LECTURE #3
So far we have applied divide and conquer to arithmetic problems:

- Karatsuba: faster integer multiplication \((n^2 \rightarrow n^{\log_2 3})\)
- Strassen: faster matrix multiplication \((n^3 \rightarrow n^{\log_2 7})\)

Today we apply divide and conquer to common tasks on lists:

1. Sorting
2. Finding the median
**Sorting**: given a list of numbers $a_1, \ldots, a_n$, output them in increasing order (or decreasing).

Idea is to:
1. split the list into two halves
2. recursively sort each half
3. merge the two sorted halves

$\text{MERGESORT}(a_1, \ldots, a_n) :=$
1. if $n=1$, return $a_1$
2. $S_L := \text{MERGESORT}(a_1, \ldots, a_{n/2})$
3. $S_R := \text{MERGESORT}(a_{n/2+1}, \ldots, a_n)$
4. $S = \text{merge}(S_L, S_R)$
5. return $S$

How to implement $\text{merge}$? Take smaller element from the two sorted lists and repeat.

Ex: \[
\begin{array}{cccccc}
3 & 7 & 10 & 13 & 15 \\
2 & 6 & 11 & 14 & 15 \\
\end{array}
\]

Hence $\text{merge}(S_L, S_R)$ runs in time $O(|S_L| + |S_R|)$.

(Each iteration compares two elements and removes one.)
The running time is $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$.

By the Master Theorem on Recurrences: $a=2, b=2, d=1 \Rightarrow \frac{a}{b^d} = \frac{2}{2^1} = 1 \Rightarrow O(n^d \log n) = O(n \log n)$.  

All the "real work" is in merging, as nothing happens till the recursion hits the base case. This naturally leads to an iterative algorithm that maintains a queue on lists:

There are $O(\log n)$ passes, each taking time $O(n)$, which double the size of the lists on the queue.
Q: can we do better than mergesort?  No and Yes

No: mergesort is a comparison sort, i.e., an algorithm in which the only operation performed on the input elements are comparisons (their values are otherwise ignored).

\[ \text{Theorem: Any comparison sorting algorithm requires } \Omega(n \log n) \text{ comparisons to sort lists of } n \text{ elements.} \]

So mergesort is optimal among comparison sorting algorithms.

Yes: there are sorting algorithms that are not solely based on comparisons.

For example, if the elements are \( w \) bits long, then:

- radix sort uses \( O(w \cdot n) \) bit operations
- merge sort uses \( O(w \cdot n \cdot \log n) \) bit operations \((a \text{ comparison costs } O(w) \text{ bit operations})\)

There are many sorting algorithms and the "best" one depends on the application. (Data resides in RAM vs disk, mergesort works better on linked lists,...)
Theorem: Any comparison sorting algorithm requires $\Omega(n \log n)$ comparisons to sort lists of $n$ elements.

Fix an algorithm $A$. Without loss of generality (WLOG), focus on input lists $a_1, \ldots, a_n$ where elements are distinct.

The computation of $A$ on $a_1, \ldots, a_n$ defines a permutation $\pi : [n] \rightarrow [n]$ (the output is $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$).

Every permutation is a possible output.

Let $S$ denote the set of possible permutations at a given point in $A$'s computation.

Before algorithm starts: $|S| = |\{\text{all possible permutations}\}| = n!$

At each comparison: if $a_i < a_j$ then $S \rightarrow S_1$,
if $a_i > a_j$ then $S \rightarrow S_2$

Since $S_1 \cup S_2 = S$, we know that $|S_1| \geq |S|/2$ or $|S_2| \geq |S|/2$.

So a comparison divides possible outputs by at most 2.

Hence, "# of comparisons until $|S|=1$"

$\geq \log_2(n!) \geq \log_2 \left( \frac{n}{e} \right)^n = n \log n - n \log e = \Omega(n \log n)$

Note: this is a worst-case lower bound (depth of deepest leaf is $\omega(n \log n)$), but can be improved to an average-case lower bound (average depth of leaf is $\Theta(n \log n)$).
Median Finding:

given \( S = \{a_1, \ldots, a_n\} \) output \( \text{median}(S) := \{a \in S \text{ s.t. half of } S \text{ is smaller & half of } S \text{ is bigger}\}

How is \( \text{median}(S) \) different from \( \text{average}(S) = (\Sigma_{i=1}^n a_i)/n \)?

Ex: 
- \((1,1,1) \rightarrow \text{avg} = 1 \quad \text{median} = 1\) median is one of the elements, 
- \((1,1,10) \rightarrow \text{avg} = 4 \quad \text{median} = 1\) and is less sensitive to outliers 
- \((1,1,100) \rightarrow \text{avg} = 34 \quad \text{median} = 1\)
- \((1,1,1000) \rightarrow \text{avg} = 334 \quad \text{median} = 1\)

How to compute median?

Idea 1: sort and take middle element — \( O(n \log n) \)

Idea 2: we do not care about the order of elements above and below the median. We use divide and conquer to solve a harder problem:

\[ \text{Selection} \] 

input: set of numbers \( S \), index \( k \in [n] \) 

output: \( k \)-th smallest element in \( S \)

Note: \( k = \frac{|S|}{2} = \frac{n}{2} \) is the median (some def's average two middles when \( S \) is even)
Idea: pick a.e.S and split S into
\[ S_L := \{ \text{elts in } S \text{ smaller than } a \} \]
\[ S_a := \{ \text{elts in } S \text{ equal to } a \} \]
\[ S_R := \{ \text{elts in } S \text{ greater than } a \} \]

Then reverse in a straightforward way.

Select \((S, k)\): pick a.e.S and compute \(S_L, S_a, S_R\) \(S\) can split in linear time

- if \(k \leq |S_L|\) then Select \((S_L, k)\)
- if \(|S_L| < k \leq |S_L| + |S_a|\) then return \(a\)
- if \(|S_L| + |S_a| < k\) then Select \((S_R, k - |S_L| - |S_a|)\)

We go from list size \(|S|\) to list size \(\max\{|S_L|, |S_R|\}\).

How to pick a?

Bad case is if a is always the largest (or smallest) element of the current set:
\[ O(n) + O(n-1) + O(n-2) + \ldots = O(n^2) \]

Good case is if a always splits \(S\) roughly in half: \(|S_L|, |S_R| \approx |S|/2\)
In this case we get the recurrence: \[ T(n) = T(n/2) + O(n) = O(n) \]

Problem: picking a.e.S as above requires... finding median!
Idea: pick \( a \in S \) at random!

We say that \( a \) is **good** if \( a \) is between 25th and 75th percentiles: \([\frac{25}{4}, \frac{75}{4}]\).

When \( a \) is good, the new set shrinks by a constant factor:

\[
\max \{ |S_L|, |S_R| \} \leq \frac{3}{4} |S|.
\]

There are many good elements: \( \Pr[a \text{ is good}] = \frac{1}{2} \).

So in expectation it takes 2 tries to get a good \( a \).

The expected running time is:

\[
\mathbb{E}T(n) \leq \mathbb{E}T\left(\frac{3}{4}n\right) + \mathbb{E}\left[ \text{time to find good } a \right] \cdot O(n) + O(n)
\]

\[
= \mathbb{E}T\left(\frac{3}{4}n\right) + 2 \cdot O(n)
\]

\[
= \mathbb{E}T\left(\frac{3}{4}n\right) + O(n)
\]

\[
= O(n)
\]

On any input \( S \) and integer \( k \), \( \text{Select}(S,k) \) returns the correct answer in a number of steps that is \( O(n) \) in expectation.