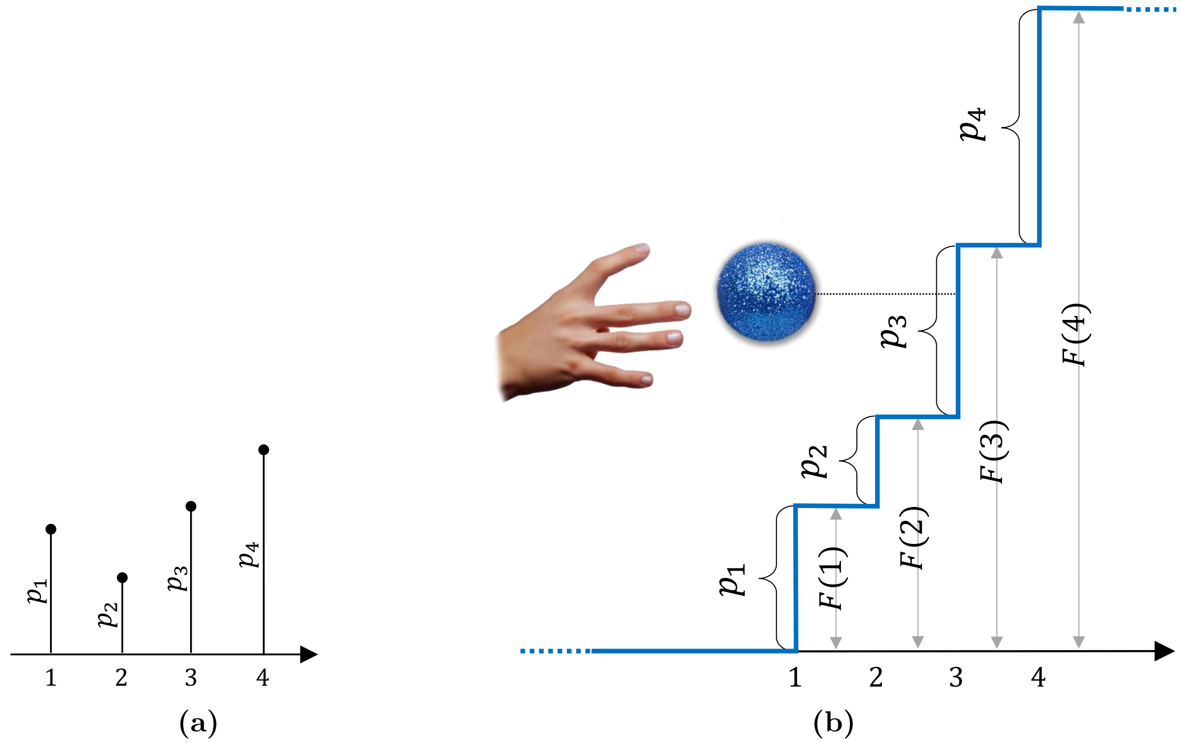


Figure 2.4: (a) The probability mass function of a distribution. (b) The cumulative distribution function of that same distribution. The x-coordinate at which the ball hits the CDF is $x = 3$.

★2.3.2 General continuous random variables

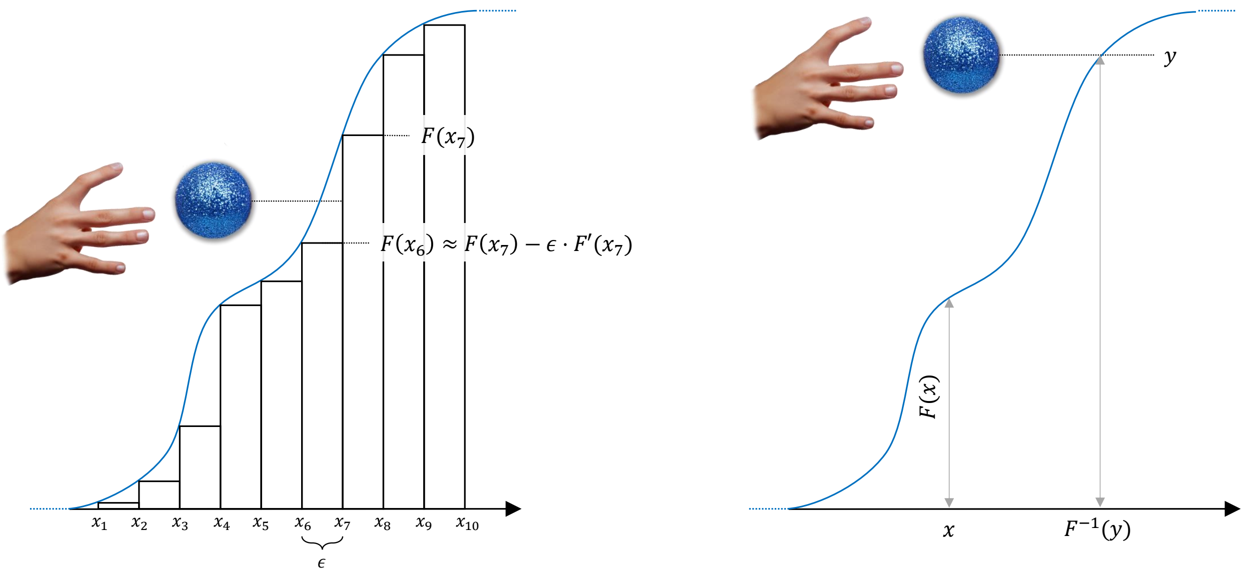
Algorithm 2.8 shows how to sample from any *finite* distribution by using its CDF F. A slight modification of this approach works for all random variables.

To start, first consider the case of a continuous random variable. This means that F is differentiable and its derivative F' is the ***probability density function***. Let $\epsilon > 0$ be a small value. Imagine discretizing the domain into multiples of ϵ, say $x_i = \epsilon \cdot i$ for all integers i. This is illustrated in Figure 2.5 (a). The appropriate modification of line 3 is

$$\textbf{return } x_i \text{ where } Y \in [F(x_{i-1}), F(x_i)).$$

The probability of outputting x_i is $F(x_i) - F(x_{i-1})$ which, by Taylor's theorem, is approximately $\epsilon F'(x_i)$. Thus x_i is output with probability (approximately) proportional to $F'(x_i)$, which is the density at x_i.

The preceding discussion is just for intuition, because it does not really generate a continuous random variable. Rather, the output will only be the discrete values that are the multiples of ϵ. However, as the discretization parameter ϵ decreases to 0, we have $x_{i-1} \to x_i$, so the condition $Y \in [F(x_{i-1}), F(x_i))$ simply becomes $Y = F(x)$. We can invert this relationship to get the equivalent condition $x = F^{-1}(Y)$, under some minor assumptions. This idea is illustrated in the following pseudocode.



(a) **(b)**

Figure 2.5: (a) Sampling a continuous random variable by discretization. The x-coordinate at which the ball hits the discretized CDF is $x = x_7$, so the output is x_7. (b) Sampling a continuous random variable without discretization.

Algorithm 2.9 Generating a continuous RV using the CDF.

---

function SAMPLECONTINUOUS(CDF F)
    Let $Y \leftarrow$ CONTINUOUSUNIFORM()
    return $F^{-1}(Y)$
end function

Theorem 2.3.2. Consider a distribution whose CDF F is continuous and strictly increasing. Then SAMPLECONTINUOUS outputs a RV with that distribution.

Our preceding discussion has given a heuristic justification of this theorem. bext we give a proof that is concise and correct, but perhaps less intuitive.

Proof. The output of the algorithm is $F^{-1}(Y)$. Its distribution is completely determined by its CDF, which is the function mapping x to $\Pr\left[F^{-1}(Y) \leq x\right]$. We apply the function F to both sides of the inequality in that event. Since F is strictly increasing, this preserves the inequality, and we get

$$\Pr\left[F^{-1}(Y) \leq x\right] \;=\; \Pr\left[Y \leq F(x)\right] \;=\; \Pr\left[Y \in [0, F(x)]\right] \;=\; F(x).$$

Here we have used that Y is a RV that is uniform on $[0,1]$, and equation (A.3.4). This shows that F is the CDF of $F^{-1}(Y)$, which is the output of the algorithm. $\square$

References: (Anderson et al., 2017, Example 5.30), (Grimmett and Stirzaker, 2001, Exercise 2.3.3 and Theorem 4.11.1(a)), Wikipedia.

Problem to Ponder 2.3.3. How could you modify SAMPLECONTINUOUS to handle functions F that are not continuous or not strictly increasing?

2.4 Fair coin from a biased coin

Suppose you have a coin that you can flip to make random decisions. The coin is called *fair* (or unbiased) if it has equal probability of heads and tails, otherwise it is called *biased*. Suppose we want to run a randomized algorithm that requires an fair coin, but all we have at our disposal is a biased coin. Is there some way to obtain a fair coin from our biased coin?

The first reaction might be to physically modify our biased coin to try to make it more fair. But, in the spirit of algorithm design, perhaps we can devise a new method rather than engineering a physical solution.

To make the goal concrete, suppose our goal is to write a random generator FAIR () given access to a random generator BIASED (). These functions both output a random bit, and they satisfy

$$
\begin{aligned}
&\Pr[\,\textsc{Fair}() = \textsf{True}\,] = 1/2 \qquad &&\Pr[\,\textsc{Fair}() = \textsf{False}\,] = 1/2 \\
&\Pr[\,\textsc{Biased}() = \textsf{True}\,] = p \qquad &&\Pr[\,\textsc{Biased}() = \textsf{False}\,] = 1 - p,
\end{aligned}
$$

where $p \neq 1/2$.

A single call to Biased () First let's imagine an algorithm that calls BIASED just once. As illustrated in Figure 2.6, the call to BIASED splits the execution into two possible execution paths. In one path, which has probability p of occurring, BIASED returned True. In the other path, which has probability $1 - p$ of occurring, BIASED returned False. In each of those paths, the algorithm must produce exactly one output (because it has no inputs, and no additional source of randomness).

If the former path outputs True and the latter outputs False (or vice versa), then this algorithm's output is clearly biased. Even worse, if both paths return the *same* output then either True or False has probability 0, so the output is very biased!

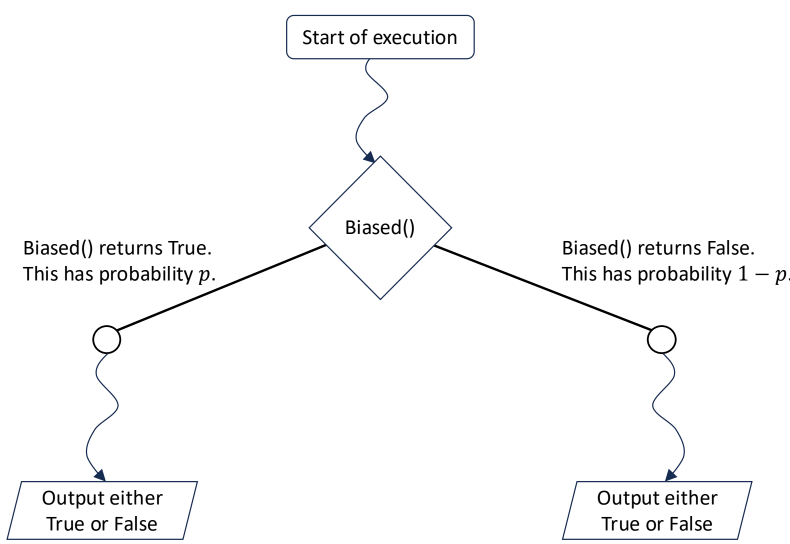


Figure 2.6: A diagram illustrating the execution of an algorithm that makes one call to BIASED (). Since there are only two different execution paths, the output cannot be unbiased.

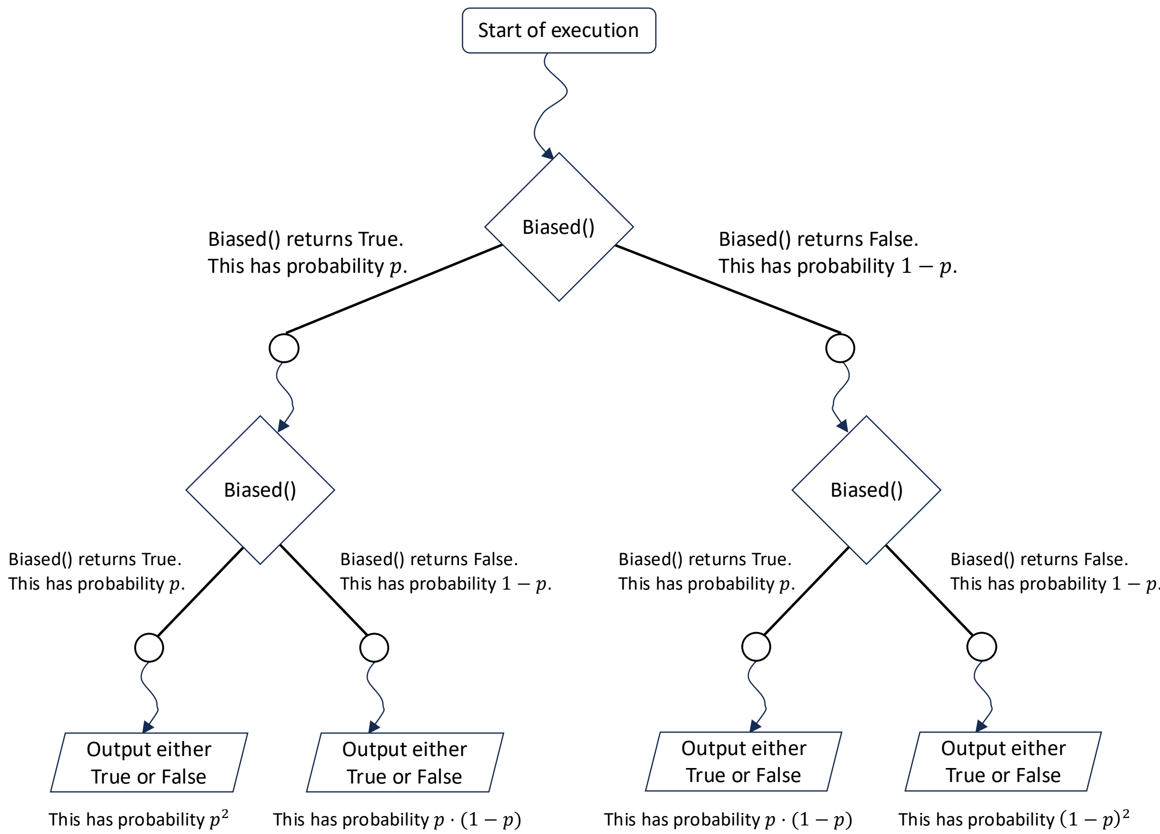


Figure 2.7: A diagram illustrating the execution of an algorithm that makes exactly two calls to Biased ().

Two calls to Biased () Next let's imagine an algorithm that calls Biased exactly twice. The execution of such an algorithm is illustrated in Figure 2.7. There are four possible paths that produce an output, and their respective probabilities are shown in the figure and in the following table.

First call to Biased \ Second call to Biased	True	False
True	p^2	$p(1-p)$
False	$p(1-p)$	$(1-p)^2$

Figure 2.8: Probabilities of different outcomes.

To make this more concrete, imagine that the bias is $p = 0.75$. Then the probabilities of the outcomes are as follows.

First call to Biased \ Second call to Biased	True	False
True	0.5625	0.1875
False	0.1875	0.0625

One may verify that no subset of these numbers adds up to 0.5. This means that there is no way to choose outputs for these execution paths so that the algorithm's output is unbiased. So it seems that we cannot design the FAIR () algorithm with unbiased output using only two calls to BIASED ().

Inspiration from rejection sampling. However, in the table above, we can notice something interesting: there are two entries with the same value 0.1875. (These correspond to the execution paths in which one call to BIASED () returned True and the other returned False.) Perhaps we could arrange it so that one of these paths returns True and the other returns False. So far, True and False have the same probability. Then what should we do with the other execution paths?

A clever idea is to handle the other execution paths using the *rejection sampling* approach from Section 2.1.3: if both calls to BIASED () return True or both return False, then simply run the algorithm again. Pseudocode for this is shown in Algorithm 2.10, and the idea is illustrated in Figure 2.9.

Algorithm 2.10 An implementation of FAIR based on rejection sampling.

```
function FAIR
    while True do
        c1 ← BIASED()
        c2 ← BIASED()
        if c1 = True and c2 = False then return True
        if c1 = False and c2 = True then return False
    end while
end function
```

Claim 2.4.1. This algorithm outputs True and False with equal probability.

Proof sketch. Consider any iteration of the loop. Let $U = c_1c_2$ be the string containing the outcomes of the two biased coins. For example, if c_1 and c_2 are both True, we write $U = \mathsf{TT}$.

The algorithm returns True if $U = \mathsf{TF}$ and False if $U = \mathsf{FT}$. Since those two events have the same probability, conditioned on the loop terminating, then the output is unbiased. $\square$

We can make this proof slightly more formal, along the lines of Claim 2.1.3.

Proof. Recall that the loop terminates when $U \in \{\mathsf{TF}, \mathsf{FT}\}$. Then

$$
\begin{aligned}
\Pr[\,U = \mathsf{TF} \mid U \in \{\mathsf{TF},\mathsf{FT}\}\,] &= \frac{\Pr[\,U = \mathsf{TF} \wedge U \in \{\mathsf{TF},\mathsf{FT}\}\,]}{\Pr[\,U \in \{\mathsf{TF},\mathsf{FT}\}\,]} && \text{(definition of conditional probability)}\\
&= \frac{\Pr[\,U = \mathsf{TF}\,]}{\Pr[\,U \in \{\mathsf{TF},\mathsf{FT}\}\,]} && \text{(since } U = \mathsf{TF} \Rightarrow U \in \{\mathsf{TF},\mathsf{FT}\})\\
&= \frac{\Pr[\,U = \mathsf{TF}\,]}{\Pr[\,U = \mathsf{TF}\,] + \Pr[\,U = \mathsf{FT}\,]} && \text{(disjoint events)}\\
&= \frac{1}{2} && \text{(since } \Pr[\,U = \mathsf{TF}\,] = \Pr[\,U = \mathsf{FT}\,]).
\end{aligned}
$$

Similarly, $\Pr[\,U = \mathsf{FT} \mid U \in \{\mathsf{TF},\mathsf{FT}\}\,] = 1/2$. $\square$

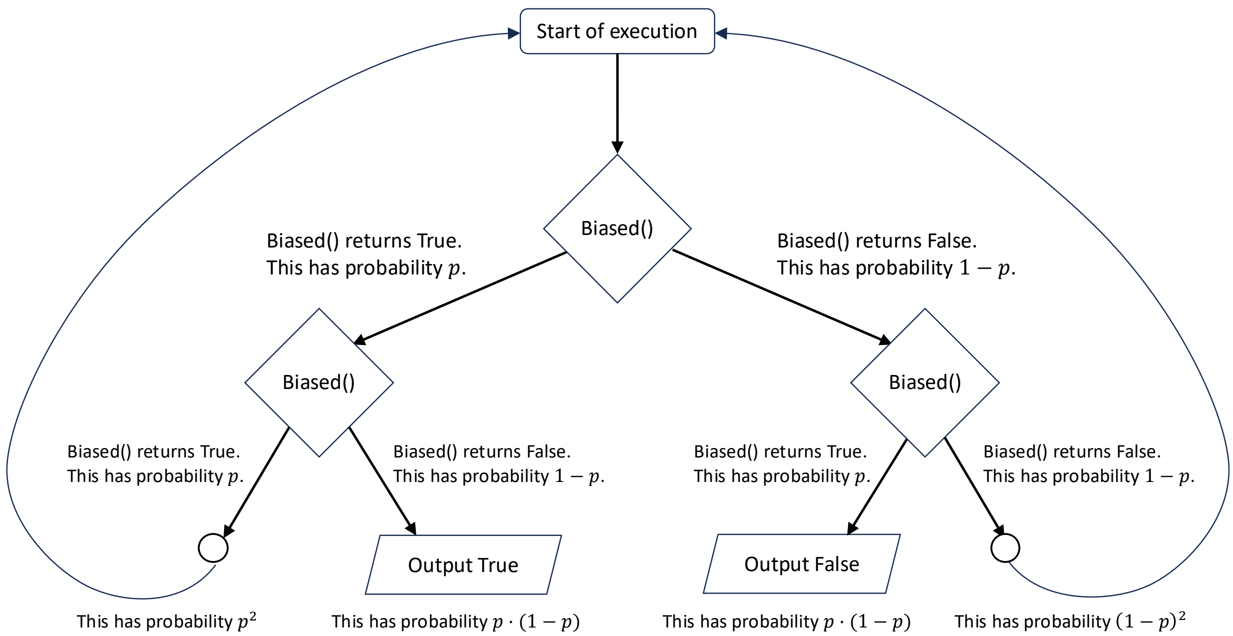


Figure 2.9: A diagram illustrating the execution of an algorithm that makes exactly two calls to Biased ().

Runtime. Like several of the algorithms we have seen so far, the runtime of $\textsc{Fair}()$ is also random. As before, we can analyze it using a geometric random variable. We may view each iteration as a random trial where "success" means that $U \in \{\mathsf{TF}, \mathsf{FT}\}$ (i.e., the coin flips differ), so that the algorithm terminates. For each trial, we have

$$\Pr[\text{trial succeeds}] \;=\; \Pr[U = \mathsf{TF}] + \Pr[U = \mathsf{FT}] \;=\; 2p(1-p).$$

The number of iterations until the first trial succeeds is a geometric random variable, which we will denote by X. Once again using Fact A.3.21, we have

$$\mathrm{E}[X] \;=\; \frac{1}{\Pr[\text{trial succeeds}]} \;=\; \frac{1}{2p(1-p)}.$$

Notice that if the biased coin has $p \approx 0$ or $p \approx 1$ then the expected number of iterations is very large. For example, $p = 0.01$ then $\mathrm{E}[X] > 50$. This makes sense: if tossing a random coin that is extremely biased, then it should take a lot of trials to extract enough randomness to generate an unbiased random bit.

References: (Motwani and Raghavan, 1995, Exercise 1.1), (Cormen et al., 2001, Exercise 5.1-3), (Feller, 1968, Exercise IX.9.10).

History. John von Neumann presented this scheme in four sentences in a 1951 paper.

> *If independence of successive tosses is assumed, we can reconstruct a 50-50 chance out of even a badly biased coin by tossing twice. If we get heads-heads or tails-tails, we reject the tosses and try again. If we get heads-tails (or tails-heads), we accept the result as heads (or tails). The resulting process is rigorously unbiased, although the amended process is at most 25 percent as efficient as ordinary coin-tossing.*

Interview Question 2.4.2. This is apparently a common interview question. See, e.g., here, or here.

2.5 Exercises

Exercise 2.1 Generating uniform random numbers. Suppose you have access to a random number generator Rng() that generates uniform random numbers in $[\![n]\!]$. Note that Rng takes no arguments, so you cannot change the value of n, although you do know the value of n. You would like to generate a uniform random number in $[\![m]\!]$, where $m \leq n$.

Part I. Consider the most obvious approach shown below. Prove that, *for every* $n > 2$, there exists $m \leq n$ such that this function *doesn't* generate uniform random numbers in $[\![m]\!]$.

```
function BadSampler(integer m)
    return Rng() mod m
end function
```

Part II. Design a function GoodSampler(integer m) that uses Rng() as its source of randomness. It knows the value of n, although it cannot change this value. The function must satisfy two conditions.

1. Its output is uniform in $[\![m]\!]$.

2. Each call to GoodSampler(m) makes $O(1)$ calls to Rng() in expectation.

You should prove these two properties.

Exercise 2.2. Consider the following pseudocode.

```
function Mystery(int n)
    p ← ContinuousUniform()
    for i = 1, ..., n do
        X_i ← BiasedBit(p)
    end for
    X ← Σ_{i=1}^{n} X_i
    return X
end function
```

Implement this algorithm in your favourite programming language. What do you think is the distribution of X?

Exercise 2.3 Fast sampling from finite distributions. Let $p_1, \ldots, p_k$ be probabilities satisfying $\sum_{i=1}^{k} p_i = 1$. As in Section 2.3.1, we want to preprocess these values in polynomial time, such that we can generate samples from a random variable X satisfying $\Pr[\, X = i \,] = p_i$.

Whereas Algorithm 2.7 used ContinuousUniform as its source of randomness, we now consider how to do it using the function BiasedBit(b) as the source of randomness. Assume for simplicity that BiasedBit runs in $O(1)$ time.

Part I. Show how to generate a sample in $O(k)$ time, given only the probabilities $p_1, \ldots, p_k$. Argue that your algorithm is correct.

Part II. Show how to use $O(k)$ preprocessing time so that samples can be generated in $O(\log k)$ time. Argue that your algorithm is correct.

Exercise 2.4 Faster sampling from finite distributions. This exercise considers the same problem as the previous exercise, but develops an even faster algorithm. For simplicity, assume that k is a power of two.

The algorithm should use $O(k)$ time for preprocessing and $O(1)$ expected time to generate each sample. It will need to use BiasedBit(), and will also assume that UniformLBitInteger($\lg k$) requires only $O(1)$ time.

Part I. Imagine breaking up $[0, 1]$ into k intervals, where the i^{th} interval is labeled i and has length p_i. Find a way to divide each intervals into pieces, and group those pieces into pairs, such that each pair has combined length exactly $1/k$. Hence, the total number of pieces is exactly $2k$. Each piece carries the same label as the interval that it came from.

Describe pseudocode that performs this task in $O(k)$ time.

Part II. Design a function that takes those pairs of pieces as input, then creates a random output X satisfying $\Pr[X = i] = p_i$. The function should have expected runtime $O(1)$.

Exercise 2.5. A fundamental continuous distribution is the ***standard normal distribution***.

References: (Anderson et al., 2017, Definition 3.60), (Grimmett and Stirzaker, 2001, Definition 4.4.4), Wikipedia.

Its CDF can be written

$$F(x) \;=\; \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right),$$

where erf is a special function called the error function.

Implement and run code in Julia to generate a standard normal random variable using Algorithm 2.9 and Julia's library of special functions. How fast is this approach compared to Julia's `randn` function?

Exercise 2.6 Random primes. Suppose you are given an integer x and you want to find the smallest prime number that exceeds x. There are inefficient algorithms for this problem, such as exhaustive search. Unfortunately there are no known efficient algorithms, by which we mean one with running time $O\big((\log n)^k\big)$ for some constant k.

Instead, let us consider an easier problem. Suppose you wish to pick a prime uniformly at random from the set

$$P_n \;=\; \{\, x \,:\, x \text{ is prime and } x \leq 20n \,\}\,.$$

Give an algorithm to do this with expected running time $O\big((\log n)^{10}\big)$, for $n \geq 2$. You may use the fact that there is a deterministic algorithm with runtime $O\big((\log n)^7\big)$ to test if the number n is prime.

Exercise 2.7 Continuous Rejection Sampling. Let's consider uniformly sampling from some continuous sets. The following is a continuous analog of Claim 2.1.3.

Claim. Let S be some set of finite volume and let R be a subset of S. (We assume R to be non-empty and of finite volume.) Suppose that X is uniformly distributed on S. Then, conditioned on lying in R, X is uniformly distributed on R.

Part I. Let $S = [-1,1]^2$ be the square in the plane of side length 2 centered at the origin. Let R be the disk (i.e., solid circle) of radius 1 centered at the origin. (Note that $R \subseteq S$.)

Suppose we use sample from S and use rejection sampling to obtain a uniform sample on R. How many iterations does this require in expectation, and why?

Part II. Let $S = [-1,1]^{20}$ be the 20-dimensional cube of side length 2 centered at the origin. Let

$$R \;=\; \left\{ x \in \mathbb{R}^{20} \,:\, \sqrt{\sum_{i=1}^{20} x_i^2} \leq 1 \right\}$$

be the 20-dimensional ball of radius 1 centered at the origin. It is true that $R \subseteq S$.

Suppose we use sample from S and use rejection sampling to obtain a uniform sample on R. How many iterations does this require in expectation, and why? Does this seem like a good algorithm for sampling from R?

Exercise 2.8. Let K be a positive number. Define the function

$$F(x) \;=\; \begin{cases} 0 & (x < -K) \\ \frac{1}{2} + \frac{\arctan(x)}{2\arctan(K)} & (|x| \leq K) \\ 1 & (x > K) \end{cases}$$

Since $\arctan(x)$ is continuous and strictly increasing, so is $F(x)$. Moreover, we have $\lim_{x\to-\infty} f(x) = 0$ and $\lim_{x\to\infty} f(x) = 1$. These properties imply that $F(x)$ can be the cumulative distribution function of a random variable.

Part I. What is the density function of this distribution?

Part II. Give an explicit algorithm to sample a random variable having this distribution.

Chapter 3

Sampling objects

3.1 Random permutations

Let C be a list of n distinct items. Suppose we want to put them in a uniformly random order. What does that mean? How can we do that? You may recall from an introductory discrete math course that there are $n!$ orderings (or ***permutations***) of C. So a ***uniformly random ordering*** means to choose each of those orderings with probability $1/n!$.

We will generate a uniformly random ordering by considering a more general problem. A k***-permutation*** of C is an ordered sequence of k distinct items from C. So a total ordering of C is the same as an n-permutation. The number of k-permutations is

$$n \cdot (n-1) \cdots (n-k+1) \;=\; \frac{n!}{(n-k)!}. \tag{3.1.1}$$

When $k = n$, this agrees with our remark that there are $n!$ orderings of C, since $0! = 1$.

References: (Anderson et al., 2017, (1.8)), (Cormen et al., 2001, (C.1)), (Feller, 1968, Section II.2), Wikipedia.

Algorithm 3.1 presents an approach to generate a uniformly random k-permutation. Although it is perhaps too simple to deserve a name, it is usually called the "Fisher-Yates shuffle".

References: (Grimmett and Stirzaker, 2001, Exercise 4.11.2), Wikipedia.

Algorithm 3.1 Generate a uniformly random k-permutation of C. Let $n = |C|$.

function GenKPerm(list C, int k)
    Let L be an empty list
    for $i = 1, \ldots, k$
        Let c be a uniformly random element of C $\triangleright$ *Length*$(C) = n - i + 1$
        Remove c from C
        Append c to L
    return L
end function

To analyze this algorithm, we will need to use conditional probabilities. Informally, the notation $\Pr[\, A \mid B \,]$ means the probability that A happens, assuming that we already know that B has happened. The following is a key fact about conditional probabilities.

Fact A.3.5 (Chain rule). Let $A_1, \ldots, A_t$ be arbitrary events. Then

$$\Pr[\, A_1 \wedge \cdots \wedge A_t \,] \;=\; \prod_{i=1}^{t} \Pr[\, A_i \;\mid\; A_1 \wedge \cdots \wedge A_{i-1} \,],$$

if we assume that $\Pr[\, A_1 \wedge \cdots A_{t-1} \,] > 0$.

Claim 3.1.1. GenKPerm generates a uniformly random k-permutation.

Proof. Let L be the output of the algorithm, and consider any fixed k-permutation B. Let $\mathcal{E}_i$ be the event that $L[i] = B[i]$. As long as $B[i]$ has not been picked in iterations $1, ..., i-1$, it has probability $1/(n-i+1)$ of being picked in iteration i. Thus

$$\Pr[\, \mathcal{E}_i \;\mid\; \mathcal{E}_1 \wedge \cdots \wedge \mathcal{E}_{i-1} \,] \;=\; \frac{1}{n-i+1}.$$

Thus, by the chain rule,

$$\begin{aligned} \Pr[\, L = B \,] &= \Pr[\, \mathcal{E}_1 \wedge \cdots \wedge \mathcal{E}_k \,] \\ &= \prod_{i=1}^{k} \Pr[\, \mathcal{E}_i \;\mid\; \mathcal{E}_1 \wedge \cdots \wedge \mathcal{E}_{i-1} \,] \;=\; \prod_{i=1}^{k} \frac{1}{n-i+1} \;=\; \frac{(n-k)!}{n!}, \end{aligned}$$

the reciprocal of (3.1.1). It follows that each k-permutation B is picked with uniform probability. □

Random permutations (or k-permutations) have several interesting properties.

Claim 3.1.2. Let L be the output of GenKPerm(C, k).

- **(a)** For any $i \leq k$, the entry $L[i]$ is a uniformly random element of C.
- **(b)** For any $i \in C$, $\Pr[\, i \in L \,] = k/n$.

References: (Feller, 1968, pp. 31).

Proof sketch. It is immediate from the pseudocode of GenKPerm that $L[1]$ is uniformly distributed in C. One may see by symmetry that, for any x, the number of k-permutations with $L[1] = x$ is the same as the number of them with $L[i] = x$. This implies that $L[i]$ is also uniformly distributed. Then, by Fact A.3.7,

$$\Pr[\, i \in L \,] \;=\; \sum_{j=1}^{k} \Pr[\, L[j] = i \,] \;=\; \sum_{j=1}^{k} \frac{1}{n} \;=\; \frac{k}{n}. \qquad \square$$

Question 3.1.3. If each entry $L[i]$ is a uniformly random element of C, does that mean that the entries of L might not be distinct?

Answer.

No. Each of them is uniformly distributed, but they are not independent.

Problem to Ponder 3.1.4. Prove Claim 3.1.2 part (a) using Exercise A.7.

3.1.1 Efficiency

A subtle question is: what is the efficiency of Algorithm 3.1? For simplicity, let us focus on the case $k = n$. The efficiency depends heavily on the data structure for representing C. There are two natural options.

- If C is implemented as an array, then line 4 can be implemented in $O(1)$ time, but line 5 would require $\Omega(n - i)$ time.

- If C is implemented as a linked list, then line 5 can be implemented in $O(1)$ time, but line 4 would require $\Omega(n - i)$ time.

With either option, the runtime would be proportional to $\sum_{i=1}^{n}(n-i) = \Theta(n^2)$, which is not as fast as one might hope.

Problem to Ponder 3.1.5. Using a balanced binary tree to represent C, show that Algorithm 3.1 can be implemented in $O(n \log n)$ time.

A very efficient implementation of the GenKPerm algorithm is to simply swap entries of C as they are selected. This idea was published by Durstenfeld in 1964.

Algorithm 3.2 A linear-time, in-place algorithm to generate a uniformly random k-permutation of C.

```
1: function GenKPerm(array C, int k)
2:     for i = 1, ..., k
3:         Let r be a uniformly random integer in {i, ..., n}
4:         Swap C[i] and C[r]
5:     return C[1..k]
6: end function
```

Theorem 3.1.6. GenKPerm produces a uniformly random k-permutation. Assuming that each random number can be generated in $O(1)$ time, the overall runtime is $O(k)$.

Note that, by taking $k = n$, GenKPerm produces a full permutation in linear time.

References: (Cormen et al., 2001, Lemma 5.5), (Knuth, 2014, Algorithm 3.4.2.P).

Proof. The runtime analysis is straightforward: lines 3 and 4 each require constant time, so the total runtime is $O(k)$.

At the start of the i^{th} iteration, the *set* of elements in $C[i..n]$ (in which we ignore their ordering) is the set of elements that have yet to be chosen. Since the algorithm picks a uniformly random element from $C[i..n]$, their ordering is irrelevant. So the i^{th} iteration chooses $C[i]$ to be a uniformly random element that has yet to be chosen. This is exactly the same behavior as the original algorithm in Algorithm 3.1. So, by Claim 3.1.1, the implementation in Algorithm 3.2 also generates a uniformly random k-permutation. □

⋆3.1.2 An approach based on sorting

Let us briefly consider an alternative approach to generating a uniformly random permutation.

Algorithm 3.3 An algorithm based on sorting to generate a uniformly random permutation of C.

```
function GenPermBySorting(array C[1..n])
    Create an array X[1..n] containing independent, uniform numbers in [0, 1]
    Sort C[1..n] using X[1..n] as the sorting keys
    return C
end function
```

The expected runtime of this algorithm is $O(n \log n)$ if we use QuickSort; see Section 4.4. Interestingly, the expected runtime can be improved to $O(n)$ if we use a specialized approach for sorting random numbers; see Exercise 7.10.

Claim 3.1.7. GenPermBySorting returns a uniformly random ordering of C.

Proof. Since $X_1, \ldots, X_n$ are independent and identically distributed,

$$X_1, \ldots, X_n \text{ has the same distribution as } X_{\pi(1)}, \ldots, X_{\pi(n)}$$

for every permutation π. (The technical term is that $X_1, \ldots, X_n$ are ***exchangeable***. See (Anderson et al., 2017, Definition 7.15 and Theorem 7.20).) So, for all permutations π, σ,

$$\Pr\left[\, X_{\pi(1)} < \cdots < X_{\pi(n)} \,\right] \;=\; \Pr\left[\, X_{\sigma(1)} < \cdots < X_{\sigma(n)} \,\right].$$

That is, every ordering is equally likely to be the unique sorted ordering of X.

There is one small detail: if X has any identical entries, then it does not have a unique sorted ordering. However, since we are using random real numbers in $[0, 1]$, there is zero probability that X has any identical entries; see Exercise 7.6. □

3.2 Random subsets

A very common task in data analysis and randomized algorithms is to pick a random subset of the data. There are multiple ways to do so, and it is useful to be aware of their differences.

To make matters concrete, suppose the set of items is $C = [m]$. We would like to pick a sample S of about k items randomly from C.

Sampling with replacement. With this approach, our sample is a list[1] $S = [s_1, \ldots, s_k]$ of items. Each s_i is chosen independently and uniformly from C. There is no requirement that the samples are distinct, so S might contain fewer than k distinct items. As an extreme example, it is possible that *all* of the samples equal c_1.

References: (Anderson et al., 2017, page 6).

Sampling without replacement. With this approach, our sample is a set $S = \{s_1, \ldots, s_k\}$ consisting of k distinct items, assuming $k \leq m$. The set S is chosen uniformly at random amongst all subsets of C that have size k. This is also called a ***simple random sample***.

References: (Anderson et al., 2017, page 8), Wikipedia.

[1] Or, if we forget the ordering of S, it is a multiset.

Algorithm 3.4 Three approaches to generating random subsets of C. Assume that $|C| = m$.

```
function SampleWithReplacement(list C, int k)
    Let S be an empty multiset
    for i = 1, ..., k
        Let c be a uniformly random element of C
        Add c to S
    return S
end function

function SampleWithoutReplacement(list C, int k)
    L ← GenKPerm(C, k)                                  ▷ See Algorithm 3.2.
    return L, but converted into a set to forget its order
end function

function BernoulliSample(list C, real k)
    Let S be an empty set
    foreach i ∈ C
        Flip a coin that is heads with probability p = k/m
        if the coin is heads then add i to S
    return S
end function
```

Bernoulli sampling. With this approach, we consider each item in turn, and flip a coin to decide whether the item should be added to the sample S. The bias of the coin is chosen so that each item is independently added with probability[2] k/m, assuming $k \leq m$.

References: Wikipedia.

Let's consider the size of S under these different approaches.

- *Sampling with replacement.* The size of S is exactly k, if we count duplicates. (For the number of distinct items, see Exercise 7.9.)
- *Sampling without replacement.* S contains exactly k items, all of which are distinct.
- *Bernoulli sampling.* The size of S is a random variable having a ***binomial distribution*** $B(m, k/m)$; see Section A.3.3. The size of S has expectation $m \cdot (k/m) = k$, by (A.3.5). Again, all items are distinct.

Question 3.2.1. What is the probability that $S = \{1, \ldots, k\}$ when using sampling without replacement?

Answer.

There are $\binom{m}{k}$ sets of size k, so each subset is sampled with probability $1/\binom{m}{k}$.

Problem to Ponder 3.2.2. What is the probability that $S = \{1, \ldots, k\}$ when using sampling with replacement? Compare this to the probability when using sampling without replacement.

[2] The more general scenario in which items are added with different probabilities is called Poisson sampling. This name may be somewhat confusing because it has nothing to do with the Poisson distribution. This more general scenario will not be needed in this book.

It is often much easier to deal with multiple events or random variables when they are independent. It is worthwhile to consider whether, in some sense, these three sampling methods produce independent samples. The answer requires phrasing the question carefully.

- *Sampling with replacement.* Let $S = [s_1, \ldots, s_k]$ be the sample, viewed as a list. The samples s_i and s_j are independent for $i \neq j$. They are often called ***i.i.d. samples***, which stands for "independent and identically distributed".
- *Bernoulli sampling.* Let S be the sample, which is a set. We cannot speak of the i^{th} and j^{th} element in the sample — for example, it is possible that $S = \emptyset$. However, for distinct $c, c' \in C$, the events "$c \in S$" and "$c' \in S$" are independent.
- *Sampling without replacement.* This sampling scheme has neither of those independence properties.

We summarize these properties in the following table.

	Runtime	Size of set	Independence
SampleWithReplacement	$O(k)$	k, including duplicates	yes
SampleWithoutReplacement	$O(k)$	k	no
BernoulliSample	$O(n)$	k in expectation	yes

Although BernoulliSample has the rather slow runtime of $O(n)$, it can be improved to $O(k)$ in expectation. See Exercise 3.7.

3.3 Random partitions

Let C be a finite set. A ***partition*** of C into at most ℓ parts is a family $A_1, \ldots, A_\ell$ of subsets of C that contain every element of C exactly once. In mathematical notation,

$$A_1 \cup \cdots \cup A_\ell = C \qquad \text{and} \qquad A_i \cap A_j = \emptyset \quad \forall \text{distinct } i, j \in [\ell].$$

References: (Lehman et al., 2018, Definition 10.5.6), Wikipedia.

There are many scenarios in which it is useful to *randomly* partition a set. One example is for cross-validation, an approach for training and validating machine learning models.

How might one generate a random partition? Two of our approaches from Section 3.2 for generating random *subsets* can be modified to generate random partitions. Pseudocode is shown in Algorithm 3.5. We will let $m = |C|$ and assume that $k = m/\ell$ is an integer.

Analysis of BernoulliPartition[3]. The second algorithm is the simpler of the two. It is easy to see that each element of C is equally likely to end up in any part of the partition. In mathematical symbols,

$$\Pr[\, c \in A_i \,] \;=\; \frac{1}{\ell} \;=\; \frac{k}{m} \qquad \forall c \in C,\ i \in [\ell].$$

This holds simply because the RV X in line 10 is uniform.

The BernoulliPartition algorithm has an intriguing property. Although it generates several sets A_i, each individual A_i has the same distribution as if we had just generated a single subset using Bernoulli sampling.

[3] The name of this algorithm comes from its connection to Bernoulli sampling; see Claim 3.3.1. However the RV generated in line 10 is not a Bernoulli RV.

Algorithm 3.5 Two approaches to randomly partition C into ℓ subsets. Assume that $|C| = m$ is a multiple of ℓ. Let $k = m/\ell$.

function PartitionWithoutReplacement(list C, int ℓ)
  for $i = 1, \ldots, \ell$ **do**
    $A_i \leftarrow$ GenKPerm($C \setminus (A_1 \cup \cdots \cup A_{i-1})$, k) ▷ See Algorithm 3.2.
  end for
  return $A_1, \ldots, A_\ell$
end function

function BernoulliPartition(list C, int ℓ)
  Let $A_1, \ldots, A_\ell$ be empty sets
  foreach $c \in C$
    Let X be a RV that is uniform on $[\ell]$
    Add c to A_X
  return $A_1, \ldots, A_\ell$
end function

Claim 3.3.1. A_i has the same distribution as BernoulliSample(C, k), for each $i \in [\ell]$.

Proof. As argued above, each item c appears in A_i with probability $1/\ell = k/m$; furthermore, these events are independent. The same properties hold for the output of BernoulliSample. These properties completely describe the distribution. □

Next let us consider the runtime. We will assume that generating X takes constant time. If each set A_i is implemented as a linked list, then adding elements to A_i also takes constant time. Summing over all iterations, the total runtime is $O(\ell + m) = O(m)$.

Analysis of PartitionWithoutReplacement. The PartitionWithoutReplacement algorithm has the same intriguing property as the BernoulliPartition algorithm. Although it generates several sets A_i, each individual A_i has the same distribution as if we had just generated a single subset using sampling without replacement. This seems quite mysterious, because A_i is defined to be a sample without replacement from $C \setminus (A_1 \cup \cdots \cup A_{i-1})$. Nevertheless, A_i has the same distribution as a sample without replacement from C.

Claim 3.3.2. A_i has the same distribution as SampleWithoutReplacement(C, k), for every $i \in [\ell]$.

We will prove this below, after reinterpreting the algorithm. Recall that GenKPerm(C, k) can be viewed as returning the first k elements in a full permutation of C. So, instead of calling GenKPerm multiple times, we could instead generate a full permutation of C and use consecutive groups of k elements as the partition $A_1, \ldots, A_\ell$. That is,

$$\begin{aligned}
A_1 &= L[1..k] \\
A_2 &= L[1+k..2k] \\
A_3 &= L[1+2k..3k] \\
&\vdots
\end{aligned}$$

The corresponding pseudocode is as follows.

Algorithm 3.6 A revised approach to partitioning without replacement.

```
function PartitionWithoutReplacement(list C, int k)
    L ← GenKPerm(C, m)                                          ▷ See Algorithm 3.2.
    for i = 1, ..., ℓ do
        A_i is the i^th group of k consecutive elements in L, namely L[1 + (i−1)·k..i·k]
    end for
    return A_1, ..., A_ℓ
end function
```

This modified pseudocode is equivalent to the original pseudocode, but easier to analyze. First let us consider the runtime. Since $\textsc{GenKPerm}(C, m)$ takes $O(m)$ time (Theorem 3.1.6), the overall runtime of $\textsc{PartitionWithoutReplacement}$ is $O(m + \ell \cdot k) = O(m)$ time.

Proof of Claim 3.3.2. For any distinct values $x_1, \cdots, x_k \in C$, there are exactly $(m-k)!$ permutations with

$$L[1] = x_1 \quad \wedge \quad L[2] = x_2 \quad \wedge \quad \cdots \quad \wedge \quad L[k] = x_k,$$

because one can simply fill the remaining entries of L with a permutation of the remaining $m-k$ elements of C. This is the same number of permutations with

$$L[1 + (i-1) \cdot k] = x_1 \quad \wedge \quad L[2 + (i-1) \cdot k] = x_2 \quad \wedge \quad \cdots \quad \wedge \quad L[i \cdot k] = x_k,$$

for the same reason. Thus, the probability that the entries $L[1..k]$ take any particular values is the same as for the entries $L[1 + (i-1) \cdot k..i \cdot k]$. It follows that A_1 has the same distribution as A_i. Since A_1 is the output of $\textsc{SampleWithoutReplacement}(C, k)$, this completes the proof. □

Problem to Ponder 3.3.3. Prove Claim 3.3.2 using Exercise A.7.

3.4 Reservoir sampling

The previous sections considered methods to sample from a distribution that was known in advance. Now let us imagine that we would like to uniformly sample an item from a sequence whose entries are only revealed one-by-one. The length of the sequence is not known in advance, but at every point in time we would like to have a uniform sample from the items seen so far.

It is not hard to think of examples where this could be useful. For example, perhaps a network router wants to uniformly sample one packet from all packets that it sees each day. Of course, it would take too much space store all the packets and then pick one at the end of the day.

To describe this problem more formally, suppose that at each time step i an item s_i is received. Suppose for simplicity that all items are distinct. The algorithm must select, at every time i, an item X_i satisfying that

$$\Pr[\, X_i = s_j \,] \;=\; \frac{1}{i} \qquad \forall j \in [i]. \tag{3.4.1}$$

However, the algorithm cannot store items $s_1, \ldots, s_i$. Instead, when s_i arrives at time i, the algorithm must perform one of the following actions.

- *Ignore* s_i, and set $X_i \leftarrow X_{i-1}$.
- *Take* s_i, and set $X_i \leftarrow s_i$.

The algorithm must randomly choose which action to take in order to ensure that (3.4.1) holds.

Algorithm 3.7 The reservoir sampling algorithm. Here X_i is the item that is held by the algorithm.

```
function ReservoirSampling
    for i = 1, 2, ... do
        Receive item s_i
        Flip a biased coin that is heads with probability 1/i
        if coin was heads then
            Set X_i ← s_i
        else
            Set X_i ← X_{i-1}
        end if
    end for
end function
```

3.4.1 Example

To understand how this can be done, let us consider an example.

Time 1. There is no previous element, so the only choice is to *take* s_1. We set $X_1 \leftarrow s_1$

Time 2. The condition (3.4.1) is simply that $\Pr[X_2 = s_1] = \Pr[X_2 = s_2] = 1/2$. Since we know that $X_1 = s_1$, we simply flip a fair coin to decide whether to *ignore* or *take* s_2. That is, we set $X_2 \leftarrow X_1$ with probability $1/2$, and $X_2 \leftarrow s_2$ with probability $1/2$.

Time 3. Now things are more intricate. We want $\Pr[X_3 = s_i] = 1/3$ for all i. In particular we need $\Pr[X_3 = s_3] = 1/3$, so that suggests that the algorithm should *take* s_3 with probability $1/3$ (regardless of the value of X_2). Thus, we will *ignore* s_3 with probability $2/3$. Does this ensure a uniform distribution on the three items?

Let's consider the probability of having the item s_1 at time 3.

$$\begin{aligned}\Pr[X_3 = s_1] &= \Pr[X_2 = s_1 \wedge \text{algorithm ignores } s_3] \\ &= \Pr[X_2 = s_1] \cdot \Pr[\text{algorithm ignores } s_3] \\ &= \frac{1}{2} \cdot \frac{2}{3} = \frac{1}{3}.\end{aligned}$$

Here we are using that the probability of ignoring s_3 is independent of the value of X_2. An identical calculation shows that $\Pr[X_3 = s_2] = 1/3$.

3.4.2 The algorithm

The above ideas are formalized as pseudocode in Algorithm 3.7.

Theorem 3.4.1. At every time i, X_i is a uniformly random sample from the (distinct) items $s_1, \ldots, s_i$.

Proof. We will show by induction that $\Pr[X_i = s_j] = 1/i$ for all $j \leq i$.

Base case: It is clear that $\Pr[X_1 = 1] = 1$ because the first coin is heads with probability 1.

Inductive step: The algorithm guarantees $\Pr[X_i = s_i] = 1/i$ because the i^{th} coin flip is heads with probability $1/i$. Furthermore, for $j < i$,

$$\begin{aligned}
\Pr[X_i = s_j] &= \Pr\Big[X_{i-1} = s_j \text{ and } i^{\text{th}} \text{ coin is tails}\Big] \\
&= \Pr[X_{i-1} = s_j] \cdot \Pr\Big[i^{\text{th}} \text{ coin is tails}\Big] \qquad \text{(independence)} \\
&= \frac{1}{i-1} \cdot \frac{i-1}{i} \qquad \text{(by induction)} \\
&= \frac{1}{i}.
\end{aligned}$$

□

Interview Question 3.4.2. This is apparently a common interview question. See, e.g., here, here, or here.

Question 3.4.3. Above we assumed that the items were distinct. Suppose instead that the items are *not* distinct, and we want that the probability of holding each item to be proportional to the number of times it has appeared. Can you modify the algorithm to accomplish this?

Answer.

Actually, no modifications are required! The analysis shows that, at each time i, the last time an item was taken is uniformly distributed on $\{1, \ldots, i\}$. Thus, if an item s has appeared k times so far, then $\Pr[X_i = s] = k/i$.

3.5 Exercises

Exercise 3.1. Suppose we want to generate a subset of C that is uniformly random among all 2^m subsets of C. How can we do this using the functions in Algorithm 3.4?

Exercise 3.2. Let $n > a > b$. Define

$$\begin{aligned} A &= \textsc{SampleWithoutReplacement}([n], a) \\ B &= \textsc{SampleWithoutReplacement}(A, b). \end{aligned}$$

Show that B has the same distribution as $\textsc{SampleWithoutReplacement}([n], b)$.

Exercise 3.3. Consider the following algorithm for generating a uniformly random k-permutation.

Algorithm 3.8 Generate a uniformly random k-permutation of C. Let $C = |n|$.

```
1: function GenKPermLazy(array C, int k)
2:     Let L be an empty list
3:     for i = 1, ..., k do
4:         repeat
5:             Let r be a uniformly random value in [n]
6:         until C[r] ≠ Null
7:         Append C[r] to L
8:         C[r] ← Null
9:     end for
10:    return L
11: end function
```

Part I. Explain why the value $C[r]$ in line 7 is a uniformly random chosen value from the remaining items in C.

Part II. Assume that the random number generation takes $O(1)$ time. Show that $\textsc{GenKPermLazy}$ can be modified to run in $O(k)$ expected time by occasionally removing all the null elements from C.

Exercise 3.4. The implementation of $\textsc{GenKPerm}$ from Algorithm 3.2 runs in $O(k)$ time, but has one flaw: it requires modifying C. This may be undesirable in some scenarios. Let us consider some ways to remove that flaw while keeping the same runtime. You may not use a hash table because, thus far, we have no guarantees on its performance.

Part I. First suppose that $k = \Omega(n)$. Find an algorithm to generate a k-permutation without modifying C. The algorithm should have expected runtime $O(k)$.

Part II. Now assume that $k = o(n)$. Suppose that you are given free access to an array of n bits that you can modify and is initialized to zero. Find an algorithm that uses this array to generate a k-permutation without modifying C. The algorithm should have expected runtime $O(k)$.

Part III. Again assume that $k = o(n)$. Suppose that you are given free access to an array of n integers that you can modify but is *initialized to arbitrary values* (possibly adversarially chosen). Find an algorithm that uses this array and $O(k)$ additional space to generate a k-permutation without modifying C. The algorithm should have expected runtime $O(k)$.

Exercise 3.5. Suppose we generate a sample S using SampleWithoutReplacement(C, k). Later, we change our mind and decide that we really wanted a sample *with* replacement. How can we modify S to make a new sample S' whose distribution is the same as calling SampleWithReplacement(C, k).

Exercise 3.6. In this exercise, we consider how to implement SampleWithoutReplacement(C, k) using BernoulliSample. Assume that $C = [m]$.

Part I. Let S be the random set output by BernoulliSample(C, k). For any fixed set $A \subseteq C$, show that $\Pr[S = A]$ depends only on the *size* of A and not its contents.

Part II. Design a new algorithm that implements SampleWithoutReplacement(C, k), but can only call BernoulliSample as its source of randomness. The algorithm should call BernoulliSample at most $O(m)$ times in expectation, and it should pass in the same set C as the first parameter.

Hint: see Appendix A.3.3.

*** **Part III.** Give an algorithm that calls BernoulliSample at most $O(\sqrt{k})$ times in expectation. You should pass in the same set C as the first parameter.

Exercise 3.7. In Section 3.2 we saw that the runtime of BernoulliSample is $O(n)$. Let us consider a more efficient approach. There exists[4] a randomized algorithm $\mathcal{A}$ that outputs a *random number* that has the binomial distribution $B(n, p)$, and has expected runtime $O(np)$. Using $\mathcal{A}$ (and possibly other functions), design new pseudocode for an implementation of BernoulliSample(C, k) that has expected runtime $O(k)$. Prove that your pseudocode works.

[4]See, e.g., Devroye "Non-Uniform Random Variate Generation", Section X.4.3.

Chapter 4

Expectation

Several aspects of a randomized algorithm require analysis. Typically we must analyze the success probability, as we did in Question 1.2.1. For some algorithms the runtime is also random, so that is another aspect that must be analyzed.

A random variable might sometimes be large and sometimes be small, so we often summarize its magnitude by looking at its expectation. For example, Chapter 2 considered several algorithms and analyzed their expected runtime.

In this chapter we discuss expectation in more depth, and mention some of its nice properties. We will introduce ***linearity of expectation***, especially via ***decomposition into indicator random variables***.

4.1 Runtime for generating a k-permutation

Durstenfeld's implementation of GenKPerm, shown in Algorithm 3.2, produces a uniformly random k-permutation. It requires no extra space, and runs in $O(k)$ time. Or does it?

Theorem 3.1.6 is slightly misleading. It assumes that $O(1)$ time suffices to generate a uniform random integer in $\{i, \ldots, n\}$. That assumption is not entirely justified. As we have seen in Section 2.1.3, the function UniformInt can do this with $O(1)$ *expected* runtime. Since UniformInt is called multiple times, the overall runtime involves the sum of many random variables. Some care is required to properly analyze this sum.

Consider the i^{th} iteration of GenKPerm, at the step in which UniformInt is called. The number of iterations required inside this call to UniformInt is a random variable, which we denote X_i. The overall runtime of GenKPerm is therefore proportional to $\sum_{i=1}^{k} X_i$. So the *expected* runtime of GenKPerm is proportional to

$$\mathrm{E}\left[\sum_{i=1}^{k} X_i\right].$$

How can we analyze this expectation? The following fact is called ***linearity of expectation***, and it is extremely useful.

Fact A.3.11. Let $X_1, \ldots, X_n$ be random variables and $w_1, \ldots, w_n$ arbitrary real numbers. Then

$$\mathrm{E}\left[\sum_{i=1}^{n} w_i X_i \right] = \sum_{i=1}^{n} w_i \, \mathrm{E}[\, X_i \,].$$

Applying this to our scenario, we take each $w_i = 1$ and obtain that

$$\begin{aligned} \mathrm{E}\left[\sum_{i=1}^{k} X_i \right] &= \sum_{i=1}^{k} \mathrm{E}[\, X_i \,] && \text{(by Fact A.3.11)} \\ &< \sum_{i=1}^{k} 2 && \text{(by Claim 2.1.8)} \\ &= O(k). \end{aligned}$$

We conclude that the *expected* runtime of GenKPerm is $O(k)$.

Since each call to UniformInt uses fresh random bits, the random variables $X_1, \ldots, X_k$ are independent. However, the preceding analysis does *not* use that fact. Linearity of expectation applies to *all* random variables, regardless of whether they are independent.

4.2 Fixed points of a permutation

Next let us consider a silly problem. The CS department has run out of money for TAs, so it was decided that students will have to grade each other's homework. To try to make things fair, the instructor must gather all homework into a big pile, randomly shuffle the pile, then hand the homework back out to the students one-by-one. That is, we use a *uniformly random permutation* to swap the students' homeworks.

Note that we haven't avoided the possibility that some student grades their own homework. The instructor hopes that this is quite unlikely. If there are n students, how many students do we expect will grade their own homework?

Complicated approach. Let X be the random variable giving the number of students who grade their own homework. By definition, $\mathrm{E}[\, X \,] = \sum_{i=0}^{n} i \cdot \Pr[\, X = i \,]$. To use this formula, it seems we would have to analyze $\Pr[\, X = i \,]$, which is not so easy. In particular, $\Pr[\, X = 0 \,]$ is the fraction of derangements, which takes some work to compute.

References: (Anderson et al., 2017, Example 1.27).

Easy approach. Decomposition into indicator random variables gives a much simpler approach. Let $\mathcal{E}_i$ be the event that student i gets their own homework. Referring to Definition A.3.13, the corresponding ***indicator random variable*** is

$$\mathbf{1}_{\mathcal{E}_i} = \begin{cases} 1 & \text{(if } i \text{ gets their own homework)} \\ 0 & \text{(otherwise).} \end{cases}$$

The decomposition of X into indicator RVs is

$$X = \sum_{i=1}^{n} \mathbf{1}_{\mathcal{E}_i}.$$

We can analyze the expectation of X using linearity of expectation. Since X is a sum of indicator random variables, we can use a formula that is tailored to this case.

Fact A.3.14. Suppose that X is a random variable that can be decomposed as $X = \sum_{i=1}^{n} \mathbf{1}_{\mathcal{E}_i}$, where X_i is the indicator of an event $\mathcal{E}_i$. Then

$$\mathrm{E}\left[\, X \,\right] \;=\; \sum_{i=1}^{n} \mathrm{E}\left[\, \mathbf{1}_{\mathcal{E}_i} \,\right] \;=\; \sum_{i=1}^{n} \Pr\left[\, \mathcal{E}_i \,\right].$$

It remains to determine $\Pr\left[\, \mathcal{E}_i \,\right]$, which fortunately is easy. The uniformly random permutation can be generated using GenKPerm; since it is a full permutation, we take $k = n$. Let us call that permutation L. Then we have

$$\Pr\left[\, \mathcal{E}_i \,\right] \;=\; \Pr\left[\, \text{student } i \text{ gets their own homework} \,\right] \;=\; \Pr\left[\, L[i] = i \,\right] \;=\; \frac{1}{n},$$

since Claim 3.1.2 implies that $L[i]$ is a uniformly random number in $[n]$. Plugging in to Fact A.3.14, we obtain

$$\mathrm{E}\left[\, X \,\right] \;=\; \sum_{i=1}^{n} \frac{1}{n} \;=\; 1.$$

So the instructor can be relieved that, in expectation, exactly one student grades their own homework.

References: (Lehman et al., 2018, Section 19.5.2).

4.3 Polling

The Zoodle corporation is considering producing an avocado slicer. To help decide if this is a good idea, it would like to estimate the fraction of customers who like avocados. Let's say that there are m customers and that f is the true fraction who like avocados. A natural idea is to sample a subset of customers, then use that to estimate f. We will explore this idea using the three sampling approaches presented in Section 3.2.

In more detail, let $C = [m]$ be the set of customers and let $\mathcal{A} \subseteq C$ be subset who like avocados. The fraction who like avocados is $f = |\mathcal{A}|/m$. For all three sampling approaches we will construct a sample S, which is a subset (or multiset) of elements from C of size roughly k. Using S we will construct an estimate $\hat{f}$ of f, then show that it is an ***unbiased estimator***, meaning that $\mathrm{E}\left[\, \hat{f} \,\right] = f$.

Pseudocode for the three estimators is shown in Algorithm 4.1. In all three cases the estimator is

$$\hat{f} \;=\; \frac{\text{number of sampled customers who like avocados}}{k} \;=\; \frac{|S \cap \mathcal{A}|}{k}.$$

However, we must be aware of a subtlety with the notation $|S \cap \mathcal{A}|$. When sampling with replacement is used S is a multiset. We assume that the notation $|S \cap \mathcal{A}|$ will count duplicate samples multiple times.

4.3.1 Sampling with replacement

Our first approach is PollWithReplacement, which is based on SampleWithReplacement. Its pros and cons are as follows.

Algorithm 4.1 Three approaches to using polling to estimate the fraction of customers who like avocados. These are based on the three approaches for generating subsets in Algorithm 3.4.

function PollWithReplacement(list C, int k, set $\mathcal{A}$)
  $S \leftarrow$ SampleWithReplacement(C, k)
  $\hat{f} \leftarrow \frac{1}{k} \cdot$ (total number of customers in S who are also in $\mathcal{A}$)
  return $\hat{f}$
end function

function PollWithoutReplacement(list C, int k, set $\mathcal{A}$)
  $S \leftarrow$ SampleWithoutReplacement(C, k)
  $\hat{f} \leftarrow \frac{1}{k} \cdot$ (total number of customers in S who are also in $\mathcal{A}$)
  return $\hat{f}$
end function

function BernoulliPolling(list C, int k, set $\mathcal{A}$)
  $S \leftarrow$ BernoulliSample(C, k)
  $\hat{f} \leftarrow \frac{1}{k} \cdot$ (total number of customers in S who are also in $\mathcal{A}$)
  return $\hat{f}$
end function

- *Pros:* This approach produces exactly k samples. Moreover, the samples are ***independent, identically distributed***, or ***i.i.d.*** Furthermore, its analysis is quite simple.
- *Cons:* This approach produces a multiset rather than a set. Consequently, it is possible that customers could be polled more than once, which could be undesirable.

Let $\mathcal{E}_i$ be the event that the i^{th} sampled person likes avocados. The estimator can be written as

$$\hat{f} \;=\; \frac{1}{k}\sum_{i=1}^{k} \mathbf{1}_{\mathcal{E}_i}. \tag{4.3.1}$$

This estimator is often called the ***sample mean***, or ***empirical average***.

Claim 4.3.1. $\hat{f}$ is unbiased.

Proof. Since the i^{th} sampled person is chosen uniformly at random, the probability that they like avocados is exactly f, so $\Pr[\mathcal{E}_i] = f$. Thus, by linearity of expectation (Fact A.3.14),

$$\mathrm{E}\left[\hat{f}\right] \;=\; \frac{1}{k}\sum_{i=1}^{k}\mathrm{E}[\mathbf{1}_{\mathcal{E}_i}] \;=\; \frac{1}{k}\sum_{i=1}^{k}\Pr[\mathcal{E}_i] \;=\; \frac{1}{k}\sum_{i=1}^{k} f \;=\; f. \qquad \square$$

References: (Anderson et al., 2017, Fact 8.14).

4.3.2 Sampling without replacement

Our second approach is PollWithoutReplacement, which is based on SampleWithoutReplacement. Its pros and cons are as follows.

- *Pros:* This approach produces a subset of size exactly k, with no repeated elements.

- *Cons:* This approach produces non-independent samples, which complicates the analysis.

As above, let $\mathcal{E}_i$ be the event that the i^{th} sampled person likes avocados. The estimator, which is again the sample mean, is

$$\hat{f} \;=\; \frac{1}{k}\sum_{i=1}^{k} \mathbf{1}_{\mathcal{E}_i}. \tag{4.3.2}$$

Claim 4.3.2. $\hat{f}$ is unbiased.

Proof. Recall from Algorithm 3.4 that the sample S is generated using a random k-permutation. By Claim 3.1.2 (a), each sample is a uniformly random customer, and therefore

$$\Pr[\,\mathcal{E}_i\,] \;=\; \frac{|\mathcal{A}|}{|C|} \;=\; \frac{fm}{m} \;=\; f.$$

One may observe that $\mathcal{E}_1, \ldots, \mathcal{E}_m$ are *not* independent, but fortunately that is irrelevant. Using linearity of expectation (Fact A.3.14), we have

$$\mathrm{E}\left[\,\hat{f}\,\right] \;=\; \frac{1}{k}\sum_{i=1}^{k} \mathrm{E}[\,\mathbf{1}_{\mathcal{E}_i}\,] \;=\; \frac{1}{k}\sum_{i=1}^{k} \Pr[\,\mathcal{E}_i\,] \;=\; \frac{1}{k}\cdot k \cdot f \;=\; f. \qquad \square$$

Problem to Ponder 4.3.3. The number of sampled people who like avocados is a well-known distribution that has a name. Find the name of this distribution, and then use its properties to prove Claim 4.3.2.

4.3.3 Bernoulli sampling

Our third approach is BernoulliPolling, which is based on BernoulliSample. Its pros and cons are as follows.

- *Pros:* This approach produces a sample with no repeated elements. The events of different customers appearing in the sample are independent. Consequently, the analysis is quite straightforward.
- *Cons:* The size of the sample is random, and need not be exactly k. Constructing the sample takes $\Theta(m)$ time rather than $O(k)$ time. (However it can be improved to $O(k)$ via Exercise 3.7.)

Recall that the previous two approaches decomposed the estimator $\hat{f}$ into indicators using events $\mathcal{E}_1, \ldots, \mathcal{E}_k$, with one event for each *sample*. Now the decomposition will be a bit different. We will have m events $\mathcal{E}_1, \ldots, \mathcal{E}_m$, with one event for each *customer*. The estimator is now

$$\hat{f} \;=\; \frac{1}{k}\sum_{i\in\mathcal{A}} \mathbf{1}_{\mathcal{E}_i},$$

where $\mathcal{E}_i$ is the event that the i^{th} customer is sampled. Note that this estimator is not exactly the sample mean: it does not divide by the actual number of sampled customers, and instead divides by $k = pm$, which is the *expected* number of sampled customers. Never mind. This is a minor difference, and it helps to simplify matters: for example, we needn't worry about dividing by zero if no customers are sampled.

Claim 4.3.4. $\hat{f}$ is unbiased.

Proof. The Bernoulli sampling process explicitly puts each customer into the sample with probability p, so $\Pr[\mathcal{E}_i] = p$. Using linearity of expectation and $|\mathcal{A}| = fm$, we have

$$\mathrm{E}\left[\hat{f}\right] \;=\; \frac{1}{pm}\sum_{i\in\mathcal{A}} \Pr[\mathcal{E}_i] \;=\; \frac{1}{pm}\sum_{i\in\mathcal{A}} p \;=\; \frac{1}{pm}\cdot fm\cdot p \;=\; f. \qquad \square$$

4.3.4 Summary

We have explained three different approaches to polling, and discussed their respective pros and cons. It is not really possible to say which approach is “the best”. Indeed, any of the approach could be the most appropriate one, depending on the certain circumstances.

Notice that our analyses above did rely on any properties of the integer k. Later, in Section 11.2, we will discuss how to choose the value of k to ensure that the estimate $\hat{f}$ is highly accurate.

4.4 QuickSort

So far in this chapter, the examples we have discussed are fairly straightforward. More substantial problems can also be analyzed by the technique of decomposition into indicator random variables. In this section we apply that technique to randomized QuickSort, one of the most well-known sorting algorithms.

Algorithm 4.2 Pseudocode for Randomized QuickSort. Assume that the elements in A are distinct.

```
1: function QuickSort(set A)
2:     if Length(A) = 0 then return A
3:     Select an element p ∈ A uniformly at random          ▷ The pivot element
4:     Construct the sets Left = { x ∈ A : x < p } and Right = { x ∈ A : x > p }
5:     return the concatenation [ QuickSort(Left), p, QuickSort(Right) ]
6: end function
```

Observe that this algorithm only compares elements in line 4. Moreover, this line compares the pivot p to every other element in A exactly once.

Theorem 4.4.1. *For a set A with n distinct elements, the expected number of comparisons of* QuickSort *is $O(n \log n)$.*

References: (Cormen et al., 2001, Section 7.4.2), (Motwani and Raghavan, 1995, Theorem 1.1), (Mitzenmacher and Upfal, 2005, Section 2.5).

For concreteness, let us write the elements of A in *sorted order* as $a_1 < a_2 < \ldots < a_n$.

Let X be the random variable giving the number of comparisons performed by the algorithm. It is easy to see that every pair of elements is compared at most once, since all comparisons involve the current pivot, which is not passed into either of the recursive children. So we may decompose X into a sum of indicator RVs as follows.

For $i < j$, define $\mathcal{E}_{i,j}$ to be the event that elements a_i and a_j are compared. Then, using Fact A.3.14, we have

$$\begin{aligned} X &= \sum_{i<j} \mathbf{1}_{\mathcal{E}_{i,j}} \\ \text{and} \qquad \mathrm{E}[\,X\,] &= \sum_{i<j} \mathrm{E}[\,\mathbf{1}_{\mathcal{E}_{i,j}}\,] \;=\; \sum_{i<j} \Pr[\,\mathcal{E}_{i,j}\,]. \end{aligned}$$

The heart of the analysis is to figure out these probabilities by subtle reasoning. Fix $i < j$ and let $R = \{a_i, \ldots, a_j\}$. Note that R is contiguous in the *sorted order* of A, not necessarily in A itself.

Claim 4.4.2. Event $\mathcal{E}_{i,j}$ occurs if and only if the first pivot selected from R is either a_i or a_j.

See Figure 4.1 for an illustration of this claim.

Proof. Recall that a_i and a_j are compared iff they are still in the same subproblem at the time that one of them is chosen as the pivot. Looking at the algorithm, we see that a_i and a_j are split into different recursive subproblems at precisely the time that the first pivot is selected from R. If this pivot is either a_i or a_j, then they will be compared; otherwise, they will not. □

Claim 4.4.3.

$$\Pr[\,\mathcal{E}_{i,j}\,] \;=\; \frac{2}{j-i+1}.$$

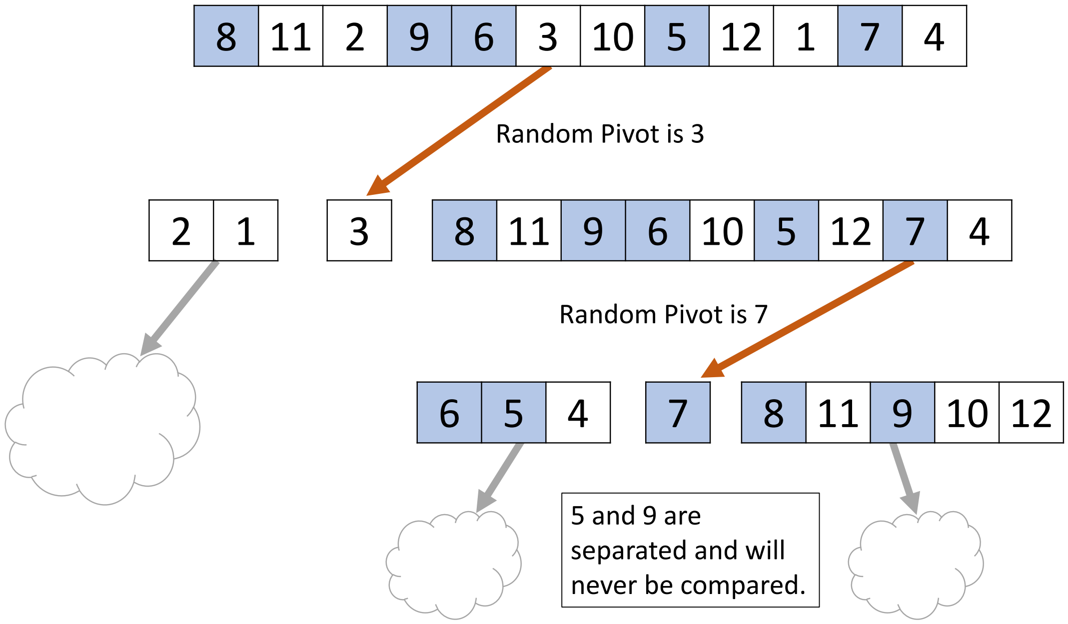
Proof. Due to Claim 4.4.2 we must analyze the probability that a_i or a_j is the first pivot selected from R. Note that all elements of R appear together in the same subproblems until the first pivot is selected from R.

Let A denote that the first subproblem in which a pivot p is chosen from R. Then p is uniformly random on A, but we have conditioned on the event $p \in R$, so p is uniform on R (by Claim 2.1.3). So the probability that a_i or a_j is chosen is $2/|R|$. □

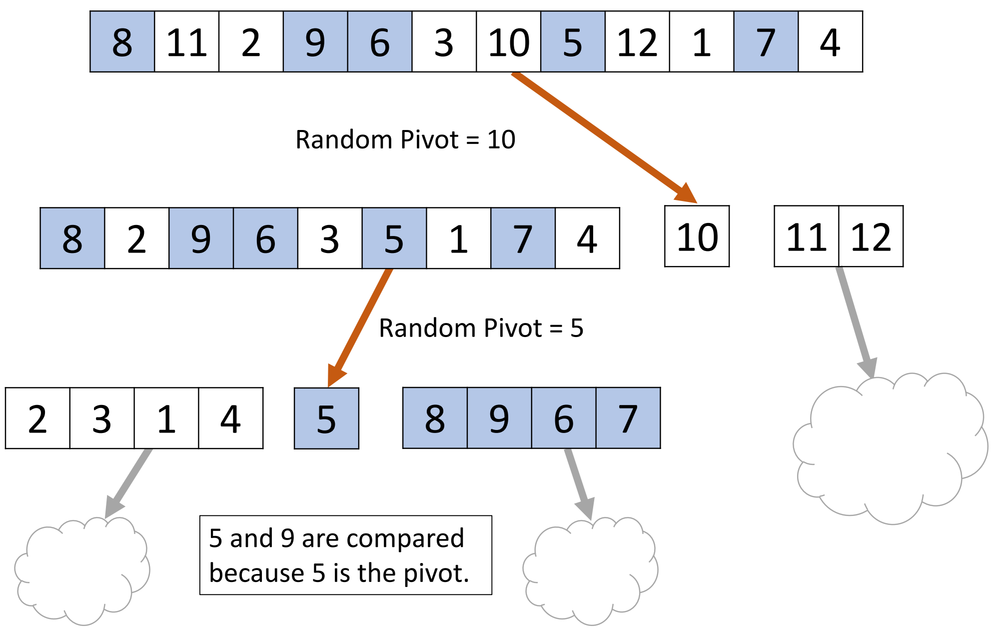
We can combine these ideas to analyze the expected number of comparisons.

$$\begin{aligned} \mathrm{E}[\,X\,] &= \sum_{i<j} \Pr[\,\mathcal{E}_{i,j}\,] && \text{(by Fact A.3.14)} \\ &= \sum_{i=1}^{n-1} \Big(\sum_{j=i+1}^{n} \frac{2}{j-i+1} \Big) && \text{(by Claim 4.4.3)} \\ &= 2 \sum_{i=1}^{n-1} \Big(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1} \Big) \\ &\leq 2 \sum_{i=1}^{n-1} \Big(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \Big) \\ &\leq 2 \sum_{i=1}^{n-1} \ln(n) \\ &< 2n \ln(n) \end{aligned}$$

In the penultimate inequality, we have bounded the harmonic sum using Fact A.2.1.



(a)



(b)

Figure 4.1: An example of randomized QuickSort. Consider the items 5 and 9. The set $R = \{5, 6, 7, 8, 9\}$ is shown in blue. Elements 5 and 9 are compared by the algorithm if the first pivot selected from R is either 5 or 9. (a) The first pivot picked from R is 7, so 5 and 9 are put into separate subproblems and will never be compared. (b) In this example, the first pivot picked from R is 5, so 5 and 9 are compared when 9 is put into the right subproblem.

4.5 Exercises

Exercise 4.1. Let X be a random variable that takes only non-negative integer values. Prove that $\Pr[X \geq 1] \leq \mathrm{E}[X]$.

Exercise 4.2. Let $Z_1, \ldots, Z_k$ be i.i.d. copies of *any* random variable Z. Let $g : \mathbb{R} \to \{0,1\}$ be a Boolean predicate. Define the Bernoulli RVs $X_1, \ldots, X_k$ by $X_i = g(Z_i)$.

For example, this is analogous to Section 4.3.1, where Z_i is a uniformly and independently chosen customer, and g is the predicate that indicates whether that customer is in the set $\mathcal{A}$, and $X_i = \mathbf{1}_{\mathcal{E}_i}$.

Prove that $\hat{f} = \sum_{i=1}^{k} X_i/k$ is an unbiased estimator of $\mathrm{E}[g(Z)]$.

Exercise 4.3. Give an asymptotically tight bound on the expected runtime of GenKPermLazy, which appeared in Algorithm 3.8. Your bound should be a function of k and n, and should not involve a summation.

Exercise 4.4. This exercise consider an alternative analysis for sampling without replacement.

Let $\mathcal{E}_i$ be the event that the i^{th} customer is sampled. The estimator is

$$\hat{f} \;=\; \frac{1}{k} \sum_{i \in \mathcal{A}} \mathbf{1}_{\mathcal{E}_i}.$$

The same definitions of $\mathcal{E}_i$ and $\hat{f}$ were used above to analyze Bernoulli sampling.

Part I. Show that $\Pr[\mathcal{E}_i] = k/m$.

Part II. Show that $\hat{f}$ is an unbiased estimator of f.

Exercise 4.5 Analysis of Randomized QuickSelect. The QuickSelect algorithm is introduced in Section 5.2. In this exercise, we will show how to analyze it by modifying our analysis of QuickSort.

As above, let $a_1 < \cdots < a_n$ be the elements of A in sorted order. For $i < j$, let $R = \{a_i, \ldots, a_j\}$ and let $\mathcal{E}_{i,j}$ be the event that elements a_i and a_j are compared.

The following claim is similar to Claim 4.4.2, with some changes shown in yellow.

Claim 4.5.1. Event $\mathcal{E}_{i,j}$ occurs if and only if some pivot is eventually selected from $\{a_i, \ldots, a_j\}$ and that first pivot is either a_i or a_j.

Part I. Prove Claim 4.5.1.

The following claim is similar to Claim 4.4.3, with some changes shown in yellow.

Claim 4.5.2. Suppose $i < k < j$, so that $a_k \in R$. Then

$$\Pr[\mathcal{E}_{i,j}] \;=\; \frac{2}{j-i+1}.$$

Proof. A pivot must eventually be selected from R since the algorithm terminates when a_k is selected as the pivot, and $a_k \in R$. Note also that the current subproblem must contain all of R until the the first pivot is selected from R.

Let A be the subproblem at the time when the first pivot from R is chosen. Since the pivot p is selected uniformly at random from A, but we have conditioned on the event $p \in R$, so p is uniform on R (by Claim 2.1.3). So the probability that a_i or a_j is chosen is $2/|R| = 2/(j-i+1)$. $\square$

Interestingly, when k is *not* between i and j, the probability of $\mathcal{E}_{i,j}$ changes, as the following claims show.

Claim 4.5.3. Suppose $i < j \leq k$. Then $\Pr[\mathcal{E}_{i,j}] = 2/(k-i+1)$.

Claim 4.5.4. Suppose $k \leq i < j$. Then $\Pr[\mathcal{E}_{i,j}] = 2/(j-k+1)$.

Part II. Prove Claim 4.5.3. (As one would expect, a symmetric argument proves Claim 4.5.4.)

Part III. Use linearity of expectation and the claims above to prove the following theorem.

Theorem 4.5.5. Let X be the number of comparisons performed by QuickSelect. Let $n = |A|$. Then $\mathrm{E}[X] = O(n)$.

Chapter 5

Randomized recursion

In this chapter we will consider some basic randomized algorithms that involve recursion. A prototypical algorithm of this sort is QuickSort, which we have just seen in the previous chapter. However, our analysis of it was not very recursive in nature. In this chapter we will see analyses that are more tailored to randomized recursion.

5.1 Sampling a categorical distribution

As in Section 2.3.1, let $p_1, \ldots, p_k$ be the probabilities of a categorical distribution. (For simplicity, assume that they are all positive.) Consider the problem of sampling a random variable with this distribution. We have seen how to do this with continuous uniform random variables in Section 2.3.1, and using BiasedBit in Exercise 2.3.

In this section we will see how to do it using UnbiasedBit. Of course, we already know how to do this, since one can implement BiasedBit using UnbiasedBit as in Section 2.2. However, in this section we will explore a more direct approach. As usual, define $q_0 = 0$ and $q_i = \sum_{1 \leq j \leq i} p_j$.

Recall from Algorithm 2.6 that BiasedBit(b) generates a uniform continuous random variable $X = \langle 0.X_1X_2X_3\ldots\rangle$ by generating the bits X_i one-by-one. It stops when the bits generated so far are not a prefix of b because then X is comparable to b, i.e., we know that either $X < b$ or $X > b$.

Here we will use a natural generalization of that idea. We will generate the bits of X one-by-one until X is comparable to *every* q_j. The algorithm stops when, for every q_j, we know that either $X < q_j$ or $X > q_j$. Once this happens, X is known to lie in an interval $[q_{a-1}, q_a)$, so the algorithm can output a.

The pseudocode in Algorithm 5.1 implements this idea. We will show that it uses linear time in expectation to generate one sample.

Theorem 5.1.1. The expected runtime of CategoricalSampler is $O(k)$.

To analyze this algorithm, we will focus on the set S_i, which contains the indices of all q_j values that are still not comparable with X at the end of iteration i. Specifically, we are interested in the size $Z_i = |S_i|$. Our key observation is that the expected value of Z_i is exponentially decreasing in i.

Claim 5.1.2. For all $i \geq 0$, we have $\mathrm{E}[\, Z_i \,] = k2^{-i}$.

Algorithm 5.1 Generating a categorical random variable using UNBIASEDBIT. Here $q_j = \sum_{i \leq j} p_j$.

1: **function** CATEGORICALSAMPLER(reals $q[1..k]$)
2:     Let $S_0 \leftarrow \{0, 1, \ldots, k-1\}$
3:     **for** $i = 1, 2, \ldots$ ▷ S_{i-1} contains all j such that $\langle 0.X_1 \ldots X_{i-1} \rangle$ is a prefix of $\langle q_j \rangle$
4:         $S_i \leftarrow \emptyset$
5:         $X_i \leftarrow$ UNBIASEDBIT()
6:         **foreach** $j \in S_{i-1}$
7:             **if** (i^{th} bit of q_j) $= X_i$ **then** insert j to S_i
8:         **if** $S_i = \emptyset$ **then break**   ▷ Now X is comparable to every q_j
9:     **return** the index a such that $X \in [q_{a-1}, q_a)$
10: **end function**

Proof. We use decomposition into indicator random variables to write

$$Z_i \;=\; \sum_{j \in [k]} \mathbf{1}_{\mathcal{E}_{j,i}},$$

where $\mathcal{E}_{j,i}$ is the event that $j \in S_i$. Note that $\mathcal{E}_{j,i}$ occurs if and only if the first i bits of X agree with the first i bits of q_j. Since the bits of X are uniform and independent, $\Pr[\,\mathcal{E}_{j,i}\,] = 2^{-i}$. Thus, by linearity of expectation,

$$\mathrm{E}[\,Z_i\,] \;=\; \sum_{j=1}^{k} \mathrm{E}\left[\,\mathbf{1}_{\mathcal{E}_{j,i}}\,\right] \;=\; \sum_{j=1}^{k} \Pr[\,\mathcal{E}_{j,i}\,] \;=\; k2^{-i}. \qquad \square$$

To prove the theorem, we will need an extension of linearity of expectation to infinitely many random variables.

Fact A.3.12 (Infinite linearity of expectation). Let $X_0, X_1, X_2, \ldots$ be non-negative random variables. Then

$$\mathrm{E}\left[\sum_{i \geq 0} X_i\right] \;=\; \sum_{i \geq 0} \mathrm{E}[\,X_i\,].$$

Proof of Theorem 5.1.1. Line 9 requires only $O(\log k)$ time, using binary search, so we focus on the for loop. The i^{th} iteration takes time proportional to Z_{i-1}, the size of the set S_{i-1}. So the total time is proportional to the sum of the set sizes over all iterations. We can analyze the expected total time using infinite linearity of expectation.

$$\begin{aligned}
\mathrm{E}\Big[\sum_{i \geq 0} Z_i\Big] &= \sum_{i \geq 0} \mathrm{E}[\,Z_i\,] && \text{(by Fact A.3.12)} \\
&= \sum_{i \geq 0} k \cdot (1/2)^i && \text{(by Claim 5.1.2)} \\
&= 2k && \text{(by Fact A.2.2).} \quad \square
\end{aligned}$$

5.1.1 Improving efficiency with data structures

CATEGORICALSAMPLER is somewhat inefficient: it takes expected time $O(k)$ just to generate one sample from the categorical distribution. The underlying cause is the foreach loop, which uses time linear in

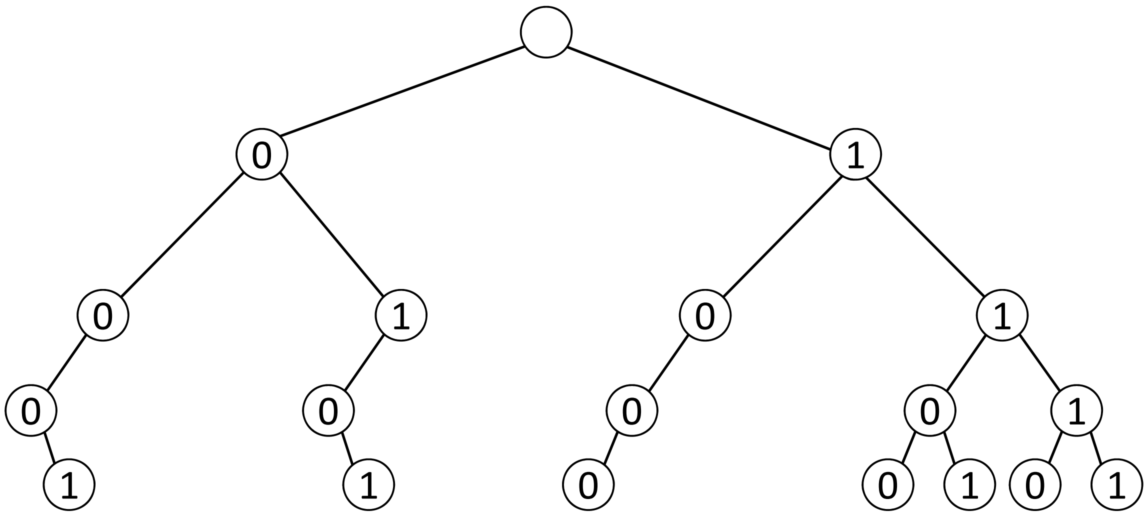


Figure 5.1: A trie that stores the binary representations $q_1, \ldots, q_7$ shown in (5.1.1).

$|S_{i-1}|$ to find the elements whose i^{th} bit match X_i. If we maintained the sets as an array in sorted order, then those q_j whose i^{th} bit is 0 would all be on the left, whereas those whose i^{th} bit is 1 would all be on the right. Perhaps we could use binary search to find the transition point between those two groups?

There is a better idea: effectively, we can precompute the location of the transition point so that the binary search becomes unnecessary. The elegant way to do this is with a ***trie*** data structure. A trie is a rooted b-ary tree that is used to store strings of symbols in $[\![b]\!]$. Each node is labeled with a single symbol. The sequence of symbols on any root-to-leaf path are exactly[1] the strings stored in the trie.

We will create a trie with $b = 2$ to store the binary representation of the values $q_1, \ldots, q_{k-1}$. If those values each use ℓ bits, then the trie will have $O(k\ell)$ nodes. This is the same as the number of bits needed to represent the q_j values anyways.

Consider the example

$$\begin{array}{llllllll} p_1 = \frac{1}{16} & p_2 = \frac{4}{16} & p_3 = \frac{3}{16} & p_4 = \frac{4}{16} & p_5 = \frac{1}{16} & p_6 = \frac{1}{16} & p_7 = \frac{1}{16} & p_8 = \frac{1}{16} \\ q_1 = \frac{1}{16} & q_2 = \frac{5}{16} & q_3 = \frac{8}{16} & q_4 = \frac{12}{16} & q_5 = \frac{13}{16} & q_6 = \frac{14}{16} & q_7 = \frac{15}{16} & q_8 = 1. \end{array}$$

The binary representations of those values are as follows.

$$\begin{array}{llll} q_1 = \langle 0.0001 \rangle & q_2 = \langle 0.0101 \rangle & q_3 = \langle 0.1000 \rangle & q_4 = \langle 0.1100 \rangle \\ q_5 = \langle 0.1101 \rangle & q_6 = \langle 0.1110 \rangle & q_7 = \langle 0.1111 \rangle & q_8 = \langle 1.0000 \rangle \end{array} \tag{5.1.1}$$

Figure 5.1 shows a trie that stores the binary representations of $q_1, \ldots, q_7$. It is not necessary to insert q_0, since X is always larger than it, and similarly for q_8.

The process of generating the bits X_i amounts to randomly walking down the trie. We can stop the process when there are no leaves beneath the current node, because that implies that X is no longer a prefix of any q_j, and hence it is comparable to every q_j.

Theorem 5.1.3. The expected number of iterations of CategoricalSamplerUsingTrie is at most $\lceil \lg k \rceil + 2$.

[1] Strings that are prefixes of other strings must be dealt with specially.

Algorithm 5.2 The input is a trie T containing the bit representation of the q_j values.

```
function CategoricalSamplerUsingTrie(trie T)
    curNode ← T.root
    for i = 1, 2, ...
        X_i ← UnbiasedBit()
        if X_i = 0 then
            curNode ← curNode.left
        else
            curNode ← curNode.right
        if curNode = Null then break                ▷ Now X is comparable to every q_j
    return the index a such that X ∈ [q_{a−1}, q_a)
end function
```

Let Z_i be the number of leaves in the subtree beneath $curNode$ after the i^{th} iteration, and let $Z_0 = k$.

Claim 5.1.4. For all $i \geq 0$, we have $\mathrm{E}[\, Z_i \,] = k2^{-i}$.

Proof. Consider the j^{th} leaf; it corresponds to the binary representation of q_j. Let $\mathcal{E}_{j,i}$ be the event that this leaf is still contained in the subtree beneath $curNode$ at the end of iteration i. This happens if the first i bits of X agree with $\langle q_j \rangle$. Since the bits of X are uniform and independent, we have $\Pr[\, \mathcal{E}_{j,i} \,] = 2^{-i}$.

We can decompose Z_i into a sum of indicators as $\sum_{j=1}^{k} \mathbf{1}_{\mathcal{E}_{i,j}}$. Then linearity of expectation yields

$$\mathrm{E}[\, Z_i \,] \;=\; \sum_{j=1}^{k} \Pr[\, \mathcal{E}_{j,i} \,] \;=\; k2^{-i}. \qquad \square$$

To prove the theorem, we will need the following useful formula.

Fact A.3.10. Let X be a random variable whose value is always a non-negative integer. Then

$$\mathrm{E}[\, X \,] \;=\; \sum_{t \geq 1} \Pr[\, X \geq t \,].$$

As a simple example, we get a solution to Exercise 4.1. For any non-negative, integer-valued random variable, we have

$$\Pr[\, X \geq 1 \,] \;\leq\; \sum_{t \geq 1} \Pr[\, X \geq t \,] \;=\; \mathrm{E}[\, X \,]. \tag{5.1.2}$$

Proof of Theorem 5.1.3. Let Y be the number of iterations performed by the algorithm. Slightly modifying Fact A.3.10, we can write

$$\mathrm{E}[\, Y \,] \;=\; \sum_{i \geq 0} \Pr[\, Y \geq i+1 \,]. \tag{5.1.3}$$

Next we note that the event $Y \geq i+1$ holds if and only if there is at least one leaf remaining at the end of iteration i. This is the same as the event that $Z_i \geq 1$. Thus, by (5.1.2) and Claim 5.1.4, we have

$$\Pr[\, Y \geq i+1 \,] \;=\; \Pr[\, Z_i \geq 1 \,] \;\leq\; \mathrm{E}[\, Z_i \,] \;=\; k2^{-i}. \tag{5.1.4}$$

This bound is useless if $\mathrm{E}[Z_i] > 1$; it would be better to bound the probability by 1. With that idea in mind, let us define

$$\tau \;=\; \lceil \lg k \rceil,$$

then use the bound

$$\Pr[Y \geq i+1] \;\leq\; \begin{cases} 1 & (\text{for } i \leq \tau - 1) \\ k2^{-i} & (\text{for } i \geq \tau). \end{cases} \tag{5.1.5}$$

Plugging this bound into (5.1.3), we obtain

$$\begin{aligned} \mathrm{E}[Y] &= \sum_{i \geq 0} \Pr[Y \geq i+1] \\ &\leq \sum_{i=0}^{\tau-1} 1 + \sum_{i \geq \tau} k2^{-i} \qquad (5.1.6) \\ &= \tau + k2^{-\tau} + k2^{-\tau-1} + k2^{-\tau-2} + \cdots \\ &= \tau + \underbrace{k2^{-\tau}}_{\leq 1} \underbrace{\Big(1 + \frac{1}{2} + \frac{1}{4} + \cdots\Big)}_{=2} \\ &\leq \tau + 2. \end{aligned}$$

In the penultimate line we have used the geometric sum formula (A.2.3). □

5.2 QuickSelect

How can one find the k^{th} smallest element in an unsorted array? If one first sorted the array, then finding the k^{th} smallest becomes trivial. The QuickSort algorithm from Section 4.4 can do this in expected $O(n \log n)$ time for an array of size n. Below we present the QuickSelect algorithm, which uses similar ideas to QuickSort. It can find the k^{th} smallest element in expected $O(n)$ time.

Algorithm 5.3 Pseudocode for Randomized QuickSelect. Returns the k^{th} smallest element of A, assuming $1 \leq k \leq |A|$. Assume that the elements in A are distinct.

```
function QuickSelect(array A, int k)
    if Length(A) = 1 then return A[1]
    Select an element p ∈ A uniformly at random                      ▷ The pivot element
    Construct the sets Left = { x ∈ A : x < p } and Right = { x ∈ A : x > p }
    Let r ← |Left| + 1                              ▷ The pivot element is the r^th smallest in A
    if k = r then return p                                       ▷ k^th smallest is the pivot
    else if r > k then return QuickSelect(Left, k)               ▷ k^th smallest in A is in Left
    else return QuickSelect(Right, k − r)                        ▷ k^th smallest in A is in Right
end function
```

The key technical step of this theorem is to understand the size of the arrays in each recursive call. This is accomplished by the following lemma. Let Z_i be the number of elements in the array A after i recursive calls, so $Z_0 = n$.

Lemma 5.2.1. For all $i \geq 0$, we have $\mathrm{E}[Z_i] \leq (7/8)^i \cdot n$.

Proof. The proof is by induction. The base case, when $i = 0$, is immediate because $Z_0 = n$.

The main idea is to show that the child subproblem's array has constant probability of being a constant factor smaller than the parent. This then implies that the *expected* size of the child is a constant factor smaller than the parent.

Consider the i^{th} recursive call. Let us fix $m = Z_{i-1}$ to be the size of A at the start of this function. Intuitively, the algorithm makes good progress if the pivot splits the array roughly in half. When this does not occur, then either r is too big or too small, which we formalize as the "bad" event

$$\mathcal{B} \;=\; \text{“}r < \lceil m/4 \rceil \;\vee\; r > \lceil 3m/4 \rceil\text{”}.$$

Since r is uniform on $[m]$, the probability is

$$\Pr\left[\,\mathcal{B}\,\right] \;=\; \frac{(\lceil m/4 \rceil - 1) + (m - \lceil 3m/4 \rceil)}{m} \;\leq\; 1/2. \tag{5.2.1}$$

If $\mathcal{B}$ occurs, then we cannot say much about the child, except that its size less than m, since it is smaller than the parent. We may write this as

$$\mathrm{E}\left[\,Z_i \mid \mathcal{B}\,\right] \;\leq\; m.$$

However if $\mathcal{B}$ does not occur then both *Left* and *Right* have size at most $3m/4$, so the child must have size at most $3m/4$ too (or, if $k = r$, the algorithm terminates). We may write this as

$$\mathrm{E}\left[\,Z_i \mid \overline{\mathcal{B}}\,\right] \;\leq\; 3m/4.$$

These bounds are combined using the Law of Total Expectation (Fact A.3.15).

$$\begin{aligned}
\mathrm{E}\left[\,Z_i\,\right] &= \mathrm{E}\left[\,Z_i \mid \mathcal{B}\,\right] \cdot \Pr\left[\,\mathcal{B}\,\right] \;+\; \mathrm{E}\left[\,Z_i \mid \overline{\mathcal{B}}\,\right] \cdot \Pr\left[\,\overline{\mathcal{B}}\,\right] \\
&\leq m \cdot \Pr\left[\,\mathcal{B}\,\right] \;+\; (3m/4) \cdot (1 - \Pr\left[\,\mathcal{B}\,\right]) \\
&= (m/4)\Pr\left[\,\mathcal{B}\,\right] + 3m/4 \\
&\leq (7/8)m \qquad \text{(by (5.2.1))}.
\end{aligned}$$

Here we have assumed that m is fixed, but in fact $m = Z_{i-1}$, which is itself random. The proper way to write this argument is to take a expectation *conditioned on* the previous array sizes, which would be written

$$\mathrm{E}\left[\,Z_i \mid Z_1, \ldots, Z_{i-1}\,\right] \;\leq\; (7/8)Z_{i-1}.$$

Taking the expectation once more yields $\mathrm{E}\left[\,Z_i\,\right] \leq (7/8)\,\mathrm{E}\left[\,Z_{i-1}\,\right]$, so the claim follows by induction. $\square$

Theorem 5.2.2. For any array A with n distinct elements, and for any $k \in [n]$, QuickSelect(A, k) performs fewer than $8n$ comparisons in expectation.

References: (Dasgupta et al., 2006, Section 2.4), (Kleinberg and Tardos, 2006, Section 13.5), (Motwani and Raghavan, 1995, Problem 1.9), (Cormen et al., 2001, Section 9.2), Wikipedia.

Proof. In the i^{th} call to the algorithm, the time required is proportional to Z_{i-1}, the size of the its array A. The total time of the algorithm is proportional to the sum of the sizes of the arrays in all recursive calls. In expectation, this is

$$\begin{aligned}
\mathrm{E}\left[\,\sum_{i \geq 0} Z_i\,\right] &= \sum_{i \geq 0} \mathrm{E}\left[\,Z_i\,\right] && \text{(by Fact A.3.12)} \\
&\leq \sum_{i \geq 0} (7/8)^i \cdot n && \text{(by Lemma 5.2.1)} \\
&= 8n && \text{(by Fact A.2.2)}. \quad \square
\end{aligned}$$

5.3 QuickSort, recursively

The techniques we have developed in this section can be used to analyze the QuickSort algorithm from Algorithm 4.2.

Theorem 5.3.1. The expected runtime of QuickSort is $O(n \log n)$.

We will show that the expected number of levels of recursion in $O(\log n)$. We begin with an easy claim showing how recursion depth relates to comparisons.

Claim 5.3.2. If the recursion tree has d levels then there are at most nd comparisons.

Proof. Each element belongs to at most one subproblem at level i. Each non-pivot element is involved in at most one comparison at level i. So there are at most n comparisons in all subroutines at level i of the recursion tree. Summing over levels proves the claim. $\square$

Proof of Theorem 5.3.1. Due to Claim 5.3.2, we just need to show that the recursion tree has $O(\log n)$ levels in expectation.

Consider a fixed element j in the array. It appears in at most one subproblem at each level of the recursion. Let $Z_{j,i}$ be the number of elements in the subproblem that contains j at recursion level i; this could be zero if j no longer appears at level i. Initially $Z_{j,0} = n$. The argument of Lemma 5.2.1 shows that

$$\mathrm{E}[\, Z_{j,i} \,] \leq (7/8)^i \cdot n. \tag{5.3.1}$$

Let L be the number of levels of recursion, excluding the root level. By Fact A.3.10, we have

$$\mathrm{E}[\, L \,] \;=\; \sum_{i \geq 1} \Pr[\, L \geq i \,].$$

We note that the event $L \geq i$ holds if and only if some element still exists in level i of the recursion. To formalize this, define $X_i = \sum_{j=1}^{n} Z_{j,i}$. The key observation is that $L \geq i$ holds if and only if $X_i \geq 1$. Thus

$$\begin{aligned}
\Pr[\, L \geq i \,] &= \Pr[\, X_i \geq 1 \,] \\
&\leq \mathrm{E}[\, X_i \,] && \text{(by Exercise 4.1)} \\
&= \sum_{j=1}^{n} \mathrm{E}[\, Z_{j,i} \,] && \text{(by linearity of expectation)} \\
&\leq n \cdot (7/8)^i \cdot n && \text{(by (5.3.1))}.
\end{aligned}$$

Following the idea of (5.1.5), we will use this bound for large i and a trivial bound for small i. The threshold between these cases is a parameter τ, to be chosen later.

$$\Pr[\, L \geq i \,] \;\leq\; \begin{cases} 1 & \text{(for } i < \tau) \\ n^2 \cdot (7/8)^i & \text{(for } i \geq \tau) \end{cases} \tag{5.3.2}$$

We plug that bound into our expression for $\mathrm{E}[L]$ to get

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{i \geq 1} \Pr[L \geq i] \\
&\leq \sum_{i=1}^{\tau-1} 1 + \sum_{i \geq \tau} n^2 \cdot (7/8)^i \\
&< \tau + n^2 (7/8)^\tau \cdot \underbrace{\sum_{i \geq 0} (7/8)^i}_{=8}.
\end{aligned} \tag{5.3.3}$$

This bound holds for all τ, so we may choose a value of τ that gives a good bound. We will choose

$$\tau = \left\lceil \log_{8/7}(n^2) \right\rceil.$$

This is a good choice for two reasons. First of all,

$$\tau \leq 1 + \frac{2\ln(n)}{\ln(8/7)} = O(\log n).$$

Second of all, by (A.2.5), we have

$$(7/8)^\tau \leq (7/8)^{\log_{8/7}(n^2)} = \frac{1}{n^2}.$$

Plugging those into (5.3.3), we get

$$\mathrm{E}[L] \leq \tau + n^2 \cdot \frac{1}{n^2} \cdot 8 = O(\log n).$$

Including the root level as well, the expected number of levels of recursion is $1 + \mathrm{E}[L] = O(\log n)$. $\square$

5.4 Exercises

Exercise 5.1 The expected end of the recursion. In the recursive algorithms of Sections 5.1 and 5.2, we have random variables

$$Z_0 \;\geq\; Z_1 \;\geq\; Z_2 \;\geq\; \cdots$$

The recursion stops at level i if $Z_i < 1$. Thus, the *expected* level at which the recursion ends is

$$\mathrm{E}\left[\, \min\{\, i \,:\, Z_i < 1 \,\}\,\right].$$

One might imagine that this is the same as the first level of the recursion at which we expect less than 1 item remaining; this would be written

$$\min\{\, i \,:\, \mathrm{E}\left[\, Z_i \,\right] < 1 \,\}.$$

These quantities are not the same, as shown in the following two parts.

Part I. Let $Z_0 = n$. Let $X_1, X_2, \ldots$ be independent Bernoulli random variables with parameter $1/2$. Let $Z_i = X_i \cdot Z_{i-1}$. Show that

$$\mathrm{E}\left[\, \min\{\, i \,:\, Z_i < 1 \,\}\,\right] \;<\; \min\{\, i \,:\, \mathrm{E}\left[\, Z_i \,\right] < 1 \,\}.$$

You may assume that $n \geq 10$.

Part II. Find two Bernoulli random variables $Z_0 \geq Z_1$ such that

$$\mathrm{E}\left[\, \min\{\, i \,:\, Z_i < 1 \,\}\,\right] \;>\; \min\{\, i \,:\, \mathrm{E}\left[\, Z_i \,\right] < 1 \,\}.$$

Exercise 5.2. Consider the following randomized form of binary search.

```
function BinarySearch(array A[1..n], int key)
    Let L ← 1, R ← n
    repeat
        Let r be a uniform random number in {L, ..., R}
        if key = A[r] then return r
        else if key > A[r] then L ← r + 1
        else if key < A[r] then R ← r − 1
    until L > R
end function
```

Prove that the expected time required is $O(\log n)$.

**** Exercise 5.3.** Improve Lemma 5.2.1 to show that, for all $i \geq 0$, we have $\mathrm{E}\left[\, Z_i \,\right] \leq (3/4)^i \cdot n$.

***** Exercise 5.4.** Let us consider QuickSort, shown in Algorithm 4.2. In this question we analyze it by expanding the expectation, and using induction.

Let $f(n)$ denote the expected number of comparisons performed by QuickSort for an array of size n. Since the expected number of comparisons performed by a subproblem only depends on the *size* of the subproblem, we can write the following recurrence.

$$\begin{aligned} f(n) &= n-1+\mathrm{E}\left[\, f(\mathit{Length}(\mathit{Left})) + f(\mathit{Length}(\mathit{Right}))\,\right] \\ &= n-1+\sum_{i=0}^{n-1} \Pr\left[\, \mathit{Length}(\mathit{Left}) = i\,\right] \cdot \big(f(i) + f(n-1-i)\big) \\ &= n-1+\frac{1}{n}\sum_{i=0}^{n-1} \big(f(i) + f(n-1-i)\big). \end{aligned}$$

By induction, prove that there is a constant c such that $f(n) \leq cn\log n$ for all $n \geq 1$.

Hint: Use Fact A.2.7 and a very careful calculation.

Chapter 6

Skip Lists

One of the most important data structures is a balanced tree. As discussed in introductory algorithms textbooks, e.g. (Cormen et al., 2001), they support INSERT, DELETE, SEARCH in $O(\log n)$ time for data structures containing n elements. Other operations (MIN, MAX, etc.) can also be efficiently implemented. There are various designs for balanced trees, including AVL trees, Red-Black trees, B-Trees, Splay Trees, etc.

One concern with these typical balanced trees is that they are all somewhat intricate to implement and to analyze. In practice, programmers would be advised to use a highly vetted implementation from a standard library rather than coding one themselves.

A Skip List is an amazingly simple randomized data structure that supports the same operations as balanced trees, and with the same runtime guarantees, in expectation or with very high probability. Their design is very easy to remember, and they are easy to implement.

6.1 A perfect Skip List

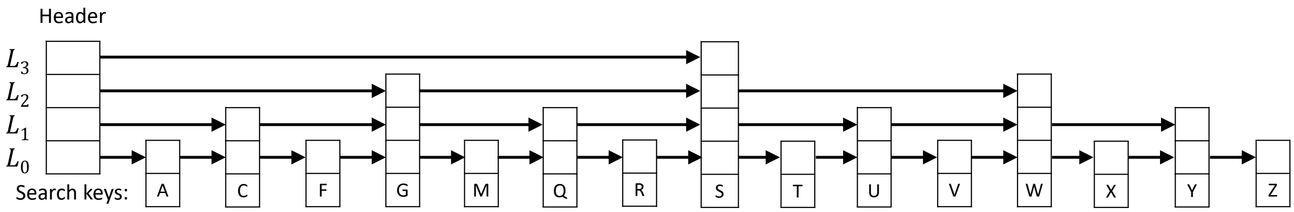
Before discussing randomized Skip Lists, let us first discuss a deterministic version that has most of the key ideas but is not practical at all.

A perfect Skip List is comprised of a collection of sorted linked lists $L_0, L_1, L_2, \ldots$. To insert an item, we create a single node that will be added the list L_0 and possibly other lists. The lists have the nested structure

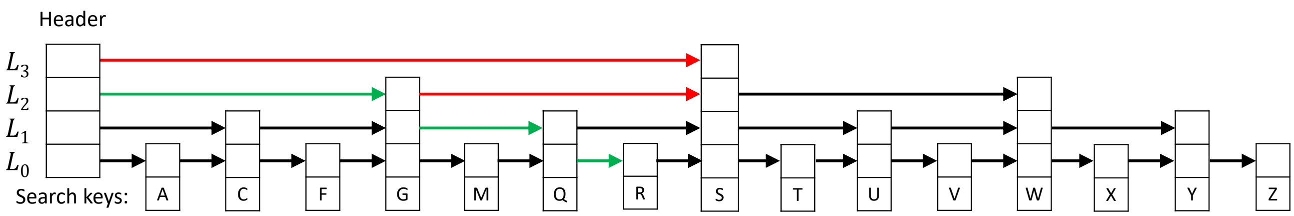
$$L_0 \supseteq L_1 \supseteq L_2 \supseteq \cdots$$

by which we mean that L_0 contains all nodes in L_1 (and possibly others), L_1 contains all nodes in L_2 (and possibly others), etc. Which nodes belong to which lists is rigidly prescribed:

- L_0 contains all of the nodes;
- L_1 contains every second node from L_0;
- L_2 contains every second node from L_1, so every fourth node from L_0;
- L_3 contains every second node from L_2, so every eighth node from L_0;
- etc.



(a) A perfect Skip List.



(b) Searching for the node R in the perfect Skip List. The red pointers are not traversed because they pass beyond the node R. The green pointers are traversed until arriving at R.

Figure 6.1

The list $L_{\lceil \lg n \rceil}$ might contain only a single node, and $L_{\lceil \lg n \rceil + 1}$ will be empty. For convenience, we will also include a *header node* that points to the first node of every list. An example of this is shown in Figure 6.1.

It is easy to perform a Search operation in a perfect Skip List. The higher level lists can be used as "highways" to rapidly make progress towards the destination node. In fact the process is nearly equivalent to binary search: initially the destination is within a range of n nodes. After the next iteration, the destination is within a range of at most $n/2$ nodes. This continues for $O(\log n)$ iterations.

Question 6.1.1. How many pointers are in a perfect Skip List?

Answer.

$$|L_0| + |L_1| + |L_2| + \cdots \;=\; n + (n/2) + (n/4) + \cdots \;=\; O(n).$$

The reason that perfect Skip Lists are not practical is that the structure is too rigid to be efficiently maintained under insertions and deletions. For example, if the node R is deleted, then S is now the seventh node so it should now only belong to L_0. The new eighth node is T, which must be added to L_1, L_2 and L_3. The node U should be removed from L_1, and so on. In the worst case, $\Omega(n)$ nodes may need their pointers adjusted. Thus, although a perfect Skip List supports Search in $O(\log n)$ time, its Insert and Delete operations require $\Omega(n)$ time, so it is no more appealing than a sorted array.

6.2 A randomized Skip List

The clever insight of Skip Lists is that we can use randomization to relax the rigidly prescribed structure of perfect Skip Lists. The idea is very simple: instead of every second node joining the next list, each node independently joins the next list *with probability* 1/2.

- L_0 contains all of the nodes;

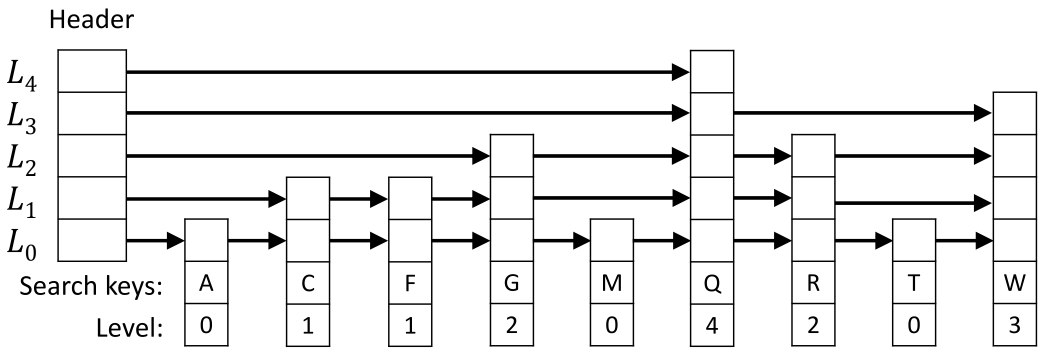


Figure 6.2: A randomized Skip List. Each node's level determines the highest list that it joins.

- L_1 contains every node from L_0 independently with probability $1/2$;
- L_2 contains every node from L_1 independently with probability $1/2$;
- L_3 contains every node from L_2 independently with probability $1/2$;
- etc.

An example of this is shown in Figure 6.2.

The intuition is that $L_{\lceil \lg(n) \rceil}$ will likely contain one or two nodes, so searching in this list will identify a range of roughly $n/2$ nodes that contain the destination. Since $L_{\lceil \lg(n) \rceil - 1}$ contains about twice as many nodes, continuing the search in this list will reduce the range to roughly $n/4$ nodes. Continuing the search in $L_{\lceil \lg(n) \rceil - 2}$ will reduce the range to roughly $n/8$ nodes, etc.

There is a clever way to construct these lists using geometric random variables. When a key v is inserted into the data structure, a new node is created to contain that key. Next, we generate a random *level* for node v, denoted $\mathrm{Level}(v)$, which determines which lists node v will join. The members of each list are easy to describe.

$$L_k \;=\; \{\, \text{node } v \,:\, \mathrm{Level}(v) \geq k \,\}.$$

To achieve the properties of a randomized Skip List, we want

$$\Pr[\,\mathrm{Level}(v) \geq 0\,] = 1, \qquad \Pr[\,\mathrm{Level}(v) \geq 1\,] = 1/2, \qquad \Pr[\,\mathrm{Level}(v) \geq 2\,] = 1/4, \qquad \ldots$$

This ensures that L_0 contains every node, since every node v has $\mathrm{Level}(v) \geq 0$. However L_1 contains every node independently with probability $1/2$, since $\Pr[\,\mathrm{Level}(v) \geq 1\,] = 1/2$.

The desired properties of $\mathrm{Level}(v)$ are achieved by a geometric RV with parameter $1/2$ that counts the number of trials *strictly before* the first success. As stated in (A.3.7), for all $a \geq 0$ we have

$$\begin{aligned} \Pr[\,\mathrm{Level}(i) = a\,] &\;=\; 2^{-(a+1)} \\ \Pr[\,\mathrm{Level}(i) \geq a\,] &\;=\; 2^{-a}. \end{aligned}$$

Properties. We are now equipped to show some basic properties of Skip Lists.

In Chapter 5 we saw several randomized recursive algorithms whose subproblems decrease exponentially with their level. An analogous property holds for Skip Lists. Let $|L_k|$ denote the size of the list L_k, excluding the header node. The next claim shows exponential decrease for $|L_k|$, in a manner completely analogous to Claim 5.1.2.

Claim 6.2.1 (Expected level size). For a Skip List with n nodes, we have $\mathrm{E}[\,|L_k|\,] = n2^{-k}$ for all $k \geq 0$.

Proof. We use decomposition into indicator random variables to write

$$|L_k| \;=\; \sum_{\text{node } i} \mathbf{1}_{i \in L_k}.$$

By the definition of the lists, the event that $i \in L_k$ is equivalent to the event $\mathrm{Level}(i) \geq k$. Thus, by linearity of expectation,

$$\mathrm{E}[\,|L_k|\,] \;=\; \sum_{\text{node } i} \mathrm{E}[\,\mathbf{1}_{i \in L_k}\,] \;=\; \sum_{\text{node } i} \Pr[\,\mathrm{Level}(i) \geq k\,] \;=\; n2^{-k}. \qquad \square$$

Due to the exponential dependence on k in Claim 6.2.1, we can determine the highest list that is expected to be non-empty. Recall that, in perfect Skip Lists, the list $L_{\lceil \lg(n) \rceil + 1}$ was definitely empty. With randomized Skip Lists, a similar property holds in expectation. Define the threshold

$$\tau \;:=\; \lceil \lg(n) \rceil\,.$$

Then Claim 6.2.1 states that $\mathrm{E}[\,|L_\tau|\,] = n2^{-\lceil \lg n \rceil} \leq 1$. So the list could be either empty or non-empty, but in expectation it has at most one node.

In fact, we can show something even stronger: the *total* size of all lists above level τ is at most 2. This idea is analogous to (5.1.6).

Claim 6.2.2 (Total expected size of levels above threshold). $\sum_{k \geq \tau} \mathrm{E}[\,|L_k|\,] \leq 2$.

Proof.

$$\begin{aligned}
\sum_{k \geq \tau} \mathrm{E}[\,|L_k|\,] &= \sum_{k \geq \tau} n2^{-k} && \text{(by Claim 6.2.1)} \\
&= n\big(2^{-\tau} + 2^{-\tau-1} + 2^{-\tau-2} + 2^{-\tau-3} + \cdots\big) \\
&= n2^{-\tau}\Big(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\Big) \\
&= n2^{-\tau} \cdot 2 && \text{(geometric sum, Fact A.2.2)} \\
&\leq 2 && \text{(by choice of } \tau\text{).} \quad \square
\end{aligned}$$

Claim 6.2.3. The expected number of non-empty lists in a Skip List is $O(\log n)$.

The proof is Exercise 6.2.

Problem to Ponder 6.2.4. Another way to analyze the expected number of non-empty lists is by calculating $\mathrm{E}[\,\max_{1 \leq i \leq n} \mathrm{Level}(i) + 1\,]$. How can you do this using Fact A.3.10 and Fact A.3.8?

6.3 Search

In Skip Lists, the key operation is Search. Once we have seen how to perform searches, the other operations are easy. Pseudocode is shown in Algorithm 6.1, and an example is shown in Figure 6.3.

Algorithm 6.1 The SkipList search algorithm. The goal is to find a node with key equal to v.

```
function Search(key v)
    Let k be the height of the header node
    The current node is the header node
    repeat
        Search linearly through list L_k for the node whose key is closest to (but less than) v,
        starting from the current node.
        k ← k − 1
    until k < 0
end function
```

Theorem 6.3.1. For a Skip List with n nodes, the expected number of nodes traversed by SEARCH is $O(\log n)$.

We will analyze the expected number of pointers traversed that do not go beyond the destination. In Figure 6.3, these are shown as green arrows. SEARCH also considers some pointers that do go beyond the destination, but it does not traverse them. In Figure 6.3, these are shown as dashed blue arrows. Since there is at most one pointer per level that is considered but not traversed, the expected number of them is $O(\log n)$ by Claim 6.2.3.

To make our notation concrete, we index the nodes in the same order as their keys: node 1, node 2, etc. Suppose we perform an *unsuccessful* search for a key whose predecessor is node d, which we will call the destination node. A successful search could traverse fewer nodes because it might find the destination sooner.

The analysis will focus on X_k, the number of nodes traversed at level k. As usual, we decompose this into a sum of indicators.

$$X_k = \sum_{1 \leq i \leq d} X_{i,k} \tag{6.3.1}$$

$$\text{where} \qquad X_{i,k} = \begin{cases} 1 & \text{(if the search traversed node } i \text{ at level } k\text{)} \\ 0 & \text{(otherwise).} \end{cases}$$

Note that the formula can restrict to nodes $i \leq d$ because the search never traverses beyond node d.

The core of the analysis is the following lemma, which says that SEARCH only traverses a small number of nodes at each level.

Lemma 6.3.2. For every level $k \geq 1$, we have $\mathrm{E}[\, X_k \,] < 2$.

Proof. The first step is to understand when the search will traverse node i at level k. This will happen precisely when the following two conditions hold.

1. Node i must be in the level k list.
2. There is no way to jump over node i.

These conditions deserve some discussion. The first condition is straightforward: it happens when node i's level is at least k. For example, in Figure 6.3 the node labelled "M" is not traversed at level 1 because it is not present in L_1. The second condition is a bit trickier. The search could jump over node i if there existed a node between i and d (possibly d itself) that belonged to a higher list. For example, in Figure 6.3 the node labelled "C" is not traversed at level 1 because the node labelled "G" is closer to the destination and at a higher level.

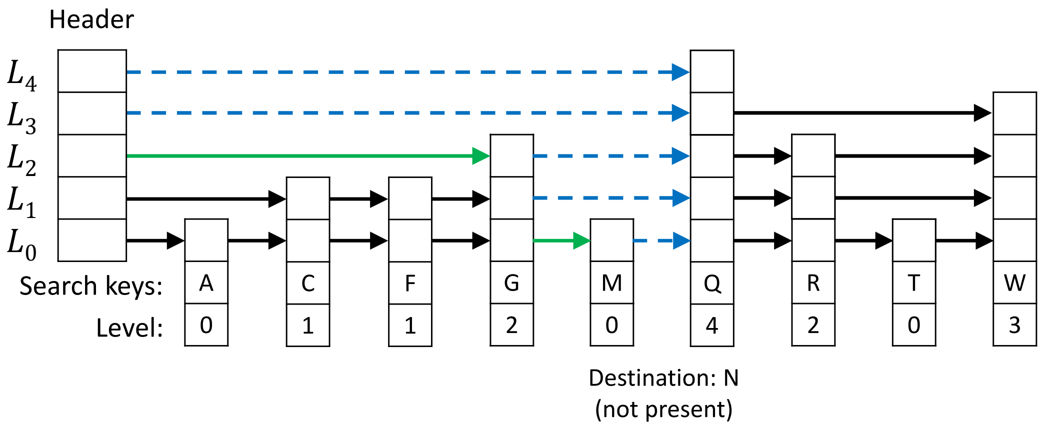


Figure 6.3: An example of an unsuccessful search in a Skip List. Green arrows are traversed towards the destination. Dotted blue arrows are not traversed because they pass beyond the destination.

Mathematically, the second condition is that $\text{Level}(j) \leq k$ for all j satisfying $i < j \leq d$. The probability that this condition holds is not hard to analyze because the nodes' levels are independent.

$$\begin{aligned}
\Pr[\, X_{i,k} = 1\,] &= \Pr[\,\text{node } i \text{ in level } k\,] \cdot \Pr[\,\text{cannot jump over node } i\,] \\
&= \Pr[\,\text{Level}(i) \geq k\,] \cdot \prod_{i<j\leq d} \Pr[\,\text{Level}(j) < k+1\,] \\
&= 2^{-k} \cdot \prod_{i<j\leq d} (1 - 2^{-(k+1)}) \qquad \text{(by (A.3.7))} \\
&= \underbrace{2^{-k} \cdot (1 - 2^{-(k+1)})^{d-i}}_{=f_k(i)}. \qquad (6.3.2)
\end{aligned}$$

This formula is important, so we will denote it $f_k(i)$. It is an exact formula for the probability that $\text{SEARCH}(d)$ traverses node i at level k. Admittedly, it is not very easy to understand. The first factor is exponentially decreasing in k, which makes sense: a given node is unlikely to belong to higher levels. The second factor is exponentially decreasing in $d - i$, the distance between node i and the destination. This is also intuitive: nodes that are farther from the destination are more likely to be skipped over at higher levels. Some plots of this formula are shown in Figure 6.4.

We can complete the proof using our familiar techniques of decomposition into indicator random variables and the formula for geometric sums.

$$\begin{aligned}
\mathrm{E}[\, X_k\,] &= \sum_{1\leq i\leq d} \Pr[\, X_{i,k} = 1\,] && \text{((6.3.1) and linearity of expectation)} \\
&= \sum_{1\leq i\leq d} 2^{-k} \cdot (1 - 2^{-k-1})^{d-i} && \text{(by (6.3.2))} \\
&< 2^{-k} \sum_{\ell\geq 0} (1 - 2^{-k-1})^{\ell} && \text{(expanding to infinite sum)} \\
&= 2^{-k} \frac{1}{1 - (1 - 2^{-k-1})} && \text{(geometric sum, Fact A.2.2)} \\
&= 2. \quad \square
\end{aligned}$$

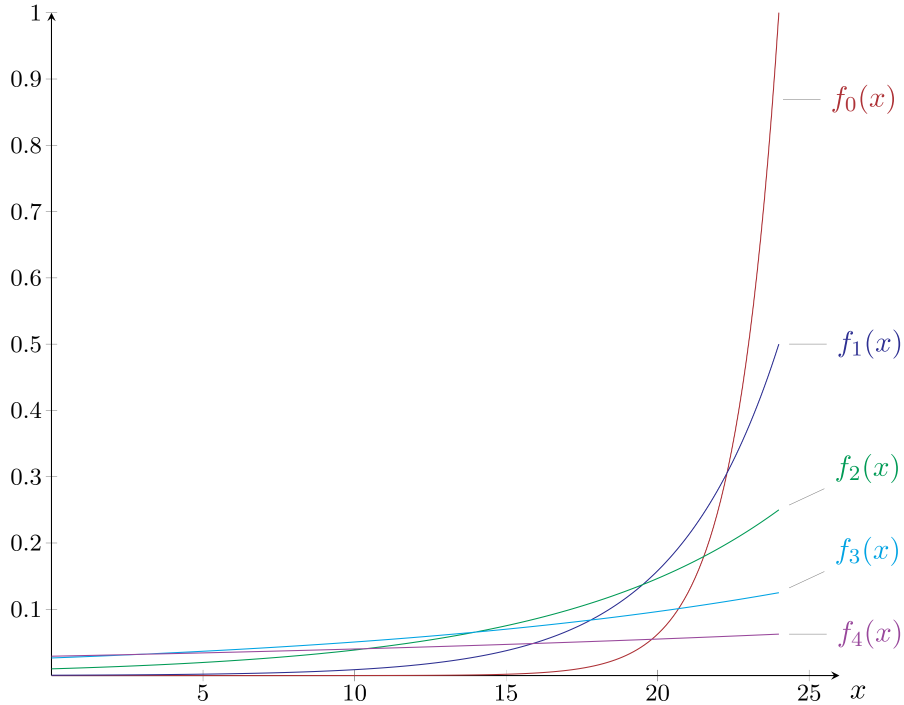


Figure 6.4: Plots of the function $f_k(i) = 2^{-k} \cdot (1 - 2^{-(k+1)})^{d-i}$ for $d = 24$, as a function of i and with various values of k. The maroon curve plots f_0. At level 0, Search is very likely to traverse nodes 23 and 24, but very unlikely to traverse any node before 20. The blue curve plots f_1. At level 1, Search is somewhat likely to traverse nodes 20-24, but unlikely to traverse any node before 15. The green curve plots f_2. The light blue curve plots f_3. The purple curve plots f_4. At level 4, Search has a nearly constant probability of traversing all nodes, but the overall probability is quite low.

Next we use this lemma to show that the expected number of nodes traversed by Search is $O(\log n)$. The intuition is straightforward: Lemma 6.3.2 shows that the number of nodes traversed per level is constant in expectation, and we know from Claim 6.2.3 that the expected number of non-empty levels is $O(\log n)$. The formal proof is slightly different.

Proof of Theorem 6.3.1. Following the idea of Theorem 5.1.1, we will separately analyze the lower levels (which are probably not empty) and higher levels (which are probably nearly empty).

$$\begin{aligned}
\mathrm{E}\,[\,\text{total \# nodes traversed}\,] &= \sum_{k \geq 0} \mathrm{E}\,[\,X_k\,] && \text{(by Fact A.3.12)} \\
&= \sum_{k=0}^{\tau-1} \mathrm{E}\,[\,X_k\,] + \sum_{k \geq \tau} \mathrm{E}\,[\,X_k\,] && \text{(separating lower and higher levels)} \\
&< 2\tau + 2 && \text{(by Lemma 6.3.2 and Claim 6.2.2)} \\
&= O(\log n). \quad \square
\end{aligned}$$

6.4 Delete

Whereas deletion in a balanced binary tree requires some care, deletion is a Skip List is almost effortless.

Algorithm 6.2 The Skip List deletion algorithm.

```
function DeleteGivenKey(key q)
    Let v be the node found by Search(q)
    DeleteGivenNode(v)
end function
function DeleteGivenNode(node v)
    for i = 0, ..., Level(v) do
        Remove v from L_i
        If L_i is now empty, remove it from the header node
    end for
end function
```

Question 6.4.1. Assuming that the lists are doubly-linked, what is the runtime of DeleteGivenNode?

Answer.

First let us ignore the maintenance of the header node. The number of iterations is $\text{Level}(v) + 1$. Since $\text{Level}(v)$ is a geometric random variable with $p = 1/2$, the expected number of iterations is $O(1)$. If the lists are doubly-linked, then removing node v can be done in $O(1)$ time. Depending on the implementation of the header node, it could take $O(1)$ time to remove empty lists.

Question 6.4.2. What is the runtime of DeleteGivenKey?

Answer.

The runtime is the same as Search, which is $O(\log n)$ time in expectation.

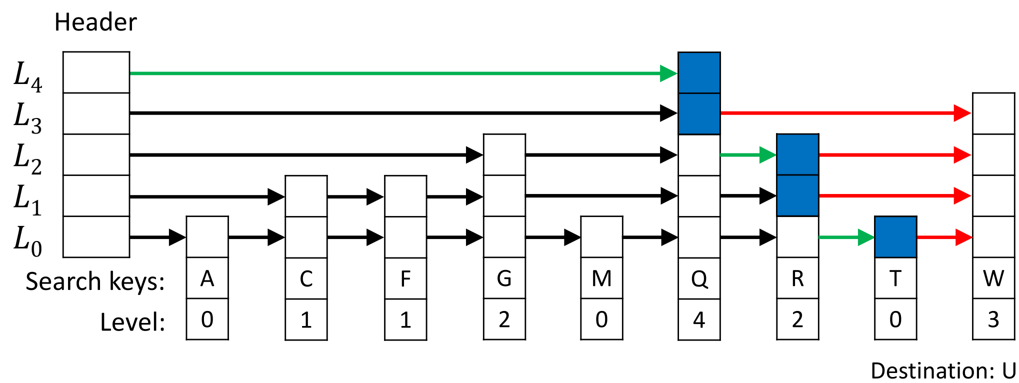
There is one minor detail that deserves some discussion: in DeleteGivenNode, how should we remove the node v from each list L_i? If the lists were doubly-linked, then this step is trivial. Node v would have pointers to its predecessor and successor in each list, so in $O(1)$ time we can update their pointers to point to each other. If the lists are singly-linked then we cannot immediately find the predecessor. However, there is a simple solution: the Search operation can return the last node traversed in each list. This would contain the predecessor of node v in every list to which it belongs.

6.5 Insert

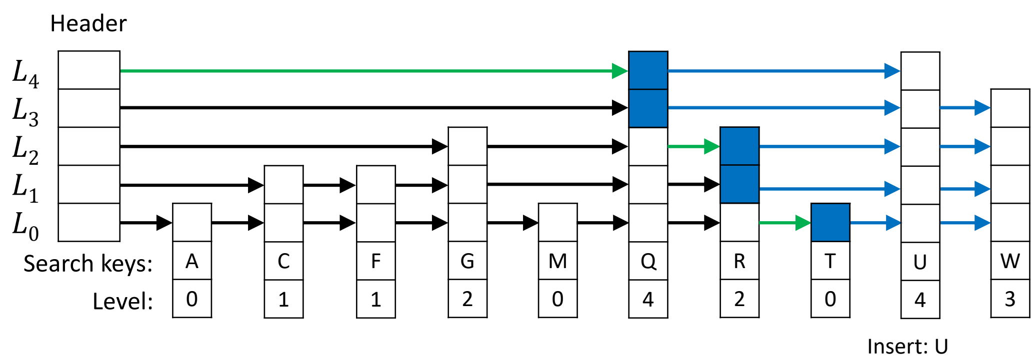
Insertion into a Skip List is also fairly simple, but does require some discussion. First let's consider the most naive algorithm.

Algorithm 6.3 An inefficient algorithm to insert v into a Skip List.

```
function SlowInsert(key q)
    Create a new node v with key q.
    Generate Level(v), a geometric RV with parameter p = 1/2.
    for i = 0, ..., Level(v) do
        If L_i does not exist, add it to the header node.
        Insert v into the sorted list L_i using linear search.
    end for
end function
```



(a) Suppose we perform SEARCH(U). The green arrows indicate the pointers that are traversed. The red arrows indicate the pointers that extend beyond the destination. The blue boxes indicate the *last* node traversed at each level.



(b) To insert the node U, we insert it into each list L_i immediately after the last node traversed in that list during SEARCH (i.e., after the blue box).

Figure 6.5

It is easy to see that this algorithm takes $\Omega(n)$ time, simply due to the linear search in L_1. To improve the speed of insertion we must quickly find the location at which the new node is to be inserted. Remarkably, this turns out to be easy: the Skip List SEARCH algorithm already identifies, in every list, the exact location at which the new node should be inserted! This is illustrated in Figure 6.5.

Algorithm 6.4 The SkipList insertion algorithm. The differences from the SEARCH algorithm are shown highlighted in yellow.

function INSERT(key q)
  Create a new node v with key q. Let Level(v) be an independent geometric random variable.
  If Level(v) exceeds the height of the Skip List, increase the height of the header node to Level(v)
  Let k be the height of the header node
  The current node is the header node
  repeat
    Search through list L_k for the node whose key is closest to (but less than) v, starting from the current node.
    If $k \leq$ Level(v), then insert v into L_k immediately after the current node.
    $k \leftarrow k - 1$
  until $k = 0$
end function

Corollary 6.5.1. The expected time for INSERT is $O(\log n)$.

Proof. The runtime of INSERT is proportional to the length of the SEARCH path, which we have shown to be $O(\log n)$ in expectation. □

6.6 Exercises

Exercise 6.1. Consider a Skip List with n nodes, so $|L_0| = n$. Prove that the total number of pointers used is $O(n)$ in expectation.

Exercise 6.2. In this exercise we consider Claim 6.2.3, which states that the expected number of non-empty lists in a Skip List is $O(\log n)$. For simplicity, we assume that n is a power of two.

Part I. Prove that

$$\min\{\, k \,:\, \mathrm{E}[\,|L_k|\,] < 1 \,\} \;=\; \lg(n) + 1.$$

Does this prove Claim 6.2.3?

Part II. Prove that

$$\mathrm{E}[\,|\{\, k \,:\, |L_k| \geq 1 \,\}|\,] \;\leq\; \lg(n) + 2.$$

Does this prove Claim 6.2.3?

Hint: Use Fact A.3.12 and Claim 6.2.2.

Exercise 6.3. Let H be the number of non-empty levels in a Skip List with n nodes. Use a union bound to prove that $\Pr[\,H \geq 100 \lg n\,] \leq O(1/n^{99})$.

Exercise 6.4. Suppose we have two Skip Lists: A with m nodes, and B with n nodes. Describe an algorithm to merge A and B into a single Skip List containing $n+m$ nodes. You cannot assume that all keys in A are less than all keys in B, or vice versa; they could be arbitrarily intermixed. Your algorithm should have expected runtime $O(n+m)$. (For simplicity, you may assume all keys are distinct.)

Chapter 7

Balls and Bins

7.1 Example problems

Many problems in computer science can be viewed mathematically as a random process where balls are thrown into bins. One canonical example is load balancing in a distributed system, and another is inserting items into a hash table.

Example 7.1.1. Suppose there are m clients that access a website. There are n identical servers that host the website. A load balancer is used that assigns each client to one of the servers at random, uniformly and independently. We would like to study properties of the load distribution on the servers.

Example 7.1.2. Suppose that there is a hash table with n locations, into which we would like to insert m items. An idealistic model is that the hash function is a ***purely random function***. We would like to analyze the number of items that hash to each location.

We will return to the topic of hash functions in Chapter 12, but for now let us briefly discuss what is meant by a purely random function. In brief, we can build such a function by generating a uniformly random output on $[n]$ for every possible input, such that these outputs are mutually independent.

7.2 The number of collisions

A UUID is a 128-bit identifier widely used in filesystems, databases, distributed systems, etc. Several different formats exist, but a commonly used format has 122 random bits.

Imagine that you work at a tech company that stores an enormous number of files in their data centers. You would like the files to have unique identifiers, so imagine assigning each file a random UUID. If the number of files is not too large one would imagine that their identifiers should all be unique.

If two files are randomly assigned the same identifier, we call this a ***collision***. To understand if using random UUIDs is a good idea, we now analyze the number of collisions. Let n be the number of distinct identifiers, which is 2^{122} in the case of UUIDs. Suppose that m files are stored. How small must m be in order for there to be less than one collision, in expectation?

We can think of this more abstractly as a "balls and bins" problem. The n distinct identifiers correspond to n bins, and the m files correspond to m balls that are randomly thrown into the bins.

Theorem 7.2.1. If $m = \sqrt{2n}$ balls are thrown at random into n bins, then the expected number of collisions is less than 1.

Proof. Let X be the number of *pairs* of balls that collide (i.e., land in the same bin). We can decompose X into a sum of indicator random variables as follows. Let $\mathcal{E}_{i,j}$ be the event that ball i and ball j collide. Then

$$X \;=\; \sum_{1 \leq i < j \leq m} \mathbf{1}_{\mathcal{E}_{i,j}}.$$

By linearity of expectation,

$$\mathrm{E}[\,X\,] \;=\; \sum_{i<j} \mathrm{E}\left[\,\mathbf{1}_{\mathcal{E}_{i,j}}\,\right] \;=\; \sum_{i<j} \Pr[\,\text{ball } i \text{ and ball } j \text{ collide}\,]. \tag{7.2.1}$$

What is the probability that two particular balls collide? Using the law of total probability and independence, we get

$$\begin{aligned}
&\Pr[\,\text{ball } i \text{ and ball } j \text{ collide}\,] \\
&\quad= \sum_{k=1}^{n} \Pr[\,\text{ball } i \text{ and ball } j \text{ both land in bin } k\,] && \text{(by Fact A.3.6)} \\
&\quad= \sum_{k=1}^{n} \Pr[\,\text{ball } i \text{ lands in bin } k\,] \cdot \Pr[\,\text{ball } j \text{ lands in bin } k\,] && \text{(independence)} \\
&\quad= \sum_{k=1}^{n} (1/n) \cdot (1/n) && \text{(uniform distribution)} \\
&\quad= n \cdot (1/n)^2 \\
&\quad= 1/n.
\end{aligned}$$

We plug this into (7.2.1) and bound the binomial coefficient with Fact A.2.8.

$$\mathrm{E}[\,\#\text{ collisions}\,] \;=\; \mathrm{E}[\,X\,] \;=\; \sum_{1 \leq i < j \leq m} (1/n) \;=\; \binom{m}{2}(1/n) \;<\; \frac{m^2}{2n} \tag{7.2.2}$$

Thus, if $m \leq \sqrt{2n}$, then there is fewer than one collision in expectation. □

Let us interpret this result in our original setting of UUIDs. There are $n = 2^{122}$ identifiers, which correspond to the n bins. There are m files, which correspond to the balls. According to Theorem 7.2.1, we expect less than than one collision if $m \leq \sqrt{2} \cdot 2^{61} \approx 3.3 \cdot 10^{18}$.

Question 7.2.2. Suppose that the world has 7 billion people, and each person stores k files. How large can k be so that (7.2.2) ensures there will be at most 1 collision?

Answer.

We need $7,000,000,000 \cdot k \leq 2^{61}$, which holds when $k \leq 329,406,144$.

The Birthday Paradox. This phenomenon described above often appears in introductory probability textbooks in a setting called "the birthday paradox". In this setting, there are m people who have uniformly random and independent birthdays among the $n = 365$ days of the year. A collision corresponds to two people sharing a birthday. How small must m be in order to have at most one collision in expectation? Using the formula (7.2.2) above, it would suffice to have $m \leq \sqrt{2 \cdot 365} \approx 27$ people. Because 27 is strikingly smaller than 365, this phenomenon is viewed as quite counterintuitive, which is why it is called a "paradox".

Probability instead of expectation. There are two different mathematical interpretations of how to choose m in order to avoid collisions.

- How can we choose m so that the expected number of collisions is less than 1?
- How can we choose m so that the probability of having no collisions is at least $1/2$?

These questions are not equivalent, and their answers are slightly different. Above we have studied only the first question, which is the simpler of the two. One connection between these questions is explored in Exercise 7.3.

References: The second question is studied in (Lehman et al., 2018, Section 17.4.1), (Anderson et al., 2017, Example 2.44), (Cormen et al., 2001, Section 5.4.1), (Mitzenmacher and Upfal, 2005, Section 5.1), (Motwani and Raghavan, 1995, Section 3.1).

7.3 The number of empty bins

Let's return to the load balancing setting discussed in Example 7.1.1. For simplicitly let's imagine that the number of clients m equals the number of servers n. Thus, in principle, we could explicitly match the clients with the servers so that each server receives exactly one client. However, since we are assigning clients to servers randomly, it may happen that some server receives no clients at all. We would like to analyze the expected number of servers that do not receive any clients.

As before, we can think of this more abstractly as a "balls and bins" problem. The n servers correspond to the bins, and the m clients correspond to balls that are randomly thrown into the bins. Our task is to analyze the number of ***empty bins***, i.e., those that do not receive any ball.

Let X be the random variable giving the number of empty bins. We can decompose this into a sum of indicator random variables as

$$X \;=\; \sum_{i=1}^{n} \mathbf{1}_{\mathcal{E}_i},$$

where $\mathcal{E}_i$ is the event that bin i is empty. By linearity of expectation,

$$\mathrm{E}\,[\,X\,] \;=\; \sum_{i=1}^{n} \mathrm{E}\,[\,\mathbf{1}_{\mathcal{E}_i}\,] \;=\; \sum_{i=1}^{n} \Pr\,[\,\mathcal{E}_i\,]. \tag{7.3.1}$$

It remains to analyze these probabilities.

We observe that bin i is empty if and only if every ball fails to land in bin i. Thus,

$$\begin{aligned}
\Pr\,[\,\mathcal{E}_i\,] &= \Pr\,[\,\text{every ball avoids bin } i\,] \\
&= \prod_{j=1}^{m} \Pr\,[\,\text{ball } j \text{ avoids bin } i\,] && \text{(independence)} \\
&= (1-1/n)^m.
\end{aligned}$$

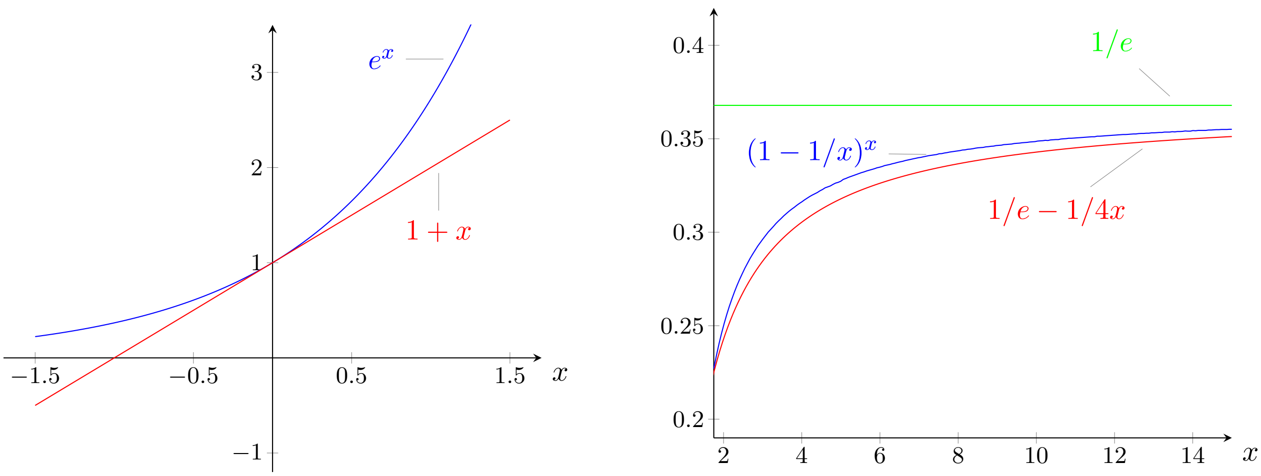
Now we introduce a math trick that is very simple but extremely useful.

Fact A.2.5 (Approximating e^x near zero). For all real numbers x,

$$1 + x \leq e^x.$$

Moreover, for x close to zero, we have $1 + x \approx e^x$.

This fact is illustrated in Figure 7.1.



(a) The function e^x is convex, whereas $1 + x$ is a straight line that lies beneath it and touches it at the point $(0, 1)$.

(b) The function $(1 - 1/x)^x$ is increasing, and is sandwiched between the functions $1/e - 1/4x$ and $1/e$. All of them converge to $1/e$ as $x \to \infty$.

Figure 7.1

Upper bound on the expectation. Applying the trick, we have

$$\Pr[\,\text{bin } i \text{ is empty}\,] \;=\; (1 - 1/n)^m \;\leq\; \left(e^{-1/n}\right)^m \;=\; e^{-n/m}. \tag{7.3.2}$$

Plugging this into (7.3.1) and using the assumption $m = n$,

$$\mathrm{E}[\,X\,] \;=\; \sum_{i=1}^{n} \Pr[\,\text{bin } i \text{ is empty}\,] \;\leq\; \sum_{i=1}^{n} e^{-m/n} \;=\; n/e \;<\; 0.37n. \tag{7.3.3}$$

This says that, when throwing n balls randomly into n bins, the number of empty bins will be at most 37%. This result feels unsettling — could it really be the case that over a third of the bins are empty? However, note that the analysis is giving an *upper bound* on the number of empty bins. Perhaps the upper bound is loose, and the actual number of empty bins is much smaller?

Lower bound on the expectation. We now show that the preceding analysis is actually tight for large n. To do so, we will need another mathematical fact that is a counterpart to Fact A.2.5.

Fact A.2.6 (Approximating $1/e$). For all $n \geq 2$,

$$\frac{1}{5} \;<\; \frac{1}{e} - \frac{1}{4n} \;\leq\; \left(1 - \frac{1}{n}\right)^n \;\leq\; \frac{1}{e} \;<\; 0.37.$$

We can lower bound the probability that a bin is empty by plugging into (7.3.2).

$$\Pr[\,\text{bin } i \text{ is empty}\,] \;=\; (1 - 1/n)^n \;\geq\; 1/e - 1/4n$$

Next we can lower bound the expected number of empty bins by plugging into (7.3.1).

$$\mathrm{E}[\,X\,] \;=\; \sum_{i=1}^{n} \Pr[\,\text{bin } i \text{ is empty}\,] \;\geq\; n/e - 1/4. \tag{7.3.4}$$

Combining our upper bound (7.3.3) and lower bound (7.3.4), we obtain a very tight estimate on the expected number of empty servers.

$$\frac{n}{e} - \frac{1}{4} \;\leq\; \mathrm{E}[\,X\,] \;\leq\; \frac{n}{e}.$$

Since the $-1/4$ term is negligible for large n, we are justified in saying that the expected number of empty bins is about $n/e \approx 0.37n$.

References: (Cormen et al., 2001, exercise C.4-5).

Problem to Ponder 7.3.1. Let us say that a ball is ***isolated*** if it is the only ball in its bin. What is the expected number of isolated balls?

7.4 Avoiding empty bins

In a distributed load balancing system, unused servers seem wasteful. Intuitively, if we added enough clients, then it seems likely that each server would receive at least one client. This section justifies that intuition by analyzing the probability that every server receives at least one client.

As before, this can be viewed more abstractly as a balls and bins problem. Suppose we throw m balls (the clients) into n bins (the servers). We would like m to be large enough that there are no empty bins. How large should m be?

Our analysis from the previous section is very relevant. If we focus on the i^{th} bin, (7.3.2) shows that

$$\Pr[\,\text{bin } i \text{ is empty}\,] \;\leq\; e^{-m/n} \qquad \forall i \in [n]. \tag{7.4.1}$$

However, this analysis holds for each bin *individually*, and does does not immediately tell us about the probability that *there exists* some empty bin. To analyze the latter probability we must introduce an extremely important trick in probability theory.

Fact A.3.8 (The union bound). Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be *any* collection of events. They could be dependent and do not need to be disjoint. Then

$$\Pr[\,\text{any of the events occurs}\,] \;=\; \Pr[\,\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n\,] \;\leq\; \sum_{i=1}^{n} \Pr[\,\mathcal{E}_i\,].$$

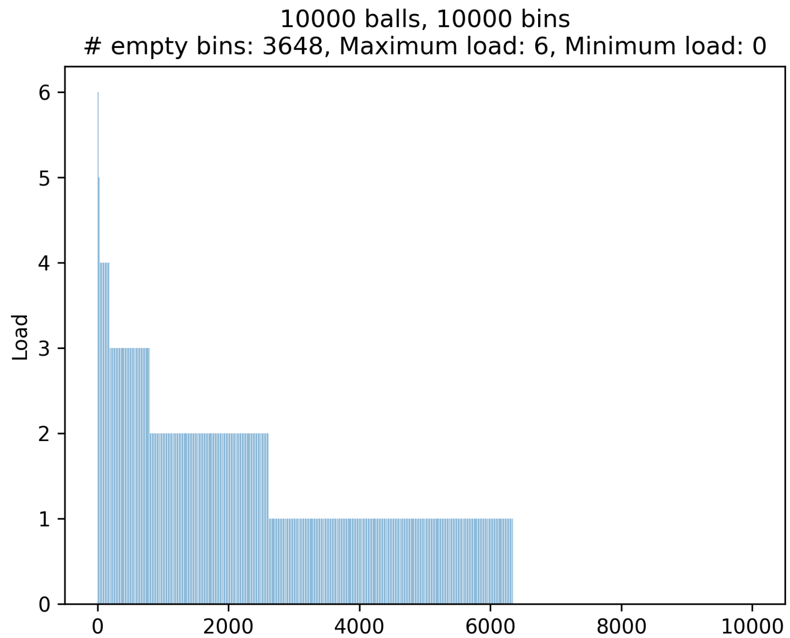
We apply this fact with $\mathcal{E}_i$ being the event that bin i is empty. We obtain

$$\begin{aligned}
\Pr[\,\text{any bin is empty}\,] &= \Pr[\,\mathcal{E}_1 \vee \mathcal{E}_2 \vee \cdots \vee \mathcal{E}_n\,] \\
&\leq \sum_{i=1}^{n} \Pr[\,\mathcal{E}_i\,] && \text{(by Fact A.3.8)} \\
&\leq \sum_{i=1}^{n} e^{-m/n} && \text{(by (7.4.1))} \\
&= ne^{-m/n}.
\end{aligned}$$

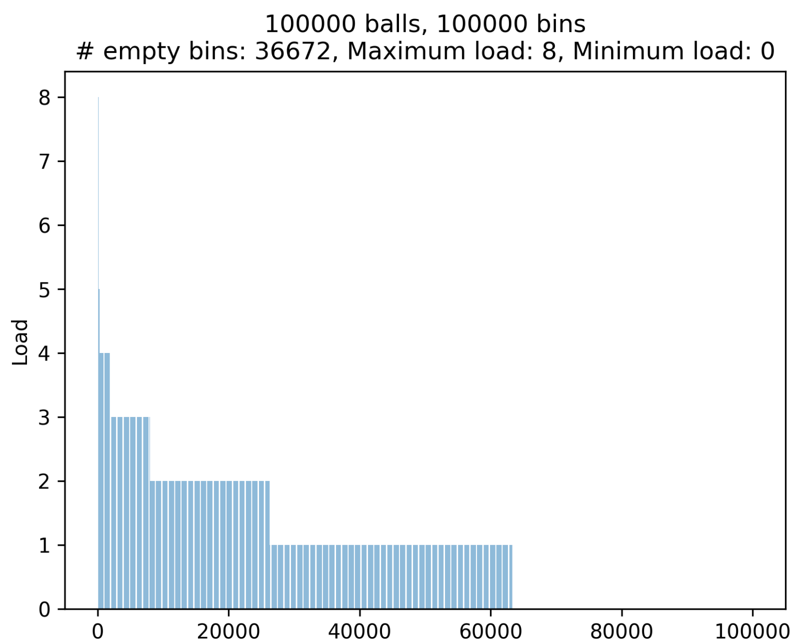
This bound is decreasing in m, so we can make it as small as we like by choosing m to be large enough. In particular, choosing $m = n\ln(2n)$ we obtain the bound

$$\Pr[\,\text{any bin is empty}\,] \;\leq\; ne^{-m/n} \;=\; n \cdot e^{-\ln(2n)} \;=\; n \cdot \frac{1}{2n} \;=\; \frac{1}{2}.$$

We may summarize this discussion with the following theorem.



(a)



(b)

Figure 7.2: Examples of randomly throwing n balls into n bins. The bins are sorted in decreasing order by the number of balls that they contain. Note that the number of empty bins is approximately $0.37n$. (a) $n = 10,000$. (b) $n = 100,000$.

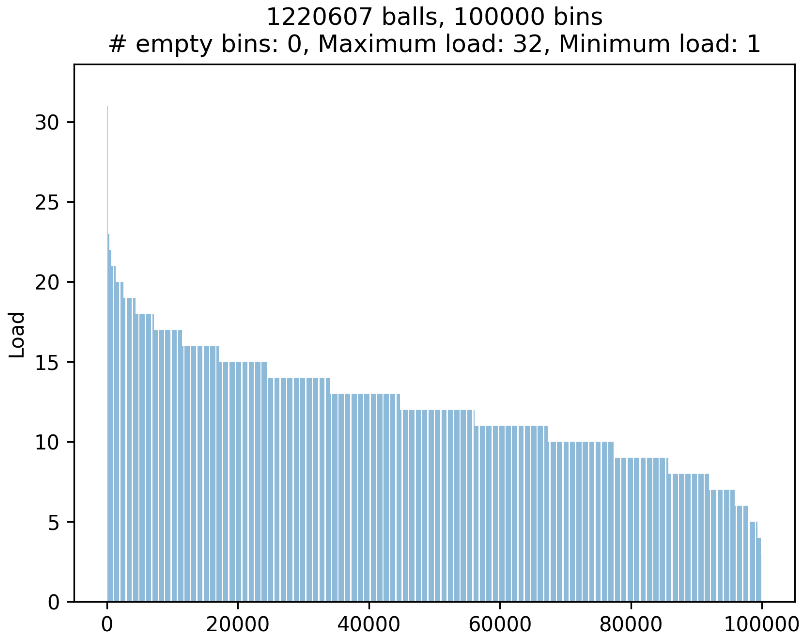


Figure 7.3: An example of randomly throwing $m = n\ln(2n)$ balls into n bins. The bins are sorted in decreasing order. In this example, $n = 100,000$ and $m = 1,220,607$.

Theorem 7.4.1. Consider throwing $m = n\ln(2n)$ balls randomly into n bins. Then with probability at least $1/2$, every bin has at least one ball.

The Coupon Collector Problem. This phenomenon described above often appears in introductory probability textbooks in a setting called "the coupon collector problem". In this setting, there are n different coupons. How many coupons must one draw in order that it is likely to have at least one coupon of each type? The theorem above states that $n\ln(2n)$ trials will suffice to succeed with probability $1/2$.

This extra $\Theta(\log n)$ factor is perhaps unexpected, but it is necessary. There are many problems in computer science that relate to the coupon collector problem, so it is valuable to understand that this extra $\Theta(\log n)$ factor is fundamental, and why it arises.

References: (Lehman et al., 2018, Section 19.5.4), (Anderson et al., 2017, Example 8.17), (Cormen et al., 2001, Section 5.4.2), (Mitzenmacher and Upfal, 2005, Example 2.4.1), (Motwani and Raghavan, 1995, Section 3.6).

7.4.1 Expectation instead of probability

How many balls should we add in order for each bin to be non-empty? There are two different mathematical interpretations of this question.

- How can we choose a fixed value of m so that the probability that all bins are non-empty is at least $1/2$?
- Suppose we keep adding balls until every bin definitely has at least one ball. Let X be the random variable giving the number of balls added. What is the expected value of X?

These questions are certainly not equivalent because X is random and m is not. Theorem 7.4.1 provides an answer for the first question. Next we consider the second question.

The main idea is to imagine throwing balls in n phases. In the i^{th} phase we keep throwing balls until there are exactly i non-empty bins. Let X_i be the RV giving the number of balls thrown during in the i^{th} phase. The total number of balls is then simply

$$X \;=\; \sum_{i=1}^{n} X_i.$$

To analyze X, we analyze each X_i random variable separately. We may view the i^{th} phase as a series of independent trials, in which each trial throws a new ball into a random bin. The trial is considered a success if that bin is currently empty, and otherwise it is a failure. Each trial during the i^{th} phase has probability of success

$$p_i \;=\; \frac{\text{number of empty bins}}{\text{number of bins}} \;=\; \frac{n-i+1}{n}. \tag{7.4.2}$$

Since X_i is the number of trials during phase i until the first success, it is a geometric RV with parameter p_i. The formula in (A.3.8) tells us that

$$\mathrm{E}\left[\, X_i \,\right] \;=\; \frac{1}{p_i} \;=\; \frac{n}{n-i+1}. \tag{7.4.3}$$

The total number of balls thrown added can be analyzed as follows.

$$\begin{aligned}
\mathrm{E}\left[\, X \,\right] \;&=\; \sum_{i=1}^{n} \mathrm{E}\left[\, X_i \,\right] && \text{(linearity of expectation)}\\
&=\; \sum_{i=1}^{n} \frac{n}{n-i+1} && \text{(by (7.4.3))}\\
&=\; n\Big(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} + \frac{1}{1}\Big)\\
&\leq\; n\big(\ln(n)+1\big) = n\ln(en) && \text{(by Fact A.2.1).}
\end{aligned}$$

This proves the following theorem.

Theorem 7.4.2. Consider throwing balls randomly into n bins until every bin has at least one ball. The expected number of balls thrown is at most $n\ln(en)$.

It is interesting to observe that Theorem 7.4.1 and Theorem 7.4.2 are very similar statements, but their proofs have different ingredients. The former uses an approximation of e^x (Fact A.2.5) whereas the latter uses a harmonic sum (Fact A.2.1).

Question 7.4.3. Theorem 7.4.2 proves an upper bound on the expected number of balls needed. Is it tight? In other words, it true that

$$\mathrm{E}\left[\,\text{number of balls added until no empty bins}\,\right] \;=\; \Theta(n\log n)?$$

Answer.

Yes, because Fact A.2.1 shows that the harmonic sum $1 + 1/2 + \cdots + 1/n = \Theta(\log n)$.

References: (Lehman et al., 2018, Section 19.5.4), (Cormen et al., 2001, Section 5.4.2), (Kleinberg and Tardos, 2006, Theorem (13.13)), (Motwani and Raghavan, 1995, Section 3.6.1), (Mitzenmacher and Upfal, 2005, Section 2.4.1).

7.5 Further questions

There are numerous interesting questions relating to balls and bins, many of which capture the mathematics of interesting computer science problems. In this section we briefly state a few known results but without giving any proofs.

Load on heaviest bin. When throwing n balls into n bins, we showed that about $0.37n$ bins will be empty in expectation. This means that some bins will have more than one ball. What is the maximum load on any bin? (Here ***load*** refers to the number of balls in a bin.)

This innocuous question has a somewhat unexpected answer: the maximum load is very likely to be $\Theta(\log n/\log\log n)$. To get a sense of how large this is, note that

$$(\ln n)^{0.99} \;<\; \frac{\ln n}{\ln\ln n} \;<\; \ln n$$

if n is large enough. In Section 7.6 we will prove the upper bound: the maximum load is very likely $O(\log n/\log\log n)$.

References: (Raab and Steger, 1998).

Nearly equal load. In Section 7.4 we showed that throwing $n\ln(2n)$ balls randomly into n bins will likely cause each bin to have at least one ball. On the other hand the maximum load on any bin is at least the average load, which is $\ln(2n)$. Thus, the ratio between the maximum and minimum loads could be $\Omega(\log n)$.

Intuitively, as we throw more balls, the bins should eventually even out, and the ratio between the maximum and minimum load should decrease to a constant. This phenomenon is illustrated in Figure 7.4 (a). How large should m be to ensure that this happens? In Section 10.1 we show that, if the number of balls is $m = 12n\ln(200n)$ then

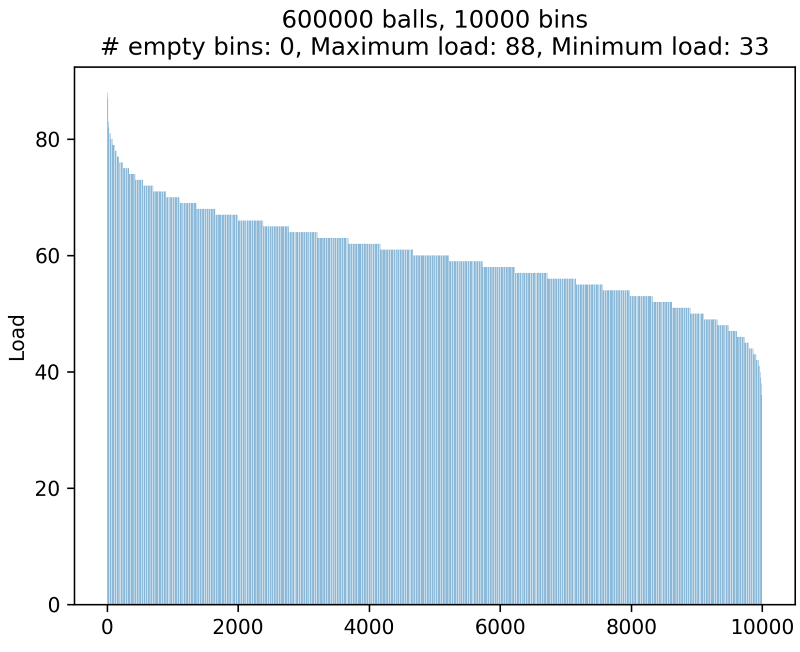
$$\frac{\text{maximum load}}{\text{minimum load}} \;\leq\; 3$$

with probability at least 0.99. We will need some more probabilistic tools from Chapter 9 in order to prove this result.

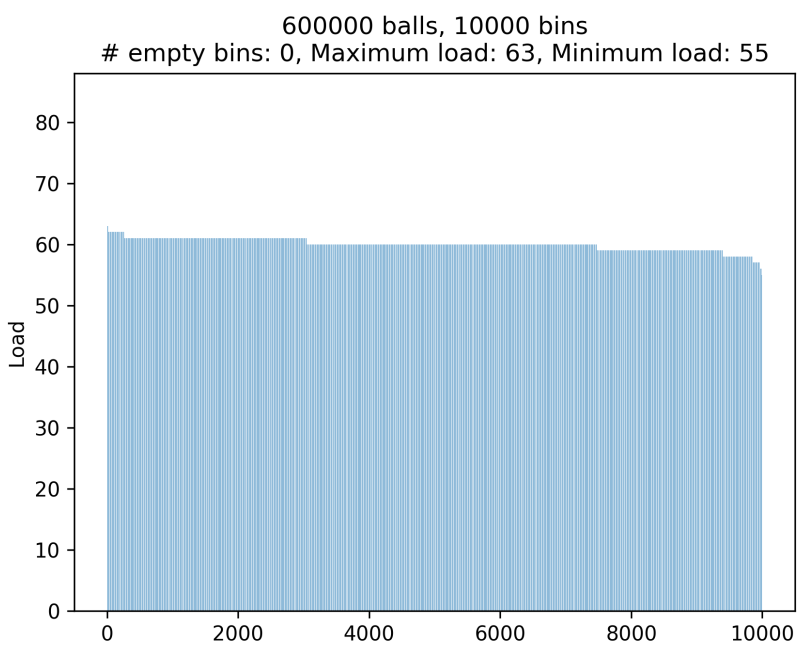
The better of two choices. When throwing each ball, suppose we pick *two bins* uniformly at random, then put the ball in the bin whose load is lower. Intuitively this should balance out the load on the bins even more, thereby improving the ratio between the maximum and minimum load. An example of this approach is illustrated in Figure 7.4 (b), which shows a substantial improvement. Indeed, this scheme is very practical and has been implemented as a load balancer in the NGINX web server. Research on this topic won the 2020 ACM Kanellakis award.

The mathematical analysis of this scheme is somewhat tricky, but rigorous results are known. For example, when throwing n balls into n bins, the maximum number of balls in any bin is very likely to be $\Theta(\log\log n)$. To get a sense of how large this is, note that for sufficiently large n we have

$$\underbrace{\log\log n}_{\substack{\text{asymptotic max load when}\\ \text{picking better of two choices}}} \;<\; (\ln n)^{0.01} \;<\; (\ln n)^{0.99} \;<\; \underbrace{\frac{\log n}{\log\log n}}_{\substack{\text{asymptotic max load when}\\ \text{picking a single random bin}}} \;<\; \ln n. \tag{7.5.1}$$



(a)



(b)

Figure 7.4: Examples with $m = 600,000$ balls and $10,000$ bins. The bins are sorted in decreasing order. (a) The balls are thrown into a uniformly random bin. The ratio of the maximum load to the minimum load is $88/33 \approx 2.66$. (b) Each ball picks *two* random bins and joins the one with the lower load. The ratio of the maximum load to the minimum load is significantly improved to $63/55 \approx 1.13$.

References: (Motwani and Raghavan, 1995, Theorem 3.1), (Mitzenmacher and Upfal, 2005, Chapter 14).

Problem to Ponder 7.5.1. Can you prove the inequalities in (7.5.1)?

Problem to Ponder 7.5.2. Example 7.1.2 claimed that balls and bins techniques are useful for analyzing hash tables. How would you design a hash table that stores each item in the better of two bins?

★7.6 Load on heaviest bin

Let's consider again the balls and bins scenario with n bins and n balls. We will show that, with probability close to 1, the heaviest bin has $O(\log n/\log\log n)$ balls.

Let X_1 be the number of balls in bin 1. The first step is to understand the distribution of X_1. We can think of each ball as being a trial which succeeds if the ball lands in bin 1. Since there are n bins, each trial succeeds with probability $1/n$. The random variable X_1 counts the number of successes among these n trials. Therefore X_1 has the binomial distribution $B(n, 1/n)$ (see Section A.3.3).

The main task is to analyze the probability that bin 1 gets at least k balls. First we will need the following bound on binomial coefficients, which is easy to prove from the definition.

$$\binom{n}{i} \;=\; \frac{n(n-1)\cdots(n-i+1)}{i!} \;\leq\; \frac{n^i}{i!} \qquad \forall 1 \leq i \leq n. \tag{7.6.1}$$

Using this, we can analyze this probability as follows.

$$\begin{aligned}
\Pr[\,X_1 \geq k\,] \;&\leq\; \binom{n}{k}(1/n)^k && \text{(by Fact A.3.19)}\\
&\leq\; \frac{n^k}{k!}(1/n)^k \;=\; \frac{1}{k!} && \text{(by (7.6.1))}
\end{aligned}$$

We would like this probability to be small, so we must choose k appropriately. To do so, we must pull a mathematical rabbit out of a hat to address the question: what is the inverse of the factorial function?

Fact 7.6.1 (Approximate inverse factorial). For all $x > 1$,

$$\Big(\frac{\ln x}{\ln(1+\ln x)}\Big)! \;\leq\; x \;\leq\; \Big(\frac{2\ln(x)+e}{\ln(e+\ln x)}\Big)!.$$

Figure 7.5 gives a plot of these inequalities, and a proof is given below.

Returning to our analysis of $\Pr[\,X_1 \geq k\,]$, we will choose

$$k \;=\; \frac{2\ln(n^2)+e}{\ln(e+\ln(n^2))} \;=\; O\Big(\frac{\log n}{\log\log n}\Big).$$

Using the upper bound from Fact 7.6.1 with $x = n^2$, we obtain $n^2 \leq k!$, so

$$\Pr[\,X_1 \geq k\,] \;\leq\; \frac{1}{k!} \;\leq\; \frac{1}{n^2}. \tag{7.6.2}$$

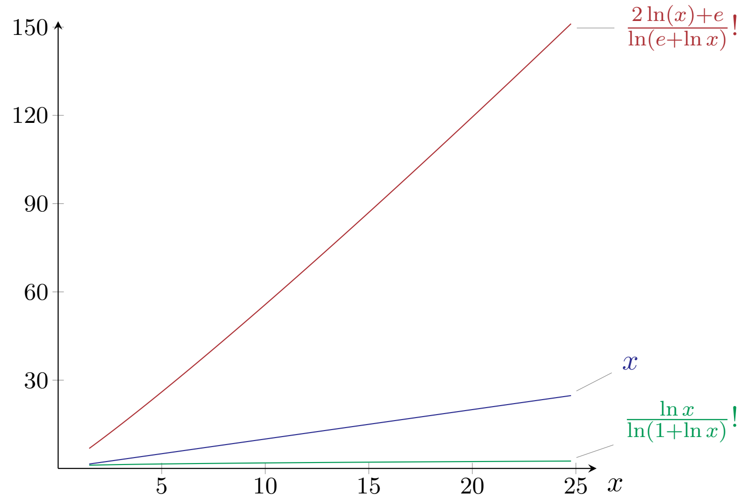


Figure 7.5: This plot illustrates the inequalities in Fact 7.6.1.

So far we have only analyzed bin 1, but the bins are all equivalent so the same analysis actually holds for all bins. The remainder of the argument is just the union bound.

$$\begin{aligned} \Pr[\,\textit{any} \text{ bin has} \geq k \text{ balls}\,] &= \Pr[\, X_1 \geq k \,\vee\, X_2 \geq k \,\vee\, \cdots \,\vee\, X_n \geq k\,] \\ &\leq \sum_{i=1}^{n} \Pr[\, X_i \geq k\,] && \text{(Fact A.3.8)} \\ &\leq \sum_{i=1}^{n} \frac{1}{n^2} && \text{(by (7.6.2))} \\ &= \frac{1}{n}. \end{aligned}$$

To summarize, we have shown that, with probability at least $1 - 1/n$, every bin has less than $k = O(\log n / \log\log n)$ balls.

References: (Mitzenmacher and Upfal, 2005, Lemma 5.1), (Motwani and Raghavan, 1995, Theorem 3.1), (Kleinberg and Tardos, 2006, (13.45)).

Proof of Fact 7.6.1.

Intuition: To approximately solve $a! = x$, we have $a! = a(a-1)(a-2)\cdots \approx a^a$. Taking the log, it suffices to solve $a \ln a = \ln x$. If we define $a = \ln(x)/\ln\ln x$, then we have

$$a \ln a \;=\; \frac{\ln x}{\cancel{\ln\ln x}}\Big(\cancel{\ln\ln x} - \underbrace{\ln\ln\ln x}_{\text{negligible}}\Big) \;\approx\; \ln x.$$

Thus $a! = \big(\ln(x)/\ln(\ln x)\big)!$ should be roughly x.

Upper bound: The aim is to find a as small as possible satisfying $a! \geq x$, or equivalently $\ln(a!) \geq \ln x$. Stirling's approximation implies that $\ln(a!) > a\ln(a/e)$ whenever $a \geq 0$. Substituting $b = a/e$ and $c = (\ln x)/e$, it suffices to satisfy $(be)\ln b \geq \ln x$, or $b \ln b \geq c$. If $x \geq 1$ then equality holds by taking $\ln b = W_0(c)$, where W_0 is the principal branch of the Lambert W function. The following upper bound holds whenever $x \geq 1/e$.

$$W_0(c) \;\leq\; \ln\Big(\frac{2c+1}{\ln(ec+e)}\Big)$$

Since we wanted $\ln b \geq W_0(c)$, it suffices to take

$$b \;=\; \frac{2c+1}{\ln(e+ec)} \;=\; \frac{\frac{2}{e}\ln(x)+1}{\ln(e+\ln x)}.$$

Recalling that $a = eb$, we have $a! \geq x$ for

$$a \;=\; \frac{2\ln(x)+e}{\ln(e+\ln x)}. \qquad \square$$

7.7 Exercises

Exercise 7.1. Suppose we throw m balls independently and uniformly into n bins. Assume that m is chosen so that the expected number of collisions is at most $1/2$. Prove that this same choice of m ensures that the probability of having no collisions is at least $1/2$.

Exercise 7.2. Suppose you throw n balls independently into n bins. Prove that the probability that bin 1 is non-empty is at least $1 - 1/e$.

Exercise 7.3. Suppose we throw m balls independently and uniformly into n bins. If $m \geq n \ln(n/\delta)$, prove that the

$$\Pr[\,\text{any bin is empty}\,] \;\leq\; \delta.$$

Exercise 7.4. Let $k \geq 1$. Suppose we throw m balls uniformly and independently into $n = m^2 k$ bins. Use the union bound to prove that the probability of any collisions is at most $1/2k$.

Exercise 7.5 The half coupon collector. Suppose that there are n different types of toy and each box of cereal contains a random toy. You plan to buy exactly m boxes of cereal. Fortunately, you are happy to only collect *half* of the coupons.

Part I. Show that if $m = 100$ and $n = 144$ then

$$\Pr[\,\text{collect toy } i\,] \;\geq\; 1/2 \qquad \forall i \in [n]. \tag{7.7.1}$$

Part II. Show that if $m = 100$ and $n = 145$ then

$$\Pr[\,\text{collect toy } i\,] \;\leq\; 1/2 \qquad \forall i \in [n].$$

Part III. Suppose that you buy m boxes of cereal. Find a large value of n (as a function of m) that ensures that (7.7.1) holds.

Exercise 7.6. Let $X_1, \ldots, X_n$ be independent random values that are uniform on $[0, 1]$. Prove that there is zero probability that there exist distinct indices i, j with $X_i = X_j$.

Hint: See Fact A.3.16.

Exercise 7.7. Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be independent events.

Part I. Suppose that $\Pr[\,\mathcal{E}_i\,] = \frac{1}{10n}$ for every i. Use the union bound to give an upper bound on $\Pr[\,\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n\,]$.

Part II. Give an exact formula for $\Pr[\,\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n\,]$.

Part III. Using a calculator, numerically compare the answers from the previous parts when $n = 100$.

Part IV. Now suppose that $\Pr[\,\mathcal{E}_i\,] = \frac{1}{100}$ for every i, and $n = 100$. Again, numerically compare the union bound and the exact value of $\Pr[\,\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n\,]$.

Exercise 7.8. Suppose we throw n balls into n bins uniformly and independently at random. Let Y be the load on the heaviest bin. Prove that $\mathrm{E}[Y] = O(\log n / \log\log n)$.

Exercise 7.9. Suppose there are m items from which you generate a sample, using sampling with replacement.

Part I. If you sample k items then what is the expected number of *distinct* items?

Part II. If you sample m items, show that the expected number of distinct items is at least $(1 - 1/e)m$.

Exercise 7.10. Let $X[1..n]$ be an array of real numbers, chosen independently and uniformly from the interval $[0, 1]$. Describe and analyze an algorithm to sort X with expected runtime $O(n)$.

Exercise 7.11 Amplifying small probabilities. Let $\mathcal{B}$ be a randomized algorithm that runs in $O(n^{1.5})$ time on inputs of size n. It has the following outcome probabilities.

Correct Output \ Algorithm's Output	Yes	No
Yes	$\geq 1/n$ true positive	$\leq 1 - 1/n$ false negative
No	0 false positive	1 true negative

Design a randomized algorithm $\mathcal{A}$ that runs in $o(n^3)$ time and has the following outcome probabilities.

Correct Output \ Algorithm's Output	Yes	No
Yes	$\geq 1 - 1/n$ true positive	$\leq 1/n$ false negative
No	0 false positive	1 true negative

Prove that your algorithm works.

Exercise 7.12. In Section 3.1 we have seen several algorithms for generating uniformly random permutations. Let us consider the algorithm from a 1961 paper of Rao. It is also described in Fisher & Yates, 6th edition, 1963, page 37.

Algorithm 7.1 Rao's method to generate a uniformly random n-permutation of C.

```
function RaoPerm(list C[1..n])
    Let L[1..n] be an array of empty lists                ▷ Here n = |C|.
    foreach c ∈ C
        Let X be a RV that is uniform on [n]
        Add c to L[X]
    for i = 1, ..., n
        if |L[i]| > 1 then
            L[i] ← RaoPerm(L[i])
    return the concatenation L[1], L[2], ..., L[n]
end function
```

Part I. For any item $j \in C$, let Y_j be the number of levels of recursion that item j is involved in. Prove that $\mathrm{E}[\,Y_j\,] \leq 5$.

Hint: At each level of recursion, consider the probability that item j is the only item in its list.

Part II. Prove that the expected runtime is $O(n)$.

Exercise 7.13. Let n be an even integer. First we define a set of vectors for which all coordinates are distinct.

$$D \;=\; \{\; x \in [n]^n \;:\; \text{all coordinates of } x \text{ are distinct} \;\}$$

Next we define a set of vectors for which each coordinate is equal to exactly one other coordinate.

$$P \;=\; \{\; x \in [n]^n \;:\; \forall i \in [n],\; \exists \text{ unique } j \in [n] \text{ with } i \neq j \text{ and } x_i = x_j \;\}.$$

Notice that the sets D and P are disjoint. Define the function $f : D \cup P \to \{0,1\}$ by

$$f(x) \;=\; \begin{cases} 0 & (\text{if } x \in D) \\ 1 & (\text{if } x \in P). \end{cases}$$

Design an RP-type algorithm for computing f. Given an input vector x, it should examine only $O(\sqrt{n})$ coordinates of x.

Part II

Concentration

Chapter 8

Markov's Inequality

8.1 Relating expectations and probabilities

Often we would like to show that a random variable is, in some sense, small. This statement could be interpreted in several ways. The two most common meanings are

(a) the random variable is small in expectation, or

(b) the random variable is unlikely to be large.

We have already seen both of these meanings being used in previous chapters. For example, Section 7.2 considered balls colliding in bins, and distinguished between two questions.

- How many balls can we throw such that the expected number of collisions is less than 1?
- How many balls can we throw such that the probability of having any collisions is at most 1/2?

Another example can be found in Section 5.1.1. When analyzing the number of iterations Y performed by CategoricalSamplerUsingTrie, the equation (5.1.3) gave a relationship between the expectation of Y and the probability that Y is large.

One might wonder if there are always relationships between interpretations (a) and (b).

(a) $\overset{?}{\Rightarrow}$ (b): If a random variable is small in expectation, is it unlikely to be large?

(b) $\overset{?}{\Rightarrow}$ (a): If a random variable is unlikely to be large, must it be small in expectation?

Unfortunately, neither of these implications is true in general, as shown in the following section. However all is not lost. The main result of this chapter is Markov's inequality, which shows that (a) $\Rightarrow$ (b) under a mild restriction.

8.2 Some examples

8.2.1 (a) $\not\Rightarrow$ (b): An unscrupulous deal

Suppose I offer to sell you a phone at a price of \$$Y$, where Y is a random variable with $\mathrm{E}[\,Y\,] = 10$. Knowing only this expectation, you must either commit to buy the phone or not. After you have committed, we will then sample the random variable Y to determine the actual price.

If Y were *always* equal to \$10, this would be a very good deal! However, being unscrupulous, I have chosen the distribution of Y as follows.

$$\Pr[Y = 10000] = 0.99 \qquad \text{and} \qquad \Pr[Y = -989000] = 0.01$$

One may easily verify that this distribution satisfies $\mathrm{E}[Y] = 10$. To clarify our understanding of Y, let us consider the two outcomes of this distribution.

- You must pay \$10000 for the phone. This is very likely — it happens with probability 0.99.
- You receive the phone and also receive \$989000. Although very profitable, this outcome is very unlikely — it happens with probability 0.01.

Depending on your risk tolerance, you might now be having second thoughts about whether you should accept the deal. We summarize this discussion as follows.

Moral 8.2.1. The expectation of a random variable might not suffice to judge whether it is good.

In particular, we have described a random variable Y that is small in expectation, but yet is very likely to take large positive values. Thus Y gives a counterexample to the possible implication (a) $\overset{?}{\Rightarrow}$ (b) that was discussed in Section 8.1.

However, a crucial aspect of the definition is that Y is also allowed to take extremely negative values. The main point of this chapter is to consider whether the possible implication (a) $\overset{?}{\Rightarrow}$ (b) might hold in the context of *non-negative* random variables.

Problem to Ponder 8.2.2. For any $\epsilon > 0$ and $k > 10$, design a distribution Y with only two outcomes such that $\Pr[Y = k] = 1 - \epsilon$ and $\mathrm{E}[Y] = 10$.

8.2.2 (b) $\not\Rightarrow$ (a): Infinite expectation

Let us consider an interesting random variable Z that often challenges our intuition about probability. The definition is as follows.

$$\Pr[Z = i] \;=\; \frac{6}{\pi^2 i^2} \qquad \forall i \geq 1.$$

To see that this random variable is well-defined, we verify that

$$\sum_{i\geq 1} \Pr[Z = i] \;=\; \frac{6}{\pi^2} \sum_{i \geq 1} \frac{1}{i^2} \;=\; 1,$$

which holds because of the mathematical identity $\sum_{i\geq 1} \frac{1}{i^2} = \pi^2/6$. This identity originates from Euler's solution to the "Basel problem" in the year 1735.

References: Wikipedia.

The curious property of this random variable is that, although it takes only finite values, its expectation is infinite.

Claim 8.2.3. $\mathrm{E}[Z] = \infty$.

Proof. From the definition of expectation, we have

$$\mathrm{E}[Z] \;=\; \sum_{i\geq 1} i \cdot \Pr[Z = i] \;=\; \sum_{i \geq 1} i \cdot \frac{6}{\pi^2 i^2} \;=\; \frac{6}{\pi^2} \sum_{i\geq 1} \frac{1}{i}.$$

This last sum turns out to be infinite. To see this, for any $n \geq 1$, we can truncate it to a harmonic sum of length n, then use Fact A.2.1. That argument shows that

$$\mathrm{E}[\, Z \,] \;\geq\; \frac{6}{\pi^2}\sum_{i=1}^{n}\frac{1}{i} \;\geq\; \frac{6}{\pi^2}\ln(n).$$

Since $\ln(n)$ can be arbitrarily large, the claim is proven. $\square$

Problem to Ponder 8.2.4. Prove that $\Pr[\, Z \geq k \,] \leq 2/k$ for all $k \geq 1$.

From the inequality $\Pr[\, Z \geq k \,] \leq 2/k$ we see that, in a certain sense, Z is unlikely to take large values. However, we have shown that $\mathrm{E}[\, Z \,] = \infty$. Thus Z gives a counterexample to the possible implication (b) $\overset{?}{\Rightarrow}$ (a) that was discussed in Section 8.1.

Problem to Ponder 8.2.5. Define a random variable $\tilde{Z}$ that equals Z with probability $1/2$ and equals $-Z$ with probability $1/2$. What can you say about $\mathrm{E}\left[\, \tilde{Z} \,\right]$?

References: (Grimmett and Stirzaker, 2001, Example 3.3.12).

8.2.3 Ludicrous load balancing

For our third example, we consider the load balancing scenario of Section 7.3 with n clients and n servers. Let X_i be the load on the i^{th} server. By a decomposition into indicators and Fact A.3.14, we have

$$\mathrm{E}[\, X_i \,] \;=\; \sum_{j=1}^{n} \Pr[\, \text{client } j \text{ assigned to server } i \,] \;=\; \sum_{j=1}^{n}\frac{1}{n} \;=\; 1.$$

Observe that this analysis does *not* rely on independence of the assignment for different clients. The reason is that we have only used linearity of expectation. The same analysis works even if all clients are heavily dependent.

For example, we could consider the following ludicrous load balancing scheme: pick a server S uniformly at random, then assign *all* clients to server S. For every server i, it remains true that the expected load on server i is $\mathrm{E}[\, X_i \,] = 1$, because (8.2.3) does not rely on independence. Nevertheless the ludicrous scheme feels much less balanced than the independent scheme. We can provide justification for this feeling by considering the probability that X_i is large. With the ludicrous scheme we have

$$\Pr[\, X_i = n \,] \;=\; \Pr[\, \text{chosen server } S \text{ is server } i \,] \;=\; \frac{1}{n}. \tag{8.2.1}$$

In contrast, with the independent scheme we have

$$\Pr[\, X_i = n \,] \;=\; \prod_{j=1}^{n} \Pr[\, \text{client } j \text{ chooses server } i \,] \;=\; \frac{1}{n^n}.$$

Thus, we can summarize our understanding of these two schemes as follows.

Independent load balancing.	$\mathrm{E}[\, X_i \,] = 1 \;\; \forall i$	$\Pr[\, X_i = n \,] = n^{-n} \;\; \forall i$
Ludicrous load balancing.	$\mathrm{E}[\, X_i \,] = 1 \;\; \forall i$	$\Pr[\, X_i = n \,] = n^{-1} \;\; \forall i$

To summarize, these two load balancing schemes are identical in terms of the expected load on each server, but yet they are very different in terms of the probability that the load equals n. Thus, these load balancing schemes provide another illustration of the moral that we stated earlier.

Moral 8.2.1. The expectation of a random variable might not suffice to judge whether it is good.

Question 8.2.6. Compare the maximum load on any server under the independent and ludicrous schemes.

Answer.

In the independent scheme we saw in Section 7.6 that maximum load is probably $O(\log n / \log\log n)$. In contrast, with the ludicrous scheme the maximum load is always exactly equal to n.

Problem to Ponder 8.2.7. Is there a dependent load balancing strategy such that each client is assigned to each server with probability $1/n$, but the expected maximum load is $O(\log^{0.99} n)$?

8.2.4 When does expectation tell us about probabilities?

Above we considered the question of whether a random variable with small expectation must be unlikely to take large values. We wrote this as the possible implication (a) $\stackrel{?}{\Rightarrow}$ (b).

The price $\$Y$ of the phone in Section 8.2.1 showed that this implication is false in general. Roughly speaking, this example had

$$\textit{Price of phone:} \qquad \mathrm{E}\left[\,Y\,\right] \text{ is small} \qquad \text{but} \qquad \Pr\left[\,Y \text{ large}\,\right] \;=\; \underbrace{0.99}_{\text{close to 1}}\;.$$

Thus the expectation of Y did not reveal much about the probability that Y is large.

In contrast, the load balancing example of Section 8.2.3 was somewhat different. Let us consider the load on server 1, say. Then, roughly speaking, this example had

$$\textit{Load balancing:} \qquad \mathrm{E}\left[\,X_1\,\right] \text{ is small} \qquad \text{but} \qquad \Pr\left[\,X_1 \text{ large}\,\right] \text{ is } \underbrace{\text{either } n^{-1} \text{ or } n^{-n}}_{\text{close to 0}},$$

depending on which load balancing scheme was used. Thus the expectation of X_1 did not reveal the exact probability that X_1 is large.

There is an important qualitative difference between the phone example and the load balancing example. In the phone example, $\Pr\left[\,Y \text{ large}\,\right]$ is close to 1, whereas with load balancing, $\Pr\left[\,X_1 \text{ large}\,\right]$ is close to 0 (regardless of which load balancing scheme is used). Thus, the phone example is a counterexample to (a) $\stackrel{?}{\Rightarrow}$ (b), whereas the load balancing example is not.

Why do these examples have this qualitative difference? There is an important underlying reason. The key property of the load X_1 is that it is a *non-negative* random variable, whereas the price Y of the phone can take negative values.

In the next section we will exploit this property in developing Markov's inequality, which shows that non-negative random variables with small expectation must be unlikely to take large values. That is, (a) $\Rightarrow$ (b) does indeed hold for non-negative random variables.

8.3 Markov's inequality

8.3.1 Intuition

Suppose that Y is a non-negative, integer-valued RV, and we know that $p = \Pr\left[\,Y \geq 100\,\right]$. What is the smallest possible value that $\mathrm{E}\left[\,Y\,\right]$ can take?

The RV Y must have p units of probability mass on the integers $\{100, 101, \ldots\}$, and $1-p$ units of probability mass on the integers $\{0, 1, \ldots, 99\}$. To make $\mathrm{E}[Y]$ as small as possible, it is clear that we should move all the probability mass as far as possible to the left. This is accomplished by putting p units on 100 and $1-p$ units on 0. So the smallest possible value for $\mathrm{E}[Y]$ is $(1-p)\cdot 0 + p \cdot 100 = 100p$. We may write this as

$$\mathrm{E}[Y] \;\geq\; 100p.$$

By rearranging, we obtain the equivalent statement

$$\Pr[Y \geq 100] \;\leq\; \frac{\mathrm{E}[Y]}{100}.$$

This argument captures the main ideas of Markov's inequality.

8.3.2 The formal statement

Fact A.3.23 (Markov's Inequality). Let Y be a random variable that only takes non-negative values. Then, for all $a > 0$,

$$\Pr[Y \geq a] \;\leq\; \frac{\mathrm{E}[Y]}{a}.$$

Note that if $a \leq \mathrm{E}[Y]$ then the right-hand side of the inequality is 1 or larger. This is not a useful bound, because the probability of any event is always 1 or smaller. Thus, Markov's inequality is only useful when $a > \mathrm{E}[Y]$.

Often when we use Markov's inequality we want the right-hand side to be some constant, like $1/b$. We can rewrite Markov's inequality in this useful form, simply by taking $a = b \cdot \mathrm{E}[Y]$.

Fact A.3.24. Let Y be a random variable that only takes nonnegative values. Then, for all $b > 0$,

$$\Pr[Y \geq b \cdot \mathrm{E}[Y]] \;\leq\; \frac{1}{b}.$$

Markov's inequality is an example of what we call a ***tail bound***. More specifically, it is a "right tail bound", which means that it gives an upper bound on the probability that Y is somewhat larger than its mean. There is the analogous notion of a "left tail bound", which is an upper bound on probability that Y is somewhat smaller than its mean. However Markov's inequality only provides a right tail bound, not a left tail bound.

In the analysis of randomized algorithms, it is very common to use tail bounds to prove that certain undesired events are unlikely to happen. Markov's inequality, while simple, can be quite a useful tool for this purpose. In Chapter 9 we will present more powerful tools that provide both left and right tail bounds in certain common scenarios.

Question 8.3.1. Explain the connection between Exercise 4.1 and Markov's inequality.

Answer.

Let X be a random variable that takes only non-negative integer values. Exercise 4.1 states that

$$\Pr[X \geq 1] \;\leq\; \mathrm{E}[X],$$

whereas Markov's inequality is the more general statement that

$$\Pr[X \geq a] \;\leq\; \frac{\mathrm{E}[X]}{a} \qquad \forall a > 0.$$

8.3.3 The proof

Proof of Fact A.3.23. By definition of expectation,

$$\mathrm{E}[Y] = \sum_{0 \le i < a} i \cdot \Pr[Y = i] + \sum_{i \ge a} i \cdot \Pr[Y = i].$$

Above we used the idea of "moving all probability mass to the left", which is accomplished by this lower bound

$$\begin{aligned} &\ge \sum_{0 \le i < a} 0 \cdot \Pr[Y = i] + \sum_{i \ge a} a \cdot \Pr[Y = i] \\ &= a \cdot \Pr[Y \ge a]. \end{aligned}$$

Rearranging, we get

$$\Pr[Y \ge a] \le \frac{\mathrm{E}[Y]}{a},$$

which is the claimed result. □

Question 8.3.2. The hypothesis of Markov's inequality is that Y takes only non-negative values. However, the proof appears to assume that Y takes non-negative *integer* values. Is that really the case? Does the proof work for any non-negative, discrete random variable Y?

Problem to Ponder 8.3.3. Prove Markov's inequality using the Law of Total Expectation (Fact A.3.15).

Problem to Ponder 8.3.4. Markov's inequality generalizes to *all* non-negative random variables, whether integer-valued or not. Book 2, Fact B.4.1 states that, for any non-negative RV X,

$$\mathrm{E}[X] = \int_0^\infty \Pr[X \ge x]\, dx.$$

Can you use this fact to prove the general form of Markov's inequality?

8.4 Example applications of Markov's inequality

8.4.1 QuickSort

Let's look at an example of where the Markov inequality is useful. Recall the Randomized QuickSort algorithm from Section 4.4. We proved that its expected number of comparisons is less than $2n \ln n$. Could the number of comparisons actually be much worse? How likely is it that the algorithm performs significantly more than $2n \ln n$ comparisons?

Let Y be the RV giving the number of comparisons. This is a non-negative RV, so we can apply the Markov inequality. Applying Fact A.3.23, we get

$$\Pr[\text{number of comparisons} \ge 200n \ln n] = \Pr[Y \ge 200n \ln n] \le \frac{\mathrm{E}[Y]}{200n \ln n} \le \frac{2n \ln n}{200n \ln n} = 0.01.$$

This shows that there is only a 1% probability that the number of comparisons made by the algorithm is $200n \ln(n)$ or greater.

It is worth being aware of a subtle detail that arises in this example. Recall that Markov's inequality bounds the probability that random variable is somewhat larger than its expectation. However, it is not necessary to know the exact value of the expectation — an upper on the expectation is also sufficient. In this example, we did not determine the exact value of $\mathrm{E}[Y]$, but our upper bound of $2n \ln n$ was sufficient to use Markov's inequality.

8.4.2 Tail bounds for load balancing

Let us revisit the load balancing scenario of Section 8.2.3, and recall that X_i denotes the load on the i^{th} server. Suppose that some randomized load balancing scheme is being used, but that we know nothing about it except that $\mathrm{E}[X_i] = 1$ for every i. Since X_i is a non-negative random variable, we can use Markov's inequality to prove an upper bound on the probability that a machine has a large load. For example, we find that there is only a 1% probability that the load is 100 or higher:

$$\Pr[X_1 \geq 100] \;\leq\; \frac{\mathrm{E}[X_1]}{100} \;=\; 0.01.$$

Next, let us consider the probability that *all* clients end up on the first server. Again using Markov's inequality, we obtain

$$\Pr[X_1 \geq n] \;\leq\; \frac{\mathrm{E}[X_1]}{n} \;=\; \frac{1}{n}.$$

This upper bound is actually tight if the ludicrous load balancing scheme is being used: recall from (8.2.1) that $\Pr[X_1 \geq n]$ is exactly equal to $1/n$. Thus, the ludicrous load balancing scheme is actually the "worst possible" load balancing scheme (under the assumption that the expected load on each server is 1). By this we mean that the ludicrous scheme has the largest possible probability that the load equals the maximum value.

Of course, Markov's inequality is not always tight, not even for the ludicrous load balancing scheme. For example, Markov's inequality implies that

$$\Pr[X_1 \geq n+1] \;\leq\; \frac{\mathrm{E}[X_1]}{n+1} \;=\; \frac{1}{n+1}.$$

However this cannot possibly be tight because the load balancing scenario has only n clients, so actually $\Pr[X_1 \geq n+1] = 0$.

When using Markov's inequality, or other tail bounds that we encounter later on, we should keep in mind that sometimes their guarantees might be very tight, and sometimes they might be quite loose. There is a certain art in knowing how to apply these techniques to give useful bounds, and we hope to convey some of this art over the course of the book.

8.5 Exercises

Exercise 8.1. Let $k \geq 1$. Suppose we throw m balls uniformly and independently into $n = m^2 k$ bins. Use Markov's inequality to prove that the probability of any collisions is at most $1/2k$.

Exercise 8.2 Tightness of Markov's inequality. In Section 9.1 we will show that Markov's inequality can be very loose, whereas in this exercise we show that it can be tight. For every $a > 1$, find a non-negative RV with $\Pr[Y \geq a] = \mathrm{E}[Y]/a$.

Exercise 8.3. Let $\mathcal{B}$ be an RP-type algorithm whose *expected* runtime is $O(n^2)$. Show how to convert $\mathcal{B}$ into an RP-type algorithm that *always* has runtime $O(n^2)$.

Exercise 8.4. Let A be an unsorted array of n distinct numbers, where n is a multiple of 4. Let us say that an element is "in the middle half" if there are at least $n/4$ elements less than it, and at least $n/4$ elements greater than it. Describe a randomized algorithm that runs in $O(n)$ time and that either outputs an element of A in the middle half, or outputs "fail". It should fail with probability at most 0.01.

Exercise 8.5. This exercise is a variant of Exercise 7.6.

Let U be a positive integer. Let $X_1, \ldots, X_n$ be independent random values that are uniform on the finite set $[U]$. Choose a value of U, as small as you can, and show that

$$\Pr[\text{there exist distinct indices } i, j \text{ with } X_i = X_j] \;\leq\; 1/100.$$

Exercise 8.6. Let H be the number of non-empty levels in a Skip List with n nodes. Use Markov's inequality to prove that

$$\Pr[H \geq 100 \lg n] \leq 1/30,$$

assuming that n is sufficiently large. Compare this to the results of Exercise 6.3.

Exercise 8.7 Reverse Markov inequality. Let Z be a random variable that is never larger than some fixed number d. Prove that, for all $c < d$,

$$\Pr[Z \leq c] \;\leq\; \frac{d - \mathrm{E}[Z]}{d - c}.$$

References: (Grimmett and Stirzaker, 2001, Theorem 7.3.5), (Lehman et al., 2018, Section 20.1.2).

Chapter 9

Concentration Bounds

9.1 Introduction to concentration

Repeating a trial multiple times is a concept that we are all undoubtedly familiar with. This idea can be found in the daily news during discussions of political polls and pharmaceutical trials. It also frequently arises in randomized algorithms, as we saw in the discussion of amplification (Section 1.3), for example. In this chapter we present mathematical tools to help us analyze the outcomes of repeated trials.

For example, suppose that X is a random variable giving the number of successful trials. As discussed in Chapter 8, it is very common to prove ***tail bounds*** on X. For example, if $a < \mathrm{E}[X] < b$, we would like to have a

- *Left tail bound:* an upper bound on $\Pr[X \leq a]$, or a

- *Right tail bound:* an upper bound on $\Pr[X \geq b]$.

Markov's inequality is a first step in this direction; however it provides only a right tail bound, and moreover it is often quite weak. The purpose of this chapter is to provide left and right tail bounds that are much stronger than Markov's inequality, and can be applied in many common scenarios. If we can show that $\Pr[X \leq a]$ and $\Pr[X \geq b]$ are both small, where a and b are quite close, then we say that X is ***concentrated around its expectation***, which is often very useful. The topic of concentration is touched upon in introductory courses in probability and statistics. Algorithm researchers use many standard techniques, of which we will compare several through discussing the following example.

Example 9.1.1. Suppose we repeatedly toss a fair coin n times. Let X be the RV giving the number of tosses that are heads. This has the binomial distribution $B(n, 1/2)$, so of course its expectation is $\mathrm{E}[X] = n/2$, by (A.3.5). What can we say about the probability that X is far from its expectation?

In order to compare the various techniques below, we must make this question more explicit. Since we want to include Markov's inequality in our comparison, we focus on the right tail, i.e., the probability that X is much larger than its expectation. To nail down what "much larger" means we will, for the sake of example, consider the probability that $X \geq 3n/4$.

Markov's inequality. In our example X is the number of heads observed, so it is a non-negative random variable. The hypothesis of Markov's inequality is satisified, so Fact A.3.23 implies that

$$\Pr\left[X \geq \frac{3n}{4}\right] \leq \frac{\mathrm{E}[X]}{3n/4} = \frac{n/2}{3n/4} = \frac{2}{3}.$$

This bound does not depend on n at all, which makes it feel very weak. For example, our intuition tells us that in 100 fair coin flips, the probability of seeing at least 75 heads should be much less than $2/3$.

Law of large numbers. This is a fundamental concept that is surely familiar both from daily life and from introductory probability courses. It says that the average of many independent samples of a random variable equals the expectation in the limit as the number of samples increases. In our example, it tells us that X/n converges to $1/2$ as $n \to \infty$. However, it unfortunately tells us nothing at all about what happens for any finite n. Thus, when we toss $n = 100$ coins, the law of large numbers provides no information about the tails of X.

References: (Anderson et al., 2017, Theorems 9.9 and 9.20), (Harchol-Balter, 2023, Exercise 18.13 and Theorem 25.15), (Lehman et al., 2018, Corollary 20.4.2), Wikipedia.

Chebyshev's inequality. Another result called ***Chebyshev's inequality*** is also often taught in introductory probability courses. This book does not discuss it in detail because it is somewhat more cumbersome than other inequalities. Omitting many details, a tail bound that one can obtain via this approach is $\Pr[X \geq 3n/4] \leq 4/n$. This is much better than the bound that we obtained via Markov's inequality because, at the very least, it tends to zero as n increases. However, as we will see below, tending to zero is often not good enough — it is also important to understand the tail bound's asymptotic behaviour as a function of n. We will see that this $4/n$ bound is asymptotically not very good. The bounds we prove below tend to zero at a much faster rate.

References: (Anderson et al., 2017, Theorem 9.5), (Lehman et al., 2018, Theorem 20.2.3), (Harchol-Balter, 2023, Section 18.2), (Motwani and Raghavan, 1995, Theorem 3.3), (Mitzenmacher and Upfal, 2005, Theorem 3.6), Wikipedia.

Central limit theorem. This is, as its name suggests, a central result of introductory probability theory. It says that the distribution of X can be translated and scaled so that, as $n \to \infty$, it increasingly looks like a standard ***normal distribution***. To be exact, the distribution of

$$\frac{X - n/2}{\sqrt{n/4}}$$

converges to a normal distribution. This allows us to *approximate* (but not rigorously bound) the tail $\Pr[X \geq 3n/4]$ using the tails of the normal distribution.

References: (Anderson et al., 2017, Sections 4.1 and 9.11), (Harchol-Balter, 2023, Section 9.4).

On the plus side, this approach does provide some information about the tails of X. On the negative side, the tails of the normal distribution have a cumbersome formula for which students are not usually taught asymptotic bounds. Furthermore, although this approximation is exact in the limit as $n \to \infty$, it tells us nothing concrete for finite n. Statistics textbooks usually introduce rules of thumb to decide whether the approximation is good enough.

> *How big should n be in order to achieve approximate Normality and an accurate standard error? Let's say that n should be at least* 60 *(other texts may differ in their advice, ranging*

from $n = 30$ to $n = 100+$, simply due to the question of how close we need to be to the specified level of confidence).

"Stats: Data and Models", De Veaux, Velleman, Bock, Vukov, Wong.

Rules of thumb aside, there are results such as the Berry-Esseen theorem that provide guarantees on the rates of convergence in the central limit theorem. However, the rate of convergence is rather slow, so the tail bounds obtained via this approach are also asymptotically not very good. The bounds we prove below tend to zero at a much faster rate.

References: Wikipedia

Exact calculation. Since X has the binomial distribution, we can use explicit formulas to calculate the tails exactly. Using (A.3.6), we get

$$\Pr\left[X \geq \frac{3n}{4}\right] = \sum_{k=3n/4}^{n} \binom{n}{k} 2^{-n}. \tag{9.1.1}$$

In particular, for $n = 100$, we have

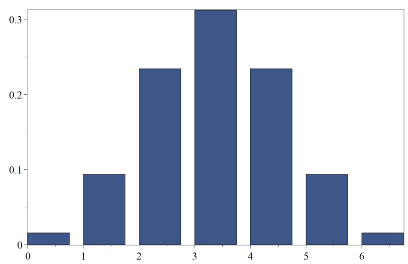
$$\Pr[X \geq 75] = \sum_{k=75}^{100} \binom{100}{k} 2^{-100} \approx .00000028.$$

This calculation confirms our earlier intuition that this probability is much less than $2/3$. However, this approach is cumbersome because mathematical software, like Wolfram Alpha, is typically needed to evaluate the formula.

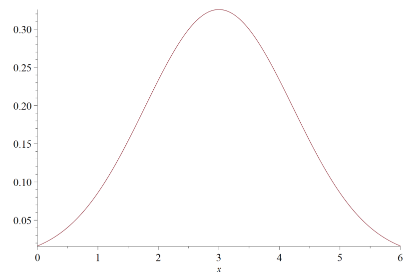
Desirable properties of a tail bound. Unfortunately the approaches described above all have some shortcomings. Ideally we would like a theorem that:

- Gives bounds on the tails of X.
- Gives results for finite n, not just results in the limit as $n \to \infty$.
- Yields tail bounds that converge to zero at an asymptotically good rate.
- Avoids cumbersome calculations with sums and binomial coefficients, like those found in the explicit formula (9.1.1).
- Is flexible enough to handle more general scenarios. For example, instead of counting heads when tossing an unbiased coin, perhaps the coin could be biased. Even more generally, perhaps the individual tosses could use different coins, each of which has its own bias.

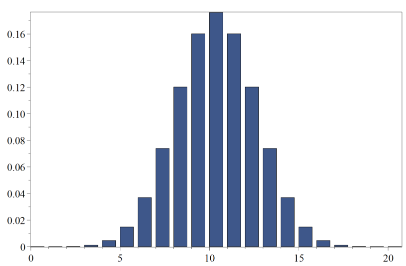
The purpose of this chapter is to introduce two new tools that do have all of these desirable properties.



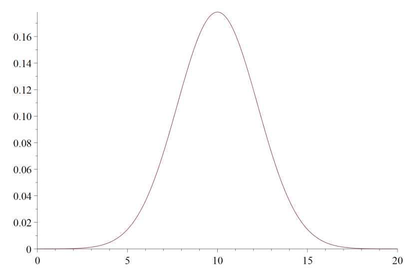
(a) $B(6, 1/2)$.



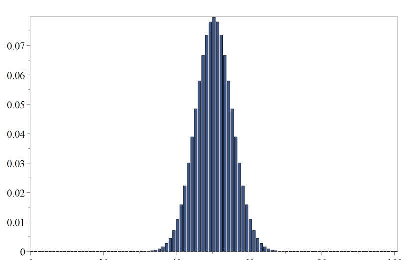
(b) $N(6/2, 6/4)$.



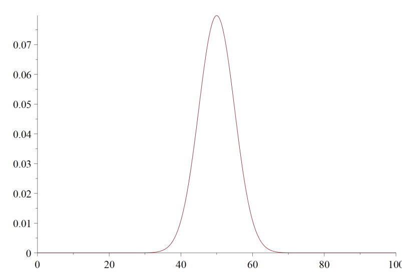
(c) $B(20, 1/2)$.



(d) $N(20/2, 20/4)$.



(e) $B(100, 1/2)$.



(f) $N(100/2, 100/4)$.

Figure 9.1: Plots of the mass function of the Binomial distribution $B(n, 1/2)$ with different values of n, compared to the density function of the Normal distribution $N(n/2, n/4)$ with mean $n/2$ and variance $n/4$. The central limit theorem implies that these plots will be similar as $n \to \infty$.

9.2 Multiplicative error: the Chernoff bound

The Chernoff bound is perhaps the most useful and powerful tool that we will encounter in this book. In its basic form, it applies to independent random variables $X_1, \ldots, X_n$ with $0 \leq X_i \leq 1$. We define

$$X = \sum_{i=1}^{n} X_i$$

$$\mu = \mathrm{E}[X], \text{ which equals } \sum_{i=1}^{n} \mathrm{E}[X_i]$$

by linearity of expectation (Fact A.3.11).

Example 9.2.1. The canonical example of this setting involves n independent trials, each of which succeeds with probability p. We let X_i be the indicator of success in the i^{th} trial. The total number of successes is $X = \sum_{i=1}^{n} X_i$, which has the binomial distribution $B(n, p)$. The expected number of successes is $\mu = np$.

Theorem 9.2.2 (Chernoff Bound). Let ϵ satisfy $0 \leq \epsilon \leq 1$. Then

$$\text{Left tail:} \qquad \Pr[X \leq (1-\epsilon)\mu] \leq e^{-\epsilon^2 \mu/3} \tag{9.2.1}$$

$$\text{Right tail:} \qquad \Pr[X \geq (1+\epsilon)\mu] \leq e^{-\epsilon^2 \mu/3} \tag{9.2.2}$$

$$\text{Combined tails:} \qquad \Pr[|X-\mu| \geq \epsilon\mu] \leq 2e^{-\epsilon^2 \mu/3}. \tag{9.2.3}$$

References: (Lehman et al., 2018, Theorem 20.5.1), (Harchol-Balter, 2023, Theorem 18.6), (Motwani and Raghavan, 1995, Section 4.1), (Mitzenmacher and Upfal, 2005, equations (4.2) and (4.5)), Wikipedia.

Book 2, Chapter 21 states a more general form of this theorem as Book 2, Theorem 21.1.1. It also contains a proof of this general theorem.

9.2.1 Coin-flipping example

As our first illustration of the Chernoff bound, let us consider the coin-flipping setting of Example 9.1.1. Recall that there are n trials; the i^{th} trial is a success if the i^{th} coin is heads, which has probability $1/2$; and we are interested in the event that there are at least $3n/4$ successes.

Using the notation of Chernoff bound, we have

$$X_i = \mathbf{1}_{\text{the } i^{\text{th}} \text{ coin is heads}}$$

Crucially, these random variables are independent, and they take values between 0 and 1.

$$\text{The total number of heads:} \quad X = \sum_{i=1}^{n} X_i$$

$$\text{The expected total number of heads:} \quad \mu = \mathrm{E}[X] = \sum_{i=1}^{n} \mathrm{E}[X_i] = n/2$$

We are interested in the event "$X \geq 3n/4$". Since $3n/4 = (1+0.5) \cdot \mu$, this is the event that X exceeds its expectation by 50%. Again, using the notation of the Chernoff bound,

$$\text{the event of interest:} \quad \text{“}X \geq 3n/4\text{”} = \text{“}X \geq (1+\epsilon)\mu\text{”}, \text{ where}$$
$$\epsilon = 0.5.$$

We analyze the probability of this event using the Chernoff bound.

$$\begin{aligned}
\Pr[\,X \geq 3n/4\,] &= \Pr[\,X \geq (1+\epsilon)\mu\,] \\
&\leq \exp(-\epsilon^2\mu/3) \qquad \text{(by Theorem 9.2.2)} \\
&= \exp\big(-(0.5)^2(n/2)/3\big) \;=\; \exp(-n/24).
\end{aligned}$$

For example, when $n = 100$ this evaluates to ≈ 0.0155. This bound is not as small as the exact answer ($\approx .00000028$), but significantly smaller than the bound obtained from Markov's inequality (2/3).

Question 9.2.3. When doing $n = 10$ fair coin flips, what is the exact probability of at most 1 heads? What estimate does the Chernoff bound give?

Answer.

Let X be the number of heads. The exact probability is

$$\Pr[\,X=0\,] + \Pr[\,X=1\,] \;=\; 2^{-10} + 10\cdot 2^{-10} \;=\; 11/1024 \;\approx\; 0.01.$$

Since $\mu = 5$, we let $\epsilon = 4/5$ and get

$$\begin{aligned}
\Pr[\,X \leq 1\,] &= \Pr[\,X \leq (1-\epsilon)\mu\,] \qquad \text{(by Theorem 9.2.2)} \\
&\leq \exp(-\epsilon^2\mu/3) \;=\; \exp(-80/75) \;<\; 0.35.
\end{aligned}$$

Problem to Ponder 9.2.4. If we did $n = 1000$ fair coin flips, what is the exact probability of at least 750 heads? What estimate does the Chernoff bound give?

9.2.2 Handling $\epsilon > 1$

The observant reader might have wondered why Theorem 9.2.2 requires that $\epsilon \leq 1$. Shouldn't we also be able to bound the probability of even larger deviations from the mean, corresponding to the scenario $\epsilon > 1$? Indeed this is possible, but now there is an important difference between the left tail and the right tail.

Question 9.2.5. Explain why the left tail bound, as stated in (9.2.1), remains true even if $\epsilon > 1$.

Answer.

If $\mu = 0$ then (9.2.1) always holds because the right-hand side is exactly 1 and the left-hand side is a probability.

If $\mu > 0$ then $(1-\epsilon)\mu < 0$ since $\epsilon > 1$. However, since X is non-negative, the probability of this event is zero, so again (9.2.1) holds.

On the other hand, the right tail (9.2.2) is *not* true in general when $\epsilon > 1$. However, a small modification can restore its validity: the ϵ^2 in the exponent must be modified to ϵ. How does this affect the tail bound? When $\epsilon > 1$, we have

$$\epsilon^2 \;\geq\; \epsilon, \qquad \text{so} \qquad \underbrace{\exp(-\epsilon^2\mu/3)}_{\text{original bound}} \;\leq\; \underbrace{\exp(-\epsilon\mu/3)}_{\text{modified bound}},$$

which means that the original bound is smaller than the modified bound. Smaller is better — however, the original bound is so small that it is now false, which is why the modification was required. Combining this modified right-tail bound with the trivial left-tail bound, we obtain the following theorem.

Theorem 9.2.6 (Chernoff for large ϵ). For any $\epsilon > 1$,

$$\begin{aligned} \text{Right tail:} \qquad & \Pr[\, X \geq (1+\epsilon)\mu \,] \;\leq\; \exp(-\epsilon\mu/3) \\ \text{Combined tails:} \qquad & \Pr[\, |X-\mu| \geq \epsilon\mu \,] \;\leq\; \exp(-\epsilon\mu/3). \end{aligned}$$

A more accurate result is presented in Book 2, Theorem 21.1.1.

9.3 Additive error: the Hoeffding bound

Recall that the Chernoff bound provides an upper bound on the probability of the following events.

$$\begin{aligned} \text{Left tail:} \qquad & X \leq (1-\epsilon)\mu \\ \text{Right tail:} \qquad & X \geq (1+\epsilon)\mu \end{aligned}$$

Note that both of the events involve a deviation from the mean by a *multiplicative* factor of ϵ. For some applications it is more convenient to consider *additive* deviations rather than multiplicative deviations. For that purpose the Hoeffding bound, which we present next, can be more convenient.

The Chernoff bound and Hoeffding bound both involve the same random variables $X_1, \ldots, X_n$, which are independent and satisfy $0 \leq X_i \leq 1$. As before, we define $X = \sum_{i=1}^n X_i$ and $\mu = \mathrm{E}[\, X \,]$.

Theorem 9.3.1 (Hoeffding Bound). For any $t \geq 0$,

$$\begin{aligned} \text{Left tail:} \qquad & \Pr[\, X \geq \mu + t \,] \;\leq\; \exp(-2t^2/n) \\ \text{Right tail:} \qquad & \Pr[\, X \leq \mu - t \,] \;\leq\; \exp(-2t^2/n). \\ \text{Combined tail:} \qquad & \Pr[\, |X-\mu| \geq t \,] \;\leq\; 2\exp(-2t^2/n). \end{aligned}$$

References: (Motwani and Raghavan, 1995, exercise 4.7(c)), (Mitzenmacher and Upfal, 2005, exercise 4.13(d)), (Grimmett and Stirzaker, 2001, Theorem 12.2.3), Wikipedia.

9.3.1 Coin-flipping example

To illustrate Hoeffding's bound, let us again consider the coin-flipping example of Example 9.1.1. Recall that there are n trials; the i^{th} trial is a success if the i^{th} coin is heads, which has probability 1/2; and we are interested in the event that there are at least $3n/4$ successes.

As in Section 9.2.1 we let X_i be the indicator of the i^{th} coin being heads, so X is the total number of coin tosses that came up heads. We are interested in the event "$X \geq 3n/4$". Using the notation of the Hoeffding bound, since $\mu = n/2$, the event of interest can be written as

$$\begin{aligned} \text{“}X \geq 3n/4\text{”} &= \text{“}X \geq \mu + t\text{”}, \text{ where} \\ t &= n/4. \end{aligned}$$

We now analyze the probability of this event using the Hoeffding bound.

$$\begin{aligned} \Pr[\, X \geq 3n/4 \,] &= \Pr[\, X \geq \mu + n/4 \,] \\ &\leq \exp\big(-2 \cdot (n/4)^2/n\big) \qquad \text{(by Theorem 9.3.1)} \\ &= \exp(-n/8). \end{aligned}$$

For example, when $n = 100$ this evaluates to ≈ 0.0000037. This is significantly closer to the exact answer ($\approx .00000028$) than the value that we obtained from the Chernoff bound in Section 9.2.1.

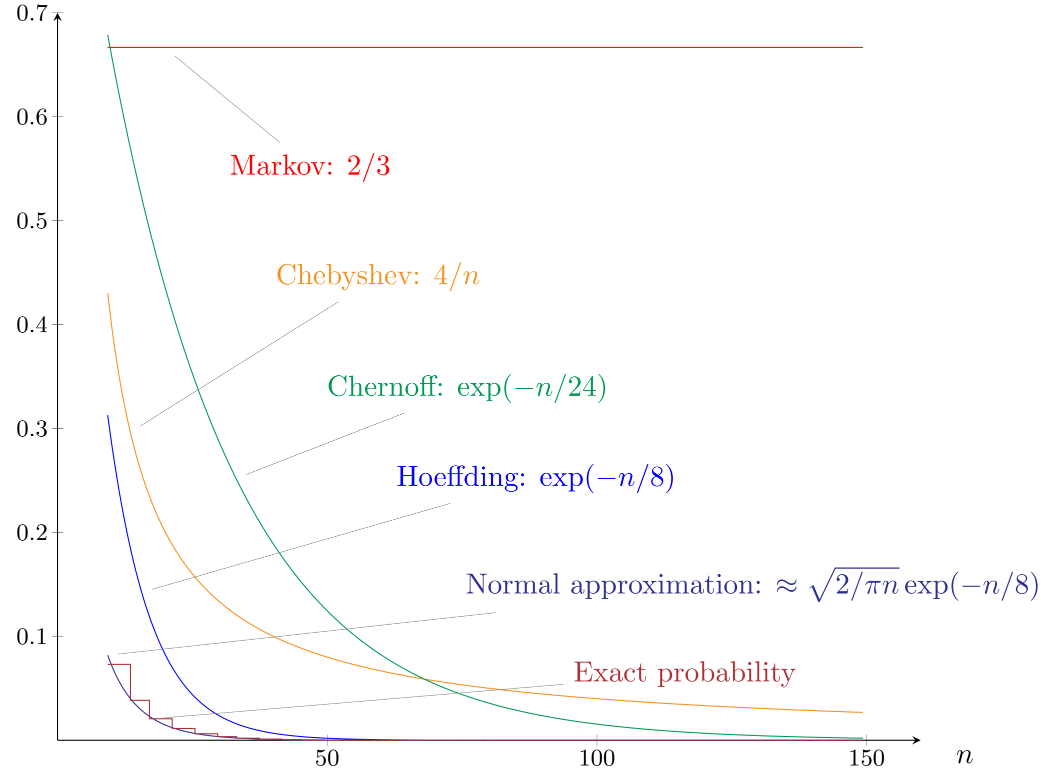


Figure 9.2: Consider the specific problem of bounding $\Pr[X \geq 3n/4]$ where X has the binomial distribution $B(n, 1/2)$. This plot shows various bounds on this probability. Note that the Chernoff bound is larger (worse) than the Chebyshev bound for $n \leq 67$, but it is smaller (better) for large n. The normal approximation is the smallest, but it is only an approximation, not a guarantee. We do not discuss the Chebyshev bound or the normal approximation in detail, but further discussion can be found in (Vershynin, 2018, Section 2.1).

9.4 Comparison of the bounds

Figure 9.2 provides a plot that illustrates the various concentration bounds that we have discussed, when applied to Example 9.1.1. Whereas the Markov bound has no decay with n, and the Chebyshev bound has linear decay with n, the Chernoff and Hoeffding bounds have exponential decay with n.

Question 9.4.1. The Chernoff and Hoeffding bounds both provide exponential upper bounds of the form

$$\Pr[X \geq 3n/4] \leq \exp(-cn)$$

for some constant c. Is it possible to prove significantly stronger bounds? For example, is it true that

$$\Pr[X \geq 3n/4] \leq \exp(-cn^2)?$$

Answer.

No. Every outcome of the coin flipping experiment has probability 2^{-n} so that is the smallest non-zero probability of any event. More formally, we can write

$$\Pr[X \geq 3n/4] \;\geq\; \Pr[X = n] \;=\; 2^{-n} \;=\; \exp\big(-\ln(2)n\big).$$

Since $\ln(2)n = o(n^2)$, we must have $\Pr[X \geq 3n/4] > \exp(-cn^2)$ when n is large enough.

9.4.1 Choosing between the Chernoff and Hoeffding bounds

As we seen above, the Chernoff and Hoeffding bounds are quite similar: they both apply to the same sort of random variables, and both provide exponentially-decreasing tail bounds. When faced with a scenario to analyze, how should one decide which of these bounds to use?

Neither the Chernoff bound nor the Hoeffding bound is definitively superior in all cases. However, depending on the application scenario, it might be the case that one of the bounds gives better results, or is more convenient to use. Here are some considerations to help choose between them.

- The Chernoff bound provides a *multiplicative* guarantee on deviations from the mean, i.e., deviations of magnitude $\epsilon\mu$. In contrast, Hoeffding is intended for an *additive* guarantee on deviations from the mean, i.e., deviations of magnitude t. Of course, these guarantees are equivalent by identifying $t = \epsilon\mu$; nevertheless, fixing parameters in this way is often quite inconvenient.

  Thus, if an application scenerio clearly demands a multiplicative guarantee or an additive guarantee, then this can often inform your choice about which bound is more appropriate.

- The Hoeffding bound is usually not useful when $t = o(\sqrt{n})$. To see why, recall that the bound on each tail is $\exp(-2t^2/n)$. Thus, if $t \leq \sqrt{n}/10$ then this bound $\exp(-2t^2/n)$ is at least

$$\exp\big(-2(n/100)/n\big) \;=\; e^{-1/50} \;>\; 0.98.$$

  Since this is very close to 1, such a bound on the tail probabilities is too large to be of much use in most application scenarios.

- The Chernoff bound might not provide good guarantees if the expected value μ is small. To see why, recall that the bound on each tail in Theorem 9.2.2 is $\exp(-\epsilon^2\mu/3)$. For example, if $\mu = 1$ and $\epsilon \leq 1$ then this bound $\exp(-\epsilon^2\mu/3)$ is at least

$$\exp(-1/3) \;>\; 0.7.$$

  Once again, such a large bound on the tail probabilities is typically not useful.

  Nevertheless, one additional idea can sometimes save the day. Recall that the Chernoff bound can also be applied in the case $\epsilon > 1$, as stated in Theorem 9.2.6. For example, if $\mu = 1$ and $\epsilon = 15$, then the tail bound is

$$\Pr[\,X \geq (1+\epsilon)\mu\,] \;\leq\; \exp(-\epsilon\mu/3) \;=\; e^{-5} \;<\; 0.01.$$

  So the Chernoff bound can give useful guarantees when the expectation μ is small, so long as we look at the probability of large deviations, i.e., we choose a value of ϵ that is much larger than 1. For example, see Exercise 10.4.

9.5 Median estimator

All of the concentration bounds that we have discussed above (Markov, Chernoff and Hoeffding) are used to show that an RV X is concentrated around its *mean*. In this section we will introduce a tool called the ***median estimator*** that shows a different sort of concentration. In brief, we will show that

> the median of a RV is likely to be close to the median of several samples of that RV.

This will be useful for several reasons. First of all, whereas the Chernoff and Hoeffding bound only apply to sums $\sum_i X_i$ with $X_i \in [0,1]$, the median estimator can be used for random variables that are not restricted to $[0,1]$. Moreover, it can even apply to random variables that are not real numbers, but instead are some other ordered objects, such as strings of text.

The setup. To begin, let us review the definition of a ***median***.

Definition 9.5.1 (Median of a list). Suppose we have numbers $Z_1, \ldots, Z_\ell$, where ℓ is odd. The definition is easiest to understand if the numbers are distinct. Suppose Z_j satisfies:

- exactly $\lfloor \ell/2 \rfloor$ of the Z_i's are less than Z_j, or equivalently,
- exactly $\lfloor \ell/2 \rfloor$ of the Z_i's are greater than Z_j.

Then we say that Z_j is the median.

If the numbers are not distinct, then the definition is slightly trickier. Suppose Z_j satisfies:

- there are at least $\ell/2$ elements that are $\leq Z_j$, and
- there are at least $\ell/2$ elements that are $\geq Z_j$.

Then we say that Z_j is the median. An alternative way to say this is as follows: if the Z_i's are ordered so that $Z_1 \leq \cdots \leq Z_n$, then the median is the value in the middle, namely $Z_{\lceil \ell/2 \rceil}$. Note that the median value is unique, although several of the Z_i's could equal that value.

In addition to defining the median of an array, we can also define the median of a random variable.

Definition 9.5.2 (Median of a random variable). Let Y be a random variable. A value η is said to be a median of Y if

- $\Pr[Y \leq \eta] \geq 1/2$, and
- $\Pr[Y \geq \eta] \geq 1/2$.

We can also think of η as a median of Y's distribution.

References: (Anderson et al., 2017, Definition 3.39).

Question 9.5.3. Is the median η of Y necessarily unique? Is it necessarily true that $\Pr[Y = \eta] > 0$?

Answer.

Both answers are no. As a simple example, let Y be a uniform random bit. For every real number $\eta \in [0,1]$, we have $\Pr[Y \leq \eta] \geq 1/2$ and $\Pr[Y \geq \eta] \geq 1/2$. This shows that there is not a unique median. Moreover for $\eta = 0.5$ we have $\Pr[Y = \eta] = 0$.

Question 9.5.4. Let Y have the binomial distribution $B(3, 1/2)$. Find a median of Y.

Answer.

Because Y has the binomial distribution, then (A.3.6) tells us that

$$\Pr[Y=0] = 1/8 \qquad \Pr[Y=1] = 3/8 \qquad \Pr[Y=2] = 3/8 \qquad \Pr[Y=3] = 1/8.$$

So any $\eta \in [1,2]$ is a median because $\Pr[Y \leq \eta] \geq 1/2$ and $\Pr[Y \geq \eta] \geq 1/2$.

The median estimator. The main setting that motivates the median estimator is as follows. Imagine that there is some random variable Z of interest whose distribution is not fully known to us. In particular, there exists a value η that is a median of Z's distribution, but η is not known to us. However, we are able to generate multiple independent *samples* of Z, which we shall call $Z_1, \ldots, Z_\ell$,

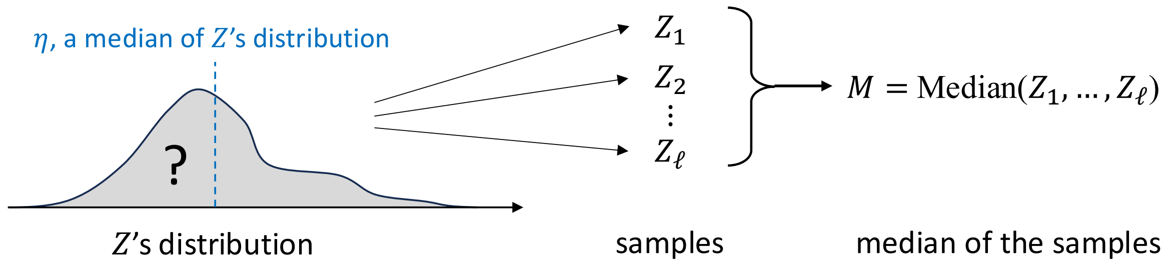


Figure 9.3: Imagine that Z is a random variable whose distribution and whose median η are unknown. If we can draw independent samples $Z_1, \ldots, Z_\ell$ of Z, then the median M of those samples can give a good estimate for η.

and which all have the same distribution as Z. Since we do know the values of the samples $Z_1, \ldots, Z_\ell$, we can determine the median M of those samples, then use M to estimate η.

The actual setting that we consider is somewhat more abstract. We assume that $Z_1, \ldots, Z_\ell$ are independent random variables, where ℓ is odd. However the Z_i's do not need to be samples of some underlying random variable Z of interest; that is, they do not need have the same distribution. Instead our assumption on the Z_i's is that, roughly speaking, their medians lie in some common interval $[L, R]$.

More precisely, we will assume that there are parameters $\alpha \geq 0$ and L, R where $L \leq R$, such that

$$\begin{aligned} \text{Left tail:} \qquad & \Pr[\, Z_i \leq L \,] \;\leq\; \frac{1}{2} - \alpha \\ \text{Right tail:} \qquad & \Pr[\, Z_i \geq R \,] \;\leq\; \frac{1}{2} - \alpha. \end{aligned} \tag{9.5.1}$$

Intuitively, L is slightly below and R is slightly above the median of each Z_i. Note that L, R and α do not depend on i.

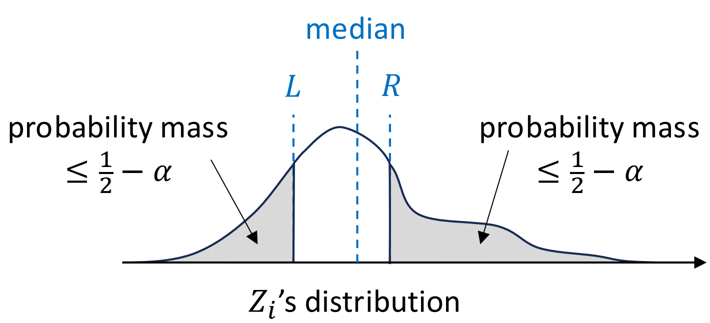


Figure 9.4: We do not assume that $Z_1, \ldots, Z_\ell$ have the same distribution, but we do assume that there are parameters $L \leq R$ and $\alpha \geq 0$ such that at most $\frac{1}{2} - \alpha$ of the probability mass is in the tail left of L, and at most $\frac{1}{2} - \alpha$ of the probability mass is in the tail right of R.

The main result is that, if we construct the samples Z_i and compute the median M of these samples, then that median value is very likely to lie between L and R.

Theorem 9.5.5. Let $Z_1, \ldots, Z_\ell$ be independent random objects that are totally ordered (for example, numbers or strings), and that satisfy (9.5.1). Let $M = \text{Median}(Z_1, \ldots, Z_\ell)$ be the random variable that equals the median of the Z_i's. Then

$$\Pr[\, M \leq L \ \vee \ M \geq R \,] \ \leq \ 2\exp(-2\alpha^2 \ell).$$

The proof of this theorem is not too difficult, so we leave it as Exercise 9.6.

Problem to Ponder 9.5.6. Earlier in the chapter we showed that RVs concentrate around their mean, whereas the present section shows that they concentrate around their median. One might be tempted to conclude that the mean of a RV is always close to the median. Is this true? *Hint:* Consider the examples of Section 8.2.

9.5.1 Application: the approximate median of an array

To illustrate the tools developed above, let us consider the problem of quickly computing an approximate median of an array.

Let A be an array of length n containing some ordered objects, such as integers or strings. In order to run certain divide-and-conquer algorithms on A, it is often useful to know the median of A. Since finding the exact median takes linear time (see Section 5.2), it might be more efficient to find instead an element that is *close* to the median, if that can be done much more quickly.

References: (Kleinberg and Tardos, 2006, Exercise 13.15)

To make this more precise, we say that an element $x \in A$ is an ***ϵ-approximate-median*** if the following two conditions hold.

- at least a $\frac{1}{2} - \epsilon$ fraction of the elements are no larger than x, i.e.,

$$|\{\, a \in A \,:\, a \leq x \,\}| \ \geq \ (\tfrac{1}{2} - \epsilon)n. \tag{9.5.2}$$

- at least a $\frac{1}{2} - \epsilon$ fraction of the elements are no smaller than x, i.e.,

$$|\{\, a \in A \,:\, a \geq x \,\}| \ \geq \ (\tfrac{1}{2} - \epsilon)n. \tag{9.5.3}$$

Such an element x must lie in the central 2ϵ fraction of the array.

Our main result is that the following algorithm is highly likely to output an ϵ-approximate-median.

Algorithm 9.1 An approximate median algorithm.

```
function ApproxMedian(set A, int ℓ)
    Select ℓ elements Z_1, ..., Z_ℓ ∈ A uniformly and independently at random
    Find the median M of Z_1, ..., Z_ℓ using InsertionSort
    return M
end function
```

Theorem 9.5.7. Assume that $0 < \epsilon < 1/2$. Let M be the output of the ApproxMedian algorithm. Then

$$\Pr[\, M \text{ is not an } \epsilon\text{-approximate-median of } A \,] \ \leq \ 2\exp(-2\epsilon^2 \ell).$$

Proof. For simplicity this proof will assume that the elements of A are distinct, and that $(\frac{1}{2} - \epsilon)n$ is an integer. Since the algorithm is unaffected by the ordering of A, we will assume for convenience that A is sorted. Define

$$L = A[(\tfrac{1}{2} - \epsilon)n] \qquad \text{and} \qquad R = A[(\tfrac{1}{2} + \epsilon)n + 1].$$

Note that any $x \in A$ with $L \leq x \leq R$ has at least $(\frac{1}{2} - \epsilon)n$ elements that are no larger than x, namely $A[1], A[2], \ldots, A[(\frac{1}{2} - \epsilon)n]$. Similarly, there are at least $(\frac{1}{2} - \epsilon)n$ elements that are no smaller than x. It follows that x is an ϵ-approximate-median.

Since each Z_i is uniformly distributed on A, and the elements of A are distinct, we have

$$\begin{aligned} \Pr[\, Z_i \leq L \,] &= \frac{(\frac{1}{2} - \epsilon)n}{n} = \tfrac{1}{2} - \epsilon \\ \Pr[\, Z_i \geq R \,] &= \frac{n - (\frac{1}{2} + \epsilon)n}{n} = \tfrac{1}{2} - \epsilon. \end{aligned} \tag{9.5.4}$$

Since M is defined to be the median of the Z_i's, we can now use our analysis of the median estimator. The tail bounds (9.5.4) show that the hypotheses of Theorem 9.5.5 are satisfied, with the parameter $\alpha = \epsilon$. Thus

$$\Pr[\, M \text{ is } \textit{not} \text{ an } \epsilon\text{-approximate-median} \,] \;\leq\; \Pr[\, M \leq L \;\vee\; M \geq R \,] \;\leq\; 2\exp(-2\epsilon^2 \ell).$$

directly from Theorem 9.5.5. $\square$

Question 9.5.8. To make the parameters more concrete, suppose we want to find a 0.05-approximate median, and we want a failure probability of 0.01. How should we choose ℓ? What is the runtime of the algorithm?

Answer.

Take

$$\ell = \left\lceil \frac{\ln(200)}{2\epsilon^2} \right\rceil = \lceil 200 \ln(200) \rceil.$$

If necessary, add 1 so that ℓ is odd. Then the failure probability is

$$2\exp(-2\epsilon^2 \ell) \;\leq\; 2\exp\Big(-2\epsilon^2 \frac{\ln(200)}{2\epsilon^2}\Big) = \frac{2}{200} = 0.01.$$

Note that ℓ depends only on the desired failure probability (0.01) and on $\epsilon = 0.05$, but does *not* depend on the size of A. Thus, the runtime is $O(\ell^2) = O(1)$.

Problem to Ponder 9.5.9. How can you extend the proof of Theorem 9.5.7 to handle the case in which the elements of A are not necessarily distinct, and $(\frac{1}{2} - \epsilon)n$ is not necessarily an integer?

9.6 Exercises

Exercise 9.1 **Hoeffding for averages.** Hoeffding's inequality as stated above is for *sums* of random variables. It is convenient to restate it for *averages*. Let $X_1, \ldots, X_n$ be independent random variables with the guarantee $0 \leq X_i \leq 1$. Define $Y = \frac{1}{n} \sum_{i=1}^{n} X_i$.

Theorem 9.6.1 (Hoeffding for averages). For any $q \geq 0$,

$$\begin{aligned} & \Pr[\,Y \geq \mathrm{E}[\,Y\,] + q\,] && \leq \exp(-2q^2 n) \\ \text{and} \quad & \Pr[\,Y \leq \mathrm{E}[\,Y\,] - q\,] && \leq \exp(-2q^2 n). \\ \text{Combining these} \quad & \Pr[\,|Y - \mathrm{E}[\,Y\,]| \geq q\,] && \leq 2\exp(-2q^2 n). \end{aligned} \tag{9.6.1}$$

Prove this theorem.

Exercise 9.2 **Evaluating loss functions.** In a simplified view of machine learning, we have "examples" $x_1, \ldots, x_m$ and a "loss function" ℓ that measures the accuracy of our predictions for each example. For simplicity, assume that $x_1, \ldots, x_m$ are real numbers and ℓ is the sigmoid function, defined by $\ell(x) = \frac{e^x}{e^x+1}$. Our overall loss for all the examples is

$$L = \frac{1}{m} \sum_{i=1}^{m} \ell(x_i).$$

(In a real machine learning context, the loss would also incorporate model parameters, which are optimized to improve predictions. However the model parameters are irrelevant to this exercise, so we omit them.)

Assume that evaluating ℓ takes $O(1)$ time. We could compute the overall loss L exactly in time $\Theta(m)$, but this is often much too slow. Design a much faster algorithm to estimate L. Let α be a fixed parameter with $0 < \alpha < 1$. Your algorithm should run in $O(1/\alpha^2)$ time, and should output a value $\hat{L}$ satisfying

$$\Pr\left[\,|L - \hat{L}| \geq \alpha\,\right] \leq 0.01.$$

Exercise 9.3. Let $X_1, \ldots, X_n$ be independent RVs satisfying $0 \leq X_i \leq 1$. Let $Y = \frac{1}{n} \sum_{i=1}^{n} X_i$. For all q satisfying $0 < q \leq 1/2$, prove that

$$\Pr\left[\,|Y - \mathrm{E}[\,Y\,]| \geq \sqrt{\ln(2/q)/2n}\,\right] \leq q.$$

Exercise 9.4. Let $X_1, \ldots, X_n$ be independent indicator RVs. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathrm{E}[\,X\,]$. Suppose that $\mu > 0$. For all q satisfying $0 < q \leq 1$, prove that

$$\Pr\left[\,X - \mu \leq -\sqrt{3\mu \ln(1/q)}\,\right] \leq q.$$

Exercise 9.5. Let $X_1, \ldots, X_n$ be independent random variables with the guarantee $0 \leq X_i \leq 1$. Define $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathrm{E}[\,X\,]$. Consider the following slightly weaker form of the Hoeffding bound.

Theorem (Weak Hoeffding). For any $t \geq 0$,

$$\Pr[\,|X-\mu| \geq t\,] \;\leq\; 2\exp(-t^2/3n).$$

Prove this theorem using the Chernoff bound.

Exercise 9.6. In this question we will prove Theorem 9.5.5. Define the events

$$\begin{aligned} \mathcal{E} &= \text{"}|\{\, i \,:\, Z_i \leq L \,\}| \geq \ell/2\text{"} \\ \mathcal{F} &= \text{"}|\{\, i \,:\, Z_i \geq R \,\}| \geq \ell/2\text{"}. \end{aligned}$$

In words, $\mathcal{E}$ is the event that at least $\ell/2$ of the Z_i RVs have value less than L.

Part I. Prove that $\Pr[\,\mathcal{E}\,] \leq \exp(-2\alpha^2\ell)$ using Hoeffding's inequality. (A similar argument also shows that $\Pr[\,\mathcal{F}\,] \leq \exp(-2\alpha^2\ell)$, so you needn't prove it.)

Part II. Prove that

$$\Pr[\,M \leq L \;\text{ or }\; M \geq R\,] \leq 2\exp(-2\alpha^2\ell).$$

Exercise 9.7. Let Z be a random variable having the distribution defined in Exercise 2.8, where we set the parameter $K = 10^6$.

Part I. What is $\mathrm{E}[\,Z\,]$?

Part II. What is the median of Z?

Part III. Implement code that draws $n = 1001$ samples, which we call $Z_1, \ldots, Z_n$, from that distribution. For every $j \leq n$, define

$$S_j \;=\; \frac{1}{j}\sum_{i=1}^{j} Z_i \qquad \text{and} \qquad M_j \;=\; \mathrm{Median}(Z_1, \ldots, Z_j).$$

(If j is even, we can define M_j to be the average of the $(j/2)^{\text{th}}$-largest and $(j/2+1)^{\text{th}}$ largest elements.) Plot S_j and M_j as a function of j for $j = 1, \ldots, n$.

Part IV. Which would you say is better: S_j as an estimate for the mean, or M_j as an estimate for the median?

Exercise 9.8 **Hoeffding is not always tight.** Continuing the example of Section 9.1, consider bounds on $\Pr[\,X \geq 3n/4\,]$ where X has the distribution $B(n, 1/2)$. Both the Hoeffding bound and the normal approximation have bounds that decrease exponentially like $e^{-n/8} = e^{-0.125n}$. Presumably this is the correct rate of exponential decrease?

Actually, no. By a direct calculation using binomial coefficients, show that $\Pr[\,X \geq 3n/4\,] < e^{-0.13n}$, if n is sufficiently large. You may assume that n is a multiple of 4.

Chapter 10

Applications of Concentration

10.1 Ratio of loads in load balancing

Let us return to a random load balancing problem discussed in Section 7.5. Suppose that there are n servers and m clients, At that time, we asked the question:

> Suppose we want all servers to have roughly the same number of clients, say up to a factor of 3. How large should m be?

Our concentration tools are the key to answering this question.

Theorem 10.1.1. Consider random load balancing with n servers and m clients where

$$m = 12n\ln(200n).$$

The expected number of clients on each server is $\mu = m/n = 12\ln(200n)$. Then

$$\Pr[\text{ on } \textit{every} \text{ server the number of clients is between } 0.5\mu \text{ and } 1.5\mu\,] \;\geq\; 0.99.$$

For example, suppose there are $n = 100$ servers and $m \approx 11884$ clients. The expected number of clients on each server is $\mu \approx 119$. With 99% probability there are between 59 and 179 clients on each server.

References: (Kleinberg and Tardos, 2006, Theorem 13.46).

Proof. Since the servers are all equivalent, we focus our attention on the first server. Let X be the number of clients that the first server gets. We can decompose this into a sum of indicator RVs:

$$X = \sum_{i=1}^{m} X_i \qquad \text{where} \qquad X_i = \begin{cases} 1 & \text{(if client } i \text{ assigned to first server)} \\ 0 & \text{(otherwise).} \end{cases}$$

Note that $\mathrm{E}[X_i] = 1/n$ and so $\mathrm{E}[X] = m/n = \mu = 12\ln(200n)$. The Chernoff bound is a great tool to analyze the probability that the first server has too few or too many clients.

$$\begin{aligned}
\Pr[X \leq 0.5\mu \;\vee\; X \geq 1.5\mu] &= \Pr\left[\,|X-\mu| \geq \underbrace{0.5}_{\delta}\,\mu\,\right] \\
&\leq 2\exp(-\delta^2\mu/3) \qquad \text{(by the Chernoff Bound, Theorem 9.2.2)} \\
&= 2\exp(-\mu/12) \;=\; 2\exp(-\ln(200n)) \;=\; \frac{1}{100n}.
\end{aligned}$$

So far we have just analyzed the first server. The servers are all symmetric, so the same argument applies to any server. The remaining issue is that we want the load to be balanced on all servers *simultaneously*. The key trick to accomplish that is the union bound.

Define the event

$$\mathcal{E}_i = \text{“the number of clients on server } i \text{ is } \textit{not} \text{ between } 0.5\mu \text{ and } 1.5\mu\text{”}.$$

We have argued that

$$\Pr[\,\mathcal{E}_i\,] \;\le\; \frac{1}{100n} \qquad \forall i \in [n].$$

By a union bound,

$$\Pr[\,\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n\,] \;\le\; \sum_{i=1}^{n} \Pr[\,\mathcal{E}_i\,] \;\le\; n \cdot \frac{1}{100n} \;=\; 0.01. \tag{10.1.1}$$

Thus, there is at most 1% probability that *any* server fails to have between 0.5μ and 1.5μ clients. $\square$

Hoeffding won't work. Let's try applying the Hoeffding bound for this problem, still assuming $m = 12n\ln(200n)$ clients. We analyze $\Pr[\,\mathcal{E}_i\,]$ using the Hoeffding bound with $t = 0.5\mu = m/2n$.

$$\begin{aligned}
\Pr[\,\mathcal{E}_i\,] &= \Pr\Big[\, |X_i - \mu| \ge \underbrace{0.5\mu}_{t} \,\Big] \\
&\le 2\exp\Big(-\frac{2t^2}{m}\Big) \qquad \text{(the Hoeffding bound, Theorem 9.3.1)} \\
&= 2\exp\Big(-\frac{m}{2n^2}\Big) \;=\; 2\exp\Big(-\frac{6\ln(200n)}{n}\Big).
\end{aligned}$$

The trouble is, as n increases, $6\ln(200n)/n$ goes to zero, so $2\exp(-6\ln(200n)/n)$ goes to 2. So we are upper bounding $\Pr[\,\mathcal{E}_i\,]$ by a value close to 2, which is completely useless because all probabilities are at most 1.

Question 10.1.2. How large should m be in order for the Hoeffding bound to guarantee that every server has roughly the same number of clients, up to a factor of 3?

Answer.

Set $m = 2n^2\ln(200n)$. As above, we get

$$\Pr[\,\mathcal{E}_i\,] \;\le\; 2\exp\Big(-\frac{m}{2n^2}\Big) \;=\; 2\exp\Big(-\frac{2n^2\ln(200n)}{2n^2}\Big) \;=\; \frac{1}{100n}.$$

Next we can apply a union bound as in (10.1.1).

10.2 Probability amplification, for two-sided error

Probability amplification is an idea that we have seen in Section 1.3. It is a way to decrease the probability of false positives/negatives in algorithms with Boolean output. We distinguished between algorithms with:

- **One-sided error.** These algorithms either have no false positives or no false negatives.

Algorithm 10.1 The algorithm $\mathcal{A}$ for amplifying success of algorithm $\mathcal{B}$. The parameter ℓ determines the number of trials.

function AMPLIFY(algorithm $\mathcal{B}$, input x, integer ℓ)
    $X \leftarrow 0$ ▷ Count of number of trials that output Yes
    for $i = 1, \ldots, \ell$ **do**
        Run $\mathcal{B}(x)$, using new independent values for its internal random choices
        if output is Yes **then** $X \leftarrow X + 1$
    end for
    if $X \geq \ell/2$
        return Yes   ▷ At least half of the outputs were Yes, so the correct output is probably Yes
    else
        return No   ▷ Less than half of the outputs were Yes, so the correct output is probably No
end function

- **Two-sided error.** These algorithms can have both false positives and false negatives.

Previously we discussed probability amplification for algorithms with one-sided error. Now we will discuss the more intricate case of two-sided error.

Recall that we defined a BPP-type algorithm as having probability of false positives and false negatives both at most 1/3. We would like to perform repeated trials and exponentially decrease the probability of false positives and false negatives.

Algorithm's Output / Correct Output	Yes	No
Yes	$\geq 2/3$ true positive	$\leq 1/3$ false negative
No	$\leq 1/3$ false positive	$\geq 2/3$ true negative

Figure 10.1: Probabilities of outcomes for a BPP-type algorithm.

Let $\mathcal{B}$ be a BPP-type algorithm. Depending on the input x, there are two cases for its behavior.

- **For input x, the correct Output is Yes**. Then we know that $\Pr[\mathcal{B}(x) \text{ outputs Yes}] \geq 2/3$.
- **For input x, the correct Output is No**. Then we know that $\Pr[\mathcal{B}(x) \text{ outputs Yes}] \leq 1/3$.

The situation is strongly analogous to Section 11.1.

$$\begin{aligned} \text{Executing } \mathcal{B}(x) \text{ when Correct Output is Yes} \quad &\cong \quad \text{Flipping Fluffy's coin} \\ \text{Executing } \mathcal{B}(x) \text{ when Correct Output is No} \quad &\cong \quad \text{Flipping Nick's coin} \end{aligned}$$

Our intuition from coin flipping was: if we see relatively few heads, we should declare it to be Nick's coin; otherwise, declare it to be Fluffy's coin. We can convert this intuition into an amplification algorithm as shown in Algorithm 10.1.

Theorem 10.2.1. Suppose $\mathcal{B}$ is a BPP-type algorithm. Then AMPLIFY$(\mathcal{B}, x, \ell)$ satisfies

$$\begin{aligned} \Pr[\text{false positive}] &\leq \exp(-\ell/18) \\ \Pr[\text{false negative}] &\leq \exp(-\ell/18). \end{aligned}$$

References: (Motwani and Raghavan, 1995, Section 6.8), (Sipser, 2012, Lemma 10.5).

It is interesting to compare the amplification results from this section and from Section 1.3.

- Theorem 1.3.1 made the strong assumption that $\mathcal{B}$ has no false negatives, then concluded that the false positive rate decreases like $2^{-\ell}$.

- Theorem 10.2.1 allowed $\mathcal{B}$ to have both false positives and false negatives, then concluded that both rates decrease like $e^{-\ell/18}$.

So the hypothesis of Theorem 10.2.1 is much weaker but its conclusion is roughly the same, if we're willing to overlook the factor 18.

Proof of Theorem 10.2.1. Let us consider an input x for which the correct output is Yes. (An analogous argument works if the correct output is No.) We would like to show that algorithm $\mathcal{A}$ is very likely to output Yes.

In each iteration, the probability of $\mathcal{B}(x)$ returning Yes is $p \geq 2/3$. So the expected number of Yes produced by $\mathcal{B}$ is

$$\mu \;=\; p\ell \;\geq\; (2/3)\ell.$$

The algorithm will only make a mistake if $X < \ell/2$. Once again, the Hoeffding bound is a great tool to analyze this.

$$\begin{aligned}
\Pr[\,X < \ell/2\,] &= \Pr[\,X < (2/3)\ell - (1/6)\ell\,] \\
&\leq \Pr\left[\,X \leq \mu - \underbrace{(1/6)\ell}_{t}\,\right] \\
&\leq \exp(-2t^2/\ell) \qquad \text{(by the Hoeffding bound, Theorem 9.3.1)} \\
&= \exp\big(-2\cdot(\ell/6)^2/\ell\big) \\
&= \exp(-\ell/18).
\end{aligned}$$

$\square$

Exercises

Exercise 10.1. Let $\mathcal{A}$ be a BPP-type algorithm solving some problem, and whose runtime is random. Specifically, we know a constant c such that $\mathrm{E}\,[\,\text{runtime of } \mathcal{A}\,] \leq cn^2$ for inputs of size n.

Using $\mathcal{A}$ (and f), create another BPP-type algorithm $\hat{\mathcal{A}}$ solving the same problem, except that $\hat{\mathcal{A}}$'s runtime is *not* random. Specifically, we *always* have runtime of $\hat{\mathcal{A}} = O(n^2)$.

Hint: `man timeout`

Exercise 10.2. Let us view the outputs of $\mathcal{B}$ as being 0/1 rather than No/Yes. Consider the following amplification algorithm.

Algorithm 10.2 An alternative algorithm for amplifying success of algorithm $\mathcal{B}$. The parameter ℓ determines the number of trials.

function AmplifyByMedian(algorithm $\mathcal{B}$, integer ℓ)
    for $i = 1, \ldots, \ell$ **do**
        Let Z_i be the output of $\mathcal{B}$ with new independent random values
    end for
    return Median$(Z_1, \ldots, Z_\ell)$
end function

Use Theorem 9.5.5 to prove the following theorem.

Theorem. Suppose $\mathcal{B}$ is a BPP-type algorithm. Then AmplifyByMedian$(\mathcal{B}, \ell)$ satisfies

$$\begin{aligned} \Pr[\,\text{false positive}\,] &\leq \exp(-\ell/18) \\ \Pr[\,\text{false negative}\,] &\leq \exp(-\ell/18). \end{aligned}$$

10.3 QuickSort, with high probability

We have discussed Randomized QuickSort several times already. Let X be the number of comparisons for an array of size n. We have already shown the following results.

$$\begin{aligned} \text{Section 4.4:} \qquad & \mathrm{E}[\,X\,] < 2n \ln n \\ \text{Section 8.4.1:} \qquad & \Pr[\,X \geq 200 n \ln n\,] \leq 0.01. \end{aligned}$$

The Chernoff bound will now be exploited to give much tighter guarantees for the right tail.

Theorem 10.3.1.

$$\Pr[\,X \geq 36 n \ln n\,] \leq 1/n.$$

Following the approach of Section 5.3, we will show that the recursion tree is very likely to have depth $O(\log n)$. By Claim 5.3.2, that will imply that the runtime is very likely to be $O(n \log n)$.

In QuickSort, each leaf of the recursion tree corresponds to a distinct element of the input array, so there are exactly n leaves. So we would like to fix some $d = O(\log n)$ and show that:

> for every element in the input array, its corresponding leaf is at level at most d.

Or equivalently,

> for every element in the input array, it belongs to at most d recursive subproblems.

Our intuitive understanding of QuickSort tells us that many of the partitioning steps will split the array roughly in half. We now turn this intuition into a precise probabilistic statement. Let us say that a partitioning step is *good* if it partitions the current array into two parts, both of which have size *at least one third* of that array. In other words, the pivot is in the middle third of that array, when viewed in sorted order. There are two reasons why we call these partitioning steps good.

- Each pivot is good with probability $1/3$.

- Each good partitioning step shrinks the size of the current array by a factor of $2/3$ or better. After k partitioning steps the array has size at most $(2/3)^k n = e^{-\ln(3/2)k} n$. Consider $k = \log_{3/2}(n)$, which equals $\ln(n)/\ln(3/2)$ by the familiar log rule (A.2.4). A quick numerical calculation tells us that $k < 3\ln(n)$. So after $3\ln n$ good partitioning steps the current array has size at most 1, which means that the recursion has hit a leaf. This tells us that every element can be involved in at most $3\ln n$ good partitioning steps.

Our only worry is that x could be involved in many bad partitioning steps. This is where the Chernoff bound comes to the rescue. Define

$$d \;=\; 36\ln n.$$

Claim 10.3.2. Fix any element v of the input array.

$$\Pr\left[\,\text{element } v \text{ is involved in} \geq d \text{ subproblems}\,\right] \;\leq\; \frac{1}{n^2}.$$

Proof. Each partitioning step involving v may be viewed as a random trial that succeeds if the pivot is good. We argued that the recursion terminates if there are at least $3\ln n$ good pivots, so we want to show that there are very likely to be $3\ln n$ success within d trials[1].

Let Z be the number of successes among d trials[2], each of which has success probability $p = 1/3$. The expected number of successes is $\mu = d/3 = 12\ln n$. Letting $\delta = 3/4$, we have

$$\begin{aligned}
&\Pr\left[\,\text{element } v \text{ is involved in} \geq d \text{ subproblems}\,\right] \\
&\leq\; \Pr\left[\,Z < 3\ln n\,\right] \\
&=\; \Pr\left[\,Z < (1-\delta)\mu\,\right] && \text{(definition of } \delta \text{ and } \mu\text{)} \\
&\leq\; \exp(-\delta^2\mu/3) && \text{(by Chernoff bound)} \\
&=\; \exp\left(-\frac{3^2}{4^2}\cdot\frac{12\ln n}{3}\right) \\
&<\; \exp(-2\ln n) \;=\; \frac{1}{n^2}.
\end{aligned}$$

□

Proof of Theorem 10.3.1. Let

$$\mathcal{E}_i \;=\; \left\{\text{element } i \text{ is involved in} \geq d \text{ subproblems}\right\}.$$

Then

$$\begin{aligned}
\Pr\left[\,\text{recursion tree has} \geq d \text{ levels}\,\right] \;&=\; \Pr\left[\,\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_m\,\right] \\
&\leq\; \sum_{i=1}^{n} \Pr\left[\,\mathcal{E}_i\,\right] && \text{(by the union bound)} \\
&\leq\; \sum_{i=1}^{n} \frac{1}{n^2} && \text{(by Claim 10.3.2)} \\
&=\; \frac{1}{n}.
\end{aligned}$$

[1]The reader who is fastidious about probability may notice a sneaky maneuver here. We have subtly switched from discussing the negative binomial distribution to the binomial distribution. This switch is perfectly legitimate.

[2]Don't let the notation confuse you. In our statement of the Chernoff bound there are n trials. In our current application of it there are d trials and n means something else.

Thus, by Claim 5.3.2,

$$\Pr[\text{number of comparisons} \geq nd] \;\leq\; \Pr[\text{recursion tree has} \geq d \text{ levels}] \;\leq\; 1/n. \qquad \square$$

References: (Mitzenmacher and Upfal, 2005, Exercise 4.20).

10.3.1 Exercises

Exercise 10.3. One idea to improve the runtime of QuickSort is the "median of three" heuristic. Without randomization, this does not improve the worst-case runtime.

In this exercise we explore a variant that defines uses the median of ℓ randomly chosen elements as the pivot. The changes from QuickSort are shown in yellow.

Algorithm 10.3 A variant of Randomized QuickSort that uses an approximate median. Assume that the elements in A are distinct.

```
1: global int ℓ
2: function MedOfℓQuickSort(set A)
3:     if Length(A) ≤ 5 then
4:         return InsertionSort (A)
5:     Select ℓ elements Z_1, ..., Z_ℓ ∈ A uniformly and independently at random
6:     Find the median p of Z_1, ..., Z_ℓ using InsertionSort        ▷ The pivot element
7:     Construct the sets Left = { x ∈ A : x < p } and Right = { x ∈ A : x > p }
8:     return the concatenation [ MedOfℓQuickSort(Left), p, MedOfℓQuickSort(Right) ]
9: end function
```

Suppose we wish to sort an array A of length n. Let $\ell = 36\ln(20n)$ and $d = \log_{3/2}(n)$.

Part I. Prove that the recursion tree has at most n subproblems in total.

Part II. Define $\mathcal{E}$ to be the event that *all* pivots are $1/6$-approximate medians. Prove that $\Pr[\mathcal{E}] \geq 0.99$.

Part III. Prove that, if event $\mathcal{E}$ happens, all leaves of the recursion are at level at most d.

Part IV. If $\mathit{Length}(A) = s$, how much time do lines 5-7 take as a function of s and ℓ?

Part V. Prove that, if event $\mathcal{E}$ happens, the overall runtime is $O(n \log^2 n)$.

Part VI. Modifying only line 3, improve the runtime to $O(n \log n)$.

10.4 Exercises

Exercise 10.4. Suppose that we use random load balancing with n clients and n servers. Let X_i be the number of clients assigned to server i. Let $X = \max_{1 \leq i \leq n} X_i$. Use the Chernoff bound to prove that $\Pr[\, X \geq 1 + 6 \ln n \,] \leq 1/n$.

Exercise 10.5. Let $\mathcal{A}$ be a BPP-algorithm solving some problem, and whose runtime is random. Specifically, we know a constant c such that $\mathrm{E}[\,\text{runtime of } \mathcal{A}\,] \leq cn^2$ for inputs of size n.

Using $\mathcal{A}$ (and c), create another BPP-algorithm $\hat{\mathcal{A}}$ solving the same problem, except that we *always* have runtime of $\hat{\mathcal{A}} = O(n^2)$. (So we have a non-random upper bound on the runtime.)

Chapter 11

Classical statistics

11.1 Distinguishing coins

Consider a silly problem about flipping coins. Suppose Harry has two biased coins:

- **Fluffy's coin:** this coin has $\Pr[\text{heads}] = 2/3$.
- **Nick's coin** this coin has $\Pr[\text{heads}] = 1/3$.

Visually, they are indistinguishable. Harry gives you one of the coins and asks you to figure out which one it is. What should you do?

The obvious idea is to flip the coin repeatedly and see what happens. Here is the result of 60 random flips.

HTHTTTTTTTTTTTTHTTHTTTTTTHTTHHTTTTTTTTTHTHHTHTTTTHHTHTHTTTTHHH

Question 11.1.1. Intuitively, is it likely is for Fluffy's coin to generate these outcomes? What about for Nick's coin? Which coin do you think you have?

There are 42 T, which significantly outnumber the 18 H. Intuition suggests that this is a very unlikely outcome for Fluffy's coin, and it is a moderately likely outcome for Nick's coin. We will show this precisely.

But first let us be clear on the precise question that we are addressing. We are *not* asking:

> Given that we have seen so few heads, what is the probability of having Fluffy's coin?

The reason is: the condition "You have Fluffy's coin" is *not* a random event. This condition can be deliberately, or even maliciously, decided by Harry. It does not make sense to talk about the probability of something that is not random. Instead, we are asking:

> Assuming that we have Fluffy's coin, what is the probability of seeing so few heads?

The conventional approach. Introductory statistics classes often discuss this sort of problem. The first question, which we won't consider any further, is an example of Bayesian inference. The second question is an example of hypothesis testing. The conventional solution is roughly as follows.

Suppose that we actually had Fluffy's coin, which has probability of heads 2/3. Let X be the number of heads observed in 60 tosses. The expectation is

$$\mu \;=\; \mathrm{E}[\,X\,] \;=\; 60 \cdot 2/3 \;=\; 40.$$

We would like to show that it is unlikely that the observed number of heads is much smaller than μ. The number of heads has a binomial distribution, so it can be approximated using the normal distribution if the number of flips is large. So it remains to determine the probability that a normal RV is far from its mean. How?

> *In practice, such values are obtained in one of two ways — either by using a normal table, a copy of which appears at the back of every statistics book, or by running a computer software package.*
>
> *"An Introduction to Mathematical Statistics", R. Larsen and M. Marx*

There are a few disadvantages of this conventional approach. The most concerning is that table lookups and software calculations are strategies geared towards small data sets, and don't tell us about asymptotic behavior. For our purposes it is more useful to have big-O bounds that are relevant for big data sets. Another disadvantage to the conventional approach is that it invokes the normal approximation without discussing precise guarantees, which might be unsettling for mathematical purists.

Our approach. We will deviate from the conventional approach by using the Hoeffding bound instead of the normal approximation. This has the advantage that it gives precise guarantees for any number of coin flips, and we won't need to refer to a normal table.

As above, let X be the number of heads observed in 60 tosses of Fluffy's coin. This has the binomial distribution $B(60, 2/3)$. The mean is $\mu = \mathrm{E}[\,X\,] = 40$. We would like to show that it is unlikely that the number of heads is much smaller than μ. The Hoeffding bound is ideal for this task.

$$\begin{aligned}
\Pr[\,X \leq 18\,] &= \Pr\left[\,X \leq \underbrace{40}_{\mu} - \underbrace{22}_{t}\,\right] \\
&\leq \exp(-2t^2/60) && \text{(by the Hoeffding bound, Theorem 9.3.1)} \\
&= \exp(-2 \cdot 22^2/60) \\
&< 0.0000001.
\end{aligned}$$

To summarize, if you actually had Fluffy's coin, then there is less than a 0.00001% chance that you would see so few heads.

Question 11.1.2. The Chernoff bound will also work for this problem. What bound do you get?

Answer.

Apply the Chernoff bound with $\delta = 22/40$ to get

$$\Pr[\,X \leq 18\,] \;=\; \Pr[\,X \leq (1-\delta)\mu\,] \;\leq\; \exp(-\delta^2\mu/3) \;<\; e^{-4} \;<\; 0.19.$$

11.2 Polling

Recall from Section 4.3 the problem of polling m customers. We considered three sampling approaches to produce an unbiased estimator $\hat{f}$. In this section, we will show that $\hat{f}$ is concentrated around its expectation f. Choosing some concrete numbers, we would there to be probability 0.95 that $\hat{f}$ lies in the interval $[f-0.03, f+0.03]$. Stated concisely, we would like to show

$$\Pr\left[|\hat{f}-f| \geq 0.03\right] \;\leq\; 0.05. \tag{11.2.1}$$

Terminology. Introductory statistics classes often discuss this sort of problem. In the terminology of statistics:

- this sort of problem is called interval estimation.
- the value 0.03 is called the margin of error.
- the value 0.95 is called the ***confidence level***.
- the interval $[f-0.03, f+0.03]$ is called a 95%-confidence interval.

References: (Anderson et al., 2017, Section 4.3), (Larsen and Marx, 2018, Section 5.3).

We will show that, for all three sampling approaches, the number of people we have to poll *does not depend on* m! Whether we wanted to poll everyone in Palau Islands (population $17,907$) or everyone in China (population $1,411,778,724$), it would suffice to poll a few thousand people.

11.2.1 Sampling with replacement

As in Section 4.3.1, we pick k customers uniformly and independently at random. We use the estimator $\hat{f}=\sum_{i=1}^{k} X_i/k$ from (4.3.1), where X_i indicates whether the i^{th} sampled person likes avocados.

Theorem 11.2.1. Let $k=2050$. Let $\hat{f}$ be the output of $\textsc{PollWithReplacement}(m,k)$. Then, with probability 0.95, $\hat{f}$ is within 0.03 of f.

We have shown in Section 4.3 that $\mathrm{E}\left[\hat{f}\right]=f$. Say we want that $\hat{f}$ is within ± 0.03 of the true value of f, with probability 0.95. Using the Hoeffding Bound for Averages (Theorem 9.6.1), we have

$$\Pr\left[|\hat{f}-f| \geq q\right] \;\leq\; 2\exp(-2q^2k).$$

Taking $q=0.03$ and $k=2050$, we get that

$$\Pr\left[|\hat{f}-f| \geq 0.03\right] \;\leq\; 2\exp(-2\cdot(0.03)^2\cdot 2050) \;<\; 0.05.$$

To conclude, if Zoodle polls 2050 customers chosen randomly with replacement then, with probability 0.95, the fraction of sampled customers who like avocados will differ from the true fraction by at most 0.03.

11.2.2 Sampling without replacement

As in Section 4.3.2, we pick k customers uniformly and independently at random. We use the estimator $\hat{f}=\sum_{i=1}^{k} X_i/k$ from (4.3.2), where X_i indicates whether the i^{th} sampled person likes avocados.

Theorem 11.2.2. Let $k=2050$. Let $\hat{f}$ be the output of $\textsc{PollWithoutReplacement}(m,k)$. Then, with probability 0.95, $\hat{f}$ is within 0.03 of f.

Recall that the random variables $X_1, \ldots, X_k$ are *not* independent, so the hypotheses of Hoeffding's bound are *not* satisfied. Remarkably, Hoeffding himself showed that his bound will still hold in this scenario. Note that the bounds here are identical to Theorem 9.6.1.

Theorem 11.2.3 (Hoeffding for averages when sampling without replacement). Let $C = [C_1, \ldots, C_m]$ be a sequence of values in $[0,1]$. Define the RVs $X_1, \ldots, X_n$ by sampling from C without replacement. Define $Y = \frac{1}{n}\sum_{i=1}^{n} X_i$. For any $q \geq 0$,

$$\begin{aligned} & \Pr[\, Y \geq \mathrm{E}[\, Y\,] + q\,] && \leq \exp(-2q^2 n) \\ \text{and} \qquad & \Pr[\, Y \leq \mathrm{E}[\, Y\,] - q\,] && \leq \exp(-2q^2 n). \\ \text{Combining these} \qquad & \Pr[\, |Y - \mathrm{E}[\, Y\,]| \geq q\,] && \leq 2\exp(-2q^2 n). \end{aligned}$$

References: Hoeffding's paper, Section 6.

We now follow the same argument used above for sampling with replacement, but use Theorem 11.2.3 instead. Plugging in $q = 0.03$ and $n = k = 2050$, we obtain

$$\begin{aligned} \Pr\left[\, |\hat{f} - f| \geq q \,\right] &\leq 2\exp(-2q^2 k) \qquad (11.2.2) \\ &= 2\exp(-2 \cdot (0.03)^2 \cdot 2050) \;<\; 0.05. \end{aligned}$$

This proves Theorem 11.2.2.

11.2.3 Bernoulli sampling

Theorem 11.2.4. Let $k = 12300$. Let $\hat{f}$ be the output of BernoulliPolling(C, k). Then, with probability 0.95, $\hat{f}$ is within 0.03 of f.

The sampling probability is $p = k/m$. Let $\mathcal{A}$ be the set of all customers who like avocados. The number of sampled customers who like avocados can be decomposed into indicators as

$$X = \sum_{i \in \mathcal{A}} X_i.$$

We can write the true fraction and the estimate as

$$\text{True fraction:} \qquad f \;=\; \frac{|\mathcal{A}|}{m} \qquad (11.2.3)$$

$$\text{Estimated fraction:} \qquad \hat{f} \;=\; \frac{X}{pm} \;=\; \frac{1}{pm}\sum_{i \in \mathcal{A}} X_i. \qquad (11.2.4)$$

We have shown in Claim 4.3.4 that $\hat{f}$ is unbiased, i.e., $\mathrm{E}\left[\,\hat{f}\,\right] = f$. Multiplying by pm, this yields

$$\mathrm{E}[\, X\,] \;=\; pmf. \qquad (11.2.5)$$

We next use the Chernoff bound to show that $\hat{f}$ is concentrated around its mean.

Lemma 11.2.5. $\Pr\left[\, |\hat{f} - f| \geq 0.03 \,\right] \leq 2\exp(-0.0003pm)$.

Proof. First we employ a small trick. Suppose that f is less than $1/2$. Then we could instead consider the fraction of customers who *dislike* avocados, namely $1 - f$, which is more than $1/2$. Our estimate for this fraction would be $1-\hat{f}$. The estimation error for the disliking fraction is the same as for the liking fraction because $|(1-\hat{f})-(1-f)| = |\hat{f}-f|$. So we may assume that $f \geq 1/2$.

Next, we set things up to use the Chernoff bound. Let $\mu = \mathrm{E}[\,X\,] = pmf$, by (11.2.5).

$$\begin{aligned}
\Pr\left[\,|\hat{f}-f| \geq 0.03\,\right] &= \Pr[\,|X - pmf| \geq 0.03pm\,] && \text{(multiply both sides of inequality by } pm\text{)} \\
&= \Pr\left[\,|X-\mu| \geq \underbrace{\frac{0.03}{f}}_{=\delta}\underbrace{pmf}_{=\mu}\,\right] && \text{(using value of } \mu\text{),}
\end{aligned}$$

where we have defined $\delta = 0.03/f$. Now we use our $f \geq 1/2$ trick[1] to conclude that $\delta \leq 1$. So Theorem 9.2.2 yields

$$\begin{aligned}
\Pr[\,|X-\mu| \geq \delta\mu\,] &\leq 2\exp(-\delta^2\mu/3) \\
&= 2\exp\left(-\left(\frac{0.03}{f}\right)^2\frac{pmf}{3}\right) \\
&\leq 2\exp(-0.03^2pm/3) \qquad \text{(using } f \leq 1\text{)} \\
&= 2\exp(-0.0003pm).
\end{aligned}$$

We conclude that $\Pr\left[\,|\hat{f}-f| \geq 0.03\,\right] = \Pr[\,|X-\mu| \geq \delta\mu\,] \leq 2\exp(-0.0003pm)$. □

Now we just need to plug in our sampling probability. If $m \leq 12300$ then the algorithm sets $p \leftarrow 1$, so the algorithm is deterministic and we have $\hat{f} = f$ exactly. If $m > 12300$ then $p = 12300/m$ and

$$\Pr\left[\,|\hat{f}-f| > 0.03\,\right] \leq 2\exp(-0.0003pm) = 2\exp(-3.69) < 0.05.$$

This proves Theorem 11.2.4.

11.3 A/B Testing

Zoodle has redesigned their website to try to make their avocado slicers seem more appealing. It decides to run an experiment to see if the new website actually increases sales. This sort of experiment is called A/B Testing.

To make the problem more concrete, we will need a fair amount of notation.

$$\begin{aligned}
C &= \{\text{all customers}\} \\
m &= |C| = \text{number of customers} \\
\mathcal{A} &= \{\text{customers who would buy from the } \textit{old} \text{ website}\} \\
\mathcal{B} &= \{\text{customers who would buy from the } \textit{new} \text{ website}\} \\
f^{\mathrm{A}} &= \frac{|\mathcal{A}|}{|C|} = \text{fraction of customers who would buy from the } \textit{old} \text{ website} \\
f^{\mathrm{B}} &= \frac{|\mathcal{B}|}{|C|} = \text{fraction of customers who would buy from the } \textit{new} \text{ website} \\
f^{\mathrm{B}} - f^{\mathrm{A}} &= \text{the improvement of the new website}
\end{aligned}$$

[1]Instead of using this trick, we could instead use Theorem 9.2.6 to handle the case of $\delta > 1$.

Algorithm 11.1 The A/B testing algorithm. We assume that $m = |C|$ is even.

```
function ABTest(array C)
    Let C^A, C^B ← PartitionWithoutReplacement(C, 2)        ▷ See Algorithm 3.5
    Let X^A ← 0                     ▷ Number of customers who bought from old website
    for c ∈ C^A
        Show user c the old website
        if user c purchases then X^A ← X^A + 1              ▷ This happens if c ∈ 𝒜
    Let X^B ← 0                     ▷ Number of customers who bought from new website
    for c ∈ C^B
        Show user c the new website
        if user c purchases then X^B ← X^B + 1              ▷ This happens if c ∈ ℬ
    return f̂^A = (2/m) X^A and f̂^B = (2/m) X^B
end function
```

If the improvement $f^{\mathrm{B}} - f^{\mathrm{A}}$ is positive, it would make sense to switch to the new website. However, we typically cannot compute the improvement exactly because we cannot show the same customer *both* versions of the website. The idea of A/B testing is to estimate both f^{A} and f^{B} using random sampling. However, this sampling requires some care: each customer can either be used to estimate f^{A} or f^{B} *but not both*. This is accomplished by the ABTest algorithm, shown in Algorithm 11.1.

The ABTest algorithm has one key idea: randomly partition the set C of customers into two groups, C^{A} and C^{B}, using the PartitionWithoutReplacement algorithm from Section 3.3 with $\ell = 2$. This ensures that each group has size exactly $m/2$, and that each customer appears in exactly one of the groups. The algorithm then simply polls each of the two groups to estimate the fraction who would buy from the respective websites. These estimates are:

$$\begin{aligned} \hat{f}^{\mathrm{A}} &= \frac{2}{m} X^{\mathrm{A}} = \frac{|C^{\mathrm{A}} \cap \mathcal{A}|}{|C^{\mathrm{A}}|} = \text{fraction of group } C^{\mathrm{A}} \text{ who would buy from old website} \\ \hat{f}^{\mathrm{B}} &= \frac{2}{m} X^{\mathrm{B}} = \frac{|C^{\mathrm{B}} \cap \mathcal{B}|}{|C^{\mathrm{B}}|} = \text{fraction of group } C^{\mathrm{B}} \text{ who would buy from new website.} \end{aligned}$$

Notice that these estimators have the same form as the polling estimator (4.3) discussed in Section 4.3. We can analyze these estimates by considering the PollWithoutReplacement algorithm from Section 4.3.2. We claim that

$$\begin{aligned} &\hat{f}^{\mathrm{A}} \text{ has the same distribution as } \textsc{PollWithoutReplacement}(C, m/2, \mathcal{A}) \\ &\hat{f}^{\mathrm{B}} \text{ has the same distribution as } \textsc{PollWithoutReplacement}(C, m/2, \mathcal{B}). \end{aligned}$$

This follows from Claim 3.3.2, which states the mysterious property of partitioning without replacement: both C^{A} and C^{B} have the same distribution as SampleWithoutReplacement(C, k).

Theorem 11.3.1. Suppose there are $m \geq 5000$ customers. With probability 0.95, the estimated improvement $\hat{f}^{\mathrm{B}} - \hat{f}^{\mathrm{A}}$ differs from the true improvement $f^{\mathrm{B}} - f^{\mathrm{A}}$ by at most 0.06.

Proof. The proof just applies the analyses of Section 11.2. Whereas that section had m fixed and sought to minimize k, in this section we have $k = m/2$ fixed and we want to minimize m. Plugging into (11.2.2) with $k = m/2$ and $m = 5000$, we get

$$\begin{aligned}
\Pr\left[\,|\hat{f}^{\mathrm{A}} - f^{\mathrm{A}}| \geq 0.03\right] &\leq 2\exp(-2\cdot(0.03)^2\cdot k) \;<\; 0.025\\
\Pr\left[\,|\hat{f}^{\mathrm{B}} - f^{\mathrm{B}}| \geq 0.03\right] &\leq 2\exp(-2\cdot(0.03)^2\cdot k) \;<\; 0.025.
\end{aligned}$$

By a union bound, there is a small probability that either estimate is inaccurate.

$$\Pr\left[\,|\hat{f}^{\mathrm{A}} - f^{\mathrm{A}}| \geq 0.03 \;\vee\; |\hat{f}^{\mathrm{B}} - f^{\mathrm{B}}| \geq 0.03\right] \;<\; 0.05$$

Taking the complement, there is a large probability that both estimates are accurate.

$$\Pr\left[\,|\hat{f}^{\mathrm{A}} - f^{\mathrm{A}}| < 0.03 \;\wedge\; |\hat{f}^{\mathrm{B}} - f^{\mathrm{B}}| < 0.03\right] \;>\; 0.95$$

Assuming this event occurs, we have the following bound on the error.

$$\begin{aligned}
|(\text{Estimated improvement}) - (\text{True improvement})| &= \left|(\hat{f}^{\mathrm{B}} - \hat{f}^{\mathrm{A}}) - (f^{\mathrm{B}} - f^{\mathrm{A}})\right|\\
&\leq \left|\hat{f}^{\mathrm{A}} - f^{\mathrm{A}}\right| + \left|\hat{f}^{\mathrm{B}} - f^{\mathrm{B}}\right|\\
&< 0.03 + 0.03 \;=\; 0.06,
\end{aligned}$$

where the first inequality is the triangle inequality, Fact A.2.4. □

Applying the theorem. How can we use Theorem 11.3.1 in practice? One option is to switch to using the new website if $\hat{f}^{\mathrm{B}} - \hat{f}^{\mathrm{A}} \geq 0.06$. With this rule, we have the following possibilities.

- **Sampling fails.** This happens with probability ≤ 0.05, in which case anything can happen.
- **Sampling succeeds.** This happens with probability ≥ 0.95. The following cases hold.
  - **True improvement is large.** If $f^{\mathrm{B}} - f^{\mathrm{A}} \geq 0.12$ then $\hat{f}^{\mathrm{B}} - \hat{f}^{\mathrm{A}} \geq 0.06$, so we definitely switch to the new website.
  - **True improvement is modest.** If $0 \leq f^{\mathrm{B}} - f^{\mathrm{A}} < 0.12$ then the estimated improvement might or might not exceed 0.06.
  - **True improvement is negative.** If $f^{\mathrm{B}} - f^{\mathrm{A}} < 0$ then $\hat{f}^{\mathrm{B}} - \hat{f}^{\mathrm{A}} < 0.06$, so we definitely *don't* switch to the new website.

11.4 Estimating Distributions

11.4.1 Bernoulli distributions

As suggested by Exercise 4.2, polling via sampling with replacement is essentially the same problem as estimating the parameter p of a Bernoulli(p) distribution. Let us explain this connection in some more detail.

Suppose we draw i.i.d. samples $X_1, \ldots, X_k$ from a Bernoulli(p) distribution. A natural estimator for p is the sample mean

$$\hat{p} \;=\; \sum_{i=1}^{k} X_i/k.$$

Statisticians are fond of this estimator because it is the *maximum likelihood estimator* (Larsen and Marx, 2018, Example 5.1.1). We will not adopt that viewpoint and instead will analyze the estimator directly using Hoeffding's inequality.

Theorem 11.4.1. Let the number of samples be $k = \ln(200)/2\epsilon^2$. Then $\hat{p}$ is

$$\begin{aligned} \text{Unbiased:} &\qquad \mathrm{E}[\,\hat{p}\,] \;=\; p \\ \text{Concentrated:} &\qquad \Pr[\,|\hat{p} - p| \geq \epsilon\,] \;\leq\; 1/100. \end{aligned}$$

Proof. The unbiasedness follows from $\mathrm{E}[\,X_i\,] = p$ and linearity of expectation.

$$\mathrm{E}[\,\hat{p}\,] \;=\; \sum_{i=1}^{k} \mathrm{E}[\,X_i\,]/k \;=\; k \cdot p/k \;=\; p.$$

The concentration follows from Hoeffding for averages (Theorem 9.6.1).

$$\begin{aligned} \Pr[\,|\hat{p} - p| \geq \epsilon\,] \;&\leq\; 2\exp(-2\epsilon^2 k) \\ &=\; 2\exp\Big(-2\epsilon^2 \cdot \frac{\ln(200)}{2\epsilon^2} \Big) \\ &=\; 2\exp\big(-\ln(200) \big) \;=\; 1/100 \end{aligned}$$

□

11.4.2 Finite distributions

Consider any distribution on a finite set; we may assume that set to be $[k]$ for simplicity. The distribution has parameters $p_1, \ldots, p_k \geq 0$ which satisfy $\sum_{j=1}^{k} p_j = 1$. A random variable X with this distribution has $p_i = \Pr[\,X = i\,]$. We would like to produce estimates

$$\hat{p}_1, \ldots, \hat{p}_k \in [0,1]$$

that are within ϵ of the true probabilities, meaning

$$|p_j - \hat{p}_j| < \epsilon \quad \forall j \in [k].$$

Suppose we draw i.i.d. samples $X_1, \ldots, X_k$ from this distribution. The natural estimator for p_j is simply the fraction of samples that equal j.

$$\hat{p}_j \;=\; \frac{\#\text{ samples that equal } j}{k}.$$

Note that these estimates also satisfy $\sum_{i=1}^{k} \hat{p}_i = 1$.

Theorem 11.4.2. Let the number of samples be $k = \ln(200k)/2\epsilon^2$. Then $\hat{p}$ is

$$\begin{aligned} \text{Unbiased:} &\qquad \mathrm{E}[\,\hat{p}_j\,] \;=\; p_j \;\forall j \\ \text{Concentrated:} &\qquad \Pr[\,\text{any } j \text{ has } |\hat{p}_j - p_j| \geq \epsilon\,] \;\leq\; 1/100. \end{aligned}$$

Proof. First we consider the unbiased property. Fix any $j \in [k]$. Let X_i be the indicator of the event that the i^{th} sample equals j. Then $\mathrm{E}[\,X_i\,] = p_j$, so

$$\mathrm{E}[\,\hat{p}_j\,] \;=\; \mathrm{E}\left[\sum_{i=1}^{k} X_i/k \right] \;=\; \sum_{i=1}^{k} \mathrm{E}[\,X_i\,]/k \;=\; k \cdot p_j/k \;=\; p_j.$$

Next we consider the concentration.

$$\Pr[\,|\hat{p}_j - p_j| \geq \epsilon\,] \;\leq\; 2\exp(-2\epsilon^2 k) \qquad \text{(by Theorem 9.6.1)}$$
$$= \; 2\exp\Big(-2\epsilon^2 \cdot \frac{\ln(200k)}{2\epsilon^2}\Big) \;=\; \frac{1}{100k}.$$

Thus, by a union bound,

$$\Pr[\,\text{any } j \text{ has } |\hat{p}_j - p_j| \geq \epsilon\,] \;\leq\; \sum_{j=1}^{k} \Pr[\,|\hat{p}_j - p_j| \geq \epsilon\,] \;\leq\; k \cdot \frac{1}{100k} \;=\; \frac{1}{100}. \qquad \square$$

★11.4.3 Estimating the cumulative distribution

Consider any distribution, and let F be its CDF. We would like to estimate F using a function $\hat{F}$ that satisfies

$$\max_{x \in \mathbb{R}} |\hat{F}(x) - F(x)| \;\leq\; \epsilon. \tag{11.4.1}$$

This is a very strong condition: $\hat{F}$ has to approximate F *at all points* x. The left-hand side of (11.4.1) has a rather grandiose name: it is called the Kolmogorov-Smirnov statistic.

Suppose we draw s samples from the distribution. Since $F(x)$ is the probability that a sample is $\leq x$, it is natural to define $\hat{F}(x)$ to be the empirical fraction of samples that are $\leq x$.

$$\hat{F}(x) \;=\; \frac{\#\text{ samples that are} \leq x}{s}$$

An amazing theorem holds for this estimator. With *no assumptions* on F at all, the number of samples needed for $\hat{F}$ to approximate F depends only on ϵ.

Theorem 11.4.3 (DKW Theorem).

$$\Pr[\,(11.4.1) \text{ fails to hold}\,] \;\leq\; 2\exp(-2\epsilon^2 s). \tag{11.4.2}$$

Consequently, taking $s = \ln(200)/2\epsilon^2$ samples ensures that (11.4.1) holds with probability at least 0.99.

References: Wikipedia.

The bound in (11.4.2) is *optimal*; even the constants of 2 cannot be improved! Proving such a strong result is challenging, but we can prove a slightly weaker result using only Hoeffding's bound.

Theorem 11.4.4 (Weak DKW Theorem). Assume that F is continuous and strictly increasing. Then

$$\Pr[\,(11.4.1) \text{ fails to hold}\,] \;\leq\; \frac{4}{\epsilon}\exp(-\epsilon^2 s/2). \tag{11.4.3}$$

Thus, taking $s = 2\ln(400/\epsilon)/\epsilon^2$ samples ensures that (11.4.1) holds with probability at least 0.99.

Proof. For simplicity, assume that $\epsilon = 1/q$, where $q \geq 2$ is an integer. Before trying to show $\hat{F}(x) \approx F(x)$ for arbitrary x, we focus our attention on certain quantiles, namely

$$z_0 = -\infty \qquad\qquad z_i = F^{-1}(i/q) \quad \forall i \in [q-1] \qquad\qquad z_q = +\infty.$$

The main idea is to show that $\hat{F}(z_i) \approx F(z_i)$ for all i. To do so, we define the bad events

$$\mathcal{E}_i \;=\; \Big\{ |\hat{F}(z_i) - F(z_i)| \geq \epsilon \Big\} \qquad \forall i \in [q].$$

Observe that $\hat{F}(z_i)$ is the empirical average of s Bernoulli RVs, each of which has mean $F(z_i)$. So, by Hoeffding for averages (Theorem 9.6.1),

$$\Pr[\, \mathcal{E}_i \,] \leq 2\exp(-2\epsilon^2 s).$$

The number of these events is $q = 1/\epsilon$, so a union bound shows that

$$\Pr[\, \mathcal{E}_1 \vee \cdots \vee \mathcal{E}_q \,] \;\leq\; \frac{2}{\epsilon}\exp(-2\epsilon^2 s). \tag{11.4.4}$$

Now assuming that none of the bad events occurs, we show that $\hat{F}$ approximates F well. Consider any $x \in \mathbb{R}$. For some $i \in [q]$, the quantiles define an interval containing x of the form $z_{i-1} \leq x < z_i$. The upper bound is argued as follows.

$$\begin{aligned}
\hat{F}(x) - F(x) \;&\leq\; \hat{F}(z_i) - F(z_{i-1}) && \text{(since } z_{i-1} \leq x < z_i\text{)}\\
&=\; \big(\hat{F}(z_i) - F(z_i)\big) + \big(F(z_i) - F(z_{i-1})\big)\\
&<\; \epsilon + \big(i/q - (i-1)/q\big) && \text{(since } \mathcal{E}_i \text{ does not occur, and by definition of } z_i\text{)}\\
&=\; 2\epsilon && \text{(since } \epsilon = 1/q\text{)}
\end{aligned}$$

The lower bound is similar.

$$\begin{aligned}
\hat{F}(x) - F(x) \;&\geq\; \hat{F}(z_{i-1}) - F(z_i) \qquad \text{(since } z_{i-1} \leq x < z_i\text{)}\\
&=\; \big(\hat{F}(z_{i-1}) - F(z_{i-1})\big) + \big(F(z_{i-1}) - F(z_i)\big)\\
&>\; -2\epsilon \qquad \text{(since } \mathcal{E}_{i-1} \text{ does not occur)}
\end{aligned}$$

In summary, assuming the bad events do not occur, we have shown that

$$|\hat{F}(x) - F(x)| \;<\; 2\epsilon \qquad \forall x \in \mathbb{R}.$$

This condition becomes (11.4.1) if we replace ϵ with $\epsilon/2$. Recall that the probability of any bad event occuring is bounded by (11.4.4). So with the adjusted value of ϵ, we obtain (11.4.3). □

11.5 Exercises

Exercise 11.1 Binary search with errors. Let $A = [a_1, \ldots, a_n]$ be a sorted array of distinct elements. We have a function $\textsc{Cmp}(x, y)$ that can compare any two elements; it returns either 0, meaning "$x < y$", or 1, meaning "$x > y$". Unfortunately the function is randomized, and it returns the wrong answer with probability 1/3.

You would like to insert a new element z (not already present in A) into its correct position in this ordering. Show that this can be done by a modified binary search that makes $O\big(\log(n)\log\log(n)\big)$ calls to $\textsc{Cmp}$. Your algorithm should return the correct position with probability at least 0.99.

Exercise 11.2. Let ϵ and α satisfy $0 < \epsilon < 1$ and $0 < \alpha < 1$. Suppose we wanted an estimate that was accurate to within ϵ with failure probability α. Show that it suffices to poll $O(\log(1/\alpha)/\epsilon^2)$ customers, in expectation.

Exercise 11.3. Instead of comparing just two versions of the website, suppose that we want to compare n versions of the website. For $i \in [n]$, let f^i be the fraction of users who would buy from website i. Let $\mathrm{OPT} = \max_{i\in[n]} f^i$ be the fraction of buyers from the best website.

Give an algorithm to identify a website i^* satisfying $f^{i^*} \geq \mathrm{OPT} - \epsilon$. Of course, you can only show each customer one website. Your algorithm should use $O(n\log(n)/\epsilon^2)$ customers and should succeed with probability 0.95.

Exercise 11.4. If $k \gg 1/\epsilon$ then Theorem 11.4.2 can be improved. Using the same estimator, show that $O(\log(1/\epsilon)/\epsilon^2)$ samples suffice to obtain

$$\Pr[\,\text{any } j \text{ has } |\hat{p}_j - p_j| \geq \epsilon\,] \;\leq\; 1/100.$$

Exercise 11.5. The DKW theorem (Theorem 11.4.3) implies Hoeffding for averages (inequality (9.6.1)), at least in the case where the X_i RVs are i.i.d.

Part I. Prove this in the case where the X_i are independent Bernoulli(p).

Part II. Prove this in the case where the X_i are i.i.d. random variables satisfying $0 \leq X_i \leq 1$.

Exercise 11.6. The weakest aspect of Theorem 11.4.4 is the constant in the exponent: whereas (11.4.3) has a constant of $-1/2$, (11.4.2) has a better constant of -2. Fortunately, this can be improved without too much effort. Prove that

$$\Pr[\,(11.4.1) \text{ fails to hold}\,] \;\leq\; O(1/\epsilon)\cdot\exp(-1.99\epsilon^2 s).$$

Exercise 11.7. Prove Theorem 11.4.4 without assuming that F is continuous or strictly increasing.

Hint: Redefine z_i to account for the fact that F might not be invertible, then carefully consider the left and right limits at those points.

Part III

Hashing

Chapter 12

Introduction to Hashing

12.1 Various types of hash functions

Hash functions are extremely useful in algorithm design. Unfortunately they are a topic in which theory and practice diverge quite substantially. There is a wide range of hash functions, from very practical ones lacking any theory, to very theoretical ones that are impractical. The following list discusses several categories of hash functions.

- Many applications, such as the Python dictionary, use a single hash function only that "seems" random. A modern example is Murmur.
  Advantages: These functions are easy to design and implement, and fast to execute.
  Disdvantages: Collisions are common, and can be caused maliciously. Since there is just one function, any two colliding inputs will *always* collide. They usually produce only a few bits of output, say 32 bits.
  Theory perspective: Not much can be proven about them.
  Stance on collisions:

  > Things will collide, but hopefully not often, because that'll slow us down.

- We can mitigate the predictability of the previous hash functions by injecting some randomness into the keys. This can help to avoid deliberate collisions caused by adversaries. This has been done in practice, e.g., in the Linux Netfilter.
  Advantages: Easy to implement and fast.
  Disadvantages: Crafty adversaries might still be able to cause collisions.
  Theory perspective: It is conceivable to prove something about them, although that's not usually the goal.
  Stance on collisions:

  > A dash of randomness might help avoid deliberate collisions.

- A single function that is judged to be "cryptographically secure", e.g., MD5 or SHA-1.
  Advantages: Collisions are quite difficult to arrange. They produce many bits of output, say 128-256 bits.
  Disdvantages: They are relatively slow to execute. Since there is just one function, any two colliding inputs will *always* collide. Originally they may be deemed secure, but eventually they

will likely be broken, as is now true for both MD5 and SHA-1.
Theory perspective: People prove things about them, usually focusing on the difficulty or ease of finding collisions. It doesn't make sense to use them in probabilistic proofs, because they are not at all random.
Stance on collisions:

> It'll take decades of research and supercomputer time to find collisions.

- A simple function designed with small amounts of randomness. The prototypical example of this is ***universal hashing***, which we will discuss in Chapter 14.
Advantages: We can prove results about the probability of collisions. They can actually be implemented.
Disdvantages: They might require some abstract algebra in their design and implementation. They may be slightly slower than Murmur, say.
Theory perspective: They are just random enough that they are useful in some applications and we can prove some things about them.
Stance on collisions:

> We can provably make simple collisions unlikely.

- Use a purely random function.
Advantages: Very convenient for proofs.
Disadvantages: An implementation would be absurdly inefficient.
Theory perspective: They are so random that they enable fantastic applications, and we can prove strong statements about them.
Stance on collisions:

> We can make all collisions as unlikely as possible.

12.1.1 What is a purely random function?

Purely random function were first briefly discussed on page 78. Now we will discuss this concept in more detail. Suppose we are interested in a hash function $h : [\![n]\!] \rightarrow [\![k]\!]$, which maps the domain $[\![n]\!]$ to the codomain $[\![k]\!]$. There are two equivalent definitions.

The first definition. Algorithm 12.1 presents the first approach for constructing a purely random function. Since there are no constraints on the values that are generated, all k^n different combinations are possible. So the number of different functions that can be constructed is k^n, and they are all equally likely.

The second definition. Let $\mathcal{H}$ be the set of *all* functions that map $[\![n]\!]$ to $[\![k]\!]$. We can write this as

$$\mathcal{H} \;=\; \{\ \text{function } h \,:\, h \text{ maps } [\![n]\!] \text{ to } [\![k]\!]\ \}\,.$$

We can also think of a purely random function as a function $h \in \mathcal{H}$ chosen uniformly at random. We have argued above that there are k^n different functions that h could be, so $|\mathcal{H}| = k^n$.

Algorithm 12.1 A class PurelyRandom that implements a purely random function whose domain is $[\![n]\!]$ and codomain is $[\![k]\!]$.

```
class PurelyRandom:
    int hashValues[]

    ▷ The constructor.
    Constructor (int n, int k)
    |   for i = 0, ..., n − 1
    |   |   hashValues[i] ← UniformInt(k)

    ▷ Return the hash value for input x.
    HashValue ( int x )
    |   return hashValues[x]
```

12.1.2 Measuring space complexity

When measuring the amount of space used by an object, there are two common conventions regarding the units of space.

- One convention is to count the number of *bits*. In this case, each unit of space can store only two different values.
- The other common convention is to count the number of *words*. It is assumed that each word can store at least n different values, where n is some understood parameter reflecting the size of the problem. This means that each word must use $\Omega(\log n)$ bits.

For example, let us consider a binary tree data structure. In introductory data structure courses, it is often stated that a binary tree with n node takes $O(n)$ space. This statement implicitly assumes the second convention, in which we measure *words* of space, not bits. This is because the keys must represent n distinct values, and the pointers must point to n distinct locations.

Next let us consider the space required to represent a purely random function h mapping $[\![n]\!]$ to $[\![k]\!]$. We will use the first convention, in which we measure *bits* of space. Since h can be any member of the set $\mathcal{H}$, the answer is provided by Fact A.2.10. The number of bits that are required to represent a function $h : [\![n]\!] \to [\![k]\!]$ is

$$\lceil \lg|\mathcal{H}| \rceil \;=\; \lceil \lg(k^n) \rceil \;\leq\; n \lg(k) + 1.$$

Question 12.1.1. Let h be a purely random function producing outputs in $[\![k]\!]$. For any distinct inputs x, y, what is $\Pr[\, h(x) = h(y) \,]$?

Answer.

By the law of total probability (Fact A.3.6), it is

$$\begin{aligned}\sum_{i \in [\![k]\!]} \Pr[\, h(x) = h(y) = i \,] &= \sum_{i \in [\![k]\!]} \Pr[\, h(x) = i \,\wedge\, h(y) = i \,] \\ &= \sum_{i \in [\![k]\!]} \Pr[\, h(x) = i \,] \cdot \Pr[\, h(y) = i \,] \\ &= \sum_{i \in [\![k]\!]} (1/k)^2 \;=\; 1/k.\end{aligned}$$

12.1.3 What hash function should we use?

As a computer scientist, how should one cope with these various categories of hash functions? Assuming that there are no security considerations, the following approach seems reasonable. One can first design algorithms assuming that a purely random function is available. To implement the algorithm, one could replace the purely random function with a fast, seemingly-random function. This will no longer have provable guarantees, but will likely work well in most applications. Alternatively, to obtain an algorithm that is both implementable and has provable guarantees, one must figure out how to replace the purely random function with a low-randomness hash function. This is the topic of Chapter 14.

12.2 MinHash: estimating set similarity

A common data analysis problem is to measure the similarity of two sets. For a very basic example, consider the following two sets containing keywords that describe Avocados and Brussels Sprouts.

$$\begin{aligned} A &= \{\text{green, seed, mushy}\} \\ B &= \{\text{green, small, bitter}\}\,. \end{aligned}$$

One way to judge the similarity of Avocados and Brussels Sprouts is to judge the similarity of the sets A and B. How can we make this mathematically precise? Here is one reasonable definition.

Definition 12.2.1. The similarity of sets A and B is

$$\mathrm{Sim}(A,B) \;=\; \frac{|A \cap B|}{|A \cup B|},$$

assuming that $A \cup B \neq \emptyset$.

References: Wikipedia.

Question 12.2.2. How can you compute $\mathrm{Sim}(A,B)$ exactly? How much time does it take if A and B have size n?

In this section we present the MinHash algorithm which preprocesses sets to enable very efficient estimation of set similarity.

12.2.1 The basic estimator

Suppose we have a purely random function h that can map any string to a uniform *real number* in the interval $[0,1]$. The benefit of hashing to real numbers is that we don't have to worry about collisions: for any two string a and b, we have

$$\Pr[\, h(a) = h(b) \,] \;=\; 0.$$

Question 12.2.3. Why does this hold?

Answer.

Since h is assumed to be a purely random function, the values $h(a)$ and $h(b)$ are independent. Since they are uniform on $[0,1]$, Fact A.3.16 tells us that they have zero probability of being equal.

Practically speaking, such a hash function may seem impractical. However, it can easily be simulated by hashing to a large universe $[U] = \{1, \ldots, U\}$. If U is sufficiently large, then any hash collisions in $A \cup B$ can be made arbitrarily unlikely.

Algorithm 12.2 The MinHashBasic class implements a data structure that allows efficient estimation of set similarity.

global hash function h

class MinHashBasic:
  float minValue

  ▷ The constructor. E is the set of entries to insert.
  Constructor (strings E)
    minValue $\leftarrow \min\{\, h(e) \,:\, e \in E \,\}$

EstimateSim (MinHashBasic A, MinHashBasic B) returns float
  if A.minValue $=$ B.minValue
    return 1
  else
    return 0

Claim 12.2.4. Suppose that $U \geq 50 \cdot |A \cup B|^2$. The probability of any hash collisions among the elements in $A \cup B$ is at most $1/100$.

Question 12.2.5. Can you prove this?

Answer.

This is related to the birthday paradox. We apply Exercise 8.1 with $m = |A \cup B|$, $n = U$, and $k = 50$. The probability of any collisions is at most $1/2k = 1/100$.

The MinHash algorithm, shown in Algorithm 12.2, is extremely simple. First it randomly chooses a hash function h, which will be used by all sets. Then, for each set, only the minimum hash value is stored.

The space usage of this data structure seems amazing: it seems to represent each set using only *constant* space. Let's think through that more carefully. Technically speaking, a random real number would require an infinite number of bits to represent. However, for sets of size n, Claim 12.2.4 tells us that we can pick hash values in $[U]$ with $U = O(n^2)$, and therefore only $O(\log n)$ bits are needed for each hash value. For concreteness, we will represent the minimum hash value as a float, which is just a single word of space.

The accuracy of this data structure is also amazing. Although it appears to do very little, in expectation it exactly determines the set similarity. In other words, it is an unbiased estimator.

Claim 12.2.6. $\mathrm{E}[\,\textsc{EstimateSim}(A, B)\,] = \mathrm{Sim}(A, B)$.

Proof. Consider the minimum hash value amongst all elements in $A \cup B$. There are three possibilities.

- The minimum lies in $A \setminus B$. Then A.minValue $\neq$ B.minValue.
- The minimum lies in $B \setminus A$. Then A.minValue $\neq$ B.minValue.
- The minimum lies in $A \cap B$. Then A.minValue $=$ B.minValue.

So EstimateSim(A, B) returns 1 precisely when the minimum hash value among $A \cup B$ happens to lie in $A \cap B$. Since h is a purely random function, every string in $A \cup B$ has equal likelihood of having the minimum hash value. So

$$\mathrm{E}\left[\,\textsc{EstimateSim}(A,B)\,\right] \;=\; \Pr\left[\,A.\text{minValue} = B.\text{minValue}\,\right] \;=\; \frac{|A\cap B|}{|A\cup B|} \;=\; \mathrm{Sim}(A,B). \tag{12.2.1}$$

□

Question 12.2.7. Do you see where we have implicitly assumed that there are no hash collisions?

Answer.

The proof assumes that the number of distinct hash values is $|A \cup B|$. So there is a unique minimum hash value, which is equally likely to come from any element of $A \cup B$.

The unbiasedness is certainly a desirable feature, but EstimateSim is still too rudimentary. Recalling our moral from Section 8.1,

> the expectation of a random variable does not tell you everything.

To illustrate this, consider the example $A = \{\text{apple}, \text{banana}\}$ and $B = \{\text{banana}\}$, so $\mathrm{Sim}(A,B) = 0.5$. The output of $\textsc{EstimateSim}(A,B)$ can only be 0 or 1, so it *always* has error exactly 0.5. The problem is that $\textsc{EstimateSim}(A,B)$ is *not concentrated*.

12.2.2 The averaged estimator

To improve the estimator, we should make its output more concentrated. The natural way to accomplish this is to average several basic estimators. The first step is to randomly choose k independent purely random functions $h_1,\dots,h_k$. Then we define a Signature to be an array of k of these minimum hash values. The similarity of two sets is estimated using the fraction of matching values in their signatures. Pseudocode is shown in Algorithm 12.3.

Analysis. Let X_i be the indicator of whether the i^{th} hash values match.

$$X_i \;=\; \begin{cases} 1 & (\text{if } A.\text{sig.minValue}[i] = B.\text{sig.minValue}[i]) \\ 0 & (\text{otherwise}). \end{cases}$$

The X_i's are independent since $h_1,\dots,h_k$ are independent. We have already shown in (12.2.1) that $\mathrm{E}\left[\,X_i\,\right] = \mathrm{Sim}(A,B)$.

EstimateSim is defined to return the average of the X_i's. That is,

$$\textsc{EstimateSim}(A,B) \;=\; \frac{X}{k} \;=\; \frac{1}{k}\sum_{i=1}^{k} X_i. \tag{12.2.2}$$

By linearity of expectation,

$$\mathrm{E}\left[\,\textsc{EstimateSim}(A,B)\,\right] \;=\; \frac{1}{k}\sum_{i=1}^{k}\mathrm{E}\left[\,X_i\,\right] \;=\; \mathrm{Sim}(A,B). \tag{12.2.3}$$

The purpose of averaging k basic estimators is to ensure that $\textsc{EstimateSim}(A,B)$ is much better concentrated around its expectation.

Theorem 12.2.8. Let $\epsilon > 0$ be arbitrary. Suppose we set $k = \left\lceil \ln(200)/2\epsilon^2 \right\rceil$. Then

$$\Pr\left[\,|\textsc{EstimateSim}(A,B) - \mathrm{Sim}(A,B)| \geq \epsilon\,\right] \leq 0.01.$$

Algorithm 12.3 The class MinHash improves on MinHashBasic by using k independent hash functions.

```
global int k
global hash functions h_1, ..., h_k

struct Signature:
    float minValue[1..k]

class MinHash:
    Signature sig

    ▷ The constructor. E is the set of entries to insert.
    Constructor ( strings E )
        for i = 1, ..., k
            sig.minValue[i] ← min { h_i(e) : e ∈ E }

EstimateSim (MinHash A, MinHash B) returns float
    X ← 0        ▷ The number of matching hash values in their signatures
    for i = 1, ..., k
        if A.sig.minValue[i] = B.sig.minValue[i] then
            X ← X + 1
    return X/k        ▷ The fraction of matching hash values
```

Proof. As shown in (12.2.2), $\textsc{EstimateSim}(A, B)$ is an average of k independent indicator RVs. Let us write $Y = \textsc{EstimateSim}(A, B)$ to simply notation. By (12.2.3), we have $\mathrm{E}[\,Y\,] = \mathrm{Sim}(A, B)$. Now we can directly apply Hoeffding for averages.

$$\begin{aligned}
\Pr[\,|\textsc{EstimateSim}(A,B) - \mathrm{Sim}(A,B)| \geq \epsilon\,] &= \Pr[\,|Y - \mathrm{E}[\,Y\,]| \geq \epsilon\,] \\
&\leq 2\exp(-2\epsilon^2 k) \qquad \text{(by Theorem 9.6.1)} \\
&= 2 \cdot \frac{1}{200} \;=\; 0.01
\end{aligned}$$

by plugging in our value of k. ∎

Broader context. The MinHash algorithm was originally developed for use in AltaVista, one of the original web search engines. This work was given the ACM Kanellakis prize in 2012. MinHash was originally intended for duplicate detection of web pages, but it has found many subsequent uses, including Google News, genomics, etc.

12.3 MinHash: near neighbours

Zoodle's customers are interested in searching for vegetables matching various criteria. Zoodle has prepared a list of many vegetables and their various properties. For example,

$$\begin{aligned}
\text{Avocado} &= \{\text{green}, \text{big seed}, \text{mushy}\} \\
\text{Beet} &= \{\text{red}, \text{sweet}, \text{needs boiling}\}
\end{aligned}$$

$$\begin{aligned}
\text{Cabbage} &= \{\text{green}, \text{large}, \text{leafy}\}\\
\text{Carrot} &= \{\text{orange}, \text{long}, \text{crunchy}\}\\
\text{Pea} &= \{\text{green}, \text{small}, \text{mushy}\}
\end{aligned}$$

This list will be preprocessed into a data structure so that customers can rapidly search for similar vegetables. For example, a customer might submit the information

$$\text{Brussels Sprout} \;=\; \{\text{green}, \text{small}, \text{leafy}\}\,.$$

From Zoodle's list, both Cabbage and Pea have similar[1] properties to the Brussels Sprout; in fact

$$\mathrm{Sim}(\text{Brussels Sprout}, \text{Cabbage}) \;=\; \mathrm{Sim}(\text{Brussels Sprout}, \text{Pea}) \;=\; \frac{2}{4} \;=\; 0.5.$$

So it seems reasonable for Zoodle to return the list [Cabbage, Pea] to the customer.

The general goal is to design a data structure that can preprocess a collection of sets. Given a query set Q, it will return several sets that are hopefully similar to Q. We will call this data structure NearNeighbour, because these queries can be viewed as a "near neighbour" query on the given sets.

The main idea is to generate ℓ independent signatures for each set. Then ℓ different dictionaries are used to identify sets that have a matching signature. The operation Query(Q) simply returns all sets in the data structure for which at least one signature matches Q's signature. Pseudocode is shown in Algorithm 12.4.

The main result is that the Query operation is very likely to output sets that are similar to Q and very unlikely to output sets that are dissimilar from Q.

Theorem 12.3.1. Consider sets A, B in the NearNeighbour data structure, and a query set Q.

$$\begin{array}{rllll}
\text{Similar sets:} & \text{if } \mathrm{Sim}(Q,A) \geq 0.9 & \text{then} & \Pr[\,\textsc{Query}\ (Q) \text{ outputs } A\,] > 0.99 & (12.3.1)\\
\text{Dissimilar sets:} & \text{if } \mathrm{Sim}(Q,B) \leq 0.5 & \text{then} & \Pr[\,\textsc{Query}\ (Q) \text{ outputs } B\,] < 0.01. & (12.3.2)
\end{array}$$

To prove this, we will first need the following lemma.

Lemma 12.3.2. For any two sets A and B and any $i \in [\ell]$,

$$\Pr[\,A.\mathrm{sig}[i] = B.\mathrm{sig}[i]\,] \;=\; \mathrm{Sim}(A,B)^k.$$

Proof.

$$\begin{aligned}
&\Pr[\,A.\mathrm{sig}[i] = B.\mathrm{sig}[i]\,]\\
&\quad= \Pr[\,A.\mathrm{sig}[i].\mathrm{minValue}[j] = B.\mathrm{sig}[i].\mathrm{minValue}[j]\ \forall j \in [k]\,]\\
&\quad= \prod_{j=1}^{k} \Pr[\,A.\mathrm{sig}[i].\mathrm{minValue}[j] = B.\mathrm{sig}[i].\mathrm{minValue}[j]\,] \qquad \text{(by independence)}\\
&\quad= \prod_{j=1}^{k} \mathrm{Sim}(A,B) \qquad \text{(by (12.2.1))}\\
&\quad= \mathrm{Sim}(A,B)^k
\end{aligned}$$
□

[1] In fact, Brussels Sprouts and Cabbages are exactly the same species, Brassica oleracea.

Algorithm 12.4 The NearNeighbour data structure for finding similar sets.

```
global int k = 11, ℓ = 13
global hash functions h[1..k, 1..ℓ]

struct Signature:
    float minValue[1..k]

class MinHash:
    Signature sig[1..ℓ]

    ▷ The constructor. E is the set of entries to insert.
    Constructor ( strings E )
        for j = 1, ..., ℓ
            for i = 1, ..., k
                sig[j].minValue[i] ← min { h_{i,j}(e) : e ∈ E }

class NearNeighbour:
    ▷ The i^th table stores all sets, using their i^th signature as the key.
    ▷ Multiple sets with the same key are allowed.
    Dictionary table[1..ℓ]

    ▷ The constructor. mySets is a list of MinHash data structures
    Constructor (MinHash mySets[1..n])
        foreach set in mySets
            for i = 1, ..., ℓ
                ▷ Insert each set into the i^th table, using its i^th signature as the key
                table[i].Insert ( set.sig[i], set )

    Query (MinHash Q) returns list
        Output ← empty list
        for i = 1, ..., ℓ
            ▷ Find all sets whose i^th signature matches Q's, and add them to the output list
            Output.append( table[i].Search (Q.sig[i]) )
        return Output
```

Proof of Theorem 12.3.1. Suppose $\mathrm{Sim}(Q, A) \geq 0.9$. The set A is *not* output if all ℓ of the signatures disagree.

$$\begin{aligned}
\Pr[\,\textsc{Query}\ (Q)\ \textit{doesn't}\ \text{output}\ A\,] &= \Pr[\,Q.\mathrm{sig}[i] \neq A.\mathrm{sig}[i] \quad \forall i \in [\ell]\,] \\
&= \prod_{i=1}^{\ell} \Pr[\,Q.\mathrm{sig}[i] \neq A.\mathrm{sig}[i]\,] \qquad \text{(by independence)} \\
&= \big(1 - \mathrm{Sim}(Q, A)^k\big)^{\ell} \qquad \text{(by Lemma 12.3.2)} \\
&\leq (1 - 0.9^k)^{\ell} \;<\; 0.01,
\end{aligned}$$

by a quick numerical calculation using $k = 11$ and $\ell = 13$. This proves (12.3.1).

Now suppose $\mathrm{Sim}(Q,B) \leq 0.5$. The set B is output if at least one of the ℓ signatures agree.

$$\begin{aligned}
\Pr[\,\textsc{Query}\ (Q) \text{ outputs } B\,] &= \Pr[\,Q.\mathrm{sig}[i] = B.\mathrm{sig}[i] \text{ for some } i \in [\ell]\,] \\
&\leq \sum_{i=1}^{\ell} \Pr[\,Q.\mathrm{sig}[i] = B.\mathrm{sig}[i]\,] \qquad \text{(by union bound)} \\
&= \ell \cdot \mathrm{Sim}(Q,B)^k \qquad \text{(by Lemma 12.3.2)} \\
&\leq \ell \cdot 0.5^k \;<\; 0.01,
\end{aligned}$$

by a quick numerical calculation using $k = 11$ and $\ell = 13$. This proves (12.3.2). □

Broader context. The approach used in this section is a simple example of locality-sensitive hashing (LSH). This was the underlying idea of all the results cited in the ACM Kanellakis prize in 2012. One of the key uses of LSH is in high-dimensional approximate near neighbour search. These technologies are receiving considerable attention from companies like Google, Facebook, Spotify, etc.

12.4 Bloom filters

Many programming languages have a built-in data structure to represent a set. For example, in Python we can create a set and test membership as follows.

```
activistPeople = {'GretaT', 'RachelC', 'DavidS'}
print('GretaT' in activistPeople)        # Outputs True
print('DonaldT' in activistPeople)       # Outputs False
```

Standard Python implementations represent a set using a hash table. The space required by the data structure is linear in the size of the set.

In this section we will imagine representing a set that is so large that the space requirement becomes the primary concern. We will design a randomized data structure and will carefully analyze the number of bits that it requires.

Example 12.4.1. Suppose you are a software engineer at Zoodle designing a web browser called Shrome (an edible fungus). You want to maintain a set of "malicious URLs" that will produce a warning if a user tries to visit one. Each URL might be, say, 30 bytes in length. Say there are 10 million known malicious URLs.

We might store these URLs in a hash table. That's at least 300MB (or 2.4 billion bits) just to represent the strings themselves, ignoring any additional overhead from pointers or the memory allocation system.

Idea: allowing errors. Here is an idea that might improve the space usage. Suppose we allow our data structure to sometimes have false positives — it might accidentally say that something is in the set when it actually isn't. Conceivably we could avoid storing the exact URLs, and thereby save space.

Example 12.4.2. Resuming our example, Shrome will keep this data structure of malicious URLs in memory. When the user tries to load a URL, Shrome will check whether it is in the malicious set. If not, the user is allowed to load the URL. If it is, Shrome will *double-check with a trusted server* whether the URL is actually malicious. If so, the URL is blocked. If not, this is a false positive: the user is allowed to load the URL.

Algorithm 12.5 A basic filter, which uses $100n$ bits to represent a set of size n.

```
class BasicFilter:
    hash function h
    Boolean array B

    ▷ The constructor. Keys is the list of keys to insert.
    Constructor ( strings Keys[1..n] )
    |   Set m ← 100n        ▷ Length of the array
    |   Create array B[1..m] of Booleans, initially all False
    |   Let h be a purely random function
    |   for j = 1, ..., n
    |   |   Set B[h(Keys[j])] ← True

    ▷ Tests whether the string x was inserted. It could make an error.
    IsMember ( string x )
    |   return B[h(x)]
```

This design could be advantageous if the memory savings are substantial, the frequency of false positives is low, and the delay of double-checking is modest.

12.4.1 The basic filters

Suppose we don't bother storing the URLs. Instead, each entry in hash table can just indicate whether *some* string hashed to that location. (Possibly multiple strings might hash to that location.) Then the hash table has just become an array of Booleans; we don't need pointers, or chaining, or any such overhead. Pseudocode is shown in Algorithm 12.5.

Question 12.4.3. What is the probability that IsMember has a false negative, i.e., incorrectly returning False for a key that was passed to Constructor?

Answer.
The probability is zero. If a key was passed to Constructor, then all its necessary bits are set to True. A bit is never changed back from True to False, so IsMember will return True.

Question 12.4.4. What is an upper bound on the probability that IsMember has a false positive, i.e., incorrectly returning True for a key that was *not* passed to Constructor?

Answer.
The table B has at most $n = m/100$ entries that are True, i.e., 1% of its entries. When IsMember is passed a string x that is not in the table, then $h(x)$ is a uniformly random location, independent from all other hash values. It follows that IsMember returns True with probability at most 0.01.

So with this design we have false positive probability at most 0.01, while using only 100 bits per entry (ignoring the space for the purely random function). In our URL example with 10 million URLs, the table B would have 1 billion bits, which is slightly better than storing the URLs explicitly.

Next we will discuss an improved filter that saves even more space.

Algorithm 12.6 A Bloom Filter. It represents a set of size n using only $14n$ bits, and with a false positive probability of 0.01.

```
global int k = ⌈lg(100)⌉ = 7        ▷ Number of hash functions

class BloomFilter:
    hash functions h_1, ..., h_k
    Boolean array B

    ▷ The constructor. Keys is the list of keys to insert.
    Constructor ( strings Keys[1..n] )
        Set m ← 2kn        ▷ Length of the array
        Create array B[1..m] of Booleans, initially all False
        Let h_1, ..., h_k be independent purely random functions
        for j = 1, ..., n
            for i = 1, ..., k
                Set B[h_i(Keys[j])] ← True

    ▷ Tests whether the string x was inserted. There can be false positives.
    IsMember ( string x )
        return B[h_1(x)] ∧ ··· ∧ B[h_k(x)]
```

12.4.2 Bloom Filters

A notable flaw of the BasicFilter is that its array B is mostly empty: at least 99% of its entries are False, which seems quite wasteful. Bloom filters address this flaw with a simple idea:

> use multiple independent hash functions to make false positives unlikely, instead of using a mostly-empty array.

This is in some ways similar to the idea of amplification, in which we use multiple independent trials to decrease false positives. The pseudocode for a Bloom filter is shown in Algorithm 12.6. With this approach, we can keep the array roughly *half empty*, which saves a lot of space.

As with the BasicFilter, it is clear that there are no false negatives. It remains to consider the probability of a false positive. We begin with a preliminary observation.

Observation 12.4.5. In a BloomFilter, at most half of the entries of B are True. To see this, note that the **for** loops set exactly kn of the entries to True, but these entries might coincide. In the rather unlikely worst case, when all those entries are distinct, then we get that at most $nk \leq m/2$ entries are set to True.

Claim 12.4.6. Suppose x *was not* inserted into the Bloom filter. Then, for all $a \in [k]$,

$$\Pr[\, B[h_a(x)] = \mathsf{True} \,] \leq 1/2.$$

Proof. Recall that we have assumed each h_i to be a purely random function. Then, for any distinct keys $x_1, x_2, x_3, \ldots$, the random variables $h_i(x_1), h_i(x_2), h_i(x_3), \ldots$ are independent.

The execution of Constructor sets at most kn of entries of B to True. However, for every $a \in [k]$, the hash value $h_a(x)$ is independent of the location of those entries that were set to true. It follows that

$$\begin{aligned}
\Pr[\, B[h_a(x)] = \mathsf{True} \,] &= \Pr[\, \exists i \in [k], j \in [n] \text{ s.t. } h_a(x) = h_i(x_j) \,] \\
&\leq \sum_{\substack{i \in [k] \\ j \in [n]}} \Pr[\, h_a(x) = h_i(x_j) \,] && \text{(by a union bound)} \\
&= \sum_{\substack{i \in [k] \\ j \in [n]}} \frac{1}{m} && \text{(by independence)} \\
&\leq \frac{kn}{m} = \frac{1}{2}. \quad \square
\end{aligned}$$

Claim 12.4.7. The BloomFilter has a false positive probability of less than 0.01.

Proof. Consider any key x that was not inserted. Then

$$\begin{aligned}
&\Pr[\, \text{IsMember}(x) = \mathsf{True} \,] \\
&\quad = \Pr[\, B[h_1(x)] = \mathsf{True} \wedge \cdots \wedge B[h_k(x)] = \mathsf{True} \,] \\
&\quad = \prod_{a=1}^{k} \Pr[\, h_a(x) = \mathsf{True} \,] && (h_1(x), \ldots, h_k(x) \text{ are independent}) \\
&\quad \leq (1/2)^k && \text{(by Claim 12.4.6)} \qquad (12.4.1) \\
&\quad < 0.01,
\end{aligned}$$

since we have chosen $k = 7$. $\square$

To conclude, the BloomFilter uses $m = 14n$ bits to represent n items, while achieving a false positive probability of less than 0.01.

Question 12.4.8. Suppose we wanted a false positive probability of ϵ. How many hash functions should we use?

Answer.

Set $k \leftarrow \lceil \lg(1/\epsilon) \rceil$. Plugging into (12.4.1), the false positive probability is at most ϵ.

Problem to Ponder 12.4.9. The pseudocode describes a Bloom filter as a static data structure, in which all entries are known at construction time. Can it support dynamic insertions or deletions?

Problem to Ponder 12.4.10. Claim 12.4.7 can be strengthened to the following statement. Let $\mathcal{E}$ be the the event that $x_1, \ldots, x_n$ were inserted, and that $h_i(x_j) = \alpha_{i,j}$ for every i and j. If x does not equal any x_j, then

$$\Pr[\, B[h_a(x)] = \mathsf{True} \mid \mathcal{E} \,] \leq 1/2 \qquad \forall a \in [k].$$

Do you see how to prove that?

12.4.3 Improved space

Let us now step back and consider an abstract problem. Suppose we want a data structure to represent a set of size n. It should support membership queries with false positive probability of ϵ, and no false negatives. How many bits of space must such a data structure use?

Algorithm 12.7 An improved Bloom Filter. It represents a set of size n using only $1.45 \lceil \lg(1/\epsilon) \rceil n$ bits, and with a false positive probability of ϵ.

global float ϵ ▷ Desired false positive probability
global int $k = \lceil \lg(1/\epsilon) \rceil$ ▷ Number of hash functions

class BloomFilter:
  hash functions $h_1, \ldots, h_k$
  Boolean array B

  ▷ The constructor. Keys is the list of keys to insert.
  Constructor (strings Keys[1..n])
    Set $m \leftarrow \lceil 1.45kn \rceil$ ▷ Length of the array
    repeat
      Create array $B[1..m]$ of Booleans, initially all False
      Let $h_1, \ldots, h_k$ be independent purely random functions
      for $j = 1, \ldots, n$
        for $i = 1, \ldots, k$
          Set $B[h_i(\text{Keys}[j])] \leftarrow$ True
    until at most $m/2$ entries of B are True

  ▷ Tests whether the string x was inserted. There can be false positives.
  IsMember (string x)
    return $B[h_1(x)] \wedge \cdots \wedge B[h_k(x)]$

- **Upper bound.** Section 12.4.2 showed that a Bloom filter solves this problem using only $m = 2kn = 2n \lceil \log(1/\epsilon) \rceil$ bits (ignoring the space for the purely random functions).
- **Lower bound.** It is known that at least $n \log(1/\epsilon)$ bits are necessary for *any* data structure that solves this problem.

So the Bloom filter uses no more than twice as much space as the best possible data structure! Can we improve it to remove this factor of 2? It turns out that Bloom filters *cannot* quite remove the 2, but we can improve the 2 to the mysterious value 1.45.

Question 12.4.11. Where was the analysis of Section 12.4.2 very pessimistic?

Answer.

When throwing kn balls into $2kn$ bins, we said that at most half of the bins will have a ball. It is likely that far fewer bins have a ball.

The pseudocode in Algorithm 12.7 describes the improvement. It differs from Algorithm 12.6 in two ways. First, it decreases the space of the array B. Second, it repeats the construction process until it can guarantee that at least half of B is empty. By removing the very pessimistic assumption, we can show that this approach will still work.

Question 12.4.12. What is the false positive probability of Algorithm 12.7?

Answer.

Since the construction process guarantees that at most $m/2$ entries of B are set to True, the proofs of Claim 12.4.6, Claim 12.4.7 and Question 12.4.8 continue to hold. Thus, the false positive probability is at most ϵ.

Claim 12.4.13. Assume that $m \geq 320$. Consider any single iteration of the **repeat** loop that constructs B. Then, for any array index i, $\Pr[\, B[i] = \mathsf{True} \,] < 0.499$.

Proof. As observed above, the creation of a Bloom filter is analogous to a balls and bins problem. During each iteration of the **repeat** loop, exactly kn indices are chosen independently at random with replacement, and the corresponding entries of B are set to True. The goal is to analyze the probability that each entry of B is set to True. This scenario is analogous to a balls and bins problem in which kn balls are thrown into m bins.

Since we want the bins to be about half empty, the situation is similar to Exercise 7.5. We have chosen m to be large enough that

$$kn \;\leq\; m/1.45. \tag{12.4.2}$$

By a typical balls and bins analysis (e.g. Section 7.3),

$$\begin{aligned}
\Pr[\, B[i] = \mathsf{True} \,] &= \Pr[\, \text{bin } i \text{ is non-empty} \,] \\
&= 1 - \Pr[\, \text{every ball misses bin } i \,] \\
&= 1 - \prod_{j=1}^{kn} \Pr[\, \text{ball } j \text{ misses bin } i \,] \\
&= 1 - (1 - 1/m)^{kn} \\
&\leq 1 - \big((1-1/m)^m\big)^{1/1.45} && \text{(by (12.4.2))} \\
&\leq 1 - \Big(\frac{1}{e} - \frac{1}{4m}\Big)^{1/1.45} && \text{(by Fact A.2.6)} \qquad (12.4.3) \\
&< 0.499,
\end{aligned}$$

by a quick numerical calculation, using our assumption that $m \geq 320$. ☐

Claim 12.4.14. The **repeat** loop performs $O(1)$ iterations, in expectation.

Proof. We can view the **repeat** loop as performing independent trials until the first success, which occurs when at most $m/2$ entries of B are True. So the number of iterations performed is a geometric random variable whose parameter p is the probability of success for each trial.

It remains to determine the value of p. Let X be the random variable denote the number of entries of B that are set to True. It follows from Claim 12.4.13 that $\mathrm{E}[\, X \,] < 0.499m$. So now we can use Markov's inequality to show that

$$1 - p \;=\; \Pr[\, \text{trial fails} \,] \;=\; \Pr[\, X > m/2 \,] \;\leq\; \frac{\mathrm{E}[\, X \,]}{m/2} \;<\; \frac{0.499m}{m/2} \;=\; 0.998.$$

This shows that $p > 0.002$.

Now that we have analyzed p, we consider the number of trials. Using the expectation of a geometric random variable (Fact A.3.21), the expected number of trials is $1/p < 1/0.002 = 500$. ☐

We summarize this analysis with the following theorem.

Theorem 12.4.15. Let $n \geq 222$. A Bloom filter can represent a set of n elements with false positive probability ϵ using only $1.45n \lceil \lg(1/\epsilon) \rceil$ bits of space. The expected time for Constructor is $O(n \log(1/\epsilon))$. The time for IsMember is $O(\log(1/\epsilon))$.

Remark 12.4.16. Where does this strange number 1.45 come from? Consider line (12.4.3), where we are trying to ensure that at most half of B is True. If we imagine that $1/4m \approx 0$, then we are roughly trying to solve the equation $1 - \left(\frac{1}{e}\right)^{1/c} = 0.5$. This has the solution $c = 1/\ln(2) \approx 1.45$.

Broader context. Bloom filters are named after Burton Bloom, who invented them in 1970. They have numerous applications in online content delivery, distributed systems, bioinformatics, etc. For example, Google Bigtable apparently uses Bloom filters to reduce disk seeks.

Interview Question 12.4.17. Bloom filters are apparently useful for interviews. See, e.g., here, here, or here.

Algorithm 12.8 A Gloom filter.

```
class GloomFilter:
    hash function h

    ▷ The constructor. Keys is the list of keys to insert.
    Constructor ( strings Keys[1..n] )
        repeat
            Let h be a purely random function mapping strings to [100]
            success ← True
            for j = 1, ..., n
                if h(Keys[j]) ≠ 1 then success ← False
        until success

    ▷ Tests whether the string x was inserted. It could make an error.
    IsMember ( string x )
        if h(x) = 1 then
            return True
        else
            return False
```

12.5 Exercises

Exercise 12.1. Sequences of bits often have special names. For example, a sequence of 4 bits is called a *nibble*, a sequence of 8 bits is called a *byte*, etc.

Let $n \geq 2$ be an arbitrary integer. Let us say that a word can store a sequence of $\lceil \lg n \rceil$ bits. Show that integers in the set $[\![m]\!]$ can be represented using $O(\log_n(m))$ words.

Exercise 12.2. Let h be a purely random function producing outputs in $[k]$. For any distinct inputs $x_1, \ldots, x_\ell$, what is $\Pr[\, h(x_1) = \cdots = h(x_\ell) \,]$?

Exercise 12.3. Algorithm 12.7 presents a better Bloom filter using only $1.45 \lceil \log(1/\epsilon) \rceil n$ bits of space. The algorithm to create this Bloom filter has a repeat loop that, according to Claim 12.4.14, executes less than 500 times in expectation. This is a constant, but still rather large. Can we create a Bloom filter much more quickly?

Consider the random variable X, the number of entries of B that are set to True. If X were a sum of independent indicator RVs, we could use the Hoeffding bound to analyze $\Pr[\, X > 0.5m \,]$. Unfortunately X is *not* a sum of independent indicators, but fortunately it turns out[2] that you can use the Hoeffding bound anyways!

Assume that $m \geq 5{,}000{,}000$. Using the Hoeffding bound (Theorem 9.3.1), prove that

$$\Pr[\,\text{repeat loop in Algorithm 12.7 runs more than once}\,] \leq 0.01.$$

[2]See Book 2, Chapter 26.

Algorithm 12.9 A modified Gloom filter which uses a single global purely random function.

```
global hash function globalH        ▷ A purely random function mapping strings to [100]

class GloomFilter:
    int q

    ▷ Hash the string x using the q^th hash function.
    Hash( string x )
        s ← q + ':' + x        ▷ Concatenate the integer q and the string x, separated by a colon
        return globalH(s)

    ▷ The constructor. Keys is the list of keys to insert.
    Constructor ( strings Keys[1..n] )
        q ← 0
        repeat
            success ← True
            for j = 1, . . . , n
                if Hash(Keys[j]) ≠ 1 then success ← False
            q ← q + 1
        until success

    ▷ Tests whether the string x was inserted. It could make an error.
    IsMember ( string x )
        if Hash(x) = 1 then
            return True
        else
            return False
```

Exercise 12.4 Gloom Filters. Professor Gloom has an idea for a new data structure, the Gloom filter. Miraculously, it uses no space at all, other than the hash function. Pseudocode is in Algorithm 12.8.

Part I. What are the false positive and false negative probabilities for this data structure? You do not need to justify your answers.

Part II. Assume that the keys given to Constructor are all distinct. What is the expected number of iterations of the **repeat** loop? Explain your answer in a few sentences.

Part III. A purely random function cannot really be implemented and would require an infinite amount of space, so we cannot really store h in the GloomFilter object. Instead, let's imagine that we have a *single* global purely random function called $globalH$. (In practice, people would usually be content to use SHA1 for this purpose.) We can then obtain different hash functions by prepending a counter q to the beginning of the key. Pseudocode is in Algorithm 12.9.

Show that, with probability at least 0.99, the GloomFilter object requires at most $7(n+1)$ bits of space.

Exercise 12.5 Faster MinHash.

Part I. In the original MinHash (Algorithm 12.3) we might want to have estimation error of 0.01 with

Algorithm 12.10 The improved MinHashBasic class that uses k values but just one hash function.

```
global int k
global hash function h

struct Signature:
    float values[1..k]

class MinHashBasic:
    Signature sig

    ▷ The constructor. E is the set of entries to insert.
    Constructor ( strings E )
        V ← { h(e) : e ∈ E }
        Sort V
        sig.values ← V[1..k]        ▷ The smallest k hash values in V

EstimateSim (MinHashBasic A, MinHashBasic B) returns float
    V ← Sort(A.sig.values ∪ B.sig.values)
    U ← V[1..k]        ▷ The smallest k hash values in V
    c ← 0
    for i = 1, ..., k
        if U[i] ∈ A.sig.values ∩ B.sig.values then
            c ← c + 1
    return c/k
```

probability 0.99. Assuming floats take 4 bytes, how many bytes of space would this use? (Ignore the space of the hash functions.)

Part II. The space is not necessarily a problem, but the constructor would take $O(nk)$ time, which could be a problem. A faster variant is shown in Algorithm 12.10.

Let $U[i]$ refer to the value in EstimateSim, and let $W[i]$ be the i^{th} smallest hash value among the elements in $A \cup B$. Prove the following claim.

Claim 12.5.1. $U[i] = W[i]$ for all $i \in [k]$.

Part III. Use Claim 12.5.1 to prove the following claim.

Claim 12.5.2. For every $i \in [k]$, the event "$U[i] \in A.\text{sig.values} \cap B.\text{sig.values}$" happens if and only if the element with hash value $W[i]$ lies in $A \cap B$.

Part IV. Use Claim 12.5.2 to prove the following claim.

Claim 12.5.3. $\Pr[\, U[i] \in A.\text{sig.values} \cap B.\text{sig.values}\,] = \text{Sim}(A, B)$.

Part V. Amazingly, the improved algorithm has the same guarantee as the original algorithm. Prove the following theorem.

Theorem 12.5.4. Let $\epsilon > 0$ be arbitrary. Suppose we set $k = \lceil \ln(200)/2\epsilon^2 \rceil$. Then

$$\Pr[\, |\text{EstimateSim}(A, B) - \text{Sim}(A, B)| \geq \epsilon \,] \leq 0.01.$$

Hint: See Section 11.2.

Exercise 12.6 **MinHash parameter tuning.** Consider the NearNeighbour data structure. Recall that n denotes the number of sets. Since Query can return a dissimilar set with probability at most $1/100$, it still might return $n/100$ sets, which seems undesirable.

In this question we will tune the parameters to return fewer sets.

Part I. Suppose we pick parameters $k = \Theta(\log n)$ and $\ell = \Theta(\sqrt{n})$. Suppose each set only contains $O(1)$ elements. How much space would the data structure take?

Part II. Find values $k = O(\log n)$ and $\ell = O(\sqrt{n})$ which guarantee that

$$\begin{array}{llll} \text{Similar sets:} & \text{if } \operatorname{Sim}(Q,A) \geq 0.9 & \text{then} & \Pr[\,\textsc{Query}\ (Q) \text{ outputs } A\,] > 0.99 \\ \text{Dissimilar sets:} & \text{if } \operatorname{Sim}(Q,B) \leq 0.81 & \text{then} & \Pr[\,\textsc{Query}\ (Q) \text{ outputs } B\,] < \dfrac{1}{\sqrt{n}}. \end{array}$$

Part III. For a query set Q, suppose that

- *Similar sets:* there are $O(1)$ sets A with $\operatorname{Sim}(A,Q) \geq 0.9$;
- *Dissimilar sets:* there are $\Theta(n)$ sets B with $\operatorname{Sim}(B,Q) \leq 0.81$;
- there are no other sets.

Assume that the dictionary is implemented as a hash table. What is the expected runtime of Query?

Chapter 13

Streaming algorithms

If we're keeping per-flow state, we have a scaling problem, and we'll be tracking millions of ants to track a few elephants.

Van Jacobson, June 2000

13.1 The streaming model

The traditional algorithms studied in introductory classes assume that all of the algorithm's data is in memory, so it can be easily accessed at will. The streaming model is a different model in which we assume that large amounts of data arrive one item at a time, and the algorithms do not have enough space to remember much about the data. This model usefully captures many computer science scenarios such as monitoring traffic at network routers, measuring usage of large websites, and gathering statistics about database systems.

13.1.1 Detailed definition

The data is a sequence of items, which we will assume to be integers in the set $[n] = \{1, \ldots, n\}$. The sequence will be denoted $a_1, a_2, \ldots, a_m$. To help remember the notation, we emphasize

$$\begin{aligned} m &= \text{(length of the stream)} \\ n &= \text{(size of the universe containing the items).} \end{aligned}$$

Each item in the sequence will be presented to the algorithm one at a time, with item a_t arriving at time t. After seeing the entire sequence (or perhaps at any point in time), the algorithm will have to answer some queries about the data. Usually we assume that m and n are known to the algorithm in advance.

In this model it is common to discuss the *frequency vector*. This is simply an array indicating the number of occurrences of each item. We will write this as an array f of n integers, where

$$f_i \;=\; (\#\text{ occurrences of item } i) \;=\; |\{\text{ times } t \,:\, a_t = i \,\}|.$$

Here are some example queries that the algorithm might face.

- Which item has the highest frequency (i.e., $\arg\max_{i\in[n]} f_i$)? What is its frequency (i.e., $\max_{i\in[n]} f_i$)?

- What is the total frequency of all items (i.e., $\sum_{i=1}^{n} f_i$)?
- What is the number of distinct items (i.e., $|\{\ i\ :\ f_i > 0\ \}|$)?
- What is the sum-of-squares of the item frequencies (i.e., $\sum_{i=1}^{n} f_i^2$)?

Question 13.1.1. Find an algorithm that can answer all of these queries in $O(m \log n)$ bits of space.

Answer.

Store the entire sequence $a_1, \ldots, a_m$. Then you can answer any queries you like.

Question 13.1.2. Find an algorithm that can answer all of these queries in $O(n \log m)$ bits of space.

Answer.

Store the entire vector $f_1, \ldots, f_n$. Then you can answer any queries you like.

Question 13.1.3. Find one of these queries that can be answered by an algorithm using only $O(\log m)$ bits of space?

Answer.

The total frequency of all items is exactly the length of the stream, i.e., $\sum_{i=1}^{n} f_i = m$.

In this chapter we will investigate whether these sorts of queries can be answered using a logarithmic[1] amount of space. For most problems, this cannot be done exactly or deterministically. Instead, we will consider approximate solutions that make use of randomization.

13.2 The majority problem

As a warm up, we will consider a problem that is a bit artificial, but is a nice stepping-stone towards more interesting problems. The task is to decide whether the data has an item that occurs a strict majority of the time. There are two cases.

- **Majority exists.** If some integer $z \in [n]$ appears in the stream *more than* half of the time, then the algorithm must return z. In symbols, z must satisfy the condition $f_z > m/2$.
- **No majority exists.** If no such z exists, then the algorithm must return "no".

Can this be done in $O(\log(nm))$ bit of space? Sadly, no. It is known that $\Omega(\min\{n, m\})$ bits of space are necessary to solve the majority problem.

A relaxed problem: ignoring the case of no majority. Interestingly, the majority problem *does* have an efficient algorithm if we allow any output in the case where no majority exists. The revised cases are as follows.

- **Majority exists.** If $z \in [n]$ appears more than half the time, then the algorithm must return z.
- **No majority exists.** If no such z exists, then the algorithm *can return anything.*

[1] We'll be happy with bounds involving $\log n$, $\log m$, $\log(nm)$, or even powers of those quantities.

Algorithm 13.1 The Boyer-Moore algorithm for finding a majority element.

```
function BoyerMoore
    Set Jar ← "empty" and Count ← 0
    for t = 1, . . . , m do
        Receive item a_t from the stream
        if Jar = "empty" then
            Set Jar ← a_t and Count ← 1                ▷ Case 1: Add item to empty jar
        else if Jar = a_t then
            Increment Count                            ▷ Case 2: New item same as jar: add it to jar
        else
            Decrement Count                            ▷ Case 3: New item differs: remove an item from jar
            if Count = 0 then Jar ← "empty"
        end if
    end for
    return Jar
end function
```

This relaxation of the problem is somewhat analogous to allowing false positives, except that the majority problem is not a Yes/No problem. In the jargon of theoretical computer science, this revised problem would be called a "promise problem". The algorithm only has to solve the problem under the promise that a majority item exists. There are no requirements if the promise is violated, meaning that there is no majority item.

Problem to Ponder 13.2.1. Explain how any algorithm for the relaxed problem can be converted into an algorithm for the original problem, if it is allowed to make *two passes* through the data stream.

13.2.1 The Boyer-Moore algorithm

Remarkably the relaxed majority problem can be solved in very low space by a deterministic algorithm! This is one of the rare deterministic algorithms we will discuss in this book. Pseudocode is shown in Algorithm 13.1.

The idea is fairly simple. Let us think of the majority item as "matter" and each non-majority item as "anti-matter". The algorithm just maintains a magical jar to which it inserts every item in the stream. Whenever the jar has particels of both matter and anti-matter, they annihilate each other. So the jar must always contain solely matter or solely anti-matter. There is not enough anti-matter to annihilate all the matter, so at the end of the algorithm the jar must contain at least one particle of matter.

That is roughly the right idea, except that the algorithm does not initially know which the majority item is. Instead, every item must annihilate any item that differs from it.

Question 13.2.2. What is the space usage of this algorithm?

Answer.

$$O(\log(n) + \log(m)) = O(\log(nm)).$$

Theorem 13.2.3. *Suppose that there is a majority item z. Then the* BoyerMoore *algorithm outputs z.*

Proof. We must show that the final value of Jar is z. Note that the value of Jar can only change when

Count $= 0$. With that in mind, let $t_1, t_2, t_3, \ldots$ be the times at which Count $= 0$. These times partition the data stream into segments. Let Jar_j be the value of Jar during segment j. During every segment j, except for the final one, exactly half of the stream items must equal Jar_j. Thus, the majority element z can be at most half of the stream items before the final segment. Since z is *more* than half of the items in the entire stream, it follows that *more* than half of the items in the final segment must equal z. Therefore the final value of Jar equals z. □

Broader context. The algorithm discussed in this section is the Boyer-Moore algorithm. There is an unrelated Boyer-Moore algorithm developed by the same pair of researchers for string searching. See also LeetCode and (Sen and Kumar, 2019, Section 16.2).

13.3 Heavy hitters

The previous section discussion the majority problem: does any item appear in the stream strictly more than a 1/2 fraction of the time? In this section we consider a generalization of this problem called the ***heavy hitters problem***: for some integer k, does any item appear in the stream strictly more than a $1/k$ fraction of the time? We will call such an item a “heavy hitter”.

Whereas there can be at most one majority element, there can be several heavy hitters. We would like the algorithm to return all of them. The formal requirements are as follows. The algorithm will return a set Output of items. For each item $h \in [n]$:

- **Heavy hitter.** If h is a heavy hitter (i.e., $f_h > m/k$) then we must have $h \in \text{Output}$.
- **Not heavy hitter.** If h is *not* a heavy hitter then it is permitted that $h \in \text{Output}$.

So far this problem is trivial: nothing prevents the algorithm from including every possible item in the output (i.e., $\text{Output} = [n]$) because that ensures that all heavy hitters would be output. We will make the problem less trivial by additionally requiring that

$$|\text{Output}| \ \leq \ k - 1,$$

because clearly there must be fewer than k heavy hitters.

13.3.1 The Misra-Gries algorithm

We present an algorithm that generalizes the BoyerMoore algorithm from Section 13.2. To explain, let us consider a boxing analogy corresponding to the case $k = 5$. There is a boxing ring with 4 corners, and there are several teams of boxers. At every time step, a boxer enters the ring. There are 3 cases.

1. If a corner is occupied by its teammates, it joins them in that corner.
2. Otherwise, if a corner is unoccupied, it goes to that corner.
3. Otherwise, if every corner is occupied by other teams, it flies into a rage and eliminates one boxer from each group, before itself collapsing and being eliminated.

At the end, the teams that remain standing are crowned the *Heavy Hitters*.

Each fight always results in 5 boxers (from different teams) being eliminated. So if some team starts with more than $1/5^{\text{th}}$ of the boxers, then it is not possible for all of its members to be eliminated. It follows that this team will be crowned a Heavy Hitter.

Algorithm 13.2 presents uses those ideas to solve the heavy hitters problem in the streaming setting.

Algorithm 13.2 The Misra-Gries algorithm for finding the heavy hitters.

```
function MisraGries
    Let C be an empty hash table
    for t = 1, ..., m do
        Receive item a_t from the stream
        if a_t ∈ C then
            Increment C[a_t]                          ▷ Case 1: Boxer joins team in corner
        else if |C| < k − 1 then
            C[a_t] ← 1                                ▷ Case 2: Boxer starts new corner
        else
            for j ∈ C                                 ▷ Case 3: Boxer starts battle
                Decrement C[j]
                if C[j] = 0 then remove j from C
        end if
    end for
    return Keys(C)
end function
```

Question 13.3.1. What is the space usage of this algorithm?

Answer.

The dictionary always has size $\leqslant k-1$. Each entry in the dictionary has a key, which takes $O(\log n)$ bits, and a count, which takes $O(\log m)$ bits. The total is $O\big((k-1)\cdot\log(nm)\big)$ bits.

Theorem 13.3.2. The output of the MisraGries algorithm contains every heavy hitter and has size at most $k-1$.

The main ideas of the proof are already present in our discussion of the boxing analogy.

Proof. Observe that each battle eliminates exactly k boxers. Since there are m boxers, there can be at most m/k battles. In fact the number of battles must be an integer, so it is at most $\lfloor m/k \rfloor$.

If team h is a heavy hitter then it has strictly more than m/k boxers. In fact the number of boxers must be an integer, so it is at least $\lfloor m/k \rfloor + 1$. This shows that not all of boxers from team h can be eliminated in the battles. Some members from team h must remain at the end of the algorithm, so h is returned in the output. □

★13.3.2 A detailed proof

In this subsection we present a more mathematical proof of Theorem 13.3.2 with painstaking detail. For notational convenience, we let $C[j]$ equal zero for all items not stored in the hash table.

Proof. The hash table always contains at most $k-1$ items, so clearly the output set has size at most $k-1$. To show that the heavy hitters are output, we will establish two invariants.

$$\textit{Invariant 1:} \qquad \underbrace{(\#\text{ battles})}_{\text{LHS}} = \underbrace{\frac{1}{k}\Big(t - \sum_{j=1}^{n} C[j]\Big)}_{\text{RHS}}.$$

Let us see why this is preserved in every iteration t.

- If there is no battle, then the LHS is unchanged. On the RHS, t increases and some $C[j]$ increases, so there is no change.
- If there is a battle, then the LHS increases. On the RHS, t increases and $k-1$ of the counters *decrease*, so the total change is $\frac{1}{k}\big(1+(k-1)\big) = 1$.

Next, consider any item h.

$$\textit{Invariant 2:} \qquad \underbrace{(\#\text{ occurrences of } h)}_{\text{LHS}} = \underbrace{C[h] + (\#\text{ battles involving } h)}_{\text{RHS}}.$$

Let us see why this is preserved in every iteration t.

- If the item that arrives is h, then the LHS increases. If h starts or joins a corner then $C[h]$ increases, otherwise h initiates a battle. In both cases the RHS increases.
- If the item that arrives is not h, then the LHS is unchanged. If this item does not initiate a battle, or if the battle does not involve h, then RHS is unchanged. However, if the battle *does* involve h then $C[h]$ decreases, so the RHS is unchanged.

We now use those invariants to complete the proof. Suppose h is a heavy hitter. Then at the end of the algorithm we have

$$\begin{aligned}
\frac{m}{k} &< f_h && (h \text{ is a heavy hitter})\\
&= (\#\text{ occurrences of } h)\\
&= C[h] + (\#\text{ battles involving } h) && (\text{by Invariant 2})\\
&\leq C[h] + (\#\text{ battles})\\
&= C[h] + \frac{1}{k}\Big(m - \sum_{j=1}^{n} C[j]\Big) && (\text{by Invariant 1 at time } t = m)\\
&\leq C[h] + \frac{1}{k}m && (C \text{ is non-negative}).
\end{aligned}$$

Rearranging, we obtain $C[h] > 0$, which implies that h will be output. □

Notes

This is the Misra-Gries algorithm. See also LeetCode and (Sen and Kumar, 2019, Section 16.2).

Algorithm 13.3 A counting filter is like a hash table that only stores the *count* of items hashing to each location.

global int $k = \lceil 1/\epsilon \rceil$

class CountingFilter:
  hash function h
  array C

  ▷ Build the data structure by processing the stream of items
  Constructor ()
    Let $C[1..k]$ be an array of integers, initially zero
    Let $h : [n] \to [k]$ be a purely random function
    for $t = 1, \ldots, m$
      Receive item a_t from the stream
      Increment $C[h(a_t)]$

  ▷ Estimates the frequency of an item
  Query (int i)
    return $C[h(i)]$

13.4 Frequency estimation

The ***frequency estimation*** problem is related to the heavy hitters problem, but is a bit different. An algorithm for the heavy hitters problem must produce a set of items containing all heavy hitters. In contrast, an algorithm for the frequency estimation problem produces a data structure that can estimate the frequency of *any* item. This data structure supports a single operation called Query(), which takes an integer $i \in [n]$ and satisfies

$$|\text{Query}(i) - f_i| \;\leq\; \epsilon m.$$

We will present several randomized algorithms for this problem. Our initial idea, shown in Algorithm 13.3, is to do something like the BasicFilter from Section 12.4, but augmented with counters.

Theorem 13.4.1. For every item $i \in [n]$,

$$\text{Lower bound:} \qquad f_i \leq \quad \text{Query}(i) \tag{13.4.1}$$

$$\text{Upper bound:} \qquad \mathrm{E}\left[\,\text{Query}(i)\,\right] \leq f_i + \epsilon m. \tag{13.4.2}$$

Proof. From the pseudocode one may see that $C[h(i)]$ counts the number of items, including i itself, having the same hash value as item i. We can express this count using indicator RVs as follows. Let X_b be the indicator of the event "$h(i) = h(b)$"; clearly $X_i = 1$. Then we have the expression

$$C[h(i)] \;=\; \sum_{b=1}^{n} X_b f_b \;=\; f_i + \sum_{b \neq i} X_b \underbrace{f_b}_{\geq 0} \;\geq\; f_i. \tag{13.4.3}$$

Since each $f_b \geq 0$ we have $\text{Query}(i) = C[h(i)] \geq f_i$, which is (13.4.1).

Now we consider the other inequality. Since h is a purely random function, we know from Question 12.1.1

that the collision probability is $\mathrm{E}[X_b] = \Pr[h(i) = h(b)] = 1/k$. So

$$\begin{aligned}
\mathrm{E}[C[h(i)]] &= \mathrm{E}\left[f_i + \sum_{b \neq i} X_b f_b\right] && \text{(expectation of (13.4.3))} \qquad (13.4.4)\\
&= f_i + \sum_{b \neq i} \mathrm{E}[X_b] f_b && \text{(linearity of expectation)}\\
&= f_i + \frac{1}{k}\sum_{b \neq i} f_b && \text{(collision probability)}\\
&\leq f_i + \frac{m}{k} && (m = \#\text{ occurrences of all items})\\
&\leq f_i + \epsilon m && \text{(by definition of } k\text{)}
\end{aligned}$$

Thus $\mathrm{E}[\textsc{Query}(i)] \leq f_i + \epsilon m$, which proves (13.4.2). □

The CountingFilter is very simple, but we have only given an analysis in expectation. Reiterating our moral from Section 8.1:

> The expectation of a random variable does not tell you everything.

Question 13.4.2. Use Markov's inequality to give an upper bound on $\Pr[\textsc{Query}(i) \geq f_i + 2\epsilon m]$?

Answer.

By definition, $\textsc{Query}(i) = C[h(i)]$. By (13.4.3), we know that $C[h(i)] - f_i \geq 0$. By (13.4.4), we know that $\mathrm{E}[C[h(i)] - f_i] \leq \epsilon m$. So by Markov's inequality, we have

$$\Pr[C[h(i)] - f_i \geq 2\epsilon m] \leq \frac{\mathrm{E}[\textsc{Query}(i) - f_i]}{2\epsilon m} \leq \frac{\epsilon m}{2\epsilon m} = \frac{1}{2}.$$

How can we refine this algorithm so that it is very likely to give good estimates?

Problem to Ponder 13.4.3. Thinking towards future topics, could we modify the algorithm to work if the stream could "delete" items as well as "inserting" them?

Question 13.4.4. How many bits of space does this algorithm use, ignoring the purely random function?

Answer.

It has an array C of size k. Each entry in the array must store a value that is at most m, which requires $O(\log m)$ bits. Therefore it uses $O(k \log m)$ bits.

13.4.1 The Count-Min Sketch

In this section, we will think about how we could improve CountingFilter to give a high-probability guarantee on its estimates. The idea is very simple, and we have seen it many times before: probability amplification by independent trials. This is analogous to how the BloomFilter improves on the BasicFilter. Pseudocode is shown in Algorithm 13.4.

Theorem 13.4.5. For all $i \in [n]$, we have

$$\begin{aligned}
&\text{Lower bound:} && f_i \leq \textsc{Query}(i) && (13.4.5)\\
&\text{Upper bound:} && \Pr[\textsc{Query}(i) \leq f_i + 2\epsilon m] \geq 0.99. && (13.4.6)
\end{aligned}$$

Algorithm 13.4 The Count-Min Sketch for frequency estimation. It runs ℓ independent copies of the CountingFilter in parallel, and returns the minimum estimate.

```
global int k = ⌈1/ε⌉
global int ℓ = ⌈lg(100)⌉ = 7

class CountMinSketch:
    hash function h_1, ..., h_ℓ
    array C

    ▷ Build the data structure by processing the stream of items
    Constructor ()
        Let C[1..ℓ, 1..k] be a two-dimensional array of integers, initially zero
        Let h_1, ..., h_ℓ : [n] → [k] be independent purely random functions
        for t = 1, ..., m
            Receive item a_t from the stream
            for j = 1, ..., ℓ
                Increment C[j, h_j(a_t)]

    ▷ Estimates the frequency of an item
    Query ( int i )
        return min {C[1, h_1(i)], ..., C[ℓ, h_ℓ(i)]}
```

Proof. Fix any $j \in [\ell]$. It will help our notation to let X_b be the indicator of the event "$h_j(i) = h_j(b)$". Following (13.4.3), we have

$$C[j, h_j(i)] \;=\; \sum_{b=1}^{n} X_b f_b \;=\; f_i + \sum_{b \neq i} X_b \underbrace{f_b}_{\geq 0} \;\geq\; f_i. \tag{13.4.7}$$

Since $C[j, h_j(i)] \geq f_i$ for each j, and $\text{Query}(i)$ is the minimum of those values, it follows that $\text{Query}(i) \geq f_i$, which proves (13.4.5).

Now we consider the other inequality. The analysis of (13.4.4) shows that, for all $j \in \{1, \ldots, \ell\}$,

$$\mathrm{E}\left[\, C[j, h_j(i)] \,\right] \;\leq\; f_i + \epsilon m$$

Now by the same argument as Question 13.4.2, we know that $C[j, h_j(i)] - f_i$ is non-negative, so Markov's inequality implies that

$$\Pr\left[\, C[j, h_j(i)] - f_i \geq 2\epsilon m \,\right] \;\leq\; \frac{\mathrm{E}\left[\, C[j, h_j(i)] - f_i \,\right]}{2\epsilon m} \;\leq\; \frac{\epsilon m}{2\epsilon m} \;=\; \frac{1}{2}.$$

Now we can use independence of the hash functions to drive down the failure probability:

$$\begin{aligned}
\Pr\left[\, \text{Query}(i) - f_i \geq 2\epsilon m \,\right] \;&=\; \Pr\left[\, \min_{1 \leq j \leq \ell} C[j, h_j(i)] - f_i \geq 2\epsilon m \,\right] \\
&=\; \prod_{1 \leq j \leq \ell} \Pr\left[\, C[j, h_j(i)] - f_i \geq 2\epsilon m \,\right] \qquad \text{(independence of the } h_j\text{)} \\
&\leq\; 2^{-\ell} \;=\; 0.01.
\end{aligned}$$

This proves (13.4.6). ∎

Question 13.4.6. How much space is used by this algorithm, ignoring the purely random functions?

Answer.

It has a table of $k\ell$ counters, each taking $O(\log m)$ bits. Since ℓ is a constant and $k = O(1/\epsilon)$, the total space is $O(\log(m)/\epsilon)$ bits.

Notes

This is the Count-min sketch, due to Graham Cormode and Muthu Muthukrishnan. Their research won the Imre Simon test-of-time award in 2014.

13.5 The streaming model with deletions

The streaming model, as described so far, involves counting the number of occurrences of different items in the stream. We could also view the algorithm as maintaining a multiset of items: at each time step a new item is inserted, which may or may not already be in the multiset.

A variant of the streaming model also allows items to be *deleted* from the multiset. For example, the streaming algorithm might be used to monitor network packets, some of which open a connection, and some of which close a connection. The items in the multiset would correspond to the currently active connections.

There are three different models that are commonly considered.

- *Insert-only model.* The stream only inserts elements. (Nickname: cash-register model.)
- *Deletions model (non-negative case).* The stream has both insertions and deletions. However, the number of deletions for item i *never* exceeds the number of insertions for item i (so $f_i \geq 0$ for all i). (Nickname: strict turnstile model.)
- *Deletions model (general case).* The stream has both insertions and deletions. However, the number of deletions for item i *can* exceed the number of insertions for item i (so $f_i < 0$ is possible). (Nickname: turnstile model.)

For example, the BoyerMoore and MisraGries algorithms are originally intended to work in the insert-only model. There are variants of these algorithms that can also handle deletions.

Question 13.5.1. Can a CountingFilter handle deletions?

Answer.

The natural idea is to increment and decrement the counter as items are inserted/deleted. In the deletions model (non-negative case), the analysis does not change at all, and we have:

$$f_i \leq \textsc{Query}(i) \quad \text{and} \quad \mathrm{E}\left[\textsc{Query}(i)\right] \leq f_i + \epsilon m.$$

In the deletions model (general case), it is not hard to see that:

$$-\epsilon m \leq \mathrm{E}\left[\textsc{Query}(i) - f_i\right] \leq \epsilon m.$$

Question 13.5.2. Can a CountMinSketch handle deletions?

Answer.

In the deletions model (non-negative case), the analysis does not change at all. In the deletions model (general case), taking the min seems problematic.

13.6 Frequency estimation with deletions

Some new ideas are needed to do frequency estimation in the deletions model (general case). A CountingFilter can give reasonable frequency estimates *in expectation*, but that is a weak guarantee. The CountMinSketch improved on the CountingFilter by using many hash functions to generate many independent estimates.

The key question is: how can we pick the best estimate? In the insert-only model, or deletions model (non-negative case), each estimate is of the form

$$C[j, h_j(i)] \;=\; f_i + \underbrace{(\text{error due to colliding items})}_{\geq 0}.$$

Since the error term is non-negative we can simply pick the minimum estimate in order to minimize the error. In the deletions model (general case), the estimate is of the form

$$C[j, h_j(i)] \;=\; f_i + \underbrace{(\text{error due to colliding items})}_{(\text{can be } \leq 0 \text{ or } \geq 0)}.$$

If we choose the minimum estimate, it might have a very negative error. If we choose the maximum estimate, it might have a very positive error. So what should we pick?

The *median* estimate seems like the ideal choice. Hopefully we can show it is neither too positive nor too negative. The pseudocode in Algorithm 13.5 implements this idea.

Question 13.6.1. How much space is used by this algorithm, ignoring the purely random functions?

Answer.

This is asymptotically the same as the CountMinSketch algorithm: $O(\log(m)/\epsilon)$ bits.

13.6.1 Analysis

Theorem 13.6.2. For all $i \in [n]$, we have

$$\Pr\left[\, |\textsc{Query}(i) - f_i| > 3\epsilon m \,\right] \;\leq\; 0.01.$$

The main difference from Theorem 13.4.5 is that we now look at the *absolute value* of the error, since it could be negative. Also, the error bound has a 3 instead of a 2, but that's just a minor detail. The proof is quite similar to Theorem 13.4.5, except that it must deal with absolute values, and it uses Theorem 9.5.5 to analyze the estimate.

Proof of Theorem 13.6.2. Fix some item $i \in [n]$, and consider the j^{th} estimate, for some $j \in [\ell]$. As in Theorem 13.4.1, we let X_b be the indicator of the event "$h_j(i) = h_j(b)$", and we recall that $\mathrm{E}\left[\, X_b \,\right] = 1/k \leq \epsilon$. The j^{th} estimate has the form

$$C[j, h_j(i)] \;=\; \sum_{b=1}^{n} X_b f_b \;=\; f_i + \sum_{b \neq i} X_b f_b.$$

Algorithm 13.5 The CountMedSketch algorithm modifies the CountMinSketch algorithm to work in the deletions model, general case.

---

global int $k = \lceil 1/\epsilon \rceil$
global int $\ell = 97$

class CountMedSketch:
  hash function $h_1, \ldots, h_\ell$
  array C

  ▷ Build the data structure by processing the stream of items
  Constructor ()
    Let $C[1..\ell, 1..k]$ be a two-dimensional array of integers, initially zero
    Let $h_1, \ldots, h_\ell : [n] \to [k]$ be independent purely random functions
    for $t = 1, \ldots, m$
      Receive item a_t from the stream
      for $j = 1, \ldots, \ell$
        if a_t says insert i
          Increment $C[j, h_j(a_t)]$
        else if a_t says delete i
          Decrement $C[j, h_j(a_t)]$

  ▷ Estimates the frequency of an item
  Query (int i)
    return Median$(C[1, h_1(i)], \ldots, C[\ell, h_\ell(i)])$

This estimate can have either positive or negative deviation from the true value, f_i. We can avoid negative numbers by looking at the absolute value:

$$\left| C[j, h_j(i)] - f_i \right| \;=\; \left| \sum_{b \neq i} X_b f_b \right| \;\leq\; \sum_{b \neq i} X_b \cdot |f_b|,$$

by the triangle inequality (Fact A.2.4). Now we consider the expected error, and use linearity of expectation.

$$\mathrm{E}\left[\, \left| C[j, h_j(i)] - f_i \right| \,\right] \;\leq\; \sum_{b \neq i} \mathrm{E}\left[\, X_b \,\right] \cdot |f_b| \;\leq\; \epsilon \sum_{b \neq i} |f_b|$$

Note that $|f_b| \leq$ (# insertions or deletions of item b), and therefore $\sum_{b=1}^{n} |f_b| \leq m$. This yields our key bound on the expected error.

$$\mathrm{E}\left[\, \left| C[j, h_j(i)] - f_i \right| \,\right] \;\leq\; \epsilon m \tag{13.6.1}$$

Next, following Theorem 13.4.5, we analyze the probability that the error is slightly larger.

$$\begin{aligned} \Pr\left[\, \left| C[j, h_j(i)] - f_i \right| \geq 3\epsilon m \,\right] &\leq \frac{\mathrm{E}\left[\, \left| C[j, h_j(i)] - f_i \right| \,\right]}{3\epsilon m} && \text{(by Markov's inequality)} \\ &\leq \frac{\epsilon m}{3\epsilon m} = \frac{1}{3} && \text{(by (13.6.1))} \end{aligned}$$

Writing out the two tails separately, we have

$$\begin{aligned} &\text{Right tail:} && \Pr\left[\, C[j, h_j(i)] \geq f_i + 3\epsilon m \,\right] \leq \frac{1}{3} \\ &\text{Left tail:} && \Pr\left[\, C[j, h_j(i)] \leq f_i - 3\epsilon m \,\right] \leq \frac{1}{3}. \end{aligned} \tag{13.6.2}$$

Now we are perfectly set up to analyze the median estimator using Theorem 9.5.5. Transitioning to the notation of that theorem, we have

$$Z_j = C[j, h_j(i)], \qquad \ell = 97, \qquad L = f_i - 3\epsilon m, \qquad R = f_i + 3\epsilon m, \qquad \alpha = 1/6.$$

Using this new notation, (13.6.2) may be restated as

$$\begin{aligned} \Pr[\, Z_j > R \,] &\leq \frac{1}{2} - \alpha \\ \Pr[\, Z_j < L \,] &\leq \frac{1}{2} - \alpha. \end{aligned}$$

This yields the following bound on the error of Query.

$$\begin{aligned} \Pr[\, |\textsc{Query}(i) - f_i| > 3\epsilon m \,] &= \Pr[\, \textsc{Query}(i) < f_i - 3\epsilon m \text{ or } \textsc{Query}(i) > f_i + 3\epsilon m \,] \\ &= \Pr[\, \mathrm{Median}(Z_1, \ldots, Z_\ell) \text{ is } < L \text{ or } > R \,] \\ &\leq 2\exp\big(- 2(1/6)^2 97 \big) \qquad \text{(by Theorem 9.5.5)} \\ &< 0.01, \end{aligned}$$

by a quick numerical calculation. □

13.7 Distinct elements

Estimating the size of a set is a remarkably useful task. For example, Zoodle has a subsidiary Shreddit that runs a website on which people discuss techniques to shred vegetables. They would like to keep track of the *number of unique visitors* to all of the Shreddit posts. The obvious approach is to maintain an explicit list of all visitors, but this will take a lot of space. Is there a more efficient way to estimate the number of unique visitors?

Brainstorming. Earlier section have described several data structures for representing sets, such as the MinHash technique to estimate similarity, and Bloom filters to determine membership. We have also seen the MisraGries and CountMinSketch algorithms, but they seem less relevant because they estimate how *many* times items appear in *multi*sets. Recall that the MinHashBasic data structure (Algorithm 12.3) represents a set S using constant space, whereas a Bloom filter uses linear space. Since the MinHashBasic is so space efficient, it would be wonderful if it could be adapted to estimate the *size* of a set.

Following Section 12.2, let's assume that the hash function maps to random values that are uniform in the interval $[0, 1]$. Then a set S of size d would be represented just by the random real number

$$Z \;=\; \min_{s \in S} h(s) \;=\; \min_{i \in [d]} U_i$$

where $U_1, \ldots, U_d$ are independent random variables that are uniform on $[0, 1]$. Given Z, can we somehow estimate d?

13.7.1 The minimum of d uniform random variables

The minimum of d uniform RVs has been well studied. For example, it is known that

$$\mathrm{E}[\,Z\,] \;=\; \frac{1}{d+1}.$$

References: (Mitzenmacher and Upfal, 2005, Lemma 8.3), StackExchange, Wikipedia.

This is good news! Given Z, perhaps we can invert this relationship to estimate d? More concretely, define $\hat{d} = \frac{1}{Z} - 1$. Might this be an unbiased estimator?

$$\mathrm{E}\left[\,\hat{d}\,\right] \;=\; \mathrm{E}\left[\,\tfrac{1}{Z} - 1\,\right] \;\overset{??}{=}\; \tfrac{1}{\mathrm{E}[\,Z\,]} - 1 \;=\; \tfrac{1}{1/(d+1)} - 1 \;=\; d.$$

Oh dear – this equation is incorrect[2]. The flaw is that expectation does not interact nicely with inverses. This relates to our moral from Section 8.1:

> The expectation of a random variable does not tell you everything.

Intuitively, if we could somehow estimate $\mathrm{E}[\,Z\,]$ very precisely, then we'd be in good shape for estimating d. So how can we do that? A natural idea is to take several independent estimates $Z_1, \ldots, Z_\ell$. But what shall we do with them?

- Take the min (or max)? This is unlikely to be useful. As discussed in Section 13.6, this would just select the estimate with the largest negative (or positive) error.
- Take the mean? Suppose we define $Y = \frac{1}{\ell}\sum_{i=1}^{\ell} Z_i$ then use the "Hoeffding for averages" bound (Theorem 9.6.1). Since the expectation is $\Theta(1/d)$, we can estimate it to within a constant factor by taking $q = \Theta(1/d)$. The failure probability is at most $2\exp(-2q^2\ell)$, and for this to be small we need $\ell = \Omega(d^2)$. This is unfortunate because d could be $\Theta(m)$, so then we would need $\ell = \Omega(m^2)$ estimates, which is much too large.
- Take the median? Theorem 9.5.5 is a powerful tool, and it was useful in Section 13.6, so we will employ that approach again in this section.

13.7.2 The algorithm

Pseudocode for our approach based on medians is shown in Algorithm 13.6. The following theorem shows that this algorithm is very likely to estimate the number of distinct elements to within 10% of the true value.

Theorem 13.7.1. The true cardinality d is contained in the interval $[d_{\min}, d_{\max}]$ computed by EstimateCardinality with probability at least 0.99. Moreover, the interval is rather small: $d_{\max}$ is less than 10% bigger than $d_{\min}$.

Question 13.7.2. How much space does this algorithm use (ignoring the purely random functions)?

Answer.

The algorithm just stores $\ell = \Theta(1/\alpha)^2$ floats. With 32-bit floats and $\alpha = 0.0165$, that's $9731 \times 4 = 38,924$ bytes of space.

[2] This equation is very far from true because, in fact, $\mathrm{E}\left[\,\hat{d}\,\right] = \infty$; see equation (2) in this paper.

Algorithm 13.6 An algorithm for estimating the number of distinct elements in a data stream.

global float $\alpha \leftarrow 0.0165$
global int $\ell \leftarrow \lceil \ln(200)/2\alpha^2 \rceil = 9731$

▷ Independent purely random functions mapping $[n]$ to $[0,1]$
global hash functions $h_1, \ldots, h_\ell$

class DistinctElements:
    array $Z[1..\ell]$

    ▷ Build the data structure by processing the stream of items
    Constructor ()
        Let $Z[1..\ell]$ be an array of floats, initially 1
        for $t = 1, \ldots, m$
            Receive item a_t from the stream
            for $i = 1, \ldots, \ell$
                $Z[i] \leftarrow \min\{Z[i], h_i(a_t)\}$

    ▷ The number of distinct elements is very likely in the interval $[d_{\min}, d_{\max}]$
    EstimateCardinality()
        Compute $M = \text{Median}(Z[1], \ldots, Z[\ell])$
        Compute $d_{\min} = \frac{\ln(0.5+\alpha)}{\ln(1-M)}$ and $d_{\max} = \frac{\ln(0.5-\alpha)}{\ln(1-M)}$
        return $[d_{\min}, d_{\max}]$

Actually, regarding the space, there are a lot of details that are being swept under the rug. Our analysis has assumed that the hash values $h_i(x)$ are real numbers with infinite precision, which is obviously unrealistic. We won't bother to analyze the precision needed by this algorithm, because a purely random function is unrealistic anyways.

13.7.3 Analysis

Analyzing the tails of Z. In order to apply the median estimator, the first step is to satisfy the inequalities (9.5.1). To do this, we consider the random variable

$$Z = \min_{i \in [d]} U_i \tag{13.7.1}$$

and then analyze its left and right tails. This is not too difficult: we can actually write down an explicit formula! Assume $0 \leq \alpha < 1/2$, and define the following mysterious values.

$$\text{Left threshold:} \qquad L = 1 - \left(\frac{1}{2} + \alpha\right)^{1/d}$$
$$\text{Right threshold:} \qquad R = 1 - \left(\frac{1}{2} - \alpha\right)^{1/d}.$$

These formulas seem to come from thin air, and it is hard to have much intuition about them. It turns out that they roughly satisfy $L \lessapprox \frac{\ln 2}{d} \lessapprox R$. Their precise definition is chosen to make the next lemma work smoothly.

Lemma 13.7.3. Let $U_1, \ldots, U_d$ be independent random variables, uniform on $[0,1]$. Then

$$\begin{aligned}
\text{Left tail:} \qquad \Pr\left[\min_{i\in[d]} U_i \leq L\right] &= \frac{1}{2} - \alpha \\
\text{Right tail:} \qquad \Pr\left[\min_{i\in[d]} U_i \geq R\right] &= \frac{1}{2} - \alpha.
\end{aligned}$$

Proof. First we analyze the right tail:

$$\Pr\left[\min_{i\in[d]} U_i \geq R\right] \;=\; (1-R)^d \;=\; \left(\left(\frac{1}{2}-\alpha\right)^{1/d}\right)^d \;=\; \frac{1}{2} - \alpha$$

by Exercise A.5 and the definition of R. The analysis of the left tail is similar:

$$1 - \Pr\left[\min_{i\in[d]} U_i \leq L\right] \;=\; \Pr\left[\min_{i\in[d]} U_i \geq L\right] \;=\; \prod_{i\in[d]} \Pr\left[\, U_i \geq L \,\right] \;=\; (1-L)^d \;=\; \frac{1}{2} + \alpha.$$

Rearranging, we conclude that $\Pr\left[\, \min_{i\in[d]} U_i \leq L \,\right] = \frac{1}{2} - \alpha$. $\square$

Applying the median estimator. Now that we have analyzed the tails of Z, we are equipped to take the median of many estimates. We will run ℓ instances of the MinHash algorithm on the set S, where the i^{th} instance has its own independent purely random function h_i. The i^{th} instance of MinHash maintains the variable

$$Z_i \;=\; \min_{s\in S} h_i(s).$$

Each of these Z_i is an independent copy of the random variable Z described in (13.7.1). The median of these estimates is

$$M \;=\; \text{Median}(Z_1, \ldots, Z_\ell).$$

We now show that M is very likely to lie within the chosen thresholds.

Claim 13.7.4. Define $\ell \;=\; \lceil \ln(200)/2\alpha^2 \rceil$, perhaps adding 1 to ensure that ℓ is odd. Then

$$\Pr\left[\, L < M < R \,\right] \;\geq\; 0.99.$$

Proof. Directly applying Theorem 9.5.5, we have

$$\Pr\left[\, L < M < R \,\right] \;\geq\; 1 - 2\exp(-2\alpha^2\ell) \;\geq\; 1 - 2\exp(-\ln(200)) \;=\; 0.99. \qquad \square$$

Estimating the value of d. After computing the median M, the algorithm must estimate the cardinality. Instead of computing a single estimate, it computes an interval $[d_{\min}, d_{\max}]$ that is likely to contain the true cardinality d. This interval's endpoints are[3]

$$d_{\min} \;=\; \frac{\ln\left(\frac{1}{2}+\alpha\right)}{\ln(1-M)} \qquad \text{and} \qquad d_{\max} \;=\; \frac{\ln\left(\frac{1}{2}-\alpha\right)}{\ln(1-M)}.$$

Note that $d_{\min}$ and $d_{\max}$ are actually random variables, because they depend on M

Much like L and R had mysterious definitions, $d_{\min}$ and $d_{\max}$ do too. In fact, these definitions are related. The following claim shows that, by inverting the relationship between M and d, the cardinalities that correspond to L and R are the values $d_{\min}$ and $d_{\max}$.

[3] Although it might not appear that $d_{\min} < d_{\max}$, that inequality does hold because both the numerators and the denominators are negative.

Claim 13.7.5. The event "$L < M < R$" is equivalent to the event "$d_{\min} < d < d_{\max}$".

Proof. The proof is just a sequence of routine algebraic manipulations.

$$\begin{array}{rcccccl} & L & < & M & < & R & \\ \iff & \left(\frac{1}{2}+\alpha\right)^{1/d} & > & 1-M & > & \left(\frac{1}{2}-\alpha\right)^{1/d} & \text{(subtracting from 1)} \\ \iff & \left(\frac{1}{2}+\alpha\right) & > & (1-M)^d & > & \left(\frac{1}{2}-\alpha\right) & \text{(raising to power } d\text{)} \\ \iff & \ln\left(\frac{1}{2}+\alpha\right) & > & d\ln(1-M) & > & \ln\left(\frac{1}{2}-\alpha\right) & \text{(taking log)} \\ \iff & \underbrace{\frac{\ln(\frac{1}{2}+\alpha)}{\ln(1-M)}}_{d_{\min}} & < & d & < & \underbrace{\frac{\ln(\frac{1}{2}-\alpha)}{\ln(1-M)}}_{d_{\max}} & \text{(dividing by } \ln(1-M)\text{)} \end{array}$$

In the last line, the inequality flips direction because $0 < M < 1$ so $\ln(1-M) < 0$. □

Armed with those two claims, the theorem is immediate.

Proof of Theorem 13.7.1. We have

$$\begin{aligned} \Pr[\, d_{\min} < d < d_{\max} \,] &= \Pr[\, L < M < R \,] && \text{(by Claim 13.7.5)} \\ &\geq 0.99 && \text{(by Claim 13.7.4).} \end{aligned}$$

To judge the size of the interval, we just plug in the definitions of $d_{\min}$ and $d_{\max}$ to obtain

$$\frac{d_{\max}}{d_{\min}} = \frac{\ln\left(\frac{1}{2}-\alpha\right)}{\ln\left(\frac{1}{2}+\alpha\right)} < 1.1,$$

by a quick numerical calculation using our chosen value $\alpha = 0.0165$. □

Problem to Ponder 13.7.6. Algorithm 13.6 produces an interval satisfying $d_{\max} \leq 1.1 \cdot d_{\min}$. Suppose you wanted to ensure that $d_{\max} \leq (1+\epsilon) \cdot d_{\min}$ for an arbitrary $\epsilon > 0$. How would you modify the algorithm's parameters α and ℓ?

Notes

The distinct elements problem is also known as the count-distinct problem. A very practical and popular algorithm for this problem is HyperLogLog. It has numerous applications, including at Google, Facebook, Amazon Web Services, Reddit, in the Redis database, etc.

In theory, a better algorithm is known, that is actually optimal! It can estimate d up to a factor of $1+\epsilon$ using $O\big(\frac{1}{\epsilon^2} + \log n\big)$ bits of space, even when using implementable hash functions with provable guarantees. There is a matching lower bound of $\Omega\big(\frac{1}{\epsilon^2} + \log n\big)$ bits. This optimal algorithm is due to Daniel Kane, Jelani Nelson, and David P. Woodruff.

13.8 Exercises

Exercise 13.1 Frequency estimation. Design a *deterministic* streaming algorithm with the following behavior. Given any parameter ϵ, with $0 < \epsilon < 1$, it processes the stream and produces a data structure using $O(\log(nm)/\epsilon)$ bits of space. The data structure has just one operation $\textsc{Query}()$ which takes an integer $i \in \{1, \ldots, n\}$ and satisfies

$$f_i - \epsilon m \;\leq\; \textsc{Query}(i) \;\leq\; f_i.$$

Prove that your approach satisfies these guarantees.

Exercise 13.2 CountMedSketch with absolute values. Define Excl_i to be the the sum of the *absolute values* of the item frequencies, *excluding* item i:

$$\mathrm{Excl}_i \;=\; \sum_{j \neq i} |f_j|.$$

Part I. Show that $\mathrm{Excl}_i \leq m$.

Part II. Improve Theorem 13.6.2 to show that, for all $i \in [n]$, we have

$$\Pr\left[\, |\textsc{Query}(i) - f_i| > 3\epsilon\, \mathrm{Excl}_i \,\right] \;\leq\; 0.01.$$

Exercise 13.3 Counting Distinct Elements. In this exercise you must (approximately) count the number of distinct strings in a given data file. The file is available at `https://www.cs.ubc.ca/~nickhar/W23/data.csv`. It has one string on each line.

Part I. First write a Python program to count the exact number of distinct strings in the data file. Provide the code and the output of your program.

Part II. Provide a Python implementation of the Distinct Element estimator from Algorithm 13.6. Run your code on the data file and report its output. What is the meaning of this output?

Part III. The $\textsc{DistinctElements}$ estimator from Algorithm 13.6 is too slow to be effective in practice due to the large number of hash functions. It is designed to be simple to analyze, not to be practical. The HyperLogLog estimator is an industrial grade estimator that uses similar ideas. It has a Python implementation in the hyperloglog package. Use this package to approximately count the number of distinct strings in the data file up to 1% error. Provide the code of your program and its output.

Exercise 13.4 Combining DistinctElements estimators. Suppose that we instantiate a $\textsc{DistinctElements}$ object A (exactly as written in Algorithm 13.6 in the text) and insert every element in some list $listA$. Later we instantiate a different $\textsc{DistinctElements}$ object B (as in Algorithm 13.6) and insert every element in some list $listB$. At this point we could use those objects to estimate the number of distinct elements in $listA$ or in $listB$.

Instead, we now decide to estimate the number of distinct elements in the concatenation $listA, listB$. Give pseudocode for an algorithm that does this, using the two objects that we have already created. The estimate should be accurate to within 10%, with probability 0.99. In a few sentences, explain why your algorithm works.

Exercise 13.5 Boyer-Moore with deletions. The Boyer-Moore algorithm is a deterministic algorithm solving the Majority problem in the insert-only model. In this problem we consider an extension to the deletions model (non-negative case).

Previously we had called element i a majority element if $f_i > m/2$. Now we revise the definition slightly: i is a majority element if $f_i > \sum_{j=1}^{n} f_j/2$ at the end of the stream.

Part I. Consider a stream with only two different items (say, "B" and "T"). Design an algorithm that uses $O(\log m)$ bits such that, if there is a majority element, it will return that element.

Part II. Consider a stream whose items are either insertions or deletions of integers in the range $\{0, \ldots, n-1\}$. Design a deterministic algorithm such that, if there is a majority element, it will return that element. Your algorithm should use $O\big(\log(n)\log(m)\big)$ bits of space.

Hint: Consider the binary representation of the items.

Exercise 13.6 Boyer-Moore being merged. Suppose that a company has two internet gateways, which we call Server A and Server B. All of the company's internet traffic traverses one of those two gateways. The company would like to monitor its internet traffic for one day. Can it determine which IP address is receiving more than 50% of the traffic?

To do so, Server A runs the BoyerMoore algorithm to monitor all its internet traffic. Separately, Server B also runs the BoyerMoore algorithm to monitor all its internet traffic. The algorithm's states on these respective servers are denoted $(\mathrm{Jar}_A, \mathrm{Count}_A)$ and $(\mathrm{Jar}_B, \mathrm{Count}_B)$. At the end of the day, the four values

$$\mathrm{Jar}_A, \mathrm{Count}_A, \mathrm{Jar}_B, \mathrm{Count}_B$$

are sent to the network administrator.

Suppose that there is a majority element M amongst *all* the internet traffic. How can the network administrator determine the majority element given just those four values? Prove that your reasoning is correct.

Chapter 14

Low-space hash functions

In the last two chapters we have seen many interesting applications of hash functions. We have been using purely random functions, which are mathematically convenient but impractical because they require enormous amounts of space. In this chapter we will address this issue by designing hash functions that require modest amounts of space and have some mathematical guarantees.

Engineering viewpoint. Depending on the application, software engineers usually get good results using non-random hash functions from standard libraries. A small detail is that algorithms sometimes require multiple independent hash functions. In practice that is not an issue: one can simply combine the index of the hash function with the string to be hashed. Consider Algorithm 12.9, or the following Python code.

```python
def Hash(i,x):
    return hash(str(i)+":"+x)

s = "I like zucchini"
print(Hash(1,s))     # Returns: 9064104254631292544
print(Hash(2,s))     # Returns: 1683624909374652516
```

A less clunky approach would be to use a hashing library whose hash function supports a "seed" or "salt". Then the i^{th} hash function can simply set the seed to i, or to a new random value. These simple approaches are likely to work well in practice, but do not have any rigorous guarantees.

Theoretical viewpoint. We want to design hash functions that are practical and useful. Some goals are:

- *Randomness.* In order to make probabilistic statements about the hash function, it must incorporate some actual randomness, perhaps as its seed.
- *Small space.* A hash function for strings of length s is considered practical if it uses $O(s)$ bits, rather than the roughly 2^s bits needed by a purely random function.
- *Mathematical guarantees.* We want our hash functions to have enough mathematical properties that they can work in many of the applications that we have seen.

14.1 Linear hashing modulo a prime

Imagine that each character comes from the set $[\![q]\!]$. For example, strings of bits would have $q = 2$, strings of bytes would have $q = 256$, and strings of Unicode characters would have $q = 65536$. We will design a hash function to output a single character.

For inspiration, let's look at a basic pseudorandom number generator. Have you ever looked at how the `rand()` function in C is implemented? It is a linear congruential generator, and basically works as follows.

```
const int x = 1103515245;
const int y = 12345;
next = x * prev + y;
```

Here x and y are "mixing parameters" that look vaguely random. This is not very impressive, but we will draw some inspiration from it.

Our goal is to hash strings whose characters are in $[\![q]\!]$. There are a lot more mathematical tools at our disposal if we work modulo prime numbers. So perhaps we can pick a prime p, then define the hash of a character a to be

$$\text{Hash of a single character } a: \qquad (x \cdot a + y) \bmod p.$$

For this to work well, we will require that $p \geq q$. In our example, we can use $p = 2$ for strings of bits, $p = 257$ for strings of bytes, and $p = 65537$ for strings of unicode symbols, since these are all prime numbers.

Question 14.1.1. This hashing scheme has a flaw if $p < q$. Note that the characters 0 and p both belong to the set $[\![q]\!]$. Explain why 0 and p will always have the same hash value.

Answer.

$$(\text{hash of } p) \;=\; (x \cdot p + y) \bmod p \;=\; y \bmod p \;=\; (\text{hash of } 0).$$

Since we want to hash *strings*, we will hash each character separately, then just add up the results. To do so, we will need mixing parameters $X_1, \ldots, X_s$ and Y in $[\![p]\!]$. Then, given a string $a = (a_1, \ldots, a_s)$, its hash value will be

$$\text{Hash of a string } a: \qquad h(a) \;=\; \Big(\sum_{i=1}^{s} a_i X_i + Y \Big) \bmod p. \tag{14.1.1}$$

The design of this hash function is called **linear hashing**. We will show that linear hashing has several nice properties.

Definition 14.1.2. A random hash function is said to be ***strongly universal***[1] if, informally,

*every **pair** of hash values is uniformly random.*

More formally, if we have a random function h mapping to $[\![p]\!]$, the condition is that

$$\Pr[\, h(a) = \alpha \,\wedge\, h(b) = \beta \,] \;=\; \frac{1}{p^2} \tag{14.1.2}$$

for every two *different* strings a, b of length s, and every two characters $\alpha, \beta \in [\![p]\!]$.

[1] The terminology is not very consistent. Some people call this property "2-universal", or "strongly 2-universal", or "pairwise independent", or "pairwise independent and uniform".

Theorem 14.1.3. Let h be a linear hash function, as in (14.1.1), where the mixing parameters $X_1, \ldots, X_s$ and Y are *mutually independent and uniformly random.* Then h is strongly universal.

Notice that this probability would be *exactly the same* if h were a purely random function. However, in this theorem h is *not* purely random: it is a linear hash function, with the form shown in (14.1.1).

Question 14.1.4. Is there a property of purely random functions that is not satisfied by linear hashing?

Answer.

For every three distinct input strings a, b, c of length s, and any characters α, β, γ, a purely random function h satisfies

$$\Pr[\, h(a) = \alpha \,\wedge\, h(b) = \beta \,\wedge\, h(c) = \gamma \,] \;=\; \frac{1}{p^3}.$$

This is not satisfied by our linear hash function h.

Question 14.1.5. How much space does it take to represent h, as a function of s, the string length, and p, the range of the outputs?

Answer.

The number of parameters is $s+1$, and each parameter takes $O(\log p)$ bits, so the total is $O(s \log p)$ bits.

14.1.1 Consequences

Corollary 14.1.6 (Hashes are uniform). Let h have the form in (14.1.1), where $X_1, \ldots, X_s$ and Y are chosen uniformly and independently at random. For any string a of length s and any character α,

$$\Pr[\, h(a) = \alpha \,] \;=\; \frac{1}{p}.$$

Question 14.1.7. Do you see how to prove Corollary 14.1.6?

Answer.

By the law of total probability (Fact A.3.6),

$$\Pr[\, h(a) = \alpha \,] \;=\; \sum_{\beta \in [\![p]\!]} \Pr[\, h(a) = \alpha \wedge h(b) = \beta \,] \;=\; \sum_{\beta \in [\![p]\!]} \frac{1}{p^2} \;=\; \frac{1}{p}.$$

Corollary 14.1.8 (Collisions are unlikely). Assume that h has the form in (14.1.1), where $X_1, \ldots, X_s$ and Y are chosen uniformly and independently at random. For any different strings a, b of length s,

$$\Pr[\, h(a) = h(b) \,] \;=\; \frac{1}{p}.$$

Proof. By the law of total probability (Fact A.3.6),

$$\begin{aligned}
\Pr[\, h(a) = h(y) \,] &= \sum_{\alpha \in [\![p]\!]} \Pr[\, h(a) = h(b) \wedge h(a) = \alpha \,] \\
&= \sum_{\alpha \in [\![p]\!]} \Pr[\, h(a) = \alpha \wedge h(b) = \alpha \,] \;=\; \sum_{\alpha \in [\![p]\!]} \frac{1}{p^2} \;=\; \frac{1}{p}.
\end{aligned}$$

□

14.1.2 Proof of Theorem

To prove the theorem, we will use the following fact.

Fact A.2.16 (Systems of equations mod p). Let p be a prime number. Let $a, b \in [\![p]\!]$ satisfy $a \neq b$. Let $c, d \in [\![p]\!]$. Let X and Y be variables. Then there is exactly one solution to

$$\begin{aligned} (aX + Y) \bmod p &= c \\ (bX + Y) \bmod p &= d \\ X, Y &\in [\![p]\!]. \end{aligned}$$

Proof of Theorem 14.1.3. Recall that the strings a and b are not equal. Interestingly, we are not assuming that a and b are *very* different; it is perfectly fine for them to differ only in a single character. So let us assume, without loss of generality, that[2] $a_s \neq b_s$.

Recalling (14.1.2), we must analyze the probability that

$$\begin{aligned} \Big(\sum_{i=1}^{s} a_i X_i + Y \Big) \bmod p &= \alpha \\ \Big(\sum_{i=1}^{s} b_i X_i + Y \Big) \bmod p &= \beta. \end{aligned}$$

Moving most of the sum to the right-hand side, these equations are equivalent to

$$\begin{aligned} (a_s X_s + Y) \bmod p &= \underbrace{\Big(\alpha - \sum_{i=1}^{s-1} a_i X_i \Big) \bmod p}_{c} \\ (b_s X_s + Y) \bmod p &= \underbrace{\Big(\beta - \sum_{i=1}^{s-1} b_i X_i \Big) \bmod p}_{d}. \end{aligned} \tag{14.1.3}$$

The key point is: no matter what values are on the right-hand side, Fact A.2.16 implies that there is a unique pair $X_s, Y \in [\![p]\!]$ such that these equations hold. Since X_s and Y are chosen uniformly and independently,

$$\begin{aligned} \Pr[\, h(a) = \alpha \wedge h(b) = \beta \,] &= \Pr[\,\text{equations (14.1.3) hold}\,] \\ &= \frac{\#\text{ solutions } (X_s, Y)}{\#\text{ values for the pair } (X_s, Y)} \\ &= \frac{1}{p^2}, \end{aligned}$$

since Fact A.2.16 says that there is a unique solution. $\square$

[2] At this point we require that $a_s, b_s \in [\![p]\!]$, which is why we require that $p \geq q$.

14.2 Binary linear hashing

The linear hashing approach allows us to hash a string of characters in $[\![q]\!]$ to a single character in $[\![p]\!]$, where $p \geq q$ and p is prime. It is neat that the approach allows an arbitrary prime, but finding a prime can be tedious. In this section, we focus on the special case $p = q = 2$, in which the input is a string of bits and the hash value is just a single bit.

It may be helpful to think how the arithmetic operations modulo 2 correspond to familiar Boolean operations.

- *Multiplication:* If $a, b \in \{0, 1\}$ then the multiplication $ab \bmod 2$ is just the Boolean And operation. This can be checked exhaustively.

$$\begin{array}{rrrr} (0 \cdot 0) \bmod 2 = 0 & (0 \cdot 1) \bmod 2 = 0 & (1 \cdot 0) \bmod 2 = 0 & (1 \cdot 1) \bmod 2 = 1 \\ 0 \wedge 0 = 0 & 0 \wedge 1 = 0 & 1 \wedge 0 = 0 & 1 \wedge 1 = 1 \end{array}$$

- *Addition:* If $a, b \in \{0, 1\}$ then the addition $(a + b) \bmod 2$ is just the Boolean Xor operation. This can be checked exhaustively.

$$\begin{array}{rrrr} (0 + 0) \bmod 2 = 0 & (0 + 1) \bmod 2 = 1 & (1 + 0) \bmod 2 = 1 & (1 + 1) \bmod 2 = 0 \\ 0 \oplus 0 = 0 & 0 \oplus 1 = 1 & 1 \oplus 0 = 1 & 1 \oplus 1 = 0 \end{array}$$

In these equations above, the only place where the $\bmod 2$ plays any role is $(1 + 1) \bmod 2 = 0$.

Building the hash function. When we work with the prime $p = 2$, all hash values lie in $[\![2]\!] = \{0, 1\}$, which seems too small to be useful. Usually we want to hash to a much bigger set, say $[\![k]\!] = \{0, \ldots, k - 1\}$. An elegant approach exists for the case that k is power of two. We adapt the sampling approach of Section 2.1.1 as follows.

- First randomly and independently choose $\lg k$ hash functions, denoted $h_1, \ldots, h_{\lg k}$. Each of these is of the form (14.1.1) with $p = 2$, so it outputs a single bit. More explicitly, each h_j is of the form

$$h_j(a) \;=\; \Big(\sum_{i=1}^{s} a_i X_i + Y \Big) \bmod 2, \tag{14.2.1}$$

  where $X_1, \ldots, X_s, Y$ are independent and uniform random bits.

- The combined hash function h produces a value in $[\![k]\!]$ by concatenating the outputs of $h_1, \ldots, h_{\lg k}$.

Theorem 14.2.1. The combined hash function h is strongly universal.

Proof. Let a and b be distinct binary strings of length s. Their combined hash values are:

$$\begin{aligned} h(a) &= h_1(a) h_2(a) \ldots h_{\lg k}(a) \\ h(b) &= h_1(b) h_2(b) \ldots h_{\lg k}(b) \end{aligned}$$

Let α and β be arbitrary integers in $[\![k]\!]$, whose binary representations are written

$$\begin{aligned} \alpha &= \alpha_1 \alpha_2 \ldots \alpha_{\lg k} \\ \beta &= \beta_1 \beta_2 \ldots \beta_{\lg k}. \end{aligned}$$

Then

$$\begin{aligned}
&\Pr[\ h(a)=\alpha\ \wedge\ h(b)=\beta\ ] \\
&\quad=\ \Pr\big[\ \big(h_1(a)=\alpha_1\wedge\cdots\wedge h_{\lg k}(a)=\alpha_{\lg k}\big)\ \wedge\ \big(h_1(b)=\beta_1\wedge\cdots\wedge h_{\lg k}(b)=\beta_{\lg k}\big)\ \big] \qquad \text{(bitwise equality)}\\
&\quad=\ \prod_{j=1}^{\lg k}\Pr[\,h_j(a)=\alpha_j\wedge h_j(b)=\beta_j\,] \qquad \text{(the hash functions are independent)}\\
&\quad=\ \prod_{j=1}^{\lg k}\frac{1}{2^2} \qquad \text{(each } h_j \text{ satisfies (14.1.2) with } p=2)\\
&\quad=\ \frac{1}{2^{2\lg k}}\ =\ \frac{1}{k^2}.
\end{aligned}$$

This shows that h satisfies the condition (14.1.2) with k instead of p, and therefore h is strongly universal. □

Question 14.2.2. How much space does it take to represent h, as a function of s, the string length, and k, the range of the outputs?

Answer.

Referring to (14.2.1), each hash function h_i can be represented using the $s+1$ random bits $X_1,\ldots,X_s,Y$. Since there are $\lg k$ such functions, the total is $(s+1)\cdot\lg k = O(s\log k)$ bits.

Question 14.2.3. What is the collision probability for h?

Answer.

Since h is strongly universal, we have

$$\Pr[\,h(a)=h(b)\,]\ =\ \sum_{\alpha\in[\![k]\!]}\Pr[\,h(a)=\alpha\wedge h(b)=\alpha\,]\ =\ 1/k$$

by the same argument as Corollary 14.1.8.

Problem to Ponder 14.2.4. How would you implement this in your favourite programming language (C, Java, etc.)?

14.3 Polynomial hashing

Hash functions might violate the strongly universal condition but still be useful. In this section we consider such a hash function called a ***polynomial hash***.

Suppose we want a hash function whose inputs are bit strings of length s, and whose outputs are values in $[\![p]\!]$, where p is prime. Whereas a linear hash function has s different random values $X_1,\ldots,X_s\in[\![p]\!]$ as its mixing parameters, the polynomial hash saves space by generating a single random value $X\in[\![p]\!]$ then using its *powers* $X^1,\ldots,X^s$ as the mixing parameters. The hash of a bitstring $a=a_1a_2\ldots a_s$ is defined to be

$$h(a)\ =\ \Big(\sum_{i=1}^{s}a_iX^i\Big)\bmod p.$$

The key analysis of a polynomial hash function is the following. Note that the theorem says nothing unless $p>s$.

Theorem 14.3.1 (Collision probability). Let a, b be *different* bit strings a, b of length s. If X is chosen uniformly at random in $[\![p]\!]$ then

$$\Pr[\, h(a) = h(b) \,] \;\leq\; \frac{s}{p}.$$

Question 14.3.2. How much space does it take to represent h, as a function of s, the string length, and p, the range of the outputs?

Answer.

Since $X \in [\![p]\!]$, it can be represented using $O(\log p)$ bits. That is all that is needed to represent h. Remarkably, the amount of space does not depend on s.

Let us now compare these properties of the polynomial hash to the linear hash.

	Output Values	Bits of space	Collision Probability
Linear Hash	$[\![p]\!]$, where p is prime	$O(s \log p)$	$\frac{1}{p}$ (Corollary 14.1.8)
Binary Linear Hash	$[\![k]\!]$, where k is a power of 2	$O(s \log k)$	$\frac{1}{k}$ (Question 14.2.3)
Polynomial Hash	$[\![p]\!]$, where p is prime	$O(\log p)$	$\leq \frac{s}{p}$ (Theorem 14.3.1)

Figure 14.1: Comparing various hash functions for strings of length s.

14.3.1 Analysis

To prove Theorem 14.3.1 we will need the following simple theorem. It is like a theorem you learned in high school about roots of polynomials, but modified to work modulo p.

Fact A.2.14 (Roots of polynomials). Let $g(x) = \sum_{i=1}^{d} c_i x^i$ be a polynomial of degree at most d in a single variable x. Let p be a prime number. We assume that at least one coefficient satisfies $c_i \bmod p \neq 0$. Then there are at most d solutions to

$$g(x) \bmod p = 0 \quad \text{and} \quad x \in [\![p]\!].$$

Such solutions are usually called *roots*.

Proof of Theorem 14.3.1. We will work with the polynomial

$$g(X) \;=\; \sum_{i=1}^{s} a_i X^i - \sum_{i=1}^{s} b_i X^i \;=\; \sum_{i=1}^{s} \underbrace{(a_i - b_i)}_{c_i} X^i.$$

Taking the mod, we have

$$g(X) \bmod p \;=\; \big(h(a) - h(b)\big) \bmod p. \tag{14.3.1}$$

The coefficient c_i of g is the *difference* of the corresponding input bits a_i and b_i. Since the bit strings a, b are different, there must exist an index i where $a_i \neq b_i$. This implies that c_i is either $+1$ or -1; in either case, $c_i \bmod p \neq 0$. Since the conditions of Fact A.2.14 are satisfied, we have

$$\begin{aligned}
\Pr[\,h(a)=h(b)\,] &= \Pr[\,g(X) \bmod p = 0\,] && \text{(by (14.3.1))}\\
&= \frac{\text{number of solutions}}{\text{number of choices for } X} && \text{(since } X \text{ is uniform)}\\
&\leq \frac{s}{p} && \text{(by Fact A.2.14).} \qquad \square
\end{aligned}$$

Notes

There are various constructions of low-space hash functions in the literature. The constructions that we have presented are as follows.

- Section 14.1: (Strong)-universal hashing via linear hashing. This appears in (Motwani and Raghavan, 1995, Section 8.4.4) and (Mitzenmacher and Upfal, 2005, Section 13.3.2).
- Section 14.2: Binary linear hashing. This is also known as matrix multiplication hashing. It can be found in (Motwani and Raghavan, 1995, Exercise 8.21).
- Section 14.3: Polynomial hashing. This is attributed to unpublished work of Rabin and Yao in 1979. It can be found in (Kushilevitz and Nisan, 1997, Example 3.5) and Theorem 5 of this survey.

14.4 Exercises

Exercise 14.1. Let p be a prime number. Let $v \in [\![p]\!]^s$ be a vector with at least one non-zero entry. Let M be a matrix with r rows and s columns such that every entry is chosen uniformly and independently from $[\![p]\!]$. We define the product of M and v, modulo p, to be the vector $w \in [\![p]\!]^r$ defined by

$$w_i \;=\; \Big(\sum_{j=1}^{s} M_{i,j} v_j \Big) \bmod p \qquad \forall i \in [r].$$

Prove that w is a uniformly random vector in $[\![p]\!]^r$.

Exercise 14.2. Suppose $h : \{\text{Keys}\} \to [\![p]\!]$ be a strongly universal hash function, where p is a prime. Let A, B be non-empty subsets of $[\![p]\!]$ that do not equal all of $[\![p]\!]$. Fix some key x. Prove that the events "$h(x) \in A$" and "$h(x) \in B$" cannot be independent.

Exercise 14.3. Let $n = p_1 \cdot p_2$ where p_1, p_2 are prime numbers and $p_1 \leq p_2$. Design a strongly universal hash function h mapping $[\![p_1]\!]$ to $[\![n]\!]$ such that there are at most n^2 different functions that h could be.

Exercise 14.4. Recall from Corollary 14.1.6 and Corollary 14.1.8 that a strongly universal hash function $h : [\![n]\!] \to [\![n]\!]$ satisfies the following two properties.

$$\text{Hash values are uniform:} \qquad \Pr[\, h(x) = y \,] \;=\; 1/n \quad \forall x, y \in [\![n]\!] \tag{14.4.1}$$
$$\text{Collisions are uniform:} \qquad \Pr[\, h(x) = h(y) \,] \;=\; 1/n \quad \forall x, y \in [\![n]\!] \tag{14.4.2}$$

The converse is not true. Show that there exists a random hash function $h : [\![3]\!] \to [\![3]\!]$ satisfying both (14.4.1) and (14.4.2), but h is *not* strongly universal.

Exercise 14.5. Let S_1 and S_2 be arbitrary finite sets, and let $n \geq 2$ be an integer. Suppose that

$$h_1 : S_1 \to [\![n]\!]$$
$$h_2 : S_2 \to [\![n]\!]$$

are both strongly universal hash functions and are independent. Using only h_1 and h_2 as the source of randomness, make a new strongly universal hash function

$$h : S_1 \times S_2 \to [\![n]\!].$$

That is, h must satisfy that

$$\Pr[\, h([a_1, a_2]) = \alpha \,\wedge\, h([b_1, b_2]) = \beta \,] \;=\; \frac{1}{n^2}$$

for any values $\alpha, \beta \in [\![n]\!]$, and for any distinct vectors $[a_1, a_2]$ and $[b_1, b_2]$, where $a_1 \in S_1$, $a_2 \in S_2$, $b_1 \in S_1$, $b_2 \in S_2$.

Chapter 15

Applications of Low-Space Hashing

15.1 Equality testing

Zoodle hosts a website at which users can download the genetic sequence of many different vegetables. A single sequences can be many gigabytes long[1]. Suppose that a user already has a genome, perhaps downloaded from another site. How can the user verify that this genome is identical to the file on the server?

For concreteness, say that the file is s bits long, the server has the bit string $a = (a_1, \ldots, a_s)$ and the client has the bit string $b = (b_1, \ldots, b_s)$. In the ***equality testing problem***, the client and the server must communicate to verify that their files are equal.

Trivial approach. The most obvious approach for this problem is for the server to send the entire file to the client, who can then compare them. This takes s bits of communication, which is far too much to be considered efficient.

Checksums. In practice, many systems would compute a *checksum* or a hash, such as CRC-32 or MD5, for example. It is worth stating explicitly what the goal of a checksum is:

- For *most* inputs a and b, the checksum will detect if they are not identical. However, it will fail for *some* inputs a and b that collide.

This guarantee is too weak. As discussed in Section 12.1, for any fixed checksum function, it is possible to create different inputs a and b that collide.

Randomized guarantees. We would like an equality testing algorithm that works for *all* possible inputs. We will aim for a guarantee of this sort:

- For *every* pair of inputs a and b, the algorithm will detect whether they are identical with constant probability.

[1]For example, the genome of the pea (*pisum sativum*) is approximately 4 gigabytes.

Algorithm 15.1 Equality testing using polynomial hashing.

```
function Server(bit string a = (a_1, ..., a_s))
    By brute force, find a prime p ∈ {2s, ..., 4s}
    Pick x ∈ ⟦p⟧ uniformly at random
    Compute h(a) = Σ_{i=1}^{s} a_i x^i mod p
    Send p, x and h(a) to the client
end function

function Client(bit string b = (b_1, ..., b_s))
    Receieve p, x and h(a) from the server
    Compute h(b) = Σ_{i=1}^{s} b_i x^i mod p
    if h(a) = h(b) then
        return True
    else
        return False
    end if
end function
```

Here the probability depends on the random numbers generated internally by the algorithm. This guarantee ensures that a failure of the algorithm is not due to undesirable inputs (a and b), but only due to unlucky random numbers.

15.1.1 A randomized solution

Instead of these deterministic hash functions, like MD5, we will use the polynomial hashing method of Section 14.3. The server will choose the hash function, then send the *hash function h itself* (represented by p and x) to the client, as well as the hash of a. The client now knows h, since it receives p and x, so it can compute the hash of b, and compare it to the hash of a. Pseudocode is shown in Algorithm 15.1.

Note that the server must find a prime number p satisfying

$$2s \;\leq\; p \;\leq\; 4s.$$

It might not be obvious that such a prime exists, but in fact one must exist. This is the statement of Bertrand's Postulate (Fact A.2.12).

Question 15.1.1. How can the server find such a prime p? How much time does it take?

Answer.

The simplest method is to try all $p \in \{2s, \ldots, 4s\}$, and to check if any integer up to $\sqrt{p}$ is a divisor of p. This takes time $O(s^{1.5})$. There are faster methods, such as the sieve of Eratosthenes.

15.1.2 Analysis

Theorem 15.1.2. The randomized protocol above solves the equality testing problem, using only $O(\log s)$ bits of communication. It has no false negatives, and false positive probability at most $1/2$.

Proof.

False negatives. If $a = b$ then $h(a) = h(b)$, so the client will not mistakenly output False.

False positives. Suppose that $a \neq b$. Then, using Theorem 14.3.1, we have

$$\begin{aligned}
\Pr[\,\text{client incorrectly outputs } \mathsf{True}\,] &= \Pr[\,h(a) = h(b)\,] \\
&\leq \frac{s}{p} \qquad \text{(by Theorem 14.3.1)} \\
&\leq \frac{s}{2s} = \frac{1}{2},
\end{aligned}$$

since we require $p \geq 2s$.

Number of bits. The server sends three values: p, x and $h(a)$. Recall that $p \leq 4s$, $x < p$ and $h(a) < p$. Each of them can be stored using $O(\log p) = O(\log s)$ bits, so the total number of bits communicated is $O(\log s)$. $\square$

Decreasing the Failure Probability. A false positive probability of $1/2$ is not too impressive. This can be improved by amplification, as explained in Section 1.3. If perform ℓ independent trials of the algorithm then, by Theorem 1.3.1, the false positive probability decreases to $2^{-\ell}$.

Notes

The number of bits communicated by Algorithm 15.1 is optimal, up to constant factors. Theorem 15.1.2 showed that $O(\log s)$ bits are communicated. It is known that *every* algorithm for this problem, randomized or not, that succeeds with constant probability requires $\Omega(\log s)$ bits of communication. Other references for this material include (Motwani and Raghavan, 1995, Section 7.5), (Kushilevitz and Nisan, 1997, Example 3.5) and (Cormen et al., 2001, Exercise 32.2-4).

15.1.3 Exercises

Exercise 15.1. Using the amplification approach described above, the equality test satisfies

$$\begin{aligned}
\text{False positive probability:} &\ \leq\ 2^{-\ell} \\
\text{Bits communicated:} &\ =\ O(\ell \cdot \log s).
\end{aligned}$$

This can be improved. Design a different approach achieving

$$\begin{aligned}
\text{False positive probability:} &\ \leq\ 2^{-\ell} \\
\text{Bits communicated:} &\ =\ O(\ell + \log s).
\end{aligned}$$

This new approach has the same $2^{-\ell}$ bound on false positives but communicates fewer bits because it depends additively rather than multiplicatively on ℓ.

Exercise 15.2. One complication with Algorithm 15.1 is that the server must find a prime number p. Let us now consider the following variant of that algorithm that avoids that complication.

Algorithm 15.2 An algorithm to test a equals b. Assume $s \geq 2$.

function SERVER(bit string $a = (a_1, \ldots, a_s)$)
  for $j = 1, \ldots, \lceil 32 \ln s \rceil$ **do**
    Pick a random number $p \in \{4s+1, \ldots, 20s\}$
    Pick $x \in \{0, \ldots, p-1\}$ uniformly at random
    Compute $h(a) = \sum_{i=1}^{s} a_i x^i \bmod p$
    Send p, x and $h(a)$ to the client
  end for
end function

function CLIENT(bit string $b = (b_1, \ldots, b_s)$)
  for $j = 1, \ldots, \lceil 32 \ln s \rceil$ **do**
    Receieve p, x and $h(a)$ from the server
    Compute $h(b) = \sum_{i=1}^{s} b_i x^i \bmod p$
    if $h(a) \neq h(b)$ **then return** False
  end for
  return True
end function

Part I. Show that this algorithm uses $O(\log^2(s))$ bits of communication.

Part II. Explain why this algorithm has no false negatives.

Assume that $a \neq b$. Say that an iteration is *good* if p is a prime in that iteration; otherwise, it is *bad*. Prove that the probability that *all* iterations are bad is at most $1/e^2$.

Hint: See the appendix.

Part III. Assume that $a \neq b$. Let $\mathcal{E}$ be the event that, in the first good iteration we have $h(a) = h(b)$, or that no good iteration exists. Prove that $\Pr[\mathcal{E}] \leq 1/2$.

Part IV. Explain why the algorithm has false positive rate at most $1/2$.

15.2 Count-Min Sketch

As another application of low-space hashing, let us consider the COUNTINGFILTER and COUNTMINSKETCH algorithms from Chapter 13. These are elegant algorithms, with the snag that they use a purely random function. Let see if we can address this issue by using a low-space hash function instead. In fact, the modification is extremely simple.

Recall our analysis of the COUNTINGFILTER algorithm in Theorem 13.4.1. We defined X_b to be the indicator of the event "$h(i) = h(b)$". The key step of the analysis is

$$\mathrm{E}[\, C[h(i)] \,] \;=\; \mathrm{E}\left[f_i + \sum_{b \neq i} X_b f_b \right] \;=\; f_i + \sum_{b \neq i} \mathrm{E}[\, X_b \,] f_b \;=\; f_i + \frac{1}{k} \sum_{b \neq i} f_b.$$

Here we have used that

$$\mathrm{E}[\, X_b \,] \;=\; \Pr[\, h(i) = h(b) \,] \;=\; \frac{1}{k} \qquad \forall b \neq i,$$

since we assume that h is a purely random function.

Algorithm 15.3 The CountingFilter, revised to use a low-space hash function.

global int $k = 2^{\lceil \lg(1/\epsilon) \rceil}$

class CountingFilter:
    hash function h
    array C

    ▷ Build the data structure by processing the stream of items
    Constructor ()
        Let C be an array of integers, with entries indexed by $[\![k]\!]$
        Let $h : [n] \rightarrow [\![k]\!]$ be a binary linear hash function
        for $t = 1, \ldots, m$
            Receive item a_t from the stream
            Increment $C[h(a_t)]$

    ▷ Estimates the frequency of an item.
    Query (int i)
        return $C[h(i)]$

This assumption is much stronger than necessary, since we are just using the probability of *two* elements colliding under the hash function. Even a universal hash function would suffice to have a small probability of two elements colliding!

The key changes that we need to make to the algorithm are using a binary linear hash function, which requires that k is a power of two. Recall from Definition A.2.3 that rounding $1/\epsilon$ up to a power of two yields the value $2^{\lceil \lg(1/\epsilon) \rceil}$.

The theorem analyzing this algorithm is identical to before.

Theorem 15.2.1. For every item $i \in [n]$,

$$\begin{aligned} &\text{Lower bound:} \quad f_i \leq \text{QUERY}(i) \\ &\text{Upper bound:} \quad \mathrm{E}[\,\text{QUERY}(i)\,] \leq f_i + \epsilon m. \end{aligned}$$

In fact, even the proof is identical to before. We are still using the fact, as above, that

$$\mathrm{E}[\,X_b\,] \;=\; \Pr[\,h(i) = h(b)\,] \;=\; \frac{1}{k} \qquad \forall b \neq i.$$

Now the reason it holds is not because h is a purely random function, but because h is a binary linear hash function. See Question 14.2.3 for this calculation.

Question 15.2.2. How much space does the CountingFilter algorithm now take?

15.3 Maximum cut via hashing

Recall the problem of finding a maximum cut in a graph, from Section 16.1. The input is an undirected graph $G = (V, E)$, where $n = |V|$. The objective is to solve

$$\max\{\, |\delta(U)| \,:\, U \subseteq V \,\}.$$

where $\delta(U) = \{ uv \in E : u \in U \text{ and } v \notin U \}$. Recall that OPT denotes the size of the maximum cut. We showed that a uniformly random set U cuts at least OPT/2 edges in expectation.

It turns out that we can achieve the same guarantee using a hash function to generate U. Why would you would want to do that? It is certainly a neat trick and, as we will see, it has some advantages.

Let us label the vertices with the integers in $[\![n]\!]$. These integers can be represented using $s = \lceil \lg n \rceil$ bits. We will use a single binary linear hash function with outputs in $\{0, 1\}$. The mixing parameters of this hash function are $X_1, \ldots, X_s, Y \in \{0, 1\}$. For an integer $a \in [\![n]\!]$ with bit representation $a = (a_1, \ldots, a_s)$, the hash value is

$$h(a) \;=\; \Big(\sum_{i=1}^{s} a_i X_i + Y \Big) \bmod 2.$$

The nice thing about this hashing approach is that it uses much less randomness. The original algorithm picks U uniformly at random, which requires n random bits. In contrast, randomly generating the mixing parameters for h requires only $s + 1 = \lceil \lg n \rceil + 1$ random bits.

The cut U is now chosen in a very simple way: it is simply the vertices that hash to zero. Formally,

$$U \;=\; \{ a \in [\![n]\!] : h(a) = 0 \}.$$

Algorithm 15.4 This algorithm gives a 1/2-approximation to the maximum cut via universal hashing.

```
function NewMaxCut(graph G = (V, E))
    Let n ← |V| and s ← ⌈lg n⌉
    Label the vertices by the integers ⟦n⟧
    Randomly choose mixing parameters X_1, ..., X_s, Y ∈ {0, 1}
    Create the binary linear hash function h with mixing parameters X_1, ..., X_s, Y
    return U = { i ∈ ⟦n⟧ : h(i) = 0 }
end function
```

Theorem 15.3.1. Let U be the output of NewMaxCut(). Then $\mathrm{E}[\,|\delta(U)|\,] \geq m/2 \geq \mathrm{OPT}/2$.

Proof. An edge uv is cut if and only if vertices u and v hash to different values, i.e., if they *don't* collide. Since h is a universal hash function, the probability of colliding is exactly 1/2. Thus,

$$\Pr[\, uv \text{ is cut} \,] \;=\; \Pr[\, h(u) \neq h(v) \,] \;=\; \frac{1}{2}.$$

Now, following the proof from Section 16.1,

$$\mathrm{E}[\,|\delta(U)|\,] \;=\; \sum_{uv \in E} \Pr[\, uv \text{ is cut} \,] \;=\; \frac{|E|}{2} \;\geq\; \frac{\mathrm{OPT}}{2}. \qquad \square$$

15.3.1 A deterministic algorithm

NewMaxCut() randomly generates a cut U satisfying $\mathrm{E}[\,|\delta(U)|\,] \geq \mathrm{OPT}/2$. However, this gives no guarantee of finding a large cut. In Section 16.1.1, we gave an algorithm with the different guarantee $\Pr[\,|\delta(U)| > 0.49 \cdot \mathrm{OPT}\,] \geq 0.01$, but this is still rather weak.

Today we will give an ironclad guarantee: there is a *deterministic* algorithm producing a cut U with $|\delta(U)| \geq \mathrm{OPT}/2$. The algorithm can be summarized in one sentence:

By brute force, try all binary linear hash functions until one gives a large enough cut.

Algorithm 15.5 This is deterministic algorithm gives 1/2-approximation to the maximum cut. It does not use any randomness whatsoever!

---

function DeterministicMaxCut(graph $G = (V, E)$)
  Let $n \leftarrow |V|$, $m \leftarrow |E|$, and $s \leftarrow \lceil \lg n \rceil$
  Label the vertices by the integers $[\![n]\!]$
  for all mixing parameters $X_1, \ldots, X_s, Y \in \{0, 1\}$ **do**
    Create universal hash function h with mixing parameters $X_1, \ldots, X_s, Y$
    Let $U = \{\ i \in [\![n]\!]\ :\ h(i) = 0\ \}$
    if $|\delta(U)| \geq m/2$ **then return** U
  end for
end function

Theorem 15.3.2. The DeterministicMaxCut algorithm produces a cut with $|U| \geq m/2 \geq \mathrm{OPT}/2$.

Proof. The only way the algorithm can fail is if all mixing parameters yield a cut with $|\delta(U)| < m/2$. If that were true then, when picking the mixing parameters randomly, we would have $\mathrm{E}[\,|\delta(U)|\,] < m/2$. This contradicts Theorem 15.3.1. □

Question 15.3.3. What is the runtime of this algorithm?

Answer.

The number of iterations of the for loop is $2^{s+1} = 2^{\lceil \lg n \rceil + 1} = O(n)$. Each iteration requires $O(n)$ time to build U and $O(m)$ time to calculate $|\delta(U)|$. The total is $O(nm)$ time.

15.3.2 A general technique

The DeterministicMaxCut algorithm employs a general principle known as the probabilistic method, which was pioneered by Paul Erdős. At a very basic level, the technique uses the following idea.

Observation 15.3.4. Let X be a random variable on a finite probability space. Suppose that $\mathrm{E}[\,X\,] \geq k$. Then there exists an outcome in the probability space for which $X \geq k$.

This is a trivial observation, but it is very powerful. In particular, it can be used to prove the existence of certain objects, for which we know no other way to prove their existence.

15.4 A hash table data structure

Hash tables are one of the fundamental data structures introduced in CPSC 221. In that class it was asserted that hash tables can store n items in $O(n)$ space with $O(1)$ query time. This assertion has two problems. First, the query time is supposedly the "expectation" for a single query, not the worst-case over all queries. Second, it requires a vague assumption that the hash values "look random". In this section we present a hash table data structure that addresses these two problems.

For simplicity, let us assume each key being hashed takes $O(\log n)$ bits, represented as a single int. This way the space of the hash function is $O(1)$ words and it can be evaluated in $O(1)$ time. It is easy to modify our discussion to allow keys that are arbitrary strings.

Algorithm 15.6 The BIRTHDAYTABLE is a static data structure to store a list of n keys using $O(n^2)$ space, and with contant-time queries.

```
class BirthdayTable:
    hash function h
    array T

    Constructor ( array Keys of ints, with entries indexed by [[n]])
        Find a prime p ∈ {n^2, ..., 20n^2} using Exercise 2.6
        repeat
            Randomly choose a linear hash function h mapping ints to [[p]]
            Create array T, whose entries are indexed by [[p]]
            for i ∈ [[n]]
                if T[h(Keys[i])] not empty then found collision
                T[h(Keys[i])] ← Keys[i]
        until no collisions

    Query (int key)
        if T[h(key)] = key
            return True
        else
            return False
```

15.4.1 A solution using too much space

First we present a data structure that uses $\Theta(n^2)$ space to store n keys, which is not very impressive. We call it a BIRTHDAYTABLE because its analysis exploits the birthday paradox.

Theorem 15.4.1.

- *Runtime:* CONSTRUCTOR takes $O(n^2)$ time in expectation.
  QUERY takes $O(1)$ time **in the worst case**.
- *Space:* The BIRTHDAYTABLE object takes $O(n^2)$ words.

Proof. For $i < j$, let $X_{i,j}$ be the indicator of the event "$h(\text{Keys}[i]) = h(\text{Keys}[j])$". In Corollary 14.1.8 we saw that the collision probability is

$$\mathrm{E}\,[\,X_{i,j}\,] \;=\; \Pr\,[\,h(\text{Keys}[i]) = h(\text{Keys}[j])\,] \;=\; \frac{1}{p}.$$

Let $X = \sum_{0 \le i < j < n} X_{i,j}$ be the total number of collisions. Then, just like in (7.2.2),

$$\mathrm{E}\,[\,X\,] \;=\; \sum_{0 \le i < j < n} \mathrm{E}\,[\,X_{i,j}\,] \;=\; \binom{n}{2} \cdot \frac{1}{p} \;<\; \frac{n^2}{2} \cdot \frac{1}{n^2} \;=\; \frac{1}{2}.$$

By Markov's inequality, $\Pr\,[\,X \ge 1\,] \le \frac{\mathrm{E}[\,X\,]}{1} < 1/2$. Taking the complement, the probability of *no* collisions is $\Pr\,[\,X = 0\,] > 1/2$.

Algorithm 15.7 The PerfectTable is a static data structure to store a list of n keys using $O(n)$ space, and with contant-time queries.

```
class PerfectTable:
    hash function h
    array T

    Constructor ( array Keys of ints, with entries indexed by [[n]])
    |   Find a prime p ∈ {n, ..., 20n} using Exercise 2.6
    |   Randomly choose a linear hash function h mapping ints to [[p]]
    |   ▷ Temporarily hash keys into linked lists
    |   Create temporary array L, indexed by [[p]], of linked lists
    |   for i ∈ [[n]]
    |   |   Insert Keys[i] into L[h(Keys[i])]
    |   ▷ Replace each linked list with a BirthdayTable
    |   Create array T of BirthdayTable objects, whose entries are indexed by [[p]]
    |   for i ∈ [[p]]
    |   |   T[i].Constructor(L[i])

    Query (int key)
    |   return T[h(key)].Query(key)
```

Runtime: Each iteration of the repeat loop succeeds with probability more than $1/2$. So the number of iterations until the first success is a geometric random variable with expectation $O(1)$. Each iteration reqires $O(p) = O(n^2)$ time to build the table and $O(n)$ time to insert the keys.

Space: The table T requires $O(p) = O(n^2)$ words. The hash function h requires $O(1)$ words. □

Problem to Ponder 15.4.2. It is possible to improve the runtime to $O(n)$, even though T has size $\Theta(n^2)$, by some data structure trickery. How can you do this?

15.4.2 Perfect hashing

Let us try to improve the BirthdayTable by hashing down to $\Theta(n)$ locations instead of $\Theta(n^2)$ locations. The trouble is that now there will be collisions. The naive way to resolve collisions is by chaining: storing a linked list of colliding items. A better way to resolve collisions is by using BirthdayTable objects!

Theorem 15.4.3.

- *Runtime:* Constructor takes $O(n)$ time in expectation.
  Query takes $O(1)$ time **in the worst case**.
- *Space:* The PerfectTable object takes $O(n)$ words in expectation.

To understand the significance of this theorem, it is instructive to make the following comparison.

- *Hashing with chaining.* For every fixed i, the expected runtime of Query(Keys[i]) is $O(1)$. However, even with a purely random function, Section 7.6 suggests that the *longest* chain might have length $\Omega(\log n/\log\log n)$. So there is likely to exist some key for which QUERY has runtime $\Omega(\log n/\log\log n)$.

- *Perfect hashing.* After the table is built, *every* call to QUERY always takes $O(1)$ time. This is a much stronger guarantee. Furthermore, the impractical purely random functions are not needed, and we can use linear hashing instead.

Proof. Let $n_i = |L[i]|$ be the number of items that hash to location i. The space of the i^{th} BIRTHDAYTABLE is $O(n_i^2)$, so we must analyze n_i^2. Fix $i \in [\![p]\!]$, and let X_j be the indicator of the event "$i = h(\text{Keys}[j])$". A familiar idea is to decompose n_i into indicators as

$$n_i \;=\; \sum_{j\in[\![n]\!]} X_j. \tag{15.4.1}$$

Armed with this decomposition, we can analyze n_i^2:

$$\begin{aligned}
\mathrm{E}\left[\,n_i^2\,\right] &= \mathrm{E}\left[\Big(\sum_{j\in[\![n]\!]} X_j\Big)\Big(\sum_{\ell\in[\![n]\!]} X_\ell\Big)\right] && \text{(each sum equals n_i by (15.4.1))}\\
&= \sum_{j,\ell\in[\![n]\!]} \mathrm{E}\left[\,X_j X_\ell\,\right] && \text{(by linearity of expectation)}\\
&= \sum_{\substack{j,\ell\in[\![n]\!]\\ j\neq \ell}} \Pr\left[\,h(\text{Keys}[j]) = i = h(\text{Keys}[\ell])\,\right] + \sum_{j\in[\![n]\!]} \Pr\left[\,h(\text{Keys}[j]) = i\,\right]\\
&= \sum_{\substack{j,\ell\in[\![n]\!]\\ j\neq \ell}} \frac{1}{p^2} + \sum_{j\in[\![n]\!]} \frac{1}{p} && \text{(since h is strongly universal)}\\
&< \frac{n^2}{p^2} + \frac{n}{p} \;\leq\; 2,
\end{aligned} \tag{15.4.2}$$

since $p \geq n$.

Space: The hash function h takes $O(1)$ words. The table T takes $O(p)$ words. Each BIRTHDAYTABLE takes at most cn_i^2 words, for some constant c. The total expected space for the BIRTHDAYTABLEs is

$$\mathrm{E}\left[\sum_{i\in[\![p]\!]} \text{size of } i^{\text{th}} \text{ BIRTHDAYTABLE}\right] \;\leq\; \sum_{i\in[\![p]\!]} \mathrm{E}\left[\,cn_i^2\,\right] \;<\; \sum_{i\in[\![p]\!]} 2c \;=\; O(p), \tag{15.4.3}$$

by (15.4.2). Since $p \leq 20n$, the total space is $O(n)$ words.

Runtime: Building the linked lists takes $O(n)$ time. For the i^{th} BIRTHDAYTABLE, calling the CONSTRUCTOR takes at most $O(n_i^2)$ time in expectation. By the same calculation as in (15.4.3), the expected runtime is $O(\sum_{i\in[\![p]\!]} n_i^2) = O(p) = O(n)$ ☐

Question 15.4.4. How can we improve the space to $O(n)$ words in the worst case?

Answer.

Simply use a repeat loop in the CONSTRUCTOR that rebuilds the whole data structure if $\sum_{i\in[\![p]\!]} n_i^2$ is too big. As in the proof of Theorem 15.4.1, the expected number of iterations is $O(1)$.

Notes

The algorithm is originally due to Fredman, Komlós and Szemerédi. Further discussion is in (Cormen et al., 2001, Section 11.5), (Motwani and Raghavan, 1995, Section 8.5) and (Mitzenmacher and Upfal, 2005, Section 13.3.3).

15.5 Exercises

Exercise 15.3 **Matching strings.** Let $A[1..s]$ and $B[1..n]$ be bitstrings, where $s \leq n$. We will design a randomized algorithm that tests if A occurs as a substring (of consecutive characters) within string B. That is, does there exist i such that $A = B[i..s+i-1]$?

First modify our polynomial hash function, basically by reversing the order of the string. That is, for a string $A[1..s]$, we define

$$h(A) \;=\; \Big(\sum_{i=1}^{s} A[i] \cdot x^{s-i+1} \Big) \bmod p.$$

Algorithm 15.8 An algorithm to test if A is a substring of B

```
1: function FindMatch(A[1..s], B[1..n])
2:     Assume we already know a prime p in [2n^2, 4n^2]
3:     Pick x ∈ {0, ..., p−1} uniformly at random
4:     Compute α ← h(A)
5:     Compute β ← h(B[1..s])
6:     for i = 1, ..., n − s + 1 do
7:         ▷ Assert: β = h(B[i..i + s − 1])
8:         if α = β then return True
9:         if i < n − s + 1 then
10:            β ← h(B[i + 1..i + s])    ▷ This can be done very efficiently!
11:        end if
12:    end for
13:    return False
14: end function
```

Assumption: the basic arithmetic operations (addition, subtraction, multiplication, division, mod) **on integers with** $O(\log n)$ **bits** take $O(1)$ time.

Part I. Explain how to compute α and β in $O(s)$ time in lines 4 and 5.

Note: don't forget the assumption above.

Part II. Explain how to implement line 10 in $O(1)$ time. (You should use the old value of β to compute the new value of β.)

Hint: See Fact A.2.11 in the book.

Part III. Give a big-O bound for the total runtime, ignoring the time in line 2. (Line 2 can be implemented and analyzed along the lines of Exercise 2.6.)

Part IV. Prove that this algorithm has no false negatives and false positive probability at most $1/2$. (Thus, it is a coRP-algorithm.)

Part IV

Other Topics

Chapter 16

Graphs

16.1 Maximum cut

Suppose we have a graph whose vertices represent students and edges represent friendships. A teacher might want to partition the students into two groups so that many pairs of friends are split into different groups. This is an example of the ***maximum cut problem***, which has been extensively studied in the fields of combinatorial optimization and algorithm design.

The problem is formally defined as follows. Let $G = (V, E)$ be an undirected graph. For $U \subseteq V$, define

$$\delta(U) \;=\; \{\, uv \in E \,:\, u \in U \text{ and } v \notin U \,\}\,.$$

The set $\delta(U)$ is called the ***cut*** determined by vertex set U. The Max Cut problem is to solve

$$\max\{\, |\delta(U)| \,:\, U \subseteq V \,\}\,.$$

This problem is NP-hard, so we cannot hope for an algorithm that solves it exactly on all instances. Instead, we will design an algorithm that finds a cut that is close to the maximum.

More precisely, let OPT denote the size of the maximum cut. An algorithm for this problem is called an α***-approximation*** if it always outputs a set U for which $|\delta(U)| \geq \alpha \cdot \mathrm{OPT}$. Naturally we want α as large as possible, but we must have $\alpha \leq 1$. If the algorithm is randomized, we will be content for this guarantee to hold in expectation.

Algorithm 16.1 presents an extremely simple randomized algorithm that is a 1/2-approximation. Bizarrely, the algorithm does not even look at the edges of G, and instead just returns a uniformly random set U. Note that, due to Exercise 3.1, the set U can be generated by adding each vertex to U with probability 1/2.

Theorem 16.1.1. Let U be the output of the algorithm. Then $\mathrm{E}[\,|\delta(U)|\,] \geq \mathrm{OPT}/2$.

References: The algorithm first appears in a paper of Erdős (Erdős, 1967). See also (Motwani and Raghavan, 1995, Theorem 5.1), (Mitzenmacher and Upfal, 2005, Theorem 6.3).

Proof. For every edge $uv \in E$, let X_{uv} be the indicator random variable which is 1 if $uv \in \delta(U)$. Then, by linearity of expectation,

$$\mathrm{E}[\,|\delta(U)|\,] \;=\; \mathrm{E}\left[\sum_{uv \in E} X_{uv}\right] \;=\; \sum_{uv \in E} \mathrm{E}[\,X_{uv}\,] \;=\; \sum_{uv \in E} \Pr[\,X_{uv} = 1\,]\,.$$

Algorithm 16.1 A randomized 1/2-approximation for Max Cut.

function MaxCut(graph $G = (V, E)$)
  Let U be a uniformly random subset of V ▷ See Exercise 3.1.
  return U
end function

Now we note that

$$\begin{aligned}
\Pr[\, X_{uv} = 1 \,] &= \Pr[\ (u \in U \wedge v \notin U) \ \vee \ (u \notin U \wedge v \in U)\ ] \\
&= \Pr[\, u \in U \wedge v \notin U \,] \ + \ \Pr[\, u \notin U \wedge v \in U \,] \quad \text{(union of disjoint events, Fact A.3.7)} \\
&= \Pr[\, u \in U \,] \cdot \Pr[\, v \notin U \,] \ + \ \Pr[\, u \notin U \,] \cdot \Pr[\, v \in U \,] \quad \text{(independence)} \\
&= 1/2.
\end{aligned}$$

Thus $\mathrm{E}[\, |\delta(U)| \,] = |E|/2 \geq \mathrm{OPT}/2$, since clearly OPT cannot exceed $|E|$. □

16.1.1 The probability of a large cut

Above we have shown that the algorithm outputs a cut whose *expected size* is large. But how likely is it that $|\delta(U)| \geq \mathrm{OPT}/2$? Perhaps we have some undesirable scenario, like

$$\begin{aligned}
\Pr[\, |\delta(U)| \leq \mathrm{OPT}/100 \,] &= 0.99 \\
\Pr[\, |\delta(U)| \geq \mathrm{OPT}/2 \,] &= 0.01.
\end{aligned}$$

We can use the Reverse Markov inequality (Exercise 8.7) to show that $|\delta(U)|$ is likely to be quite large. Let $Z = |\delta(U)|$, $d = \mathrm{OPT}$ and $c = (\frac{1}{2} - \epsilon)\mathrm{OPT}$. Note that Z is never larger than d. Then

$$\begin{aligned}
\Pr[\, |\delta(U)| \leq (1/2 - \epsilon)\mathrm{OPT} \,] &= \Pr[\, Z \leq c \,] && \text{(definition of } Z \text{ and } c\text{)} \\
&\leq \frac{d - \mathrm{E}[\, Z \,]}{d - c} && \text{(Reverse Markov inequality)} \\
&= \frac{\mathrm{OPT} - \mathrm{OPT}/2}{\mathrm{OPT} - (\frac{1}{2} - \epsilon)\mathrm{OPT}} && \text{(recall } \mathrm{E}[\, |\delta(U)| \,] = \mathrm{OPT}/2\text{)} \\
&= \frac{1/2}{1/2 + \epsilon} \ = \ (1 + 2\epsilon)^{-1} && \text{(simplifying).}
\end{aligned}$$

To judge whether this guarantee is any good, consider setting $\epsilon = 0.01$. Then the probability of a small cut is

$$\begin{aligned}
& \Pr[\, |\delta(U)| \leq 0.49 \cdot \mathrm{OPT} \,] \ \leq \ \frac{1}{1.02} \ < \ 0.99. \\
\text{Equivalently,} \quad & \Pr[\, |\delta(U)| > 0.49 \cdot \mathrm{OPT} \,] \ \geq \ 0.01.
\end{aligned}$$

This bound is very weak, but at least the probability of a large cut is some positive constant.

Problem to Ponder 16.1.2. How might we find a cut U satisfying $|\delta(U)| > 0.49 \cdot \mathrm{OPT}$ with 99% probability instead of with 1% probability?

16.2 Minimum cut

In Section 16.1 we discussed the problem of finding a maximum cut in a graph. That problem is NP-complete, so we presented a very simple algorithm that produces a 1/2-approximate solution. Now we will discuss the ***minimum cut problem***, which can be solved exactly in polynomial time. We will present a very simple randomized algorithm for this problem.

This algorithm has many appealing properties.

- It illustrates that non-trivial optimization problems can sometimes be solved by very simple algorithms.
- The analysis is quite interesting.
- It has strong implications about the structure of graphs, which will be useful in Chapter 16.3.

16.2.1 Definition

Let $G = (V, E)$ be an undirected graph. As before, for every $U \subseteq V$ we define

$$\delta(U) = \{\, uv \in E \,:\, u \in U \text{ and } v \notin U \,\}.$$

The *Min Cut* problem (or *Global Min Cut* problem) is to solve

$$\min \{\, |\delta(U)| \,:\, U \subseteq V \text{ where } U \neq \emptyset \text{ and } U \neq V \,\}.$$

Here we are minimizing over all subsets U of the vertices, except for $U = \emptyset$ and $U = V$ because those two sets always have $|\delta(U)| = 0$.

Comparison to s-t cuts. You should not confuse the Global Min Cut problem and the *Min s-t Cut problem*. In the latter problem, there are two distinguished vertices $s, t \in V$ and we must solve

$$\min \{\, |\delta(U)| \,:\, U \subseteq V \text{ s.t. } s \in U,\ t \notin U \,\}.$$

This problem can be solved by network flow techniques, since the Max-Flow Min-Cut theorem tells us that the solution equals the maximum amount of flow that can be sent from s to t. (See (Kleinberg and Tardos, 2006, Chapter 7) or (Cormen et al., 2001, Chapter 26).)

Question 16.2.1. How can you solve the Global Min Cut problem using the Min s-t Cut problem?

Answer.

The solution to the Min Cut problem equals the minimum over all pairs $s, t \in V$ of the solution to the Min s-t Cut problem. See, e.g., (Kleinberg and Tardos, 2006, Theorem 13.4).

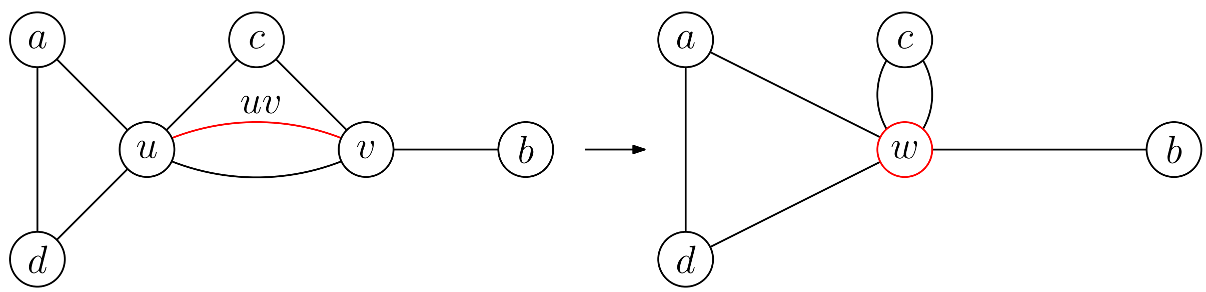
We will present a remarkable randomized algorithm for solving the Min Cut problem that avoids network flow techniques entirely. It uses only one simple idea: contracting edges in the graph.

16.2.2 Edge Contractions

Let $G = (V, E)$ be a multigraph, meaning that we allow E to contain multiple "parallel" edges with the same endpoints. Suppose that $uv \in E$ is an edge. Let us now define what it means to ***contract*** an edge.

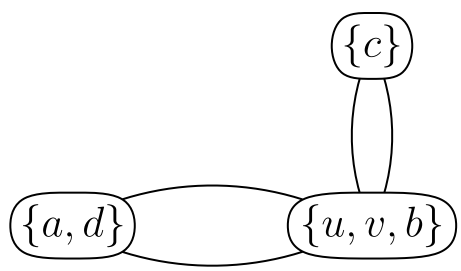
Algorithm 16.2 Contracting an edge uv means to apply the following operations.

1: Add a new vertex w.
2: For every edge xu or xv, add a new edge xw. This can create new parallel edges, because it might be the case that xu and xv *both* existed, in which case we will create *two* new edges xw.
3: The vertices u and v are deleted, together with any edges that are incident on them.
4: All self-loops are removed.



The graph that results from contracting the edge uv is written G/uv. This process essentially "merges" the two vertices u and v into a "supervertex" w which corresponds to the pair of vertices $\{u, v\}$. If we perform many contraction operations, we can think of each vertex in the resulting graph as being a "supervertex" that contains many vertices from the original graph.

The following figure shows the result of contracting the edges uv, vb and ad. In each supervertex we show the set of vertices from the original graph that were contracted together to form the supervertex.



Suppose we perform several contractions in graph G, producing the graph H.

Observation 16.2.2. Every cut in H corresponds to some cut in G.

Proof sketch. Each contraction operation basically "glues" two vertices together, forcing them to be on the same side of the cut. Every cut in H is just a cut in G that keeps all glued pairs together. □

Corollary 16.2.3. The size of a minimum cut in H is at least the size of a minimum cut in G.

Proof. Consider any minimum cut in H. By Observation 16.2.2, this is also a cut in G. This might be a minimum cut in G, but G might have an even smaller cut. □

In the original example above, the minimum cut is 1 due to the cut $\delta(\{b\})$, but in the contracted example the minimum cut is 2.

16.2.3 The Contraction Algorithm

The contraction algorithm, shown in Algorithm 16.3, is one of the most famous randomized algorithms. It outputs a cut, which may or may not be a minimum cut.

Algorithm 16.3 The Contraction Algorithm. Assume that the graph G is connected.

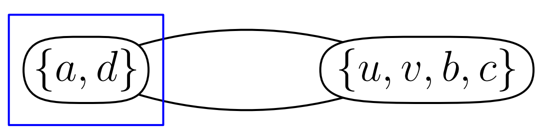
function ContractionAlg(graph $G = (V, E)$)
    while the graph has more than two supervertices remaining **do**
        Pick an edge e uniformly at random, and contract it
    end while
    Let u be one of the two remaining vertices. (It does not matter which).
    Let U be the vertices of the original graph contained in u.
    Output the cut $\delta(U)$.
end function

Remarkably, although the algorithm seems to do nothing at all, it has decent probability of outputting a minimum cut.

Theorem 16.2.4. Suppose that G is connected, and fix any minimum cut C. The contraction algorithm outputs C with probability at least $\frac{2}{n(n-1)}$.

References: (Motwani and Raghavan, 1995, Section 1.1), (Mitzenmacher and Upfal, 2005, Section 1.4), (Kleinberg and Tardos, 2006, Section 13.2), (Sen and Kumar, 2019, Section 10.5), Wikipedia.

Continuing our example above, the algorithm might decide to contract one of the edges between $\{c\}$ and $\{u, v, b\}$ (say, the edge cv in the original graph). The resulting graph is shown below. Then the algorithm outputs the cut $\delta(\{a, d\})$, which is the same as the cut $\delta(\{u, v, b, c\})$, and which contains two edges. However, this is not a minimum cut of G as the cut $\delta(\{b\})$ contains just one edge.



$\delta(\{a, d\})$

16.2.4 Probability of Success

At first glance it seems that the algorithm has a fairly small probability of outputting a min cut. But using the "Probability Amplification" technique we can boost the probability of success.

Question 16.2.5. Should we use Probability Amplification for one-sided error, or for two-sided error?

Corollary 16.2.6. Fix any $\delta > 0$. Suppose we run the Contraction Algorithm $n^2 \ln(1/\delta)$ times and output the smallest cut that it finds. Then this will output a min cut with probability at least $1 - \delta$.

Proof. Obviously we cannot output anything *smaller* than a min cut. So as long as we find a min cut on at least one of the trials, then the algorithm will succeed.

The probability that the Contraction Algorithm finds a minimum cut in one trial is at least $\frac{2}{n(n-1)} > \frac{1}{n^2}$. So the probability that it *fails* in *all* trials is less than

$$\Big(1 - \frac{1}{n^2}\Big)^{n^2 \ln(1/\delta)} \;\leq\; \Big(e^{-1/n^2}\Big)^{n^2 \ln(1/\delta)} \;=\; \delta,$$

by Fact A.2.5. $\square$

16.2.5 The proof

Before proving the theorem we need two more preliminary claims.

Claim 16.2.7. Let G be a graph with n vertices in which the minimum size of a cut is k. Then G must have at least $nk/2$ edges.

Proof. Every vertex must have degree at least k, otherwise the edges incident on that vertex would constitute a cut of size less than k. Any graph where the minimum degree is at least k must have at least $nk/2$ edges, since the sum of the vertex degrees is exactly twice the number of edges (by the handshaking lemma.) $\square$

Claim 16.2.8. The algorithm outputs $\delta(U) \iff$ the algorithm never contracts an edge in $\delta(U)$.

Proof. Suppose some edge $uv \in \delta(U)$ is contracted. Then uv disappears from the graph, so it is certainly not in the cut that is output. It follows that the algorithm does not output $\delta(U)$.

Conversely, suppose that the algorithm keeps performing contractions until two supervertices remain, but yet never contracts an edge in $\delta(U)$. Then every edge uv that was contracted must either have both $u, v \in U$ or both $u, v \in \overline{U}$ (otherwise $uv \in \delta(U)$!). So the algorithm must have contracted all of U into one supervertex, and all of $\overline{U}$ into the other supervertex. $\square$

Proof of Theorem 16.2.4. Recall that we have fixed some particular minimum cut $C = \delta(U)$, and that the min cut size is $k = |C|$. We want to analyze the probability that this particular cut C is output. By Claim 16.2.8, this happens if and only if no edge in C is contracted. Since the contracted edges are randomly chosen, we can analyze the probability that any of those contracted edges lie in C.

Let us consider the start of the i^{th} iteration. Each contraction operation decreases[1] the number of vertices exactly by one. So at this point there are exactly $n - i + 1$ vertices. What can we say about the probability of contracting an edge in C during the i^{th} iteration, supposing we have not yet done so?

Note that the current graph still has min cut size exactly k. (It is at least k by Claim 16.2.3, and it is at most k because we suppose that the cut C still survives.) So Claim 16.2.7 implies that the graph still has at least $(n - i + 1)k/2$ edges. The probability that the randomly chosen edge in the i^{th} iteration lies in C is

$$\frac{|C|}{\#\text{ remaining edges}} \;\leq\; \frac{k}{(n-i+1)k/2} \;=\; \frac{2}{n-i+1}.$$

Writing this more formally, we have

$$\begin{aligned}&\Pr\Big[\text{edge contracted in } i^{\text{th}} \text{ iteration } \textit{not} \text{ in } C \;\Big|\; \text{haven't contracted an edge in } C \text{ so far}\Big]\\ &\geq\; 1 - \frac{2}{n-i+1} \;=\; \frac{n-i-1}{n-i+1}.\end{aligned}$$

[1]This is why we defined the contraction operation so that it does not create self-loops.

Now we apply Fact A.3.5 to analyze all iterations.

$$
\begin{aligned}
&\Pr[\,\text{never contract an edge in } C\,] \\
&= \prod_{i=1}^{n-2} \Pr\left[\,\text{edge contracted in } i^{\text{th}} \text{ iteration } \textit{not} \text{ in } C \;\middle|\; \text{haven't contracted an edge in } C \text{ so far}\,\right] \\
&\geq \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} \\
&= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \\
&= \frac{2}{n(n-1)},
\end{aligned}
$$

by massive amounts of cancellation. So, by Claim 16.2.8, the probability that the algorithm outputs the cut C is at least $2/n(n-1)$. □

16.2.6 Extensions

The contraction algorithm is interesting not only because it gives a simple method to compute minimum cuts, but also because there are several interesting corollaries and extensions. First we present a bound on the number of minimum cuts in a graph.

Corollary 16.2.9. In any connected, undirected graph the number of minimum cuts is at most $\binom{n}{2}$.

Proof. Let $C_1, \ldots, C_\ell$ be the minimum cuts of the graph. Then we have

$$
\begin{aligned}
1 &\geq \Pr[\,\text{algorithm outputs } \textit{some} \text{ min cut}\,] \\
&= \Pr[\,\text{algorithm outputs } C_1 \;\vee\; \cdots \;\vee\; \text{algorithm outputs } C_\ell\,] \\
&= \sum_{i=1}^{\ell} \Pr[\,\text{algorithm outputs } C_i\,] \qquad \text{(union of disjoint events, Fact A.3.7)} \\
&\geq \ell \cdot \frac{2}{n(n-1)} \qquad \text{(by Theorem 16.2.4)}.
\end{aligned}
$$

Rearranging, we conclude that $\ell \leq n(n-1)/2 = \binom{n}{2}$. □

Problem to Ponder 16.2.10. Amazingly, Corollary 16.2.9 is exactly tight! For every n, there is a connected graph with n vertices and exactly $\binom{n}{2}$ minimum cuts. Can you think of this graph?

Answer.

Consider the cycle with n vertices. A min cut has two edges, and there are exactly $\binom{n}{2}$ ways to choose two edges from the cycle.

The next corollary proves a similar result for *approximate* minimum cuts.

Definition 16.2.11. For any $\alpha \geq 1$, a cut is called an ***α-small-min-cut*** if its number of edges is at most α times the size of a minimum cut.

Corollary 16.2.12. In any undirected graph, and for any integer $\alpha \geq 1$, the number of α-small-min-cuts is less than $n^{2\alpha}$.

Proof. The idea here is very simple: if we stop the contraction algorithm early (i.e., before contracting down to just two supervertices) then each α-small-min-cut has a reasonable probability of surviving.

Formally, let $r = 2\alpha$. Run the contraction algorithm until the contracted graph has only r supervertices, which means that it has at most 2^r cuts. Output one of those cuts chosen uniformly at random.

The probability that a particular α-small-min-cut survives contraction down to r vertices is

$$\begin{aligned}
\Pr[\text{survives}] &\geq \prod_{i=1}^{n-r} \Big(1 - \frac{\alpha c}{(n-i+1)c/2}\Big) \\
&= \prod_{i=1}^{n-r} \frac{n-i+1-2\alpha}{n-i+1} \\
&= \frac{(n-2\alpha)(n-2\alpha-1)\cdots(r-2\alpha+1)}{n(n-1)\cdots(r+1)} \\
&= (n-2\alpha)! \cdot \frac{r!}{n!}
\end{aligned}$$

So, the probability that a particular α-small-min-cut is output by the algorithm is at least

$$\Pr[\text{survives}] \cdot \Pr[\text{output} \mid \text{survives}] \;=\; (n-2\alpha)! \cdot \frac{r!}{n!} \cdot 2^{-r} \;>\; n^{-2\alpha},$$

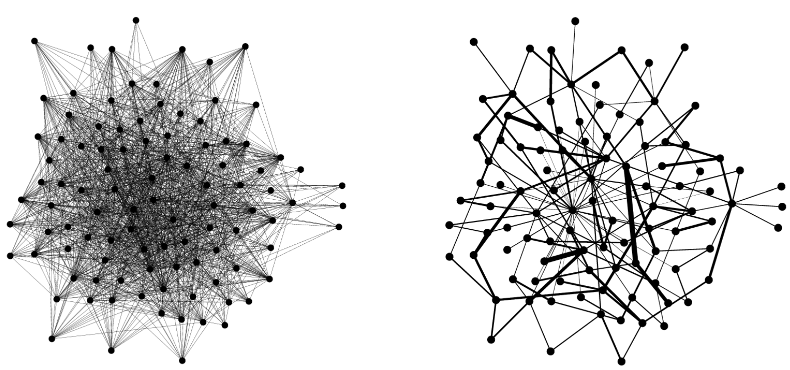
where we have used the inequalities $2^r \leq r!$ and $n!/(n-2\alpha)! < n^{2\alpha}$. $\square$

16.3 Graph Skeletons

16.3.1 Compressing Graphs

Graphs are extremely useful in computer science, but sometimes they are big and dense and hard to work with. Perhaps there is some generic approach to make life easier?

> Can we approximate any graph by a sparse graph?



This question does not have a definite answer because we have not said what we want to approximate. For example, one might aim to approximate distances between vertices. We will look at *approximating the cuts* of a graph $G = (V, E)$. As usual G is undirected, $n = |V|$ and $m = |E|$.

Recall that a *cut* is the *set of edges leaving a set* $U \subseteq V$, written

$$\delta(U) = \{ \, uv \in E \, : \, u \in U \text{ and } v \notin U \, \} \, .$$

To emphasize that this cut is in the graph G, we will also write it as $\delta_G(U)$.

The compression plan. We would like to randomly sample edges from G with probability p, yielding a "compressed" subgraph H of G. Let $\delta_H(U)$ refer to the cuts in H. Intuitively we would like to say that

$$\text{Silly goal 1:} \qquad |\delta_H(U)| \;\approx\; |\delta_G(U)| \qquad \forall U \subseteq V.$$

This is clearly silly because H should only have a p-fraction of the edges in G. So instead we should aim to say:

> The cuts of H are almost the same size as the cuts in G, just scaled down by a factor p.

The cuts of H have random size, so perhaps we should look at their expectation?

$$\text{Silly goal 2:} \qquad \mathrm{E}\,[\,|\delta_H(U)|\,] \;=\; p \cdot |\delta_G(U)| \qquad \forall U \subseteq V.$$

This is much too weak because it *always* holds exactly, no matter what value of p we pick! Even if we picked $p = 2^{-100n}$, the equality still holds, even though we're very unlikely to sample any edges at all!

We need a stronger goal. Instead of looking at the expectation, we will hope that *all* cuts in H are *simultaneously* close to the size as the cuts in G, just scaled down by a factor p. How close? Let's say that they agree up to a factor of $1 \pm \epsilon$. In mathematics, we define the events

$$\begin{aligned} U \text{ is good} \;&=\; \left\{ \, 1 - \epsilon \leq \frac{|\delta_H(U)|}{p \cdot |\delta_G(U)|} \leq 1 + \epsilon \, \right\} \\ H \text{ is good} \;&=\; \{ \, U \text{ is good } \; \forall U \subseteq V \, \} \, . \end{aligned}$$

This is an extremely strong statement because it has conditions on *every* cut in the graph, and there are 2^n of them! With considerable optimism, our goal is

$$\text{Actual goal:} \qquad \Pr\,[\, H \text{ is good} \,] \;\geq\; 0.5.$$

16.3.2 The Compression Algorithm

We will discuss the following algorithm for compressing a graph.

Algorithm 16.4 The Compression Algorithm. We assume $0 < \epsilon < 1$.

```
function Compress(graph G = (V, E), float ε)
    Let c be the size of the minimum cut in G.
    Let p = 15 ln(2n)/(ε²c).
    Create graph H by sampling every edge in G with probability p.
    Output H.
end function
```

Our theorem shows that this algorithm shrinks the number of edges by nearly a factor of c, while approximately preserving all cuts.

Theorem 16.3.1. The compression algorithm produces a graph H satisfying

$$\begin{aligned} \mathrm{E}[\,\#\text{ edges in } H\,] &= O\big(\tfrac{m \log n}{\epsilon^2 c}\big) \\ \Pr[\,H \text{ is good}\,] &\geq 1 - 2/n. \end{aligned}$$

References: Karger's PhD thesis Corollary 6.2.2.

The proof of this theorem follows our usual recipe of "Chernoff bound + union bound", but with one extra ingredient that we saw last time.

Analysis of a single cut. How likely is any single cut to be good? Fix some $U \subseteq V$. The expectation is $\mu = \mathrm{E}[\,|\delta_H(U)|\,] = p \cdot |\delta_G(U)|$. We have

$$\begin{aligned} \Pr[\,U \text{ is bad}\,] &= \Pr[\,|\delta_H(U)| \leq (1-\epsilon)\mu\,] \;+\; \Pr[\,|\delta_H(U)| \geq (1+\epsilon)\mu\,] \\ &\leq 2\exp(-\epsilon^2\mu/3) \qquad \text{(by the Chernoff bound)} \\ &= 2\exp\Big(-\epsilon^2 \cdot \frac{p \cdot |\delta_G(U)|}{3}\Big) \qquad \text{(definition of } \mu) \\ &= 2\exp\Big(-\frac{5\ln(2n)|\delta_G(U)|}{c}\Big) \qquad \text{(definition of } p) \qquad (16.3.1) \\ &\leq 2\exp\Big(-5\ln(2n)\Big) \;<\; 1/n^5. \end{aligned}$$

The last inequality holds because $|\delta_G(U)|$ is at least c, the minimum cut value. So every cut is very likely to be close to its expectation.

Question 16.3.2. What happens if we now try to union bound over *all minimum cuts*? What about over *all cuts*?

A more careful union bound. Let us say that a cut is an *nearly-2^i-min-cut* if its size is in $\{2^i c, \ldots, 2^{i+1}c\}$, for $i \geq 0$. Since these cuts are all reasonably big, this actually helps us get a better guarantee from the Chernoff bound! Plugging into (16.3.1), we have

$$\begin{aligned} \Pr[\,U \text{ is bad}\,] &\leq 2\exp\Big(-\frac{5\ln(2n)|\delta_G(U)|}{c}\Big) \\ &\leq 2\exp\Big(-\frac{5\ln(2n)2^i c}{c}\Big) \qquad \text{(it is a nearly-}2^i\text{-min-cut)} \\ &< n^{-5\cdot 2^i}. \end{aligned}$$

Now we will union bound over all the nearly-2^i-min-cuts. These are all α-small-min-cuts with $\alpha = 2^{i+1}$, so by Corollary 16.2.12 we have

$$(\text{number of nearly-}2^i\text{-min-cuts}) \;\leq\; n^{2\alpha} \;=\; n^{2\cdot 2^{i+1}} \;=\; n^{4\cdot 2^i}.$$

Thus, by a union bound,

$$\begin{aligned} \Pr\big[\,\textit{any} \text{ nearly-}2^i\text{-min-cut is bad}\,\big] &\leq (\text{number of nearly-}2^i\text{-min-cuts}) \cdot (\text{failure probability}) \\ &\leq n^{4\cdot 2^i} \cdot n^{-5\cdot 2^i} \\ &= n^{-2^i}. \end{aligned}$$

Now doing another union bound over i,

$$\begin{aligned}
\Pr[\,\textit{any}\text{ cut is bad}\,] &\leq \sum_{i\geq 0} \Pr\left[\,\textit{any}\text{ nearly-}2^i\text{-min-cut is bad}\,\right] \\
&\leq \frac{1}{n} + \frac{1}{n^2} + \frac{1}{n^4} + \frac{1}{n^8} + \cdots \\
&\leq \frac{1}{n}\sum_{i\geq 0} n^{-i} \\
&\leq \frac{2}{n},
\end{aligned}$$

by our standard bound on geometric sums. Thus, $\Pr[\,H \text{ is good}\,] \geq 1 - 2/n$.

16.3.3 Possible improvements

There are a few possible complaints that one might have about this algorithm.

- **Runtime.** The slowest step is compute the minimum cut size c. This can be done in polynomial time using the algorithm of Section 16.2.

  Solution. It turns out that c can be computed in $O(m \log^3(n))$ time, by a different algorithm of Karger!

- **Number of edges.** This algorithm scales down the number of edges by a factor $\Theta(\log(n)/\epsilon^2 c)$, so it is only useful if $c \gg \log n$.

  Solution. A refined approach by Benczur & Karger produces an H with $O\left(\frac{n \log n}{\epsilon^2}\right)$ edges, regardless of the value of c!

16.4 Exercises

Exercise 16.1 Finding all min cuts. Corollary 16.2.6 shows that the contraction algorithm can find a particular minimum cut in a graph with probability at least $1 - \delta$, for any given $\delta > 0$. Explain how to use or modify the algorithm so that, given δ, with probability at least $1 - \delta$, it can output a list containing *every* minimum cut in the graph.

Chapter 17

Distributed Computing

17.1 Randomized exponential backoff

Suppose there are n clients that want to access a shared resource. The canonical example of this is sending a packet on a wireless network; other examples include acquiring a lock on a database row, clients connecting to servers, etc. If the clients all try to transmit on a wireless network simultaneously, then all transmissions will fail. Fortunately, clients can tell whether or not their attempt succeeded, so they can try again if necessary. However if two clients repeatedly kept trying in the *same* time slots, then they would livelock and never manage to transmit.

Some mechanism is needed for the clients to retry their transmissions, ideally with different clients trying in different time slots. Unfortunately the clients cannot communicate, other than by detecting these failed transmissions. To make matters worse, the clients don't even know how many other clients there are.

Formalizing this problem requires specifying when the clients arrive. One common approach would be to assume that clients arrive according to a stochastic model (e.g., a Poisson process). We will avoid elaborate modelling, and instead will just assume that there are n clients who all arrive at the same initial time.

There is a widely used algorithm for solving this problem called Randomized Exponential Backoff (or Binary Exponential Backoff). This is commonly taught in undergrad classes on computer networking or distributed systems. The algorithm, shown in Algorithm 17.1, is very simple: try to acquire the resource at a random time in the current window. If you fail, double the size of the window.

Let us define the ***completion time***[1] to be the time at which the *last* client accesses the resource. Our goal is to analyze the completion time.

Question 17.1.1. Suppose there are n clients. Is there an obvious lower bound on the completion time, as a function of n?

Answer.

Since all n clients must access the resource in distinct time slots, the completion time must be at least n.

Next we would like to find a simple upper bound on the completion time. Consider a single iteration of the while loop with window size W. If W is sufficiently large, will the clients probably pick distinct time

[1] This is the same idea as makespan in the scheduling literature.

Algorithm 17.1 The Randomized Exponential Backoff algorithm.

```
function Backoff
    Let W ← 1                                          ▷ The current window size
    while haven't yet accessed resource do
        Generate X ∈ [W] uniformly at random
        Try to access the resource in time slot X of current window
        if conflict when accessing resource then
            Wait until the end of the current window
            W ← 2 · W                                  ▷ Double the window size
        end if
    end while
end function
```

slots? This is precisely the birthday paradox! We saw in (7.2.1) that the expected number of colliding pairs is less than $n^2/2W$. Since each colliding *pair* causes *two* clients to conflict, the expected number of *clients* who conflict is less than n^2/W. Thus, when $W = 2n^2$, the expected number of clients who conflict is less than $1/2$.

This implies that the completion time is probably $O(n^2)$, and definitely $\Omega(n)$. We now prove a much better guarantee.

Theorem 17.1.2. The completion time is $O\big(n \log(n) \log\log(n)\big)$, with probability at least $1/2$.

The protocol runs for multiple iterations. In a generic iteration, we will use the following notation.

- W is the window size.
- R_{before} is the number of remaining clients at the start of the iteration.
- R_{after} is the number of remaining clients at the end of the iteration.

The goal is to analyze R_{after}, and to show that it will drop to zero after a small number of iterations. As above, each iteration is just a "balls and bins" scenario with R_{before} balls and W bins. Using (7.2.1) again, we obtain[2]

$$\mathrm{E}\left[\, R_{\mathrm{after}} \,\right] < R_{\mathrm{before}}^2 / W. \tag{17.1.1}$$

Let's ignore the first few iterations because probably all clients will conflict. Instead, we focus on the k iterations numbered

$$\lg(4nk) + 1, \quad \lg(4nk) + 2, \quad \cdots, \quad \lg(4nk) + k.$$

We will think of these k iterations as k random trials. Let $\mathcal{E}_i$ be the event that the i^{th} trial is a *failure*, which we define to be

$$\mathcal{E}_i \;=\; \text{“}R_{\mathrm{after}} \geq 2k R_{\mathrm{before}}^2 / W\text{”}.$$

[2] The fastidious reader will note that this expectation should be conditioning on the outcomes of all previous iterations.

We can analyze the probability of this event as follows.

$$\begin{aligned}
\Pr[\mathcal{E}_i] &= \Pr\left[R_{\text{after}} \geq 2kR_{\text{before}}^2/W\right] \\
&\leq \frac{\mathrm{E}[R_{\text{after}}]}{2kR_{\text{before}}^2/W} && \text{(by Markov's inequality, Fact A.3.23)} \\
&\leq \frac{R_{\text{before}}^2/W}{2kR_{\text{before}}^2/W} && \text{(by (17.1.1))} \\
&= \frac{1}{2k}
\end{aligned}$$

We can analyze the probability of *any* failure during these iterations with a union bound.

$$\Pr[\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_k] \;\leq\; \sum_{i=1}^{k} \Pr[\mathcal{E}_i] \;\leq\; k \cdot \frac{1}{2k} \;=\; \frac{1}{2}.$$

Thus, with probability at least $1/2$, all trials are successful.

So assume that all trials are succesful. Note that the i^{th} trial has window size

$$(\text{window size during } i^{\text{th}} \text{ trial}) \;=\; 2^{\lg(4nk)+i} \;\geq\; 4nk, \tag{17.1.2}$$

since the window size doubles with each iteration. The crux of the analysis is the following lemma.

Lemma 17.1.3. Assume that all trials are succesful. Then the number of remaining clients after the i^{th} trial is at most $2n/2^{2^i}$.

Proof. The proof is by induction. The base case is when $i = 0$. This is trivial because $2n/2^{2^0} = n$, and there are initially n clients.

So assume $i \geq 1$ and let R_{before} be the number of remaining clients *before* the i^{th} trial. By induction, $R_{\text{before}} \leq 2n/2^{2^{i-1}}$. We have observed in (17.1.2) that $W \geq 4nk$. So, after the i^{th} trial, the number of remaining clients is at most

$$\frac{2kR_{\text{before}}^2}{W} \;\leq\; \frac{2k \cdot \left(2n/2^{2^{i-1}}\right)^2}{4nk} \;=\; \frac{8n}{4 \cdot 2^{2^i}} \;=\; \frac{2n}{2^{2^i}}. \qquad \square$$

Since the number of remaining clients decreases *doubly exponentially* in i, we can ensure that no clients remain by defining

$$k \;=\; \lg\lg(4n).$$

If all k trials succeed, Lemma 17.1.3 tells us that the number of remaining clients is at most

$$\frac{2n}{2^{2^k}} \;=\; \frac{2n}{2^{2^{\lg\lg(4n)}}} \;=\; \frac{2n}{2^{\lg(4n)}} \;=\; \frac{1}{2}.$$

In fact, the number of remaining clients must be zero, because it is an integer.

It remains to analyze the completion time, which no more than the total number of time slots up to the end of iteration $\lg(4nk) + k$. This is just the sum of all window sizes, which is

$$\begin{aligned}
1 + 2 + 4 + \cdots + 2^{\lg(4nk)+k} &< 2 \cdot 2^{\lg(4nk)+k} && \text{(geometric sum, (A.2.1))} \\
&= 8nk2^k \\
&= 8n \lg\lg(4n) \lg(4n) && \text{(by definition of } k\text{)} \\
&= O\big(n \log(n) \log\log(n)\big).
\end{aligned}$$

Broader context. The randomized exponential backoff algorithm seems to originate in the research of Simon S. Lam and Leonard Kleinrock in 1975.

The problem of communication on a shared channel is heavily studied, even from a theoretical perspective. See, e.g., Leslie Ann Goldberg's survey. Nevertheless, it is a very simple example of the much broader topic of managing access to a shared resource. Examples include congestion control of TCP connections, allocating machines to tasks in data centers, scheduling medical resources, etc. Queueing theory and control theory are heavily used techniques in this field. Naturally a lot more modeling assumptions are necessary when studying these more elaborate settings. This section has studied a heavily simplified problem in order to see some key ideas without getting bogged down in details.

17.1.1 Exercises

Exercise 17.1. Prof. Smackoff has an idea to improve the Randomized Exponential Backoff protocol.

She observes that we only need a window size of $W \approx n$ to guarantee significant decrease in the number of remaining clients. But after $\log \log n$ iterations the protocol increases the window to $W \approx n \log n$, which seems unnecessarily. She thinks that this is what slows down the Backoff protocol.

Her new idea is to slow down the growth of window size by setting $W \leftarrow \lceil W \cdot (1 + 1/\lg W) \rceil$ instead of $W \leftarrow 2W$. (She starts with an initial window $W = 2$ to avoid division by 0.)

Does her idea improve the completion time (i.e., the end of the window in which the last clients finishes)? We will figure this out by running experiments to compare the Backoff and Smackoff protocols. You should run both protocols 5 times for

$$n \in \{50000, 100000, 150000, 200000, \ldots, 500000\}$$

and plot the average completion time over 5 random trials.

***** Exercise 17.2.** Improve the analysis of the Backoff algorithm to show that it has completion time $O(n \log n)$ with probability at least $1/2$.

17.2 Consistent hashing

In ordinary uses of hash tables, the data is all stored in main memory on a single machine. This assumption may be false in in a modern computing environment with large scale data. Often we have multiple machines, or nodes, forming a *distributed system*, and we want to spread the data across those nodes. This can be advantageous if the amount of data, or the workload to access it, exceeds the capacity of a single node.

There are two key objectives for such a system.

- The system should be fair to the nodes, meaning that each node should store roughly the same amount of data.
- The system must efficiently support dynamic addition and removal of nodes from the system. The reason is that it is common to add servers as the system grows, and occasionally to remove servers if they require maintanence.

Consistent hashing is a clever idea that achieves these design goals. Let us explain it by contrast to a traditional in-memory hash table. Ordinarily there is a universe U of "keys", a collection B of "buckets", and a hash function $h : U \to B$. Data is stored in the buckets somehow, for example using linked lists.

The key point is that traditional in-memory hash tables have a *fixed* collection of buckets. In our distributed system, the nodes correspond to the buckets, and we want them to change dynamically. Naively, it seems that any change to set of buckets requires discarding the hash function and generating a new one.

Consistent hashing improves on that naive approach. The main idea can be explained in two sentences.

- The nodes are given random locations on the unit circle, and the data is hashed to the unit circle.
- Each item of data is stored on the node whose location is closest when moving to the left.

To make this more concrete, define C to be the interval $[0, 1]$. We will view C as a circle by having this interval wrap around at its endpoints. The leftward-distance from a to b is

$$(a - b) \bmod 1.$$

Example 17.2.1. The leftward-distance from 0.6 to 0.4 is

$$(0.6 - 0.4) \bmod 1 \;=\; 0.2.$$

However, the distance is not symmetric: the leftward distance from 0.4 to 0.6 is

$$(0.4 - 0.6) \bmod 1 \;=\; (-0.2) \bmod 1 \;=\; 0.8.$$

Configuring the nodes. Let B be our set of nodes. To participate in the system, each node $x \in B$ chooses its location to be a uniformly random point $y \in C$.

Mapping data to nodes. The key question is how to assign data to nodes. Instead of using a hash function $h : U \to B$ to map data directly to B, we will instead use a hash function

$$h : U \to C$$

that maps data to the *circle*. So far this does not specify which node will store an item of data — we need an additional rule. The rule is simple: a data item z is mapped to the node whose location y that is closest to $h(z)$ in the leftward direction. In other words, we want $(h(z) - y) \bmod 1$ to be as small as possible.

The system's functionality is implemented as follows.

- **Initial setup.** The nodes choose their locations randomly from C, then arrange themselves into a doubly-linked, circular linked list, sorted by their locations in C. Network connections are used to represent the links in the list. Lastly, the hash function $h : U \to C$ is chosen, and made known to all nodes and all users.

- **Storing/retrieving data.** Suppose a user wishes to store or retrieve some data with a key k. She first applies the function $h(k)$, obtaining a point on the circle. Then she searches through the linked list of nodes to find the node b whose location is closest to $h(k)$. The data is stored or retrieved from node b. To search through the list of nodes, we can use naive exhaustive search, which may be reasonable if the number of nodes is small.

- **Adding a node.** Suppose a new node b is added to the system. It chooses a random location in C, then is inserted into the sorted linked list of nodes at the appropriate location.

  After doing so, we now violate the property that data is stored on its closest node. There might be some existing data k in the system for which the new node's location is now the closest to $h(k)$. That data is currently stored on some other node b', so it must now *migrate* to node b. Note that b' must necessarily be a neighbor of b in the linked list. So b can simply ask its leftward neighbor to send all of its data which for which b's location is now the closest.

- **Removing a node.** To remove a node, we do the opposite of addition. Before b is removed from the system, it first sends all of its data to its leftward neighbor b'.

17.2.1 Analysis

By randomly mapping nodes and data to the unit circle, the consistent hashing scheme tries to ensure that no node stores a disproportionate fraction of the data. We now quantify this.

A node's fraction of the circle. Suppose the nodes are numbered $1, 2, \ldots, n$. Let X_i be the fraction of the circle for which node i is the closest node (in the leftward direction). Our first claim says that each node is responsible for a fair fraction of the circle, at least in expectation.

Claim 17.2.2. $\mathrm{E}[X_i] = 1/n$ for all $i \in [n]$.

Proof. Our first observation is that

$$\mathrm{E}[X_i] \;=\; \mathrm{E}[X_j] \qquad \forall i, j. \tag{17.2.1}$$

This is because their positions on the circle are purely based on the random locations that they chose, and nothing to do with[3] our numbering of the nodes as $1, \ldots, n$.

Our second observation is that the nodes are jointly responsible for the entire circle. In symbols,

$$1 \;=\; (\text{total length of circle}) \;=\; \sum_{i=1}^{n} X_i.$$

We now apply linearity of expectation. For any i,

$$1 \;=\; \mathrm{E}\left[\sum_{j=1}^{n} X_j\right] \;=\; \sum_{j=1}^{n} \mathrm{E}[X_j] \;=\; n\,\mathrm{E}[X_i],$$

now using (17.2.1). Dividing by n completes the proof. □

The preceding claim shows that the system is fair to the nodes *in expectation*. However, reiterating our moral from Section 8.1:

[3] The technical term for this is that $X_1, \ldots, X_n$ are exchangeable random variables.

The expectation of a random variable does not tell you everything.

Question 17.2.3. Instead of the nodes picking a location uniformly from C, suppose they picked a location uniformly from $[0, 0.000001]$. Is that still fair to the nodes?

Answer.

It is still true that each node is responsible for a $1/n$ fraction of the circle in expectation. However the node with the largest location is responsible for approximately a 0.999999 fraction of the circle, which is quite unfair.

The following claim makes a stronger statement about the fairness of consistent hashing. For notational simplicity, we will henceforth assume that n is a power of two.

Lemma 17.2.4. With probability at least $1-1/n$, every node is responsible for at most a $O(\log(n)/n)$ fraction of the circle. This is just a $O(\log n)$ factor larger than the expectation.

Proof. Let $\ell = \lg n$. Define overlapping arcs $A_1, \ldots, A_n$ on the circle as follows:

$$A_i \;=\; [\; i/n,\; (i+2\ell)/n \bmod 1 \;].$$

Note that these are "circular intervals", in that the right endpoint can be smaller than the left endpoint. The length of each arc A_i is exactly $2\ell/n$.

We will show that every such arc probably contains a node. That implies that the fraction of the circle for which any node is responsible is at most twice the length of an arc, which is $4\ell/n$.

Let $\mathcal{E}_i$ be the event that none of the nodes' locations lie in A_i. Then

$$\begin{aligned}
\Pr[\mathcal{E}_i] &= \prod_{j=1}^{n} \Pr\left[j^{\text{th}} \text{ node not in } A_i \right] && \text{(by independence)}\\
&= (1-2\ell/n)^n && \text{(length of } A_i \text{ is } 2\ell/n\text{, and (A.3.4))}\\
&\le \exp(-2\ell/n)^n && \text{(by Fact A.2.5)}\\
&= \exp(-2\lg(n)) \;<\; 1/n^2.
\end{aligned}$$

We want to show that it is unlikely that *any* arc A_i contains no points. This is accomplished by a union bound.

$$\Pr[\text{any arc } A_i \text{ has no nodes}] \;=\; \Pr[\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n] \;\le\; \sum_{i=1}^{n} \Pr[\mathcal{E}_i] \;=\; n \cdot (1/n^2) \;=\; 1/n. \qquad \square$$

A node's fraction of the data. The preceding discussion tells us that each node is responsible for a fair fraction of the circle. However, one would additionally like to say that, each node is responsible for a fair fraction of the *data*. Fortunately, this also turns out to be largely true.

Problem to Ponder 17.2.5. Suppose that the distributed hash table contains n nodes and m items of data. Let X_i be the number of data items stored on node i. What is $\mathrm{E}[X_i]$?

Broader context. Consistent hashing is used in many real-world products, including Akamai's content distribution network, Amazon Dynamo, Apache Cassandra, Discord, etc.

17.3 Exercises

Exercise 17.3. we claimed that consistent hashing is fair because

- In expectation, each node is responsible for a $1/n$ fraction of the circle.
- With probability close to 1, every node is responsible for an $O(\log(n)/n)$ fraction of the circle.

In this question, we consider the possibility that some nodes are responsible for a tiny fraction of the circle.

For each value of $n \in \{100, 1000, 10000, 100000\}$ perform the following experiment $t = 1000$ times.

- Have n nodes choose random locations in the unit circle. Compute the *minimum* distance between any two nodes on the circle. (Note: due to wrap-around, the distance between 0.95 and 0.05 is 0.1.)

After performing these t trials, determine the average value of this *minimum* distance. For each value of n, report your value for the average *minimum* distance. These numbers will be quite small, so it is probably more useful to present them in a table, rather than a plot.

What would you guess is the formula for the expected *minimum* distance between n nodes, as a function of n? Do you think consistent hashing is fair?

Exercise 17.4. Let X_i be the fraction of the circle that node i is responsible for. Let $k = 100n^2$. Prove that $\Pr[\text{any node has } X_i \leq 1/2k] \leq 1/100$.

17.4 Efficient Routing

Now let's consider how to link the machines together to enable efficient routing when n is large. Our basic design had just used a linked list. There is a general principle here that is worth stating explicitly.

> Data structures that organize objects in memory can be adapted to organize machines in a distributed system.

Armed with this principle, we can imagine adapting other familiar data structures to organize the nodes (i.e., machines) in this distributed hash table.

The trusty balanced binary tree seems a reasonable candidate, so let's explore its pros and cons.

- **Pro: low diameter.** Any node can route to any other node with $O(\log n)$ hops.
- **Con: poor fault-tolerance.** A failure of a single node can drastically disconnect the network. For example, the failure of the root prevents the nodes in the left subtree (about half the nodes) from searching for nodes in the right subtree (also about half the nodes).
- **Con: congestion.** If any node in the left-subtree searches for any node in the right-subtree, then this network traffic must go through the root node. So the root must cope with significantly more network traffic than, say, a leaf of the tree.

What about using Skip Lists? This idea was proposed by a student during the last lecture. This feels like it might be a good way to augment our original linked list with long-distance links. Unfortunately it suffers from the same cons as a binary tree. The highest-level list will contain just one or two nodes. The bulk of the searches will go through those nodes. Furthermore, if one of those nodes fails, the network is split roughly in half.

17.5 SkipNet

There is a twist on the Skip List idea that solves these problems. The new approach improves the fault-tolerance by giving every node $O(\log n)$ pointers. It also avoids congestion by allowing *every* node to have long-distance pointers. This idea is called a Skip Net or a Skip Graph.

To explain, let's review the construction of Skip Lists.

- Every node joins the lowest list, L_1.
- From L_1, roughly half of the nodes join a higher list L_2, and the remainder join no further lists.
- From L_2, roughly half of the nodes join a higher list L_3, and the remainder join no further lists.
- etc.

The new idea is quite simple to illustrate.

- Every node joins the lowest list.
- From L_1, roughly half of the nodes join a higher list L_2; the remainder join a different list L_2'.
- From L_2, roughly half of the nodes join a higher list L_3; the remainder join a different list L_3'.
- From L_2', roughly half of the nodes join a higher list L_3''; the remainder join a different list L_3'''.
- etc.

The apostrophes are clearly getting out of control, so let's devise some new notation. At the lowest level there is one list. At the next level there are two lists. At the next level there are four lists. These are growing like powers of two, so it seems like a useful idea to label each list with a bitstring.

- **Level** 0**:** there is a single list L_ϵ containing all the nodes. (Here ϵ denotes the "empty string". It doesn't mean that the list is empty.)
- **Level** 1**:** there are two lists L_0 and L_1.
  L_0 and L_1 are obtained by splitting L_ϵ randomly in two.
- **Level** 2**:** there are four lists $L_{00}, L_{01}, L_{10}, L_{11}$.
  L_{00} and L_{01} are obtained by splitting L_0 randomly in two.
  L_{10} and L_{11} are obtained by splitting L_1 randomly in two.
- etc.

In general, for any binary string b of length k, the list L_b is randomly split into two lists L_{b0} and L_{b1} at level $k+1$. This process continues until each node is alone in some list. Of course, each list is kept sorted by its node's keys.

How to implement this splitting? With Skip Lists we grouped nodes into lists by having them generate their level as a geometric random variable. Now we must adopt a different scheme. Each node will generate a random bitstring of sufficient[4] length. The rule for joining lists is:

> A node joins the list L_p for every *prefix* p of its random bitstring.

For example, if a node's random bitstring is

$$010010110\ldots$$

then it will join the lists

$$L_\epsilon,\quad L_0,\quad L_{01},\quad L_{010},\quad L_{0100},\quad L_{01001},\quad L_{010010},\quad L_{0100101},\quad L_{01001011},\quad L_{010010110},\ldots$$

As desired, all nodes join L_ϵ, half the nodes join L_0, the other half join L_1, etc.

17.5.1 Searching by key

Question 17.5.1. Suppose we start at a node s and are searching for the key $destKey$. How should we perform the search?

There are a couple of answers to the question. We will discuss an answer that probably isn't the first that comes to mind. The reason will become clear later.

Observation 17.5.2. For any node s, the set of lists to which it belongs forms a Skip List containing all the nodes[5]!

Algorithm 17.2 The search algorithm. We are starting at node s and searching for $destKey$.

1: **function** SEARCHBYKEY(node s, key $destKey$)
2: Restrict attention to the Skip List consisting of the lists to which s belongs. (Imagine temporarily deleting all other lists.)
3: Perform a search for $destKey$ in that Skip List.
4: **end function**

17.5.2 Searching by random bitstring

Now we will consider a strange question: how could you find a node whose random bitstring equals (or is closest to) a given bitstring b. This is not so obvious, because the lowest list L_ϵ is sorted by key and *not* sorted by the random bitstrings. Nevertheless, it is possible, as we shall see.

Algorithm 17.3 Searching by bitstring. We are starting at a node d and searching for the bitstring b.

1: **function** SEARCHBYBITSTRING(node d, bitstring b)
2: The current node is d
3: The current list is L_ϵ
4: **repeat**
5: Step one-by-one through the nodes in the current list, starting from the current node, until finding a node whose bitstring matches one more bit of b.
6: **until** the current list contains a single node
7: **end function**

[4] In principle the bitstring could have infinite length as the bits can be generated on demand.

[5] Well, it won't have a header node at the very left. The node s itself effectively acts as a header node, since it belongs to all the lists in this Skip List.

Claim 17.5.3. SearchByBitstring takes $O(\log n)$ time.

Proof sketch. We observe that

the path traversed by SearchByBitstring () from d for $s.bitstring$

is the same as

the path traversed by SearchByKey from s for $d.key$,

just in backwards order! We already know that the latter path has length $O(\log n)$ in expectation. □

17.5.3 Join (or insertion) of a new node

Inserting a new node is quite easy, now that we have the strange SearchByBitstring () operation.

Algorithm 17.4 Inserting a new node with key v.

```
function Insert(node key v)
    Create a new node with key v and a new random bitstring b
    Perform SearchByBitstring(b), arriving at a node s. So s.bitstring has the longest
    common prefix with b, among all existing nodes.
    Restrict attention to the Skip List consisting of the lists to which s belongs. (Imagine
    temporarily deleting all other lists.)
    Perform Insert(v) in this Skip List.
end function
```

Chapter 18

Learning

18.1 Distinguishing distributions

Suppose there are two machines that generate random numbers in $[M]$. Let us call them Machine P and Machine Q. We are given a single random number that was either generated by P or by Q, but we do not know which. Based on the random value that we receive, and complete knowledge of the distributions of P and Q, our task is to guess which of the machines generated it.

Remarks on terminology. In the statistics literature, this problem is well-studied under the name hypothesis testing. We will use terminology that differs from that literature because our motivations are different. The conventional terminology emerged a century ago in the study of testing whether a certain scientific hypothesis is true or false — for example, did a medical intervention have any effect?

Analogous to how defendents are considered "innocent until proven guilty", statisticians often assume that an intervention does not work unless there is overwhelming evidence that it does. This asymmetry is reflected in their terminology "null hypothesis", meaning that the intervention did not work, and "alternative hypothesis", meaning that it did work.

In our problem of interest, distinguing between P and Q, there may be no special importance of either P or Q, so we are content to treat them symmetrically. For this reason we will avoid the conventional statistics terminology, and instead use terminology tailored to our problem.

18.1.1 The formal setup

We will think of P and Q as probability distributions, described by the values

$$\begin{aligned} p_i &= \Pr[\, P \text{ generates the value } i\,] \\ \text{and} \quad q_i &= \Pr[\, Q \text{ generates the value } i\,] \qquad \forall i \in [M]. \end{aligned}$$

An algorithm $\mathcal{A}$ for distinguishing P and Q behaves as follows. First, the distributions $P = [p_1, \ldots, p_M]$ and $Q = [q_1, \ldots, q_M]$ are provided to $\mathcal{A}$. (One could think of P and Q as inputs, but it is perhaps better to think of them as hard-coded parameters.) Next, $\mathcal{A}$ receives a random value X, generated either by P or by Q. Using X it must then decide whether to output either "P" or "Q" as its guess. Thus, $\mathcal{A}$ can be viewed as a deterministic function $f_{P,Q}$ of X.

For any event E, we use the notation $\Pr_P[E]$ to denote the probability of event E when using distribution P. Similarly, let $\Pr_Q[E]$ be the probability of E when using distribution Q. To describe the algorithm $\mathcal{A}$'s failure probability, we use the following notation.

- *False positive probability*: $\mathrm{FP}(\mathcal{A}) = \Pr_Q[\,\mathcal{A} \text{ outputs "}P\text{"}\,]$.
  When the true distribution is Q, this is the probability that $\mathcal{A}$ erroneously guesses P.
- *False negative probability*: $\mathrm{FN}(\mathcal{A}) = \Pr_P[\,\mathcal{A} \text{ outputs "}Q\text{"}\,]$.
  When the true distribution is P, this is the probability that $\mathcal{A}$ erroneously guesses Q.

18.1.2 Distinguishing identical distributions

Suppose that machines P and Q generate numbers with exactly the same distribution, i.e., $P = Q$. Intuition suggests that, given a random number generated either by P or by Q, it is impossible to tell which distribution generated it. This is indeed true and, using our notation above, we will be able to state it precisely.

Simply observe that the algorithm's probability of outputting "P" is the same, regardless of whether P or Q generated the random input, since $P = Q$. Thus,

$$\mathrm{FP}(\mathcal{A}) = \Pr_Q[\,\mathcal{A} \text{ outputs "}P\text{"}\,] = \Pr_P[\,\mathcal{A} \text{ outputs "}P\text{"}\,] = 1 - \Pr_P[\,\mathcal{A} \text{ outputs "}Q\text{"}\,] = 1 - \mathrm{FN}(\mathcal{A}).$$

This shows that there is an inherent tradeoff between false positives and false negatives. For example, if an algorithm has probability 0.1 of false positives, then it must have probability 0.9 of false negatives. Perhaps the best one could hope to do is to minimize the *larger* of the two, but still one would have

$$\max\{\mathrm{FP}(\mathcal{A}), \mathrm{FN}(\mathcal{A})\} \;\geq\; 0.5,$$

since the maximum of two values is no smaller than the average. Thus, there is no useful algorithm for distinguishing these distributions, since the same guarantee is achieved by a randomized algorithm that ignores its input and simply returns an unbiased coin flip. We summarize this discussion as follows.

Theorem 18.1.1. If $P = Q$ then every algorithm $\mathcal{A}$ for distinguishing P and Q must satisfy

$$\mathrm{FP}(\mathcal{A}) + \mathrm{FN}(\mathcal{A}) \;=\; 1 \qquad \text{and therefore} \qquad \max\{\mathrm{FP}(\mathcal{A}), \mathrm{FN}(\mathcal{A})\} \;\geq\; 0.5. \tag{18.1.1}$$

18.1.3 Distinguishing different distributions

Since distinguishing identical distributions is impossible, perhaps distinguishing very similiar distributions is quite hard? The answer depends on

Let us now consider the scenario where P and Q are somewhat different. Perhaps it is possible to make a smart guess? We will discuss how to quantify the difference between distributions.

One might imagine that, if these probabilities are quite close, it will not be possible to guess well. Depending on how one defines "quite close", that intuition might not be correct.

Claim 18.1.2. The are distributions for Machines P and Q satisfying two properties.

- $|p_i - q_i| \leq 0.03$ for every $i \in [100]$.
- Given a sample from either P or Q, it is possible to guess correctly with probability 0.75 which distribution generated the sample.

Proof. Suppose that Machine P returns a uniform random number in $[100]$, whereas Machine Q returns a uniform random number in $[25]$. Then $p_i = 0.01$ for all i, $q_i = 0.04$ for $i \leq 25$ and $q_i = 0$ for $i > 25$. Note that $|p_i - q_i| \leq 0.03$ for all i.

If we receive a value $X > 25$ we will guess that P generated it; otherwise we will guess that Q generated it. The only possibility of a mistake is when P generates a number in $[25]$, which happens with probability exactly 0.25. $\square$

The previous example shows that, even though each value $|p_i - q_i|$ is quite small, it is still possible to guess well. It turns out that the *total* difference in the probabilities is a more relevant quantity. More precisely, we will sum up the total probability of outcomes for which distribution P has a *larger* probability than distribution Q.

Definition 18.1.3. The ***total variation distance*** between distributions P and Q on $[M]$ is

$$\mathrm{TV}(P, Q) \;=\; \sum_{\substack{i \in [M] \\ p_i > q_i}} (p_i - q_i).$$

References: (Cover and Thomas, 1991, pp. 299–300), Wikipedia.

Question 18.1.4. For the two distributions in the proof of Claim 18.1.2, what is $\mathrm{TV}(P, Q)$?

Answer.

We have $p_i > q_i$ for $i > 25$, so $\mathrm{TV}(P, Q) = \sum_{i=26}^{100} (p_i - q_i) = 75 \cdot 0.01 = 0.75$.

There are many equivalent formulas for the total variation distance, some of which are explored in Exercise 18.1. In the next theorem we discuss one important such formula.

Theorem 18.1.5.

$$\mathrm{TV}(P, Q) \;=\; \max_{E} \left| \Pr_P[E] - \Pr_Q[E] \right|,$$

where the maximization is over all events E.

References: (Motwani and Raghavan, 1995, Exercise 6.24), (Cover and Thomas, 1991, eq. (12.137)), Wikipedia.

Proof. Let X be the randomly chosen value in $[M]$. By the law of total probability (Fact A.3.6),

$$\Pr_P[E] - \Pr_Q[E] \;=\; \sum_{i \in [M]} \big(\Pr_P[E \wedge X{=}i] - \Pr_Q[E \wedge X{=}i] \big).$$

Note that $X = i$ is a single outcome in the probability space. If event E does occur when $X = i$, then

$$\Pr_P[E \wedge X{=}i] = \Pr_P[X = i] = p_i \qquad \text{and} \qquad \Pr_Q[E \wedge X{=}i] = \Pr_Q[X = i] = q_i.$$

Otherwise, if E does not occur when $X = i$, then both of these probabilities are 0. Thus,

$$\Pr_P[E] - \Pr_Q[E] \;=\; \sum_{i \text{ where } E \text{ occurs}} (p_i - q_i). \tag{18.1.2}$$

This sum is maximized by choosing E to sum only the positive terms, or minimized by choosing E to sum only the negative terms. Thus,

$$\sum_{i \text{ where } q_i > p_i} (p_i - q_i) \;\leq\; \Pr_P[E] - \Pr_Q[E] \;\leq\; \sum_{i \text{ where } p_i > q_i} (p_i - q_i).$$

The right-hand side is, by definition, equal to $\mathrm{TV}(P,Q)$. By Exercise 18.1 the left-hand side equals $-\mathrm{TV}(P,Q)$. Together they imply that $|\Pr_P[E] - \Pr_Q[E]| \leq \mathrm{TV}(P,Q)$.

The maximum value of (18.1.2) can be achieved by choosing E to be the event that $p_X > q_X$. For this event we have

$$\Pr_P[E] - \Pr_Q[E] \;=\; \sum_{i \text{ where } p_i > q_i} p_i - \sum_{i \text{ where } p_i > q_i} q_i \;=\; \mathrm{TV}(P,Q). \tag{18.1.3}$$

Since this is non-negative we may take the absolute value, which completes the proof. ☐

Next we show that the total variation distance determines the success probability of the optimal algorithm for distinguishing between distributions P and Q.

Theorem 18.1.6 (Total variation determines optimal failure probability). Every algorithm $\mathcal{A}$ satisfies

$$\mathrm{FP}(\mathcal{A}) + \mathrm{FN}(\mathcal{A}) \;\geq\; 1 - \mathrm{TV}(P,Q).$$

Moreover, let $\mathcal{A}^*$ be the algorithm that outputs "P" if $p_X > q_X$, and otherwise outputs "Q". Then

$$\mathrm{FP}(\mathcal{A}^*) + \mathrm{FN}(\mathcal{A}^*) \;=\; 1 - \mathrm{TV}(P,Q).$$

Proof. Let $\mathcal{A}$ be arbitrary. Let E be the event that $\mathcal{A}$ outputs "P". By Theorem 18.1.5 we have

$$\Pr_P[E] - \Pr_Q[E] \;\leq\; \mathrm{TV}(P,Q). \tag{18.1.4}$$

Negating this and adding $\Pr_P[E] + \Pr_P\left[\overline{E}\right] = 1$ we obtain

$$\underbrace{\Pr_P\left[\overline{E}\right]}_{=\mathrm{FN}(\mathcal{A})} + \underbrace{\Pr_Q[E]}_{=\mathrm{FP}(\mathcal{A})} \;\geq\; 1 - \mathrm{TV}(P,Q). \tag{18.1.5}$$

For the algorithm $\mathcal{A}^*$, the event E occurs when the input X satisfies $p_X > q_X$. For this event, (18.1.3) shows that (18.1.4) holds with equality, and therefore (18.1.5) also holds with equality. ☐

Example 18.1.7. Let us consider again the example of Claim 18.1.2. The algorithm $\mathcal{A}$ described in that claim outputs "P" if $X > 25$. It has

$$\begin{aligned}
\mathrm{FP}(\mathcal{A}) &= \Pr_Q[\,\mathcal{A} \text{ outputs "}P\text{"}\,] \;=\; 0 \\
\mathrm{FN}(\mathcal{A}) &= \Pr_P[\,\mathcal{A} \text{ outputs "}Q\text{"}\,] \;=\; 0.25.
\end{aligned}$$

Those quantities sum to 0.25, which equals $1 - \mathrm{TV}(P,Q)$ since we found that $\mathrm{TV}(P,Q) = 0.75$ in Question 18.1.4 above. Thus, according to Theorem 18.1.6, this algorithm is optimal.

Remarks. Theorem 18.1.6 is not easy to find in the statistics literature, but it can be viewed as a variant of the Neyman-Pearson Lemma. It can be found in (Lehmann and Romano, 2022, Theorem 15.1.1), equation (4.5) of Wu's lecture notes, or Theorem 6.3 of Polyanskiy and Wu's lecture notes. Whereas Theorem 18.1.6 characterizes the algorithm $\mathcal{A}$ minimizing $\mathrm{FP}(\mathcal{A}) + \mathrm{FN}(\mathcal{A})$, the Neyman-Pearson lemma characterizes the algorithm that minimizes $\mathrm{FN}(\mathcal{A})$ amongst all algorithms with a fixed value of $\mathrm{FP}(\mathcal{A})$. The latter statement seems to be more relevant to statisticians due to the asymmetric treatment of the null and alternative hypotheses. The optimal algorithm $\mathcal{A}^*$ is an example of a likelihood ratio test.

18.2 Exercises

Exercise 18.1. Let P and Q be distributions on $[M]$.

Part I. Prove that

$$\mathrm{TV}(P,Q) = \max_E \left(\Pr_P[E] - \Pr_Q[E]\right),$$

where the maximization is over all events E. (Note the lack of absolute value.)

Part II. Prove that

$$\mathrm{TV}(P,Q) = \sum_{i\in[M],\, q_i>p_i} (q_i - p_i).$$

Part III. Prove that

$$\mathrm{TV}(P,Q) = \frac{1}{2}\sum_{i\in[M]} |p_i - q_i|.$$

Part IV. Prove that

$$\mathrm{TV}(P,Q) = 1 - \sum_{i\in[M]} \min\{p_i, q_i\}.$$

Exercise 18.2. Let P and Q be distributions. You must design an algorithm that is given k independent samples. Assume that either all sample are drawn independently from P, or all are drawn independently from Q. Give an algorithm that uses only $k = O(1/\mathrm{TV}(P,Q)^2)$ samples, and can distinguish between P and Q with $\mathrm{FP}(A) \leq 0.1$ and $\mathrm{FN}(A) \leq 0.1$.

Chapter 19

Secrecy and Privacy

19.1 Information-theoretic encryption

The Zoodle founder, Larry Sage, wants to send a secret message to the CEO, Sundar Peach. He is worried that the message might be intercepted by their competitor, Elon Muskmelon. They decide to use an encryption scheme that Elon cannot decrypt, even if he has the world's best computers at his disposal. An encryption scheme with this guarantee is said to be information-theoretically secure.

The one-time pad, which we now present, is the canonical example of such a scheme. As in Chapter 14, we will assume that Larry's message is a string of length s where each character is in $[\![q]\!]$. We allow q to be any integer at least 2, not necessarily a prime.

$$\begin{aligned} \text{Larry's message:} \quad & M_1, \ldots, M_s \\ \text{where} \quad & \text{each } M_i \in [\![q]\!] \end{aligned}$$

Larry and Sundar have agreed ahead of time on a secret key. This is simply a string of characters in $[\![q]\!]$ which are generated independently and uniformly at random.

$$\text{Random secret key:} \quad R_1, \ldots, R_s$$

Both Larry and Sundar know the key, but it is kept secret from any others. The key is only to be used once, hence the "one-time" descriptor.

The encryption process is simple: component-wise addition modulo q.

$$\begin{aligned} \text{Encrypted message:} \quad & E_1, \ldots, E_s \\ \text{where} \quad & E_i = (M_i + R_i) \bmod q \end{aligned}$$

The encrypted message is what Larry sends to Sundar.

The decryption process is equally simple. Since Sundar also knows the random key, he can recover Larry's message using component-wise subtraction.

$$\begin{aligned} \text{Decrypted message:} \quad & D_1, \ldots, D_s \\ \text{where} \quad & D_i = (E_i - R_i) \bmod q. \end{aligned}$$

Claim 19.1.1. The decrypted message D equals Larry's message M.

Proof. The idea is simple: subtraction is the inverse of addition. The only wrinkle is dealing with the mod operations carefully. By the principle of "taking mod everywhere" (Fact A.2.11), we have

$$D_i \;=\; (E_i - R_i) \bmod q \;=\; \Big(\big((M_i + R_i) \bmod q\big) - R_i\Big) \bmod q \;=\; (M_i + R_i - R_i) \bmod q \;=\; M_i$$

for every i. So the messages are equal in every component. □

Suppose that Elon intercepts the encrypted message E. This poses no threat because, since Elon lacks the random key R, he can learn nothing about Larry's message M.

Theorem 19.1.2. If R is unknown, then E reveals nothing about M.

Proof. The main idea is to show that E is a uniformly random string. Since this is true *regardless* of Larry's message M, it follows that E cannot be used to infer anything about M.

First let's focus on the individual characters of E. Applying Fact A.3.25 with $X = R_i$ and $y = M_i$, we obtain that

$$E_i \;=\; (R_i + M_i) \bmod q \;=\; (X + y) \bmod q$$

is uniformly distributed in $[\![q]\!]$. That is, $\Pr[\, E_i = e_i \,] = 1/q$ for every fixed character e_i.

Next let's consider the whole string E. Since the components of R are independent, it follows that the components of E are independent. So, for any fixed string $e_1, \ldots, e_s$,

$$\Pr[\, E_1 E_2 \ldots E_s \;=\; e_1 e_2 \ldots e_s \,] \;=\; \prod_{i=1}^{s} \Pr[\, E_i = e_i \,] \;=\; 1/q^s,$$

which means that E is uniformly random, regardless of what M was.

We now argue that Elon cannot tell which message was sent. Let P be the distribution of E when the message M is sent. Fix any other message, and let Q be the distribution on E if that message were sent. We have argued that P and Q are both identical to the uniform distribution, so by Theorem 18.1.1, there is no algorithm that can use E to determine which message was sent. □

Implementations of this scheme commonly use the case $q = 2$, in which the messages and key are bit strings.

Question 19.1.3. How are addition and subtraction implemented in the case $q = 2$?

Answer.

Both addition and subtraction are simply the Boolean Xor operation. See page 181.

Notes

The one-time pad is originally due to Vernam in his 1926 paper. He says:

> *If... we employ a key composed of letters selected absolutely at random, a cipher system is produced which is absolutely unbreakable.*

The notion of information-theoretic security was called "perfect secrecy" by Shannon; see Section 10 of his 1949 paper. A more detailed presentation can be found in Example 2.1 and Theorem 2.2 in the book of Boneh and Shoup.

19.2 Randomized response

During the COVID-19 pandemic, UBC sent a survey to faculty, staff and students asking for disclosure of their COVID-19 vaccination status. What might the university do with this information? One possibility would be to tell instructors the fraction of students in their class who are vaccinated. That seems useful, perhaps to help decide whether to open the windows.

Privacy is the snag with that plan. Early in the term, students might add or drop the class. If the instructor knew the vaccinated fraction both before and after a student dropped the class, then this would reveal the student's vaccination status.

Is there some way that instructors can estimate the fraction of vaccinated students without violating their privacy? For concreteness, let's say that f is the actual fraction of vaccinated students.

Question 19.2.1. Can you think of some ways to estimate f while preserving privacy? Does your approach require trusting somebody? Will it work if people add or drop the class?

There is a slick randomized protocol to solve this problem. It doesn't involve any private communication channels or trusted parties. Each student just needs a ***biased random coin***, that comes up heads with probability b. The value of b is the same for every student, and publicly known.

The idea is very simple: each student flips their biased coin. If it comes up tails, they publicly announce their true vaccination status — this happens with probability $1-b$. If it comes up heads, they *lie* and publicly announce the opposite — this happens with probability b.

To see how this works, let's imagine picking a random student in a class and seeing what their response might be. The probability that they announce being vaccinated can be analyzed as follows.

$$\begin{aligned}
\Pr[\,\text{says "vaccinated"}\,] &= \Pr[\,\text{is vaccinated and tells truth}\,] + \Pr[\,\text{is unvaccinated and lies}\,] \\
&= f\cdot(1-b) + (1-f)\cdot b \\
&= f\cdot(1-2b) + b \qquad (19.2.1)
\end{aligned}$$

The first line uses that the event of saying "vaccinated" can be decomposed into two disjoint events (see Fact A.3.7). The second line uses that the events of being vaccinated and telling the truth are independent (see Definition A.3.3).

Similarly, imagine gathering the responses from all students and letting X be the fraction that said "vaccinated". With a bit of thought, one can see that

$$\mathrm{E}[\,X\,] \;=\; f\cdot(1-2b)+b, \qquad (19.2.2)$$

which is exactly the same as in (19.2.1).

The instructor learns the value of X by having the students execute the protocol. Using X, the instructor would like to estimate the true value of f. Inspired by (19.2.2), it is natural to use the estimate

$$\hat{f} \;=\; \frac{X-b}{1-2b} \qquad (19.2.3)$$

This quantity $\hat{f}$ is called an *unbiased estimator* because its expected value equals the true value. That is,

$$\mathrm{E}\left[\,\hat{f}\,\right] \;=\; \mathrm{E}\left[\frac{X-b}{1-2b}\right] \;=\; \frac{\mathrm{E}[\,X\,]-b}{1-2b} \;=\; f, \qquad (19.2.4)$$

by plugging in (19.2.2).

Scenario: Rare lies. To make our formulas a bit more concrete, suppose that the coin is extremely biased, say $b = 0.01$, which means the students have a low probability of lying. Plugging into the above formulas,

$$\begin{aligned} \mathrm{E}\left[\,X\,\right] &= 0.98f + 0.01 \\ \hat{f} &= \frac{X - 0.01}{0.98} \\ \text{Unbiased estimate:} \qquad \mathrm{E}\left[\,\hat{f}\,\right] &= f. \end{aligned}$$

The last equality comes directly from (19.2.4).

Should students feel comfortable about the privacy of this scheme? On the plus side, the students have *plausible deniability*. If a student announces being unvaccinated, it *might* be actually be a lie because the coin flip came up heads — we cannot be certain. That said, the probability of coming up heads is only $b = 0.01$, so lies are rare and a student's announcement seems to reveal a lot about their vaccination status.

Scenario: Frequent lies. Observe the formula (19.2.4) holds regardless of b. We could just set $b = 0.49$, so that the students lie nearly half the time. Now we would have $\hat{f} = (X - 0.49)/0.02$, which is once again an unbiased estimate (directly from (19.2.4)).

This larger value of b allows us to feel more confident in the privacy. If a student announces being unvaccinated, it *might* be a lie, but only if the coin flip was heads, which has probability 0.49. Since the coin is *nearly* a fair coin, it feels like each announcement reveals almost nothing about the student's vaccination status. Remarkably, using these announcements that reveal almost nothing, the instructor can still produce an unbiased estimate for f!

Problem to Ponder 19.2.2. What happens if we set $b = 0.99$? Or $b = 0.5$?

Problem to Ponder 19.2.3. Each student is likely taking multiple classes. What happens if multiple instructors use this protocol. Can they collude to learn more information about the students?

19.2.1 How good is the estimator?

Above we have shown that $\hat{f}$ is an unbiased estimator for f (except if $b = 0.5$). Does this mean that $\hat{f}$ is likely to be close to f? Can we use the fact that $\hat{f}$ is unbiased?

Question 19.2.4. Does $\mathrm{E}\left[\,\hat{f}\,\right] = f$ imply that $\hat{f}$ is likely to be close to f?

Answer.

No: suppose that there is only one student. If the student announces being vaccinated, then $X = 1$ and our estimate is $\hat{f} = (1-\alpha)/(1-2\alpha) = 25.5$. Otherwise $X = 0$ then our estimate is $\hat{f} = (0-\alpha)/(1-2\alpha) = -24.5$. Now $\hat{f}$ looks like an awful estimate because the true value of f is a fraction, so it is between 0 and 1!

A key topic for this class is understanding whether an estimate is likely to be close to its expected value. We will discuss many tools for that purpose. At this point, your intuition might tell you that if b is close to $1/2$, the estimate $\hat{f}$ is "noisier", so we need more students in order for $\hat{f}$ to be close to f. This is indeed true, as we will see in Section 19.3 after discussing the necessary tools.

19.2.2 Broader context

Basic classes in statistics will talk about unbiased estimators. Why does our present discussion belong in an algorithms class rather than a statistics class? In my mind, statistics is about modeling and measuring properties of populations that already exist. In contrast, this class is about designing algorithms that *deliberately inject randomness* in order to make the algorithm better. The randomization might make the algorithm faster, or simpler, or, as shown in this section, more private.

Differential privacy is the area that studies how to report properties of a data set without violating privacy of the individual constituents. The randomized response protocol described above is a basic example from that area. These techniques are used by technology companies, such as in Apple's iOS to collect usage data from devices. Within our department, professors Mijung Park and Mathias Lécuyer do research in this area.

19.2.3 Exercises

Exercise 19.1. Suppose we modify the randomized response protocol as follows: each student randomly decides to tell the truth with probability 1/2, lie with probability 1/4, or stay silent with probability 1/4. Using the students' responses (or lack thereof), how can the instructor give an unbiased estimator for the fraction of vaccinated students?

19.3 Concentration for randomized response

Let us revisit the randomized response protocol from Chapter **??**. Recall that f is the true fraction of students who are vaccinated, and X was the fraction who *said* that they were vaccinated. Our estimator is

$$\hat{f} \;=\; \frac{X-b}{1-2b}. \tag{19.3.1}$$

It is unbiased, meaning $\mathrm{E}\left[\,\hat{f}\,\right] = f$.

We are interested to know how likely it is that $\hat{f}$ is close to f. Let us set things up to use the Hoeffding bound.

First, let $\mathcal{V}$ be the set of vaccinated students and $\mathcal{U}$ be the set of unvaccinated students. Recall that $|\mathcal{V}| = fn$ and $|\mathcal{U}| = (1-f)n$. Define the random variable Z_i to be the indicator of the event that student i lies. So $\mathrm{E}[\,Z_i\,] = b$. Then define

$$\begin{aligned}
\#\text{ unvaccinated liars:} \qquad U &= \sum_{i\in\mathcal{U}} Z_i \\
\#\text{ vaccinated liars:} \qquad V &= \sum_{i\in\mathcal{V}} Z_i.
\end{aligned}$$

Then $\mathrm{E}[\,U\,] = b(1-f)n$ and $\mathrm{E}[\,V\,] = bfn$.

The number of students who claim to be vaccinated is the number of students in $\mathcal{V}$ who tell the truth plus the number of students in $\mathcal{U}$ who lie. Dividing by n, the fraction who claim to be vaccinated is

$$\begin{aligned} X &= \frac{1}{n}\left(\sum_{i\in\mathcal{V}}(1-Z_i)+\sum_{i\in\mathcal{U}}Z_i\right) \\ &= \frac{1}{n}(fn-V+U) \;=\; f-V/n+U/n. \end{aligned} \tag{19.3.2}$$

Thus, as we already knew,

$$\mathrm{E}[\,X\,] \;=\; f-bf+b(1-f) \;=\; (1-2b)f+b. \tag{19.3.3}$$

Using a familiar Hoeffding bound, we can prove the following concentration result for X.

Lemma 19.3.1.

$$\Pr\left[\,|X-\mathrm{E}[\,X\,]|<4/\sqrt{n}\,\right] \;\geq\; 0.99.$$

Now we need to analyze the error in the estimator $\hat{f}$. We have

$$\begin{aligned} \Pr\left[|\hat{f}-f|<\frac{4}{(1-2b)\sqrt{n}}\right] &= \Pr\left[\left|\frac{X-b}{1-2b}-f\right|<\frac{4}{(1-2b)\sqrt{n}}\right] && \text{(by (19.3.1))} \\ &= \Pr\left[\big|X-\big(f(1-2b)+b\big)\big|<\frac{4}{\sqrt{n}}\right] && \text{(multiplying by } 1-2b) \\ &= \Pr\left[|X-\mathrm{E}[\,X\,]|<\frac{4}{\sqrt{n}}\right] && \text{(by (19.3.3))} \\ &\geq 0.99 && \text{(by Lemma 19.3.1).} \end{aligned}$$

We have shown that the error in our estimator is less than

$$\frac{4}{(1-2b)\sqrt{n}},$$

with probability at least 0.99. Note that, as b approaches $1/2$, the denominator decreases (which means more error), but as n increases, the denominator increases (which means less error).

Question 19.3.2. Suppose we want the error to be ϵ. How should we choose n?

Answer.

Define

$$n=\left(\frac{4}{\epsilon(1-2b)}\right)^2.$$

Then the error will be at most ϵ with probability at least 0.99.

Exercises

Exercise 19.2. Prove Lemma 19.3.1.

19.4 Privately computing the average

Zoodle has a subsidiary called Korn Academy that teaches online classes. The students in their Kernel Hacking class have just taken a midterm and received their grades. They would like to know the average grade, but the professor refuses to tell them. Instead, the students decide to communicate among themselves to determine the average grade. The catch is, each student wants to keep his or her own grade secret from the other students! Is this a contradiction, or is it possible?

Question 19.4.1. Is this possible if there are two students?

Answer.

No: if a student knows the average grade and their own grade, then it is easy to compute the other student's grade.

Suppose that there are three students. Each student knows only their own grade, which is an integer between 0 and 100.

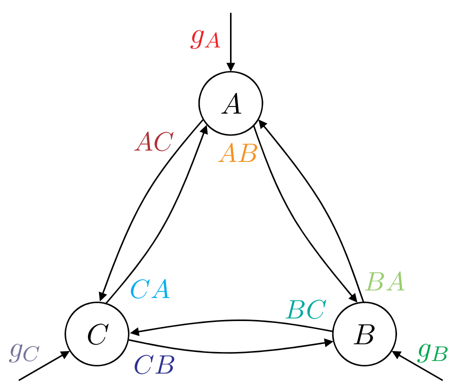
$$\begin{array}{rccc} \text{Students:} & A & B & C \\ \text{Grades:} & g_A & g_B & g_C & \in \{0, \ldots, 100\}\,. \end{array}$$

The mathematics becomes more convenient if, instead of computing the average, they compute the sum mod 400. The value 400 is not special, it is just any integer bigger than the actual sum.

The blinds. Each student creates two blinds, which are *uniformly random* numbers in $[\![400]\!]$, and remembers them.

$$\begin{array}{rccc} \text{Student } A \text{ generates blinds:} & AB & \text{and} & AC \\ \text{Student } B \text{ generates blinds:} & BA & \text{and} & BC \\ \text{Student } C \text{ generates blinds:} & CA & \text{and} & CB \end{array}$$

Exchanging blinds. Taking care to remember the blinds that they generated, each student (privately) sends a blind to each other student.



$$\begin{array}{rccc} \text{Student } A \text{ receives:} & BA & \text{and} & CA \\ \text{Student } B \text{ receives:} & AB & \text{and} & CB \\ \text{Student } C \text{ receives:} & AC & \text{and} & BC \end{array}$$

Public announcement. Now each student announces a public value which is their own grade, minus their blinds, plus what they received, all modulo 400.

$$\begin{array}{rcccccccc} & \text{public value} & = (& \text{grade} & - & \text{sent blinds} & + & \text{received blinds} &) \bmod 400 \\ \text{Student } A \text{ announces:} & p_A & = (& g_A & - & AB - AC & + & BA + CA &) \bmod 400 \\ \text{Student } B \text{ announces:} & p_B & = (& g_B & - & BA - BC & + & AB + CB &) \bmod 400 \\ \text{Student } C \text{ announces:} & p_C & = (& g_C & - & CA - CB & + & AC + BC &) \bmod 400 \end{array}$$

Computing the sum. Every student now adds up all the public values, modulo 400. The clever observation is that all the blinds cancel!

$$\text{Sum of public values:} \quad (p_A + p_B + p_C) \bmod 400 = (g_A + g_B + g_C) \bmod 400$$

Since we know that $0 \leq g_A + g_B + g_C < 400$, that last "mod 400" does nothing. In other words,

$$(\text{sum of grades}) \;=\; g_A + g_B + g_C \;=\; (p_A + p_B + p_C) \bmod 400.$$

We conclude that each student learns the sum of the grades, and therefore each can compute the average grade.

Is secrecy guaranteed? The main question is: does this protocol guarantee secrecy?

Question 19.4.2. In the procotol above, what does student C learn about g_A and g_B?

Answer.
Everybody learns $g_A + g_B + g_C$, so student C can compute $g_A + g_B$. (There might be some ancillary consequences; for example, if $g_A + g_B = 200$ then $g_A = g_B = 100$.)

Suppose student C is overly curious about the grades of A and B. Perhaps student C deviates from the protocol and uses some nefarious approach to compute CA, CB and p_C. Can C learn more information about g_A and g_B? Interestingly, the answer is no.

Theorem 19.4.3. Even if a student is dishonest, they cannot learn any more information about the other students' grades.

To prove this theorem, we will only need a simple fact about how uniformly random numbers behave under modular arithmetic.

Corollary A.3.26. Let $q \geq 2$ be an arbitrary integer. Let X be uniformly random in $[\![q]\!]$. Let Y be any random integer that is independent from X. Then

$$(X + Y) \bmod q$$

is also uniformly random in $[\![q]\!]$.

Proof of Theorem 19.4.3. Suppose that C is the dishonest student, whereas A and B are honest.

Student C receives the blinds AC, BC, and sees the public values p_A and p_B. Let us consider the value of p_A.

$$p_A \;=\; (\underbrace{BA}_{X} + \underbrace{g_A - AB - AC + CA}_{Y}) \bmod 400$$

This is uniformly random in $[\![400]\!]$ by Corollary A.3.26, because BA is just a uniformly random value in $[\![400]\!]$ (and independent of all other values inside Y).

Similarly,

$$p_B \;=\; (\underbrace{AB}_{X} + \underbrace{g_B - BA - BC + CB}_{Y}) \bmod 400$$

is uniformly random in $[\![400]\!]$ by Corollary A.3.26 since AB is uniformly random, and independent of all other values inside Y.

However, student C does know *something* about p_A and p_B. Note that

$$(p_A + p_B) \bmod 400 \;=\; (g_A + g_B \underbrace{- AC + CA - BC + CB}_{\text{known to } C}) \bmod 400. \tag{19.4.1}$$

Student C knows CA and CB, having generated them, and knows AC and BC, having received them.

To conclude, what does student C learn? The public values p_A and p_B, each of which is a uniform random number in $[\![400]\!]$. These numbers are *not* independent, but satisfy the equation (19.4.1). Note that this joint distribution of p_A and p_B remains unchanged if we modify the values of g_A and g_B, so long as $g_A + g_B$ remains unchanged. It follows that student C has learned nothing about g_A and g_B except their sum. Student C could have learned this by being honest. $\square$

Notes

A fundamental idea of this protocol is to split the secret g_A into three values $AB, AC, g_A - AB - AC$. All three of these values can be used to determine g_A, but no two of them would be sufficient. This is a simple special case of what is called secret sharing.

The idea of collaboratively computing a function on data without revealing your own secret data is called secure multiparty computation. These sorts of techniques are being incorporated into software today, for example in federated learning. Inpher is an example of a startup using these techniques.

This topic is also vaguely related to differential privacy, in that the goal is to compute aggregate information amount people without revealing their private information.

Part V

Back matter

Acknowledgements

Thanks to my teaching assistants (Arsh, Chris, Emmanuel, Victor) and various students over the years (Hayden, Jesse, Michael, ...) for many ideas, suggestions, and corrections. Thanks also to Rebecca for some figures!

Appendix A

Mathematical Background

A.1 Notation

As much as possible I will try to use notation that is helpful but not overwhelming. Some notation that we will use regularly is listed here.

$$
\begin{aligned}
[\![p]\!] &= \{0, 1, \ldots, p-1\} \qquad \text{(this is not standard[1] notation)} \\
[p] &= \{1, 2, \ldots, p\} \\
\mathbb{R} &= \{\text{all real numbers}\} \\
[a, b] &= \{\text{ real numbers } x \,:\, a \le x \le b \,\} \\
(a, b] &= \{\text{ real numbers } x \,:\, a < x \le b \,\} \\
[a, b) &= \{\text{ real numbers } x \,:\, a \le x < b \,\} \\
\ln x &= \log_e(x) \\
\lg x &= \log_2(x) \\
\log x &= \text{logarithm to a base that is unspecified or irrelevant} \\
\vee &= \text{Boolean Or} \\
\wedge &= \text{Boolean And} \\
\langle b_k b_{k-1} \ldots b_1 b_0 \rangle &= \text{the integer } \textstyle\sum_{i=0}^{k} b_i 2^i\text{, written in its binary representation} \\
\langle 0.b_1 b_2 \ldots \rangle &= \text{the real number } \textstyle\sum_{i \ge 1} b_i 2^i\text{, written in its binary representation} \\
|A| &= \text{the cardinality of set } A \\
\overline{\mathcal{A}} &= \text{the complement of set } \mathcal{A}\text{, or the negation of event } \mathcal{A}
\end{aligned}
$$

[1]The standard notation in mathematics would be $\mathbb{Z}_p$ or $\mathbb{Z}/p\mathbb{Z}$. I will avoid that notation because it emphasizes the $\mathbb{Z}$ rather than the p.

A.2 Math

Fact A.2.1 (Harmonic sums). The sum $\sum_{i=1}^{n} \frac{1}{i}$ is called a harmonic sum. These frequently appear backward, like

$$\sum_{i=1}^{n} \frac{1}{n-i+1} \;=\; \sum_{i=1}^{n} \frac{1}{i}.$$

Some common bounds are

$$\ln(n) \;\leq\; \sum_{i=1}^{n} \frac{1}{i} \;\leq\; \ln(n) + 1.$$

References: (Lehman et al., 2018, Equation 14.21), (Cormen et al., 2001, equation (A.13) and (A.14)), (Kleinberg and Tardos, 2006, Theorem 13.10).

Fact A.2.2 (Geometric sums). For any c satisfying $0 < c < 1$,

$$\sum_{i=0}^{\infty} c^i \;=\; \frac{1}{1-c}.$$

If the sum is finite and $c \neq 1$, then

$$\sum_{i=0}^{n} c^i \;=\; \frac{1-c^{n+1}}{1-c} \;=\; \frac{c^{n+1}-1}{c-1}.$$

Some useful consequences are

$$\sum_{i=0}^{n} 2^i \;=\; 2^{n+1} - 1 \tag{A.2.1}$$

$$\sum_{i=0}^{\infty} 2^{-i} \;=\; 2 \tag{A.2.2}$$

$$\sum_{i=1}^{\infty} 2^{-i} \;=\; 1 \tag{A.2.3}$$

References: (Lehman et al., 2018, Theorem 14.1.1), (Cormen et al., 2001, equations (A.5) and (A.6)).

Standard logarithm rules.

$$\log(ab) \;=\; \log(a) + \log(b)$$

$$\log(x^k) \;=\; k \log(x)$$

$$\log_b(x) \;=\; \frac{\ln(x)}{\ln(b)} \tag{A.2.4}$$

Manipulating exponents.

$$(a^b)^c \;=\; a^{b \cdot c} \;=\; (a^c)^b.$$

Exponents of logs.

$$a^{\log_a(x)} \;=\; x.$$

In particular, $e^{\ln x} = x$ and $2^{\lg x} = x$. From that one can derive another useful formula:

$$a^{\log_{1/a}(x)} \;=\; \frac{1}{x}. \tag{A.2.5}$$

Definition A.2.3 (Rounding up to a power of two). In many scenarios it is convenient to work with powers of two rather than arbitrary integers. Assume $\alpha > 0$. Then α ***rounded up to a power of two*** means the value 2^ℓ satisfying

$$\tfrac{1}{2} \cdot 2^\ell \;<\; \alpha \;\leq\; 2^\ell, \tag{A.2.6}$$

and ℓ an integer. This condition is satisfied by taking $\ell = \lceil \lg \alpha \rceil$, so we can also define α rounded up to a power of two to be $2^{\lceil \lg \alpha \rceil}$.

Fact A.2.4 (Triangle inequality). For any real numbers a, b, we have

$$|a + b| \;\leq\; |a| + |b|.$$

More generally, for real numbers $a_1, \ldots, a_n$, we have

$$|a_1 + \ldots + a_n| \;\leq\; |a_1| + \cdots + |a_n|.$$

Fact A.2.5 (Approximating e^x near zero). For all real numbers x,

$$1 + x \leq e^x.$$

Moreover, for x close to zero, we have $1 + x \approx e^x$.

References: (Motwani and Raghavan, 1995, Proposition B.3.1), (Cormen et al., 2001, Equation (3.12)).

Fact A.2.6 (Approximating $1/e$). For all $n \geq 2$,

$$\frac{1}{5} \;<\; \frac{1}{e} - \frac{1}{4n} \;\leq\; \left(1 - \frac{1}{n}\right)^n \;\leq\; \frac{1}{e} \;<\; 0.37.$$

References: (Motwani and Raghavan, 1995, Proposition B.3.2), (Kleinberg and Tardos, 2006, Theorem 13.1).

Fact A.2.7 (Approximating sums by integrals). Let f be a non-negative function defined on $[0, a]$ that is either monotone non-decreasing or non-increasing. Define

$$S \;=\; \sum_{i=0}^{a} f(i) \qquad \text{and} \qquad I \;=\; \int_0^a f(x)\, dx.$$

Then

$$I + \min\{f(0), f(a)\} \;\leq\; S \;\leq\; I + \max\{f(0), f(a)\}.$$

References: (Lehman et al., 2018, Theorem 14.3.2).

A.2.1 Counting

Counting subsets. Let V be a set of size n. Then

$$|\{\, U \,:\, U \subseteq V \,\}| \;=\; 2^n.$$

That is, the number of subsets of V is 2^n.

References: (Lehman et al., 2018, Sections 4.1.3 and 15.2.2).

Fact A.2.8 (Number of unordered pairs). The number of pairs of integers (i,j) with $1 \leq i < j \leq n$ is

$$\binom{n}{2} \;=\; \frac{n(n-1)}{2}.$$

Moreover,

$$\binom{n}{2} \;\leq\; \frac{n^2}{2} \qquad \text{and} \qquad \binom{n}{2} \;\to\; \frac{n^2}{2} \text{ as } n \to \infty$$

in the sense that the limit of their ratio tends to 1.

References: (Lehman et al., 2018, Section 15.5).

Fact A.2.9 (Binary representation of a number). Every number in $x \in [\![2^k]\!]$ has a unique binary representation, which we write as $\langle x_{k-1} \dots x_0 \rangle$, where each $x_i \in \{0,1\}$. Their mathematical relationship is

$$x \;=\; \sum_{i=0}^{k-1} x_i 2^i.$$

As a consequence, we can determine the minimum number of bits to represent a member of a set.

Fact A.2.10 (Bit complexity). Let S be a finite set of size $n \geq 1$. Using $\lceil \lg |S| \rceil$ bits, one can identify a single element of S.

Proof. Let $n = |S|$. We can write the elements of S as $s_0, s_1, \dots, s_{n-1}$. This gives a one-to-one correspondence between S and $[\![n]\!]$. It follows from Fact A.2.9 that any integer in the set $[\![n]\!]$ can be represented using $\lceil \lg n \rceil$ bits. Thus, using the correspondence, $\lceil \lg n \rceil$ bits suffice to represent an element in S. □

A.2.2 Number theory

Fact A.2.11 (Take mod everywhere). For any integers a, b and positive integer n,

$$\begin{aligned}
(a+b) \bmod n &= \big((a \bmod n) + (b \bmod n)\big) \bmod n\\
(a-b) \bmod n &= \big((a \bmod n) - (b \bmod n)\big) \bmod n\\
(a \cdot b) \bmod n &= \big((a \bmod n) \cdot (b \bmod n)\big) \bmod n
\end{aligned}$$

References: (Lehman et al., 2018, Lemma 9.6.4), (Cormen et al., 2001, page 940).

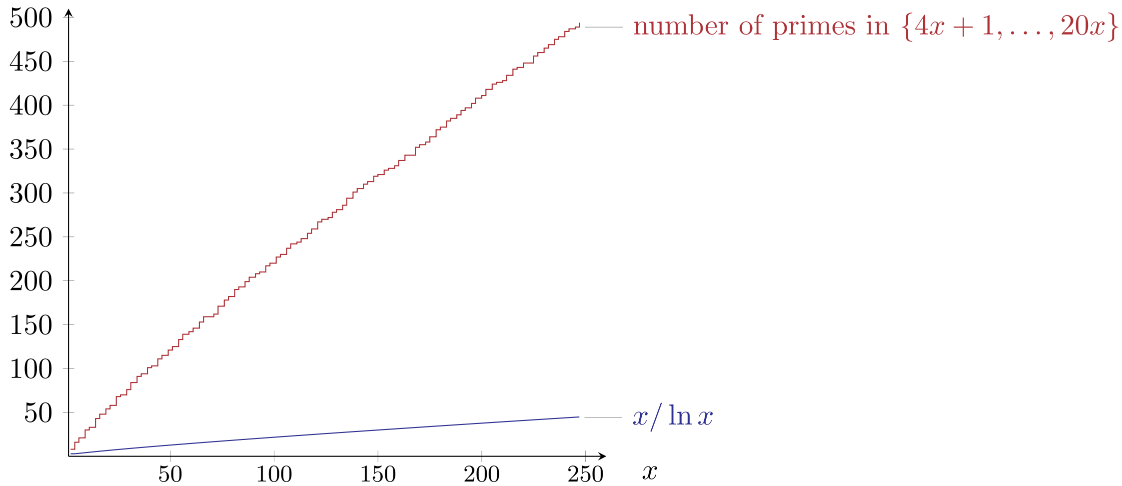
Fact A.2.12 (Existence of primes). For any integer $n \geq 1$, there exists a prime $p \in \{n, \dots, 2n\}$.

References: This fact is known as Bertrand's Postulate.

Not only can we assert the existence of a prime, but we can also estimate the number of primes of a certain size. A precise statement is as follows.

Fact A.2.13 (Number of primes). For any integer $n \geq 2$, the number of primes in $\{4n+1, \dots, 20n\}$ is at least $n / \ln n$.

References: This is a non-asymptotic form of the Prime Number Theorem. It can be derived from this paper.



Fact A.2.14 (Roots of polynomials). Let $g(x) = \sum_{i=1}^{d} c_i x^i$ be a polynomial of degree at most d in a single variable x. Let p be a prime number. We assume that at least one coefficient satisfies $c_i \bmod p \neq 0$. Then there are at most d solutions to

$$g(x) \bmod p = 0 \quad \text{and} \quad x \in [\![p]\!].$$

Such solutions are usually called *roots*.

Fact A.2.15 (Inverses mod p). For any prime p and any $x \in \{1,\ldots,p-1\}$, there is a unique solution y to

$$xy \bmod p = 1 \quad \text{and} \quad y \in \{1,\ldots,p-1\}.$$

This value y is usually denoted x^{-1}.

References: (Lehman et al., 2018, Section 9.9.1), (Cormen et al., 2001, Corollary 31.26), (Motwani and Raghavan, 1995, page 395).

Fact A.2.16 (Systems of equations mod p). Let p be a prime number. Let $a, b \in [\![p]\!]$ satisfy $a \neq b$. Let $c, d \in [\![p]\!]$. Let X and Y be variables. Then there is exactly one solution to

$$\begin{aligned}(aX+Y) \bmod p &= c\\ (bX+Y) \bmod p &= d\\ X, Y &\in [\![p]\!].\end{aligned}$$

Intuition. Roughly, Fact A.2.16 is just looking at the linear system $\left(\begin{smallmatrix} a & 1 \\ b & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} X \\ Y \end{smallmatrix}\right) = \left(\begin{smallmatrix} c \\ d \end{smallmatrix}\right)$, but doing all arithmetic modulo p. The matrix $\left(\begin{smallmatrix} a & 1 \\ b & 1 \end{smallmatrix}\right)$ has determinant $a-b$, which is non-zero since $a \neq b$. Thus, the linear system has a unique solution.

Proof sketch. We can solve the system of equations by Gaussian elimination. Subtracting the two equations, the variables Y cancel and we get

$$(a-b)X \bmod p \;=\; (c-d) \bmod p.$$

By Fact A.2.15, this is solved if and only if $X = (c-d)(a-b)^{-1} \bmod p$. Plugging this X back in to the first equation, we get that Y must have the value $Y = (c-aX) \bmod p = \big(c-a(c-d)(a-b)^{-1}\big) \bmod p$. $\square$

A.3 Probability

In this section we review some key definitions and results from introductory probability theory. A more thorough development of this theory can be found in the references (Lehman et al., 2018) (Anderson et al., 2017) (Feller, 1968) (Grimmett and Stirzaker, 2001).

A.3.1 Events

An event is a random object that either happens or does not, with certain probabilities.

Fact A.3.1 (Smaller events). Suppose that whenever event $\mathcal{A}$ happens, event $\mathcal{B}$ must also happen. (This is often written $\mathcal{A} \Rightarrow \mathcal{B}$ or $\mathcal{A} \subseteq \mathcal{B}$.) Then $\Pr[\mathcal{A}] \leq \Pr[\mathcal{B}]$.

References: (Lehman et al., 2018, Section 17.5.2), (Anderson et al., 2017, (1.15)), (Cormen et al., 2001, Appendix C.2), (Grimmett and Stirzaker, 2001, Lemma 1.3.4(b)).

Fact A.3.2 (Complementary events). Let $\mathcal{E}$ be an event. Then

$$\Pr[\mathcal{E} \text{ does not happen}] \;=\; 1 - \Pr[\mathcal{E} \text{ happens}].$$

We will usually use $\overline{\mathcal{E}}$ to denote the event that $\mathcal{E}$ does not happen.

References: (Lehman et al., 2018, Section 17.5.2), (Anderson et al., 2017, (1.13)), (Cormen et al., 2001, Appendix C.2), (Grimmett and Stirzaker, 2001, Lemma 1.3.4(a)).

Definition A.3.3 (Independence). Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be events. They are called ***mutually independent*** if

$$\Pr\left[\bigwedge_{i \in I} \mathcal{E}_i\right] \;=\; \prod_{i \in I} \Pr[\mathcal{E}_i] \qquad \forall I \subseteq [n].$$

They are called ***pairwise independent*** if the following weaker condition holds.

$$\Pr[\mathcal{E}_i \wedge \mathcal{E}_j] \;=\; \Pr[\mathcal{E}_i] \Pr[\mathcal{E}_j] \qquad \forall \text{distinct } i, j \in [n].$$

Usually when we say ***independent*** we mean mutually independent.

References: (Lehman et al., 2018, Section 18.8), (Anderson et al., 2017, Definition 2.22), Wikipedia.

Definition A.3.4. Let A and B be arbitrary events. The ***conditional probability*** $\Pr[B \mid A]$ is defined to be

$$\Pr[B \mid A] \;=\; \frac{\Pr[A \wedge B]}{\Pr[A]}$$

and undefined when $\Pr[A] = 0$.

References: (Lehman et al., 2018, Definition 18.2.1), (Anderson et al., 2017, Definition 2.1), (Kleinberg and Tardos, 2006, page 771), (Motwani and Raghavan, 1995, Definition C.4), (Mitzenmacher and Upfal, 2005, Definition 1.4), (Cormen et al., 2001, equation (C.14)).

Fact A.3.5 (Chain rule). Let $A_1, \ldots, A_t$ be arbitrary events. Then

$$\Pr[A_1 \wedge \cdots \wedge A_t] \;=\; \prod_{i=1}^{t} \Pr[A_i \mid A_1 \wedge \cdots \wedge A_{i-1}],$$

if we assume that $\Pr[A_1 \wedge \cdots A_{t-1}] > 0$.

References: (Lehman et al., 2018, Problem 18.1), (Anderson et al., 2017, Fact 2.6), (Motwani and Raghavan, 1995, equation (1.6)), (Mitzenmacher and Upfal, 2005, page 7), (Cormen et al., 2001, Exercise C.2-5), (Grimmett and Stirzaker, 2001, Exercise 1.4.2), Wikipedia.

Fact A.3.6 (Law of total probability). Let $\mathcal{B}_1, \ldots, \mathcal{B}_n$ be events such that exactly one of these events must occur. Let $\mathcal{A}$ be any event. Then

$$\Pr[\mathcal{A}] \;=\; \sum_{i=1}^{n} \Pr[\mathcal{A} \wedge \mathcal{B}_i].$$

Furthermore, if $\Pr[\mathcal{B}_i] > 0$ for each i, then

$$\Pr[\mathcal{A}] \;=\; \sum_{i=1}^{n} \Pr[\mathcal{A} \mid \mathcal{B}_i] \Pr[\mathcal{B}_i].$$

References: (Lehman et al., 2018, Section 18.5), (Harchol-Balter, 2023, Theorem 2.18), (Anderson et al., 2017, Fact 2.10), (Grimmett and Stirzaker, 2001, Lemma 1.4.4 and Exercise 1.8.10), (Motwani and Raghavan, 1995, Proposition C.3), (Mitzenmacher and Upfal, 2005, Theorem 1.6).

Disjoint events. Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be events such that *no two of them can simultaneously occur*. In other words, $\Pr[\mathcal{E}_i \wedge \mathcal{E}_j] = 0$ whenever $i \neq j$. Such events are called *disjoint*.

Fact A.3.7 (Union of disjoint events). Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be disjoint events. Then

$$\Pr[\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n] \;=\; \sum_{i=1}^{n} \Pr[\mathcal{E}_i].$$

References: (Lehman et al., 2018, Rule 17.5.3), (Anderson et al., 2017, Fact 1.2), (Mitzenmacher and Upfal, 2005, Definition 1.2 condition 3), (Cormen et al., 2001, page 1190, axiom 3), (Kleinberg and Tardos, 2006, (13.49)).

Fact A.3.8 (The union bound). Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be *any* collection of events. They could be dependent and do not need to be disjoint. Then

$$\Pr[\text{any of the events occurs}] \;=\; \Pr[\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n] \;\leq\; \sum_{i=1}^{n} \Pr[\mathcal{E}_i].$$

An easier form that is often useful is:

$$\Pr[\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n] \;\leq\; n \cdot \max_{1 \leq i \leq n} \Pr[\mathcal{E}_i].$$

The union bound may also be equivalently stated as

$$\Pr\left[\overline{\mathcal{E}_1} \wedge \cdots \wedge \overline{\mathcal{E}_n}\right] \;\geq\; 1 - \sum_{i=1}^{n} \Pr[\mathcal{E}_i]. \qquad \text{(A.3.1)}$$

References: (Lehman et al., 2018, Rule 17.5.4), (Anderson et al., 2017, Exercise 1.43), (Motwani and Raghavan, 1995, page 44), (Mitzenmacher and Upfal, 2005, Lemma 1.2), (Kleinberg and Tardos, 2006, (13.2) and (13.50)), (Cormen et al., 2001, Exercise C.2-2), (Feller, 1968, (IV.5.7)), (Grimmett and Stirzaker, 2001, Exercise 1.8.11), Wikipedia.

A.3.2 Random variables

Whereas events are binary valued — they either occur or do not — a ***random variable*** can take a range of real values with certain probabilities. The term "random variable" will be used so frequently that we will abbreviate it to ***RV***.

In this book we will use almost exclusively ***discrete RVs***, which can only take values in some set $\{v_1, v_2, v_3, \cdots\}$, which is called the ***support***. We will occasionally use some other RVs (often called ***continuous RVs***) which can take a continuous range of values, although we will not define them properly. The reader should consult the many excellent references for proper definitions (Anderson et al., 2017) (Grimmett and Stirzaker, 2001).

To analyze a random variable it is often useful to consider its ***expectation***, which is also known as its ***mean***. We only define the expectation for discrete RVs since they predominate in this book.

Definition A.3.9. Let X be a random variable. There are several equivalent definitions of the ***expected value***. The most common definition is

$$\mathrm{E}[\,X\,] \;=\; \sum_r r \cdot \Pr[\,X = r\,].$$

The sum[2] is over all values r in the range of X.

References: (Lehman et al., 2018, Theorem 19.4.3), (Anderson et al., 2017, Definition 3.21), (Cormen et al., 2001, equation (C.20)).

There is an important relationship between the expectation of a random variable and the probability that it takes large values.

Fact A.3.10 (Expected value for non-negative integer RVs). Let X be a random variable whose value is always a non-negative integer. Then

$$\mathrm{E}[\,X\,] \;=\; \sum_{t \geq 1} \Pr[\,X \geq t\,].$$

References: (Cormen et al., 2001, Equation (C.25)), (Harchol-Balter, 2023, Exercise 4.16), (Motwani and Raghavan, 1995, Proposition C.7, part 3), (Mitzenmacher and Upfal, 2005, Lemma 2.9).

The preceding fact generalizes to arbitrary random variables; see Book 2, Fact B.4.1.

Fact A.3.11 (Linearity of expectation). Let $X_1, \ldots, X_n$ be random variables and $w_1, \ldots, w_n$ arbitrary real numbers. Then

$$\mathrm{E}\left[\sum_{i=1}^{n} w_i X_i\right] \;=\; \sum_{i=1}^{n} w_i \,\mathrm{E}[\,X_i\,].$$

References: (Lehman et al., 2018, Corollary 19.5.3), (Anderson et al., 2017, Fact 8.1), (Feller, 1968, Theorem IX.2.2), (Mitzenmacher and Upfal, 2005, Proposition C.5), (Kleinberg and Tardos, 2006, Theorem 13.8).

Linearity of expectation even holds for infinitely many random variables, under some mild conditions. One such statement is given in the following fact, which we call[3] "infinite linearity of expectation".

[2] A minor detail is that the sum could be undefined if X can take infinitely many positive and negative values. This issue will not arise in this book.

[3] This name is somewhat inaccurate because it cannot handle sums of the form $\sum_{i \geq 0} w_i X_i$ where some w_i are negative.

Fact A.3.12 (Infinite linearity of expectation). Let $X_0, X_1, X_2, \ldots$ be non-negative random variables. Then

$$\mathrm{E}\left[\sum_{i \geq 0} X_i\right] = \sum_{i \geq 0} \mathrm{E}\left[\, X_i \,\right].$$

References: (Grimmett and Stirzaker, 2001, equation (5.6.13)).

Definition A.3.13 (Indicator random variable). Let $\mathcal{E}$ be an event. The *indicator* of the event $\mathcal{E}$ is the random variable $\mathbf{1}_{\mathcal{E}}$, which takes the value 1 if $\mathcal{E}$ occurs, and 0 otherwise. By the definition of expectation, we have

$$\mathrm{E}\left[\, \mathbf{1}_{\mathcal{E}} \,\right] = \Pr\left[\, \mathcal{E} \,\right].$$

References: (Lehman et al., 2018, Lemma 19.4.2), (Anderson et al., 2017, equation (3.20)).

Fact A.3.14 (Expected sum of indicators). Suppose that X is a random variable that can be decomposed as $X = \sum_{i=1}^{n} \mathbf{1}_{\mathcal{E}_i}$, where X_i is the indicator of an event $\mathcal{E}_i$. Then

$$\mathrm{E}\left[\, X \,\right] = \sum_{i=1}^{n} \mathrm{E}\left[\, \mathbf{1}_{\mathcal{E}_i} \,\right] = \sum_{i=1}^{n} \Pr\left[\, \mathcal{E}_i \,\right].$$

References: (Lehman et al., 2018, Theorem 19.5.4).

Fact A.3.15 (Law of total expectation). Let $\mathcal{B}_1, \ldots, \mathcal{B}_n$ be events such that exactly one of these events must occur, and each $\Pr\left[\, \mathcal{B}_i \,\right] > 0$. Let X be any random variable. Then

$$\mathrm{E}\left[\, X \,\right] = \sum_{i=1}^{n} \mathrm{E}\left[\, X \mid \mathcal{B}_i \,\right] \Pr\left[\, \mathcal{B}_i \,\right].$$

References: (Lehman et al., 2018, Theorem 19.4.6), (Harchol-Balter, 2023, Theorem 4.22), (Anderson et al., 2017, Facts 10.4 and 10.10), (Mitzenmacher and Upfal, 2005, Lemma 2.6).

A.3.3 Common Distributions

In probability theory it is common to study particular types of random variables. The same is true in randomized algorithms: we often use familiar types of random variables in our algorithms. The most common ones are described below.

Bernoulli distribution

This is the name for a random variable X that is either 0 or 1, where $\Pr\left[\, X = 1 \,\right] = p$ (so $\Pr\left[\, X = 0 \,\right] = 1 - p$). Naturally, can also think of it as a random Boolean variable, or as the outcome of flipping a random coin. If $p \neq 1/2$ we say that the coin is biased. We will sometimes refer to this distribution as Bernoulli(p).

References: (Anderson et al., 2017, Definition 2.31).

Uniform distribution (finite)

Let S be a finite set. A random variable X is *uniformly distributed* on S if

$$\Pr\left[\, X = s \,\right] = \frac{1}{|S|} \qquad \forall s \in S. \tag{A.3.2}$$

For any set $R \subseteq S$, a union of disjoint events (Fact A.3.7) implies that

$$\Pr[X \in R] \;=\; \frac{|R|}{|S|}. \tag{A.3.3}$$

References: (Lehman et al., 2018, Definition 17.5.5).

Uniform distribution (continuous)

Let $u > 0$. A random variable X is *uniformly distributed* on the interval $[0,u] = \{\, x : 0 \leq x \leq u \,\}$ if

$$\Pr[X \in [a,b]] \;=\; \frac{b-a}{u}, \tag{A.3.4}$$

whenever $0 \leq a \leq b \leq u$.

References: (Anderson et al., 2017, Example 1.17 and (3.9)), (Grimmett and Stirzaker, 2001, Definition 4.4.1).

The following fact gives two scenarios in which continuous uniform random variables have *zero* probability of taking a particular value.

Fact A.3.16. Let $u > 0$ be arbitrary.

- Let X be uniformly distributed on $[0,u]$. For any real number x, $\Pr[X = x] = 0$.
- Let X and Y be independent and uniformly distributed on $[0,u]$. Then $\Pr[X = Y] = 0$.

References: (Anderson et al., 2017, Fact 3.2 and (6.17)).

Using this fact, we also can determine the probability that a uniformly distributed random variable lies in open or half-open intervals.

Fact A.3.17. Let $u > 0$ be arbitrary and let X be uniformly distributed on $[0,u]$. Then

$$\Pr[X \in (a,b)] \;=\; \Pr[X \in (a,b]] \;=\; \Pr[X \in [a,b)] \;=\; \frac{b-a}{u}$$

whenever $0 \leq a \leq b \leq u$.

Binomial distribution

Suppose we have a biased coin that comes up heads with probability p. We perform n independent flips and let X be the number of times the coin was heads. Then X is a ***binomial random variable***. This distribution is denoted $B(n,p)$. The key properties of this distribution are as follows.

$$\mathrm{E}[X] \;=\; np \tag{A.3.5}$$

$$\Pr[X = i] \;=\; \binom{n}{i} p^i (1-p)^{n-i} \qquad \forall i \in \{0,1,\ldots,n\} \tag{A.3.6}$$

References: (Lehman et al., 2018, Sections 19.3.4 and 19.5.3), (Anderson et al., 2017, Definition 2.32), (Cormen et al., 2001, equations (C.34) and (C.37)), Wikipedia

Since np is the expectation of a binomial random variable, one's intuition might suggest that np is the most likely outcome (i.e., the mode). This is not quite true — for example, np need not be an integer, in which case it cannot be an outcome. However, a most likely outcome is not far away.

Fact A.3.18. Assuming that $p < 1$, $\lfloor (n+1)p \rfloor$ is a most likely outcome of a binomial random variable.

A detailed discussion of tail bounds for binomial random variables is in Chapter 9. Here we will just mention a small claim that is occasionally useful.

Fact A.3.19. Let X have the binomial distribution $B(n,p)$. Then $\Pr[\, X \geq k \,] \leq \binom{n}{k} p^k$.

Proof. Let $\mathcal{S}$ be the collection of all sets of k trials, and note that $|\mathcal{S}| = \binom{n}{k}$. If $X \geq k$ then there must be some set of k trials that succeeded. Thus

$$\begin{aligned}
\Pr[\, X \geq k \,] &\leq \Pr[\, \exists S \in \mathcal{S} \text{ such that all trials in } S \text{ succeeded} \,] \\
&\leq \sum_{S \in \mathcal{S}} \Pr[\, \text{all trials in } S \text{ succeeded} \,] \qquad \text{(by the union bound, Fact A.3.8)} \\
&= |\mathcal{S}| \cdot p^{|S|} \\
&= \binom{n}{k} p^k
\end{aligned}$$

□

References: (Cormen et al., 2001, Theorem C.2).

Geometric distribution

These random variables have many nice uses, so we'll discuss them in more detail. Suppose we have a biased random coin that comes up heads with probability $p > 0$. A ***geometric random variable*** describes the number of independent flips needed until seeing the first heads.

Confusion sometimes arises because there are two commonly used definitions. To clarify matters, let us briefly summarize the two definitions and their key properties. The first definition is as follows.

$$\begin{array}{rll}
\multicolumn{3}{c}{\underline{\#\text{ trials } \textit{strictly before} \text{ the first head}}} \\
\Pr[\, X = k \,] = & (1-p)^k p & \forall k \geq 0 \\
\Pr[\, X \geq k \,] = & (1-p)^k & \forall k \geq 0 \\
\mathrm{E}[\, X \,] = & \frac{1}{p} - 1 &
\end{array} \tag{A.3.7}$$

The X in the second definition differs from the X in the first definition simply by adding 1.

$$\begin{array}{rll}
\multicolumn{3}{c}{\underline{\#\text{ trials } \textit{up to and including} \text{ the first head}}} \\
\Pr[\, X = k \,] = & (1-p)^{k-1} p & \forall k \geq 1 \\
\Pr[\, X \geq k \,] = & (1-p)^{k-1} & \forall k \geq 1 \\
\mathrm{E}[\, X \,] = & \frac{1}{p} &
\end{array} \tag{A.3.8}$$

References: (Lehman et al., 2018, Definition 19.4.7 and Lemma 19.4.8), (Anderson et al., 2017, Definition 2.34), (Cormen et al., 2001, equations (C.31) and (C.32)), (Kleinberg and Tardos, 2006, Theorem 13.7).

For the remainder of this section, we'll discuss the second definition. Let H denote heads and T denote tails. A sequence HT denotes that the first outcome was heads and the second was tails, etc. Then

- $\Pr[\, \mathsf{H} \,] = p$
- $\Pr[\, \mathsf{TH} \,] = (1-p)p$
- $\Pr[\, \mathsf{TTH} \,] = (1-p)^2 p$

Generally, the probability that the first head happens on the k^{th} trial is $(1-p)^{k-1}p$. Now let X be the RV taking the value k if the first head appeared on the k^{th} trial. Then

$$\Pr[X = k] \;=\; (1-p)^{k-1}p. \tag{A.3.9}$$

We now prove two properties of X that were mentioned above.

Claim A.3.20. For all $k \geq 1$, $\Pr[X \geq k] = (1-p)^{k-1}$.

Proof sketch. The event "$X \geq k$" happens precisely when the first $k-1$ tosses are tails. This happens with probability $(1-p)^{k-1}$. □

Fact A.3.21. Let X be a geometric random variable with parameter p. Then $\mathrm{E}[X] = 1/p$.

References: (Lehman et al., 2018, Lemma 19.4.8), (Cormen et al., 2001, equations (C.32)), (Kleinberg and Tardos, 2006, Theorem (13.7)).

This is very intuitive: if a trial succeeds 1/5th of the time, it seems obvious that it takes 5 trials to see a success. Nevertheless, a quick proof is required.

Proof of Fact A.3.21.

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k \geq 1} \Pr[X \geq k] \quad (\text{by Fact A.3.10})\\
&= \sum_{k \geq 1} (1-p)^{k-1} \quad (\text{by Claim A.3.20})\\
&= \frac{1}{1-(1-p)} \;=\; \frac{1}{p}.
\end{aligned}$$

Here we have used the formula for a geometric series discussed in Fact A.2.2. □

Problem to Ponder A.3.22. What is $\Pr[X = \infty]$?

References: (Anderson et al., 2017, Example 1.16).

A.3.4 Markov's Inequality

Fact A.3.23 (Markov's Inequality)**.** Let Y be a random variable that only takes non-negative values. Then, for all $a > 0$,

$$\Pr[Y \geq a] \;\leq\; \frac{\mathrm{E}[Y]}{a}.$$

References: (Lehman et al., 2018, Theorem 20.1.1), (Anderson et al., 2017, Theorem 9.2), (Cormen et al., 2001, Exercise C.3-6), (Harchol-Balter, 2023, Theorem 5.16), (Motwani and Raghavan, 1995, Theorem 3.2), (Mitzenmacher and Upfal, 2005, Theorem 3.1), (Grimmett and Stirzaker, 2001, Lemma 7.2.7), Wikipedia.

Fact A.3.24. Let Y be a random variable that only takes nonnegative values. Then, for all $b > 0$,

$$\Pr[Y \geq b \cdot \mathrm{E}[Y]] \;\leq\; \frac{1}{b}.$$

Proof. Simply apply Fact A.3.23 with $a = b \cdot \mathrm{E}[Y]$. □

References: (Lehman et al., 2018, Corollary 20.1.2), (Motwani and Raghavan, 1995, Theorem 3.2).

A.3.5 Random numbers modulo q

Fact A.3.25. Let $q \geq 2$ be an arbitrary integer. Let X be uniformly random in $[\![q]\!]$. Then, for any fixed integer y,

$$(X + y) \bmod q$$

is also uniformly random in $[\![q]\!]$.

Proof. Fix any any $z \in [\![q]\!]$. By the principle of "taking mod everywhere" (Fact A.2.11), we have

$$\Pr[\,(X + y) \bmod q = z\,] \;=\; \Pr[\,X = (z - y) \bmod q\,] \;=\; 1/q$$

since X is uniform. $\square$

Corollary A.3.26. Let $q \geq 2$ be an arbitrary integer. Let X be uniformly random in $[\![q]\!]$. Let Y be any random integer that is independent from X. Then

$$(X + Y) \bmod q$$

is also uniformly random in $[\![q]\!]$.

A.4 Exercises

Exercise A.1. Let $\mathcal{A}$ and $\mathcal{B}$ be events with $\Pr[\mathcal{B}] > 0$. Prove that $\Pr[\mathcal{A} \wedge \mathcal{B}] \leq \Pr[\mathcal{A} \mid \mathcal{B}]$.

Exercise A.2. For any events $\mathcal{A}$ and $\mathcal{B}$, prove that

$$\Pr\left[\mathcal{A} \wedge \overline{\mathcal{B}}\right] \;\geq\; \Pr[\mathcal{A}] - \Pr[\mathcal{B}].$$

Exercise A.3. The usual form of the union bound is

$$\Pr[\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n] \;\leq\; \sum_{i=1}^{n} \Pr[\mathcal{E}_i]. \tag{A.4.1}$$

for any events $\mathcal{E}_i$. Eq. (A.3.1) states the equivalent form

$$\Pr\left[\overline{\mathcal{E}_1} \wedge \cdots \wedge \overline{\mathcal{E}_n}\right] \;\geq\; 1 - \sum_{i=1}^{n} \Pr[\mathcal{E}_i]. \tag{A.4.2}$$

Prove (A.4.2) using (A.4.1).

Exercise A.4. Let $\mathcal{E}$ and $\mathcal{F}$ be events with $\Pr[\mathcal{E}] < 1$. Prove that $\Pr[\mathcal{F}] \leq \Pr[\mathcal{E}] + \Pr\left[\mathcal{F} \mid \overline{\mathcal{E}}\right]$.

Exercise A.5. Let $U_1, \ldots, U_d$ be continuous random variables that are independent and uniformly distributed on the interval $[0,1]$. Prove that $\Pr[\min\{U_1, \ldots, U_d\} > r] = (1-r)^d$.

Exercise A.6. Prove Corollary A.3.26 using Fact A.3.25.

Hint: First prove that $\Pr[(X+Y) \bmod m = z \mid Y = y] = 1/m$ for every integer y and every $z \in [\![m]\!]$.

Exercise A.7. Let G be a finite group. (For a definition, see (Cormen et al., 2001, pp. 939), or Wikipedia.)

Part I. Let X be uniformly random in G. Prove that, for any fixed group element y, $X + y$ is also a uniformly random element of G.

Part II. Let X be uniformly random in G. Let Y be any random element of G that is independent from X. Prove that $X + Y$ is also uniformly random in G.

Hint: The argument is identical to Exercise A.6.

Exercise A.8. Prove Fact A.2.16 using two applications of Fact A.2.14.

Bibliography

Anderson, D. F., Sepäläinen, T., and Valkó, B. (2017). *Introduction to Probability.* Cambridge.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms.* MIT Press, Cambridge, MA, second edition.

Cover, T. and Thomas, J. (1991). *Elements of Information Theory.* Wiley-Interscience.

Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. (2006). *Algorithms.* McGraw-Hill Education.

Erdős, P. (1967). Gráfok páros körüljárású részgráfjairól (On bipartite subgraphs of graphs, in Hungarian). *Mat. Lapok*, 18:283–288.

Feller, W. (1968). *An Introduction to Probability Theory and Its Applications, Volume I.* John Wiley & Sons, third edition.

Grimmett, G. and Stirzaker, D. (2001). *Probability and Random Processes.* Oxford University Press, third edition.

Harchol-Balter, M. (2023). *Introduction to Probability for Computing.* Cambridge University Press.

Kleinberg, J. and Tardos, E. (2006). *Algorithm Design.* Pearson.

Knuth, D. E. (2014). *Art of computer programming, volume 2: Seminumerical algorithms.* Addison-Wesley Professional.

Kushilevitz, E. and Nisan, N. (1997). *Communication Complexity.* Cambridge University Press.

Larsen, R. J. and Marx, M. L. (2018). *An Introduction to Mathematical Statistics and Its Applications.* Pearson, sixth edition.

Lehman, E., Leighton, F. T., and Meyer, A. R. (2018). Mathematics for computer science. https://courses.csail.mit.edu/6.042/spring18/mcs.pdf.

Lehmann, E. L. and Romano, J. P. (2022). *Testing Statistical Hypotheses.* Springer.

Mitzenmacher, M. and Upfal, E. (2005). *Probability and computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press.

Motwani, R. and Raghavan, P. (1995). *Randomized Algorithms.* Cambridge University Press.

Raab, M. and Steger, A. (1998). "balls into bins" – a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer.

Sen, S. and Kumar, A. (2019). *Design and Analysis of Algorithms: A Contemporary Perspective*. Cambridge University Press.

Sipser, M. (2012). *Introduction to the Theory of Computation*. Course Technology, third edition.

Vershynin, R. (2018). *High-Dimensional Probability: An Introduction with Applications in Data Science*. Cambridge University Press.
https://www.math.uci.edu/~rvershyn/papers/HDP-book/HDP-book.pdf.