A first course in randomized algorithms

Nick Harvey
University of British Columbia

September 1, 2022
Contents

Preface 7

I Basics 8

1 Sampling numbers 9
1.1 Uniform random variables 9
1.1.1 Uniform \( \ell \)-bit integers 9
1.1.2 Continuous random variables 10
1.1.3 Uniform on any finite set 10
1.2 Biased coin from unbiased coin 12
1.3 General distributions 13
1.4 Fair coin from a biased coin 15

2 Sampling objects 20
2.1 Random permutations 20
2.1.1 Efficiency 21
2.1.2 An approach based on sorting 22
2.2 Random subsets 23
2.3 Random partitions 24
2.4 Reservoir sampling 27

3 Errors in randomized algorithms 30
3.1 Toy example: testing if zero 30
3.2 Different types of error 31
3.3 Probability amplification, for one-sided error 32

4 Expectation 35
4.1 Toy example: fixed points of a permutation 35
10.1 Distinguishing coins
10.2 Polling
  10.2.1 Sampling with replacement
  10.2.2 Sampling without replacement
  10.2.3 Bernoulli sampling
10.3 A/B Testing
10.4 Estimating Distributions
  10.4.1 Bernoulli distributions
  10.4.2 Finite distributions
  10.4.3 Estimating the cumulative distribution

III Hashing

11 Introduction to Hashing
  11.1 Various types of hash functions
  11.2 MinHash: estimating set similarity
  11.3 MinHash: near neighbours
  11.4 Bloom filters

12 Streaming algorithms
  12.1 The streaming model
  12.2 The majority problem
  12.3 Heavy hitters
  12.4 Frequency estimation
  12.5 The streaming model with deletions
  12.6 Frequency estimation with deletions
  12.7 Distinct elements

13 Low-space hash functions
  13.1 Linear hashing modulo a prime
  13.2 Binary linear hashing
  13.3 Polynomial hashing

14 Applications of Low-Space Hashing
  14.1 Equality testing
  14.2 Count-Min Sketch
A.2.2 Number theory
A.3 Probability
A.3.1 Events
A.3.2 Random variables
A.3.3 Common Distributions
A.3.4 Markov’s Inequality
A.3.5 Random numbers modulo q

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

Shankar Vedantam

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.

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

Sampling numbers

It is quite common for computer programs to invoke a function that generates numbers that seem random. 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 1.1 illustrates the different methods and the subroutines that they use.

1.1 Uniform random variables

Perhaps the simplest random variable to generate is an unbiased random bit, which we will also call a fair coin, or a Bernoulli random variable with probability $p = 1/2$. In this section we will generate various random variables, assuming access to subroutine **UnbiasedBit** that returns an unbiased random bit, independent from all previous bits.

1.1.1 Uniform $\ell$-bit integers

Suppose we want to generate a uniform $\ell$-bit integer. The natural idea is to use our subroutine **UnbiasedBit** to generate the independent bits $X_1, \ldots, X_\ell$. Concatenating those bits gives a $\ell$-bit integer, which we claim is uniform.

To see this, note that there are $2^\ell$ different $\ell$-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 $\ell$-bit integer $x_1x_2\ldots x_\ell$, we have

$$
\Pr[X_1X_2\ldots X_\ell = x_1x_2\ldots x_\ell] = \Pr[X_1 = x_1 \wedge X_2 = x_2 \wedge \cdots \wedge X_\ell = x_\ell] \\
= \Pr[X_1 = x_1] \cdot \Pr[X_2 = x_2] \cdots \Pr[X_\ell = x_\ell] \\
= \frac{1}{2} \cdot \frac{1}{2} \cdots \frac{1}{2} = 2^{-\ell}.
$$

(1.1.1)

Here the second equality uses independence, and the third equality uses the fact that each $X_i$ is unbiased.
1.1.2 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. Instead, if using fixed-point arithmetic, we could use $\ell$ 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 1.1** Generating a uniform random variable on $[0, 1]$.

1: global $\ell$ \hspace{1cm} \triangleright Number of bits of precision
2: function ContinuousUniform
3: \hspace{1cm} $X \leftarrow \text{UniformLBitInteger}(\ell)$
4: \hspace{1cm} return $X \cdot 2^{-\ell}$
5: 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.3.15), 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 $\ell$, so ContinuousUniform is still useful in many applications.

1.1.3 Uniform on any finite set

We can also use UniformLBitInteger to generate a random variable that is uniform on any finite set. Actually, it suffices to generate a random variable that is uniform on

$$\lbrack n \rbrack = \{0, 1, \ldots, n - 1\}$$

because any finite set has a one-to-one correspondence with such a set. If $n$ were a power of two, we’d be done — this is precisely the set of $\lg(n)$-bit integers, which we handled above.
So suppose $n$ is not a power of two. One idea is to increase $n$ by rounding up to a power of two by creating some “dummy items”. This rounding concept is explained in Definition A.2.3. Including the dummies, there will now be $2^{\lceil \lg n \rceil}$ items. We then create a uniform sample from this larger set, which is simply a $\lceil \lg n \rceil$-bit integer. What should we do if we sample a dummy element? A natural idea is to retry and draw a new sample. Pseudocode implementing this idea is as follows.

\begin{algorithm}
\caption{Generating a uniform random variable on $[n]$.}
\begin{algorithmic}[1]
\Function{UniformInt}{int $n$}
\State $\ell \leftarrow \lceil \lg n \rceil$
\While{True}
\State $U \leftarrow \text{UniformLBitInteger}(\ell)$
\If{$U < n$} \Return $U$ \EndIf
\EndWhile
\EndFunction
\end{algorithmic}
\end{algorithm}

The main idea of this algorithm is quite intuitive: if $U$ is uniformly distributed on $[2^\ell]$ then, conditioned on lying in $[n]$, $U$ becomes uniformly distributed on $[n]$. The algorithm 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.

\begin{algorithm}
\caption{Given sets $R \subseteq S$, use uniform samples on $S$ to generate uniform samples on $R$.}
\begin{algorithmic}[1]
\Function{RejectionSample}{set $R$, set $S$}
\While{True}
\State Let $U$ be a RV that is uniform on $S$
\If{$U \in R$} \Return $U$ \EndIf
\EndWhile
\EndFunction
\end{algorithmic}
\end{algorithm}

First we argue that \texttt{RejectionSample} generates uniform samples on $R$.

**Claim 1.1.1.** Let $R$ and $S$ be finite non-empty sets with $R \subseteq S$. Suppose that $U$ is uniform on $S$. Then, conditioned on lying in $R$, $U$ is uniform on $R$. In other words,

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

**Proof.** This is just the definition of conditional probability.

$$
\Pr[U = x \mid U \in R] = \frac{\Pr[U = x \land U \in R]}{\Pr[U \in R]} = \frac{1/|S|}{\sum_{r \in R} 1/|S|} = \frac{1/|S|}{|R|/|S|} = \frac{1}{|R|}.
$$

© Nicholas Harvey 2021. All rights reserved.
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.3.18.** Let $X$ be a geometric random variable with parameter $p$. Then $E[X] = 1/p$.

This discussion proves the following claim.

**Claim 1.1.2.** The expected number of iterations of REJECTIONSAMPLE is $1/p = |S|/|R|$.

**Interview Question 1.1.3.** 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

Note that $\text{UNIFORMINT}(n)$ is equivalent to $\text{REJECTIONSAMPLE}([n], [2^\ell])$ where $\ell = \lceil \log n \rceil$. So Claim 1.1.1 implies that the output is uniform on $[n]$. Next let us consider the runtime. By our choice of $\ell$, (A.2.3) tells us that $\frac{1}{2} \cdot 2^\ell < n \leq 2^\ell$. Rearranging the first inequality, we get

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

So Claim 1.1.2 implies that $\text{UNIFORMINT}$ performs at most 2 iterations in expectation.

### 1.2 Biased coin from unbiased coin

How might we generate samples from such a biased coin for which the probability of heads is $b$?

**Question 1.2.1.** How do you simulate a biased random coin in your favourite programming language?

**Answer.**

```python
import random

def Flip(b):
    if random.uniform(0,1) <= b:
        print("Heads")
    else:
        print("Tails")
```

The approach used in the preceding answer is to generate a continuous random variable $X$ that is uniform on the interval $[0,1]$, then declare that the coin is heads if $X \leq b$. This event has probability $b$, as discussed in Section A.3.3.

There are some drawbacks of this approach. First of all, a computer cannot fully generate a real number uniformly on $[0,1]$ because it would require infinitely many bits to represent! Instead, a computer would typically generate a floating point number with only 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 $\Pr \left[ X \leq 1/3 \right]$ will not be exactly $1/3$.

Another drawback is wastefulness. We could generate an unbiased coin flip via the fact that $\Pr \left[ X \leq 1/2 \right] = 1/2$. But that seems pointless: generating $X$ presumably takes many random bits, whereas an unbiased coin flip should only need a single random bit!

A nice idea is to use a \textit{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

$$b = 0.b_1b_2b_3\ldots = 0.01010101\ldots$$

since $1/3 = 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots$. Similarly, let us represent $X$ in binary as $0.X_1X_2X_3\ldots$. 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 $X \leq b$ or $X > b$. This idea is implemented in the following pseudocode.

\begin{algorithm}
\caption{Generating a random bit that is 1 with probability $b$.}
\begin{algorithmic}[1]
\Function{BiasedBit}{real $b$}
\State Let $0.b_1b_2b_3\ldots$ be the binary representation of $b$.
\For{$i = 1, 2, \ldots$} \Comment{The first $i − 1$ bits of $X$ match $b$}
\State Let $X_i \leftarrow \text{UnbiasedBit}()$
\If{$X_i > b_i$} \Comment{We now know that $X > b$}
\State \Return 0
\EndIf
\If{$X_i < b_i$} \Comment{We now know that $X < b$}
\State \Return 1
\EndIf
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

\textbf{Question 1.2.2.} In expectation, how many iterations of the for loop are required?

\textbf{Answer.} The function terminates at the first iteration $i$ for which $X_i$ and $b_i$ disagree (either $X_i = 1 \land b_i = 0$, or $X_i = 0 \land b_i = 1$). Let $\mathbb{E} \left[ i \right]$ be the expected number of iterations when $b_i = 0$. Then $\mathbb{E} \left[ i \right]$ is a geometric random variable with expectation 2. (Cormen et al., 2001, Exercise C.2-6).

\textbf{References:} (Cormen et al., 2001, Exercise C.2-6).

\section{1.3 General distributions}

### 1.3.1 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 \textit{non-uniform} probabilities on a finite set. It suffices to consider sampling from $[k]$, because any finite set can be mapped to such a set. This is called a \textit{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

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

The following approach is very natural. First, partition the unit interval $[0, 1]$ into $k$ segments, where
the $i^{th}$ segment has length $p_i$. Then, generate a random variable $Y$ that is uniform on $[0, 1]$. If $Y$ lands in the $i^{th}$ segment, then return the value $i$.

The only aspect that requires any thought is computing the endpoints of the segments. We can arrange the segments so that the right endpoint of the $i^{th}$ segment has the value $\sum_{1 \leq j \leq i} p_j$. This can be implemented with the following pseudocode.

**Algorithm 1.5** Generating a categorical random variable.

1: function GENERALFINITEDISTR(float $p[1..k]$)  
2:   for $i = 0, \ldots, k$  
3:     $q_i \leftarrow \sum_{1 \leq j \leq i} p_j$  
4:     Let $Y \leftarrow \text{ContinuousUniform}()$  
5:     Find $i$ such that $Y$ lies in the interval $(q_{i-1}, q_i]$  
6:     return $i$  
7: end function

**Question 1.3.1.** Suppose you wanted to generate many samples from the same categorical distribution. If implementing this approach as a data structure, how much time would be needed for preprocessing and how much to generate each sample?

**Answer.**

Each query can be done with a binary search in $\mathcal{O}(\log k)$ time. The data structure would store the $q_i$ values, the preprocessing can be implemented in $\mathcal{O}(k)$ time.

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 $F$ where

$$F(i) = \Pr[X \leq i] = \sum_{1 \leq j \leq i} p_j.$$ 

Using the notation of the CDF, the algorithm could be rewritten as follows.

**Algorithm 1.6** Generating a categorical random variable using the CDF.

1: function GENERALFINITEDISTR(CDF $F$)  
2:   Let $Y \leftarrow \text{ContinuousUniform}()$  
3:   return $i$ such that $Y \in (F(i-1), F(i)]$  
4: end function

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

### 1.3.2 General continuous random variables

Algorithm 1.6 shows how to sample from any finite distribution by using the CDF. A similar approach works for all random variables. We can imagine discretizing an arbitrary RV into discrete bins of width $\epsilon$, and rewriting line 3 as

\[
\text{return } x \text{ such that } Y \in (F(x - \epsilon), F(x)].
\]

As the bin width $\epsilon$ decreases to 0, this condition becomes $Y = F(x)$. This is equivalent to $x = F^{-1}(Y)$ if $F$ is sufficiently nice. This idea is illustrated in the following pseudocode.
Algorithm 1.7 Generating a continuous RV using the CDF.

1: function GeneralContDistr(CDF $F$)
2:   Let $Y \leftarrow \text{ContinuousUniform}()$
3:   return $F^{-1}(Y)$
4: end function

Theorem 1.3.2. Consider a distribution whose CDF $F$ is continuous and strictly increasing. Then GeneralContDistr outputs a RV with that distribution.

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 $\Pr[F^{-1}(Y) \leq x]$. 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

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

Here we have used that $Y$ is a RV that is uniform on $[0,1]$, and equation (A.3.2). This shows that $F$ is the CDF of $F^{-1}(Y)$, which is the output of the algorithm.

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

Keener Kwestion 1.3.3. How could you modify GeneralContDistr to handle $F$ that are not continuous or not strictly increasing?

1.4 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 this is a computer science class, not an engineering class. So our solution should involve writing code, not mechanical repairs.

More concretely, 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

$$\Pr[\text{Fair()} = \text{True}] = 1/2 \quad \text{and} \quad \Pr[\text{Fair()} = \text{False}] = 1/2$$

$$\Pr[\text{Biased()} = \text{True}] = p \quad \text{and} \quad \Pr[\text{Biased()} = \text{False}] = 1 - p.$$

What can we do?

Just one call? Is it possible to do anything useful if our function Fair calls Biased just once? This call to Biased splits our execution into two possible “execution paths”: the one in which Biased returned True, and the one in which it returned False. In each of those paths, Fair can only return one possible value (because it has no inputs, and no additional source of randomness). One path has probability $p$ of occurring, and the other path has $1 - p$ of occurring. So some output has probability $p$ and the other has probability $1 - p$, which is not what we want. (Even worse, if both paths return the same value, then that output will have probability 1, and the other will have probability 0!)
Just two calls? Is it possible to do anything useful if our function `FAIR` calls `BIASED` twice? Let’s plot the possible outcomes.

<table>
<thead>
<tr>
<th>First call to <code>BIASED</code></th>
<th>Second call to <code>BIASED</code></th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td></td>
<td>$p^2$</td>
<td>$p(1 - p)$</td>
</tr>
<tr>
<td>False</td>
<td></td>
<td>$p(1 - p)$</td>
<td>$(1 - p)^2$</td>
</tr>
</tbody>
</table>

**Figure 1.2:** Probabilities of different outcomes.

Now there are four outcomes, each of which must generate some particular `True/False` output. We want:

- some subset of the outcomes should cause `FAIR` to output `True`, and the total probability of those outcomes should be 0.5.
- the remaining outcomes should cause `FAIR` to output `False`, and the total probability of those outcomes should also be 0.5.

For example, if $p = 0.75$ then we have

<table>
<thead>
<tr>
<th>First call to <code>BIASED</code></th>
<th>Second call to <code>BIASED</code></th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td></td>
<td>0.5625</td>
<td>0.1875</td>
</tr>
<tr>
<td>False</td>
<td></td>
<td>0.1875</td>
<td>0.0625</td>
</tr>
</tbody>
</table>

Now we want to find some subset of these outcomes whose total probability is 0.5. But clearly no such subset exists! So, sadly, there is no way to produce `True` and `False` with probability 0.5.

A sneaky idea. Staring at the previous table, we notice something interesting, and a sneaky idea occurs. There are two entries in the table with the same value 0.1875. Maybe one of those could output `True` and the other output `False`? Then at least `True` and `False` would have the same probability. Admittedly this probability is 0.1875, which is not what we wanted, but it feels like we’re making progress.

What to do about the other table entries? One idea is to employ the rejection sampling approach from Section 1.1.3. We can adapt that idea for our current setting as follows: whenever the outcome is one of the undesired table entries, simply restart the algorithm and try again!

**Algorithm 1.8** An implementation of `FAIR` based on rejection sampling.

1: function `FAIR`
2:    while True do
3:       $c_1 \leftarrow \text{BIASED}()$
4:       $c_2 \leftarrow \text{BIASED}()$
5:       if $c_1 = \text{True}$ and $c_2 = \text{False}$ then return `True`
6:       if $c_1 = \text{False}$ and $c_2 = \text{True}$ then return `False`
7:    end while
8: end function

**Claim 1.4.1.** This algorithm outputs `True` and `False` with equal probability.
Proof sketch. Upon starting the $i$th iteration, the two calls to Biased are equally likely to return TF or FT. So the True output always has the same probability as the False output. 

Runtime. A puzzling aspect of the FAIR algorithm is that its runtime is random! But fortunately the runtime is easy to analyze using geometric random variables. We may view each iteration as a random trial where “success” means that it returns a True/False output. For each trial, its probability of success is $2p(1-p)$. Since the algorithm continues until the first success, the number of iterations $X$ is a geometric random variable. Once again using Fact A.3.18, we have

$$E[X] = \frac{1}{\text{probability of success}} = \frac{1}{2p(1-p)}.$$  

Notice that if the biased coin is very biased then the expected number of iterations is very large. For example, $p = 0.01$ then $E[X] > 50$. This makes sense: if the biased coin is not very random, then it takes a lot of trials to extract an unbiased random bit.


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

Interview Question 1.4.2. This is apparently a common interview question. See, e.g., here, or here.
1.5 Exercises

Exercise 1.1 Generating uniform random numbers. Suppose you have access to a random number generator \( \text{Rng}() \) that generates uniform random numbers in \([n]\). Note that \( \text{Rng}() \) 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 \([m]\).

1: function BadSampler(integer \( m \))  
2: \quad \text{return} \text{Rng}() \mod m  
3: end function

**Part II.** Design a function \( \text{GoodSampler}(\text{integer} \ m) \) that uses \( \text{Rng}() \) is 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 \( \text{GoodSampler}(m) \) makes \( O(1) \) calls to \( \text{Rng}() \) in expectation.

You should prove these two properties.

Exercise 1.2. Consider the following pseudocode.

1: function Mystery(int \( n \))  
2: \quad p \leftarrow \text{ContinuousUniform}()  
3: \quad \text{for} i = 1, \ldots, n \ \text{do}  
4: \quad \quad X_i \leftarrow \text{BiasedBit}(p)  
5: \quad \text{end for}  
6: \quad \text{return} \sum_{i=1}^{n} X_i  
7: end function

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

Exercise 1.3. 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

\[
F(x) = \frac{1}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right),
\]

where \( \text{erf} \) is a special function called the error function.

Implement code in Julia to generate a standard normal random variable using Algorithm 1.7 and Julia’s library of special functions. How fast is this approach compared to Julia’s \texttt{randn} function?
Exercise 1.4 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 $O((\log n)^k)$ 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 $O((\log n)^{10})$. You may use the fact that there is a deterministic algorithm with runtime $O((\log n)^7)$ to test if the number $n$ is prime.

Exercise 1.5 Continuous Rejection Sampling. Let’s consider uniformly sampling from some continuous sets. The following is a continuous analog of Claim 1.1.1.

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 $R$?
Chapter 2
Sampling objects

2.1 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 total ordering of $C$ is the same as an $n$-permutation. The number of $k$-permutations is

$$n \cdot (n-1) \cdots (n-k+1) = \frac{n!}{(n-k)!}.$$  \hspace{1cm} (2.1.1)

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 2.1 presents a very natural approach to generate a uniformly random $k$-permutation. It is usually attributed to Fischer and Yates, despite being rather obvious.

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

Algorithm 2.1 Generate a uniformly random $k$-permutation of $C$. Let $n = |C|$.

1: function GENKPERM(list $C$, int $k$)
2:     Let $L$ be an empty list
3:     for $i = 1, \ldots, k$ do
4:         Let $c$ be a uniformly random element of $C$ \hspace{1cm} \triangleright \text{Length}(C) = n - i + 1
5:         Remove $c$ from $C$
6:         Append $c$ to $L$
7:     end for
8:     return $L$
9: 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

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

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

**Claim 2.1.1.** GENKPERM generates a uniformly random $k$-permutation.

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

$$
\Pr [E_i \mid E_1 \land \cdots \land E_{i-1}] = \frac{1}{n-i+1}.
$$

Thus, by the chain rule,

$$
\Pr [L = B] = \Pr [E_1 \land \cdots \land E_k] = \prod_{i=1}^{k} \Pr [E_i \mid E_1 \land \cdots \land E_{i-1}] = \prod_{i=1}^{k} \frac{1}{n-i+1} = \frac{(n-k)!}{n!},
$$

the reciprocal of (2.1.1). It follows that each $k$-permutation $B$ is picked with uniform probability.

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

**Claim 2.1.2.** 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 $i \in C$, $\Pr [i \in L] = k/n$.

**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,

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

**Question 2.1.3.** 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.

**Keener Kwestion 2.1.4.** Prove Claim 2.1.2 part (a) using Exercise A.7.

### 2.1.1 Efficiency

A subtle question is: what is the efficiency of Algorithm 2.1? 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.

**Keener Kwestion 2.1.5.** Using a balanced binary tree to represent \( C \), show that Algorithm 2.1 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 2.2** 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 element of \{1, ..., n\}
4:        Swap \( C[i] \) and \( C[r] \)
5:    return \( C[1..k] \)
6: end function
```

**Theorem 2.1.6.** GENKPERM produces a \( k \)-permutation. Assuming that the uniformly random numbers can be generated in constant 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 immediate: lines 3 and 4 each time 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 2.1. So, by Claim 2.1.1, the implementation in Algorithm 2.2 also generates a uniformly random permutation.

\[\Box\]

**2.1.2 An approach based on sorting**

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

**Algorithm 2.3** An algorithm based on sorting to generate a uniformly random permutation of \( C \).

```
1: function GENPERMBYSORTING(array C[1..n])
2:    Create an array \( X[1..n] \) containing independent random real numbers in \([0, 1]\)
3:    Sort \( C[1..n] \) using \( X[1..n] \) as the sorting keys
4:    return \( C \)
5: 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 5.6.

**Claim 2.1.7.** 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 $\pi$. (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 $\pi, \sigma$,

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

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 5.2. \qed

### 2.2 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.

**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 5.5.)

---

\(^1\)Or, if we forget the ordering of $S$, it is a multiset.

\(^2\)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.
Algorithm 2.4 Three approaches to generating random subsets of $C$. Assume that $|C| = m$.

```plaintext
1: function SampleWithReplacement(list $C$, int $k$)
2:   Let $S$ be an empty multiset
3:   for $i = 1, \ldots, k$
4:     Let $c$ be a uniformly random element of $C$
5:     Add $c$ to $S$
6:   return $S$
7: end function

8: function SampleWithoutReplacement(list $C$, int $k$)
9:   $L \leftarrow \text{GenKPerm}(C, k)$ \Comment{See Algorithm 2.2.}
10:  return $L$, but converted into a set to forget its order
11: end function

12: function BernoulliSample(list $C$, int $k$)
13:   Let $S$ be an empty set
14:   foreach $i \in C$
15:     Flip a coin that is heads with probability $p = k/m$
16:     if the coin is heads then add $i$ to $S$
17:   return $S$
18: end function

- 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 $B(m, k/m)$, so it has expectation $m \cdot (k/m) = k$. Again, all items are distinct.

Question 2.2.1. When sampling without replacement, what is the probability that $S = \{1, \ldots, k\}$?

Answer.

\[
\left( \frac{\binom{m}{k}}{\binom{m}{k}} \right) / 1 \text{ There are sets of size } k \text{, so each subset is sampled with probability } \frac{\binom{m}{k}}{\binom{m}{k}}.\]

Question 2.2.2. Under these different approaches, are the samples independent? The phrasing of this question is deliberately vague.

Answer.

- Bernoulli sampling, for distinct items $c_i, c_j$ the events $c_i \in S$, $c_j \in S$ are independent.
- Sampling without replacement for distinct items $c_i, c_j$ the events $c_i \in S$, $c_j \in S$ are not independent.
- $\forall \ell \in [\ell], \{S_1, \ldots, S_\ell\} = \emptyset \text{, the samples } S_1, \ldots, S_\ell \text{ are independent.}$

2.3 Random partitions

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

\[ A_1 \cup \cdots \cup A_\ell = C \quad \text{and} \quad A_i \cap A_j = \emptyset \quad \forall \text{distinct } i, j \in [\ell]. \]
Algorithm 2.5 Two approaches to randomly partition $C$ into $\ell$ subsets. Assume that $|C| = m$ is a multiple of $\ell$. Let $k = m/\ell$.

1: function PartitionWithoutReplacement(list $C$, int $\ell$)
2:     for $i = 1, \ldots, \ell$ do
3:         $A_i \leftarrow \text{GenKPerm}(C \setminus (A_1 \cup \cdots \cup A_{i-1}), k)$ ▷ See Algorithm 2.2.
4:     end for
5:     return $A_1, \ldots, A_\ell$
6: end function

7: function BernoulliPartition(list $C$, int $\ell$)
8:     Let $A_1, \ldots, A_\ell$ be empty sets
9:     foreach $c \in C$
10:         Let $X$ be a RV that is uniform on $[\ell]$
11:         Add $c$ to $A_X$
12:     return $A_1, \ldots, A_\ell$
13: end function

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 2.2 for generating random subsets can be modified to generate random partitions. Pseudocode is shown in Algorithm 2.5. We will let $m = |C|$ and assume that $k = m/\ell$ is an integer.

Analysis of BernoulliPartition\textsuperscript{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,

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

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.

Claim 2.3.1. $A_i$ 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. \qed

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 $O(\ell + m) = O(m)$.

\textsuperscript{3}The name of this algorithm comes from its connection to Bernoulli sampling; see Claim 2.3.1. However the RV generated in line 10 is not a Bernoulli RV.
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 2.3.2. $A_i$ 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,

\[
A_1 = L[1..k] \\
A_2 = L[1+k..2k] \\
A_3 = L[1+2k..3k] \\
\vdots
\]

The corresponding pseudocode is as follows.

Algorithm 2.6 A revised approach to partitioning without replacement.

1: function PartitionWithoutReplacement(list $C$, int $k$)
2: $L \leftarrow$ GenKPerm$(C,m)$ \quad \triangleright\text{See Algorithm 2.2.}$
3: for $i = 1, \ldots, \ell$ do
4: \hspace{1em} $A_i$ is the $i^{th}$ group of $k$ consecutive elements in $L$, namely $L[1 + (i-1) \cdot k..i \cdot k]$
5: end for
6: return $A_1, \ldots, A_\ell$
7: end function

This modified pseudocode is equivalent to the original pseudocode, but easier to analyze. First let us consider the runtime. Since GenKPerm$(C, m)$ takes $O(m)$ time (Theorem 2.1.6), the overall runtime of PartitionWithoutReplacement is $O(m + \ell \cdot k) = O(m)$ time.

Proof of Claim 2.3.2. For any distinct values $x_1, \ldots, x_k \in C$, there are exactly $(m-k)!$ permutations with

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

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

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

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 SampleWithoutReplacement$(C, k)$, this completes the proof.

Keener Kuestion 2.3.3. Prove Claim 2.3.2 using Exercise A.7.
2.4 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 to 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 $i$ an item $s_i$ is received. Suppose for simplicity that all items are distinct. The algorithm must maintain at every time $i$ an item $X_i$ that is a uniform sample amongst all items seen so far. At time $i$, when $s_i$ arrives, the algorithm must either

- ignore $s_i$, and set $X_i \leftarrow X_{i-1}$, or
- take $s_i$, and set $X_i \leftarrow s_i$.

2.4.1 Brainstorming

Let’s think through an example to see what to do.

**Time 1.** There is no choice: the algorithm must set $X_1 \leftarrow s_1$.

**Time 2.** We want to have $\Pr [X_2 = s_1] = \Pr [X_2 = s_2] = 1/2$. We already have $X_1 = s_1$. So the obvious idea is 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 the third item with probability 1/3 (regardless of the value of $X_2$). Conversely, $X_2$ will be kept 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 $3$.

$$\Pr [X_3 = s_1] = \Pr [X_2 = s_1 \land \text{algorithm ignores } s_3]$$
$$= \Pr [X_2 = s_1] \cdot \Pr [\text{algorithm ignores } s_3]$$
$$= \frac{1}{2} \cdot \frac{2}{3} = \frac{1}{3}.$$  

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

2.4.2 The algorithm

The above ideas are formalized as pseudocode in Algorithm 2.7.

**Theorem 2.4.1.** 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.
Algorithm 2.7 The reservoir sampling algorithm. Here $X_i$ is the item that is held by the algorithm.

1: function RESERVOIRSAMPLING
2: for $i = 1, 2, \ldots$ do
3: Receive item $s_i$
4: Flip a biased coin that is heads with probability $1/i$
5: if coin was heads then
6: Set $X_i \leftarrow s_i$
7: else
8: Set $X_i \leftarrow X_{i-1}$
9: end if
10: end for
11: end function

**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 $j < i$,

$$
\Pr[X_i = s_j] = \Pr[X_{i-1} = s_j \text{ and } i\text{th coin is tails}] \\
= \Pr[X_{i-1} = s_j] \cdot \Pr[i\text{th coin is tails}] \quad (\text{independence}) \\
= \frac{1}{i-1} \cdot \frac{i-1}{i} \quad (\text{by induction}) \\
= \frac{1}{i}.
$$

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

**Question 2.4.3.** 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.**
2.5 Exercises

Exercise 2.1. Consider the following algorithm for generating a uniformly random $k$-permutation.

Algorithm 2.8 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 $i = 1, \ldots, k$ do
4:     repeat
5:         Let $r$ be a uniformly random value in $[n]$
6:     until $C[r] \neq$ Null
7:     Append $C[r]$ to $L$
8:     $C[r] \leftarrow$ 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. Give an asymptotically tight bound on the expected runtime of GenKPermLazy as a function of $k$ and $n$. Your bound should not involve a summation.

Exercise 2.2. Show that GenKPermLazy can be modified to run in $O(k)$ time by periodically removing all the null elements from $C$.

Exercise 2.3. 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 2.4?

Exercise 2.4. 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($C, k$).
Chapter 3

Errors in randomized algorithms

In this chapter we will focus on decision problems, which are tasks that have a Yes/No (i.e., Boolean) output. We will discuss what sort of errors a randomized algorithm can have for such problems. Then we will describe a technique to decrease errors.

3.1 Toy example: testing if zero

Let us begin with a toy example of a decision problem that can be solved by a randomized algorithm.

Is a vector all zeros? Let \( a = (a_1, a_2, \ldots, a_n) \) be a string of \( n \) bits, with \( n \) even. Suppose we know that there are two possibilities.

Case 1. All bits in \( a \) are 0.
Case 2. Exactly half the bits in \( a \) are 0 and half are 1. However, we do not know in advance which bits are 0.

We would like to determine which of these cases holds. A trivial deterministic algorithm, meaning one without randomness, is as follows: examine all \( n \) bits, and return “Yes” if they are all 0. A trivial randomized algorithm is as follows: randomly sample a single bit, and return “Yes” if it is 0.

Notice that the randomized algorithm can make an error. Suppose we are in the second case, in which half the bits of \( a \) are 1, but the sampled bit happens to be 0. In this case the algorithm would incorrectly say “Yes”. However the probability of making this error is exactly 1/2 since, in this second case, the probability of randomly sampling a 0 bit is exactly 1/2.

Question 3.1.1. If we are in the first case, can the algorithm make an error?

Answer. No. In the first case, all bits are 0, so the sampled bit is 0, so the algorithm will say “Yes”.

Keener Kwestion 3.1.2. Is there a deterministic algorithm for this toy problem that examines fewer than \( n \) bits in the worst case? What about \( o(n) \) bits?

Although this toy problem might seem trivial, it is fundamental. It is called the Deutsch-Jozsa problem, and it is an early example of a problem for which quantum computers have a clear advantage. In this book we will encounter many more decision problems that can be solved by interesting randomized algorithms — for example, Section 15.2 discusses an algorithm that can decide whether a graph has a
cut of size at most $k$.

### 3.2 Different types of error

It is interesting to consider the types of error that can occur in randomized algorithms for decision problems.

<table>
<thead>
<tr>
<th>Correct Output</th>
<th>Algorithm’s Output</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>true positive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>false positive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: The possible outcomes and their correctness.

You may be familiar with this terminology from statistical tests, such as medical diagnostic tests. In that context, the matrix shown above is sometimes called a confusion matrix. Naturally, we want the probabilities of false positives and false negatives to be as small as possible. In the statistical literature one will encounter terminology such as “sensitivity” or “precision”, which are ways to measure the quality of a test using ratios of false positives, false negatives, etc.

In the study of randomized algorithms, we will instead consider certain types of algorithms for which the probabilities of these table entries satisfy certain constraints. We discuss three important types below. The first type is an “RP algorithm”.

<table>
<thead>
<tr>
<th>Correct Output</th>
<th>Algorithm’s Output</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>$\geq \frac{1}{2}$ true positive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>$\leq \frac{1}{2}$ false negative</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2: Probabilities of outcomes for an RP-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. 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. It is important to note that there are no assumptions regarding any distribution on the inputs.

It turns out to make very little difference if we changed the constant 1/2 in this table to any other constant strictly between 0 and 1. This follows from the discussion in Section 3.3 below.

**Question 3.2.1.** Suppose we required instead that true positives have probability $\leq \frac{1}{2}$ and false negatives had probability $\geq \frac{1}{2}$. Would this definition make sense too?

**Answer.**

It does not make sense. The algorithm that ignores the input and always returns No would satisfy this.

A “coRP algorithm” is analogous to an “RP algorithm” except that it makes no errors if the correct output is Yes.
If an algorithm has either no false positives or no false negatives, we sometimes say that it has \textit{one-sided error}. Otherwise, it has \textit{two-sided error}, and the key definition is as follows.

\begin{figure}[h!]
\centering
\begin{tabular}{|c|c|c|}
\hline
Correct Output & Algorithm’s Output & \multicolumn{1}{c|}{\textbf{Yes}} & \multicolumn{1}{c|}{\textbf{No}} \\
\hline
\textbf{Yes} & & \begin{tabular}{l}
true positive \\
\textbf{1} \\
\end{tabular} & \begin{tabular}{l}
false negative \\
\textbf{0} \\
\end{tabular} \\
\hline
\textbf{No} & & \begin{tabular}{l}
false positive \\
\textbf{≤} 1/2 \\
\end{tabular} & \begin{tabular}{l}
true negative \\
\textbf{≥} 1/2 \\
\end{tabular} \\
\hline
\end{tabular}
\caption{Probabilities of outcomes for a \textbf{coRP-algorithm}}
\end{figure}

As we will see in Section 9.2, 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 $1$.

\textbf{Question 3.2.2.} Consider our example of testing if a bitvector is zero. Say that “Yes” means the algorithm thinks the bitvector is zero, and “No” means it thinks it’s non-zero. Is this an RP-algorithm, coRP-algorithm or BPP-algorithm?

\textbf{Answer.}

It is a \textbf{coRP algorithm} because:

\begin{itemize}
\item \textbf{Otherwise its probability of error is exactly $1/2$.} \\
\item \textbf{It makes no errors if the correct output is \textit{Yes} (meaning the bitvector is zero).}
\end{itemize}

\subsection*{3.3 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 $B$ is an algorithm with $\Pr[\text{false positive}] \leq 1/2$ and $\Pr[\text{false negative}] = 0$ (a \textbf{coRP-algorithm}). So, if the algorithm outputs “no” then that is definitely the correct output. We design an “amplified” algorithm $A$ by performing repeated trials.
Algorithm 3.1 The algorithm $\mathcal{A}$ for amplifying success of algorithm $\mathcal{B}$. The parameter $\ell$ determines the number of trials.

1: function AUMLIFY(algorithm $\mathcal{B}$, integer $\ell$)
2:     for $i = 1, \ldots, \ell$ do
3:         Run algorithm $\mathcal{B}$ with “fresh randomness” (i.e., independent random bits)
4:         if $\mathcal{B}$ outputs no then
5:             Output no \(\triangleright\) This is definitely correct
6:         end if
7:     end for
8:     Output yes \(\triangleright\) $\mathcal{B}$’s output was always Yes, so this is probably correct
9: end function

Theorem 3.3.1. Assume that $\mathcal{B}$ is a coRP algorithm, meaning that it satisfies

$$\Pr[\text{false positive}] \leq 1/2$$

and

$$\Pr[\text{false negative}] = 0.$$ 

Then $\mathcal{A}$ satisfies

$$\Pr[\text{false positive}] \leq \frac{1}{2^\ell}$$

and

$$\Pr[\text{false negative}] = 0.$$ 

Proof. 

Case 1: the correct answer is yes. By hypothesis, $\mathcal{B}$ always outputs yes, so $\mathcal{A}$ outputs yes. So $\mathcal{A}$ has no false negatives. 

Case 2: the correct answer is no. We must analyze the probability of a false positive.

$$\Pr[\mathcal{A} \text{ incorrectly outputs yes}] = \Pr[\text{in all } \ell \text{ trials, } \mathcal{B} \text{ incorrectly outputs yes}]$$

$$= \prod_{i=1}^{\ell} \Pr[\text{in } i^{\text{th}} \text{ trial, } \mathcal{B} \text{ incorrectly outputs yes}]$$

$$\leq \prod_{i=1}^{\ell} \frac{1}{2}$$

(since $\mathcal{B}$ is a coRP-algorithm)

$$= \frac{1}{2^\ell}.$$

Amplification to our toy example. Recall that our zero-testing algorithm of Section 3.1 is a coRP-algorithm, so it satisfies the hypotheses of Theorem 3.3.1. If we run the amplification algorithm with $\ell = \lg(k)$, the failure probability is at most $1/2^\ell = 1/k$. 

©Nicholas Harvey 2021. All rights reserved.
### 3.4 Exercises

**Exercise 3.1 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.

<table>
<thead>
<tr>
<th>Correct Output</th>
<th>Algorithm’s Output</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>( \geq 1/n )</td>
<td>( \leq 1 - 1/n )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>true positive</td>
<td>false negative</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>false positive</td>
<td>true negative</td>
<td></td>
</tr>
</tbody>
</table>

Design a randomized algorithm \( \mathcal{A} \) that runs in \( o(n^3) \) time and has the following outcome probabilities.

<table>
<thead>
<tr>
<th>Correct Output</th>
<th>Algorithm’s Output</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>( \geq 1 - 1/n )</td>
<td>( \leq 1/n )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>true positive</td>
<td>false negative</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>false positive</td>
<td>true negative</td>
<td></td>
</tr>
</tbody>
</table>

Prove that your algorithm works.
Chapter 4

Expectation

Some probabilistic scenarios might quite difficult to analyze via one approach, but quite easy to analyze via another approach. Thus it is useful to identify approaches that often lead to simple analyses. One such approach is to leverage linearity of expectation, especially via decomposition into indicator random variables. We illustrate several examples in this chapter.

4.1 Toy example: fixed points of a permutation

Let us begin by discussing a toy problem. The CS department has run out of money for TAs, so it is decreed that students will have to grade each others’ 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. (In other words, 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 fairly unlikely. 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, $E[X] = \sum_{i=0}^{n} i \cdot Pr[X = i]$. So now we would have to figure out $Pr[X = i]$, which is a big mess! Even $Pr[X = 0]$ is the fraction of derangements, which already 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 person $i$ gets their own homework. Referring to Definition A.3.12, the corresponding indicator random variable is

$$X_i = \begin{cases} 
1 & \text{(if } i \text{ gets their own homework)} \\
0 & \text{(otherwise)}.
\end{cases}$$

The decomposition of $X$ into indicator RVs is

$$X = \sum_{i=1}^{n} X_i.$$
One reason why this is useful is the following fact. It is important to notice that this fact does not require independence of the events.

**Fact A.3.13.** Suppose that $X$ is a random variable that can be decomposed as $X = \sum_{i=1}^{n} X_i$, where $X_i$ is the indicator of an event $\mathcal{E}_i$. Then

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} Pr[\mathcal{E}_i].$$

It remains to determine $Pr[\mathcal{E}_i]$, which fortunately is easy. The homework that student $i$ receives is equally likely to be any student’s homework, so it has probability $1/n$ of being their own homework. Thus

$$Pr[\mathcal{E}_i] = Pr[\text{student } i \text{ gets their own homework}] = 1/n.$$

It follows that

$$E[X] = \sum_{i=1}^{n} \frac{1}{n} = 1.$$

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).

### 4.2 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 2.2.

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 estimator $\hat{f}$ of $f$, then show that it is unbiased, meaning that $E[\hat{f}] = f$.

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

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

Beware of a subtlety with the notation $|S \cap \mathcal{A}|$: when sampling with replacement is used, it counts duplicated samples multiple times.

#### 4.2.1 Sampling with replacement

- **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.
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 2.4.

1: function PollWithReplacement(list C, int k, set A)
2: 
3: S ← SampleWithReplacement(C, k)
4: 
5: ˆf ← 1/k · (total number of customers in S who are also in A)
6: 
7: return ˆf
8: end function

9: function PollWithoutReplacement(list C, int k, set A)
10: 
11: S ← SampleWithoutReplacement(C, k)
12: 
13: ˆf ← 1/k · (total number of customers in S who are also in A)
14: 
15: return ˆf
16: end function

17: function BernoulliPolling(list C, int k, set A)
18: 
19: S ← BernoulliSample(C, k/m)
20: 
21: ˆf ← 1/k · (total number of customers in S who are also in A)
22: 
23: return ˆf
24: end function

Let \( X_i \) be the indicator of the event that the \( i \)th sampled person likes avocados. The estimator can be written

\[
\hat{f} = \frac{1}{k} \sum_{i=1}^{k} X_i.
\]  

(4.2.1)

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

Claim 4.2.1. \( \hat{f} \) is unbiased.

Proof. Since the \( i \)th sampled person is chosen uniformly at random, the probability that they like avocados is exactly \( f \). Thus, by linearity of expectation,

\[
E[\hat{f}] = \frac{1}{k} \sum_{i=1}^{k} \Pr[i \text{th sample likes avocados}] = \frac{1}{k} \sum_{i=1}^{k} f = f.
\]  

(4.2.2)

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

4.2.2 Sampling without replacement

- 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 \( X_i \) be the indicator of the event that the \( i \)th sampled person likes avocados. The estimator, which is again the sample mean, is

\[
\hat{f} = \frac{1}{k} \sum_{i=1}^{k} X_i.
\]  

(4.2.2)

Claim 4.2.2. \( \hat{f} \) is unbiased.
Proof. Recall from Algorithm 2.4 that the sample $S$ comes from a random $k$-permutation. By Claim 2.1.2 (a), each sample is a uniformly random customer, and therefore

$$E[X_i] = \frac{|A|}{|C|} = \frac{fm}{m} = f.$$  

Note that $X_1, \ldots, X_m$ are not independent. Nevertheless, using linearity of expectation,

$$E[\hat{f}] = \frac{1}{k} \sum_{i=1}^{k} E[X_i] = \frac{1}{k} \cdot k \cdot f = f. \quad \square$$

Remark 4.2.3. The number of sampled people who like avocados is known to have the hypergeometric distribution. The expectation of this distribution can be used to prove Claim 4.2.2.

4.2.3 Bernoulli sampling

- **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.

Unlike the previous two approaches, we now let $X_i$ be the indicator of the event that the $i^{th}$ customer is sampled. The estimator is

$$\hat{f} = \frac{1}{k} \sum_{i \in A} X_i.$$  

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 4.2.4.** $\hat{f}$ is unbiased.

**Proof.** Our sampling process has $E[X_i] = p$. Thus, using linearity of expectation and $|A| = fm$,

$$E[\hat{f}] = \frac{1}{pm} \sum_{i \in A} E[X_i] = \frac{1}{pm} \sum_{i \in A} p = \frac{1}{pm} \cdot fm \cdot p = f. \quad \square$$

4.2.4 Exercises

**Exercise 4.1.** 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, in Section 4.2.1 above, each $Z_i$ was a uniformly and independently chosen customer, and $g$ was the predicate that indicates whether that customer is in the set $A$.

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

©Nicholas Harvey 2021. All rights reserved.
Exercise 4.2. This exercise considers an alternative analysis for sampling without replacement. Let $X_i$ be the indicator of the event that the $i$th customer is sampled. The estimator is

$$\hat{f} = \frac{1}{k} \sum_{i \in A} X_i.$$  

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

**Part I.** Show that $E[X_i] = k/m$.

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

4.3 QuickSelect

Algorithm 4.2 Pseudocode for Randomized QuickSelect. Returns the $k$th largest element of $A$, assuming $1 \leq k \leq |A|$. Assume that the elements in $A$ are distinct.

1: function QuickSelect(array $A$, int $k$)
2: if Length($A$) = 1 then return $A[1]$ 
3: Select an element $p \in A$ uniformly at random $\triangleright$ The pivot element 
4: Construct the sets $Left = \{x \in A : x < p\}$ and $Right = \{x \in A : x > p\}$ 
5: Let $r \leftarrow |Left| + 1$ $\triangleright$ The pivot element is the $r$th largest in $A$ 
6: if $k = r$ then return $p$ $\triangleright$ $k$th largest is the pivot 
7: else if $r > k$ then return QuickSelect($Left$, $k$) $\triangleright$ $k$th largest in $A$ is in $Left$ 
8: else return QuickSelect($Right$, $k - r$) $\triangleright$ $k$th largest in $A$ is in $Right$ 
9: end function

Theorem 4.3.1. 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.

The key technical step of this theorem is to understand the size of the child problem. This is accomplished by the following lemma. (If there is no child subproblem, we say it has size 0.)

Lemma 4.3.2. The child subproblem’s array has expected size at most $7n/8$.

We now prove Theorem 4.3.1 by induction. The base case is trivial: if $n = 1$, no comparisons are performed. So instead consider a subproblem with array $A$ of size $n > 1$, and some value $k \in [n]$. The comparisons are performed in two places:

- by line 4, which compares every other element to the pivot exactly once, which is $n - 1$ comparisons,
- and by the recursive descendents.

This leads to the following analysis.

$$\begin{align*}
E[(\# \text{ comparisons for QuickSelect}(A, k))] &= E[(n - 1) + (\# \text{ comparisons in descendents})] \\
&= (n - 1) + E[(\# \text{ comparisons in descendents})] \quad \text{(by linearity of expectation)}
\end{align*}$$
< n + E [8 \cdot (\text{size of child subproblem})] \quad \text{(by induction)}
< n + 8 \cdot (7n/8) \quad \text{(by Lemma 4.3.2)}
= 8n

This completes the induction, which proves Theorem 4.3.1. It remains to prove the lemma.

**Proof of Lemma 4.3.2.** 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.

Intuitively, the algorithm makes good progress if the pivot splits the array roughly in half. To make this precise, define the event

\[ \mathcal{E} = \ \left[ \left\lceil \frac{n}{4} \right\rceil \leq r \leq \left\lceil \frac{3n}{4} \right\rceil \right] \]

This event is quite likely: since \( r \) is uniform on \([n]\), we have

\[
\Pr [\mathcal{E}] = \frac{\left\lceil \frac{3n}{4} \right\rceil - \left\lceil \frac{n}{4} \right\rceil + 1}{n} \geq 1/2. \tag{4.3.1}
\]

If \( \mathcal{E} \) occurs then both \( \text{Left} \) and \( \text{Right} \) have size at most \( 3n/4 \), so the child must have size at most \( 3n/4 \) (or, if \( k = r \), the algorithm terminates). On the other hand, if \( \mathcal{E} \) does not occur, then we simply use that the child has size less than \( n \), since it is smaller than the parent. These bounds are combined using the Law of Total Expectation (Fact A.3.14).

\[
\mathbb{E} [\text{child size}] = \mathbb{E} [\text{child size} | \mathcal{E}] \cdot \Pr [\mathcal{E}] + \mathbb{E} [\text{child size} | \overline{\mathcal{E}}] \cdot \Pr [\overline{\mathcal{E}}]
\leq (3n/4) \cdot \Pr [\mathcal{E}] + n \cdot \Pr [\overline{\mathcal{E}}]
\leq (3n/4) \cdot (1/2) + n \cdot (1/2) \quad \text{(by (4.3.1))}
= 7n/8 \quad \square
\]

**Exercises**

** Exercise 4.3. ** Improve Lemma 4.3.2 to show that the child subproblem’s array has expected size at most \( 3n/4 \).

** Exercise 4.4. ** Let us consider QUICKSORT, shown in Algorithm 4.3. In this question we analyze it by applying linearity of expectation in a manner similar to Theorem 4.3.1.

Let \( f(n) \) denote the expected number of comparisons performed by 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.

\[
f(n) = n - 1 + \mathbb{E} [f(\text{Length(Left))} + f(\text{Length(Right))}] \\
= n - 1 + \sum_{i=0}^{n-1} \Pr [\text{Length(Left)} = i] \cdot (f(i) + f(n - 1 - i)) \\
= n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (f(i) + f(n - 1 - i)).
\]
As in Theorem 4.3.1, we can analyze it 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.

### 4.4 QuickSort

The techniques that we have discussed are effective for more substantial problems too. Let’s show how they can be used to analyze randomized QuickSort, which you have presumably seen in an introductory algorithms class.

**Algorithm 4.3** Pseudocode for Randomized QuickSort. Assume that the elements in $A$ are distinct.

```plaintext
1: function QUICKSORT(set $A$)
2:   if Length($A$) = 0 then return $A$
3:   Select an element $p \in A$ uniformly at random \(\triangleright \) The pivot element
4:   Construct the sets $Left = \{ x \in A : x < p \}$ and $Right = \{ x \in A : x > p \}$
5:   return the concatenation $[\text{QUICKSORT}(Left), p, \text{QUICKSORT}(Right)]$
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 the pivot element $p$ 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. The corresponding indicator random variable is

$$X_{i,j} = \begin{cases} 
1 & \text{(if } a_i \text{ and } a_j \text{ are compared)} \\
0 & \text{(otherwise)}
\end{cases}$$

Then, using Fact A.3.13, we have

$$X = \sum_{i<j} X_{i,j}$$

and

$$E[X] = \sum_{i<j} E[X_{i,j}] = \sum_{i<j} \Pr[\mathcal{E}_{i,j}] .$$

The heart of the analysis is to figure out these probabilities by subtle reasoning. Fix $i < j$ and let $R = \{a_i, \ldots, a_j\}$. (Remember, $R$ 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.
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.
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. \hfill \qed

Claim 4.4.3.

\[ \Pr[a_i \text{ or } a_j \text{ is the first pivot selected from } R] = \frac{2}{|R|} = \frac{2}{j - i + 1}. \]

Proof. Until any pivot is selected from $R$, these elements must all appear together in the same subproblems. Consider the first subproblem $A$ in which a pivot is chosen from $R$. Let $X$ denote the randomly chosen pivot element. Then $X$ is uniformly random on $A$, but we have conditioned on the event $X \in R$, so $X$ is uniform on $R$ (by Claim 1.1.1). So the probability that $a_i$ or $a_j$ is chosen is $2/|R|$. \hfill \qed

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

\[
\mathbb{E}[X] = \sum_{i<j} \Pr[E_{i,j}]
\]

(\by Fact A.3.13)

\[
= \sum_{i=1}^{n-1} \left( \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \right)
\]

(by Claim 4.4.3)

\[
= 2 \sum_{i=1}^{n-1} \left( \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n - i + 1} \right)
\]

\[
\leq 2 \sum_{i=1}^{n-1} \left( \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right)
\]

\[
\leq 2 \sum_{i=1}^{n-1} \ln(n)
\]

\[
< 2n \ln(n)
\]

In the penultimate inequality, we have bounded the harmonic sum using Fact A.2.1.
Chapter 5

Balls and Bins

A lot of problems in computer science boil down to analyzing a random process where balls are thrown into bins. Two canonical examples are hash tables, and load balancing in a distributed system.

**Example 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 are interested to study properties of the load distribution on the servers.

**Example 2.** Suppose that there are $m$ items to be inserted into a hash table that has $n$ locations. 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 11, but for now let us briefly discuss what is meant by a purely random function. One viewpoint is to list all the functions from the domain to $[n]$, then to pick one of these functions uniformly at random. Another viewpoint is: 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.

5.1 Unique identifiers

A **UUID** is a 128-bit identifier widely used in filesystems, databases, distributed systems, etc. There are different formats, but version 4 just uses 122 random bits.

Imagine that you work at a high-tech company called Zoodle that manufactures vegetable processing equipment. Your customers submit various files to you, such as purchase orders, recipes, luscious vegetable photographs, etc. Each such file is assigned a random UUID then stored in Zoodle’s storage system.

A **collision** occurs if two files are randomly assigned the same UUID. Let $n = 2^{122}$ be the number of UUIDs. Suppose that $m$ files are submitted. How small must $m$ be in order for there to be less than one collision, in expectation?

Let $X$ be the number of pairs of files with colliding UUIDs. We can decompose this into a sum of indicator random variables as $X = \sum_{1 \leq i < j \leq m} X_{i,j}$, where

$$X_{i,j} = \begin{cases} 1 & \text{(if file } i \text{ and file } j \text{ collide)} \\ 0 & \text{(otherwise).} \end{cases}$$
By linearity of expectation,

\[ E[X] = \sum_{i<j} E[X_{i,j}] = \sum_{i<j} \Pr[\text{file } i \text{ and file } j \text{ collide}] . \]  

(5.1.1)

What is the probability that two particular files collide? It is

\[ \Pr[\text{file } i \text{ and file } j \text{ collide}] = \sum_{k=1}^{n} \Pr[\text{file } i \text{ assigned UUID } k] \cdot \Pr[\text{file } j \text{ assigned UUID } k] \]  

(by Fact A.3.6)

\[ = \sum_{k=1}^{n} \Pr[\text{file } i \text{ assigned UUID } k] \cdot \Pr[\text{file } j \text{ assigned UUID } k] \]  

(independence)

\[ = \sum_{k=1}^{n} (1/n) \cdot (1/n) \]  

(uniform distribution)

\[ = n \cdot (1/n)^2 \]

\[ = 1/n. \]

So, we can plug into (5.1.1) and bound the binomial coefficient with Fact A.2.8 as follows.

\[ E[X] = \sum_{1 \leq i < j \leq m} (1/n) = \left( \frac{m}{2} \right) (1/n) < \frac{m^2}{2n} \]  

(5.1.2)

To summarize, if \( m \leq \sqrt{2n} \), then there is fewer than one collision in expectation. For our particular setting of UUIDs we have \( n = 2^{122} \), so we expect fewer than one collision if \( m \leq \sqrt{2} \cdot 2^{61} \), which is about 3 quintillion.

**Question 5.1.1.** If the world has 7 billion people, and each person stores 200 million files, how many unique identifiers would we need?

This same phenomenon is often described in a toy scenario in which people have random birthdays uniformly among the \( n = 365 \) days of the year. A collision corresponds to two people sharing a birthday. How many people do we need in order to have one collision in expectation? By the formula (5.1.2) above, \( m \approx \sqrt{2} \cdot 365 \approx 27 \) people would suffice. This counterintuitive answer led to this phenomenon being called the **birthday paradox**.

**Remark 5.1.2.** We have chosen the value of \( m \) in order to ensure that the expected number of collisions is less than 1. People often study the related question: how large can \( m \) be to ensure a decent probability of no collisions? Unsurprisingly, these questions have essentially the same answer.

References: (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).

### 5.2 Equal number of clients and servers

Let’s return to the load balancing setting discussed in Example 1. To start off, let’s imagine that there is an equal number of clients and servers (i.e., \( m = n \)). If we were explicitly assigning clients to servers, then we could ensure that each server gets exactly one client. But, since we are assigning them randomly, it may happen that some server gets no clients at all. Let us call these **empty servers**.
Let $X$ be the random variable giving the number of empty servers. We can decompose this into a sum of indicator random variables as $X = \sum_{i=1}^{n} X_i$ where

$$X_i = \begin{cases} 1 & \text{(if server } i \text{ is empty)} \\ 0 & \text{(otherwise)} \end{cases}$$

By linearity of expectation,

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \Pr[\text{server } i \text{ is empty}]. \quad (5.2.1)$$

It remains to analyze these probabilities.

We observe that server $i$ is empty if and only if every client fails to choose that server. Thus,

$$\Pr[\text{server } i \text{ is empty}] = \Pr[\text{every client does not choose server } i]$$

$$= \prod_{j=1}^{m} \Pr[\text{client } j \text{ does not choose server } i] \quad \text{(independence)}$$

$$= \prod_{j=1}^{m} (1 - \Pr[\text{client } j \text{ does choose server } i]) \quad \text{(complementary events)}$$

$$= \prod_{j=1}^{m} (1 - 1/n) \quad \text{(uniform distribution)}$$

$$= (1 - 1/n)^m.$$ 

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 5.1.

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

$$\Pr[\text{server } i \text{ is empty}] = (1 - 1/n)^m \leq (e^{-1/n})^m. \quad (5.2.2)$$

Plugging this into (5.2.1) and using the assumption $m = n$,

$$E[X] = \sum_{i=1}^{n} \Pr[\text{server } i \text{ is empty}] \leq \sum_{i=1}^{n} e^{-m/n} = n/e < 0.37n.$$ 

This analysis is a bit disconcerting. It says that, when randomized load balancing with an equal number of clients and servers, at most 37% of the servers will be unused. That seems wasteful — perhaps our upper bound is loose?
The function $e^x$ (in blue) is convex, whereas $1 + x$ (in red) is a straight line that lies beneath it and touches it at the point $(0, 1)$.

**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 $n \geq 2$,

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

We can lower bound the probability that a server is empty by plugging into (5.2.2).

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

Next we can lower bound the expected number of empty servers by plugging into (5.2.1):

\[
E[X] = \sum_{i=1}^{n} \Pr[\text{server } i \text{ is empty}] \geq n/e - 1/4 \approx 0.36n.
\]

To conclude, our upper bound was not pessimistic. When randomized load balancing with an equal number of clients and servers, we expect that between 36%-37% of the servers will be unused.

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

### 5.3 Avoiding empty servers

Having empty servers seems wasteful. If there were enough clients, then presumably this wouldn’t happen. Let us now analyze the probability that there are no empty servers.

If we focus on just the $i$th server, the analysis was already performed above.

\[
\Pr[\text{server } i \text{ is empty}] \leq e^{-m/n}.
\]
Figure 5.2: Examples of randomly throwing $n$ balls into $n$ bins. The bins are sorted in decreasing order. Note that the number of empty bins is approximately $0.37n$. (a) $n = 10,000$. (b) $n = 100,000$. 
Now we introduce an important trick from probability.

**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

$$\Pr[\text{any of the events occurs}] = \Pr[\mathcal{E}_1 \cup \cdots \cup \mathcal{E}_n] \leq \sum_{i=1}^{n} \Pr[\mathcal{E}_i].$$

We apply this fact with $\mathcal{E}_i$ being the event that server $i$ is empty. We obtain

$$\Pr[\text{any server is empty}] \leq \sum_{i=1}^{n} \Pr[\text{server } i \text{ is empty}] \leq \sum_{i=1}^{n} e^{-m/n} = ne^{-m/n}.$$ 

Plugging in $m = n \ln(2n)$, this gives the bound

$$\Pr[\text{any server is empty}] \leq ne^{-m/n} = ne^{-\ln(2n)} = n(1/2n) = 1/2.$$ 

To conclude, suppose that there are $m = n \ln(2n)$ clients. Then, with probability at least 1/2, every server has at least one client.

This same phenomenon is often described in a toy scenario in which people randomly draw coupons until they have collected at least one coupon of all $n$ different types. The analysis above suggests that roughly $n \ln n$ trials are needed to get $n$ coupons. This extra $\ln n$ factor is perhaps unexpected, and has led to this phenomenon being called the the **coupon collector problem**.

**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).

**An expectation analysis.** The analysis above fixes the number of clients being added, then analyzes the probability of no empty servers:

$$\Pr[\text{no empty servers with } n \ln(2n) \text{ clients}] \geq 1/2. \quad (5.3.1)$$

A variant of this problem keeps adding clients one-by-one until there are no empty servers, then analyzes the expected number of clients required. We will show

$$\mathbb{E}[\text{number of clients added until no empty servers}] \leq n(\ln(n) + 1). \quad (5.3.2)$$

Let $X$ be the RV giving the number of clients we must add until there are no empty servers. Imagine that there are $n$ phases, and in the $i^{th}$ phase we are adding clients until there are $i$ non-empty servers. Then we can write $X = \sum_{i=1}^{n} X_i$, where $X_i$ is the number of clients added in the $i^{th}$ phase.

Let us analyze $X_i$. The $i^{th}$ phase can be viewed as a series of independent trials. In each trial, a client is randomly assigned to a server. This is deemed a success if that server is currently empty; otherwise it is a failure. Each trial during the $i^{th}$ phase has probability of success

$$p_i = \frac{\text{number of empty servers}}{\text{number of servers}} = \frac{n - i + 1}{n}. \quad (5.3.3)$$
Figure 5.3: Examples of randomly throwing \( m = n \ln(2n) \) balls into \( n \) bins. The bins are sorted in decreasing order. In this example, \( n = 10,000 \) and \( m = 99,035 \).

Since \( X_i \) is the number of trials until success during phase \( i \), it is geometric with parameter \( p_i \). Armed with this knowledge, we can analyze the number of trials over all phases.

\[
E[X] = \sum_{i=1}^{n} E[X_i] \quad \text{(linearity of expectation)}
\]

\[
= \sum_{i=1}^{n} \frac{1}{p_i} \quad \text{(by Fact A.3.18)}
\]

\[
= \sum_{i=1}^{n} \frac{n}{n-i+1} \quad \text{(by (5.3.3))}
\]

\[
= n \left( \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} + \frac{1}{1} \right)
\]

\[
\leq n \left( \ln(n) + 1 \right) \quad \text{(by Fact A.2.1)}.
\]

This proves equation (5.3.2).

**Question 5.3.1.** Above we have shown an upper bound. Is it tight? In other words, is it true that

\[
E[\text{number of clients added until no empty servers}] = \Theta(n \log n)\?
\]

**Answer.**

Yes, because Fact A.2.1 shows that the harmonic sum \( H_n \) grows like \( \Theta(n \log n) \).

**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).
It is interesting to observe that (5.3.1) and (5.3.2) 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).

Exercises

Exercise 5.1 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

$$\Pr \{ \text{collect toy } i \} \geq 1/2 \quad \forall i \in [n]. \quad (5.3.4)$$

Part II. Show that if $m = 100$ and $n = 145$ then

$$\Pr \{ \text{collect toy } i \} \leq 1/2 \quad \forall i \in [n].$$

Part III. If $m$ and $n$ are general variables, what value of $n$ ensures that (5.3.4) holds?

5.4 More questions

There are numerous other interesting questions relating to balls and bins. Let us briefly mention a few more.

Load on heaviest bin. When throwing $n$ balls into $n$ bins, we showed that about 0.37$n$ bins will be empty in expectation. This means that some bins will have more than one ball. What is the maximum number of balls in any bins? This innocuous question has a somewhat unexpected answer: the maximum number of balls is very likely to be $\Theta(\log n/\log \log n)$. We will show the upper bound in Section 5.5.

Small ratio of loads. 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 (as a function of $n$) to ensure this? Figure 5.4 (a) suggests that if $m$ is sufficiently large then the ratio will be small. After developing more tools, we will prove this in Section 9.1.

The better of two choices. Suppose each client randomly picks two servers, then connects to the server with lower load. Intuitively this should improve the ratio of the maximum load to the minimum load. Figure 5.4 (b) suggests that the improvement is substantial. 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.

Analyzing 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 $\Theta(\log \log n)$. This improves on the $\Theta(\log n/\log \log n)$ result mentioned above.

References: (Motwani and Raghavan, 1995, Theorem 3.1), (Mitzenmacher and Upfal, 2005, Chapter 14).
Figure 5.4: Examples with $m = 600,000$ balls and 10,000 bins. The bins are sorted in decreasing order. (a) The balls are thrown into a random bin. The ratio of the maximum load to the minimum load is $\approx 2.9$. (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 $\approx 1.13$. 
Keener Kwestion 5.4.1. Example 2 claimed that balls and bins techniques are useful for analyzing hash tables. Can you design a hash table that stores each item in the better of two bins?

★5.5 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.

\[
\binom{n}{i} = \frac{n(n-1) \cdots (n-i+1)}{i!} \leq \frac{n^i}{i!} \quad \forall 0 \leq i \leq n. \tag{5.5.1}
\]

Using this, we can analyze this probability as follows.

\[
\Pr \left[ X_1 \geq k \right] \leq \binom{n}{k} \left( \frac{1}{n} \right)^k \quad \text{(by Fact A.3.16)}
\]

\[
\leq \frac{n^k}{k!} \left( \frac{1}{n} \right)^k = \frac{1}{k!} \quad \text{(by (5.5.1))}
\]

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 5.5.1 (Approximate inverse factorial). For all \( x \geq 1 \),

\[
\left( \frac{\ln x}{\ln(1 + \ln x)} \right)! \leq x \leq \left( \frac{2 \ln x}{\ln(1 + \ln x)} \right)!. \tag{5.5.2}
\]

This would require a messy proof, so instead we will just give a plot in Figure 5.5.

Thus, plugging in

\[
k = \frac{2 \ln(n^2)}{\ln(1 + \ln(n^2))},
\]

and using Fact 5.5.1 with \( x = n^2 \), we get

\[
\Pr \left[ X_1 \geq k \right] \leq \frac{1}{k!} \leq \frac{1}{n^2}. \tag{5.5.2}
\]

Here we have 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.

\[
\Pr \left[ \text{any bin has } \geq k \text{ balls} \right] = \Pr \left[ X_1 \geq k \lor X_2 \geq k \lor \cdots \lor X_n \geq k \right]
\]

\[
\leq \sum_{i=1}^{n} \Pr \left[ X_i \geq k \right] \quad \text{(Fact A.3.8)}
\]

\[
\leq \sum_{i=1}^{n} \frac{1}{n^2} \quad \text{(by (5.5.2))}
\]

\[
= \frac{1}{n}.
\]
Figure 5.5: The green function, \((\frac{2 \ln x}{\ln(1+\ln x)})!\), is slightly larger than \(x\). The maroon function, \((\frac{\ln x}{\ln(1+\ln x)})!\), is slightly smaller than \(x\).

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)).
5.6 Exercises

Exercise 5.2. 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.3.15.

Exercise 5.3. Suppose $\mathcal{E}_1, \ldots, \mathcal{E}_n$ are independent events with $\Pr[\mathcal{E}_i] = \frac{1}{10n}$. What is the exact value of $\Pr[\mathcal{E}_1 \lor \cdots \lor \mathcal{E}_n]$? What bound on this quantity is provided by the union bound? How do these compare when $n = 100$?

Exercise 5.4. Suppose we throw $n$ balls into $n$ bins uniformly and independently at random. Let $Y$ be the load on the heaviest bin. Prove that $\mathbb{E}[Y] = O(\log n / \log \log n)$.

Exercise 5.5. 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 5.6. Let $X[1..n]$ be an array of real numbers, chosenly independently and uniformly from the interval $[0, 1]$. Describe and analyze an algorithm to sort $X$ with expected runtime $O(n)$.

Exercise 5.7. In Section 2.1 we have seen several algorithms for generating uniformly random permutations. Let us consider another nice algorithm from a paper of Rao in 1961.

Algorithm 5.1 Rao’s method to generate a uniformly random $n$-permutation of $C$. Let $n = |C|$.

1: function RaoPerm(list $C$)
2:   Set $m \leftarrow \ldots$ \hspace{1cm} $\triangleright$ This can be a function of $n$
3:   Let $L[1..m]$ be an array of empty lists
4:   for $c \in C$
5:      Let $X$ be a RV that is uniform on $[m]$
6:      Add $c$ to $L[X]$
7:   for $i = 1, \ldots, m$
8:      if $|L[i]| > 1$ then RaoPerm($L[i]$)
9:   return the concatenation $L[1], L[2], \ldots, L[m]$
10: end function

Choose the value of $m$ appropriately and prove that this algorithm runs in $O(n)$ time.

Exercise 5.8 Network collisions. You are hired as a junior engineer at Risco, a company dedicated to using randomness in computer networking. They want to develop a new wireless protocol that works as follows.

Suppose the wireless network has $n$ machines. Time is divided into discrete slots. In any slot, any
machine can try to broadcast a packet. The broadcast will succeed if exactly one machine tried to broadcast in that slot; however, the machine cannot determine whether the broadcast was successful. If multiple machines broadcast in the same slot, the packets are garbled and cannot be transmitted.

**Part I.** Suppose that only $m$ machines wish to broadcast ($m \leq n$), and the value of $m$ is known to all machines. Devise a (very simple) randomized protocol that ensures that, in every time slot, the probability that some broadcast succeeds is at least $1/10$. The protocol must perform no extra communication other than the attempted broadcasts.

**Part II.** In networking, it is hard to know the exact number of nodes who want to communicate. Again, let $m > 0$ be the number of machines wish to broadcast. Suppose that the machines do not know $m$, but instead know an estimate $\hat{m}$ satisfying

$$m \leq \hat{m} \leq 2m.$$ 

(All machines know the same value $\hat{m}$.) Devise a randomized protocol that ensures that, in every time slot, the probability that some broadcast succeeds is at least $1/10$. The protocol must perform no extra communication other than the attempted broadcasts.

**Part III.** Let us make the task more challenging by removing our assumption that the machines know this estimate $\hat{m}$ on $m$. Instead, they have:

- a (very weak) estimate $\hat{n}$ for $n$, satisfying $n \leq \hat{n} \leq 100n^2$.
- an unlimited number of shared random bits. That is, in every time slot $t$, each machine can generate by itself some random variable $X_t$. All machines will obtain exactly the same value for $X_t$ (because their randomness is shared).
- They can still locally generate non-shared random bits too.

Devise a randomized protocol that ensures that, in every time slot, the probability that some broadcast succeeds is $\Omega(1/\log n)$. The protocol must perform no extra communication other than the attempted broadcasts.
Chapter 6

Skip Lists

One of the most important data structures is the balanced tree. As discussed in introductory courses, 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.

A trouble with all of these usual designs is that they are all somewhat intricate to implement and to analyze. Programmers would be advised to use a highly vetted implementation from a standard library.

A Skip List is an amazingly simple randomized data structure that supports the same operations as balanced trees, and with the same runtime (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 the randomized data structure, let us first discuss a deterministic version that has most of the key ideas, but also has a fatal flaw.

A perfect Skip List is comprised of a collection of sorted, doubly-linked lists $L_1, L_2, \ldots$. When INSERT($q$) is called, a node is created with $q$ as its key. This node will be added to $L_1$ and possibly to more of these lists. The lists are nested, meaning that $L_1 \supseteq L_2 \supseteq L_3 \supseteq \cdots$. More specifically:

- $L_1$ contains all of the nodes;
- $L_2$ contains every second node from $L_1$;
- $L_3$ contains every second node from $L_2$, so every fourth node from $L_1$;
- $L_4$ contains every second node from $L_3$, so every eighth node from $L_1$;
- etc.

The list $L_{\lceil \log n \rceil + 1}$ might contain only a single node, and $L_{\lceil \log n \rceil + 2}$ 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
(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
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.**

\[
|\mathcal{O}_{u}(v)\rangle = \cdots + \left( 1/\varphi \right) u + u \mathcal{Z} = \cdots + |\mathcal{E}| \mathcal{Z} + |\mathcal{E}| \mathcal{Z} + |\mathcal{E}| \mathcal{Z}
\]

Recall that each list is a doubly-linked list, so the number of pointers is \( 2 |L| + 2 |L| + 2 |L| + \cdots \). The fatal flaw of a perfect Skip List 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 \( L_1 \). The new eighth node is \( T \), which must be added to \( L_2, L_3 \) and \( L_4 \). The node \( U \) should be removed from \( L_2 \). In the worst case, \( \Omega(n) \) nodes may need their pointers adjusted. Due to this flaw, a perfect Skip List is no more appealing than a sorted array.

### 6.2 A randomized Skip List

The key insight of Skip Lists is that we can use randomization to relax the rigid structure of perfect Skip Lists. The key change is as follows:

- \( L_1 \) contains all of the nodes;
- \( L_2 \) contains every node from \( L_1 \) with probability \( 1/2 \);
- \( L_3 \) contains every node from \( L_2 \) with probability \( 1/2 \);
- \( L_4 \) contains every node from \( L_3 \) with probability \( 1/2 \);
- etc.

An example of this is shown in Figure 6.2.

The intuition is that \( L_{\lg(n)} \) 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 \( L_{\lg(n)}-1 \) contains twice as many nodes, continuing the search in this list will reduce the range to roughly \( n/4 \) nodes. Continuing the search in \( L_{\lg(n)}-2 \) reduce the range to \( n/8 \) nodes, etc.

There is a slick way to implement 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 \( \text{Level}(v) \), which is a geometric RV with parameter \( 1/2 \). Recalling our facts about geometric RVs, for all \( k \geq 1 \) we have

\[
\Pr[\text{Level}(i) = k] = 2^{-k} \quad \text{(by (A.3.5))}
\]

\[
\Pr[\text{Level}(i) \geq k] = 2^{-k+1} \quad \text{(by Claim A.3.17)}
\]

For every \( k \geq 1 \), the list \( L_k \) contains, in sorted order, all nodes whose level is at least \( k \). For example, every node belongs to \( L_1 \) since every node \( v \) has \( \text{Level}(v) \geq 1 \). We start by observing some basic properties. Let \( |L_k| \) denote the size of the list \( L_k \), excluding the header node.

**Claim 6.2.1 (Expected level size).** \( \mathbb{E}[|L_k|] = n 2^{-k+1} \).
Proof. By linearity of expectation,

$$E[|L_k|] = \sum_{\text{node } i} \Pr[i \in L_k] = \sum_{i=1}^{n} \Pr[\text{Level}(i) \geq k] = n2^{-k+1}.$$  

With perfect Skip Lists, we knew with certainly that $L_{\lg(n)+2}$ was empty. With randomized Skip Lists, a similar property holds in expectation. Define the threshold

$$\tau := 1 + \lg(n).$$

Then the level-$\tau$ list is almost empty, because $E[|L_\tau|] = 1$.

Notice the exponential dependence on $k$ in Claim 6.2.1: lists are exponentially shrinking as they get higher. Since the size of the lists is shrinking so rapidly, we can actually show that the total size of all lists above level $\tau$ is also tiny.

Claim 6.2.2 (Total size of levels above threshold). $\sum_{k>\tau} E[|L_k|] \leq 1$.

Proof. Because the levels shrink exponentially, we can use a geometric sum:

$$\sum_{k>\tau} E[|L_k|] = \sum_{k>\tau} n2^{-k+1}$$  

(by Claim 6.2.1)

$$= n2^{-\tau}(1 + \frac{1}{2} + \frac{1}{4} + \cdots)$$

$= n2^{-\tau+1}$  

(geometric sum, Fact A.2.2)

$= 1$  

(by choice of $\tau$). □

6.3 Search

In Skip Lists, the key operation is SEARCH. Once we have seen how to perform searches, the other operations are easy.
Algorithm 6.1 The SkipList search algorithm. We are searching for a node with key equal to $v$.

1: function Search(key $v$)
2: Let $k$ be the height of the header node
3: The current node is the header node
4: repeat
5: Search through list $L_k$ for the node whose key is closest to $v$. Crucially, this search doesn’t start from the leftmost node in the list, it starts from the current node.
6: $k \leftarrow k - 1$
7: until $k = 0$
8: end function

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

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 $d$, 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.

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

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

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 $E[X_k] < 2$.

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 $i$.

The first condition is straightforward: it happens when node $i$’s level is at least $k$. 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. So the second condition is that $\text{Level}(j) \leq k$ for all $i < j \leq d$.

Fortunately the nodes’ levels are independent, so we can just multiply all these probabilities.

$$\Pr[X_{i,k} = 1] = \Pr[\text{node } i \text{ in level } k] \cdot \Pr[\text{cannot jump over node } i]$$

$$= \Pr[\text{Level}(i) \geq k] \cdot \prod_{i < j \leq d} \Pr[\text{Level}(j) \leq k]$$

$$= 2^{-k+1} \cdot \prod_{i < j \leq d} (1 - 2^{-k})$$

$$= 2^{-k+1} \cdot (1 - 2^{-k})^{d-i}.$$ (6.3.2)
This is an exact formula for the probability that \texttt{SEARCH}(d) traverses node \(i\) at level \(k\). The formula is admittedly not very easy to understand. The first term is exponentially decreasing in \(k\), which makes sense: a given node is unlikely to belong to higher levels. The second term 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 6.3.

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

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

\textbf{Proof of Theorem 6.3.1.} We will separately analyze the lower levels (which are probably not empty) and higher levels (which are nearly empty).

\textbf{Levels below} \(\tau\). For any level \(k \leq \tau\), Lemma 6.3.2 implies \(E[X_k] < 2\). Summing over all lower levels gives \(\sum_{1 \leq k \leq \tau} E[X_k] < 2\tau\).

\textbf{Levels above} \(\tau\). For all levels \(k > \tau\), we will use a silly bound: the number of nodes traversed by \texttt{SEARCH} at level \(k\) is no more than the total size of level \(k\). In symbols, \(X_k \leq |L_k|\). So
\[
\sum_{k > \tau} E[X_k] \leq \sum_{k > \tau} E[|L_k|] \leq 1 \quad (\text{by Claim 6.2.2}).
\]

By linearity of expectation, we combine the “below \(\tau\)” and “above \(\tau\)” analyses to get
\[
E[\text{total \# nodes traversed}] = \sum_{1 \leq k \leq \tau} E[X_k] + \sum_{k > \tau} E[X_k] < 2\tau + 1 = O(\log n). \quad \Box
\]

Now that we’ve analyzed \texttt{SEARCH}, we can derive several interesting results as a consequence.

\textbf{Corollary 6.3.3.} The expected number of non-empty lists in a Skip List is \(O(\log n)\).

\textbf{Proof.} Suppose we search for the \textit{last} node in a Skip List. The \texttt{SEARCH} algorithm traverses at least one node at every level. So the length of the search path is at least the number of levels. Taking the expectation,
\[
E[\text{number of levels}] \leq E[\text{length of search path}] = O(\log n). \quad \Box
\]

\textbf{Keener Kwestion 6.3.4.} The most direct way to analyze the expected number of non-empty lists is by calculating \(E[\max_{1 \leq i \leq n} \text{Level}(i)]\). Can you calculate this expectation?
Figure 6.3: Plots of the function $2^{-k+1} \cdot (1 - 2^{-k})^{d-i}$ for $d = 40$, as a function of $i$ and with various values of $k$. The maroon curve plots this function for $k = 1$. At level 1, SEARCH is very likely to traverse nodes 39 and 40, but very unlikely to traverse any node before 35. The blue curve plots this function for $k = 2$. At level 2, SEARCH is somewhat likely to traverse nodes 35-40, but unlikely to traverse any node before 30. The green curve plots this function for $k = 3$. The light blue curve plots this function for $k = 4$. The purple curve plots this function for $k = 5$. At level 5, SEARCH has a nearly constant probability of traversing all nodes, but the overall probability is quite low.
6.4 Delete

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

Algorithm 6.2 The Skip List deletion algorithm.

1: function DELETEGIVENKEY(key q)
2: Let v be the node found by SEARCH(q)
3: DELETEGIVENODE(v)
4: end function
5: function DELETEGIVENODE(node v)
6: for i = 1, ..., Level(v) do
7: Remove v from L_i.
8: If L_i is now empty, remove it from the header node.
9: end for
10: end function

Question 6.4.1. What is the runtime of DELETEGIVENODE?

Answer. Ignoring maintenance of the header node, the runtime is $O(\text{Level}(v))$. Recalling that \text{Level}(v) is a geometric random variable with $p = 1/2$, the expected runtime is $O(1)$. 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.

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.

1: function SLOWINSERT(key q)
2: Create a new node $v$ with key $q$.
3: Generate Level($v$), a geometric RV with parameter $p = 1/2$.
4: for i = 1, ..., Level($v$) do
5: If $L_i$ does not exist, add it to the header node.
6: Insert $v$ into the sorted list $L_i$ using linear search.
7: end for
8: end function

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

1: function Insert(key q)
2:     Create a new node v with key q. Let Level(v) be a fresh geometric random variable.
3:     If Level(v) exceeds the height of the Skip List, increase the height of the header node to Level(v)
4:     Let k be the height of the header node
5:     The current node is the header node
6:     repeat
7:         Search through list $L_k$ for the node whose key is closest to v. Crucially, this search doesn’t
             start from the leftmost node in the list, it starts from the current node.
8:         If $k \leq \text{Level}(v)$, then insert v into $L_k$ immediately after the current node.
9:         $k \leftarrow k - 1$
10:    until $k = 0$
11: 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. \qed

6.6 Exercises

Exercise 6.1. Consider a Skip List with n nodes (i.e., $|L_1| = n$). Prove that the total amount of space used is $O(n)$ in expectation.

Exercise 6.2. Let $H$ be the number of non-empty levels in a Skip List with n nodes. Use a union bound to prove that $\Pr[H \geq 100 \log n] \leq O(1/n^{99})$.

Exercise 6.3. 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.)

Hint: There are two natural ways to do this. One is slightly messy, and one is very simple.
Part II

Concentration
Chapter 7

Markov’s Inequality

7.1 Introduction

7.1.1 A tempting gamble

Suppose I offer you to play a gambling game with me. In this game there are only two outcomes: either you win some money, or you lose some money. Your winnings are described by a random variable $X$. To make the game very tempting to you, I tell you that $E[X] = 100$, i.e., you expect to win $100! Should you play this game? Your expected winnings are positive, which suggests that you should.

Since I’m rather devious, I’ve arranged the outcomes of the game as follows.

$$Pr[X = -1000] = 0.99999 \quad \text{and} \quad Pr[X = 109,999,000] = 0.00001.$$  

This has $E[X] = 100$, as you can check.

Should you play this game? You are very likely to lose, which suggests that you shouldn’t.

Concretely, 99.999% of the time you must pay me $1000, which is a nice outcome for me. On the other hand, 0.001% of the time I must pay you $109,999,000, which is a wonderful outcome for you. Of course I do not actually have that much money, so at that point I would have to turn to a life of crime, or some other drastic action. But realistically, my chance of dying on any given day probably already exceeds 0.00001, so I might not be bothered by this tiny probability of losing the gamble.

The moral of the story is:

The expectation of a random variable does not tell you everything.

7.1.2 Ludicrous load balancing

Let’s give another illustration of this phenomenon by considering the load balancing scenario of Section 5.2 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.3.13, we have

$$E[X_i] = \sum_{j=1}^{n} Pr[\text{client } j \text{ assigned to server } i] = \sum_{j=1}^{n} \frac{1}{n} = 1.$$
However 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 completely dependent.

For example, we could consider the following ludicrous load balancing scheme: pick a random server $S$, then assign all clients to server $S$. It is still true that the expected load on server $i$ is $E[X_i] = 1$.

Of course the ludicrous scheme feels much less balanced than the independent scheme. We can formalize this feeling by considering the probability that $X_i$ is large. With the ludicrous scheme we have

$$Pr[X_i = n] = Pr[\text{chosen server } S \text{ is server } i] = \frac{1}{n}.$$ 

In contrast, with the independent scheme we have

$$Pr[X_i = n] = \prod_{j=1}^{n} Pr[\text{client } j \text{ chooses server } i] = \frac{1}{n^n}.$$ 

To summarize, the ludicrous scheme is decidedly not balanced but it looks fine if we consider only the expectation. The underlying moral is again:

The expectation of a random variable does not tell you everything.

**Question 7.1.1.** Compare the maximum load on any server under the independent and ludicrous schemes.

**Answer.**

In the independent scheme we saw in Section 5.5 that maximum load is $O(\log n / \log \log n)$. In contrast, with the ludicrous scheme the maximum load is definitely $n$.

**Keener Kwestion 7.1.2.** 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 $O(1)$?

### 7.1.3 When does expectation tell us about probabilities?

In order to cope with these sorts of issues, we need a better understanding of how the expectation of a random variable relates to the probabilities of its outcomes. Concretely, we’d like to know: can we use the expectation to guarantee that a bad outcome is unlikely?

**Example 3.** Consider the gambling scenario. The expectation was positive, which seems good for you. However the bad outcome, in which you lose money, had probability 0.99999.

**Example 4.** Consider the ludicrous load balancing scheme again. The expected load on server $i$ is 1, which seems good. The bad outcome is having all clients assigned to server $i$. This could happen, but only with probability $1/n$, which is very small.

Why is there such a difference in the probability of the bad outcomes? In the first example it is close to 1, but in the second example it is close to 0. The answer lies in the following principle:

The expectation of a **non-negative** random variable tells us quite a lot.

We will make this principle more precise shortly. For now let us observe that the principle applies to the load balancing scenario because the load on server $i$ is non-negative.
7.2 Markov’s inequality

7.2.1 Intuition

Imagine we have a non-negative RV $Y$ with $E[Y] = 1$. Let $p = \Pr[Y \geq 100]$. How large might $p$ be? This problem is easier if we think about it in reverse. Suppose that we have fixed $p = \Pr[Y \geq 100]$. How small might $E[Y]$ be?

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 $E[Y]$ as small as possible, it’s clear that we should move all the probability mass as far as possible to the left. Concretely, we should put $p$ units on 100 and $1 - p$ units on 0. Thus, the smallest possible value for $E[Y]$ is $(1 - p) \cdot 0 + p \cdot 100 = 100p$. We conclude that

$$E[Y] \geq 100p,$$

or equivalently

$$\Pr[Y \geq 100] \leq \frac{E[Y]}{100}.$$

This argument illustrates the ideas of Markov’s inequality.

7.2.2 The formal version

**Fact A.3.20** (Markov’s Inequality). Let $Y$ be a random variable that only takes non-negative values. Then, for all $a > 0$,

$$\Pr[Y \geq a] \leq \frac{E[Y]}{a}.$$

Note that if $a \leq E[Y]$ then the right-hand side of the inequality is at least 1, and so the statement is not giving a useful bound on the probability. (All probabilities are at most 1!) So Markov’s inequality is only useful when $a > E[Y]$.

Often when we use Markov’s inequality we want the right-hand side to be some constant, like $1/b$. We can reformulate Markov’s inequality in this useful form.

**Fact A.3.21.** Let $Y$ be a random variable that only takes nonnegative values. Then, for all $b > 0$,

$$\Pr[Y \geq b \cdot E[Y]] \leq \frac{1}{b}.$$

**Proof.** Simply apply Fact A.3.20 with $a = b \cdot E[Y]$.  

We say that Markov’s inequality bounds the right tail of $Y$, i.e., the probability that $Y$ is much greater than its mean.
7.2.3 Example 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 $2n \ln n$. That is a somewhat weak statement because it only mentions the expectation. Could the runtime actually be much worse?

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.3.20, we get

$$\Pr \left[ \text{number of comparisons} \geq 200n \ln n \right] = \Pr \left[ Y \geq 200n \ln n \right] \leq \frac{E[Y]}{200n \ln n} \leq \frac{2n \ln n}{200n \ln n} = 0.01.$$  

So we can be relieved that there is only a 1% probability of the algorithm making $200n \ln(n)$ comparisons.

7.2.4 Proof of Markov’s Inequality

**Proof of Fact A.3.20.** By definition of expectation,

$$E[Y] = \sum_{0 \leq i < a} i \cdot \Pr[Y = i] + \sum_{i \geq a} i \cdot \Pr[Y = i] \geq \sum_{0 \leq i < a} 0 \cdot \Pr[Y = i] + \sum_{i \geq a} a \cdot \Pr[Y = i] = a \cdot \Pr[Y \geq a].$$

Rearranging, we get

$$\Pr[Y \geq a] \leq \frac{E[Y]}{a},$$

which is the claimed result. $\square$

**Question 7.2.1.** Does this proof require that $Y$ takes only integer values?

**Keener Kwestion 7.2.2.** Prove Markov’s inequality using the Law of Total Expectation (Fact A.3.14).

7.2.5 Example 2: load balancing revisited

Let us revisit the intuitive argument of Section 7.2.1. Here $Y$ is the number of clients assigned to your machine. We know that $E[Y] = 1$.

We can analyze the probability that the server has too many clients. using Markov’s inequality. Applying Fact A.3.20 with $a = 1000$, we get

$$\Pr \left[ \text{(# clients on your server) } \geq 1000 \right] = \Pr \left[ Y \geq 1000 \right] \leq \frac{E[Y]}{1000} = 0.001.$$  

This matches our earlier intuitive calculation.

Now let’s try pushing the Markov inequality to its limits! What is the probability that your server has more than $n$ clients? Applying Fact A.3.20 with $a = n + 1$, we get

$$\Pr \left[ \text{(# clients on your server) } \geq n + 1 \right] = \Pr \left[ Y \geq n + 1 \right] \leq \frac{E[Y]}{n + 1} = \frac{1}{n + 1}.$$  

This analysis is mathematically correct, but unfortunately it is overly pessimistic. Clearly the probability that server 1 has $n+1$ clients is actually zero because there are only $n$ clients! We conclude that Markov’s inequality, while useful, still has its weaknesses.
7.3 Exercises

Exercise 7.1. 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 7.2 Tightness of Markov’s inequality. In Section 8.1 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 \( \Pr [ Y \geq a ] = \mathbb{E} [ Y ] / a \).

Exercise 7.3. This exercise is a variant of Exercise 5.2.
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

\[
\Pr \left[ \text{there exist distinct indices } i, j \text{ with } X_i = X_j \right] \leq 1/100.
\]

Exercise 7.4. Let \( H \) be the number of non-empty levels in a Skip List with \( n \) nodes. Use Markov’s inequality to prove that

\[
\Pr \left[ H \geq 100 \lg n \right] \leq 1/30,
\]
assuming that \( n \) is sufficiently large. Compare this to the results of Exercise 7.4.

Exercise 7.5 Reverse Markov inequality. Let \( Z \) be a random variable that is never larger than some fixed number \( d \). Prove that, for all \( c < d \),

\[
\Pr [ Z \leq c ] \leq \frac{d - \mathbb{E} [ Z ]}{d - c}.
\]

References: (Grimmett and Stirzaker, 2001, Theorem 7.3.5), (Lehman et al., 2018, Section 20.1.2).
Chapter 8

Concentration Bounds

8.1 Introduction to concentration

Repeating a trial multiple times is a concept that we are all extremely familiar with. This idea is pervasive in probability, algorithms, healthcare, etc. Understanding the probability of the outcomes is extremely powerful and important.

Example 5. Suppose we repeatedly flip a fair coin \( n \) times. Let \( X \) be the number of heads that we see. The RV \( X \) has the \textit{binomial distribution} \( B(n, 1/2) \) (see Section A.3.3). So we know that \( E[X] = n/2 \), by (A.3.3). What can we say about the tails of \( X \): the probability that \( X \) is much larger (or much smaller) than its expectation?

This topic is certainly discussed in introductory courses in probability and statistics. There are several standard ways to tackle this problem.

Markov inequality. Since \( X \) is a non-negative random variable, we can use the Markov inequality. For example, Fact A.3.20 implies

\[
\Pr \left[ X \geq \frac{3n}{4} \right] \leq \frac{E[X]}{3n/4} = \frac{n/2}{3n/4} = \frac{2}{3}.
\]

This bound does not depend on \( n \) at all, which makes it feels 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 \( 2/3 \).

Chebyshev’s inequality. There is another approach, called \textit{Chebyshev’s inequality}, that you may have seen in an introductory probability course. We will not discuss it in detail because it is somewhat weak and cumbersome. For general \( n \), this approach gives the bound \( \Pr \{ X \geq 3n/4 \} \leq 4/n \), which is certainly better than Markov’s inequality.

References: (Anderson et al., 2017, Theorem 9.5), (Lehman et al., 2018, Theorem 20.2.3), (Motwani and Raghavan, 1995, Theorem 3.3), (Mitzenmacher and Upfal, 2005, Theorem 3.6), Wikipedia.

Law of large numbers. This says that, as \( n \to \infty \), the fraction of heads will be 1/2. Unfortunately it says nothing concrete if \( n = 100 \), and it says nothing about the tails of \( X \).

References: (Anderson et al., 2017, Theorem 9.9 and 9.20), (Lehman et al., 2018, Corollary 20.4.2), Wikipedia.
Central limit theorem. This says that, as \( n \to \infty \), then we can translate and scale the distribution of \( X \) to make it look like a standard normal distribution. Concretely,
\[
\frac{X - n/2}{\sqrt{n/4}}
\]
has a distribution that converges to a normal distribution. This allows us to approximate \( \Pr \left[ X \geq 3n/4 \right] \) using the tails of the normal distribution.

References: (Anderson et al., 2017, Sections 4.1 and 9.11).

On the plus side, this tells us something about the tails of \( X \). On the negative side, the tails of the normal distribution have a cumbersome formula that students are not usually taught how to deal with. Furthermore, this approximation provides no accuracy guarantees, so it tells us nothing concrete if, say, \( n = 30 \). 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).

“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).


Exact calculation. Since \( X \) has the binomial distribution, we can use explicit formulas to calculate the tails exactly. Using (A.3.4), we get
\[
\Pr \left[ X \geq \frac{3n}{4} \right] = \sum_{k=3n/4}^{n} \binom{n}{k} 2^{-n}. \tag{8.1.1}
\]

In particular, for \( n = 100 \), we have
\[
\Pr \left[ X \geq 75 \right] = \sum_{k=75}^{100} \binom{100}{k} 2^{-100} \approx .00000028.
\]

Our intuition was correct that this probability is much less than \( 2/3 \)! But this approach is pretty cumbersome because I had to use Wolfram Alpha to evaluate the formula.

The ultimate tool. We want something better than these approaches. We want a tool that can:

- Tell us about the shape of \( X \)’s tails.
- Handle finite \( n \), not just \( n \to \infty \).
- Give us much tighter results than Markov.
- Avoid the cumbersome calculations of the explicit formula (8.1.1).
- Handle more general scenarios: maybe the coin is biased, maybe the bias changes over time, etc.

We will introduce two different tools that can handle these goals.
Figure 8.1: 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 $n \to \infty$. 
8.2 Multiplicative error: the Chernoff bound

Let $X_1, \ldots, X_n$ be independent random variables with $0 \leq X_i \leq 1$. Define

$$X = \sum_{i=1}^{n} X_i$$

$$\mu = E[X] = \sum_{i=1}^{n} E[X_i],$$

by linearity of expectation (Fact A.3.11).

The canonical example of this is performing $n$ independent trials, each of which succeeds with probability $p$. We can 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 $B(n, p)$. In this case, the expected number of successes is $\mu = np$.

**Theorem 8.2.1** (Chernoff Bound). Let $\delta$ satisfy $0 \leq \delta \leq 1$. Then

\[
\text{Left tail: } \Pr[X \leq (1 - \delta)\mu] \leq e^{-\delta^2 \mu/3}. \quad (8.2.1)
\]

\[
\text{Right tail: } \Pr[X \geq (1 + \delta)\mu] \leq e^{-\delta^2 \mu/3}. \quad (8.2.2)
\]

\[
\text{Combined tails: } \Pr[|X - \mu| \geq \delta\mu] \leq 2e^{-\delta^2 \mu/3}. \quad (8.2.3)
\]

**References:** (Lehman et al., 2018, Theorem 20.5.1), (Motwani and Raghavan, 1995, Section 4.1), (Mitzenmacher and Upfal, 2005, equations (4.2) and (4.5)), Wikipedia.

**What about $\delta > 1$?** You might wonder what we can do if $\delta > 1$. In this case, there is a big difference between the left tail and the right tail.

**Question 8.2.2.** Do you see why the left tail still holds even when $\delta > 1$?

**Answer.**

\[X \text{ occurs, } 0 > n\varepsilon \Rightarrow 0 < n\mu \text{ then } 0 = n\mu\]

For the right tail, if $\delta > 1$ then we need to modify the bound by changing $\delta^2$ to $\delta$. Annoying, but sometimes the world is a bit messier than we’d like. Combining the tails is trivial because the left tail is trivial.

**Theorem 8.2.3** (Chernoff for large $\delta$). For any $\delta > 1$,

\[
\text{Right tail: } \Pr[X \geq (1 + \delta)\mu] \leq \exp(-\delta\mu/3). \quad (8.2.4)
\]

\[
\text{Combined tails: } \Pr[|X - \mu| \geq \delta\mu] \leq \exp(-\delta\mu/3). \quad (8.2.5)
\]

8.2.1 Coin-flipping example

Let us try out the Chernoff bound on our coin-flipping example. There are $n$ trials. A success means seeing heads, which has probability $p = 1/2$. The expected number of heads is $\mu = np = n/2$. 
We are interested in the event \( X \geq 3n/4 \). Since \( 3n/4 = 1.5\mu \), this is the event that \( X \) exceeds its expectation by 50%. So we set \( \delta = 0.5 \) and plug into the formula.

\[
\Pr[X \geq 3n/4] = \Pr[X \geq (1 + \delta)\mu] \quad \text{(by Theorem 8.2.1)} \\
\leq \exp(-\delta^2 \mu/3) = \exp\left(- (0.5)^2 (n/2)/3 \right) = \exp(-n/24).
\]

For example, when \( n = 100 \) this evaluates to \( \approx 0.0155 \). This bound is not as small as the exact answer (\( \approx .00000028 \)), but significantly smaller than Markov’s bound (2/3).

**Question 8.2.4.** When doing \( n = 100 \) fair coin flips, what is the probability of 25 or fewer heads? What estimate does the Chernoff bound give?

**Keener Kwestion 8.2.5.** 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?

### 8.3 Additive error: the Hoeffding bound

Let \( X, X_1, \ldots, X_n \) and \( \mu \) be as in Section 8.2.

**Theorem 8.3.1** (Hoeffding Bound). For any \( t \geq 0 \),

\[
\Pr[X \geq \mu + t] \leq \exp(-2t^2/n) \\
\text{and} \quad \Pr[X \leq \mu - t] \leq \exp(-2t^2/n).
\]

Combining these

\[
\Pr[|X - \mu| \geq t] \leq 2\exp(-2t^2/n).
\]

**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.

#### 8.3.1 Coin-flipping example

Let us try the coin-flipping example again using Hoeffding’s bound. Recall that there \( n \) trials and \( \mu = n/2 \).

We are interested in the event \( X \geq 3n/4 \). Since \( 3n/4 = \mu + n/4 \), we set \( t = n/4 \) and plug into the formula.

\[
\Pr[X \geq 3n/4] = \Pr[X \geq \mu + n/4] \\
\leq \exp\left(- 2 \cdot (n/4)^2 / n \right) \quad \text{(by Theorem 8.3.1)} \\
= \exp(-n/8).
\]

For example, when \( n = 100 \) this evaluates to \( \approx 0.0000037 \). This is significantly closer to the exact answer (\( \approx .00000028 \)) than we got from the Chernoff bound in Section 8.2.1.

### 8.4 Comparison of the bounds

Let us compare the strength of the various concentration bounds that we have discussed. 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 \( n \).
Bound on \( \Pr [ X \geq 3n/4 ] \)

<table>
<thead>
<tr>
<th>Bound</th>
<th>( \Pr [ X \geq 3n/4 ] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Markov</td>
<td>( \leq 2/3 )</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>( \leq 4/n )</td>
</tr>
<tr>
<td>Chernoff</td>
<td>( \leq \exp(-n/24) )</td>
</tr>
<tr>
<td>Hoeffding</td>
<td>( \leq \exp(-n/8) )</td>
</tr>
<tr>
<td>Normal approximation</td>
<td>( \approx \sqrt{2/\pi n} \exp(-n/8) )</td>
</tr>
</tbody>
</table>

**Figure 8.2:** Bounds on the tail of a random variable \( X \) with the binomial distribution \( B(n, 1/2) \). The normal approximation is the smallest, but it is only an approximation, not a guarantee. Further discussion of the Chebyshev bound and the normal approximation can be found in (Vershynin, 2018, Section 2.1).

**Question 8.4.1.** Is exponential decay with \( n \) the correct behaviour?

**Answer.** Some detailed bounds are known. The only question is what is the rate of exponential decay. As we can see, the Chernoff bound and Hoeffding bound have similar strength. How should one decide which of these to use? Depending on the scenario, one or the other might be more convenient. Here are some considerations.

- The Chernoff bound provides a *multiplicative-type* guarantee on deviations from the mean, i.e., deviations of size \( \delta \mu \). In contrast, Hoeffding gives *additive-type* guarantee on deviations from the mean, i.e., deviations of size \( t \).

- Hoeffding’s inequality is useless unless \( t = \Omega(\sqrt{n}) \). The Chernoff bound might still give useful guarantees for very small deviations.

- The Chernoff bound has an exponent of \(-\Theta(\mu)\), which might not be useful if \( \mu \) is very small. However it could still be useful if \( \delta \gg 1 \); see Exercise 9.1, for example.

### 8.5 Median estimator

All of the concentration bounds that we have discussed above (Markov, Chernoff and Hoeffding) are used to show that a RV is likely to be close to its *mean*. In this section we will introduce a tool that focuses on the *median* instead of the mean. In a nutshell, we will show that

the median of a RV is likely to be close to the median of several samples of that RV.

**Definitions.** First let us review the definition of *median*. Suppose we have numbers \( Z_1, \ldots, Z_\ell \), where \( \ell \) is odd. The definition is easiest to understand if the numbers are distinct. Suppose \( Z_j \) satisfies:

- exactly \( [\ell/2] \) of the \( Z_i \)'s are less than \( Z_j \), and
- exactly \( [\ell/2] \) 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. Alternatively, if the \( Z_i \)'s are ordered so that \( Z_1 \leq \cdots \leq Z_n \), then the median is \( Z_{\lceil \ell/2 \rceil} \). 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 \( Y \). A value \( M \) is said to be a median if

\[ \Pr[Y \leq M] \geq 1/2, \quad \Pr[Y \geq M] \geq 1/2. \]

References: (Anderson et al., 2017, Definition 3.39).

**Question 8.5.1.** Let \( Y \) have the binomial distribution \( B(3, 1/2) \). Find a median of \( Y \).

**Answer.**

\[ \Pr[Y \leq M] \geq 1/2, \quad \Pr[Y \geq M] \geq 1/2. \]

Keener **Kuestion 8.5.2.** We are showing how RVs concentrate around their mean and also around their median. Does this imply that the median is always close to the mean? Consider the random variable \( X \) which has \( \Pr[X = i] = \frac{6}{\pi^2} i^2 \) for all integers \( i \geq 1 \). What are the median and mean of \( X \)?

**The median of samples.** Let \( Z_1, \ldots, Z_\ell \) be independent random variables. We do not assume that they are between 0 and 1. Instead, suppose that we can bound the tails of \( Z_i \) with parameters \( \alpha \geq 0 \) and \( L, R \) satisfying

\[
\begin{align*}
\Pr[Z_i < L] &\leq \frac{1}{2} - \alpha \\
\Pr[Z_i > R] &\leq \frac{1}{2} - \alpha.
\end{align*}
\]

Next, suppose that \( \ell \) is odd and let \( M \) be the (unique) median of the \( Z_i \)'s.

**Theorem 8.5.3.** The probability that \( M \) fails to lie between \( L \) and \( R \) is

\[ \Pr[M < L \text{ or } M > R] \leq 2 \exp(-2\alpha^2 \ell). \]

Exercise 8.5 discusses the proof of this theorem.

**Example: Approximate Median of an Array.** As an example, let us consider the following problem (Kleinberg and Tardos, 2006, Exercise 13.15). Let \( A \) be an array of \( n \) distinct numbers. In Section 4.3 we have seen the QuickSelect algorithm that can find the exact median of \( A \) in \( O(n) \) time. We will show that we can find an element close to the median in \( O(1) \) time.

To be precise, let us say that an element \( x \in A \) is an \( \epsilon \)-approximate-median if
• \(|\{ a \in A : a < x \}| \geq (0.5 - \epsilon)n\), and

• \(|\{ a \in A : a > x \}| \geq (0.5 - \epsilon)n\).

Such an element \(x\) must lie in the central \(2\epsilon\) fraction of the array.

**Algorithm 8.1 An approximate median algorithm.**

1: function APPROXMEDIAN(set \(A\), int \(\ell\))
2: Select \(\ell\) elements \(Z_1, \ldots, Z_\ell \in A\) uniformly and independently at random
3: Find the median \(M\) of \(Z_1, \ldots, Z_\ell\) using INSERTIONSORT
4: return \(M\)
5: end function

We will analyze the APPROXMEDIAN algorithm shown above. First, imagine that \(B\) is a sorted copy of \(A\); the algorithm doesn’t need to compute \(B\). Define \(L = B[(0.5 - \epsilon)n]\) and \(R = B[(0.5 + \epsilon)n]\). Then

\[
\begin{align*}
\Pr[Z_i < L] &\leq 0.5 - \epsilon \\
\Pr[Z_i > R] &\leq 0.5 - \epsilon
\end{align*}
\]

Define \(M\) to be the median of the \(Z_i\)'s.

Then we have

\[
\Pr[M \text{ is not an } \epsilon\text{-approximate median of } A] = \Pr[M < L \text{ or } M > R] \leq 2\exp(-2\epsilon^2\ell)
\]

Suppose we want this probability to be 0.01 and we want \(\epsilon = 0.05\).

**Question 8.5.4.** How should we choose \(\ell\)?

**Answer.**

\[
10^0 \cdot \frac{002}{\zeta} = \frac{(\frac{\epsilon^2\zeta}{(002)^{\eta_1}} - \epsilon^2\zeta)}{\zeta} = (\frac{\epsilon^2\zeta}{(002)^{\eta_1}} - \epsilon^2\zeta)
\]

If necessary, add 1 so that \(\eta\) is odd. Then the failure probability is

\[
\left[\frac{(002)^{\eta_1}002}{(002)^{\eta_1}}\right] = \beta
\]

Note that \(\ell\) 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 \(O(\ell^2) = O(1)\).
8.6 Exercises

Exercise 8.1. 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 8.6.1** (Hoeffding for averages). For any $q \geq 0$,

\[
\Pr\left[ Y \geq \mathbb{E}[Y] + q \right] \leq \exp\left(-\frac{2q^2}{n}\right)
\]

and

\[
\Pr\left[ Y \leq \mathbb{E}[Y] - q \right] \leq \exp\left(-\frac{2q^2}{n}\right).
\]

Combining these

\[
\Pr\left[ |Y - \mathbb{E}[Y]| \geq q \right] \leq 2\exp\left(-\frac{2q^2}{n}\right).
\]

Prove this theorem.

Exercise 8.2. 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

\[
\Pr\left[ |Y - \mathbb{E}[Y]| \geq \sqrt{\ln(2/q)/2n} \right] \leq q.
\]

Exercise 8.3. Let $X_1, \ldots, X_n$ be independent indicator RVs. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbb{E}[X]$. 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 8.4. 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 $\mu = \mathbb{E}[X]$. Consider the following slightly weaker form of the Hoeffding bound.

**Theorem** (Weak Hoeffding). For any $t \geq 0$,

\[
\Pr\left[ |X - \mu| \geq t \right] \leq 2\exp\left(-t^2/3n\right).
\]

Prove this theorem using the Chernoff bound.

Exercise 8.5. In this question we will prove Theorem 8.5.3. Define the events

\[
\mathcal{E} = \{ i : Z_i < L \} \geq \ell/2
\]

\[
\mathcal{F} = \{ i : Z_i > R \} \geq \ell/2
\]

**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

\[
\Pr\left[ M < L \text{ or } M > R \right] \leq 2\exp(-2\alpha^2 \ell).
\]

©Nicholas Harvey 2021. All rights reserved.
Chapter 9

Applications of Concentration

9.1 Ratio of loads in load balancing

Let us return to a random load balancing problem discussed in Section 5.4. 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 9.1.1.** 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

\[
\Pr \left[ \text{on every server the number of clients is between } 0.5\mu \text{ and } 1.5\mu \right] \geq 0.99.
\]

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:

\[
X = \sum_{i=1}^{m} X_i \quad \text{where} \quad X_i = \begin{cases} 1 & \text{(if client } i \text{ assigned to first server)} \\ 0 & \text{(otherwise)} \end{cases}
\]

Note that \( E[X_i] = 1/n \) and so \( E[X] = m/n = \mu = 12 \ln(200n) \). The Chernoff bound is a great tool to analyze the probability that the first server has too few or too many clients.

\[
\Pr[|X - \mu| \geq 0.5\mu] \leq 2 \exp(-\delta^2\mu/3) \quad \text{(by the Chernoff Bound, Theorem 8.2.1)}
\]

\[
= 2 \exp(-\mu/12) = 2 \exp(- \ln(200n)) = \frac{1}{100n}.
\]
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

\[ E_i = \text{"the number of clients on server } i \text{ is not between } 0.5 \mu \text{ and } 1.5 \mu". \]

We have argued that

\[ \Pr [E_i] \leq \frac{1}{100n} \quad \forall i \in [n]. \]

By a union bound,

\[ \Pr [E_1 \lor \cdots \lor E_n] \leq \sum_{i=1}^{n} \Pr [E_i] \leq n \cdot \frac{1}{100n} = 0.01. \] (9.1.1)

Thus, there is at most 1% probability that any server fails to have between 0.5\( \mu \) and 1.5\( \mu \) clients.

**Hoeffding won’t work.** Let’s try applying the Hoeffding bound for this problem, still assuming \( m = 12n \ln(200n) \) clients. We analyze \( \Pr [E_i] \) using the Hoeffding bound with \( t = 0.5 \mu = m/2n \).

\[
\Pr [E_i] = \Pr \left[ |X_i - \mu| \geq 0.5 \mu \right] \\
\leq 2 \exp \left( - \frac{2t^2}{m} \right) \quad \text{(the Hoeffding bound, Theorem 8.3.1)} \\
= 2 \exp \left( - \frac{m}{2n^2} \right) = 2 \exp \left( - \frac{6 \ln(200n)}{n} \right).
\]

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 \( \Pr [E_i] \) by a value close to 2, which is completely useless because all probabilities are at most 1.

**Question 9.1.2.** 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.**

\[
\begin{aligned}
&. \frac{\mu_{00I}}{1} = \left( \frac{\varepsilon^{u\mathcal{Z}}}{(u00\mathcal{Z})^{\mu_1} \varepsilon^{u\mathcal{Z}}} - \right) d\mathcal{X} \mathcal{Z} = \left( \frac{\varepsilon^{u\mathcal{Z}}}{u} - \right) d\mathcal{X} \mathcal{Z} \\
&\quad \geq [^\dagger 3]. \quad \text{by above.}\end{aligned}
\]

Next we can apply a union bound as in (1.1.6).

**Exercises**

**Exercise 9.1.** 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 \( \Pr [X \geq 1 + 6 \ln n] \leq 1/n. \)
9.2 Probability amplification, for two-sided error

Probability amplification is an idea that we have seen before. 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.
- **Two-sided error.** These algorithms can have both false positives and false negatives.

In Section 3.3 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 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.

![Figure 9.1: Probabilities of outcomes for a BPP-algorithm](image)

Let $B$ be a BPP-algorithm. Depending on its input, there are two cases for its behavior.

- **Correct Output is Yes.** Then we know that $\Pr[B \text{ outputs Yes}] \geq 2/3$.
- **Correct Output is No.** Then we know that $\Pr[B \text{ outputs Yes}] \leq 1/3$.

The situation is strongly analogous to Section 10.1.

```
Executing $B$ when Correct Output is Yes $\approx$ Flipping Fluffy’s coin
Executing $B$ when Correct Output is No $\approx$ Flipping Nick’s coin
```

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 algorithm as follows.

**Algorithm 9.1** The algorithm $A$ for amplifying success of algorithm $B$. The parameter $\ell$ determines the number of trials.

```
1: function AMPLIFY(algorithm $B$, integer $\ell$)
2:     $X \leftarrow 0$ \hfill $\triangleright$ Count of number of trials that output Yes
3:   for $i = 1, \ldots, \ell$ do
4:     Run algorithm $B$ with new independent random values
5:     if $B$ outputs Yes then $X \leftarrow X + 1$
6:   end for
7:   if $X \geq \ell/2$ then return Yes else return No
8: end function
```
**Theorem 9.2.1.** Suppose $B$ is a BPP-algorithm. Then $\text{Amplify}(B, \ell)$ satisfies

\[
\begin{align*}
\Pr[\text{false positive}] & \leq \exp(-\ell/18) \\
\Pr[\text{false negative}] & \leq \exp(-\ell/18).
\end{align*}
\]

**References:** (Motwani and Raghavan, 1995, Section 6.8), (Sipser, 2012, Lemma 10.5).

It is interesting to compare the amplification results from today and from Section 3.3.

- Theorem 3.3.1 made the strong assumption that $B$ has no false negatives, then concluded that the false positive rate decreases like $2^{-\ell}$.
- Theorem 9.2.1 allowed $B$ to have both false positives and false negatives, then concluded that both rates decrease like $e^{-\ell/18}$.

So today’s hypothesis is much weaker but its conclusion is roughly the same, if we’re willing to overlook the factor 18.

**Proof of Theorem 9.2.1.** Let us consider the case in which the Correct Output is Yes. (An analogous argument works in the other case.) We would like to show that algorithm $A$ is very likely to output Yes.

In each iteration, the probability of $B$ returning Yes is $p \geq 2/3$. So the expected number of Yes produced by $B$ 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.

\[
\Pr[X < \ell/2] = \Pr[X < (2/3)\ell - (1/6)\ell] \\
\leq \Pr\left[X \leq \mu - \frac{(1/6)\ell}{t}\right] \\
\leq \exp(-2t^2/\ell) \quad \text{(by the Hoeffding bound, Theorem 8.3.1)} \\
= \exp\left(-2\cdot(\ell/6)^2/\ell\right) \\
= \exp(-\ell/18).
\]

**Exercises**

**Exercise 9.2.** Let $A$ be a BPP-algorithm solving some problem, and whose runtime is random. Specifically, we know a constant $c$ such that $E[\text{runtime of } A] \leq cn^2$ for inputs of size $n$.

Using $A$ (and $f$), create another BPP-algorithm $\hat{A}$ solving the same problem, except that $\hat{A}$’s runtime is not random. Specifically, we always have runtime of $\hat{A} = O(n^2)$.

**Hint:** man timeout

**Exercise 9.3.** Let us view the outputs of $B$ as being 0/1 rather than No/Yes. Consider the following amplification algorithm.
Algorithm 9.2 An alternative algorithm for amplifying success of algorithm \(B\). The parameter \(\ell\) determines the number of trials.

1: function \textsc{AmplifyByMedian}(algorithm \(B\), integer \(\ell\))
2:  for \(i = 1, \ldots, \ell\) do
3:    Let \(Z_i\) be the output of \(B\) with new independent random values
4:  end for
5:  return Median\((Z_1, \ldots, Z_\ell)\)
6: end function

Use Theorem 8.5.3 to prove the following theorem.

Theorem. Suppose \(B\) is a BPP-algorithm. Then \textsc{AmplifyByMedian}(\(B\), \(\ell\)) satisfies

\[
\Pr[\text{false positive}] \leq \exp(-\ell/18) \\
\Pr[\text{false negative}] \leq \exp(-\ell/18).
\]

9.3 Concentration for QuickSort

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.

Section 4.4: \(\mathbb{E}[X] < 2n \ln n\)
Section 7.2.3: \(\Pr[X \geq 200n \ln n] \leq 0.01\).

The Chernoff bound will now be exploited to give much tighter guarantees for the right tail.

Theorem 9.3.1.

\(\Pr[X \geq 36n \ln n] \leq 1/n\).

We will prove this by a completely different strategy than our earlier analysis. We will show that the recursion tree is very likely to have depth \(O(\log n)\). We begin with an easy claim showing how recursion depth relates to comparisons.

Claim 9.3.2. If the recursion tree has \(d\) levels then there are at most \(nd\) comparisons.

Proof. Notice that each non-pivot element is involved in at most one comparison at level \(i\). It follows that there are at most \(n\) comparisons in all subroutines at level \(i\) of the recursion tree. Summing over levels proves the claim. \(\square\)

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 $(2/3)^kn = e^{-\ln(3/2)k}n$. Consider $k = \log_{3/2}(n)$, which equals $\ln(n)/\ln(3/2)$ by the familiar log rule (A.2.2). 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 9.3.3.** 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, 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

$$\Pr \left[ \text{element } v \text{ is involved in } \geq d \text{ subproblems} \right] \leq \Pr \left[ Z < (1 - \delta)\mu \right] \leq \exp(-\delta^2\mu/3) \leq \exp(-2 \ln n) = \frac{1}{n^2}. \quad \Box$$

**Proof of Theorem 9.3.1.** Let

$$\mathcal{E}_i = \{\text{element } i \text{ is involved in } \geq d \text{ subproblems}\}.$$  

\(^1\)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.
Then

\[ \Pr[\text{recursion tree has } \geq d \text{ levels}] = \Pr[\mathcal{E}_1 \lor \cdots \lor \mathcal{E}_m] \]

\[ \leq \sum_{i=1}^{n} \Pr[\mathcal{E}_i] \quad \text{(by the union bound)} \]

\[ \leq \sum_{i=1}^{n} \frac{1}{n^2} \quad \text{(by Claim 9.3.3)} \]

\[ = \frac{1}{n}. \]

Thus, by Claim 9.3.2,

\[ \Pr[\text{number of comparisons } \geq nd] \leq \Pr[\text{recursion tree has } \geq d \text{ levels}] \leq 1/n. \]

References: (Mitzenmacher and Upfal, 2005, Exercise 4.20).

9.3.1 Exercises

Exercise 9.4. 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 \( \ell \) randomly chosen elements as the pivot. The changes from QUICKSORT are shown in yellow.

Algorithm 9.3 A variant of Randomized QuickSort that uses an approximate median. Assume that the elements in \( A \) are distinct.

1: global int \( \ell \)
2: function MEDOf\(\ell\)QuickSort(set \( A \))
3: if \( \text{Length}(A) \leq 5 \) then
4: \hspace{1em} return INSERTIONSORT \((A)\)
5: \hspace{1em} Select \( \ell \) elements \( Z_1, \ldots, Z_\ell \in A \) uniformly and independently at random
6: \hspace{1em} Find the median \( p \) of \( Z_1, \ldots, Z_\ell \) using INSERTIONSORT \( \triangleright \) The pivot element
7: \hspace{1em} Construct the sets \( \text{Left} = \{ x \in A : x < p \} \) and \( \text{Right} = \{ x \in A : x > p \} \)
8: \hspace{1em} return the concatenation \( [\text{MEDOf}\ell\text{QuickSort(Left)}, p, \text{MEDOf}\ell\text{QuickSort(Right)}] \)
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 \( \text{Length}(A) = s \), how much time do lines 5-7 take as a function of \( s \) and \( \ell \)?

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 \( O(n \log n) \).
9.4 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

$$\hat{f} = \frac{X - b}{1 - 2b}.$$  \hfill (9.4.1)

It is unbiased, meaning $E[\hat{f}] = f$.

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 $E[Z_i] = b$. Then define

- # unvaccinated liars: $U = \sum_{i \in \mathcal{U}} Z_i$
- # vaccinated liars: $V = \sum_{i \in \mathcal{V}} Z_i$

Then $E[U] = b(1 - f)n$ and $E[V] = bf n$.

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

$$X = \frac{1}{n} \left( \sum_{i \in \mathcal{V}} (1 - Z_i) + \sum_{i \in \mathcal{U}} Z_i \right)$$

$$= \frac{1}{n} (fn - V + U) = f - V/n + U/n.$$  \hfill (9.4.2)

Thus, as we already knew,

$$E[X] = f - bf + b(1 - f) = (1 - 2b)f + b.$$  \hfill (9.4.3)

Using a familiar Hoeffding bound, we can prove the following concentration result for $X$.

**Lemma 9.4.1.**

$$\Pr[|X - E[X]| < 4/\sqrt{n}] \geq 0.99.$$  \hfill (by Lemma 9.4.1)

Now we need to analyze the error in the estimator $\hat{f}$. We have

$$\Pr\left[|\hat{f} - f| < \frac{4}{(1 - 2b)\sqrt{n}}\right] = \Pr\left[\left|\frac{X - b}{1 - 2b} - f\right| < \frac{4}{(1 - 2b)\sqrt{n}}\right]$$

(by (9.4.1))

$$= \Pr\left[|X - (f(1 - 2b) + b)| < \frac{4}{\sqrt{n}}\right]$$

(multiplying by $1 - 2b$)

$$= \Pr\left[|X - E[X]| < \frac{4}{\sqrt{n}}\right]$$

(by (9.4.3))

$$\geq 0.99$$

(by Lemma 9.4.1).
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 9.4.2.** Suppose we want the error to be \( \epsilon \). How should we choose \( n \)?

**Answer.**

Define
\[
 n = \left(\frac{4\epsilon}{(1 - 2b)^2}\right)^2.
\]
Then the error will be at most \( \epsilon \) with probability at least 0.99.

**Exercises**

**Exercise 9.5.** Prove Lemma 9.4.1.

**9.5 Exercises**

**Exercise 9.6 Removing Runtime Randomness.** Let \( \mathcal{A} \) be a BPP-algorithm solving some problem, and whose runtime is random. Specifically, we know a constant \( c \) such that \( \mathbb{E}[\text{runtime of } \mathcal{A}] \leq cn^2 \) for inputs of size \( n \).

Using \( \mathcal{A} \) (and \( c \)), create another BPP-algorithm \( \mathcal{A}' \) solving the same problem, except that *always* have runtime of \( \mathcal{A}' = O(n^2) \). (So we have a non-random upper bound on the runtime.)

**Hint:** man timeout
Chapter 10

Classical statistics

10.1 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.

```
HTHTTTTTTTTTHTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
```

**Question 10.1.1.** 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

$$\mu = \mathbb{E}[X] = 60 \cdot \frac{2}{3} = 40.$$ 

We would like to show that it is unlikely that the observed number of heads is much smaller than $\mu$. 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 $\mu = \mathbb{E}[X] = 40$. We would like to show that it is unlikely that the number of heads is much smaller than $\mu$. The Hoeffding bound is ideal for this task.

$$\Pr[X \leq 18] = \Pr \left[ X \leq \frac{40}{\mu} - \frac{22}{\mu} \right] \leq \exp(-2t^2/60) \quad \text{(by the Hoeffding bound, Theorem 8.3.1)}$$

$$= \exp(-2 \cdot 22^2/60) < 0.000001.$$

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 10.1.2.** The Chernoff bound will also work for this problem. What bound do you get?

**Answer.**

Apply the Chernoff bound with $\gamma = \frac{1}{2}$ to get

$$\Pr[X \leq \frac{40}{\mu} - \frac{22}{\mu}] \leq \exp(-\delta^2/3) < e^{-4} < 0.019.$$ 

### 10.2 Polling

Recall from Section 4.2 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
\[
\Pr \left[ |\hat{f} - f| \geq 0.03 \right] \leq 0.05.
\] (10.2.1)

**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.

### 10.2.1 Sampling with replacement

As in Section 4.2.1, we pick $k$ customers uniformly and independently at random. We use the estimator $\hat{f} = \sum_{i=1}^{k} X_i / k$ from (4.2.1), where $X_i$ indicates whether the $i^{th}$ sampled person likes avocados.

**Theorem 10.2.1.** Let $k = 2050$. Let $\hat{f}$ be the output of POLLWITHREPLACEMENT($m, k$). Then, with probability 0.95, $\hat{f}$ is within 0.03 of $f$.

We have shown in Section 4.2 that $\mathbb{E} \left[ \hat{f} \right] = f$. Say we want that $\hat{f}$ is within $\pm 0.03$ of the true value of $f$, with probability 0.95. Using the Hoeffding Bound for Averages (Theorem 8.6.1), 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.

### 10.2.2 Sampling without replacement

As in Section 4.2.2, we pick $k$ customers uniformly and independently at random. We use the estimator $\hat{f} = \sum_{i=1}^{k} X_i / k$ from (4.2.2), where $X_i$ indicates whether the $i^{th}$ sampled person likes avocados.

**Theorem 10.2.2.** Let $k = 2050$. Let $\hat{f}$ be the output of POLLWITHOUTREPLACEMENT($m, k$). Then, with probability 0.95, $\hat{f}$ is within 0.03 of $f$. 

©Nicholas Harvey 2021. All rights reserved.
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 8.6.1.

**Theorem 10.2.3** (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 \(q \geq 0\),

\[
\Pr \left[ Y \geq E[Y] + q \right] \leq \exp(-2q^2n)
\]

and

\[
\Pr \left[ Y \leq E[Y] - q \right] \leq \exp(-2q^2n).
\]

Combining these

\[
\Pr \left[ |Y - E[Y]| \geq q \right] \leq 2 \exp(-2q^2n).
\]

**References:** Hoeffding’s paper, Section 6.

We now follow the same argument used above for sampling with replacement, but use Theorem 10.2.3 instead. Plugging in \(q = 0.03\) and \(n = k = 2050\), we obtain

\[
\Pr \left[ |\hat{f} - f| \geq 0.03 \right] \leq 2 \exp(-2 \cdot 0.03^2 \cdot 2050) < 0.05.
\]

This proves Theorem 10.2.2.

### 10.2.3 Bernoulli sampling

**Theorem 10.2.4.** Let \(k = 12300\). Let \(\hat{f}\) be the output of \(\text{BERNOULLIPOLLING}(C,k)\). 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

\[
\text{True fraction: } f = \frac{\left| \mathcal{A} \right|}{m},
\]

\[
\text{Estimated fraction: } \hat{f} = \frac{X}{pm} = \frac{1}{pm} \sum_{i \in \mathcal{A}} X_i.
\]

We have shown in Claim 4.2.4 that \(\hat{f}\) is unbiased, i.e., \(E[\hat{f}] = f\). Multiplying by \(pm\), this yields

\[
E[X] = pmf.
\]

We next use the Chernoff bound to show that \(\hat{f}\) is concentrated around its mean.

**Lemma 10.2.5.** \(\Pr \left[ |\hat{f} - f| \geq 0.03 \right] \leq 2 \exp(-0.0003pm)\).

**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 \( \mu = \mathbb{E}[X] = pmf \), by (10.2.5).

\[
\Pr\left[ |\hat{f} - f| \geq 0.03 \right] = \Pr\left[ |X - pmf| \geq 0.03 pm \right] \quad \text{(multiply both sides of inequality by pm)}
\]

\[
= \Pr\left[ \left| X - \mu \right| \geq \frac{0.03 \ pm \ f}{\mu} = \delta \right] \quad \text{(using value of \( \mu \)),}
\]

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 8.2.1 yields

\[
\Pr\left[ \left| X - \mu \right| \geq \delta \mu \right] \leq 2 \exp(-\delta^2 \mu/3)
\]

\[
= 2 \exp\left(-\left(\frac{0.03 \ pm \ f}{f} \right)^2 \frac{pm \ f}{3} \right)
\]

\[
\leq 2 \exp(-0.03^2 pmf / 3) \quad \text{(using \( f \leq 1 \))}
\]

\[
= 2 \exp(-0.0003 pm).
\]

We conclude that \( \Pr\left[ |\hat{f} - f| \geq 0.03 \right] = \Pr\left[ |X - \mu| \geq \delta \mu \right] \leq 2 \exp(-0.0003 pm) \).

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

\[
\Pr\left[ |\hat{f} - f| > 0.03 \right] \leq 2 \exp(-0.0003 pm) = 2 \exp(-3.69) < 0.05.
\]

This proves Theorem 10.2.4.

10.2.4 Exercises

Exercise 10.1. Let \( \epsilon \) and \( \alpha \) satisfy \( 0 < \epsilon < 1 \) and \( 0 < \alpha < 1 \). Suppose we wanted an estimate that was accurate to within \( \epsilon \) with failure probability \( \alpha \). Show that it suffices to poll \( O(\log(1/\alpha)/\epsilon^2) \) customers, in expectation.

10.3 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.

\(^1\)Instead of using this trick, we could instead use Theorem 8.2.3 to handle the case of \( \delta > 1 \).
Algorithm 10.1 The A/B testing algorithm. We assume that \( m = |C| \) is even.

1: function \( \text{ABTest}(\text{array } C) \)
2: Let \( C^A, C^B \leftarrow \text{PartitionWithoutReplacement}(C, 2) \) \( \triangleright \) See Algorithm 2.5
3: Let \( X^A \leftarrow 0 \) \( \triangleright \) Number of customers who bought from old website
4: for \( c \in C^A \)
5: Show user \( c \) the old website
6: if user \( c \) purchases then \( X^A \leftarrow X^A + 1 \) \( \triangleright \) This happens if \( c \in A \)
7: Let \( X^B \leftarrow 0 \) \( \triangleright \) Number of customers who bought from new website
8: for \( c \in C^B \)
9: Show user \( c \) the new website
10: if user \( c \) purchases then \( X^B \leftarrow X^B + 1 \) \( \triangleright \) This happens if \( c \in B \)
11: return \( \hat{f}^A = \frac{2}{m} X^A \) and \( \hat{f}^B = \frac{2}{m} X^B \)
12: end function

To make the problem more concrete, we will need a fair amount of notation.

\[
C = \{\text{all customers}\}
\]
\[
m = |C| = \text{number of customers}
\]
\[
A = \{\text{customers who would buy from the old website}\}
\]
\[
B = \{\text{customers who would buy from the new website}\}
\]
\[
f^A = \frac{|A|}{|C|} = \text{fraction of customers who would buy from the old website}
\]
\[
f^B = \frac{|B|}{|C|} = \text{fraction of customers who would buy from the new website}
\]
\[
f^B - f^A = \text{the improvement of the new website}
\]

If the improvement \( f^B - f^A \) 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 \textit{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 \textit{not both}. This is accomplished by the \textsc{ABTest} algorithm, shown in Algorithm 10.1.

The \textsc{ABTest} algorithm has one key idea: randomly partition the set \( C \) of customers into two groups, \( C^A \) and \( C^B \), using the \textsc{PartitionWithoutReplacement} algorithm from Section 2.3 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:

\[
\hat{f}^A = \frac{2}{m} X^A = \frac{|C^A \cap A|}{|C^A|} = \text{fraction of group } C^A \text{ who would buy from old website}
\]
\[
\hat{f}^B = \frac{2}{m} X^B = \frac{|C^B \cap B|}{|C^B|} = \text{fraction of group } C^B \text{ who would buy from new website.}
\]

Notice that these estimators have the same form as the polling estimator (4.2) discussed in Section 4.2. We can analyze these estimates by considering the \textsc{PollWithoutReplacement} algorithm from Sec-
We claim that
\[
\hat{f}^A \text{ has the same distribution as } \text{PollWithoutReplacement}(C, m/2, A)
\]
\[
\hat{f}^B \text{ has the same distribution as } \text{PollWithoutReplacement}(C, m/2, B).
\]

This follows from Claim 2.3.2, which states the mysterious property of partitioning without replacement: both \(C^A\) and \(C^B\) have the same distribution as \(\text{SampleWithoutReplacement}(C, k)\).

**Theorem 10.3.1.** Suppose there are \(m \geq 5000\) customers. With probability 0.95, the estimated improvement \(\hat{f}^B - \hat{f}^A\) differs from the true improvement \(f^B - f^A\) by at most 0.06.

**Proof.** The proof just applies the analyses of Section 10.2. 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 (10.2.2) with \(k = m/2\) and \(m = 5000\), we get
\[
\Pr \left[ |\hat{f}^A - f^A| \geq 0.03 \right] \leq 2 \exp(-2 \cdot (0.03)^2 \cdot k) < 0.025
\]
\[
\Pr \left[ |\hat{f}^B - f^B| \geq 0.03 \right] \leq 2 \exp(-2 \cdot (0.03)^2 \cdot k) < 0.025.
\]
By a union bound, there is a small probability that either estimate is inaccurate.
\[
\Pr \left[ |\hat{f}^A - f^A| \geq 0.03 \lor |\hat{f}^B - f^B| \geq 0.03 \right] < 0.05
\]
Taking the complement, there is a large probability that both estimates are accurate.
\[
\Pr \left[ |\hat{f}^A - f^A| < 0.03 \land |\hat{f}^B - f^B| < 0.03 \right] > 0.95
\]
Assuming this event occurs, we have the following bound on the error.
\[
|\text{(Estimated improvement)} - \text{(True improvement)}| = \left| (\hat{f}^B - \hat{f}^A) - (f^B - f^A) \right|
\]
\[
\leq |\hat{f}^A - f^A| + |\hat{f}^B - f^B|
\]
\[
< 0.03 + 0.03 = 0.06,
\]
where the first inequality is the triangle inequality, Fact A.2.4.

**Applying the theorem.** How can we use Theorem 10.3.1 in practice? One option is to switch to using the new website if \(\hat{f}^B - \hat{f}^A \geq 0.06\). With this rule, we have the following possibilities.

- **Sampling fails.** This happens with probability \(\leq 0.05\), in which case anything can happen.
- **Sampling succeeds.** This happens with probability \(\geq 0.95\). The following cases hold.
  - **True improvement is large.** If \(f^B - f^A \geq 0.12\) then \(\hat{f}^B - \hat{f}^A \geq 0.06\), so we definitely switch to the new website.
  - **True improvement is modest.** If \(0 \leq f^B - f^A < 0.12\) then the estimated improvement might or might not exceed 0.06.
  - **True improvement is negative.** If \(f^B - f^A < 0\) then \(\hat{f}^B - \hat{f}^A < 0.06\), so we definitely don’t switch to the new website.
10.3.1 Exercises

Exercise 10.2. 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 $\text{OPT} = \max_{i \in [n]} f^i$ be the fraction of buyers from the best website.

Give an algorithm to identify a website $i^*$ satisfying $f^{i^*} \geq \text{OPT} - \epsilon$. 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.

10.4 Estimating Distributions

10.4.1 Bernoulli distributions

As suggested by Exercise 4.1, 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

$$\hat{p} = \frac{1}{k} \sum_{i=1}^{k} X_i.$$

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 10.4.1.** Let the number of samples be $k = \ln(200)/2\epsilon^2$. Then $\hat{p}$ is

Unbiased: $\mathbb{E}[\hat{p}] = p$

Concentrated: $\Pr[|\hat{p} - p| \geq \epsilon] \leq 1/100$.

**Proof.** The unbiasedness follows from $\mathbb{E}[X_i] = p$ and linearity of expectation.

$$\mathbb{E}[\hat{p}] = \sum_{i=1}^{k} \mathbb{E}[X_i]/k = k \cdot p/k = p.$$

The concentration follows from Hoeffding for averages (Theorem 8.6.1).

$$\Pr[|\hat{p} - p| \geq \epsilon] \leq 2 \exp(-2\epsilon^2 k) = 2 \exp\left(-2\epsilon^2 \cdot \frac{\ln(200)}{2\epsilon^2}\right) = 2 \exp\left(-\ln(200)\right) = 1/100$$

10.4.2 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 \( \epsilon \) 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 10.4.2.** Let the number of samples be \( k = \ln(200k)/2\epsilon^2 \). Then \( \hat{p} \) is

- **Unbiased:** \( E[ \hat{p}_j ] = p_j \forall j \)
- **Concentrated:** \( \Pr[ \text{any } j \text{ has } |\hat{p}_j - p_j| \geq \epsilon ] \leq \frac{1}{100} \).

**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 \( E[ X_i ] = p_j \), so
\[
E[ \hat{p}_j ] = E \left[ k \sum_{i=1}^{k} X_i / k \right] = \sum_{i=1}^{k} E[ X_i ] / k = k \cdot p_j / k = p_j.
\]
Next we consider the concentration.
\[
\Pr[ |\hat{p}_j - p_j| \geq \epsilon ] \leq 2 \exp(-2\epsilon^2 k) \quad \text{(by Theorem 8.6.1)}
\]
\[
= 2 \exp \left( -2\epsilon^2 \cdot \frac{\ln(200k)}{2\epsilon^2} \right) = \frac{1}{100k}.
\]
Thus, by a union bound,
\[
\Pr[ \text{any } j \text{ has } |\hat{p}_j - p_j| \geq \epsilon ] \leq \sum_{j=1}^{k} \Pr[ |\hat{p}_j - p_j| \geq \epsilon ] \leq k \cdot \frac{1}{100k} = \frac{1}{100}. \quad \square
\]

**10.4.3 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
\[
|\hat{F}(x) - F(x)| \leq \epsilon \quad \forall x \in \mathbb{R}.
\]
(10.4.1)
This is a very strong condition: \( \hat{F} \) has to approximate \( F \) at all points \( x \).

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 \( \epsilon \).
This condition becomes (10.4.1) if we replace $\epsilon$ occurring is bounded by (10.4.4). So with the adjusted value of $\epsilon$

In summary, assuming the bad events do not occur, we have shown that $q$

The number of these events is $Hoeffding for averages (Theorem 8.6.1), Observe that $\hat{F}(x)$

The lower bound is similar.

In summary, assuming the bad events do not occur, we have shown that $|\hat{F}(x) - F(x)| < 2\epsilon \quad \forall x \in \mathbb{R}$.

This condition becomes (10.4.1) if we replace $\epsilon$ with $\epsilon/2$. Recall that the probability of any bad event occurring is bounded by (10.4.4). So with the adjusted value of $\epsilon$, we obtain (10.4.3).
10.4.4 Exercises

Exercise 10.3. If \( k \gg 1/\epsilon \) then Theorem 10.4.2 can be improved. Using the same estimator, show that \( O(\log(1/\epsilon)/\epsilon^2) \) samples suffice to obtain

\[
\Pr \left[ \text{any } j \text{ has } |\hat{p}_j - p_j| \geq \epsilon \right] \leq 1/100.
\]

Exercise 10.4. The DKW theorem (Theorem 10.4.3) implies Hoeffding for averages (inequality (8.6.1)), 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 10.5. The weakest aspect of Theorem 10.4.4 is the constant in the exponent: whereas (10.4.3) has a constant of \(-1/2\), (10.4.2) has a better constant of \(-2\). Fortunately, this can be improved without too much effort. Prove that

\[
\Pr \left[ \text{(10.4.1) fails to hold} \right] \leq O(1/\epsilon) \cdot \exp(-1.99\epsilon^2 s).
\]

Exercise 10.6. Prove Theorem 10.4.4 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 11

Introduction to Hashing

11.1 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. Here are various different options for hash function:

- Just use single function that “seems” random, e.g., Murmur.
  
  Advantages: These functions are easy to design and implement, and fast to execute.
  Disadvantages: 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.

- Use a single hash function, but inject some randomness into keys. 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.
  Disadvantages: The notion of security changes over time. They are relatively slow to execute. Since there is just one function, any two colliding inputs will always collide. MD5 is very broken, and SHA-1 is very broken too.
  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 13.

**Advantages:** We can prove results about the probability of collisions. They can actually be implemented.

**Disadvantages:** They might require some number theory/abstract algebra in its 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.

**Question 11.1.1.** What does “purely random function” mean? How many bits of space does it take to represent a purely random function from \([n]\) to \([k]\)?

**Answer.**

The meaning was discussed on page 44: you could list all the functions from \([n]\) to \([k]\), then pick one of these functions uniformly at random. Alternatively, for each possible input in \([n]\), you could randomly and independently generate an output in \([k]\). The number of such functions is \(k^n\), and representing such a function requires \(\Theta(n \log k)\) bits.

**Remark 11.1.2.** When discussing space complexity, there are two common conventions regarding the units of space.

• One convention is to count the number of **bits**; thus, each unit of space can store only two different values. This convention was used in Question 11.1.1 above.

• 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 of the problem.

For example, if we say that an ordered binary tree with \(n\) entries takes \(O(n)\) space, we are implicitly referring to **words** of space, not bits, because each key and each pointer would take \(\Omega(\log n)\) bits to represent the \(n\) distinct values.

**Question 11.1.3.** Let \(h\) be a purely random function producing outputs in \([k]\). For any distinct inputs \(x, y\), what is \(\Pr[h(x) = h(y)]\)?

©Nicholas Harvey 2021. All rights reserved.
Answer.

\[ \begin{align*}
\mathcal{Y}/1 &= \sum_{\gamma \in \mathcal{Y}} \mathbb{P}(\gamma/1) \\
\mathbb{P}(\gamma = \mathcal{Y}/1) &= \mathbb{P}(\gamma = \mathcal{Y} \cap \mathcal{Y}/1) \\
\mathbb{P}(\gamma = \mathcal{Y} \cap \mathcal{Y}/1) &= \mathbb{P}(\gamma = \mathcal{Y}/1)
\end{align*} \]

Exercises

Exercise 11.1. Let \( h \) be a purely random function producing outputs in \([k]\). For any distinct inputs \( x_1, \ldots, x_\ell \), what is \( \mathbb{P}(h(x_1) = \cdots = h(x_\ell)) \)?

11.2 MinHash: estimating set similarity

A common data analysis problem is to measure the similarity of two sets. For example, maybe

\[ A = \{ \text{Angela’s favourite books} \} \]
\[ B = \{ \text{Boris’s favourite books} \} \]

Then we might judge the similarity of Angela and Boris by judging the similarity of the sets \( A \) and \( B \). How can we make this mathematically precise? Here is one reasonable definition.

Definition 11.2.1. The similarity of sets \( A \) and \( B \) is

\[ \text{Sim}(A, B) = \frac{|A \cap B|}{|A \cup B|}, \]

assuming that \( A \cup B \neq \emptyset \).

References: Wikipedia.

Question 11.2.2. How can you compute \( \text{Sim}(A, B) \) 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.

11.2.1 The basic estimator

Suppose we have a purely random function \( h \) that can map any string to a uniform real number in \([0,1] = \{ x : 0 \leq x \leq 1 \} \). The key advantage of hashing to a real number is that we don’t have to worry about collisions. More precisely, for any two string \( a \) and \( b \), we have

\[ \text{No collisions: } \mathbb{P}(h(a) = h(b)) = 0. \]
Algorithm 11.1 The Set class implements a data structure that allows efficient estimation of set similarity.

```
// global hash function h

class Set:

    float minValue

    // The constructor. E is the set of entries to insert.
    Constructor (strings E)
        minValue ← min{ h(e) : e ∈ E }

    EstimateSim (Set A, Set B) returns float
        if A.minValue = B.minValue
            return 1
        else
            return 0
```

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. For example, if $U \geq 100 \cdot |A \cup B|^2$ then there is only a 0.01 probability of any collisions, due to the birthday paradox (see Exercise 7.1).

The algorithm, shown in Algorithm 11.1, 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.

Amazingly, it seems that this data structure represents 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, using the discretization scheme described above, it suffices to take $U = O(n^2)$ for sets of size $n$, and therefore only $O(\log n)$ bits are needed to represent a hash value. For concreteness, we will represent the minimum hash value as a float, which is just a single word of space.

Although this data structure appears to do very little, in expectation it exactly determines the set similarity. In other words, the estimated similarity is unbiased.

**Claim 11.2.3.** $E[\text{EstimateSim}(A, B)] = \text{Sim}(A, B)$.

**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 $\text{EstimateSim}(A, B)$ only returns 1 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

$$E[\text{EstimateSim}(A, B)] = \Pr[A$.minValue $= B$.minValue $] = \frac{|A \cap B|}{|A \cup B|} = \text{Sim}(A, B). \quad (11.2.1)$$
Does this mean that $\text{EstimateSim}$ is good? Reiterating our moral from Section 7.1:

The expectation of a random variable does not tell you everything.

In fact, $\text{EstimateSim}$ is usually terrible. Consider the example $A = \{\text{apple, banana}\}$ and $B = \{\text{banana}\}$, so $\text{Sim}(A, B) = 0.5$. The output of $\text{EstimateSim}(A, B)$ can only be 0 or 1, so it always has error exactly 0.5.

### 11.2.2 The averaged estimator

Unsurprisingly, we get an improved estimator by averaging multiple basic estimators. The first step is to randomly choose $k$ independent purely random functions $h_1, \ldots, h_k$. 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.

**Algorithm 11.2** The improved Set class that uses $k$ independent hash functions.

```plaintext
global int $k$
global hash functions $h_1, \ldots, h_k$

struct Signature:
    float minValue[1..$k$]

class Set:

    Signature $\text{sig}$

    $\triangleright$ The constructor. $E$ is the set of entries to insert.
    \hspace{1em} \text{CONSTRUCTOR ( strings $E$ )}
    \hspace{1em} \hspace{1em} \textbf{for} $i = 1, \ldots, k$
    \hspace{1em} \hspace{2em} $\text{sig}.\text{minValue}[i] \leftarrow \min \{ h_i(e) : e \in E \}$

    $\text{EstimateSim} (\text{Set } A, \text{ Set } B)$ returns float
    \hspace{1em} $X \leftarrow 0$ \hspace{1em} $\triangleright$ The number of matching hash values in their signatures
    \hspace{1em} \textbf{for} $i = 1, \ldots, k$
    \hspace{1em} \hspace{2em} \textbf{if} $A.\text{sig}.\text{minValue}[i] = B.\text{sig}.\text{minValue}[i]$
    \hspace{1em} \hspace{3em} $X \leftarrow X + 1$
    \hspace{1em} \textbf{return} $X/k$ \hspace{1em} $\triangleright$ The fraction of matching hash values
```

**Analysis.** Let $X_i$ be the indicator of whether the $i^{\text{th}}$ hash values match.

$$X_i = \begin{cases} 1 & \text{(if } A.\text{sig}.\text{minValue}[i] = B.\text{sig}.\text{minValue}[i] \text{)} \\ 0 & \text{(otherwise)}. \end{cases}$$

The $X_i$’s are independent since $h_1, \ldots, h_k$ are independent. We have already shown in (11.2.1) that $\mathbb{E}[X_i] = \text{Sim}(A, B)$.

$\text{EstimateSim}$ is defined to return the average of the $X_i$’s. That is,

$$\text{EstimateSim}(A, B) = \frac{X}{k} = \frac{1}{k} \sum_{i=1}^{k} X_i. \quad (11.2.2)$$
By linearity of expectation,
\[
E[\text{EstimateSim}(A, B)] = \frac{1}{k} \sum_{i=1}^{k} E[X_i] = \text{Sim}(A, B). \tag{11.2.3}
\]

The purpose of averaging \(k\) basic estimators is to ensure that \(\text{EstimateSim}(A, B)\) is much better concentrated around its expectation.

**Theorem 11.2.4.** Let \(\epsilon > 0\) be arbitrary. Suppose we set \(k = \ln(200)/2\epsilon^2\). Then
\[
\Pr[|\text{EstimateSim}(A, B) - \text{Sim}(A, B)| \geq \epsilon] \leq 0.01.
\]

**Proof.** As shown in (11.2.2), \(\text{EstimateSim}(A, B)\) is an average of \(k\) independent indicator RVs. Let us write \(Y = \text{EstimateSim}(A, B)\) to simply notation. By (11.2.3), we have \(E[Y] = \text{Sim}(A, B)\). Now we can directly apply Hoeffding for averages.
\[
\Pr[|\text{EstimateSim}(A, B) - \text{Sim}(A, B)| \geq \epsilon]
= \Pr[|Y - E[Y]| \geq \epsilon]
\leq 2 \exp(-2\epsilon^2 k) \quad \text{(by Theorem 8.6.1)}
= 2 \cdot \frac{1}{200} = 0.01
\]
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.

### 11.3 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,

- Avocado = \{green, big seed, mushy\}
- Beet = \{red, sweet, needs boiling\}
- Cabbage = \{green, large, leafy\}
- Carrot = \{orange, long, crunchy\}
- Pea = \{green, small, mushy\}

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

Brussels Sprout = \{green, small, leafy\}.

From Zoodle’s list, both Cabbage and Pea have similar\(^1\) properties to the Brussels Sprout; in fact
\[
\text{Sim}(\text{Brussels Sprout, Cabbage}) = \text{Sim}(\text{Brussels Sprout, Pea}) = \frac{2}{4} = 0.5.
\]

\(^1\)In fact, Brussels Sprouts and Cabbages are exactly the same species, *Brassica oleracea*. 
Algorithm 11.3 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 Set:
    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 Set data structures
    Constructor ( Set 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 ( Set 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
```

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 11.3.

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 11.3.1.** Consider sets $A$, $B$ in the NearNeighbour data structure, and a query set $Q$.

- **Similar sets:** if $\text{Sim}(Q, A) \geq 0.9$ then $\Pr[\text{QUERY}(Q) \text{ outputs } A] > 0.99$ \quad (11.3.1)
- **Dissimilar sets:** if $\text{Sim}(Q, B) \leq 0.5$ then $\Pr[\text{QUERY}(Q) \text{ outputs } B] < 0.01$. \quad (11.3.2)

To prove this, we will first need the following lemma.

**Lemma 11.3.2.** For any two sets $A$ and $B$ and any $i \in [\ell]$, \[ \Pr[A\text{.sig}[i] = B\text{.sig}[i]] = \text{Sim}(A, B)^k. \]

**Proof.** \[ \Pr[A\text{.sig}[i] = B\text{.sig}[i]] = \Pr[A\text{.sig}[i]\text{.minValue}[j] = B\text{.sig}[i]\text{.minValue}[j] \forall j \in [k]] = \prod_{j=1}^{k} \Pr[A\text{.sig}[i]\text{.minValue}[j] = B\text{.sig}[i]\text{.minValue}[j]] \quad (\text{by independence}) \]

\[ = \prod_{j=1}^{k} \text{Sim}(A, B) \quad (\text{by (11.2.1)}) \]

\[ = \text{Sim}(A, B)^k \quad \square \]

**Proof of Theorem 11.3.1.** Suppose $\text{Sim}(Q, A) \geq 0.9$. The set $A$ is not output if all $\ell$ of the signatures disagree.

\[ \Pr[\text{QUERY}(Q) \text{ doesn't output } A] = \Pr[Q\text{.sig}[i] \neq A\text{.sig}[i] \forall i \in [\ell]] = \prod_{i=1}^{\ell} \Pr[Q\text{.sig}[i] \neq A\text{.sig}[i]] \quad (\text{by independence}) \]

\[ = (1 - \text{Sim}(Q, A))^k \ell \quad (\text{by Lemma 11.3.2}) \]

\[ \leq (1 - 0.9^k)^\ell < 0.01, \]

by a quick numerical calculation using $k = 11$ and $\ell = 13$. This proves (11.3.1).

Now suppose $\text{Sim}(Q, B) \leq 0.5$. The set $B$ is output if at least one of the $\ell$ signatures agree.

\[ \Pr[\text{QUERY}(Q) \text{ outputs } B] = \Pr[Q\text{.sig}[i] = B\text{.sig}[i] \text{ for some } i \in [\ell]] \leq \sum_{i=1}^{\ell} \Pr[Q\text{.sig}[i] = B\text{.sig}[i]] \quad (\text{by union bound}) \]

\[ = \ell \cdot \text{Sim}(Q, B)^k \quad (\text{by Lemma 11.3.2}) \]

\[ \leq \ell \cdot 0.5^k < 0.01, \]

by a quick numerical calculation using $k = 11$ and $\ell = 13$. This proves (11.3.2). \( \square \)

**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.
11.4 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.

```python
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 6.** 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 7.** 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.

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.

11.4.1 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 11.4.

**Question 11.4.1.** 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 bit is never changed back from `True` to `False`, so `IsMember` will return `True`. ©Nicholas Harvey 2021. All rights reserved.
Algorithm 11.4 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 $j = 1, \ldots, n$
        Set $B[h(Keys[j])] ← True$

▷ Tests whether the string $x$ was inserted. It could make an error.
ISMEMBER (string $x$)
    return $B[h(x)]$

Question 11.4.2. 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 $n/100 \leq n$ entries that are True, i.e., $1/100$ 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.

11.4.2 Bloom Filters

A notable flaw of the BasicFilter is that its array $B$ 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: using multiple independent trials to decrease false positives. The pseudocode for a Bloom filter is shown in Algorithm 11.5. 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 11.4.3. In a BloomFilter, at most half of the entries of $B$ are True. To see this, note that the for loops set exactly $nk$ 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.
Algorithm 11.5 A Bloom Filter. It represents a set of size \( n \) using only \( 14n \) bits, and with a false positive probability of 0.01.

\[
\text{global int } k = \lceil \lg(100) \rceil = 7 \quad \triangleright \text{ Number of hash functions}
\]

class BLOOMFILTER:
   hash functions \( h_1, \ldots, h_k \)
   Boolean array \( B \)

\[
\triangleright \text{ The constructor. Keys is the list of keys to insert.}
\]

CONSTRUCTOR ( strings Keys[1..n] )
   Set \( m \leftarrow 2^{kn} \) \quad \triangleright \text{ Length of the array}
   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 \( B[h_i(\text{Keys}[j])] \leftarrow \text{True} \)

\[
\triangleright \text{ Tests whether the string } x \text{ was inserted. There can be false positives.}
\]

ISMEMBER ( string \( x \) )
   return \( B[h_1(x)] \land \cdots \land B[h_k(x)] \)

Claim 11.4.4. Suppose \( x \) was \textit{not} inserted into the Bloom filter. Then, for all \( i \),
\[
\Pr [ B[h_i(x)] = \text{True} \mid \text{whatever } B \text{ is} ] \leq 1/2.
\]

If you removed the conditioning from Claim 11.4.4, the statement would still be true, but weaker. Claim 11.4.4 says that, regardless of what happens during the construction of \( B \), the event \( B[h_i(x)] = \text{True} \) still has probability at most 1/2.

Proof. Recall our assumption that \( h_i \) is 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. (That is exactly what it means for \( h_i \) to be a purely random function.)

So, even if we know the locations of all the \textbf{True} entries of \( B \), then the hash value \( h_i(x) \) is still a uniformly random value in \([m]\). It follows that
\[
\Pr [ B[h_i(x)] = \text{True} \mid \text{whatever } B \text{ is} ] = \frac{\#	ext{True entries in } B}{\text{size of } B} \leq \frac{m/2}{m} = \frac{1}{2},
\]
by Observation 11.4.3.

Claim 11.4.5. Regardless of what array \( B \) is produced by CONSTRUCTOR, the false positive probability is less than 0.01.

Proof. Consider any key \( x \) that was \textit{not} inserted.
\[
\Pr [ \text{ISMEMBER}(x) = \text{True} \mid \text{whatever } B \text{ is} ]
= \Pr [ B[h_1(x)] = \text{True} \land \cdots \land B[h_k(x)] = \text{True} \mid \text{whatever } B \text{ is} ]
\]
\[
\begin{align*}
= \prod_{i=1}^{k} \Pr [ h_i(x) = \text{True} \mid \text{whatever } B \text{ is}] & \quad (h_1(x), \ldots, h_k(x) \text{ are independent}) \\
\leq (1/2)^k & \quad (\text{by Claim 11.4.4}) \quad (11.4.1) \\
< 0.01,
\end{align*}
\]

since we have chosen \( k = 7 \).

To conclude, the \textsc{BloomFilter} uses \( m = 14n \) bits to represent \( n \) items. It has a false positive probability of less than 0.01.

**Question 11.4.6.** Suppose we wanted a false positive probability of \( \epsilon \). How many hash functions should we use?

**Answer.**

\[ k \leftarrow \lceil \log(1/\epsilon) \rceil. \]

**Keener Kwestion 11.4.7.** 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?

### 11.4.3 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 \( \epsilon \), and no false negatives. How many bits of space must such a data structure use?

- **Upper bound.** Section 11.4.2 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 11.4.8.** Where was the analysis of Section 11.4.2 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 11.6 describes the improvement. It differs from Algorithm 11.5 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 11.4.9.** What is the false positive probability of Algorithm 11.6?

**Answer.**

Since the construction process guarantees that at most \( n^2/2 \) entries of \( B \) are set to true, the proof of Claim 11.4.4 and Question 11.4.5 continue to hold. Thus, the false positive probability is
Algorithm 11.6 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 \( \epsilon \).

```
global float \( \epsilon \)  \Comment{Desired false positive probability}  
global int \( k = \lceil \lg(1/\epsilon) \rceil \)  \Comment{Number of hash functions}  

class BloomFilter:
    hash functions \( h_1, \ldots, h_k \)
    Boolean array \( B \)

\( \triangleright \) The constructor. Keys is the list of keys to insert.
constructor ( strings Keys[1..n] )
    Set \( m \leftarrow \lceil 1.45 kn \rceil \)  \Comment{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 \( B[h_i(\text{Keys}[j])] \leftarrow \text{True} \)
        until at most \( m/2 \) entries of \( B \) are True

\( \triangleright \) Tests whether the string \( x \) was inserted. There can be false positives.
isMember ( string \( x \) )
    return \( B[h_1(x)] \land \cdots \land B[h_k(x)] \)
```

Claim 11.4.10. Assume that \( m \geq 320 \). Consider any single iteration of the repeat loop that constructs \( B \). Then, for any array index \( i \), \( \Pr [ B[i] = \text{True} ] < 0.499 \).

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 5.1. We have chosen \( m \) to be large enough that

\[
kn \leq m/1.45.
\] (11.4.2)

By a typical balls and bins analysis (e.g. Section 5.2),

\[
\Pr [ B[i] = \text{True} ] = \Pr [ \text{bin } i \text{ is non-empty} ]
= 1 - \Pr [ \text{every ball misses bin } i ]
= 1 - \prod_{j=1}^{kn} \Pr [ \text{ball } j \text{ misses bin } i ]
= 1 - (1 - 1/m)^{kn}
\leq 1 - \left( (1 - 1/m)^m \right)^{1/1.45} \quad \text{(by (11.4.2))}
\leq 1 - \left( \frac{1}{e} - \frac{1}{4m} \right)^{1/1.45} \quad \text{(by Fact A.2.6)} \quad (11.4.3)
\]
by a quick numerical calculation, using our assumption that $m \geq 320$.

Claim 11.4.11. The \texttt{repeat} loop performs $O(1)$ iterations, in expectation.

\textit{Proof.} We can view the \texttt{repeat} loop as performing independent trials until the first success, which occurs when at most $m/2$ entries of $B$ are \texttt{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 \texttt{True}. It follows from Claim 11.4.10 that $\mathbb{E}[X] < 0.499m$. So now we can use Markov’s inequality to show that

$$1 - p = \Pr[\text{trial fails}] = \Pr[X > m/2] \leq \frac{\mathbb{E}[X]}{m/2} < \frac{0.499m}{m/2} = 0.998.$$ 

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.3.18), the expected number of trials is $1/p < 1/0.002 = 500$.

We summarize this analysis with the following theorem.

Theorem 11.4.12. Let $n \geq 222$. A Bloom filter can represent a set of $n$ elements with false positive probability $\epsilon$ using only $1.45n \lceil \lg(1/\epsilon) \rceil$ bits of space. The expected time for \textsc{Constructor} is $O(n \log(1/\epsilon))$. The time for \textsc{IsMember} is $O(\log(1/\epsilon))$.

Remark 11.4.13. Where does this strange number 1.45 come from? It is chosen so that in line (11.4.3) we roughly satisfy the equation $(\frac{1}{2})^{1/c} = \frac{1}{2}$, which has the solution $c = 1/\ln(2) \approx 1.45$. This equation arises when we try to ensure that at most half of $B$ is \texttt{True}.

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 11.4.14. Bloom filters are apparently useful for interviews. See, e.g., here, here, or here.
11.5 Exercises

Exercise 11.2 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

- **Similar sets:** if \( \text{Sim}(Q, A) \geq 0.9 \) then \( \Pr[\text{QUERY}(Q) \text{ outputs } A] > 0.99 \)
- **Dissimilar sets:** if \( \text{Sim}(Q, B) \leq 0.81 \) then \( \Pr[\text{QUERY}(Q) \text{ outputs } B] < \frac{1}{\sqrt{n}} \).

Part III. For a query set \( Q \), suppose that

- **Similar sets:** there are \( O(1) \) sets \( A \) with \( \text{Sim}(A, Q) \geq 0.9 \);
- **Dissimilar sets:** there are \( \Theta(n) \) sets \( B \) with \( \text{Sim}(B, Q) \leq 0.81 \);
- there are no other sets.

Assume that the dictionary is implemented as a hash table. What is the expected runtime of QUERY?

Exercise 11.3. Algorithm 11.6 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 11.4.11, 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 8.3.1), prove that

\[ \Pr[\text{repeat loop in Algorithm 11.6 runs more than once}] \leq 0.01. \]

\(^2\)See Chapter 25.
Chapter 12

Streaming algorithms

12.1 The streaming model

The traditional algorithms that we study in introductory classes assume that all the algorithm’s data is in memory, and 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 is a useful reflection of the task facing network routers, popular internet websites, etc.

12.1.1 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

\[
m = \text{(length of the stream)}
\]

\[
n = \text{(size of the universe containing the items)}
\]

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 a vector (or array) indicating the number of occurrences of each item. We will think of this as a vector \(f\) 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.

- What item has the highest frequency (i.e., \(\arg \max_{1 \leq i \leq n} f_i\))? What is its frequency (i.e., \(\max_{1 \leq i \leq n} f_i\))?
- 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 12.1.1. 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 12.1.2. 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 12.1.3. 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., $m = \sum_{i=1}^{n} f_i$.

We are interested in whether all these sorts of queries can be answered using a logarithmic\(^1\) amount of space. This will usually involve some approximation, randomization, etc.

### 12.2 The majority problem

The algorithm must decide whether the data has a (strict) majority item. More concretely, there are two cases.

- **Majority exists.** If some integer $M \in [n]$ appears in the stream more than half of the time, then the algorithm must return $M$. In symbols, the condition is that $M$ must satisfy $f_M > m/2$.

- **No majority exists.** If no such $M$ 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.

**Ignoring the case of no majority.** Interestingly, the majority problem *does* have an efficient algorithm if we remove any requirements from the scenario where no majority exists. The revised cases are as follows.

- **Majority exists.** If $M \in [n]$ appears more than half the time, then the algorithm must return $M$.

- **No majority exists.** If no such $M$ exists, then the algorithm *can return anything*.

This relaxation of the problem is somewhat analogous to allowing false positives, except that the majority problem is not a Yes/No problem. Researchers would call this revised problem a “promise problem”. The algorithm is required to solve the problem under the promise that a majority item exists. The algorithm has no requirements if the promise is violated, meaning that there is no majority item.

---

\(^1\)We’ll be happy with bounds involving $\log n$, $\log m$, $\log(nm)$, or even powers of those quantities.
Algorithm 12.1 The Boyer-Moore algorithm for finding a majority element.

1: function BoyerMoore
2: Set Jar $\leftarrow$ “empty” and Count $\leftarrow$ 0
3: for $t = 1, \ldots, m$ do
4:    Receive item $a_t$ from the stream
5:    if Jar = “empty” then
6:        Set Jar $\leftarrow a_t$ and Count $\leftarrow 1$ $\triangleright$ Case 1: Add item to empty jar
7:    else if Jar = $a_t$ then
8:        Increment Count $\triangleright$ Case 2: New item same as jar: add it to jar
9:    else
10:       Decrement Count $\triangleright$ Case 3: New item differs: remove an item from jar
11:       if Count = 0 then Jar $\leftarrow$ “empty”
12:    end if
13: end for
14: return Jar
15: end function

12.2.1 The Boyer-Moore algorithm

Remarkably this revised 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 12.1.

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 both a matter and an anti-matter particle, 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 know which the majority item is! So each item must annihilate any differing item.

Question 12.2.1. What is the space usage of this algorithm?

Answer.

$$O((\log n) + \log m) = O(\log nm).$$

Theorem 12.2.2. The BoyerMoore algorithm finds a majority item, assuming that one exists.

Proof. Let $M$ be any item. The amount by which it is ahead at time $t$ is

$$\text{Ahead}_t = (\text{number of occurrences of } M \text{ up to time } t) - (\text{number of other items up to time } t).$$

Note that Ahead$_t$ can be either positive or negative, depending on how many times $M$ has appeared up to time $t$.

Let Count$_t$ denote the value of the variable Count at time $t$. We claim that the following inequality holds.

$$\text{Ahead}_t \leq \begin{cases} \text{Count}_t & \text{(if Jar = } M) \\ -\text{Count}_t & \text{(if Jar } \neq M) \end{cases} \quad (12.2.1)$$
It clearly holds at time $t = 0$ because $\text{Ahead}_0 = \text{Count}_0 = 0$. At a time $t > 0$, there are two cases to consider.

- The next stream item is $M$. Then the left-hand side (i.e., $\text{Ahead}_t$) and right-hand side of (12.2.1) both increase. This preserves the inequality.
- The next stream item is not $M$. Then the left-hand side (i.e., $\text{Ahead}_t$) of (12.2.1) decreases, whereas the right-hand might increase or decrease. Again, this preserves the inequality.

Now assume that $M$ is a strict majority. Then at the end of the stream we must have $\text{Ahead}_m > 0$. Since $\text{Count}_t \geq 0$ always holds, then referring to (12.2.1) it is clear that we must be in the case $\text{Jar} = M$. 

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).

12.3 Heavy hitters

In Section 12.2 we saw the majority problem: deciding whether the data has a majority item. That is, does any item appear in the stream strictly more than a $1/2$ fraction of the time?

Today 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.

Formally, the algorithm will return a set $\text{Output}$ of items. For each item $h$:

- **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 because the algorithm can simply output everything, i.e., $\text{Output} = \{1, \ldots, n\}$! We will make the problem more realistic by additionally requiring that:

$|\text{Output}| \leq k - 1,$

because clearly there must be less than $k$ actual heavy hitters.

12.3.1 The Misra-Gries algorithm

We present an algorithm which is a generalization of the BOYERMOORE algorithm from Section 12.2. 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 $1/5$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 12.2 presents uses those ideas to solve the heavy hitters problem in the streaming setting.

Algorithm 12.2 The Misra-Gries algorithm for finding the heavy hitters.

1: function MisraGries
2: Let $C$ be an empty hash table
3: for $t = 1, \ldots, m$ do
4:   Receive item $a_t$ from the stream
5:   if $a_t \in C$ then
6:     Increment $C[a_t]$ \hfill $\triangleright$ Case 1: Boxer joins team in corner
7:   else if $|C| < k - 1$ then
8:     $C[a_t] \leftarrow 1$ \hfill $\triangleright$ Case 2: Boxer starts new corner
9:   else
10:     for $j \in C$ do \hfill $\triangleright$ Case 3: Boxer starts battle
11:     Decrement $C[j]$
12:     if $C[j] = 0$ then remove $j$ from $C$
13:   end for
14: end if
15: end for
16: return Keys($C$)
17: end function

Question 12.3.1. What is the space usage of this algorithm?

Answer.

The dictionary always has size at most $k - 1$. Each entry in the dictionary has a key, which takes $O \log n$ bits, and a count, which takes $O \log \frac{n}{k}$ bits, and a constant, which takes $O(1)$ bits. Thus, the total space is $O(n \log \frac{n}{k})$.

Theorem 12.3.2. The output of the MISRAGRES 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. We now present a more detailed technical proof. 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.

$$\text{Invariant 1: } (\# \text{ battles}) = \frac{1}{k} \left( t - \sum_{j=1}^{n} C[j] \right).$$

Let us see why this is preserved in every iteration $t$. 

© Nicholas Harvey 2021. All rights reserved.
• 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 $\frac{1}{k}(1+(k-1)) = 1$.

Next, consider any item $h$.

**Invariant 2:** 
\[
(# \text{ occurrences of } h) = C[h] + (# \text{ battles involving } h).
\]

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

\[
\frac{m}{k} < f_h
\]

\[
(h \text{ is a heavy hitter})
\]

\[
= (# \text{ occurrences of } h)
\]

\[
= C[h] + (# \text{ battles involving } h)
\]

\[
(\text{by Invariant 2})
\]

\[
\leq C[h] + (# \text{ battles})
\]

\[
= C[h] + \frac{1}{k}\left(m - \sum_{j=1}^{n} C[j]\right)
\]

\[
(\text{by Invariant 1 at time } t = m)
\]

\[
\leq C[h] + \frac{1}{k}m
\]

\[
(C \text{ is non-negative}).
\]

Rearranging, we obtain $C[h] > 0$, which implies that $h$ will be output. \qed

**Notes**

This is the Misra-Gries algorithm. See also LeetCode and (Sen and Kumar, 2019, Section 16.2).

### 12.4 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 \text{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 12.3, is to do something like the BasicFilter from Section 11.4, but augmented with counters.
**Algorithm 12.3** 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 $h : [n] \rightarrow [k]$ 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 $C[h(i)]$

**Theorem 12.4.1.** For every item $i \in [n]$,

- Lower bound: $f_i \leq \text{QUERY}(i)$ (12.4.1)
- Upper bound: $E[\text{QUERY}(i)] \leq f_i + \epsilon m$. (12.4.2)

**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

$$C[h(i)] = \sum_{b=1}^{n} X_b f_b = f_i + \sum_{b \neq i} X_b f_b \geq f_i.$$ (12.4.3)

Since each $f_b \geq 0$ we have $\text{QUERY}(i) = C[h(i)] \geq f_i$, which is (12.4.1).

Now we consider the other inequality. Since $h$ is a purely random function, we know from Question 11.1.3 that the collision probability is $E[X_b] = \Pr[h(i) = h(b)] = 1/k$. So

$$E[C[h(i)]] = E\left[f_i + \sum_{b \neq i} X_b f_b\right] \quad \text{ (expectation of (12.4.3))}$$ (12.4.4)

$$= f_i + \sum_{b \neq i} E[X_b] f_b \quad \text{ (linearity of expectation)}$$

$$= f_i + \frac{1}{k} \sum_{b \neq i} f_b \quad \text{ (collision probability)}$$

$$\leq f_i + \frac{m}{k} \quad \text{ (by definition of $k$)}$$

Thus $E[\text{QUERY}(i)] \leq f_i + \epsilon m$, which proves (12.4.2). \[\square\]
The CountingFilter is very simple, but we have only given an analysis in expectation. Reiterating our moral from Section 7.1:

The expectation of a random variable does not tell you everything.

Question 12.4.2. Use Markov’s inequality to give an upper bound on \( \Pr \left[ \text{Query}(i) \geq f_i + 2\epsilon m \right] \).

Answer.

\[
\frac{Z}{\epsilon} \geq \frac{\sum_{b=1}^{m} Z_{i,b}}{\epsilon} \geq \left[ \sum_{b=1}^{m} Z_{i,b} \right] \geq [\sum_{b=1}^{m} Z_{i,b} \geq 0] \geq \left[ \sum_{b=1}^{m} Z_{i,b} \geq f_i \right] \geq \frac{1}{\epsilon} \cdot \sum_{b=1}^{m} Z_{i,b} \geq \sum_{b=1}^{m} Z_{i,b} \geq f_i \geq f_i.
\]

How can we refine this algorithm so that it is very likely to give good estimates?

Keener Question 12.4.3. Thinking towards future topics, could we modify the algorithm to work if the stream could “delete” items as well as “inserting” them?

Question 12.4.4. 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

\[
O(k \log m) \text{ bits.}
\]

12.4.1 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.

Theorem 12.4.5.

Lower bound: \( f_i \leq \text{Query}(i) \quad \forall i \in [n]. \) \hspace{1cm} (12.4.5)

Furthermore, with probability at least 0.99,

Upper bound: \( \text{Query}(i) \leq f_i + 2\epsilon m \quad \forall i \in [n]. \) \hspace{1cm} (12.4.6)

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 (12.4.3), we have

\[
C[j, h_j(i)] = \sum_{b=1}^{n} X_b f_b = f_i + \sum_{b \neq i} X_b f_b \geq f_i.
\]

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 (12.4.5).

Now we consider the other inequality. The analysis of (12.4.4) shows that, for all \( j \in \{1, \ldots, \ell\} \),

\[
E[C[j, h_j(i)]] \leq f_i + \epsilon m
\]
Algorithm 12.4 The Count-Min Sketch for frequency estimation.

```plaintext
global int \( k = \lceil 1/\epsilon \rceil \)

global int \( \ell = \lceil \log(100) \rceil = 7 \)

class COUNTMINSKETCH:
    hash function \( h_1, \ldots, h_\ell \)
    array \( C \)

\[ \text{Build the data structure by processing the stream of items} \]

**CONSTRUCTOR ()**

- Let \( C[1..\ell, 1..k] \) be a two-dimensional array of integers, initially zero
- Let \( h_1, \ldots, h_\ell : [n] \rightarrow [k] \) be independent purely random functions
  
  for \( t = 1, \ldots, m \)
  
  Receive item \( a_t \) from the stream
  
  for \( j = 1, \ldots, \ell \)
  
  Increment \( C[j, h_j(a_t)] \)

\[ \text{Estimates the frequency of an item.} \]

**QUERY (int \( i \))**

- return \( \min \{ C[h_1(i)], \ldots, C[h_\ell(i)] \} \)
```

Now by the same argument as Question 12.4.2, we know that \( C[j, h_j(i)] - f_i \) is non-negative, so Markov’s inequality implies that

\[
\Pr \left[ C[j, h_j(i)] - f_i \geq 2\epsilon m \right] \leq \frac{E[C[j, h_j(i)] - f_i]}{2\epsilon m} \leq \frac{\epsilon m}{2\epsilon m} = \frac{1}{2}.
\]

Now we can use independence of the hash functions to drive down the failure probability:

\[
\Pr \left[ \text{QUERY}(i) - f_i \geq 2\epsilon m \right] = \Pr \left[ \min_{1 \leq j \leq \ell} C[j, h_j(i)] - f_i \geq 2\epsilon m \right]
\]

\[
= \prod_{1 \leq j \leq \ell} \Pr \left[ C[j, h_j(i)] - f_i \geq 2\epsilon m \right] \quad \text{(independence of the } h_j) \]

\[
\leq 2^{-\ell} = 0.01.
\]

This proves (12.4.6).

**Question 12.4.6.** How much space is used by this algorithm (ignoring the impractical hash functions)?

**Answer.**

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.
12.5 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 12.5.1.** Can a CountingFilter handle deletions?

**Answer.**

In the deletions model (general case), it is not hard to see that:

\[
\sum_i |f_i - \text{Query}(i)| \leq \epsilon m.
\]

In the deletions model (non-negative case), the analysis does not change at all, and we have:

\[
|f_i + \text{Query}(i)| \leq |f_i| - \text{Query}(i) \leq |f_i|
\]

**Question 12.5.2.** 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.

12.6 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.
Algorithm 12.5 The **CountMedSketch** algorithm modifies the **CountMinSketch** algorithm to work in the deletions model, general case.

```plaintext
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(\( C[h_1(i)], \ldots, C[h_\ell(i)] \))
```

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

\[
C[j, h_j(i)] = f_i + \text{(error due to colliding items)} \geq 0
\]

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

\[
C[j, h_j(i)] = f_i + \text{(error due to colliding items)}
\]

(can be \( \leq 0 \) 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 12.5 implements this idea.

**Question 12.6.1.** How much space is used by this algorithm (ignoring the impractical hash functions)?

**Answer.**

It is asymptotically the same as the **CountMinSketch** algorithm (plus the impractical hash functions).
12.6.1 Analysis

Theorem 12.6.2. For all $i \in [n]$, we have

$$\Pr \left[ |\text{QUERY}(i) - f_i| > 3\epsilon m \right] \leq 0.01.$$

The main difference from Theorem 12.4.5 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 12.4.5, except that it must deal with absolute values, and it uses Theorem 8.5.3 to analyze the estimate.

Proof of Theorem 12.6.2. Fix some item $i \in [n]$, and consider the $j^{th}$ estimate, for some $j \in [\ell]$. As in Theorem 12.4.1, we let $X_b$ be the indicator of the event “$h_j(i) = h_j(b)$”, and we recall that $E[X_b] = 1/k \leq \epsilon$. The $j^{th}$ estimate has the form

$$C[j, h_j(i)] = \sum_{b=1}^{n} X_b f_b = f_i + \sum_{b \neq i} X_b f_b.$$ 

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:

$$|C[j, h_j(i)] - f_i| = \left| \sum_{b \neq i} X_b f_b \right| \leq \sum_{b \neq i} X_b \cdot |f_b|,$$

by the triangle inequality (Fact A.2.4). Now we consider the expected error, and use linearity of expectation.

$$E \left[ |C[j, h_j(i)] - f_i| \right] \leq \sum_{b \neq i} E[X_b] \cdot |f_b| \leq \epsilon \sum_{b \neq i} |f_b|$$

Note that $|f_b| \leq (\# \text{ insertions or deletions of item } b)$, and therefore $\sum_{b=1}^{n} |f_b| \leq m$. This yields our key bound on the expected error.

$$E \left[ |C[j, h_j(i)] - f_i| \right] \leq \epsilon m \quad (12.6.1)$$

Next, following Theorem 12.4.5, we analyze the probability that the error is slightly larger.

$$\Pr \left[ |C[j, h_j(i)] - f_i| \geq 3\epsilon m \right] \leq \frac{E \left[ |C[j, h_j(i)] - f_i| \right]}{3\epsilon m} \quad (\text{by Markov’s inequality})$$

$$\leq \frac{\epsilon m}{3\epsilon m} = \frac{1}{3} \quad (\text{by (12.6.1)})$$

Writing out the two tails separately, we have

Right tail: $$\Pr \left[ C[j, h_j(i)] \geq f_i + 3\epsilon m \right] \leq \frac{1}{3} \quad (12.6.2)$$

Left tail: $$\Pr \left[ C[j, h_j(i)] \leq f_i - 3\epsilon m \right] \leq \frac{1}{3}.$$

Now we are perfectly set up to analyze the median estimator using Theorem 8.5.3. Transitioning to the notation of that theorem, we have

$$Z_j = C[j, h_j(i)], \quad \ell = 97, \quad L = f_i - 3\epsilon m, \quad R = f_i + 3\epsilon m, \quad \alpha = 1/6.$$
Using this new notation, (12.6.2) may be restated as

\[ \Pr[Z_j > R] \leq \frac{1}{2} - \alpha \]
\[ \Pr[Z_j < L] \leq \frac{1}{2} - \alpha. \]

This yields the following bound on the error of Query.

\[ \Pr[|\text{QUERY}(i) - f_i| > 3\epsilon m] = \Pr[\text{QUERY}(i) < f_i - 3\epsilon m \text{ or } \text{QUERY}(i) > f_i + 3\epsilon m] \]
\[ = \Pr[\text{Median}(Z_1, \ldots, Z_\ell) < L \text{ or } R] \]
\[ \leq 2 \exp\left(-2(1/6)^2.97\right) \quad \text{(by Theorem 8.5.3)} \]
\[ < 0.01, \]

by a quick numerical calculation.

### 12.7 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 multisets. Recall that the MinHash Set data structure (Algorithm 11.2) represents a set \( S \) using constant space, whereas a Bloom filter uses linear space. Since the MinHash Set is so space efficient, it would be wonderful if it could be adapted to estimate the size of a set.

Following Section 11.2, 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 \( d \)?

#### 12.7.1 The minimum of \( d \) uniform random variables

The minimum of \( d \) uniform RVs has been well studied. For example, it is known that

\[ \mathbb{E}[Z] = \frac{1}{d+1}. \]

**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?

$$E\left[\hat{d}\right] = E\left[\frac{1}{Z} - 1\right] = \frac{1}{E[Z]} - 1 = \frac{1}{1/(d+1)} - 1 = d.$$  

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 7.1:

The expectation of a random variable does not tell you everything.

Intuitively, if we could somehow estimate $E[Z]$ 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 12.6, 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 8.6.1). 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 8.5.3 is a powerful tool, and it was useful in Section 12.6, so we will employ that approach again in this section.

### 12.7.2 The algorithm

Pseudocode for our approach based on medians is shown in Algorithm 12.6. 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 12.7.1.** The true cardinality $d$ is contained in the interval $[d_{\min}, d_{\max}]$ computed by $\text{EstimateCardinality}$ with probability at least 0.99. Moreover, the interval is rather small: $d_{\max}$ is less than 10% bigger than $d_{\min}$.

**Question 12.7.2.** How much space does this algorithm use (ignoring the purely random functions)?

**Answer.**

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.

---

\(^2\)This equation is very far from true because, in fact, $E\left[\hat{d}\right] = \infty$; see equation (2) in this paper.
Algorithm 12.6 An algorithm for estimating the number of distinct elements in a data stream.

```plaintext
global float α ← 0.0165
global int ℓ ← ⌈ln(200)/2α^2⌉ = 9731
```

class DISTINCTELEMENTS:
hash functions \( h_1, \ldots, h_\ell \)
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
- Let \( h_1, \ldots, h_\ell : [n] \to [0, 1] \) be purely random functions
- 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 \([d_{\text{min}}, d_{\text{max}}]\)

**EstimateCardinality()**
- Compute \( M = \text{Median}(Z[1], \ldots, Z[\ell]) \)
- Compute \( d_{\text{min}} = \frac{\ln(0.5+\alpha)}{\ln(1-M)} \) and \( d_{\text{max}} = \frac{\ln(0.5-\alpha)}{\ln(1-M)} \)
- return \([d_{\text{min}}, d_{\text{max}}]\)

12.7.3 Analysis

**Analyzing the tails of \( Z \).** In order to apply the median estimator, the first step is to satisfy the inequalities \((8.5.1)\). To do this, we consider the random variable

\[
Z = \min_{i \in [d]} U_i
\]  

(12.7.1)

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.

Left threshold: \( L = 1 - \left( \frac{1}{2} + \alpha \right)^{1/d} \)

Right threshold: \( R = 1 - \left( \frac{1}{2} - \alpha \right)^{1/d} \).

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 \( L \lesssim \frac{\ln^2 d}{d} \lesssim R \). Their precise definition is chosen to make the next lemma work smoothly.

**Lemma 12.7.3.** Let \( U_1, \ldots, U_d \) be independent random variables, uniform on \([0, 1]\). Then

Left tail: \( \Pr \left[ \min_{i \in [d]} U_i < L \right] = \frac{1}{2} - \alpha \)

Right tail: \( \Pr \left[ \min_{i \in [d]} U_i > R \right] = \frac{1}{2} - \alpha. \)
Proof. First we analyze the right tail:

$$\Pr \left[ \min_{i \in [d]} U_i > R \right] = (1 - R)^d = \left( \left( \frac{1}{2} - \alpha \right)^{1/d} \right)^d = \frac{1}{2} - \alpha$$

by Exercise A.5 and the definition of $R$. The analysis of the left tail is similar:

$$1 - \Pr \left[ \min_{i \in [d]} U_i < L \right] = \Pr \left[ \min_{i \in [d]} U_i > L \right] = \prod_{i \in [d]} \Pr \left[ U_i > L \right] = (1 - L)^d = \frac{1}{2} + \alpha.$$

Rearranging, we conclude that $\Pr \left[ \min_{i \in [d]} U_i < L \right] = \frac{1}{2} - \alpha$.

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 $\ell$ 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 (12.7.1). 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 12.7.4.** Define $\ell = \lceil \ln(200)/2\alpha^2 \rceil$, perhaps adding 1 to ensure that $\ell$ is odd. Then

$$\Pr \left[ L \leq M \leq R \right] \geq 0.99.$$

**Proof.** Directly applying Theorem 8.5.3, we have

$$\Pr \left[ L \leq M \leq R \right] \geq 1 - 2\exp(-2\alpha^2 \ell) \geq 1 - 2\exp(-\ln(200)) = 0.99. \quad \Box$$

**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 $[d_{\text{min}}, d_{\text{max}}]$ that is likely to contain the true cardinality $d$. This interval’s endpoints are

$$d_{\text{min}} = \frac{\ln \left( \frac{1}{2} + \alpha \right)}{\ln(1 - M)} \quad \text{and} \quad d_{\text{max}} = \frac{\ln \left( \frac{1}{2} - \alpha \right)}{\ln(1 - M)}.$$

Note that $d_{\text{min}}$ and $d_{\text{max}}$ are actually random variables, because they depend on $M$.

Much like $L$ and $R$ had mysterious definitions, $d_{\text{min}}$ and $d_{\text{max}}$ 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 $d_{\text{min}}$ and $d_{\text{max}}$.

**Claim 12.7.5.** The event “$L \leq M \leq R$” is equivalent to the event “$d_{\text{min}} \leq d \leq d_{\text{max}}$”.

---

3Although it might not appear that $d_{\text{min}} < d_{\text{max}}$, that inequality does hold because both the numerators and the denominators are negative.
Proof. The proof is just a sequence of routine algebraic manipulations.

\[ L \leq M \leq R \]

\[ \iff \left( \frac{1}{2} + \alpha \right)^{1/d} \geq 1 - M \geq \left( \frac{1}{2} - \alpha \right)^{1/d} \quad \text{(subtracting from 1)} \]

\[ \iff \left( \frac{1}{2} + \alpha \right) \geq (1 - M)^d \geq \left( \frac{1}{2} - \alpha \right) \quad \text{(raising to power } d) \]

\[ \iff \ln \left( \frac{1}{2} + \alpha \right) \geq d \ln(1 - M) \geq \ln \left( \frac{1}{2} - \alpha \right) \quad \text{(taking log)} \]

\[ \iff \frac{\ln(1 - M)}{d_{\min}} \leq d \leq \frac{\ln(1 - M)}{d_{\max}} \quad \text{(dividing by } \ln(1 - M)) \]

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 12.7.1. We have

\[ \Pr [ d_{\min} \leq d \leq d_{\max} ] = \Pr [ L \leq M \leq R ] \quad \text{(by Claim 12.7.5)} \]

\[ \geq 0.99 \quad \text{(by Claim 12.7.4)} \]

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 \(\alpha = 0.0165\). \(\blacksquare\)

Keener Kwestion 12.7.6. Algorithm 12.6 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 \(\alpha\) and \(\ell\)?

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 \(O\left( \frac{1}{\epsilon^2} + \log n \right)\) bits of space, even when using implementable hash functions with provable guarantees. There is a matching lower bound of \(\Omega\left( \frac{1}{\epsilon^2} + \log n \right)\) bits. This optimal algorithm is due to Daniel Kane, Jelani Nelson, and David P. Woodruff.
12.8 Exercises

Exercise 12.1 Frequency estimation. Design a deterministic streaming algorithm with the following behavior. Given any parameter \( \epsilon \), 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 \( \text{QUERY}(i) \) which takes an integer \( i \in \{1, \ldots, n\} \) and satisfies
\[
f_i - \epsilon m \leq \text{QUERY}(i) \leq f_i.
\]
Prove that your approach satisfies these guarantees.

Exercise 12.2 Majority Revisited. 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 \( O(\log(n) \log(m)) \) bits of space.

Hint: Consider the binary representation of the items.

Exercise 12.3. Define \( \text{Excl}_i \) to be the the sum of the absolute values of the item frequencies, excluding item \( i \):
\[
\text{Excl}_i = \sum_{j \neq i} |f_j|.
\]

Part I. Show that \( \text{Excl}_i \leq m \).

Part II. Improve Theorem 12.6.2 to show that, for all \( i \in [n] \), we have
\[
\Pr[|\text{QUERY}(i) - f_i| > 3\epsilon \text{Excl}_i] \leq 0.01.
\]

Exercise 12.4 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 \textsc{BoyerMoore} algorithm to monitor all its internet traffic. Separately, Server B also runs the \textsc{BoyerMoore} algorithm to monitor all its internet traffic. The algorithm’s states on these respective servers are denoted \((\text{Jar}_A, \text{Count}_A)\) and \((\text{Jar}_B, \text{Count}_B)\). At the end of the day, the four values \( \text{Jar}_A, \text{Count}_A, \text{Jar}_B, \text{Count}_B \) 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 13

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, as in 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.
13.1 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 just does

```c
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

\[
\text{Hash of a single character } a: \quad (x \cdot a + y) \mod p.
\]

Now, we don’t want the \(\mod\) operation to lose any valid characters in the output, so 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.

Since we want to hash \(\text{strings}\), we will hash each character separately, then just add up the results. Formally, 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

\[
\text{Hash of a string } a: \quad h(a) = \left( \sum_{i=1}^{s} a_i X_i + Y \right) \mod p.
\]  

(13.1.1)

The design of this hash function is called linear hashing. We will show that linear hashing has several nice properties.

**Definition 13.1.1.** 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

\[
\Pr\left[ h(a) = \alpha \land h(b) = \beta \right] = \frac{1}{p^2}
\]

(13.1.2)

for every two different strings \(a, b\) of length \(s\), and every two characters \(\alpha, \beta \in [p]\).

**Theorem 13.1.2.** Let \(h\) be a linear hash function, as in (13.1.1), 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 (13.1.1).

**Question 13.1.3.** Is there a property of purely random functions that is not satisfied by linear hashing?

\(^1\)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”.

©Nicholas Harvey 2021. All rights reserved.
Answer.

For every three distinct input strings \( a, b, c \) of length \( s \), and any characters \( \alpha, \beta, \gamma \), a purely random function \( h \) satisfies

\[
\Pr \left[ h(a) = \alpha \land h(b) = \beta \land h(c) = \gamma \right] = \frac{1}{p^3}.
\]

This is not satisfied by our linear hash function \( h \).

13.1.4 Question. 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.

13.1.1 Consequences

Corollary 13.1.5 (Hashes are uniform). Let \( h \) have the form in (13.1.1), 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 \( \alpha \),

\[
\Pr \left[ h(a) = \alpha \right] = \frac{1}{p}.
\]

Question 13.1.6. Do you see how to prove Corollary 13.1.5?

Answer.

By the law of total probability (Fact A.3.6),

\[
\Pr \left[ h(a) = \alpha \right] = \sum_{\beta} \Pr \left[ h(a) = \alpha \land h(b) = \beta \right] = \frac{1}{p}.
\]

Corollary 13.1.7 (Collisions are unlikely). Assume that \( h \) has the form in (13.1.1), where \( X_1, \ldots, X_s \) and \( Y \) are chosen uniformly and independently at random. For any different strings \( a, b \) of length \( s \),

\[
\Pr \left[ h(a) = h(b) \right] = \frac{1}{p}.
\]

Proof. By the law of total probability (Fact A.3.6),

\[
\Pr \left[ h(a) = h(y) \right] = \sum_{a \in [p]} \Pr \left[ h(a) = h(b) \land h(a) = \alpha \right] = \sum_{a \in [p]} \frac{1}{p^2} = \frac{1}{p}.
\]

13.1.2 Proof of Theorem

To prove the theorem, we will use the following fact.

Fact A.2.15 (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

\[
\begin{align*}
(aX + Y) \mod p &= c \\
(bX + Y) \mod p &= d
\end{align*}
\]

\( X, Y \in [p] \).
Proof of Theorem 13.1.2. 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 $a_s \neq b_s$.

Recalling (13.1.2), we must analyze the probability that

$$\left( \sum_{i=1}^{s} a_i X_i + Y \right) \mod p = \alpha$$

$$\left( \sum_{i=1}^{s} b_i X_i + Y \right) \mod p = \beta.$$ 

Moving most of the sum to the right-hand side, these equations are equivalent to

$$(a_s X_s + Y) \mod p = \left( \alpha - \sum_{i=1}^{s-1} a_i X_i \right) \mod p$$

$$(b_s X_s + Y) \mod p = \left( \beta - \sum_{i=1}^{s-1} b_i X_i \right) \mod p.$$ 

(13.1.3)

The key point is: no matter what values are on the right-hand side, Fact A.2.15 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,

$$\Pr \left[ h(a) = \alpha \land h(b) = \beta \right] = \Pr \left[ \text{equations (13.1.3) hold} \right] = \frac{\# \text{ solutions } (X_s, Y)}{\# \text{ values for the pair } (X_s, Y)} = \frac{1}{p^2},$$

since Fact A.2.15 says that there is a unique solution.

### 13.2 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 make be helpful to think how the arithmetic operations modulo 2 correspond to familiar Boolean operations from CPSC 121.

- **Multiplication:** If $a, b \in \{0, 1\}$ then the multiplication $ab \mod 2$ is just the Boolean And operation. This can be checked exhaustively.

<table>
<thead>
<tr>
<th>$0 \cdot 0$ mod 2 = 0</th>
<th>$0 \cdot 1$ mod 2 = 0</th>
<th>$1 \cdot 0$ mod 2 = 0</th>
<th>$1 \cdot 1$ mod 2 = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \land 0 = 0$</td>
<td>$0 \land 1 = 0$</td>
<td>$1 \land 0 = 0$</td>
<td>$1 \land 1 = 1$</td>
</tr>
</tbody>
</table>
• **Addition:** If \( a, b \in \{0, 1\} \) then the addition \((a + b) \mod 2\) is just the Boolean Xor operation. This can be checked exhaustively.

\[
\begin{align*}
(0 + 0) \mod 2 &= 0 & (0 + 1) \mod 2 &= 1 & (1 + 0) \mod 2 &= 1 & (1 + 1) \mod 2 &= 0 \\
0 \oplus 0 &= 0 & 0 \oplus 1 &= 1 & 1 \oplus 0 &= 1 & 1 \oplus 1 &= 0
\end{align*}
\]

In these equations above, the only place where the \( \mod 2 \) plays any role is \((1 + 1) \mod 2 = 0\).

**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 1.1.1 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 (13.1.1) with \( p = 2 \), so it outputs a single bit. More explicitly, each \( h_j \) is of the form

\[
h_j(a) = \left( \sum_{i=1}^{s} a_i X_i + Y \right) \mod 2,
\]

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 13.2.1.** 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:

\[
\begin{align*}
h(a) &= h_1(a)h_2(a) \cdots h_{\lg k}(a) \\
h(b) &= h_1(b)h_2(b) \cdots h_{\lg k}(b)
\end{align*}
\]

Let \( \alpha \) and \( \beta \) be arbitrary integers in \([k]\), whose binary representations are written

\[
\begin{align*}
\alpha &= \alpha_1\alpha_2 \cdots \alpha_{\lg k} \\
\beta &= \beta_1\beta_2 \cdots \beta_{\lg k}.
\end{align*}
\]

Then

\[
\Pr[ h(a) = \alpha \land h(b) = \beta ] = \prod_{j=1}^{\lg k} \Pr[ h_j(a) = \alpha_j \land h_j(b) = \beta_j ] \quad \text{(bitwise equality)}
\]

\[
= \prod_{j=1}^{\lg k} \frac{1}{2^2} \quad \text{(each } h_j \text{ satisfies (13.1.2) with } p = 2) \\
= \frac{1}{2^{2\lg k}} = \frac{1}{k^2}.
\]

This shows that \( h \) satisfies the condition (13.1.2) with \( k \) instead of \( p \), and therefore \( h \) is strongly universal.
**Question 13.2.2.** 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.**

Since there are \( k \) such functions, the total is \( k \cdot (1 + s) \).

Returning to (13.2.1), each hash function \( h \) can be represented using the \( s + 1 \) random bits \( X_1^{a_1} \cdots X_1^{a_{s+1}} \).

**Question 13.2.3.** What is the collision probability for \( h \)?

**Answer.**

By the same argument as Corollary 13.1.7,

\[
\Pr \left[ h(a) = h(b) \right] = \sum_{\varnothing \subseteq \varnothing} \Pr \left[ h(a) = \varnothing \right] \cdot \Pr \left[ h(b) = \varnothing \right] = \frac{1}{k}
\]

Since \( h \) is strongly universal, we have

**Keener Kwestion 13.2.4.** How would you implement this in your favourite programming language (C, Java, etc.)?

### 13.3 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

\[
h(a) = \left( \sum_{i=1}^{s} a_i X^i \right) \mod p.
\]

The key analysis of a polynomial hash function is the following. Note that the theorem says nothing unless \( p > s \).

**Theorem 13.3.1** (Collision probability). Let \( a, b \) be different bit strings \( a, b \) of length \( s \). If \( X \) is chosen uniformly at random in \([p]\) then

\[
\Pr \left[ h(a) = h(b) \right] \leq \frac{s}{p}.
\]

**Question 13.3.2.** 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.**

Remarkably, the amount of space does not depend on \( s \). Since \( h \) can be represented using \( \log p \) bits, and \( [d] \subseteq X \),

Let us now compare these properties of the polynomial hash to the linear hash.
### 13.3.1 Analysis

To prove Theorem 13.3.1 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.2.13** (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 $c_i \mod p \neq 0$. Then there are at most $d$ solutions to $g(x) \mod p = 0$ and $x \in [p]$.

Such solutions are usually called roots.

**Proof of Theorem 13.3.1.** We will work with the polynomial

$$g(X) = \sum_{i=1}^{s} a_i X^i - \sum_{i=1}^{s} b_i X^i = \sum_{i=1}^{s} (a_i - b_i) X^i.$$ 

Taking the mod, we have

$$g(X) \mod p = (h(a) - h(b)) \mod p.$$ 

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, $c_i \mod p \neq 0$. Since the conditions of Fact A.2.13 are satisfied, we have

$$\Pr [ h(a) = h(b) ] = \Pr [ g(X) \mod p = 0 ]$$

(by (13.3.1))

$$= \frac{\text{number of solutions}}{\text{number of choices for } X}$$

(since $X$ is uniform)

$$\leq \frac{s}{p}$$

(by Fact A.2.13).

### Notes

There are various constructions of low-space hash functions in the literature. The constructions that we have presented are as follows.

- **Section 13.1**: (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 13.2: Binary linear hashing. This is also known as matrix multiplication hashing. It can be found in (Motwani and Raghavan, 1995, Exercise 8.21).

• Section 13.3: 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.
Chapter 14

Applications of Low-Space Hashing

14.1 Equality testing

Zoodle hosts a website at which users can download the genetic sequence of many different vegetables. A single sequence can be many gigabytes long\textsuperscript{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 11.1, 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.

\textsuperscript{1}For example, the genome of the pea (*pisum sativum*) is approximately 4 gigabytes.
Algorithm 14.1 Equality testing using polynomial hashing.

1: function Server(bit string \(a = (a_1, \ldots, a_s)\))
2: By brute force, find a prime \(p \in \{2s, \ldots, 4s\}\)
3: Pick \(x \in \{p\}\) uniformly at random
4: Compute \(h(a) = \sum_{i=1}^{s} a_i x^i \mod p\)
5: Send \(p, x\) and \(h(a)\) to the client
6: end function

7: function Client(bit string \(b = (b_1, \ldots, b_s)\))
8: Receive \(p, x\) and \(h(a)\) from the server
9: Compute \(h(b) = \sum_{i=1}^{s} b_i x^i \mod p\)
10: if \(h(a) = h(b)\) then
11: return True
12: else
13: return False
14: end if
15: 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.

14.1.1 A randomized solution

Instead of these deterministic hash functions, like MD5, we will use the polynomial hashing method of Section 13.3. 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 14.1. 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.2.11).

Question 14.1.1. How can the server find such a prime \(p\)? How much time does it take?

Answer.

The simplest method is to try all divisors of \(\{s+1, s+2, \ldots, 2s\}\). If this takes time \(O(d)\), where \(d\) is a divisor of \(p\), then there are faster methods, such as the sieve of Eratosthenes. This takes time \(O\left(\sqrt{p}\log p\right)\), which is faster.

14.1.2 Analysis

Theorem 14.1.2. 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 13.3.1, we have

\[
\Pr[\text{client incorrectly outputs True}] = \Pr[h(a) = h(b)] \\
\leq \frac{s}{p} \quad \text{(by Theorem 13.3.1)} \\
\leq \frac{s}{2s} = \frac{1}{2},
\]

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 3.3. If perform \( \ell \) independent trials of the algorithm then, by Theorem 3.3.1, the false positive probability decreases to \( 2^{-\ell} \).

Notes

The number of bits communicated by Algorithm 14.1 is optimal, up to constant factors. Theorem 14.1.2 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).

14.1.3 Exercises

Exercise 14.1. Using the amplification approach described above, the equality test satisfies

\[
\begin{align*}
\text{False positive probability:} & \leq 2^{-\ell} \\
\text{Bits communicated:} & = O(\ell \log s).
\end{align*}
\]

This can be improved. Design a different approach achieving

\[
\begin{align*}
\text{False positive probability:} & \leq 2^{-\ell} \\
\text{Bits communicated:} & = O(\ell + \log s).
\end{align*}
\]

This new approach has the same \( 2^{-\ell} \) bound on false positives but communicates fewer bits because it depends additively rather than multiplicatively on \( \ell \).

Exercise 14.2. One complication with Algorithm 14.1 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 14.2 An algorithm to test $a$ equals $b$. Assume $s \geq 2$.

1: function SERVER(bit string $a = (a_1, \ldots, a_s)$)
2:  for $j = 1, \ldots, \lceil 32 \ln s \rceil$ do
3:      Pick a random number $p \in \{4s + 1, \ldots, 20s\}$
4:      Pick $x \in \{0, \ldots, p - 1\}$ uniformly at random
5:      Compute $h(a) = \sum_{i=1}^{s} a_i x^i \mod p$
6:      Send $p$, $x$ and $h(a)$ to the client
7:  end for
8: end function

9: function CLIENT(bit string $b = (b_1, \ldots, b_s)$)
10:  for $j = 1, \ldots, \lceil 32 \ln s \rceil$ do
11:      Receive $p$, $x$ and $h(a)$ from the server
12:      Compute $h(b) = \sum_{i=1}^{s} b_i x^i \mod p$
13:      if $h(a) \neq h(b)$ then return False
14:  end for
15: return True
16: 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 $E$ be the event that, in the first good iteration we have $h(a) = h(b)$, or that no good iteration exists. Prove that $\Pr [E] \leq 1/2$.

Part IV. Explain why the algorithm has false positive rate at most $1/2$.

14.2 Count-Min Sketch

As another application of low-space hashing, let us consider the COUNTINGFILTER and COUNTMINSKETCH algorithms from Chapter 12. 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 12.4.1. We defined $X_b$ to be the indicator of the event “$h(i) = h(b)$”. The key step of the analysis is

$$
E [C[h(i)]] = E \left[ f_i + \sum_{b \neq i} X_b f_b \right] = f_i + \sum_{b \neq i} E [X_b] f_b = f_i + \frac{1}{k} \sum_{b \neq i} f_b.
$$

Here we have used that

$$
E [X_b] = \Pr [h(i) = h(b)] = \frac{1}{k}, \quad \forall b \neq i,
$$

since we assume that $h$ is a purely random function.
Algorithm 14.3 The COUNTINGFILTER, revised to use a low-space hash function.

global int \( k = 2^\lceil \log(1/\epsilon) \rceil \)

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 \( h : [n] \to [k] \) 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 \( 2^\lceil \log(1/\epsilon) \rceil \).

The theorem analyzing this algorithm is identical to before.

**Theorem 14.2.1.** For every item \( i \in [n] \),

\[
\begin{align*}
\text{Lower bound:} & \quad f_i \leq \text{QUERY}(i) \\
\text{Upper bound:} & \quad \mathbb{E}[\text{QUERY}(i)] \leq f_i + \epsilon m.
\end{align*}
\]

In fact, even the proof is identical to before. We are still using the fact, as above, that

\[
\mathbb{E}[X_b] = \Pr[h(i) = h(b)] = \frac{1}{k} \quad \forall b \neq i.
\]

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 13.2.3 for this calculation.

**Question 14.2.2.** How much space does the COUNTINGFILTER algorithm now take?

### 14.3 Maximum cut via hashing

Recall the problem of finding a maximum cut in a graph, from Section 15.1. 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 $\text{OPT}$ denotes the size of the maximum cut. We showed that a uniformly random set $U$ cuts at least $\text{OPT}/2$ edges in expectation.

It turns out that we can achieve the same guarantee using a hash function to generate $U$. Why would you 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 $\left[ n \right]$. 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 $a \in \left[ n \right]$ with bit representation $a = (a_1, \ldots, a_s)$, the hash value is

$$h(a) = \left( \sum_{i=1}^{s} a_i X_i + Y \right) \mod 2.$$

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,

$$U = \{ a \in \left[ n \right] : h(a) = 0 \}.$$

Algorithm 14.4 This algorithm gives a 1/2-approximation to the maximum cut via universal hashing.

1. function NewMaxCut(graph $G = (V, E)$)
2. Let $n \leftarrow |V|$ and $s \leftarrow \lceil \lg n \rceil$
3. Label the vertices by the integers $\left[ n \right]$
4. Randomly choose mixing parameters $X_1, \ldots, X_s, Y \in \{0, 1\}$
5. Create the binary linear hash function $h$ with mixing parameters $X_1, \ldots, X_s, Y$
6. return $U = \{ i \in \left[ n \right] : h(i) = 0 \}$
7. end function

Theorem 14.3.1. Let $U$ be the output of NewMaxCut(). Then $E[|\delta(U)|] \geq m/2 \geq \text{OPT}/2$.

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 15.1,

$$E[|\delta(U)|] = \sum_{uv \in E} \Pr[uv \text{ is cut}] = \frac{|E|}{2} \geq \frac{\text{OPT}}{2}. \qed$$

14.3.1 A deterministic algorithm

NewMaxCut() randomly generates a cut $U$ satisfying $E[|\delta(U)|] \geq \text{OPT}/2$. However, this gives no guarantee of finding a large cut. In Section 15.1.1, we gave an algorithm with the different guarantee $Pr[|\delta(U)| > 0.49 \cdot \text{OPT}] \geq 0.01$, but this is still rather weak.

Today we will give an ironclad guarantee: there is a deterministic algorithm producing a cut $U$ with $|\delta(U)| \geq \text{OPT}/2$. 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 14.5 This is deterministic algorithm gives 1/2-approximation to the maximum cut. It does not use any randomness whatsoever!

1: function `DeterministicMaxCut(graph G = (V, E))`
2: Let \( n \leftarrow |V|, m \leftarrow |E|, \) and \( s \leftarrow \lceil \log n \rceil \)
3: Label the vertices by the integers \([n]\)
4: for all mixing parameters \(X_1, \ldots, X_s, Y \in \{0, 1\}\) do
5:  Create universal hash function \(h\) with mixing parameters \(X_1, \ldots, X_s, Y\)
6:  Let \(U = \{i \in [n] : h(i) = 0\}\)
7:  if \(|\delta(U)| \geq m/2\) then return \(U\)
8: end for
9: end function

Theorem 14.3.2. The `DeterministicMaxCut` algorithm produces a cut with \(|U| \geq m/2 \geq \text{OPT}/2\).

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 \(\mathbb{E}[|\delta(U)|] < m/2\). This contradicts Theorem 14.3.1.

Question 14.3.3. What is the runtime of this algorithm?

Answer. \(O(n)\) time to build and \(O(n)\) time to calculate \(\delta(U)\), \(O(s)\) to build hash function \(h\), \(O(1)\) time to calculate \(|\delta(U)|\), \(O(s)\) time per loop iteration, \(O(1)\) time to build hash function for loop iteration, \(O(1)\) time to return.

14.3.2 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 14.3.4. Let \(X\) be a random variable on a finite probability space. Suppose that \(\mathbb{E}[X] \geq k\). 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.

14.4 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 14.6 The BirthdayTable is a static data structure to store a list of *n* keys using $O(n^2)$ space, and with constant-time queries.

```java
class BirthdayTable:
    hash function *h*
    array *T*

    Constructor ( array *Keys* of ints, with entries indexed by $[n]$)
        Assume we know a prime $p \in \{n^2, \ldots, 2n^2\}$
        repeat
            Randomly choose a linear hash function *h* mapping ints to $[p]$
            Create array *T*, whose entries are indexed by $[p]$
            for $i \in [n]$
                if $T[h(Keys[i])]$ not empty then found collision
                $T[h(Keys[i])] \leftarrow Keys[i]$
        until no collisions

    Query (int *key*)
        if $T[h(key)] = key$ then return True
        else return False
```

14.4.1 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 14.4.1.**

- *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.** Let $X_{i,j}$ be the indicator of the event “$h(Keys[i]) = h(Keys[j])$”. In Corollary 13.1.7 we saw that the collision probability is $E[X_{i,j}] = \Pr[h(Keys[i]) = h(Keys[j])] = 1/p$. Let $X = \sum_{0 \leq i < j < n} X_{i,j}$ be the total number of collisions. Then, just like in (5.1.2),

$$E[X] = \sum_{0 \leq i < j < n} E[X_{i,j}] = \binom{n}{2} \cdot \frac{1}{p} < \frac{n^2}{2} \cdot \frac{1}{n^2} = \frac{1}{2}.$$  

By Markov’s inequality, $\Pr[X \geq 1] \leq \frac{E[X]}{1} < 1/2$. Taking the complement, the probability of no collisions is $\Pr[X = 0] > 1/2$.

**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 requires $O(p) = O(n^2)$ 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.  

© Nicholas Harvey 2021. All rights reserved.
Algorithm 14.7 The **PerfectTable** is a static data structure to store a list of $n$ keys using $O(n)$ space, and with constant-time queries.

```java
class PerfectTable:
    hash function $h$
    array $T$

    Constructor (array Keys of ints, with entries indexed by $[n]$)
    Assume we know a prime $p \in \{n, \ldots, 2n\}$
    Randomly choose a linear hash function $h$ mapping ints to $[p]$
    $\triangleright$ Temporarily hash keys into linked lists
    Create temporary array $L$, indexed by $[p]$, of linked lists
    for $i \in [n]$
        Insert $\text{Keys}[i]$ into $L[h(\text{Keys}[i])]$
    $\triangleright$ Replace each linked list with a BirthdayTable
    Create array $T$ of BirthdayTable objects, whose entries are indexed by $[p]$
    for $i \in [p]$
        $T[i].\text{Constructor}(L[i])$

    Query (int key)
    return $T[h(\text{key})].\text{Query}(\text{key})$
```

Keener Kwestion 14.4.2. 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?

### 14.4.2 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 14.4.3.**

- **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, according to Section 5.5, 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.
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

\[
n_i = \sum_{j \in [n]} X_j.
\]

(14.4.1)

Armed with this decomposition, we can analyze \( n_i^2 \):

\[
\begin{align*}
E[n_i^2] &= E \left( \left( \sum_{j \in [n]} X_j \right) \left( \sum_{\ell \in [n]} X_{\ell} \right) \right) \\
&= \sum_{j, \ell \in [n]} E[X_j X_{\ell}] \quad \text{(by linearity of expectation)} \\
&= \sum_{j, \ell \in [n]} \Pr[h(\text{Keys}[j]) = i = h(\text{Keys}[\ell])] + \sum_{j \in [n]} \Pr[h(\text{Keys}[j]) = i] \\
&= \sum_{j, \ell \in [n]} \frac{1}{p^2} + \sum_{j \in [n]} \frac{1}{p} \quad \text{(since \( h \) is (strongly) universal)} \\
&< \frac{n^2}{p^2} + \frac{n}{p} \leq 2,
\end{align*}
\]

(14.4.2)

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

\[
E \left[ \sum_{i \in [p]} \text{size of } i^{\text{th}} \text{ BirthdayTable} \right] \leq \sum_{i \in [p]} E[n_i^2] < \sum_{i \in [p]} 2c = O(p),
\]

(14.4.3)

by (14.4.2). Since \( p \leq 2n \), 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 (14.4.3), the expected runtime is \( O(\sum_{i \in [p]} n_i^2) = O(p) = O(n) \).

Question 14.4.4. 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

\[
\sum_{i \in [\lceil \sqrt{\frac{n}{2}} \rceil]} n_i^2 \geq \text{too big.}
\]

In the proof of Theorem 14.4.1, the expected number of iterations is

\[
\mathbb{E}\left(\text{repeat loop in the Constructor that rebuilds the whole data structure if } \sum_{i \in [\lceil \sqrt{\frac{n}{2}} \rceil]} n_i^2 \geq \text{too big.}\right) \leq 1.
\]

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).
14.5 Exercises

Exercise 14.3 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

$$h(A) = \left( \sum_{i=1}^{s} A[i] \cdot x^{s-i+1} \right) \mod p.$$ 

Algorithm 14.8 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 $x \in \{0, \ldots, p-1\}$ uniformly at random
4: Compute $\alpha \leftarrow h(A)$
5: Compute $\beta \leftarrow h(B[1..s])$
6: for $i = 1, \ldots, n-s+1$ do
7:   ▷ Assert: $\beta = h(B[i..i+s-1])$
8:   if $\alpha = \beta$ then return True
9:   if $i < n-s+1$ then
10:      $\beta \leftarrow h(B[i+1..i+s])$  ▷ 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 $\alpha$ and $\beta$ 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 $\beta$ to compute the new value of $\beta$.)

Hint: See Fact A.2.10 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 1.4.)

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 15

Graphs

15.1 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 $\alpha$-approximation if it always outputs a set $U$ for which $|\delta(U)| \geq \alpha \cdot \OPT$. (Naturally we want $\alpha$ 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 15.1 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 $U$.

Theorem 15.1.1. Let $U$ be the output of the algorithm. Then $E[|\delta(U)|] \geq \OPT/2$.

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,

$$E[|\delta(U)|] = E\left[ \sum_{uv \in E} X_{uv} \right] = \sum_{uv \in E} E[X_{uv}] = \sum_{uv \in E} \Pr[X_{uv} = 1].$$
Algorithm 15.1 A randomized $1/2$-approximation for Max Cut.

1: function MaxCut(graph $G = (V,E)$)
2: Let $U$ be a uniformly random subset of $V$ \[ \triangleright \text{See Exercise 2.3.} \]
3: return $U$
4: end function

Now we note that

\[
\Pr[X_{uv} = 1] = \Pr[(u \in U \land v \notin U) \lor (u \notin U \land v \in U)] \\
= \Pr[u \in U \land v \notin U] + \Pr[u \notin U \land v \in U] \quad \text{(union of disjoint events, Fact A.3.7)} \\
= \Pr[u \in U] \cdot \Pr[v \notin U] + \Pr[u \notin U] \cdot \Pr[v \in U] \quad \text{(independence)} \\
= 1/2.
\]

Thus $E[|\delta(U)|] = |E|/2 \geq \text{OPT}/2$, since clearly OPT cannot exceed $|E|$.

15.1.1 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 $|\delta(U)| \geq \text{OPT}/2$? Perhaps we have some undesirable scenario, like

\[
\Pr[|\delta(U)| \leq \text{OPT}/100] = 0.99 \\
\Pr[|\delta(U)| \geq \text{OPT}/2] = 0.01.
\]

We can use the Reverse Markov inequality (Exercise 7.5) to show that $|\delta(U)|$ is likely to be quite large. Let $Z = |\delta(U)|$, $d = \text{OPT}$ and $c = (\frac{1}{2} - \epsilon)\text{OPT}$. Note that $Z$ is never larger than $d$. Then

\[
\Pr[|\delta(U)| \leq (1/2 - \epsilon)\text{OPT}] = \Pr[Z \leq c] \quad \text{(definition of } Z \text{ and } c) \\
\leq \frac{d - E[Z]}{d - c} \quad \text{(Reverse Markov inequality)} \\
= \frac{\text{OPT} - \text{OPT}/2}{\text{OPT} - (\frac{1}{2} - \epsilon)\text{OPT}} \quad \text{(recall } E[|\delta(U)|] = \text{OPT}/2) \\
= \frac{1/2}{1/2 + \epsilon} = (1 + 2\epsilon)^{-1} \quad \text{(simplifying)}.
\]

To judge whether this guarantee is any good, consider setting $\epsilon = 0.01$. Then the probability of a small cut is

\[
\Pr[|\delta(U)| \leq 0.49 \cdot \text{OPT}] \leq \frac{1}{1.02} < 0.99.
\]

Equivalently,

\[
\Pr[|\delta(U)| > 0.49 \cdot \text{OPT}] \geq 0.01.
\]

This bound is very weak, but at least the probability of a large cut is some positive constant.

Keener Kwestion 15.1.2. How might we find a cut $U$ satisfying $|\delta(U)| > 0.49 \cdot \text{OPT}$ with 99% probability instead of with 1% probability?
15.2 Minimum cut

In Section 15.1 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 15.4.

15.2.1 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 15.2.1. How can you solve the Global Min Cut problem using the Min $s$-$t$ Cut problem?

Answer.

See, e.g., (Kleinberg and Tardos, 2006, Theorem 13.1). The solution to the Min $s$-$t$ Cut problem equals the minimum over all pairs $s,t \in V$ of the minimum flow from $s$ to $t$.

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.

15.2.2 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 15.2 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 15.2.2.** 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 15.2.3.** 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 15.2.2, 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.
15.2.3 The Contraction Algorithm

The contraction algorithm, shown in Algorithm 15.3, is one of the most famous randomized algorithms. It outputs a cut, which may or may not be a minimum cut.

Algorithm 15.3 The Contraction Algorithm. Assume that the graph $G$ is connected.

1: function CONTRACTIONALG(graph $G = (V, E)$)
2:     while the graph has more than two supervertices remaining do
3:         Pick an edge $e$ uniformly at random, and contract it
4:     end while
5:     Let $u$ be one of the two remaining vertices. (It does not matter which).
6:     Let $U$ be the vertices of the original graph contained in $u$.
7:     Output the cut $\delta(U)$.
8: end function

Remarkably, although the algorithm seems to do nothing at all, it has decent probability of outputting a minimum cut.

Theorem 15.2.4. 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.

\[
\begin{array}{c}
\{a, d\} \\
\{u, v, b, c\} \\
\delta(\{a, d\})
\end{array}
\]

15.2.4 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 15.2.5. Should we use Probability Amplification for one-sided error, or for two-sided error?

Corollary 15.2.6. 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

\[
\left(1 - \frac{1}{n^2}\right)^{n^2 \ln(1/\delta)} \leq \left(e^{-1/n^2}\right)^{n^2 \ln(1/\delta)} = \delta,
\]
15.2.5 The proof

Before proving the theorem we need two more preliminary claims.

**Claim 15.2.7.** 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.)

**Claim 15.2.8.** 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.

*Proof of Theorem 15.2.4.* 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 15.2.8, 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 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 15.2.3, and it is at most $k$ because we suppose that the cut $C$ still survives.) So Claim 15.2.7 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

$$\Pr\left[\text{edge contracted in } i\text{th iteration not in } C \mid \text{haven’t contracted an edge in } C \text{ so far}\right] \geq 1 - \frac{2}{n - i + 1} = \frac{n - i - 1}{n - i + 1}.$$
Now we apply Fact A.3.5 to analyze all iterations.

\[
\Pr[\text{never contract an edge in } C] = \prod_{i=1}^{n-2} \Pr[\text{edge contracted in } i^{th} \text{ iteration not in } C \mid \text{haven't contracted an edge in } C \text{ so far}]
\]

\[
\geq \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1}
\]

\[
= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}
\]

\[
= \frac{2}{n(n-1)},
\]

by massive amounts of cancellation. So, by Claim 15.2.8, the probability that the algorithm outputs the cut \(C\) is at least \(2/n(n-1)\).

15.2.6 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 15.2.9.** In any connected, undirected graph the number of minimum cuts is at most \((n^2)/2\).

**Proof.** Let \(C_1, \ldots, C_\ell\) be the minimum cuts of the graph. Then we have

\[
1 \geq \Pr[\text{algorithm outputs some min cut}]
= \Pr[\text{algorithm outputs } C_1 \lor \cdots \lor \text{algorithm outputs } C_\ell]
= \sum_{i=1}^{\ell} \Pr[\text{algorithm outputs } C_i] \quad \text{(union of disjoint events, Fact A.3.7)}
\geq \ell \cdot \frac{2}{n(n-1)} \quad \text{(by Theorem 15.2.4)}.
\]

Rearranging, we conclude that \(\ell \leq n(n-1)/2 = \binom{n}{2}\).

**Keener Kwestion 15.2.10.** Amazingly, Corollary 15.2.9 is exactly tight! For every \(n\), there is a connected graph with \(n\) vertices and exactly \((n^2)/2\) minimum cuts. Can you think of this graph?

**Answer.** Consider the cycle with \(n\) vertices. A min cut has two edges, and there are \(\binom{n}{2}\) ways to choose two edges from the cycle.

The next corollary proves a similar result for approximate minimum cuts.

**Definition 15.2.11.** For any \(\alpha \geq 1\), a cut is called an \(\alpha\)-small-min-cut if its number of edges is at most \(\alpha\) times the size of a minimum cut.

**Corollary 15.2.12.** In any undirected graph, and for any integer \(\alpha \geq 1\), the number of \(\alpha\)-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 $\alpha$-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 $\alpha$-small-min-cut survives contraction down to $r$ vertices is

$$\Pr[\text{survives}] \geq \prod_{i=1}^{n-r} \left(1 - \frac{\alpha c}{(n-i+1)c/2}\right)$$

$$= \prod_{i=1}^{n-r} \frac{n-i+1-2\alpha}{n-i+1}$$

$$= \frac{(n-2\alpha)(n-2\alpha-1)\cdots(r-2\alpha+1)}{n(n-1)\cdots(r+1)}$$

$$= (n-2\alpha)! \cdot \frac{r!}{n!}$$

So, the probability that a particular $\alpha$-small-min-cut is output by the algorithm is at least

$$\Pr[\text{survives}] \cdot \Pr[\text{output | survives}] = (n-2\alpha)! \cdot \frac{r!}{n!} \cdot 2^{-r} > n^{-2\alpha},$$

where we have used the inequalities $2^r \leq r!$ and $n!/(n-2\alpha)! < n^{2\alpha}$. \qed
15.3 Exercises

Exercise 15.1 Finding all min cuts. Corollary 15.2.6 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 $\delta$, with probability at least $1 - \delta$, it can output a list containing every minimum cut in the graph.

15.4 Graph Skeletons

15.4.1 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

Silly goal 1: $|\delta_H(U)| \approx |\delta_G(U)| \quad \forall U \subseteq V.$

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?

Silly goal 2: $E[|\delta_H(U)|] = p \cdot |\delta_G(U)| \quad \forall U \subseteq V.$

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

$$U \text{ is good } = \left\{ 1 - \epsilon \leq \frac{|\delta_H(U)|}{p \cdot |\delta_G(U)|} \leq 1 + \epsilon \right\}$$

$$H \text{ is good } = \{ U \text{ is good } \forall U \subseteq V \}.$$  

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

Actual goal: $\Pr[H \text{ is good}] \geq 0.5$.

15.4.2 The Compression Algorithm

We will discuss the following algorithm for compressing a graph.

**Algorithm 15.4 The Compression Algorithm.** We assume $0 < \epsilon < 1$.

1: function COMPRESS(graph $G = (V, E)$, float $\epsilon$)
2: Let $c$ be the size of the minimum cut in $G$.
3: Let $p = 15 \ln(2n) / (\epsilon^2 c)$.
4: Create graph $H$ by sampling every edge in $G$ with probability $p$.
5: Output $H$.
6: 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 15.4.1.** The compression algorithm produces a graph $H$ satisfying

$$E[\# \text{ edges in } H] = O\left(\frac{\log n}{\epsilon^2 c}\right)$$

$$\Pr[H \text{ is good}] \geq 1 - 2/n.$$  

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 $\mu = \mathbb{E} [ |\delta_H(U)| ] = p \cdot |\delta_G(U)|$. We have

$$\Pr [ U \text{ is bad} ] = \Pr [ |\delta_H(U)| \leq (1 - \epsilon)\mu ] + \Pr [ |\delta_H(U)| \geq (1 + \epsilon)\mu ]$$

$$\leq 2 \exp (-\epsilon^2 \mu / 3) \quad \text{(by the Chernoff bound)}$$

$$= 2 \exp \left( - \epsilon^2 \cdot \frac{p \cdot |\delta_G(U)|}{3} \right) \quad \text{(definition of } \mu)$$

$$= 2 \exp \left( - \frac{5 \ln(2n)|\delta_G(U)|}{c} \right) \quad \text{(definition of } p) \quad \text{(15.4.1)}$$

$$\leq 2 \exp \left( - \frac{5 \ln(2n)}{c} \right) < 1/n^5.$$}

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 15.4.2.** 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 $n$-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 (15.4.1), we have

$$\Pr [ U \text{ is bad} ] \leq 2 \exp \left( - \frac{5 \ln(2n)\delta_G(U)}{c} \right)$$

$$\leq 2 \exp \left( - \frac{5 \ln(2n)2^i c}{c} \right) \quad \text{(it is a nearly-2-$i$-min-cut)}$$

$$< n^{-5 \cdot 2^i}.$$}

Now we will union bound over all the nearly-2-$i$-min-cuts. These are all $\alpha$-small-min-cuts with $\alpha = 2^{i+1}$, so by Corollary 15.2.12 we have

$$(\text{number of nearly-2-$i$-min-cuts}) \leq n^{2\alpha} = n^{2^i} = n^4 2^i.$$}

Thus, by a union bound,

$$\Pr [ \text{any nearly-2-$i$-min-cut is bad} ] \leq (\text{number of nearly-2-$i$-min-cuts}) \cdot (\text{failure probability})$$

$$\leq n^{4 \cdot 2^i} \cdot n^{-5 \cdot 2^i}$$

$$= n^{-2^i}.$$}

Now doing another union bound over $i$,

$$\Pr [ \text{any cut is bad} ] \leq \sum_{i \geq 0} \Pr [ \text{any nearly-2-$i$-min-cut is bad} ]$$

$$\leq \frac{1}{n} + \frac{1}{n^2} + \frac{1}{n^4} + \frac{1}{n^8} + \cdots$$

$$\leq \frac{1}{n} \sum_{i \geq 0} n^{-i}$$

$$\leq \frac{2}{n},$$}

by our standard bound on geometric sums. Thus, $\Pr [ H \text{ is good} ] \geq 1 - 2/n.$
15.4.3 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 15.2.

  *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 $O\left(\frac{n \log n}{\epsilon^2 c}\right)$ edges, regardless of the value of $c$!
Chapter 16

Distributed Computing

16.1 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 16.1, 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 16.1.1.** 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 16.1 The Randomized Exponential Backoff algorithm.

1: function \textsc{Backoff} \\
2:   \begin{align*}
3:     & \text{Let } W \leftarrow 1 \quad \triangleright \text{ The current window size} \\
4:     \text{while haven’t yet accessed resource do} \\
5:     \text{Generate } X \in [W] \text{ uniformly at random} \\
6:     \text{Try to access the resource in time slot } X \text{ of current window} \\
7:     \text{if conflict when accessing resource then} \\
8:     \text{Wait until the end of the current window} \\
9:     \text{end if} \\
10: \text{end while} \\
11: \end{align*}

slots? This is precisely the birthday paradox! We saw in (5.1.1) 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.

\textbf{Theorem 16.1.2.} The completion time is $O(n \log(n) \log \log(n))$, 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.
- $R_{\text{before}}$ is the number of remaining clients at the start of the iteration.
- $R_{\text{after}}$ is the number of remaining clients at the end of the iteration.

The goal is to analyze $R_{\text{after}}$, 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 $R_{\text{before}}$ balls and $W$ bins. Using (5.1.1) again, we obtain\(^2\)

\[ E[R_{\text{after}}] < R_{\text{before}}^2/W. \] \hspace{1cm} (16.1.1)

Let’s ignore the first few iterations because probably all clients will conflict. Instead, we focus on the $k$ iterations numbered \[ \lg(4nk) + 1, \ \lg(4nk) + 2, \ldots, \ \lg(4nk) + k. \]

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

\[ \mathcal{E}_i = \text{“} R_{\text{after}} \geq 2kR_{\text{before}}^2/W \text{”}. \]

\(^2\)The fastidious reader will note that this expectation should be conditioning on the outcomes of all previous iterations.

©Nicholas Harvey 2021. All rights reserved.
We can analyze the probability of this event as follows.

\[
\Pr [E_i] = \Pr \left[ R_{\text{after}} \geq 2kR_{\text{before}}^2/W \right] \\
\leq \frac{E[R_{\text{after}}]}{2kR_{\text{before}}^2/W} \quad \text{(by Markov’s inequality, Fact A.3.20)} \\
\leq \frac{R_{\text{before}}^2/W}{2kR_{\text{before}}^2/W} \quad \text{(by (16.1.1))} \\
= \frac{1}{2k}
\]

We can analyze the probability of any failure during these iterations with a union bound.

\[
\Pr [E_1 \vee \cdots \vee E_k] \leq \sum_{i=1}^{k} \Pr [E_i] \leq k \cdot \frac{1}{2k} = \frac{1}{2}.
\]

Thus, with probability at least 1/2, all trials are successful.

So assume that all trials are successful. Note that the \(i^{th}\) trial has window size

\[
\text{(window size during } i^{th} \text{ trial)} = 2^{i\log(4nk)}/2 \geq 4nk,
\]

since the window size doubles with each iteration. The crux of the analysis is the following lemma.

**Lemma 16.1.3.** Assume that all trials are successful. Then the number of remaining clients after the \(i^{th}\) trial is at most \(2n/2^i\).

*Proof.* The proof is by induction. The base case is when \(i = 0\). This is trivial because \(2n/2^0 = n\), and there are initially \(n\) clients.

So assume \(i \geq 1\) and let \(R_{\text{before}}\) be the number of remaining clients before the \(i^{th}\) trial. By induction, \(R_{\text{before}} \leq 2n/2^{i-1}\). We have observed in (16.1.2) that \(W \geq 4nk\). So, after the \(i^{th}\) trial, the number of remaining clients is at most

\[
\frac{2kR_{\text{before}}^2}{W} \leq \frac{2k \cdot (2n/2^{i-1})^2}{4nk} = \frac{8n}{4 \cdot 2^i} = \frac{2n}{2^i}. \quad \square
\]

Since the number of remaining clients decreases *doubly exponentially* in \(i\), we can ensure that no clients remain by defining

\[k = \log \log (4n).\]

If all \(k\) trials succeed, Lemma 16.1.3 tells us that the number of remaining clients is at most

\[
\frac{2n}{2^{2k}} = \frac{2n}{2^{2\log(4n)}} = \frac{2n}{2^{\log(4n)}} = \frac{1}{2}.
\]

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 \(\log(4nk) + k\). This is just the sum of all window sizes, which is

\[
1 + 2 + 4 + \cdots + 2^{\log(4nk)+k} < 2 \cdot 2^{\log(4nk)+k} \quad \text{(geometric sum, (A.2.1))} \\
= 8nk2^k \\
= 8n \log \log (4n) \log (4n) \quad \text{(by definition of } k\text{)} \\
= O(n \log(n) \log \log(n)).
\]
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.

16.1.1 Exercises

Exercise 16.1. 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 16.2. Improve the analysis of the Backoff algorithm to show that it has completion time \( O(\log n) \) with probability at least 1/2.

16.2 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 \( h : U \rightarrow B \). 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

\[
(a - b) \mod 1.
\]

**Example 8.** The leftward-distance from 0.6 to 0.4 is

\[
(0.6 - 0.4) \mod 1 = 0.2.
\]

However, the distance is not symmetric: the leftward distance from 0.4 to 0.6 is

\[
(0.4 - 0.6) \mod 1 = (-0.2) \mod 1 = 0.8.
\]

**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 \( h : U \rightarrow B \) to map data directly to \( B \), we will instead use a hash function \( h : U \rightarrow C \) 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 \((h(z) - y) \mod 1\) 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 \( h : U \rightarrow C \) 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 \( b' \).

16.2.1 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 16.2.1.** \( E[X_i] = 1/n \) for all \( i \in [n] \).

**Proof.** Our first observation is that

\[
E[X_i] = E[X_j] \quad \forall i, j.
\]

(16.2.1)

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 \( i \),

\[
1 = E\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} E[X_j] = n E[X_i],
\]

now using (16.2.1). Dividing by \( n \) completes the proof. \( \square \)

The preceding claim shows that the system is fair to the nodes in expectation. However, reiterating our moral from Section 7.1:

\(^3\)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 16.2.2.** 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 16.2.3.** 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:

$$A_i = [i/n, (i + 2\ell)/n \mod 1].$$

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

$$\Pr[\mathcal{E}_i] = \prod_{j=1}^n \Pr[j^{th} \text{ node not in } A_i] \quad \text{(by independence)}$$

$$= (1 - 2\ell/n)^n \quad \text{(length of } A_i \text{ is } 2\ell/n, \text{ and (A.3.2))}$$

$$\leq \exp(-2\ell/n)^n \quad \text{(by Fact A.2.5)}$$

$$= \exp(-2\ell/n) < 1/n^2.$$

We want to show that it is unlikely that any arc $A_i$ contains no points. This is accomplished by a union bound.

$$\Pr[\text{any arc } A_i \text{ has no nodes}] = \Pr[\mathcal{E}_1 \lor \cdots \lor \mathcal{E}_n] \leq \sum_{i=1}^n \Pr[\mathcal{E}_i] = n \cdot (1/n^2) = 1/n.$$

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.

**Keener Kwestion 16.2.4.** 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 $E[X_i]$?

**Broader context.** Consistent hashing is used in many real-world products, including Akamai’s content distribution network, Amazon Dynamo, Apache Cassandra, Discord, etc.
16.3 Exercises

**Exercise 16.3.** 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 16.4.** Let $X_i$ be the fraction of the circle that node $i$ is responsible for. Let $k = 100n^2$. Prove that $\Pr[\text{any node has } X_i \leq 1/2k] \leq 1/100$.

16.4 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.

### 16.5 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 $L_\epsilon$ containing all the nodes. (Here $\epsilon$ 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 $L_\epsilon$ 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 $L_{10}$ 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 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

\[ L_\epsilon, L_0, L_{01}, L_{010}, L_{0100}, L_{01001}, L_{010010}, L_{0100101}, L_{01001011}, \ldots \]

As desired, all nodes join \( L_\epsilon \), half the nodes join \( L_0 \), the other half join \( L_1 \), etc.

16.5.1 Searching by key

Question 16.5.1. 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 16.5.2. For any node \( s \), the set of lists to which it belongs forms a Skip List containing all the nodes. In principle the bitstring could have infinite length as the bits can be generated on demand.

Algorithm 16.2 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

16.5.2 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 \( L_\epsilon \) is sorted by key and not sorted by the random bitstrings. Nevertheless, it is possible, as we shall see.

Algorithm 16.3 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 \( L_\epsilon \)
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
Claim 16.5.3. 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. □

16.5.3 Join (or insertion) of a new node

Inserting a new node is quite easy, now that we have the strange SearchByBitstring() operation.

Algorithm 16.4 Inserting a new node with key $v$.

1: function Insert(node key $v$)
2: Create a new node with key $v$ and a new random bitstring $b$
3: Perform SearchByBitstring($b$), arriving at a node $s$. So $s.bitstring$ has the longest
4: common prefix with $b$, among all existing nodes.
5: Restrict attention to the Skip List consisting of the lists to which $s$ belongs. (Imagine
6: temporarily deleting all other lists.)
7: Perform INSERT($v$) in this Skip List.
8: end function
Chapter 17

Learning

17.1 Distinguishing distributions

Suppose there are two machines that generate random numbers in [100]. Let us call them Machine A and Machine B. We are given a single random number that was either generated by A or B, but we do not know which. Based on the value that we receive, our task is to guess which of the machines generated it.

Intuition tells us that, if A and B generate numbers with exactly the same distribution, then this task is impossible. However, if the distributions are somewhat different, then perhaps it might be possible.

For concreteness, define

\[ a_i = \Pr[A \text{ generates the value } i] \]
\[ b_i = \Pr[B \text{ generates the value } i] \]

Claim 17.1.1. There are two distributions with \(|a_i - b_i| \leq 0.01\) for every \(i\), but we can guess correctly with probability 0.5.

Proof. Suppose that Machine A returns a uniform random number in [100], whereas Machine B returns a uniform random number in [50]. Then \(a_i = 0.01\) for all \(i\), \(b_i = 0.02\) for \(i \leq 50\) and \(b_i = 0\) for \(i > 50\). Note that \(|a_i - b_i| = 0.01\) for all \(i\).

If we receive a value \(X > 50\) we will guess that A generated it; otherwise we will guess that B generated it. The only possibility of a mistake is when A generates a number in [50], which happens with probability at most 0.5. \(\square\)
Chapter 18

Secrecy

18.1 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 13, 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.

Larry’s message: $M_1, \ldots, M_s$

where each $M_i \in [q]$

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.

Random secret key: $R_1, \ldots, R_s$

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 $q$.

Encrypted message: $E_1, \ldots, E_s$

where $E_i = (M_i + R_i) \mod q$

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.

Decrypted message: $D_1, \ldots, D_s$

where $D_i = (E_i - R_i) \mod q$.

Claim 18.1.1. The decrypted message $D$ equals Larry’s message $M$. 

181
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.2.10), we have

\[ D_i = (E_i - R_i) \mod q = \left( (M_i + R_i) \mod q - R_i \right) \mod q = (M_i + R_i - R_i) \mod q = M_i \]

for every \( i \). So the messages are equal in every component. \( \square \)

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 18.1.2.** If \( R \) is unknown, then \( E \) reveals nothing about \( M \).

Proof. We will 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.3.22 with \( X = R_i \) and \( y = M_i \), we obtain that

\[ E_i = (R_i + M_i) \mod q = (X + y) \mod q \]

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 \( e_1, \ldots, e_s \),

\[ \Pr[E_1E_2\ldots E_s = e_1e_2\ldots e_s] = \prod_{i=1}^{s} \Pr[E_i = e_i] = 1/q^s, \]

which means that \( E \) is uniformly random. \( \square \)

Implementations of this scheme commonly use the case \( q = 2 \), in which the messages and key are bit strings.

**Question 18.1.3.** How are addition and subtraction implemented in the case \( q = 2 \)?

**Answer.**

Both addition and subtraction are simply the Boolean \( \lor \) operation. See page 140.

**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.
18.2 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 18.2.1.** 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.

\[
\Pr[\text{says "vaccinated"}] = \Pr[\text{is vaccinated and tells truth}] + \Pr[\text{is unvaccinated and lies}]
\]

\[
= f \cdot (1 - b) + (1 - f) \cdot b
\]

\[
= f \cdot (1 - 2b) + b
\]

(18.2.1)

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

\[
E[X] = f \cdot (1 - 2b) + b,
\]

(18.2.2)

which is exactly the same as in (18.2.1).

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 (18.2.2), it is natural to use the estimate

\[
\hat{f} = \frac{X - b}{1 - 2b}
\]

(18.2.3)

This quantity $\hat{f}$ is called an unbiased estimator because its expected value equals the true value. That is,

\[
E[\hat{f}] = E\left[\frac{X - b}{1 - 2b}\right] = \frac{E[X] - b}{1 - 2b} = f,
\]

(18.2.4)

by plugging in (18.2.2).
**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,

\[
E[X] = 0.98f + 0.01
\]

\[
\hat{f} = \frac{X - 0.01}{0.98}
\]

Unbiased estimate: \( E[\hat{f}] = f \).

The last equality comes directly from (18.2.4).

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 (18.2.4) 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 (18.2.4)).

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 $f$!

**Keener Kwestion 18.2.2.** What happens if we set $b = 0.99$? Or $b = 0.5$?

**Keener Kwestion 18.2.3.** 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?

**18.2.1 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 18.2.4.** Does $E[\hat{f}] = f$ imply that $\hat{f}$ is likely to be close to $f$?

**Answer.**

\[
\text{No: suppose that there is only one student. If the student announces being vaccinated, then } \hat{f} = 1, \text{ and if vaccinated, then } \hat{f} = 0, \text{ which is a rather narrow estimate!}
\]

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 9.4 after discussing the necessary tools.
18.2.2 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 in today’s example, 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.

18.2.3 Exercises

Exercise 18.1. 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?

18.3 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 18.3.1. 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.

Students: $A \quad B \quad C$
Grades: $g_A \quad g_B \quad g_C \in \{0, \ldots, 100\}$.

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.

Student $A$ generates blinds: $AB$ and $AC$
Student $B$ generates blinds: $BA$ and $BC$
Student $C$ generates blinds: $CA$ and $CB$
Exchanging blinds. Taking care to remember the blinds that they generated, each student (privately) sends a blind to each other student.

Public announcement. Now each student announces a public value which is their own grade, minus their blinds, plus what they received, all modulo 400.

Computing the sum. Every student now adds up all the public values, modulo 400. The clever observation is that all the blinds cancel!

Is secrecy guaranteed? The main question is: does this protocol guarantee secrecy?

Question 18.3.2. In the protocol above, what does student C learn about $g_A$ and $g_B$?

Answer.

Is secrecy guaranteed? The main question is: does this protocol guarantee secrecy?

Question 18.3.2. In the protocol above, what does student C learn about $g_A$ and $g_B$?

Answer.
Theorem 18.3.3. 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.3.23. 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

\[
(X + Y) \mod q
\]

is also uniformly random in \([q]\).

Proof of Theorem 18.3.3. 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 \( p_A \).

\[
p_A = \left( \frac{BA + g_A - AB - AC + CA}{X} \right) \mod 400
\]

This is uniformly random in \([400]\) by Corollary A.3.23, because \( BA \) is just a uniformly random value in \([400]\) (and independent of all other values inside \( Y \)).

Similarly,

\[
p_B = \left( \frac{AB + g_B - BA - BC + CB}{X} \right) \mod 400
\]

is uniformly random in \([400]\) by Corollary A.3.23 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

\[
(p_A + p_B) \mod 400 = \left( g_A + g_B - AC + CA - BC + CB \right) \mod 400. \tag{18.3.1}
\]

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 (18.3.1). 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.

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 about people without revealing their private information.

©Nicholas Harvey 2021. All rights reserved.
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 a rule I will try to avoid using excessive notation. For the sake of clarity and conciseness, here is some notation that we will use.

\[
[p] = \{0, 1, \ldots, p - 1\} \\
[p] = \{1, 2, \ldots, p\} \\
[a, b] = \{ \text{real numbers } x : a \leq x \leq b \} \\
\ln x = \log_e(x) \\
\lg x = \log_2(x) \\
\lor = \text{Boolean Or} \\
\land = \text{Boolean And}
\]

\(\log(x)\) denotes the logarithm to a base that is unspecified or irrelevant.

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
\[
\sum_{i=0}^{n} c^i = \frac{1 - c^{n+1}}{1 - c} = \frac{c^{n+1} - 1}{c - 1}.
\]
A useful special case is
\[
\sum_{i=0}^{n} 2^i = 2^{n+1} - 1. \tag{A.2.1}
\]

References:  (Lehman et al., 2018, Theorem 14.1.1), (Cormen et al., 2001, equations (A.5) and (A.6)).

**Standard logarithm rules.**

\[
\log(ab) = \log(a) + \log(b)
\]
\[
\log(x^k) = k \log(x)
\]
\[
\log_b(x) = \frac{\ln(x)}{\ln(b)} \tag{A.2.2}
\]

**Manipulating exponents.**

\[(a^b)^c = a^{bc} = (a^c)^b.\]

**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:

\[a^{\log_{1/a}(x)} = \frac{1}{x}.\]

**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 $\alpha$ **rounded up to a power of two** means the value $2^\ell$ satisfying
\[
\frac{1}{2} \cdot 2^\ell < \alpha \leq 2^\ell, \tag{A.2.3}
\]
and $\ell$ an integer. This condition is satisfied by taking $\ell = \lceil \lg \alpha \rceil$, so we can also define $\alpha$ rounded up to a power of two to be $2^{\lfloor \lg \alpha \rfloor}$.

**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
\[|a_1 + \ldots + a_n| \leq |a_1| + \ldots + |a_n|.
\]

**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 $n \geq 2$,
\[
\frac{1}{5} \leq \frac{1}{e} - \frac{1}{4n} \leq \left(1 - \frac{1}{n}\right)^n \leq \frac{1}{e} < 0.37.
\]

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
\[
S = \sum_{i=0}^{a} f(i) \quad \text{and} \quad I = \int_{0}^{a} f(x) \, dx.
\]

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 $(i, j)$ satisfying $1 \leq i < j \leq n$ is
\[
\binom{n}{2} = \frac{n(n-1)}{2}.
\]

References: (Lehman et al., 2018, Section 15.5).

Fact A.2.9 (Bit Representation). Every number in $x \in [2^k]$ has a unique binary representation $x_0, \ldots, x_{k-1} \in \{0, 1\}$, satisfying
\[
x = \sum_{i=0}^{k-1} x_i 2^i.
\]
Consequently, for every $n \geq 1$, the integers in the set $\left[ n \right]$ can be represented using $\lceil \lg n \rceil$ bits. More generally, for every set of size $n \geq 1$, its elements can be identified using $\lceil \lg n \rceil$ bits because they can be labeled by numbers in $\left[ n \right]$.

A.2.2 Number theory

Fact A.2.10 (Take mod everywhere). For any integers $a, b$ and positive integer $n$,
\[
(a + b) \mod n = ((a \mod n) + (b \mod n)) \mod n
\]
\[
(a - b) \mod n = ((a \mod n) - (b \mod n)) \mod n
\]
\[
(a \cdot b) \mod n = ((a \mod n) \cdot (b \mod n)) \mod n
\]

Fact A.2.11 (Existence of primes). For any integer \( n \geq 1 \), there exists a prime \( p \in \{n, \ldots, 2n\} \).

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.2.12 (Number of primes). For any integer \( n \geq 2 \), the number of primes in \( \{4n + 1, \ldots, 20n\} \) 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.2.13 (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 \( c_{i} \mod p \neq 0 \). Then there are at most \( d \) solutions to

\[
g(x) \mod p = 0 \quad \text{and} \quad x \in \mathbb{Z}/p\mathbb{Z}.
\]

Such solutions are usually called roots.

Fact A.2.14 (Inverses mod \( p \)). For any prime \( p \) and any \( x \in \{1, \ldots, p-1\} \), there is a unique solution \( y \) to

\[
xy \mod p = 1 \quad \text{and} \quad y \in \{1, \ldots, p-1\}.
\]

This value \( y \) is usually denoted \( x^{-1} \).


Fact A.2.15 (Systems of equations mod \( p \)). Let \( p \) be a prime number. Let \( a, b \in \mathbb{Z}/p\mathbb{Z} \) satisfy \( a \neq b \). Let \( c, d \in \mathbb{Z}/p\mathbb{Z} \). Let \( X \) and \( Y \) be variables. Then there is exactly one solution to

\[
\begin{align*}
(aX + Y) \mod p &= c \\
(bX + Y) \mod p &= d
\end{align*}
\]

\( X, Y \in \mathbb{Z}/p\mathbb{Z} \).

Intuition. Roughly, Fact A.2.15 is just looking at the linear system \( \begin{pmatrix} a & b \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} c \\ d \end{pmatrix} \), but doing all arithmetic modulo \( p \). The matrix \( \begin{pmatrix} a & b \end{pmatrix} \) 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 $Y$ cancel and we get

$$(a-b)X \mod p = (c-d) \mod p$$

By Fact A.2.14, this is solved if and only if $X = (c-d)(a-b)^{-1} \mod p$. Plugging this $X$ back in to the first equation, we get that $Y$ must have the value $Y = (c-aX) \mod p = (c-a(c-d)(a-b)^{-1}) \mod p$. □

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 $A$ happens, event $B$ must also happen. (This is often written $A \Rightarrow B$ or $A \subseteq B$.) Then $\Pr [A] \leq \Pr [B]$.

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

$$\Pr [\text{$\mathcal{E}$ does not happen}] = 1 - \Pr [\text{$\mathcal{E}$ happens}].$$

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

$$\Pr \left[ \bigwedge_{i \in I} \mathcal{E}_i \right] = \prod_{i \in I} \Pr [\mathcal{E}_i] \quad \forall I \subseteq [n].$$

They are called pairwise independent if the following weaker condition holds.

$$\Pr [\mathcal{E}_i \land \mathcal{E}_j] = \Pr [\mathcal{E}_i] \Pr [\mathcal{E}_j] \quad \forall \text{distinct } i, j \in [n].$$

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

$$\Pr [B \mid A] = \frac{\Pr [A \land B]}{\Pr [A]}$$

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

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

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

**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 $B_1, \ldots, B_n$ be events such that exactly one of these events must occur. Let $A$ be any event. Then

$$
\Pr[A] = \sum_{i=1}^{n} \Pr[A \land B_i].
$$

Furthermore, if $\Pr[B_i] > 0$ for each $i$, then

$$
\Pr[A] = \sum_{i=1}^{n} \Pr[A \mid B_i] \Pr[B_i].
$$

**References:** (Lehman et al., 2018, Section 18.5), (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, $\Pr[\mathcal{E}_i \land \mathcal{E}_j] = 0$ 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

$$
\Pr[\mathcal{E}_1 \lor \cdots \lor \mathcal{E}_n] = \sum_{i=1}^{n} \Pr[\mathcal{E}_i].
$$

**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

$$
\Pr[\text{any of the events occurs}] = \Pr[\mathcal{E}_1 \lor \cdots \lor \mathcal{E}_n] \leq \sum_{i=1}^{n} \Pr[\mathcal{E}_i].
$$

An easier form that is often useful is:

$$
\Pr[\mathcal{E}_1 \lor \cdots \lor \mathcal{E}_n] \leq n \cdot \max_{1 \leq i \leq n} \Pr[\mathcal{E}_i].
$$

The union bound may also be equivalently stated as

$$
\Pr[\bar{\mathcal{E}}_1 \land \cdots \land \bar{\mathcal{E}}_n] \geq 1 - \sum_{i=1}^{n} \Pr[\mathcal{E}_i]. \tag{A.3.1}
$$

**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, \ldots\}, 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. One way to define the expected value is

\[
E[X] = \sum_r r \cdot Pr[X = r].
\]

The sum 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

\[
E[X] = \sum_{n \geq 1} Pr[X \geq n].
\]


The preceding fact generalizes to arbitrary random variables; see 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

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

**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).

**Definition A.3.12** (Indicator random variable). Let \(E\) be an event. The indicator of the event \(E\) is the random variable \(X\) taking value 1 if \(E\) occurs, and 0 otherwise. By the definition of expectation, we have

\[
E[X] = Pr[E].
\]

\(^1\) 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.
Fact A.3.13 (Expected sum of indicators). Suppose that $X$ is a random variable that can be decomposed as $X = \sum_{i=1}^{n} X_i$, where $X_i$ is the indicator of an event $E_i$. Then

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \Pr[E_i].$$

Fact A.3.14 (Law of total expectation). Let $B_1, \ldots, B_n$ be events such that exactly one of these events must occur, and each $\Pr[B_i] > 0$. Let $X$ be any random variable. Then

$$E[X] = \sum_{i=1}^{n} E[X | B_i] \Pr[B_i].$$

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 $\Pr[X = 1] = p$ (so $\Pr[X = 0] = 1 - p$). 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 some finite set. A random variable $X$ is uniformly distributed on $S$ if

$$\Pr[X = s] = \frac{1}{|S|} \quad \forall s \in S.$$

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

$$\Pr[X \in [a, b]] = \frac{b-a}{u}, \quad (A.3.2)$$

References: (Anderson et al., 2017, Facts 10.4 and 10.10), (Mitzenmacher and Upfal, 2005, Lemma 2.6).
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.3.15.** 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)).

**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.

$$E[X] = np \quad (A.3.3)$$
$$\Pr [X = i] = \binom{n}{i} p^i (1 - p)^{n-i} \quad \forall i \in \{0, 1, \ldots, n\} \quad (A.3.4)$$

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

A detailed discussion of tail bounds for binomial random variables is in Chapter 8. Here we will just mention a small claim that is occasionally useful.

**Fact A.3.16.** Let $X$ have the binomial distribution $B(n, p)$. Then $\Pr [X \geq k] \leq \binom{n}{k} p^k$.

*Proof.* Let $S$ be the collection of all sets of $k$ trials, and note that $|S| = \binom{n}{k}$. If $X \geq k$ then there must be some set of $k$ trials that succeeded. Thus

$$\Pr [X \geq k] \leq \Pr [\exists S \in S \text{ such that all trials in } S \text{ succeeded}]$$
$$\leq \sum_{S \in S} \Pr [\text{all trials in } S \text{ succeeded}] \quad \text{(by the union bound, Fact A.3.8)}$$
$$= |S| \cdot p^{|S|}$$
$$= \binom{n}{k} p^k \quad \square$$

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.

\[
\begin{array}{ll}
\text{# trials strictly before first head} & \text{# trials up to and including first head} \\
\Pr [X = k] = (1 - p)^k p & \forall k \geq 0 \\
\Pr [X \geq k] = (1 - p)^k & \forall k \geq 0 \\
E [X] = \frac{1}{p} - 1 & \forall k \geq 0
\end{array}
\]

\[
\begin{array}{ll}
\Pr [X = k] = (1 - p)^{k-1} p & \forall k \geq 1 \\
\Pr [X \geq k] = (1 - p)^{k-1} & \forall k \geq 1 \\
E [X] = \frac{1}{p} & \forall k \geq 1
\end{array}
\]

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

- \( \Pr [H] = p \)
- \( \Pr [TH] = (1 - p)p \)
- \( \Pr [TTH] = (1 - p)^2 p \)

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

\[
\Pr [X = k] = (1 - p)^{k-1} p.
\]

We now prove two properties of \( X \) that were mentioned above.

**Claim A.3.17.** 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.3.18.** Let \( X \) be a geometric random variable with parameter \( p \). Then \( E [X] = 1/p \).

**Proof of Fact A.3.18.**

\[
E [X] = \sum_{k \geq 1} \Pr [X \geq k] \quad \text{(by Fact A.3.10)}
\]

\[
= \sum_{k \geq 1} (1 - p)^{k-1} \quad \text{(by Claim A.3.17)}
\]

\[
= \frac{1}{1 - (1 - p)} = \frac{1}{p}.
\]

Here we have used the formula for a geometric series discussed in Fact A.2.2.

**Keener Kwestion A.3.19.** What is \( \Pr [X = \infty] \)?

References: (Anderson et al., 2017, Example 1.16).
A.3.4 Markov’s Inequality

Fact A.3.20 (Markov’s Inequality). Let $Y$ be a random variable that only takes non-negative values. Then, for all $a > 0$,

$$\Pr [ Y \geq a ] \leq \frac{\mathbb{E}[Y]}{a}.$$  

References:  
(Lehman et al., 2018, Theorem 20.1.1), (Anderson et al., 2017, Theorem 9.2), (Cormen et al., 2001, Exercise C.3-6), (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.3.21. Let $Y$ be a random variable that only takes nonnegative values. Then, for all $b > 0$,

$$\Pr [ Y \geq b \cdot \mathbb{E}[Y] ] \leq \frac{1}{b}.$$  

Proof. Simply apply Fact A.3.20 with $a = b \cdot \mathbb{E}[Y]$. \qed

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.3.22. Let $q \geq 2$ be an arbitrary integer. Let $X$ be uniformly random in $[q]$. Then, for any fixed integer $y$,

$$(X + y) \text{ mod } q$$

is also uniformly random in $[q]$.

Proof. Fix any any $z \in [q]$. By the principle of “taking mod everywhere” (Fact A.2.10), we have

$$\Pr [(X + y) \text{ mod } q = z] = \Pr [ X = (z - y) \text{ mod } q] = 1/q$$

since $X$ is uniform. \qed

Corollary A.3.23. 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

$$(X + Y) \text{ mod } q$$

is also uniformly random in $[q]$. 
A.4 Exercises

Exercise A.1. For any events $A$ and $B$, prove that
\[ \Pr[A \cap \overline{B}] \geq \Pr[A] - \Pr[B]. \]

Exercise A.2. The usual form of the union bound is that
\[ \Pr[E_1 \cup \cdots \cup E_n] \leq \sum_{i=1}^{n} \Pr[E_i] \]
for any events $E_i$. Eq. (A.3.1) states the equivalent form
\[ \Pr[\overline{E_1} \cap \cdots \cap \overline{E_n}] \geq 1 - \sum_{i=1}^{n} \Pr[E_i]. \]
Prove the latter inequality using the former one.

Exercise A.3. Let $A$ and $B$ be events with $\Pr[B] > 0$. Prove that $\Pr[A \cap B] \leq \Pr[A \mid B]$.

Exercise A.4. Let $E$ and $F$ be events with $\Pr[E] < 1$. Prove that $\Pr[F] \leq \Pr[E] + \Pr[F \mid \overline{E}]$.

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.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 $G$.

Bibliography


