Introduction

Reading:
- CLRS: “Sorting in Linear Time” 8
- GT: “Sorting, Sets, and Selection” 4.4-4.5

Motivation:
- Insertion sort is worst-case $O(n^2)$.
- Other algorithms are worst-case $\Theta(n \log n)$.
  - e.g. mergesort and heapsort
- Can we do better?
  - Use decision trees to model sorting algorithms.
  - Decisions trees have worst-case $\Omega(n \log n)$.
  - Go outside decision tree model to do better ($\Theta(n)$).

Comparison Sorts

Recall: Sorting
- Input: A sequence of $n$ values $a_1, a_2, \ldots, a_n$.
- Output: A permutation $b_1, b_2, \ldots, b_n$ of $a_1, a_2, \ldots, a_n$ such that $b_1 \leq b_2 \leq \ldots \leq b_n$.
- Instance: 3, 8, 2, 5.

In a comparison sort algorithm, the sorted order is determined by a sequence of comparisons between pairs of elements.
- Insertion sort, selection sort, bubble sort, quicksort, mergesort, and heapsort are comparison sorts.

Using a decision tree, we show that every comparison sort requires $\Omega(n \log n)$ comparisons in the worst-case.

Decision Tree

- Represents every sequence of comparisons that an algorithm might make on an input of size $n$.
- Nodes annotated with the orderings consistent with the comparisons made so far.
- Edges denote the result of a single comparison.
- Total order at leaves.
Claim
The depth of a decision tree for a given value of \( n \) is \( \Omega(n \log n) \).

Proof.
There are \( n! \) leaves. A tree of height \( h \) has at most \( 2^{h+1} \) nodes. So

\[
2^{h+1} \geq n!
\]

\[
h + 1 \geq \log_2 n! = \log_2 (1 \cdot 2 \cdot \ldots \cdot n)
\]

\[
= \log_2 1 + \log_2 2 + \ldots + \log_2 n
\]

\[
> (n/2) \log_2 (n/2)
\]

\[
h \in \Omega(n \log n)
\]

Theorem
Every comparison sort requires \( \Omega(n \log n) \) comparisons in the worst-case.

Proof.
Given a comparison sort, we look at the decision tree it generates on inputs of size \( n \).

- Each path from root to leaf is one possible sequence of comparisons.
- Length of the path is the number of comparisons for that instance.
- Height of the tree is the worst-case path length (number of comparisons).

Height of the tree is \( \Omega(n \log n) \) by the previous claim. Hence, every comparison sort requires \( \Omega(n \log n) \) comparisons.

Transitivity: Indirect Comparisons
- If \( a < b \) and \( b < c \), we indirectly know that \( a < c \).
- Quicksort splits instance into sets \( A, C \) based on a pivot \( b \).
  - \( A \) is such that \( a \leq b \), for \( a \in A \).
  - \( C \) is such that \( b \leq c \), for \( c \in C \).
  - So \( a \leq c \), for \( a \in A \) and \( c \in C \) by transitivity.
- Algorithms doing better than \( \Omega(n \log n) \) in the worst-case
  - Do not pivot on an element of the instance.
    - Escapes decision tree model.
    - Use knowledge of problem domain to choose pivot independent of particular instance.

Bucket Sort (Counting Sort)
Assume keys are integers in ranging from 0 to \( N - 1 \).
- Pivots are 0, 1, ..., \( N - 1 \).
- One set (bucket) per possible key.

```
Algorithm BucketSort(A,N)
Let S be an empty list
Let B[0..N-1] be an array of empty lists
for i ← 0 to A.length-1 do
for j ← 0 to N-1 do
  for each element x of B[j] do // in order
    append x to S
return S
```

Time Complexity: \( \Theta(n + N) \)
Bucket Sort (cont’d)

- In practice, we compact $B[0], \ldots, B[N-1]$ in one array by
  - precomputing their maximum sizes (offsets) and
  - using an array of their current lengths.
- Bucket sort is **stable**:
- Stable sorts can sort a class list in two passes.
  - Sort the list by first name.
  - Then sort the list by last name.
- Which algorithms are stable?

Radix Sort

Intuitively, we sort $n$ integers with at most $d$-digits by
- binning according to their most significant digit,
- sorting each pile recursively, and
  - i.e. split on the 2nd most significant digit, then 3rd most, etc.
- merging the results.

Too slow because there are too many piles.

```
821  614  818  817 341  222  806  801
801 806 817 818 821 341 614 806
```

Radix Sort (cont’d)

Instead, we use a stable sort to get rid of the piles.
- Sort digit by digit, from the least significant digit to the most significant digit.

```
Algorithm RadixSort(A,d)
    for i ← 0 to d-1 do
        sort A on digit i using BucketSort
```

Radix Sort Correctness

**Claim**
After the $i$th iteration, the values are sorted by their last $i$ digits.

**Proof.**
Induct on $i$. Trivially true when $i = 1$. So consider $i > 1$. Let $x$ and $y$ be numbers such that the
- the last $i$ digits $x_i, x_{i-1}, \ldots, x_2, x_1$ of $x$ are less than
- the last $i$ digits $y_i, y_{i-1}, \ldots, y_2, y_1$ of $y$.

We need to show that $x$ comes before $y$.
- If $x_i = y_i$, then $x_{i-1}, \ldots, x_2, x_1 < y_{i-1}, \ldots, y_2, y_1$. So $x$ is ordered before $y$ on the $(i-1)$th iteration. BucketSort preserves this order because it is stable.
- If $x_i < y_i$ then BucketSort orders $x$ and $y$ correctly on this iteration.
Radix Sort Complexity

We call BucketSort \(d\) times.

- BucketSort takes \(\Theta(n + N)\) time.

So the complexity of RadixSort is \(\Theta(d(n + N))\).

- RadixSort is \(O(n)\), if the range of values is small
  - i.e. \(d\) is constant and \(N \in O(n)\)