Lecture 6

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

• See “In-person Logistics” Ed post for changes to the course

• About that homework 2 ...
  Tarsiers → Quokkas
Everyone can succeed in this class!

1. Work hard
2. Work smart
3. Ask for help
Roadmap

- **Sorting**
  - 5 lectures
  - Randomized Algs
  - Asymptotic Analysis
  - Recurrences

- **Data structures**
  - 2 lectures
  - More detailed schedule on the website!

- **Greedy Algs**
  - 9 lectures
  - Divide and conquer
  - Longest, Shortest, Max and Min...

- **Dynamic Programming**
  - 1 lecture

- **Graphs!**
  - The Future!

- **Divide and conquer**

- **Randomized Algs**

- **Asymptotic Analysis**

- **Recurrences**

- **MIDTERM**

- **FINAL**
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 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: 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...)
  • CountingSort and RadixSort
  • Both run in time $O(n)$
Comparison-based sorting

NO.

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

- You want to sort an array of items.
- You can’t access the items’ values directly: you can only compare two items and find out which is bigger or smaller.
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.

Algorithm

Is bigger than?

Yes

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

The genie can answer YES/NO questions of the form: is [this] bigger than [that]?

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

eg, QuickSort:

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

Pivot!

- 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.

This covers all the sorting algorithms we know!!!
Decision trees

Sort these three things.

1. 😊 ≤ 🚒? YES
2. 😊 ≤ 😏? NO
3. ☕ ≤ 🚒? YES
4. ☕ ≤ 🚒? NO

etc...

14
Decision trees

- Internal nodes correspond to yes/no questions.
- Each internal node has two children, one for “yes” and one for “no.”
- Leaf nodes correspond to outputs.
  - In this case, all possible orderings of the items.
- Running an algorithm on a particular input corresponds to a particular path through the tree.
Comparison-based algorithms look like decision trees.

Example: Sort these three things using QuickSort.

- Return
- Pivot!

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

In either case, we’re done (after some base case stuff and returning recursive calls).
Q: What’s the runtime on a particular input?

A: At least the length of the path from the root to the corresponding leaf.

If we take this path through the tree, the runtime is $\Omega$ (length of the path).
Q: What’s the worst-case runtime?
A: 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 ____ n! ____ leaves.

• The shallowest tree with n! leaves is the completely balanced one, which has depth ____ log(n!)____.

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

• n! is about (n/e)^n (Stirling’s approx.*).
• log(n!) is about n log(n/e) = Ω(n log(n)).

Conclusion: the longest path has length at least Ω(n log(n)).

*Stirling’s approximation is a bit more complicated than this, but this is good enough for the asymptotic result we want.
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:
  • (same ideas as deterministic case)
  • (you are not responsible for this proof in this class)
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.
On the bright side, **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 silly.

Can we do better?

- Is there another model of computation that’s less silly than the StickSort model, in which we can sort faster than nlog(n)?
Beyond comparison-based sorting algorithms
Another model of computation

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

\[
\begin{array}{cccccccc}
9 & 6 & 3 & 5 & 2 & 1 & 2 \\
\end{array}
\]

instead of

\[
\begin{array}{cccccccc}
\text{😊} & \text{🐼} & \text{🐢} & \text{🚒} & \text{☕} & \text{🍕} & \text{🏈} \\
\end{array}
\]
Pre-lecture exercise

• How long does it take to sort n people by their month of birth?

• [discussion]
Another model of computation

• The items you are sorting have meaningful values.

9 6 3 5 2 1 2

instead of

😊️熊猫turtle消防车一杯咖啡比萨足球
Why might this help?

CountingSort:

9 6 3 5 2 1 2

SORTED!

In time O(n).

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

Concatenate the buckets!
Assumptions

• Need to be able to know what bucket to put something in.
  • We assume we can evaluate the items directly, not just by comparison
• Need to know what values might show up ahead of time.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12345</td>
<td>13</td>
<td>$2^{1000}$</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Need to assume there are not too many such values.
RadixSort

• For sorting integers up to size M
  • or more generally for lexicographically sorting strings

• Can use less space than CountingSort

• Idea: CountingSort on the least-significant digit first, then the next least-significant, and so on.
Step 1: CountingSort on least significant digit

21 345 13 101 50 234 1

50
101
21
13
234
345
Step 2: CountingSort on the 2nd least sig. digit

101 1 13 21 234 345 50
Step 3: CountingSort on the 3rd least sig. 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>
<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>
<td></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>
<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>
<td>345</td>
<td></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>
<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>
<td></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>
<th></th>
</tr>
</thead>
</table>
| 001 | 013 | 021 | 050 | 101 | 234 | 345 | Sorted array
To prove this is correct...

• What is the inductive hypothesis?

Think-Share Terrapins
Think: 1 min (wait)
Share: 1 min (on chat)
RadixSort is correct

• 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 yet sorted, so the IH holds for k=0.

• Inductive step:
  • TO DO

• Conclusion:
  • The inductive hypothesis holds for all k, so after the last iteration, the array is sorted by all the digits. Hence, it’s sorted!
**Inductive step**

- **Need to show:** if IH holds for \( k=i-1 \), then it holds for \( k=i \).
  - Suppose that after the \( i-1 \)’st iteration, the array is sorted by the first \( i-1 \) least-significant digits.
  - Need to show that after the \( i \)’th iteration, the array is sorted by the first \( i \) least-significant digits.

**Inductive hypothesis:**
After the \( k \)'th iteration, the array is sorted by the first \( k \) least-significant digits.

---

**EXAMPLE:** \( i=2 \)

IH: this array is sorted by first digit.

<table>
<thead>
<tr>
<th></th>
<th>050</th>
<th>021</th>
<th>101</th>
<th>002</th>
<th>013</th>
<th>234</th>
<th>345</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>002</th>
<th>101</th>
<th>013</th>
<th>021</th>
<th>234</th>
<th>345</th>
<th>050</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Want to show: this array is sorted by 1\(^{st}\) and 2\(^{nd}\) digits.
Proof sketch...

proof on next (skipped) slide

Want to show: after the i’th iteration, the array is sorted by the first i least-significant digits.

• Let \( x = [x_d x_{d-1} \ldots x_2 x_1] \) and \( y = [y_d y_{d-1} \ldots y_2 y_1] \) be any \( x, y \).

• Suppose \( [x_i x_{i-1} \ldots x_2 x_1] < [y_i y_{i-1} \ldots y_2 y_1] \).

• Want to show that \( x \) appears before \( y \) at end of i’th iteration.

• **CASE 1**: \( x_i < y_i \)
  
  • \( x \) is in an earlier bucket than \( y \).

Aka, we want to show that for any \( x \) and \( y \) so that \( x \) belongs before \( y \), we put \( x \) before \( y \).
Proof sketch...
proof on next (skipped) slide

- Let \( x = [x_d x_{d-1} \ldots x_2 x_1] \) and \( y = [y_d y_{d-1} \ldots y_2 y_1] \) be any \( x, y \).
- Suppose \([x_i x_{i-1} \ldots x_2 x_1] < [y_i y_{i-1} \ldots y_2 y_1]\).
- Want to show that \( x \) appears before \( y \) at end of \( i \)'th iteration.

  - **CASE 1**: \( x_i < y_i \)
    - \( x \) is in an earlier bucket than \( y \).
  
  - **CASE 2**: \( x_i = y_i \)
    - \([x_{i-1} \ldots x_2 x_1] < [y_{i-1} \ldots y_2 y_1]\),
    - \( x \) and \( y \) in same bucket, but \( x \) was put in the bucket first.

Aka, we want to show that for any \( x \) and \( y \) so that \( x \) belongs before \( y \), we put \( x \) before \( y \).

IH: this array is sorted by first digit.

**EXAMPLE: \( i=2 \)**

Want to show: after the \( i \)'th iteration, the array is sorted by the first \( i \) least-significant digits.

Want to show: this array is sorted by \( 1^{\text{st}} \) and \( 2^{\text{nd}} \) digits.
Want to show: after the i’th iteration, the array is sorted by the first i least-significant digits.

- Let \( x = [x_d x_{d-1} \ldots x_2 x_1] \) and \( y = [y_d y_{d-1} \ldots y_2 y_1] \) be any \( x, y \).
- Suppose \( [x_i x_{i-1} \ldots x_2 x_1] < [y_i y_{i-1} \ldots y_2 y_1] \).
- Want to show that \( x \) appears before \( y \) at end of i’th iteration.

**CASE 1:** \( x_i < y_i \).
- \( x \) appears in an earlier bucket than \( y \), so \( x \) appears before \( y \) after the i’th iteration.

**CASE 2:** \( x_i = y_i \).
- \( x \) and \( y \) end up in the same bucket.
- In this case, \( [x_{i-1} \ldots x_2 x_1] < [y_{i-1} \ldots y_2 y_1] \), so by the inductive hypothesis, \( x \) appeared before \( y \) after i-1’st iteration.
- Then \( x \) was placed into the bucket before \( y \) was, so it also comes out of the bucket before \( y \) does.
  - Recall that the buckets are FIFO queues.
  - So \( x \) appears before \( y \) in the i’th iteration.
Inductive step

Inductive hypothesis: After the k’th iteration, the array is sorted by the first k least-significant digits.

• Need to show: if IH holds for k=i-1, then it holds for k=i.
  • Suppose that after the i-1’st iteration, the array is sorted by the first i-1 least-significant digits.
  • Need to show that after the i’th iteration, the array is sorted by the first i least-significant digits.

EXAMPLE: i=2

IH: this array is sorted by first digit.

Want to show: this array is sorted by 1st and 2nd digits.
RadixSort is correct

• 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, so the IH holds for $k=0$.

• Inductive step:
  • TO DO

• Conclusion:
  • The inductive hypothesis holds for all $k$, so after the last iteration, the array is sorted by all the digits. Hence, it’s sorted!
What is the running time? for RadixSorting numbers base-10.

• Suppose we are sorting n d-digit numbers (in base 10).

e.g., n=7, d=3:

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

1. How many iterations are there?

2. How long does each iteration take?

3. What is the total running time?

Think--Share Terrapins
Think: 1 min (wait)
Share: 1 min (on chat)
What is the running time?

Suppose we are sorting \( n \) \( d \)-digit numbers (in base 10).

1. How many iterations are there?
   - \( d \) iterations

2. How long does each iteration take?
   - Time to initialize 10 buckets, plus time to put \( n \) numbers in 10 buckets. \( O(n) \).

3. What is the total running time?
   - \( O(nd) \)
This doesn’t seem so great

• To sort \( n \) integers, each of which is in \{1,2,...,n\}...

• \( d = \lceil \log_{10}(n) \rceil + 1 \)
  • For example:
    • \( n = 1234 \)
    • \( \lceil \log_{10}(1234) \rceil + 1 = 4 \)
    • More explanation on next (skipped) slide.

• Time = \( O(nd) = O(n \log(n)) \).
  • Same as MergeSort!
Aside: why \( d = \lceil \log_{10}(n) \rceil + 1 \) ?

- When we write a number \( x = [x_d x_{d-1} \ldots x_1] \) base 10, that means:
  \[
  x = x_1 + x_2 \cdot 10 + \cdots + x_{d-1} \cdot 10^{d-2} + x_d \cdot 10^{d-1}
  \]
  where \( x_i \in \{0, 1, \ldots, 9\} \)

- Suppose that \( x_d \neq 0 \). Then we have
  - \( x \geq x_d \cdot 10^{d-1} \)
  - \( \log_{10}(x) + 1 - \log_{10}(x_d) \geq d \)
  - \( \log_{10}(x) + 1 > d \)
  - \( \lceil \log_{10}(n) \rceil + 1 \geq d \)

- On the other hand, we also have
  - \( x < (x_d+1) \cdot 10^{d-1} \)
  - \( \log_{10}(x) + 1 - \log_{10}(x_d+1) < d \)
  - \( \log_{10}(x) < d \)
  - \( \lceil \log_{10}(n) \rceil + 1 \leq d \)
Can we do better?

• RadixSort base 10 doesn’t seem to be such a good idea...
• But what if we change the base? (Let’s say base $r$)
• We will see there’s a trade-off:
  • Bigger $r$ means more buckets
  • Bigger $r$ means fewer digits
Example: base 100

Original array:

| 21 | 345 | 13 | 101 | 50 | 234 | 1  |
Example: base 100

Original array:

0021 0345 0013 0101 0050 0234 0001

100 buckets:

00 01 02 34 50 98 99

0101 0001 0013 0021 0234 0345 0050
Example: base 100

100 buckets:

Sorted!
Example: base 100

<table>
<thead>
<tr>
<th>Original array</th>
<th>Sorted array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0021 0345 0013 0101 0050 0234 0001</td>
<td>0101 0001 0013 0021 0234 0345 0050</td>
</tr>
<tr>
<td>0001 0013 0021 0050 0101 0234 0345</td>
<td></td>
</tr>
</tbody>
</table>

Base 100:
- $d=2$, so only 2 iterations.
- 100 buckets

Base 10:
- $d=3$, so 3 iterations.
- 10 buckets

Bigger base means more buckets but fewer iterations.
General running time of RadixSort

- Say we want to sort:
  - n integers,
  - maximum size M,
  - in base r.

- Number of iterations of RadixSort:
  - Same as number of digits, base r, of an integer x of max size M.
  - That is \( d = \lceil \log_r(M) \rceil + 1 \)

- Time per iteration:
  - Initialize r buckets, put n items into them
  - \( O(n + r) \) total time.

- Total time:
  - \( O(d \cdot (n + r)) = O((\lceil \log_r(M) \rceil + 1) \cdot (n + r)) \)

Convince yourself that this is the right formula for d.
Trade-offs

• Given \( n, M \), how should we choose \( r \)?
• Looks like there’s some sweet spot:

\[
\text{Running time: } O\left(\left(\left\lfloor \log_r(M) \right\rfloor + 1 \right) \cdot (n + r)\right)
\]
A reasonable choice: \(r=n\)

- Running time:

\[
O\left( (\lceil \log_r(M) \rceil + 1) \cdot (n + r) \right)
\]

Intuition: balance \(n\) and \(r\) here.

- Choose \(n=r\):

\[
O\left( n \cdot (\lceil \log_n(M) \rceil + 1) \right)
\]

Choosing \(r=n\) is pretty good. What choice of \(r\) optimizes the asymptotic running time? What if I also care about space?

Ollie the over-achieving ostrich
Running time of RadixSort with r=n

- To sort n integers of size at most M, time is
  \[ O(n \cdot ( \lceil \log_n(M) \rceil + 1 )) \]
- So the running time (in terms of n) depends on how big M is in terms of n:
  - If \( M \leq n^c \) for some constant c, then this is \( O(n) \).
  - If \( M = 2^n \), then this is \( O \left( \frac{n^2}{\log(n)} \right) \)
- The number of buckets needed is r=n.
What have we learned?

• RadixSort can sort $n$ integers of size at most $n^{100}$ in time $O(n)$, and needs enough space to store $O(n)$ integers.

• If your integers have size much much bigger than $n$ (like $2^n$), maybe you shouldn’t use RadixSort.

• It matters how we pick the base.
Recap

• How difficult sorting 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 it can handle arbitrary comparable objects. 😊

• If we are sorting small integers (or other reasonable data):
  • CountingSort and RadixSort
  • Both run in time $O(n)$ 😊
  • Might take more space and/or be slower if integers get too big 😞
Next time

• Binary search trees!
• Balanced binary search trees!

Before next time

• Pre-lecture exercise for Lecture 7
  • Remember binary search trees?
CHUCK NORRIS QUICKSORTS STICKS IN TIME $O(1)$