Lecture 5
Sorting lower bounds and $O(n)$-time sorting
Announcements

• HW3 posted Today
• HW3 is not for submission, but it is the best way to prepare for your midterm.

• BTW, Midterm 1 is a week from today (but you already knew that...)

• Please send any OAE letters to our head CAs (rmu, dkm0713) ASAP.
Last time: Quicksort

We want to sort this array.

First, pick a “pivot.”
(There are a few ways to do this...)

Next, partition the array into “bigger than 5” or “less than 5”

L = array with things smaller than A[pivot]
R = array with things larger than A[pivot]

Recurse on L and R:
Last time: pseudocode

- **QuickSort(A):**
  - **If** len(A) <= 1:
    - **return**
  - Pick some x = A[i] **at random.** Call this the **pivot.**
  - **PARTITION** the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - **QuickSort(L)**
  - **QuickSort(R)**
Last time: Analysis
- by counting comparisons

\[ P(X_{a,b} = 1) \]

= probability \(a, b\) are ever compared
= probability that one of \(a, b\) are picked first out of all of the \(b - a + 1\) numbers between them.

\[ = \frac{2}{b - a + 1} \]

Total Comparisons

\[ = \sum_{a=1}^{n} \sum_{b=a+1}^{n} \frac{2}{b - a + 1} \leq 2n \ln n \]
Sorting

- We’ve seen a few $O(n \log(n))$-time algorithms.
  - MERGESORT has worst-case running time $O(n \log(n))$
  - QUICKSORT has expected running time $O(n \log(n))$

Can we do better?

Depends on whom you ask...
Today: can you beat $n \log n$?

• Comparison-based sorting model
  • This includes MergeSort, QuickSort, InsertionSort
  • We’ll see that any algorithm must take at least $\Omega(n \log(n))$ steps.

• Faster sorting in other models
  • StickSort – time $O(1)$
  • BucketSort and RadixSort - time $O(n)$
An $O(1)$-time algorithm for sorting: StickSort

- Problem: sort these $n$ sticks by length.

- Algorithm:
  - Drop them on a table.

- Now they are sorted this way.
That may have been unsatisfying

• But **StickSort** does raise some important questions:
  • What is our model of computation?
    • Input: array
    • Output: sorted array
    • Operations allowed: comparisons

  -vs-

  • Input: sticks
  • Output: sorted sticks in vertical order
  • Operations allowed: dropping on tables

• What are reasonable models of computation?
Today: can you beat $n \log n$?

- Comparison-based sorting model
  - This includes MergeSort, QuickSort, InsertionSort
  - We’ll see that any algorithm must take at least $\Omega(n \log(n))$ steps.

- Faster sorting in other models
  - StickSort – time $O(1)$
  - BucketSort and RadixSort - time $O(n)$
Comparison-based sorting

NO.

CAN'T BEAT NLOG(N)
Comparison-based sorting model

There is a genie who knows what the right order is. The genie can answer YES/NO questions of the form: is [this] bigger than [that]?

Want to sort these items. There’s some ordering on them, but we don’t know what it is.

Algorithm

The algorithm’s job is to output a correctly sorted list of all the objects.

There is a genie who knows what the right order is.

The genie can answer YES/NO questions of the form: is [this] bigger than [that]?
All the sorting algorithms we have seen work like this.

eg, QuickSort:

```
7 6 3 5 1 4 2
```

Is 7 bigger than 5? **YES**

Is 6 bigger than 5? **YES**

Is 3 bigger than 5? **NO**

etc.
Lower bound of $\Omega(n \log(n))$.

• Theorem:
  • Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.
  • Any randomized comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps in expectation.

• How might we prove this?

  1. Consider all comparison-based algorithms, one-by-one, and analyze them.

  2. Don’t do that. Instead, argue that all comparison-based sorting algorithms give rise to a decision tree. Then analyze decision trees.
Decision trees

Sort these three things.

etc...
All comparison-based algorithms look like this

Example: Sort these three things using QuickSort.

In either case, we’re done (after some base case stuff and returning recursive calls).

Then we’re done (after some base-case stuff)
All comparison-based algorithms have an associated decision tree.

The leaves of this tree are all possible orderings of the items: when we reach a leaf we return it.

What does the decision tree for MERGESORTING four elements look like?

Running the algorithm on a given input corresponds to taking a particular path through the tree.
What’s the runtime on a particular input?

At least the number of comparisons that are made on that input.

If we take this path through the tree, the runtime is $\Omega$(length of the path).
What’s the **worst-case** runtime?

At least $\Omega(\text{length of the longest path})$. 
How long is the longest path?

We want a statement: in all such trees, the longest path is at least _____

Think-Pair-Share!
How long is the longest path?

We want a statement: in all such trees, the longest path is at least _____

• This is a binary tree with at least _____ leaves.

• The shallowest tree with n! leaves is the completely balanced one, which has depth _____.

• So in all such trees, the longest path is at least log(n!).

• log(n!) = \Omega(n \log(n))
  [on the board / Stirling’s approx*].

Conclusion: the longest path has length at least \Omega(n \log(n)).
Lower bound of $\Omega(n \log(n))$.

• Theorem:
  • Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.

• Proof recap:
  • Any deterministic comparison-based algorithm can be represented as a decision tree with $n!$ leaves.
  • The worst-case running time is at least the depth of the decision tree.
  • All decision trees with $n!$ leaves have depth $\Omega(n \log(n))$.
  • So any comparison-based sorting algorithm must have worst-case running time at least $\Omega(n \log(n))$. 
Aside:
What about randomized algorithms?

• For example, QuickSort?

• Theorem:
  • Any randomized comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps in expectation.

• Proof:
  • see lecture notes
  • (same ideas as deterministic case)

\end{Aside}
So that’s bad news.

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps.

- Theorem:
  - Any randomized comparison-based sorting algorithm must take $\Omega(n \log(n))$ steps in expectation.

But look on the bright side!
The good news:

**MergeSort** is optimal!

- This is one of the cool things about lower bounds like this: we know when we can declare victory!
Today: can you beat $n \log n$?

• Comparison-based sorting model
  • This includes MergeSort, QuickSort, InsertionSort
  • We’ll see that any algorithm must take at least $\Omega(n \log(n))$ steps.

• Faster sorting in other models
  • StickSort – time $O(1)$
  • BucketSort and RadixSort - time $O(n)$
2 minute pep talk
Today: can you beat $n \log n$?

- Comparison-based sorting model
  - This includes MergeSort, QuickSort, InsertionSort
  - We’ll see that any algorithm must take at least $\Omega(n \log(n))$ steps.

- Faster sorting in other models
  - StickSort – time $O(1)$
  - BucketSort and RadixSort - time $O(n)$
Beyond comparison-based sorting algorithms

YES!

WE CAN DO WAY BETTER!
Another model of computation

• The items you are sorting have meaningful values.

Instead of

9 6 3 5 2 1 2
Why might this help?

BucketSort:

Note: this is a simplification of what CLRS calls “BucketSort”

Implement the **buckets** as linked lists. They are first-in, first-out. This will be useful later.

In time $O(n)$.

Concatenate the buckets!

SORTED!
Issues

• Need to be able to know what bucket to put something in.
  • Where does \( \text{fire truck} \) go?
  • That’s okay for now: it’s part of the model.

• Need to know what values might show up ahead of time.

\[
\begin{array}{cccccc}
2 & 12345 & 13 & 2^{1000} & 50 & 100000000 & 1 \\
\end{array}
\]

• Space...
One solution: **RadixSort**

Say we’re sorting integers.

- **Idea:** BucketSort on the least-significant digit first, then the next least-significant, and so on.

**Step 1: BucketSort on LSB:**

```
21  345  13  101  50  234  1
```

```
0   1   2   3   4   5   6   7   8   9
```

```
50  21  101  21  13  234  345  345
```
Step 2: BucketSort on the 2\textsuperscript{nd} digit

\begin{align*}
50 & \quad 21 & \quad 101 & \quad 1 & \quad 13 & \quad 234 & \quad 345 \\
\begin{array}{c}
\begin{array}{c}
1 \\
101
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
13
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
21
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
234
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
345
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
50
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
0
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
1
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
2
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
3
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
4
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
5
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
6
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
7
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
8
\end{array}
\end{array} & 
\begin{array}{c}
\begin{array}{c}
9
\end{array}
\end{array}
\end{align*}
Step 3: BucketSort on the 3\textsuperscript{rd} digit

It worked!!
Why does this work?

Original array:

| 21 | 345 | 13 | 101 | 50 | 234 | 1 |

Next array is sorted by the first digit.

| 50 | 21 | 101 | 1 | 13 | 234 | 345 |

Next array is sorted by the first two digits.

| 101 | 01 | 13 | 21 | 234 | 345 | 50 |

Next array is sorted by all three digits.

| 001 | 013 | 021 | 050 | 101 | 234 | 345 |

Sorted array
Formally...

• Argue by induction.
• Inductive hypothesis:

Or at least a little formally!

Lucky the lackadaisical lemur
Why does this work?

Original array:

![Original array]

Next array is sorted by the first digit.

![Next array sorted by first digit]

Next array is sorted by the first two digits.

![Next array sorted by first two digits]

Next array is sorted by all three digits.

![Next array sorted by all three digits]

Sorted array
Formally...

• Argue by induction.
• Inductive hypothesis:
  • After the k’th iteration, the array is sorted by the first k least-significant digits.
• Base case:
  • “Sorted by 0 least-significant digits” means not sorted.
• Inductive step:
  • (See lecture notes or CLRS)
• Conclusion:
  • After the d’th iteration, the array is sorted by the d least-significant digits. Aka, it’s sorted.
What is the running time?

• Say they are \textbf{d-digit} numbers.
  • There are \textit{d} iterations.
  • Each iteration takes time \(O(n + 10) = O(n)\)
• Total time: \(O(nd)\).

• Say the biggest integer is \(M\). What is \(d\)?
  • \(d = \lfloor \log_{10}(M) \rfloor + 1\)
  • so \(O(nd) = O(n \log_{10}(M))\).

Can we do better?

what if \(M = n\)?
Trade-offs...

• RadixSort works with any base.
• Before we did it base $r=10$.
• But we could do it base $r=2$ or $r=20$ just as easily.

• Running time for general $r$ and $M$?

Think-Pair-Share!
Reminder: running time with base 10

- Say they are d-digit numbers.
  - There are d iterations.
  - Each iteration takes time $O(n + 10) = O(n)$
- Total time: $O(nd)$.
- Say the biggest integer is $M$. What is $d$?
  - $d = \lfloor \log_{10}(M) \rfloor + 1$
  - so $O(nd) = O(n \log_{10}(M))$. 

The “10” is because we are working base 10.
Trade-offs...

• RadixSort works with any base.
• Before we did it base $r=10$.
• But we could do it base $r=2$ or $r=20$ just as easily.

• Running time for general $r$ and $M$?
  • \([On board]\)
  • If we choose $r = n$, running time is
    \[ T(n) = O(n \cdot \lfloor \log_n(M) \rfloor) \]
    • If $M = O(n)$, $T(n) = O(n)$. Awesome!
    • If $M = \Omega(n^n)$, $T(n) = O(n^2)$...

Choosing $r = n$ is pretty good. What’s the optimal choice of $r$?

Ollie the over-achieving ostrich
Trade-offs ctd...

• There are $n$ numbers, biggest one is $M$.
• What should we choose for $r$ (in terms of $M,n$)?

There’s some sweet spot... (and maybe it’s growing with $M$ and $n$?)
The story so far

• If we use a comparison-based sorting algorithm, it MUST run in time $\Omega(n\log(n))$.

• If we assume a bit of structure on the values, we have an $O(n)$-time sorting algorithm.

Why would we ever use a comparison-based sorting algorithm??
Why would we ever use a comparison-based sorting algorithm?

• Lots of precision...

  • We can compare these pretty quickly (just look at the most-significant digit):
    • $\pi = 3.14$.
    • $e = 2.78$.
  • But to do RadixSort we’d have to look at every digit.
  • This is especially problematic since both of these have infinitely many digits...

• RadixSort needs extra memory for the buckets.
  • Not in-place

• I want to sort emoji by talking to a genie.
  • RadixSort makes more assumptions on the input.

Even with integers, if the biggest one is really big, RadixSort is slow.
Recap

• How difficult a problem is depends on the model of computation.
• How reasonable a model of computation is is up for debate.

• Comparison-based sorting model
  • This includes MergeSort, QuickSort, InsertionSort
  • Any algorithm in this model must use at least \( \Omega(n \log(n)) \) operations.

• But if we are sorting small integers (or other reasonable data):
  • BucketSort and RadixSort
  • Both run in time \( O(n) \)
Next time

- Binary search trees!
- Balanced binary search trees!

Before next time

- Go to section
- Send OAE letters