Lecture 6

Sorting lower bounds on O(n)-time sorting
Announcements

• HW2 due Friday

• Please send any OAE letters to Luna Frank-Fischer (luna16@stanford.edu) by April 28.
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 who you ask...
An O(1)-time algorithm for sorting: StickSort

• Problem: sort these sticks by length.

• Now they are sorted this way.

• Algorithm:
  Drop them on a table.
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: two (more) models

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

• Another model (more reasonable than the stick model…)  
  • BucketSort and RadixSort  
  • Both run in time \(O(n)\)
Comparison-based sorting

NO.

CAN'T BEAT $\Omega(N\log N)$
Comparison-based sorting algorithms

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.

Is 🐼 bigger than 🚒? 💚

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

is shorthand for “the first thing in the input list”
All the sorting algorithms we have seen work like this.

eg, QuickSort:

Pivot!

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

5 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.

- Coffee ≤ Firetruck

  - Coffee ≤ Firetruck
    - Coffee ≤ Firetruck
      - Coffee ≤ Firetruck
        - Coffee ≤ Firetruck
      - Coffee ≤ Firetruck
    - Coffee ≤ Firetruck
  - Coffee ≤ Firetruck

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

Example: Sort these three things using QuickSort.

Then we’re done (after some base-case stuff)

In either case, we’re done (after some base case stuff and returning recursive calls).
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 $\Omega(\text{length of the path})$. 
What’s the worst-case runtime?

At least $\Omega$(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 _____

• 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 \( \log(n!) \).

• \( n! \) is about \( (n/e)^n \) (Stirling’s formula).

• \( \log(n!) \) is about \( n \log(n/e) = \Omega(n \log(n)) \).

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:
  • 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:
  - at the end of today if time
  - otherwise see lecture notes
  - (same ideas as deterministic case)

\end{Aside}
So, **MergeSort** is optimal!

• This is one of the cool things about **lower bounds** like this: we know when we can declare victory!

**But what about StickSort?**

• **StickSort** can’t be implemented as a comparison-based sorting algorithm. So these lower bounds don’t apply.
• **But StickSort was kind of dumb.**

But might there be another model of computation that’s **less dumb**, in which we can **sort faster**? Especially if I have to spend time cutting all those sticks to be the right size!
Beyond comparison-based sorting algorithms
Another model of computation

- The items you are sorting have **meaningful values**.

9 6 3 5 2 1 2

instead of

😊  trúc rtle 🔥 กาแฟ 🍕 🏈
Why might this help?

BucketSort:

Implement the buckets as linked lists. They are first-in, first-out.

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

Conatenate the buckets!

SORTED! In time O(n).
Issues

• Need to be able to know what bucket to put something in.
  • That’s okay for now: it’s part of the model.
• Need to know what values might show up ahead of time.
One solution: **RadixSort**

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

**Step 1:** BucketSort on LSB:
Step 2: BucketSort on the 2\textsuperscript{nd} digit
Step 3: BucketSort on the 3\textsuperscript{rd} digit

It worked!!
Why does this work?

Original array:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>345</td>
<td>13</td>
<td>101</td>
<td>50</td>
<td>234</td>
<td>1</td>
</tr>
</tbody>
</table>

Next array is sorted by the first digit.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>21</td>
<td>101</td>
<td>1</td>
<td>13</td>
<td>234</td>
</tr>
</tbody>
</table>

Next array is sorted by the first two digits.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>01</td>
<td>13</td>
<td>21</td>
<td>234</td>
<td>345</td>
<td>50</td>
</tr>
</tbody>
</table>

Next array is sorted by all three digits.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>013</td>
<td>021</td>
<td>050</td>
<td>101</td>
<td>234</td>
<td>345</td>
</tr>
</tbody>
</table>
Formally...

• Argument via loop invariant (aka induction).
• Loop Invariant:

This is the outline of a proof, not a formal proof.

Make this formal! (or see lecture notes).

Lucky the lackadaisical lemur

Ollie the over-achieving ostrich
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...

• Argument via loop invariant (aka induction).

• Loop Invariant:
  • 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:
  • (You fill in...)

• Termination:
  • After the d’th iteration, the array is sorted by the d least-significant digits. Aka, it’s sorted.
What is the running time?

• Depends on how many digits the biggest number has.
  • Say d-digit numbers.
• There are d iterations
• Each iteration takes time \(O(n + 10)\)
  • We can change the 10 into an “r:” this is the “radix”
  • Example: if \(r = 2\), we write everything in binary and only have two buckets.
  • Example: If \(r = 10000000\), we write everything base-10000000 and have 10000000 buckets.
  • Example: if \(r = n\), we write everything in base-n and have \(n\) buckets.
• Time is \(O(d(n+r))\).
• If \(d = O(1)\) and \(r = O(n)\), running time \(O(n)\).

How big can the biggest number be if \(d = O(1)\) and \(r = n\)?

So this is a \(O(n)\)-time sorting algorithm!
The story so far

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

• If we assume that we can do a little more than compare 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?

• d might not be “constant.” (aka, it might be big)

<table>
<thead>
<tr>
<th>π</th>
<th>123456</th>
<th>e</th>
<th>140!</th>
<th>2^n</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>\frac{123456}{987654}</td>
<td>e</td>
<td>140!</td>
<td>2.1234123</td>
<td>2^n</td>
<td>42</td>
</tr>
</tbody>
</table>

• We can compare these pretty quickly (just look at the most-significant digit):
  • π = 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.
Do we have time for the lower bound on randomized algorithms?
If so...

• Recall the lower bound for a deterministic algorithm.

• The longest path in this tree has length $\Omega(n\log(n))$.

• The running time of the algorithm is at least the length of this path.
A different model

- How about a deterministic algorithm on a random input?
- Not worst-case model.
- Not our randomized algorithm model either.

- The longest path in this tree has length $\Omega(n \log(n))$.
- The running time of the algorithm is at least the average length of this path.

So a deterministic algorithm must take time $\Omega(n \log(n))$ even on random inputs.
This is a pretty strong statement!

• Before:
  
  If an adversary gets to pick the input, we need time $\Omega(n \log(n))$.

• Now:

  If the input is chosen randomly, we **still** need time $\Omega(n \log(n))$. 
But what does that model have to do with anything?

It turns out that in this case,

Deterministic algorithm on random input does at least as well as Randomized algorithm on random input does at least as well as Randomized algorithm on worst-case input

And we just showed that this didn’t do very well.

A deterministic algorithm must take time $\Omega(n\log(n))$ even on random inputs.

A randomized algorithm must take time $\Omega(n\log(n))$ on worst-case inputs.

This is what we wanted.
Recap

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

• **StickSort** can sort sticks in $O(1)$ time.
• **RadixSort** can sort smallish integers in $O(n)$ time.
• If we want to sort emoji (or arbitrary-precision numbers), we require $\Omega(n\log(n))$ time (like **MergeSort**).

Next Time

• Binary search trees