6/27 Lecture Agenda

- Announcements
- Part 2-1: k-Selection
- 10 minute break!
- Part 2-2: RadixSort and the limits of sorting

Announcements!

- Pre-HW1 due tonight, 11:59 PM
- HW1 due Thursday, 11:59 PM (not Wednesday as originally stated)
- Templates and autograders for Problem 6 are available!
- Pre-HW2 out tonight
- HW2 out Wednesday night

Policy for Coding Problems

- The autograder will show the number of test cases successfully passed. However, this does not translate directly to your score for the problem. (Coding problems are worth 8 points like every other HW problem, or possibly 4 points for a hybrid theory + coding problem)

- In general:
  - To get full points, you need to solve *all* test cases.
  - If you solve anywhere between 1 and n-1 test cases, you get 25-75% of the points.
    - That is, we want to put a premium on being *fully* correct, but also give credit for trying.

6/27 Lecture Agenda

- Announcements
- Part 2-1: *k*-Selection
- 10 minute break!
- Part 2-2: RadixSort and the limits of sorting

WORLD 2-1

k-Selection

Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

A Warm-Up Problem...

- Suppose we have a (not necessarily sorted) list of *n* integers.
  - (e.g., [9, 4, 3, 0, 5])

- We wonder:

  - What is the **largest absolute difference** between *any* pair of (not necessarily consecutive) elements in the list?
    - here, the answer is 9: 9 - 0 = 9

  - What is the **smallest absolute difference** between *any* pair of (not necessarily consecutive) elements in the list?
    - here, the answer is 1: e.g., 4 - 3 = 1

- And how do we find these efficiently?

Largest Absolute Difference

- The two most different elements in the list are the smallest and the largest!
- We can find the smallest element in O(n) time by iterating through the list, keeping track of the smallest element we've seen so far.
  - Only O(1) work to check each new element.
- Same idea for the largest element.
- We can do both of these at once, then take the difference! It's O(n) time overall.

Smallest Absolute Difference

- Now it's trickier! The answer doesn't entail just finding the largest or smallest list elements. The pair could be anywhere!

[100, -55, **-7**, 28, 34, **-6**, -38, 144]

- A brute-force O(n^2) strategy would be to check all pairs.

Smallest Absolute Difference

- Now it's trickier! The answer doesn't entail just finding the largest or smallest list elements. The pair could be anywhere!

[100, -55, **-7**, 28, 34, **-6**, -38, 144]

- A brute-force O(n^2) strategy would be to check all pairs.

- We can do better by sorting the list in O(n log n) time, and then checking every consecutive pair.
    - No non-consecutive pair in a sorted list can have the smallest absolute difference, since we could do better by just moving the indices closer to each other...

Smallest Absolute Difference: Can we do better?

- If we have to sort the list, we're stuck at $\Omega(n \log n)$.*
    - *kind of. More coming in the second half!

- Do we *have* to sort the list? It sure seems like we do!

- If you think of a way to solve this problem without sorting, or can prove that the $\Omega(n \log n)$ bound holds, let us know on Ed!

Another Problem: Find The Median

- Why should we care? Why not just take the mean?
    - The median is much less sensitive to outliers!

- Reminder of the definition:

    - In a list of odd length n, the median is the $(n+1)/2$ – th smallest value. (We'll focus on this case.)
        - in [9, 4, 3, 0, 5], it's 4.

    - In a list of even length n, the median is the mean of the $n/2$ – th and $((n/2) + 1)$ – th smallest values.
        - in [2, 0, 2, 0], it's 1.

How do we find the median of an odd-length list?

- One idea: the median is the middle of the sorted list. So sort the list and take the middle element. Running time: O(*n* log *n*).

- Are we done? Can we go home?
    - No! This summer, we are trapped in an algorithm dimension in which we must always try to do better (or prove that we can't).

But doesn't this <u>inherently</u> require sorting?

- It seems like we might be stuck at $\Omega(n \log n)$.
- However..
....
....
....

(suspense builds)

....
....
....

Commercial Break!

Hey, we could use more practice with recurrences! Let's solve this one, which I surely chose completely at random!

$$T(n) \leq T(n/5) + T(7n/10) + n, \; n > 10$$

$$T(n) = 1, \; 1 \leq n \leq 10$$

Note the use of ≤ instead of =. This is fine – we can solve the = version as an upper bound.

$T(n) \le T(n/5) + T(7n/10) + n$

How can we approach this?

- **The Master Theorem** (HW1 Problem 3): nope. This doesn't fit the form.
  - What if we say $T(n/5) \le T(7n/10)$, so we have $T(n) \le 2T(7n/10) + n$?
  - Then we have $a = 2$, $b = 10/7$, $d = 1$, and

    $T(n) = O(n^{\log_{(10/7)} 2}) \approx O(n^{1.94})$

- But this turns out to be a very loose upper bound!

$T(n) \leq T(n/5) + T(7n/10) + n$

What about **top-down unrolling?**

- $$\begin{aligned} T(n) &\leq T(n/5) + T(7n/10) + n \\ &\leq T(n/25) + T(7n/50) + n/5 \\ &\quad + T(7n/50) + T(49n/100) + 7n/10 \\ &\leq \ldots \end{aligned}$$

Like a werewolf under a full moon, this looks like it's only going to get hairier.

NOPE

$T(n) \leq T(n/5) + T(7n/10) + n$

What about bottom-up investigation?

- $T(1), \ldots, T(10) = 1$
- $T(11) \leq T(2.2) + T(7.7) \leq T(3) + T(8) = 2$
- ...
- $T(15) \leq T(3) + T(10.5) \leq T(3) + T(11) = 3$

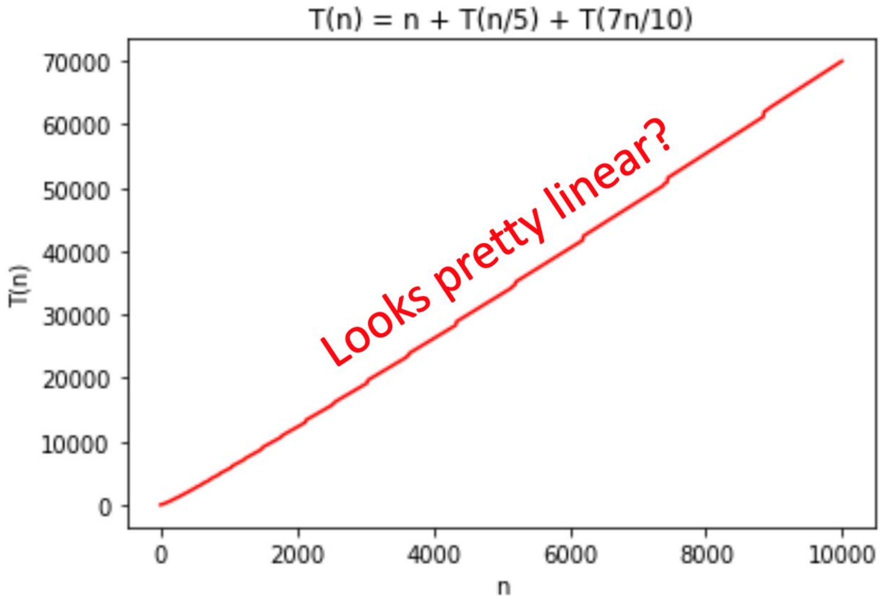
This technique works well when we can spot some obvious pattern. The weird fractions make that unlikely here.

it's fun to stay at the

NOPE

$T(n) \leq T(n/5) + T(7n/10) + n$

What if we use a program to calculate T(n) for a bunch of relatively small values, then make a plot?



But how do we prove it?

Aside: Warning!

- It may be tempting to try to prove this with the inductive hypothesis “T(n) = O(n)”
- But that doesn’t make sense!
- Formally, that’s the same as saying:
  - Inductive Hypothesis for n:
  - There is some $n_0 > 0$ and some $c > 0$ so that, for all $n \geq n_0$, $T(n) \leq c \cdot n$.

- Instead, we should pick c first…

Aside: Warning!

- It may be tempting to try to prove this with the inductive hypothesis "T(n) = O(n)"
- But that doesn't make sense!
- Formally, that's the same as saying:
  - Inductive Hypothesis for n:
  - There is some $n_0 > 0$ and some $c > 0$ so that, for all $n \geq n_0$, $T(n) \leq c \cdot n$.

The IH is supposed to hold for a *specific* n.

But now we are letting n be anything big enough!

- Instead, we should pick c first…

How do we write a proof when we don't know c?

- Go through the proof leaving *c* as a constant.

- Get some idea of what the constraints on *c* are.

- Pick a *c* that actually works.

- Rewrite the proof, using that *c* as if it fell out of the sky or sprang forth fully formed from our heads like Athena.

$T(n) \leq T(n/5) + T(7n/10) + n$

Claim: $T(n) \leq cn$, for all $n \geq 1$. *$n_0 \geq 1$.*

Base case: $T(1) = 1 \leq c * 1$. *OK, so we need $c \geq 1$.*

Inductive step: Suppose the claim holds for $1 \leq n \leq k-1$. We'll show that it also holds for $n = k$.

$$\begin{aligned} T(k) &\leq T(k/5) + T(7k/10) + k \\ &\leq ck/5 + 7ck/10 + k \\ &\leq 9ck/10 + k \end{aligned}$$

Hm, what if we make $c = 10$?

$T(n) \leq T(n/5) + T(7n/10) + n$

Claim: $T(n) \leq cn$, for all $n \geq 1$. *$n_0 \geq 1$.*

Base case: $T(1) = 1 \leq c * 1$. *OK, so we need $c \geq 1$.*

Inductive step: Suppose the claim holds for $1 \leq n \leq k-1$. We'll show that it also holds for $n = k$.

$$T(k) \leq T(k/5) + T(7k/10) + k$$

$$\leq ck/5 + 7ck/10 + k$$

$$\leq 9ck/10 + k$$ *Hm, what if we make $c = 10$?*

$$\leq 9k + k = 10k$$ *Yay!*

We used the Substitution Method!

- Guess the form of the answer to the recurrence. (e.g. by plotting it)

- Write an induction proof, figuring out the value of *c* as you go along.

End of Commercial Break!

$$T(n) \leq T(n/5) + T(7n/10) + n, \; n > 10$$

$$T(n) = 1, \; 1 \leq n \leq 10$$

Solution: $T(n) = O(n)$

Now back to median-finding!

....OK, let's put on our ADVENTURE HATS and try to do it without sorting!

Where's the room for improvement? Intuitive idea:

- If we sort, we spend a lot of time finding the *exact* relationships among values that are not even close to the median.

- What a waste! Can we not?

Medians of medians

What if we break the data into chunks, and ask each chunk what *its* median is?

Then the only sensible thing to do with those values would be to take *their* median.

Is it guaranteed that that value would be the overall median? (Can you argue this or find a counterexample?)

A counterexample

list [2, 9, 1, 3, 8, 4, 7, 6, 5], true median 5

Break into chunks of 3: [2, 9, 1], [3, 8, 4], [7, 6, 5]

Take their medians: 2, 4, 6

Take the median of those: 4. Oh no!

But we did get pretty close. Does that help?

What can we do with our estimate?

recall: list of unique elements, odd length

- Check every value in the list to see if it's greater or less than that estimate m.

- If less than half the other values are smaller than m, the true median is greater than m.

- If exactly half the other values are smaller than m, the true median is m and we are done.

- If more than half the other values are smaller than m, the true median is smaller than m.

A job for divide and conquer!

Repeat the following:

- Use the median of medians method to get an estimate m.
- Check m against all the other elements.
- If the median is smaller than m, throw out m and everything bigger. Then recurse.
    - Same idea if the median is bigger than m.
- The closer m is to correct, the more we get to throw away.

Wait a minute

- How do we recurse on a smaller list when the median we're searching for is no longer the median of that list?

- The method we described actually still works fine for finding any k-th largest element of a list! We know which half of the list our value is supposed to be in, so we estimate the median and then throw away the part of the list we know doesn't contain our value.

An example

We have a list of 101 elements. We want to find the median (51st smallest).

- Say our first estimate turns out to be the 58th largest value. Then we know the true median is one of the 57 values that (we now know) are less than that. So we save only those values.

An example

We have a list of 101 elements. We want to find the median (51st smallest).

- Say our first estimate turns out to be the 58th largest value. Then we know the true median is one of the 57 values that (we now know) are less than that. So we save only those values.

New problem! We have a list of 57 values. We want to find the 51st smallest.

- We estimate the median of this new list. It turns out to be the 23rd smallest value. We know the true median is one of the 57-23 = 34 values that (we now know) are greater than that. So we save only those values.

An example

We have a list of 101 elements. We want to find the median (51st smallest).

- Say our first estimate turns out to be the 58th largest value. Then we know the true median is one of the 57 values that (we now know) are less than that. So we save only those values.

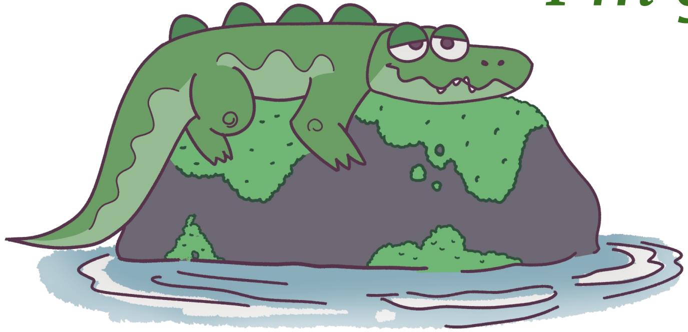
New problem! We have a list of 57 values. We want to find the 51st smallest.

- We estimate the median of this new list. It turns out to be the 23rd smallest value. We know the true median is one of the 57-23 = 34 values that (we now know) are greater than that. So we save only those values.

Another new problem! We have a list of 34 values. We want to find the (51-23) = 28th smallest...

Reminiscent of binary search!

This seems like it works!
I'm going back to sleep.



But what if our estimates are bad?

How close does median-of-medians get us?

Each of you please:

- Write down the numbers 1 through 15 in some order.
  Try to be random, or don't – it doesn't really matter.

- Break them into groups: first five, next five, last five.

- Within each group, find the median element.

- Take the median of those three medians.

We all got between 6 and 10

The true median is 8. So none of us was too far away!

In fact, there is no way anyone could have gotten a value below 6 or above 10…

Here we're using five groups of 5 (each column is a group)

a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	d_2	e_2
a_3	b_3	c_3	d_3	e_3
a_4	b_4	c_4	d_4	e_4
a_5	b_5	c_5	d_5	e_5

Suppose:

- Each column is sorted from top to bottom.
- The columns' medians a_3, b_3, c_3, d_3, e_3 are sorted from left to right.

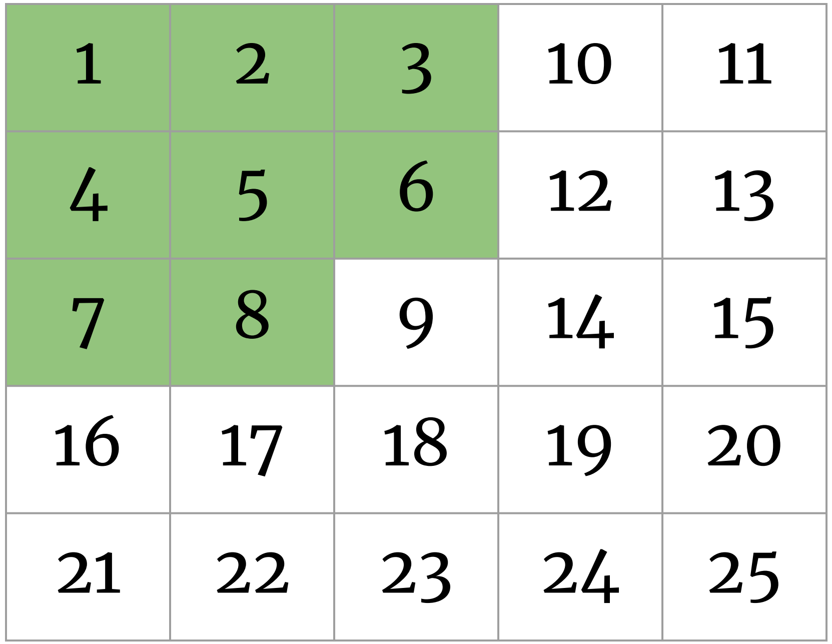
How many values are **definitely** less than c_3?

a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	d_2	e_2
a_3	b_3	c_3	d_3	e_3
a_4	b_4	c_4	d_4	e_4
a_5	b_5	c_5	d_5	e_5

Suppose:

- Each column is sorted from top to bottom.
- The columns' medians a_3, b_3, c_3, d_3, e_3 are sorted from left to right.

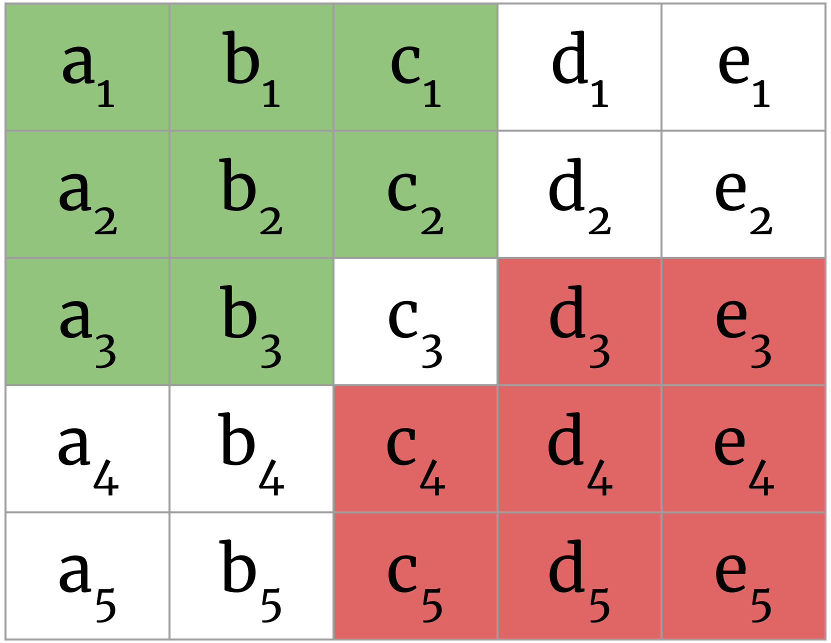
How many values are **definitely** less than c_3?



One example of the worst-case scenario!

🟩 a_1	🟩 b_1	🟩 c_1	d_1	e_1
🟩 a_2	🟩 b_2	🟩 c_2	d_2	e_2
🟩 a_3	🟩 b_3	c_3	🟥 d_3	🟥 e_3
a_4	b_4	🟥 c_4	🟥 d_4	🟥 e_4
a_5	b_5	🟥 c_5	🟥 d_5	🟥 e_5

The same logic holds for values definitely *greater* than c_3.



a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	d_2	e_2
a_3	b_3	c_3	d_3	e_3
a_4	b_4	c_4	d_4	e_4
a_5	b_5	c_5	d_5	e_5

There are $n/5$ groups of 5.

We get 3 from $((n/5)-1)/2$ of them, and 2 from the middle.

That's $(3n/10)-3/2 + 2$

i.e. just over $3n/10$.

So what?

- This means that when we throw away the part of the list we don't need, we can surely throw away just over $3n/10$ of the values, without worrying that we threw away the right one!

- And we're left with no more than $7n/10$ of the values...

Wait, where did this 5 thing come from?

- When an algorithm has some magic number in it, it's often for a very good reason.
- Why not 3? You'll investigate this on HW2!

A recurrence for this method

- Finding the median of each group of 5 takes constant time. There are $n/5$ such groups. Then we look through all n elements to see whether each is bigger than the estimate.
  So we do $O(n)$ work. *We'll handwave this as n but it could be 6n/5*

- We can **recurse** on a subproblem of size $n/5$ to find the median of medians.

- Then we **recurse** on a subproblem of size no more than $7n/10$.

$$T(n) \le T(n/5) + T(7n/10) + n$$

...which we conveniently already know is O(n)

This is a linear-time algorithm!!!

- Not only that, it lets us find any k-th smallest element, not just the median.
  - The median-of-medians part is used to divide up the current list into two parts, so that we can throw away the part we don't need, and it has a good chunk of elements (over 70%).
  - But, that said, we can search for any element we want, since we always know which of the two lists to look in.

```
• Select(A,k):
    • If len(A) <= 50:
        • MergeSort(A)
        • Return A[k-1]
    • p = MedianOfMedians(A)
    • L, R = Partition(A, p)
    • if len(L) == k-1:
        • return p
    • Else if len(L) > k-1:
        • return Select(L, k)
    • Else if len(L) < k-1:
        • return Select(R, k – len(L) – 1)
```

Base Case: If the len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k-th smallest value!

Case 2: The k-th smallest value is in the first part of the list

Case 3: The k-th smallest value is in the second part of the list

Subtleties

- **MedianOfMedians** itself needs to find a median. So it recursively calls Select! (but on a smaller list)

- **Partition** runs in O(n) time and just shoves values to the left or right of p. *Those values are not sorted relative to each other.*
    - so, sort of like MergeSort, but not quite

More subtleties

- What if we want the median of a list of **even size**?
  - Select the two middle values and average them!

- What if there are **repeated elements**?
  - This turns out not to hurt. e.g., say there are five 7s... think of them as 7.01, 7.02, 7.03, 7.04, 7.05.

- What if some sublist i**sn't a multiple of 5**?
  - This one is a more serious annoyance. We can notionally pad the final group of 5 with ∞ values, but then this eats into our guarantee (a bit) since some of the values we get to throw away are fake. But it all still holds.

6/27 Lecture Agenda

- Announcements
- Part 2-1: k-Selection
- 10 minute break!
- Part 2-2: RadixSort and the limits of sorting

6/27 Lecture Agenda

- Announcements
- Part 2-1: k-Selection
- 10 minute break!
- Part 2-2: RadixSort and the limits of sorting

WORLD 2-2

(Breaking) The Limits of Sorting

Divide and Conquer

Sorting & Randomization

Data Structures

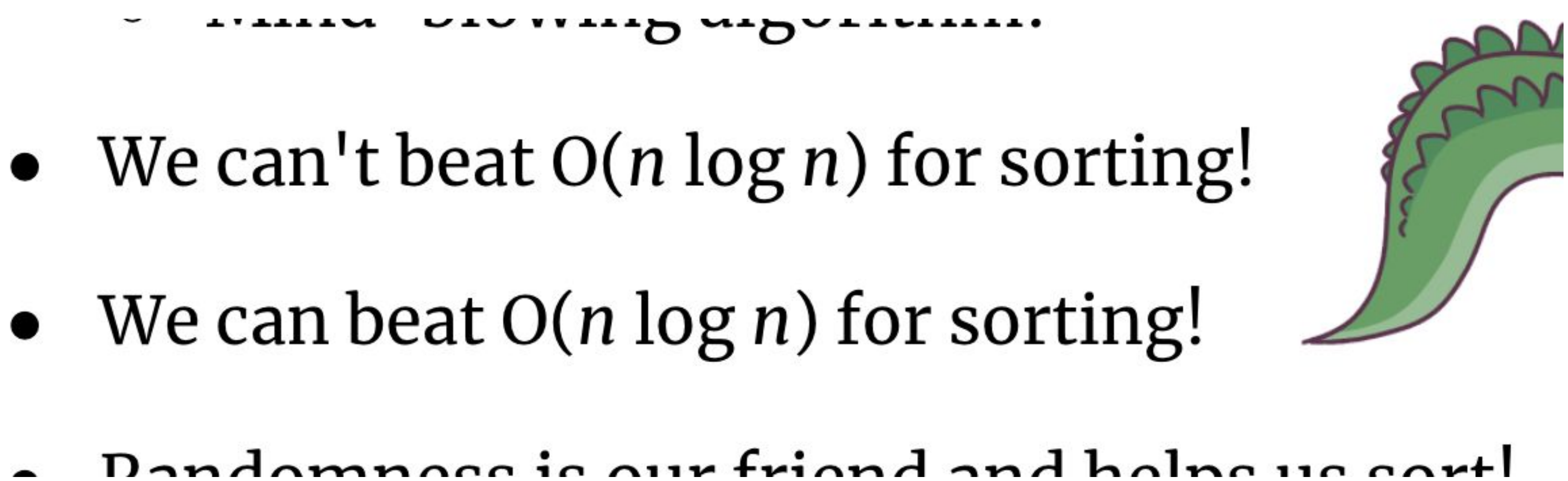
Graph Search

Dynamic Programming

Greed & Flow

Special Topics

A confusing claim last week...



- Mind-blowing algorithm!
- We can't beat O(n log n) for sorting!
- We can beat O(n log n) for sorting!
- Randomness is our friend and helps us sort!

In what sense can we not beat O(n log n)?

- In this part, we will only consider sorts based on **comparisons** of two elements at a time.
- When would this be relevant? Suppose we're sorting something that we can't quantify absolutely. e.g., rocks from less pretty to prettiest.
    - assuming transitivity (i.e., there is no cycle in which $A > B$, $B > C$, $C > A$)

How many outcomes of sorting?

- Say we have a sorting algorithm that takes in lists of length n in which all elements are distinct.

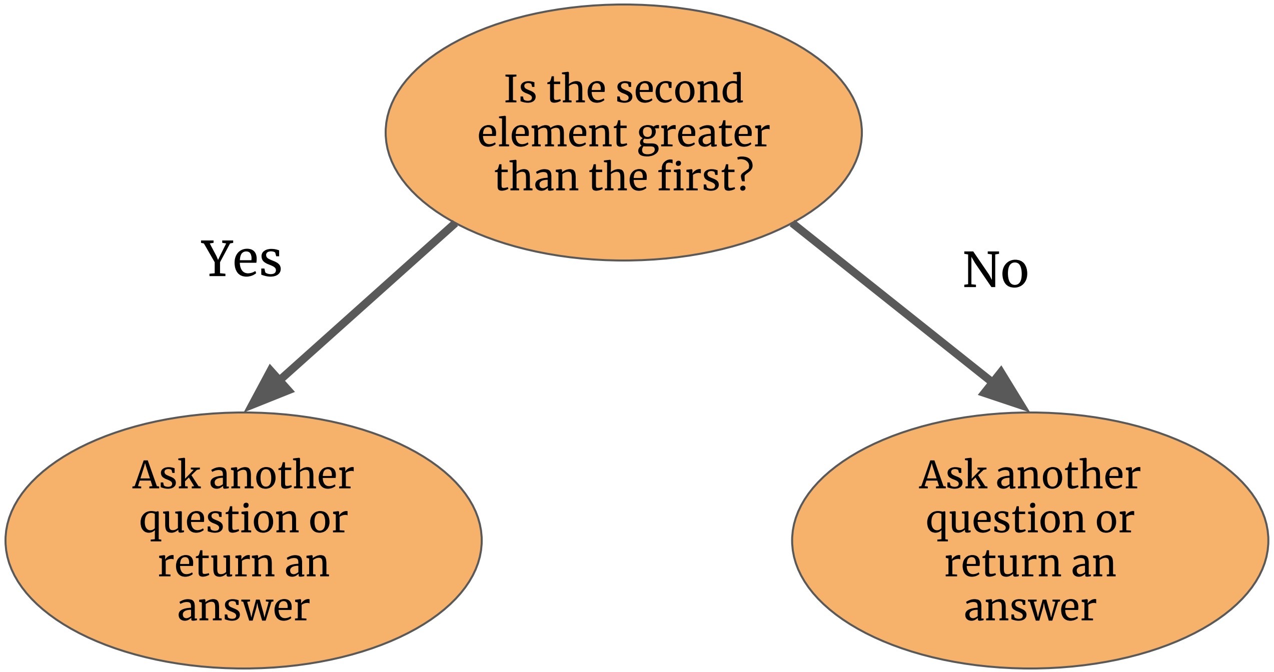
- How many possible outcomes could this return?

- There are n choices for the first element, n-1 choices for the second element, and so on… so this is $n!$ (n factorial).

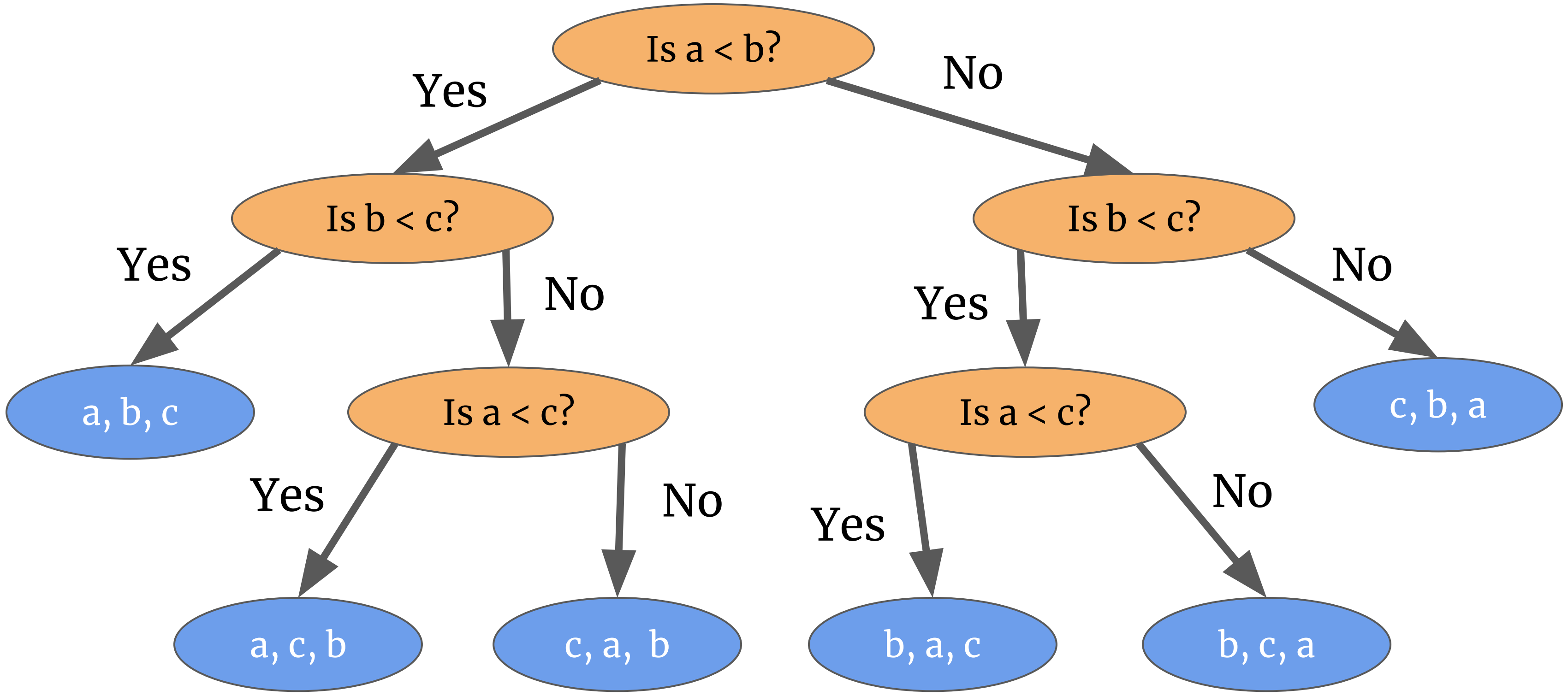
It's n!

NOT SURE IF THE COMBINATORIALIST IS EXCITED
OR IT'S JUST A FACTORIAL

A comparison-based method is a tree


Is the second element greater than the first?
Yes
No
Ask another question or return an answer
Ask another question or return an answer

Sorting a list of distinct elements [a, b, c]



How deep is the tree?

- A shallower tree means an asymptotically better algorithm!
  - Remember, we judge algorithms by how they perform on their *worst-case* inputs.

- This is a binary tree. How shallow can it possibly be, if we fill up every level as much as possible?

- We have $n!$ leaf nodes, and therefore we necessarily have exactly $n!$ - 1 internal nodes. Let's stuff 2($n!$) - 1 nodes into a tree...
  - This is a **best case**. A worse algorithm might have the same leaf node twice.

How deep is the tree?

- We can get:
  - 1 node on the first level
  - 2 nodes on the second level
  - 4 nodes on the third level...
- With n levels, we can fit at most $2^n - 1$ nodes.
- Number of levels needed to fit k nodes = floor($\log_2 k$) + 1
- To get $2(n!) - 1$ nodes into a tree... floor($\log_2(2(n!)-1)$) + 1

$\log_2$ of n!

We will handwave away $2n! - 1$ as being bounded below by $n!$.

$\log_2(n!) = \log_2(n) + \log_2(n-1) + \ldots + \log_2(2) + \log_2(1)$

- A powerful trick! Notice that each of the terms from $\log_2(n)$ to $\log_2((n/2) + 1)$ is greater than $\log_2(n/2)$, which is $\log_2 n - \log_2 2 = \log_2 n - 1$
- There are $n/2$ such terms, and so their sum is greater than $(n/2)(\log_2 n - 1)$.
- Oh no! This is $\Omega(n \log n)$.

No comparison-based sorting algorithm can be o(n log n)

This is sad! But negative results like this are useful.

The value of lower bounds

"Hey I have a binary search tree that lets you do any insertion in O(1) time and can be traversed in O(n) time"

- No you don't!

- If you did, we could do a comparison-based sort of a list in O(n) time as follows:

    - Insert all elements into the tree: n * O(1) = O(n)
    - Traverse the tree to get the sorted order: O(n)
    - Total running time: O(n)

But what if we use a sort that **isn't** limited to binary comparisons?

A real-life algorithmic triumph

- You are a grad student at UC Berkeley. (I know – boo, hiss!) You just proctored a biology exam.

- You have a pile of exams to sort by student ID.
    - The IDs are all 8 digits long. *(Here we'll pretend it's 5)*

- There are too many to lay out all around the room to do a MergeSort or something. The grad student office is small.

- You want to get this over with fast so you can go to Naan n' Curry and walk home and continue avoiding your graduate research.

94305
94301
84305
73301
94315
93301
94401
54412

Here's the initial list of IDs, in some arbitrary order.

```
94305
94301
84305
73301
94315
93301
94401
54412
```

Here's the initial list of IDs, in some arbitrary order.

What if we take advantage of the fact that there are only 10 digits possible? Then we only need 10 piles.

We could sort them by first digit and then sort each pile recursively. But notice that the distribution of digits is uneven – most start with 9, etc.

```
94305
94301
84305
73301
94315
93301
94401
54412
```

Awesome idea:

Sort by the last digit first.

Then sort by the next-to-last digit, preserving the order from the last round when there are ties.

And so on!

```
94305
94301
84305
73301
94315
93301
94401
54412
```

94305	94301
94301	73301
84305	93301
73301	94401
94315	54412
93301	94305
94401	84305
54412	94315

94305	94301
94301	73301
84305	93301
73301	94401
94315	54412
93301	94305
94401	84305
54412	94315

94305	94301	94301
94301	73301	73301
84305	93301	93301
73301	94401	94401
94315	54412	94305
93301	94305	84305
94401	84305	54412
54412	94315	94315

94305	94301	94301	94301
94301	73301	73301	73301
84305	93301	93301	93301
73301	94401	94401	94305
94315	54412	94305	84305
93301	94305	84305	94315
94401	84305	54412	94401
54412	94315	94315	54412

94305	94301	94301	94301	73301
94301	73301	73301	73301	93301
84305	93301	93301	93301	94301
73301	94401	94401	94305	94305
94315	54412	94305	84305	84305
93301	94305	84305	94315	94315
94401	84305	54412	94401	94401
54412	94315	94315	54412	54412

94305	94301	94301	94301	73301	54412
94301	73301	73301	73301	93301	73301
84305	93301	93301	93301	94301	84305
73301	94401	94401	94305	94305	93301
94315	54412	94305	84305	84305	94301
93301	94305	84305	94315	94315	94305
94401	84305	54412	94401	94401	94315
54412	94315	94315	54412	54412	94401

sorted!

RadixSort!

- Start with a list L of n items, each with d "digits", where there are b possible values (0 through b–1) for each "digit".
- Make b empty "buckets" (lists), numbered 0 through b–1.

- For i in [d, d–1, ... 1]:
    - Go through the list L, looking at the i–th digit of each element, and placing the element at the rightmost end of the bucket matching that digit.
    - (Now L is empty.)
    - For j in [0, 1, ..., b–1]:
        - Add the elements in bucket j, in order, to the rightmost end of L.

This ensures that the sort is stable.

RadixSort!

- Start with a list L of n items, each with d "digits", where there are b possible values (0 through b-1) for each "digit".
- Make b empty "buckets" (lists), numbered 0 through b-1. **$O(b)$**

$O(d)$

- For i in [d, d-1, ... 1]:
  - Go through the list L, looking at the i-th digit of each element, and placing the element at the rightmost end of the bucket matching that digit. **$O(n)$ * $O(1)$ – linked list?**
  - (Now L is empty.)
  - For j in [0, 1, ..., b-1]: **$O(b)$*$O(n)$?**
    - Add the elements in bucket j, in order, to the rightmost end of L.

RadixSort!

- Start with a list *L* of *n* items, each with *d* "digits", where there are *b* possible values (0 through *b*−1) for each "digit".
- Make *b* empty "buckets" (lists), numbered 0 through *b*−1. **O(*b*)**
- For *i* in [*d*, *d*−1, ... 1]: **O(*d*)**
    - Go through the list *L*, looking at the *i*-th digit of each element, and placing the element at the rightmost end of the bucket matching that digit. **O(*n*) * O(1)**
    - (Now *L* is empty.)
    - For *j* in [0, 1, ..., *b*−1]: ~~O(*b*) * O(*n*)?~~ **O(*b*) + O(*n*)**
        - Add the elements in bucket *j*, in order, to the rightmost end of *L*.

There are still only n elements total, though, however they're distributed.

RadixSort!

- Start with a list *L* of *n* items, each with *d* "digits", where there are *b* possible values (0 through *b*−1) for each "digit".
- Make *b* empty "buckets" (lists), numbered 0 through *b*−1. $O(b)$

- For *i* in [*d*, *d*−1, ... 1]: $O(d)$
  - Go through the list *L*, looking at the *i*-th digit of each element, and placing the element at the rightmost end of the bucket matching that digit. $O(n) * O(1)$
  - (Now *L* is empty.)
  - For *j* in [0, 1, ..., *b*−1]: $O(b) + O(n)$
    - Add the elements in bucket *j*, in order, to the rightmost end of *L*.

$O(b) + O(d*(n+b+n)) = O(d(n+b))$

- When b is assumed to be small relative to n, $O(d(n+b))$ reduces to $O(dn)$, which is how you'll usually see the running time given.

- Example: 64-bit integers
  - $d = 64$
  - $b = 2$ (each bit can be 0 or 1)
  - Overall running time on a list of size n:

    $O(64(n+2)) = O(n)$

But but didn't we just say we can't beat n log n

- Radix sort is **not a comparison-based sort**. Notice that values are never directly compared with each other!

- It relies on there being a finite (and small) number of possible values *b* for each digit.

Some details

- When $d = 1$, this is sometimes called **counting sort**.
  - Got a list of values that are all in the range 1 through 100? Why even sort them? Just count the number of each type!

- Works on, e.g., words as well! (If using, say, lowercase English letters, words are just base-26 numbers.)

- Works even if different values have different lengths: just front-pad with 0s.

- Works even if different "digits" have different bases.

- Handles duplicate values just fine.

Rough correctness argument

- Consider two list elements a and b, with $a < b$.
- Starting from the left (the most significant digit), find the first digit j at which a and b differ.
- When radix sort got around to sorting digit j, it placed a earlier in the list than b. Then it maintained that relative order in all remaining rounds, since a and b were always tied for the same digit thereafter.

Why not just use this instead of MergeSort?

- Constant factors! O(dn) is not really O(n).
  - Also, if all values are distinct, then n can be no larger than b^d, which means $d \geq \log_b n$.
    So O(dn) is O($n \log n$). (Big) oh no!

- Sometimes we may want to sort values that can't be (easily) mapped to base-b numbers.

- The bucket overhead takes extra space, and it's hard to plan for how it will be used (depends on which buckets end up getting the most elements).

Radix is Latin for "root", as in "numerical base"

These radishes are roots. But they cannot be (easily) sorted via RadixSort.

Next time

- Harnessing the power of randomness!
- We'll see the idea of pivoting / partitioning again as we study QuickSort!
- The next lecture is the most probability-intensive. A review document on probability is coming.