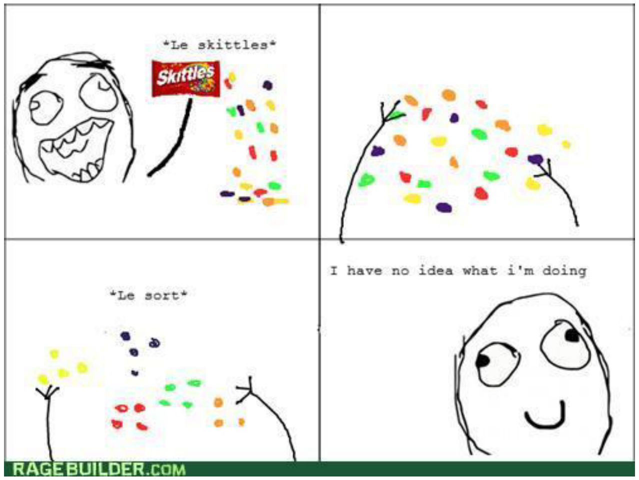
Adam Blank — Lecture 12/13 — Summer 2015

CSE 332

Data Abstractions

Sorting



Where We Are

Simple: $\mathcal{O}(n^2)$	**Fancy:** $\mathcal{O}(n \lg n)$	**Specialized:** $\mathcal{O}(n)$
Insertion Sort	Heap Sort	Counting Sort
Selection Sort	Merge Sort	Radix Sort
...	...	...

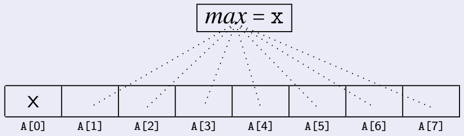
We've discussed some sorting methods

They all happened to be $\Omega(n \lg n)$. Can we do better?

Bounding The MAXIMUM Problem

Upper Bound

```
int findMax(int[] arr) {
   int max = arr[0];
   for (i = 0; i < arr.length; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }
   }
   return max;
}
```



This algorithm takes **at most** $n-1$ comparisons. So, $n-1$ is an **upper bound** for the **MAXIMUM** problem.

Bounding The MAXIMUM Problem

Lower Bounds are much more difficult to prove. We must show that **any** algorithm that solves the problem has to do something.

Lower Bound (Proof #1)

Consider an algorithm that solves the **MAXIMUM** problem in **fewer** than $n-1$ comparisons.
Since the algorithm uses fewer than $n-1$ comparisons, there must be some element of the input that wasn't compared to anything (say it's x):

a_0	a_1	a_2	x	a_3	a_4	a_5	a_6
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Consider two distinct values for x:

- $x = \min(a_0, a_1, a_2, \ldots, a_n) - 1$
- $x = \max(a_0, a_1, a_2, \ldots, a_n) + 1$

Notice that, to be correct, the algorithm must output **different** answers based on the value of x.

But it never examines x! So, it must always output the same thing on otherwise identical arrays.

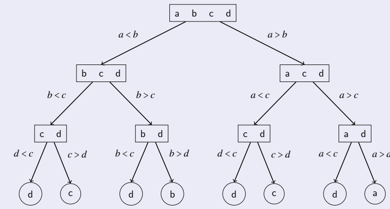
Bounding The MAXIMUM Problem

Key Ideas

- Must be able to output any valid answer (every index is the max for **some** input)
- The only computations that give information about the correct answer are the **comparisons**
- Must only have **one** valid possibility remaining before answering

Decision Tree

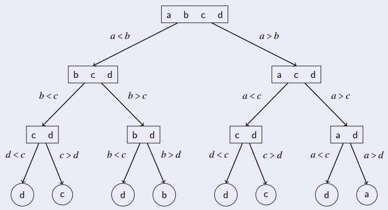
Consider the comparisons some (arbitrary) algorithm makes:



This is a **decision tree**. The nodes have **the remaining valid possibilities**. The edges represent **making a comparison**.

Bounding The MAXIMUM Problem

Lower Bound (Proof #2)



- Every valid output (element of the array) must be a leaf
- Some decision tree **completely** represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

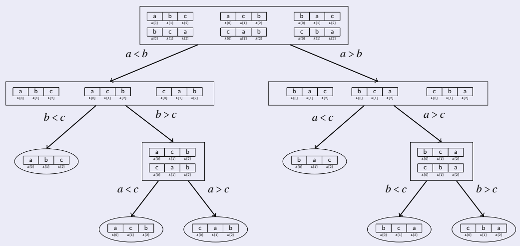
We are interested in the **worst case # of comparisons**. So, we want to know **how long** the **longest path** is (e.g. what is the height of the tree).

A single comparison can rule out (at most) one output.

We begin with n possibilities and each comparison rules out **at most one**. So, the **minimum length** of the **longest path** is $n-1$.

The Main Event!

Lower Bound for Sorting



- Every valid output (??????) must be a leaf
- Some decision tree **completely** represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

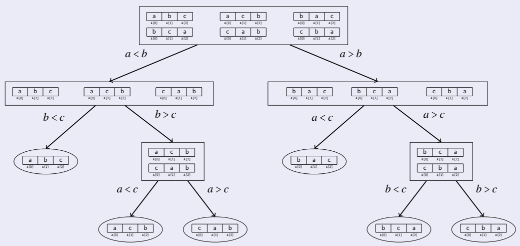
We are interested in the **worst case # of comparisons**. So, we want to know **how long** the **longest path** is (e.g. what is the height of the tree).

A single comparison can rule out (at most) **???????** output.

We begin with **????** possibilities and each comparison rules out **??????**. So, the **minimum length** of the **longest path** is **????**

The Main Event!

Filling In The Blanks



- What are the outputs?

  The outputs are permutations of the input:
  abc, acb, bac, bca, cab, cba

- How many of them are there?

  There are $n!$ permutations of n items:

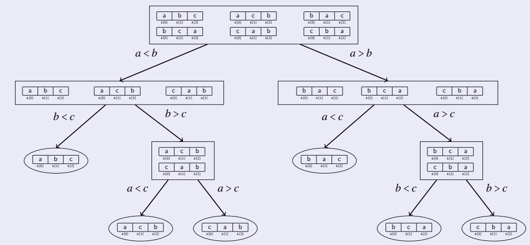
  $\underline{\quad}$ n choices $\underline{\quad}$ $n-1$ choices $\underline{\quad}$ $n-2$ choices $\ldots$ $\underline{\quad}$ 1 choice

- How many outputs does each comparison rule out (minimum)?

  Every output either goes into the left or the right side.
  So, at least one side has **half** of the elements.

The Main Event!

Lower Bound for Sorting



- Every valid output (**permutations of** `A`) must be a leaf
- Some decision tree completely represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

We are interested in the worst case # of comparisons (height of the tree).

A single comparison can rule out (at most) **half** of the outputs.

We begin with $n!$ possibilities and each comparison rules out at most **half of the remaining ones**. So, the minimum length of the longest path is: $\lg(n!)$.

The Main Event!

(Asymptotic) Lower Bound for Sorting

We've now shown that the comparison sorting problem is $\Omega(\lg(n!))$. It turns out that this is actually $\Omega(n\lg(n))$:

$$\begin{aligned}
\lg(n!) &= \lg(n(n-1)(n-2)\ldots 1) && \text{[Def. of } n!\text{]}\\
&= \lg(n) + \lg(n-1) + \ldots \lg\left(\frac{n}{2}\right) + \lg\left(\frac{n}{2}-1\right) + \ldots \lg(1) && \text{[Prop. of Logs]}\\
&\geq \lg(n) + \lg(n-1) + \ldots + \lg\left(\frac{n}{2}\right)\\
&\geq \left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right)\\
&= \left(\frac{n}{2}\right)(\lg n - \lg 2)\\
&= \frac{n\lg n}{2} - \frac{n}{2}\\
&\in \Omega(n\lg(n))
\end{aligned}$$

It follows that $\Omega(n\lg(n))$ is a lower bound for the sorting problem!

Spectrum of Sorting

Simple: $\mathcal{O}(n^2)$	**Fancy:** $\mathcal{O}(n\lg n)$	**Specialized:** $\mathcal{O}(n)$
Insertion Sort	Heap Sort	Counting Sort
Selection Sort	Merge Sort	Radix Sort
...	...	...

There are a lot of comparison based sorts, but they can't break the lower bound of $\Omega(n\lg n)$

But what about algorithm that **don't use comparisons**!

Bounded Set Returns!

Remember the assumption we made for the `BoundedSet` ADT?

`BoundedSet` ADT

`Data`	**Set** of **numerical** keys where $0 \le k \le B$ for some $B \in \mathbb{N}$
`insert(`**key**`)`	Adds **key** to set
`find(`**key**`)`	Returns `true` if **key** is in the set and false otherwise
`delete(`**key**`)`	Deletes **key** from the set

The only difference between `Set` and `BoundedSet` is that `BoundedSet` comes with an upper bound of B.

Suppose we have integers between 1 and B (just like `BoundedSet`). How could we go about sorting them?

Counting Sort

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Counting Sort

Counting Sort

Assuming all data is **ints** between 1 and B:

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Example

Input: $5\ 1\ 3\ 3\ 2\ 1\ 3\ 4\ 5\ 1\ 1$ $(B = 5)$

- Initialize the array:

A[0]	A[1]	A[2]	A[3]	A[4]

- Loop through the elements:

4	1	3	1	2
A[0]	A[1]	A[2]	A[3]	A[4]

- Loop through the indices

Output: $1\ 1\ 1\ 1\ 2\ 3\ 3\ 3\ 4\ 5\ 5$

Counting Sort Analysis

Counting Sort

Assuming all data is **ints** between 1 and B:

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Analysis

- Best Case?

$$\mathcal{O}(n+B)$$

- Worst Case?

$$\mathcal{O}(n+B)$$

- Why doesn't the sorting lower bound apply?

  It's not a comparison sort! We actually didn't use comparisons at all!
- When should we use Counting Sort?

  We should use Counting Sort when $n \approx B$.

Radix Sort

Radix Sort

- Choose a "number" representation (e.g. $(100)_{10} = (1100100)_2 = (d)_{128}$)
- For each digit from least significant to most significant, do a **stable sort** (why stable?)

Usually for the sorting step, we use **counting sort**.

Example

	Sort Yellow →		Sort Yellow →		Sort Yellow →	
478		721		003		003
537		003		009		009
009		143		721		038
721		537		537		067
003		067		038		143
038		478		143		478
143		038		067		537
067		009		478		721

Counting Sort: Analysis

Radix Sort

- Choose a “number” representation (e.g. $(100)_10 = (1100100)_2 = (d)_{128}$). Say base B.
- For each digit from least significant to most significant, do a **stable COUNTING sort**. Say there are P passes.

Analysis

- Best Case?
$$\mathcal{O}(P(B+n))$$
- Worst Case?
$$\mathcal{O}(P(B+n))$$
- Should we use radix sort?
  Consider Strings of English letters up to length 15:
  - Radix Sort will take 15(52 + n)
  - For $n < 33,000$, $n \lg n$ wins.

Applications and Related Problems

Possibly the most useful application of sorting is as a form of **pre-processing**. We sort the input in $\mathcal{O}(n \lg n)$ and then solve the actual problem using the sorted data. (e.g. if we expect to do more than $\mathcal{O}(n)$ finds, the sorting step is worth it)

Big CS Idea!

To make a repeated operation easier, do an expensive **pre-processing step** once. You saw this with DFAs and String Matching in CSE 311 as well!

The **median** problem has already come up. Let’s explore it more!

What is SELECT?

SELECT is the computational problem with the following requirements:

Inputs

- An array `A` of `E` data of length L and a number $0 \leq k < L$.
- A consistent, total ordering on all elements of type `E`:
  `compare(a, b)`

Post-Conditions

- The array remains unchanged.
- Let `B` be the ordering that **SORT** would return. We return `B`(k).

An Algorithm to Solve SELECT

Solving SELECT(k)

- Copy `A` into `B`
- Sort `B`
- Return (`B`(k))

Awesome, except this is $\mathcal{O}(n \lg n)$

Another idea, instead of “sorting”, only sort the parts we need.

QuickSort: A Reminder

- Choose a pivot in `A`: p
- Partition `A` into two arrays: `SMALLER` and `LARGER`
- QuickSort `SMALLER`.
- QuickSort `LARGER`.
- `SMALLER` + $[p]$ + `LARGER` is a sorted array.

Idea: To find the k-th element, do we need to recurse on both sides?

An Algorithm to Solve SELECT

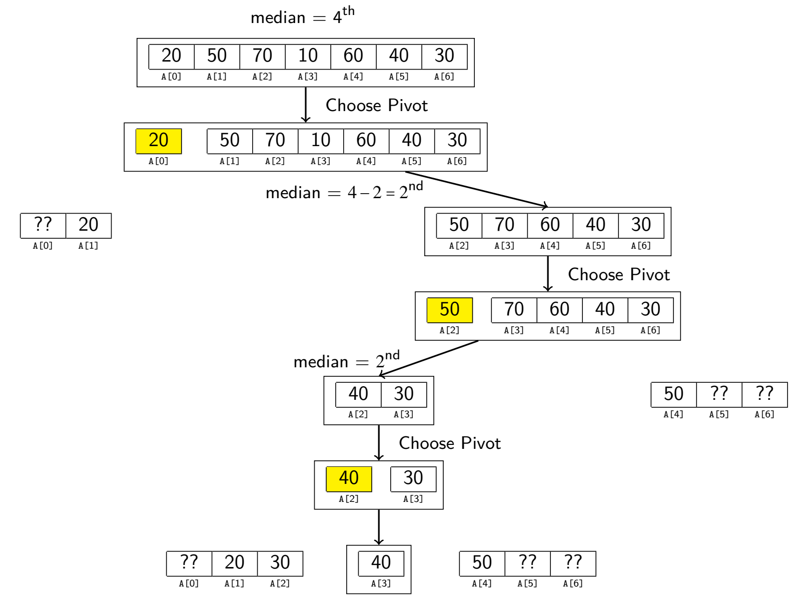
QuickSelect(A, k)

- Choose a pivot in `A`: p
- Partition `A` into two arrays: `SMALLER` and `LARGER`
- Since we know how big `SMALLER` and `LARGER` are, we know the final index of p. Call this `x`.
- If $k = \mathtt{x}$, return p.
- If $k < \mathtt{x}$, return QuickSelect(`SMALLER`, k)
- If $k > \mathtt{x}$, return QuickSelect(`LARGER`, $k - \mathtt{x}$)

Analysis

- Best Case: $T(n) = T(n/2) + cn$ (So, $\mathcal{O}(n)$)
- Worst Case: $T(n) = T(n-1) + cn$ (So, $\mathcal{O}(n^2)$)
- (Average Case is $\mathcal{O}(n)$)

QuickSelect



Deterministic QuickSelect (Median-of-Medians)

Median-of-Medians

- Split `A` into $g = n/5$ groups of 5 elements.
- Sort each group and find the medians: $m_1, m_2, \ldots, m_{n/5}$
- Find p: the median of the medians (recursively...)
- Separate the input into two groups `SMALLER` and `LARGER` and recurse on the appropriate piece

This algorithm is "basically" **QuickSelect**, but with a special pivot.

Analysis

The key to this algorithm is that whichever side we recurse on is at least 3/10 of the input. Here's why:

- Consider `SMALLER`. We know that at least $g/2$ of the groups have a median $\geq p$. Of the 5 elements in each of these groups, since the median is $\geq p$, 3 of them are $\geq p$ (possibly including the median). Putting this together, we have $3(g/2) = 3((n/5)/2) = 3n/10$ elements $\geq p$. This means we **know** we will discard at least this many. So, the maximum number of elements we could recurse on is $7n/10$.
- The other case is symmetric.

Deterministic QuickSelect (Solving the Recurrence)

Median-of-Medians

- Split `A` into $g = n/5$ groups of 5 elements.
- Sort each group and find the medians: $m_1, m_2, \ldots, m_{n/5}$
- Find p: the median of the medians (we're gonna do this recursively...)
- Separate the input into two groups `SMALLER` and `LARGER` and recurse on the appropriate piece

Solving The Recurrence

So, putting all this together gives us the recurrence

$$
\begin{aligned}
T(n) &\leq \mathcal{O}(5\lg 5)\left(\frac{n}{5}\right) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \\
&= cn \quad + T\left(\frac{2n}{10}\right) + T\left(\frac{7n}{10}\right) \\
&= cn \quad + \left(\frac{2n}{10} + T\left(2\left(\frac{2n}{10}\right)\right) + T\left(7\left(\frac{2n}{10}\right)\right)\right) \\
&\qquad + \left(\frac{7n}{10} + T\left(2\left(\frac{7n}{10}\right)\right) + T\left(7\left(\frac{7n}{10}\right)\right)\right) \\
&= cn \quad + \frac{9n}{10} + T\left(\frac{2^2 n}{10^2}\right) + 2T\left(7 \times 2 \times \left(\frac{n}{10^2}\right)\right) + T\left(\frac{7^2 n}{10^2}\right)
\end{aligned}
$$

Solving The Recurrence

So, putting all this together gives us the recurrence

$$
\begin{aligned}
T(n) \le \mathcal{O}(5 \lg 5)\left(\frac{n}{5}\right) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \\
= cn \quad & + T\left(\frac{2n}{10}\right) + T\left(\frac{7n}{10}\right) \\
= cn \quad & + \left(\frac{2n}{10} + T\left(2\left(\frac{2n}{10}\right)\right) + T\left(7\left(\frac{2n}{10}\right)\right)\right) \\
& + \left(\frac{7n}{10} + T\left(2\left(\frac{7n}{10}\right)\right) + T\left(7\left(\frac{7n}{10}\right)\right)\right) \\
= cn \quad & + \frac{9n}{10} + T\left(\frac{2^2 n}{10^2}\right) + 2T\left(7 \times 2 \times \left(\frac{n}{10^2}\right)\right) + T\left(\frac{7^2 n}{10^2}\right) \\
\le cn \quad & + \frac{9n}{10} + \frac{2^2 + 2(7 \times 2) + 7^2}{10^2} + \ldots \\
= cn \quad & + \frac{9n}{10} + \frac{9^2 n}{10^2} + \ldots \\
= cn \quad & \left(\sum_{i=0}^{\infty} 9^i 10^i\right) = cn\left(\frac{1}{1 - 9/10}\right) = 10cn
\end{aligned}
$$

Whoo hoo!