Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say \( k \) arrays of size \( \frac{n}{k} \) each?

Recall:

\[
T(n) = 2T\left(\frac{n}{2}\right) + cn \Rightarrow T(n) = O(n \log_2 n)
\]

\[
T(n) = 3T\left(\frac{n}{3}\right) + cn
\]

\[
\log_3 n
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]

\[
\frac{n}{9}
\]
In general: \[ T(n) = k T\left(\frac{n}{k}\right) + cn \]

\[ T(n) = \Theta(n \log_k n) \]

\[ \log_a x \log_b a = \log_b x \]

\[ O(n \log_2 n) = O\left(n \log_k^n \log_2^k\right) = O(n \log_k^n) \]
Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $k$ arrays of size $n/k$ each?
Pre-lecture brain teaser

Simpler case: Break into 3 lists:

\[\begin{array}{cccccccc}
1 & 5 & 6 & 8 & 2 & 9 & 7 & 3 & 4 \\
\end{array}\]
Pre-lecture brain teaser

Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say \( k \) arrays of size \( n/k \) each?

What does the recurrence for \( k = 3 \) look like?
Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $k$ arrays of size $n/k$ each?

What does the recurrence for $k = 3$ look like?

$$T(n) = 3T\left(\frac{n}{3}\right) + cn$$

What is the solution to this recurrence?
Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $k$ arrays of size $n/k$ each?

What does the recurrence for $k = 3$ look like?

$$T(n) = 3T\left(\frac{n}{3}\right) + cn$$

What is the solution to this recurrence?

$$T(n) = 3T\left(\frac{n}{3}\right) + cn = O(n \log n)$$
Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $k$ arrays of size $n/k$ each?

What does the recurrence for more general $k$ look like?
Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $k$ arrays of size $n/k$ each?

What does the recurrence for more general $k$ look like?

$$T(n) = kT\left(\frac{n}{k}\right) + cn$$

What is the solution to this recurrence?

$$T(n) = kT\left(\frac{n}{k}\right) + cn = O(n\log n)$$
Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $k$ arrays of size $n/k$ each?

What does the recurrence for more general $k$ look like?

$$T(n) = kT\left(\frac{n}{k}\right) + cn$$

What is the solution to this recurrence?

$$T(n) = kT\left(\frac{n}{k}\right) + cn = O(n\log n)$$

So why don’t we use smaller lists?
Learning Objectives
Learning Objectives

At the end of the lecture, you should be able to understand

- the idea of divide and conquer and how recursion forms a basis of it,
- the quicksort algorithm and its runtime analysis,
- the selection problem, quickselect algorithm and its runtime analysis, and
- the multiplication of numbers problem, a simple divide and conquer algorithm, and Karatsuba’s algorithm, and runtime analysis of these algorithms.
Quick Sort
Quick Sort

Quick Sort [Hoare]

1. Pick a **pivot** element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.
Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.
Quick Sort [Hoare]

1. Pick a pivot element from array $\leq O(1)$

2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$

3. Recursively sort the subarrays, and concatenate them. ?
Quick Sort [Hoare]

1. Pick a pivot element from array

2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$

3. Recursively sort the subarrays, and concatenate them.
Quick Sort: Example

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16

See visualizer: hackerearth.com/practice/algorithms/sorting/quick-sort/visualize
Let $k$ be the rank of the chosen pivot. Then,

\[ T(n) = T(k - 1) + T(n - k) + O(n) \]

For $k = 1$:

\[ T(n) = T(1-1) + T(n-1) + O(n) = T(0) + T(n-1) + O(n) = T(n-1) + O(n) \]

For $k = n$:

\[ T(n) = T(n-1) + T(n-n) + O(n) = T(n-1) + T(n-1) + O(n) = T(n-1) + O(n) \]

For $k = 1$ or $n$:

\[ T(n) = T(n-1) + O(n) \]

\[ T(n) = \Theta(n^2) \]
Let $k$ be the rank of the chosen pivot. Then,

$$T(n) = T(k - 1) + T(n - k) + O(n)$$

If $k = \lceil n/2 \rceil$ then

$$T(n) = T(\lceil n/2 \rceil - 1) + T(\lceil n/2 \rceil) + O(n) \leq 2T(n/2) + O(n).$$

Then, $T(n) = O(n \log n)$. 

$$T(n) \leq 2T(\frac{n}{2}) + O(n)$$

$$\Rightarrow T(n) = O(n \log_2 n)$$
Let $k$ be the rank of the chosen pivot. Then,
$$T(n) = T(k - 1) + T(n - k) + O(n)$$

If $k = \lceil n/2 \rceil$ then
$$T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n).$$
Then, $T(n) = O(n \log n)$. 
Let $k$ be the rank of the chosen pivot. Then,

$$T(n) = T(k - 1) + T(n - k) + O(n)$$

If $k = \lfloor n/2 \rfloor$ then

$$T(n) = T(\lfloor n/2 \rfloor - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n).$$

Then, $T(n) = O(n \log n)$.

Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k - 1) + T(n - k) + O(n))$$

In the worst case $T(n) = T(n - 1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.
Selecting in Unsorted Lists
Big problem with QuickSort is that the pivot might not be the median.

How long would it take us to find the median of an unsorted list?
The Selection Problem

Big problem with QuickSort is that the pivot might not be the median.

How long would it take us to find the median of an unsorted list?
Sort, then $A[n/2]$. Is this the optimal way?
A: an \textbf{unsorted} array of \( n \) integers

For \( 1 \leq j \leq n \), element of rank \( j \) is the \( j \)-th smallest element in \( A \).

\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
  \hline
  Unsorted array & 16 & 14 & 34 & 20 & 12 & 5 & 3 & 19 & 11 \\
  \hline
  Ranks & 6 & 5 & 9 & 8 & 4 & 2 & 1 & 7 & 3 \\
  \hline
  Sort of array & 3 & 5 & 11 & 12 & 14 & 16 & 19 & 20 & 34 \\
  \hline
\end{tabular}
**Problem - Selection**

**Input** Unsorted array $A$ of $n$ integers and integer $j$

**Goal** Find the $j$-th smallest number in $A$ (rank $j$ number)

Median: $j = \lfloor (n + 1)/2 \rfloor$

$A : 9 \ 5 \ 1 \ 7 \ 2 \ \ n = 5 \ \Rightarrow \ \frac{n+1}{2} = 3$

Sorted: $1 \ 2 \ 5 \ 7 \ 9$

Median!
Problem - Selection

**Input**  Unsorted array $A$ of $n$ integers and integer $j$

**Goal**  Find the $j$-th smallest number in $A$ (rank $j$ number)

Median:  $j = \lfloor (n + 1)/2 \rfloor$

Simplifying assumption for sake of notation:  elements of $A$ are distinct
Algorithm 1

- Sort the elements in $A$
- Pick $j$th element in sorted order

Time taken $= O(n \log n)$
Algorithm 1

- Sort the elements in $A$
- Pick $j$th element in sorted order

Time taken $= O(n \log n)$

Do we need to sort? Is there an $O(n)$ time algorithm?
Algorithm II

If $j$ is small or $n - j$ is small then

- Find $j$ smallest/largest elements in $A$ in $O(jn)$ time. (How?)
- Time to find median is $O(n^2)$.

E.g. $j = 1$: We want the (1st) smallest element from $A$!

Claim: We can do that in $O(n) = O(n^2)$ time!

$j = 2$: $O(2n)$ time!

$j = \lfloor \frac{n+1}{2} \rfloor$: $O\left(\frac{n+1}{2}n\right) = O(n^2)$
Quick select
QuickSelect

- Pick a pivot element \( a \) from \( A \)
- Partition \( A \) based on \( a \).

\[
A_{\text{less}} = \{ x \in A \mid x \leq a \} \quad \text{and} \quad A_{\text{greater}} = \{ x \in A \mid x > a \}
\]
- \( |A_{\text{less}}| = j \): return \( a \)
- \( |A_{\text{less}}| > j \): recursively find \( j \)th smallest element in \( A_{\text{less}} \)
- \( |A_{\text{less}}| < j \): recursively find \( k \)th smallest element in \( A_{\text{greater}} \)

where \( k = j - |A_{\text{less}}| \).
Example

<table>
<thead>
<tr>
<th>16</th>
<th>14</th>
<th>34</th>
<th>20</th>
<th>12</th>
<th>5</th>
<th>3</th>
<th>19</th>
<th>11</th>
</tr>
</thead>
</table>
Time Analysis

- **Partitioning step**: $O(n)$ time to scan $A$
- **How do we choose pivot?** Recursive running time?
Time Analysis

- Partitioning step: $O(n)$ time to scan $A$
- How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be $A[1]$. 
• Partitioning step: $O(n)$ time to scan $A$
• How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be $A[1]$

Say $A$ is sorted in increasing order and $j = n$.

How long does this new algorithm take?

$O(n^2)$: Quick Select
Does this help with QuickSort?

Should we combine this with QuickSort?
Does this help with QuickSort?

Should we combine this with QuickSort?

Of course not! It takes $O(n^2)$ which is already the worse case of QuickSort. Need another method....
Does this help with QuickSort?

Looking at the quicksort recurrence again:

\[ T(n) = T(k - 1) + T(n - k) + O(n) \]

Does \( k \) need to be \( n/2 \)?
Does this help with QuickSort?

Looking at the quicksort recurrence again:

\[ T(n) = T(k - 1) + T(n - k) + O(n) \]

Does \( k \) need to be \( n/2 \)?

What if \( k = \frac{3}{5} n \)?
Looking at the quicksort recurrence again:

\[ T(n) = T(k - 1) + T(n - k) + O(n) \]

Does \( k \) need to be \( n/2 \)?

What if \( k = \frac{3}{5} n \)?

What if \( k = \frac{7}{10} n \)?
Does this help with QuickSort?

Looking at the quicksort recurrence again:

\[ T(n) = T(k - 1) + T(n - k) + O(n) \]

Does \( k \) need to be \( n/2 \)?

What if \( k = \frac{3}{5} n \)?

What if \( k = \frac{7}{10} n \)?

we only need to be able to find a rough median! .... How do we do that?
Median of Medians
Divide and Conquer Approach

Idea

- Break input $A$ into many subarrays: $L_1, \ldots, L_k$.
- Find median $m_i$ in each subarray $L_i$.
- Find the median $x$ of the medians $m_1, \ldots, m_k$.
- Intuition: The median $x$ should be close to being a good median of all the numbers in $A$.
- Use $x$ as pivot in previous algorithm.

\[ n \rightarrow \text{groups of 5} \rightarrow \frac{n}{5} \text{ groups} \]
\[ \downarrow \]
\[ \text{Calculate median of each group} \rightarrow O(n) \]
<table>
<thead>
<tr>
<th>11</th>
<th>7</th>
<th>3</th>
<th>42</th>
<th>174</th>
<th>310</th>
<th>1</th>
<th>92</th>
<th>87</th>
<th>12</th>
<th>19</th>
<th>15</th>
</tr>
</thead>
</table>
The second key insight is that the total size of the two recursive subproblems is a constant factor smaller than the size of the original input array. The worst-case running time of the algorithm obeys the recurrence

\[ T(n) = O(n) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right). \]

The recursion tree method implies the solution \( T(n) = O(n) \); the total work at each level of the recursion tree is at most \( \frac{9}{10} \) the total work at the previous level. If we had used blocks of size \( \frac{n}{3} \) instead of \( \frac{n}{5} \), the running time recurrence would have been

\[ T(n) = O(n) + T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right), \]

whose solution is \( O(n \log n) \)—no better than sorting!

Finer analysis reveals that the constant hidden by the \( O() \) is quite large, even if we count only comparisons. Selecting the median of 5 elements requires at most 6 comparisons, so we need at most \( \frac{6n}{5} \) comparisons to set up the recursive subproblem. We need another \( n - 1 \) comparisons to partition the array after the recursive call returns. So a more accurate recurrence for the worst-case number of comparisons is

\[ T(n) = 3 \cdot \frac{1}{2} \left(\frac{n}{5} - 1\right) + 2 + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right). \]

The recursion tree method implies the upper bound

\[ T(n) = 11\frac{1}{5}n. \]

This algorithm isn't as awful in practice as this worst-case analysis predicts—getting a worst-case partition at every level of recursion is incredibly unlikely—but it is still worse than sorting for even moderately large arrays.
Choosing the pivot

- Partition array $A$ into $\lceil n/5 \rceil$ lists of 5 items each.
  

- For each $i$ find median $b_i$ of $L_i$ using brute-force in $O(1)$ time.

  Total $O(n)$ time

- Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

- Find median $b$ of $B$
Choosing the pivot

- Partition array $A$ into $\lceil n/5 \rceil$ lists of 5 items each.
  
  $L_1 = \{A[1], A[2], \ldots, A[5]\}$, $L_2 = \{A[6], \ldots, A[10]\}$, $\ldots$, 
  
  $L_i = \{A[5i+1], \ldots, A[5i-4]\}$, $\ldots$, 
  
  $L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4, \ldots, A[n]\}$.

- For each $i$ find median $b_i$ of $L_i$ using brute-force in $O(1)$ time.
  Total $O(n)$ time

- Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

- Find median $b$ of $B$

  Median of $B$ is an **approximate** median of $A$. That is, if $b$ is used as a pivot to partition $A$, then $|A_{\text{less}}| \leq 7n/10$ and $|A_{\text{greater}}| \leq 7n/10$. 
Algorithm for Selection

`select(A, j):`

1. Form lists $L_1, L_2, \ldots, L_{\lfloor n/5 \rfloor}$ where $L_i = \{A[5i-4], \ldots, A[5i]\}$
2. Find median $b_i$ of each $L_i$ using brute-force
3. Find median $b$ of $B = \{b_1, b_2, \ldots, b_{\lfloor n/5 \rfloor}\}$
4. Partition $A$ into $A_{\text{less}}$ and $A_{\text{greater}}$ using $b$ as pivot
   - if $|A_{\text{less}}| = j$ return $b$
   - else if $|A_{\text{less}}| > j$
     - return `select(A_{\text{less}}, j)`
   - else
     - return `select(A_{\text{greater}}, j - |A_{\text{less}}|)`
Algorithm for Selection

**select**(A, j):
Form lists \( L_1, L_2, \ldots, L_{n/5} \) where \( L_i = \{ A[5i-4], \ldots, A[5i] \} \)
Find median \( b_i \) of each \( L_i \) using brute-force
Find median \( b \) of \( B = \{ b_1, b_2, \ldots, b_{n/5} \} \)
Partition \( A \) into \( A_{\text{less}} \) and \( A_{\text{greater}} \) using \( b \) as pivot
if \(|A_{\text{less}}| = j\) return \( b \)
else if \(|A_{\text{less}}| > j\)
    return **select**(\( A_{\text{less}} \), j)
else
    return **select**(\( A_{\text{greater}} \), \( j - |A_{\text{less}}| \))

How do we find median of \( B \)?
Algorithm for Selection

**select**($A$, $j$):

Form lists $L_1, L_2, \ldots, L_{\lceil n/5 \rceil}$ where $L_i = \{A[5i - 4], \ldots, A[5i]\}$

Find median $b_i$ of each $L_i$ using brute-force

Find median $b$ of $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

Partition $A$ into $A_{\text{less}}$ and $A_{\text{greater}}$ using $b$ as pivot

if ($|A_{\text{less}}| = j$) return $b$

else if ($|A_{\text{less}}| > j$)

    return **select**($A_{\text{less}}$, $j$)

else

    return **select**($A_{\text{greater}}$, $j - |A_{\text{less}}|$)

How do we find median of $B$? **Recursively!**
Median of medians is a good median
There are at least $3n/10$ elements smaller than the median of medians $b$. 

Figure 11.8: Median of medians

Figure 11.9: Median of medians
There are at least $3n/10$ elements smaller than the median of medians $b$.

At least half of the $\lfloor n/5 \rfloor$ groups have at least 3 elements smaller than $b$, except for the group containing $b$ which has 2 elements smaller than $b$. Hence number of elements smaller than $b$ is:

$$3 \left\lfloor \frac{n/5}{2} \right\rfloor + 1 - 1 \geq 3n/10$$
There are at least $\frac{3n}{10}$ elements smaller than the median of medians $b$.

$|A_{\text{greater}}| \leq \frac{7n}{10}$.

Via symmetric argument,

$|A_{\text{less}}| \leq \frac{7n}{10}$.
Running time of deterministic median selection
Running time of deterministic median selection

\[ T(n) \leq T(\lfloor n/5 \rfloor) + \max\{ T(|A_{\text{less}}|), T(|A_{\text{greater}}|) \} + O(n) \]
Running time of deterministic median selection

\[ T(n) \leq T(\lceil n/5 \rceil) + \max \{ T(|A_{\text{less}}|), T(|A_{\text{greater}}|) \} + O(n) \]

From Lemma,

\[ T(n) \leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 \rceil) + O(n) \]

and

\[ T(n) = O(1) \quad n < 10 \]
Running time of deterministic median selection

\[ T(n) \leq T(\lfloor n/5 \rfloor) + \max\{ T(|A_{\text{less}}|), T(|A_{\text{greater}}|)\} + O(n) \]

From Lemma,

\[ T(n) \leq T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + O(n) \]

and

\[ T(n) = O(1) \quad n < 10 \]

Exercise: show that \( T(n) = O(n) \) ?
If the workload is decreasing at every level, then total work is dominated by the root.

\[
T(n) \leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 \rceil) + O(n) = O(n)
\]
\[
\left(\frac{a}{10}\right)^n + \left(\frac{a}{10}\right)^n + \ldots + \left(\frac{a}{10}\right)^{\log n} = O(n)
\]
What about QuickSort?

How would we use the median of medians approach for quicksort?
What about **QuickSort**?

How would we use the median of medians approach for **quicksort**?

Just use MoM if find pivot!

- **Original recurrence:** \( T(n) = T(k - 1) + T(n - k) + O(n) \)

- **With MoM:** \( T(n) = T\left(\frac{3}{10}n\right) + T\left(\frac{7}{10}n\right) + O(n) + O(n) \)

Solve! \( T(n) = \ldots \) (Exercise!)
Due to the following.


"Time bounds for selection".

Median of Medians Algorithm

Due to the following.


“Time bounds for selection”.


How many Turing Award winners in the author list?
Median of Medians Algorithm

Due to the following.


“Time bounds for selection”.


How many Turing Award winners in the author list?

All except Vaughan Pratt!
Median of Medians Algorithm

Due to the following.


“Time bounds for selection”.


How many Turing Award winners in the author list?

All except Vaughan Pratt!

Favorite Knuth quote: He once warned a correspondent, “Beware of bugs in the above code; I have only proved it correct, not tried it.”
Takeaway Points

- **Recursion tree method** and guess and verify are the most reliable methods to analyze recursions in algorithms.
- Recursive algorithms naturally lead to recurrences.
- Sometimes one can look for certain type of recursive algorithms (reverse engineering) by understanding recurrences and their behavior.
Problem statement: Multiplying numbers + a slow algorithm
The Problem: Multiplying numbers

Given two large positive integer numbers $b$ and $c$, with $n$ digits, compute the number $b \times c$. 
Egyptian multiplication: 1850BC (3870 years ago?)
Egyptian multiplication: 1850BC (3870 years ago?)
**Egyptian multiplication: 1850BC (3870 years ago?)**

<table>
<thead>
<tr>
<th>76</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>34 + 1</td>
</tr>
<tr>
<td>76</td>
<td>34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>152</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>152</td>
<td>16 + 1</td>
</tr>
<tr>
<td>152</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>304</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>608</td>
<td>4</td>
</tr>
<tr>
<td>1216</td>
<td>2</td>
</tr>
<tr>
<td>2432</td>
<td>1</td>
</tr>
<tr>
<td>2660</td>
<td>2</td>
</tr>
</tbody>
</table>

34
Egyptian multiplication: 1850BC (3870 years ago?)

\[
\begin{array}{ccc}
76 & 35 & \rightarrow \\
76 & 34 + 1 & 76 \\
76 & 34 & \rightarrow \\
152 & 17 & \\
\end{array}
\]

\[m \cdot n = \frac{2m \cdot n}{2}\]
Egyptian multiplication: 1850BC (3870 years ago?)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>34+1</td>
<td>76</td>
</tr>
<tr>
<td>76</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>16+1</td>
<td>152</td>
</tr>
</tbody>
</table>
Egyptian multiplication: 1850BC (3870 years ago?)

| 76  | 35  |
| 76  | 34 + 1 | 76 |
| 76  | 34  |
| 152 | 17  |
| 152 | 16 + 1 | 152 |
| 152 | 16  |
Egyptian multiplication: 1850BC (3870 years ago?)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>34 + 1</td>
<td>76</td>
</tr>
<tr>
<td>76</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>16 + 1</td>
<td>152</td>
</tr>
<tr>
<td>152</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>304</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Egyptian multiplication: 1850BC (3870 years ago?)

\[
\begin{array}{c|c|c}
76 & 35 & 76 \\
76 & 34 + 1 & 76 \\
76 & 34 & \\
152 & 17 & 152 \\
152 & 16 + 1 & 152 \\
152 & 16 & \\
304 & 8 & \\
608 & 4 & \\
\end{array}
\]
Egyptian multiplication: 1850BC (3870 years ago?)

<table>
<thead>
<tr>
<th>76</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>34 + 1</td>
</tr>
<tr>
<td>76</td>
<td>34</td>
</tr>
<tr>
<td>152</td>
<td>17</td>
</tr>
<tr>
<td>152</td>
<td>16 + 1</td>
</tr>
<tr>
<td>152</td>
<td>16</td>
</tr>
<tr>
<td>304</td>
<td>8</td>
</tr>
<tr>
<td>608</td>
<td>4</td>
</tr>
<tr>
<td>1216</td>
<td>2</td>
</tr>
</tbody>
</table>
Egyptian multiplication: 1850BC (3870 years ago?)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>34</td>
<td>76</td>
</tr>
<tr>
<td>76</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>16</td>
<td>152</td>
</tr>
<tr>
<td>152</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>304</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>608</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1216</td>
<td>1</td>
<td>2432</td>
</tr>
</tbody>
</table>
Egyptian multiplication: 1850BC (3870 years ago?)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>34 + 1</td>
<td>76</td>
</tr>
<tr>
<td>152</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>152</td>
<td>16 + 1</td>
<td>152</td>
</tr>
<tr>
<td>152</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>304</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>608</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1216</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2432</td>
<td>1</td>
<td>2432</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2660</td>
</tr>
</tbody>
</table>
The problem: Multiplying Numbers

**Problem** Given two \( n \)-digit numbers \( x \) and \( y \), compute their product.

**Grade School Multiplication**
Compute “partial product” by multiplying each digit of \( y \) with \( x \) and adding the partial products.

\[
\begin{align*}
3141 & \times 2718 \\
25128 & \\
3141 & \\
21987 & \\
6282 & \\
8537238 & \\
\end{align*}
\]

**Running time:**
\[
\text{Time: } n \cdot n = O(n^2)
\]
Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$
- Number of partial products: $\Theta(n)$
- Addition of partial products: $\Theta(n^2)$
- Total time: $\Theta(n^2)$
Multiplication using **Divide and Conquer**
Divide and Conquer

Assume $n$ is a power of 2 for simplicity and numbers are in decimal.

Split each number into two numbers with equal number of digits

- $b = b_{n-1}b_{n-2} \ldots b_0$ and $c = c_{n-1}c_{n-2} \ldots c_0$
- $b = b_{n-1} \ldots b_{n/2}0 \ldots 0 + b_{n/2-1} \ldots b_0$
- $b(x) = b_Lx + b_R$, where $x = 10^{n/2}$, $b_L = b_{n-1} \ldots b_{n/2}$ and $b_R = b_{n/2-1} \ldots b_0$
- Similarly $c(x) = c_Lx + c_R$ where $c_L = c_{n-1} \ldots c_{n/2}$ and $c_R = c_{n/2-1} \ldots c_0$
Example

\[ 1234 \times 5678 = (12x + 34) \times (56x + 78) \]
\[ = 12 \cdot 56 \cdot x^2 + (12 \cdot 78 + 34 \cdot 56)x + 34 \cdot 78. \]

\[ 1234 \times 5678 = (100 \times 12 + 34) \times (100 \times 56 + 78) \]
\[ = 10000 \times 12 \times 56 \]
\[ + 100 \times (12 \times 78 + 34 \times 56) \]
\[ + 34 \times 78 \]
Divide and Conquer for multiplication

Assume $n$ is a power of 2 for simplicity and numbers are in decimal.

- $b = b_{n-1}b_{n-2}\ldots b_0$ and $c = c_{n-1}c_{n-2}\ldots c_0$
- $b \equiv b(x) = b_Lx + b_R$
  where $x = 10^{n/2}$, $b_L = b_{n-1}\ldots b_{n/2}$ and $b_R = b_{n/2-1}\ldots b_0$
- $c \equiv c(x) = c_Lx + c_R$ where $c_L = c_{n-1}\ldots c_{n/2}$ and
  $c_R = c_{n/2-1}\ldots c_0$
Divide and Conquer for multiplication

Assume \( n \) is a power of 2 for simplicity and numbers are in decimal.

- \( b = b_{n-1} b_{n-2} \ldots b_0 \) and \( c = c_{n-1} c_{n-2} \ldots c_0 \)
- \( b \equiv b(x) = b_L x + b_R \)
  where \( x = 10^{n/2} \), \( b_L = b_{n-1} \ldots b_{n/2} \) and \( b_R = b_{n/2-1} \ldots b_0 \)
- \( c \equiv c(x) = c_L x + c_R \) where \( c_L = c_{n-1} \ldots c_{n/2} \) and
  \( c_R = c_{n/2-1} \ldots c_0 \)

Therefore, for \( x = 10^{n/2} \), we have

\[
bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R)
= b_L c_L x^2 + (b_L c_R + b_R c_L)x + b_R c_R
= 10^n b_L c_L + 10^{n/2}(b_L c_R + b_R c_L) + b_R c_R
\]
Time Analysis

\[ b = b_L \cdot x + b_R \]
\[ c = c_L \cdot x + c_R \]
\[ x = 10^{n/2} \]

\[ bc = 10^n b_L c_L + 10^{n/2} (b_L c_R + b_R c_L) + b_R c_R \]

4 recursive multiplications of number of size \( n/2 \) each plus 4 additions and left shifts (adding enough 0’s to the right)
\[ bc = 10^n b_L c_L + 10^{n/2} (b_L c_R + b_R c_L) + b_R c_R \]

4 recursive multiplications of number of size \( n/2 \) each plus 4 additions and left shifts (adding enough 0’s to the right)

\[ T(n) = 4 T(n/2) + O(n) \quad T(1) = O(1) \]
\[ bc = 10^n b_L c_L + 10^{n/2} (b_L c_R + b_R c_L) + b_R c_R \]

4 recursive multiplications of number of size \( n/2 \) each plus 4 additions and left shifts (adding enough 0's to the right)

\[ T(n) = 4 T(n/2) + O(n) \quad T(1) = O(1) \]

\[ T(n) = \Theta(n^2) \]. No better than grade school multiplication!
Faster multiplication: Karatsuba’s Algorithm
A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 “Prince of Mathematicians”

Observation: Multiply two complex numbers: \((a + bi)\) and \((c + di)\)

\[(a + bi)(c + di) = ac - bd + (ad + bc)i\]
A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 “Prince of Mathematicians”

Observation: Multiply two complex numbers: \((a + bi)\) and \((c + di)\)

\[(a + bi)(c + di) = ac - bd + (ad + bc)i\]

How many multiplications do we need?
A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 “Prince of Mathematicians”

Observation: Multiply two complex numbers: \((a + bi)\) and \((c + di)\)

\[(a + bi)(c + di) = ac - bd + (ad + bc)i\]

How many multiplications do we need?

Only 3! If we do extra additions and subtractions.
Compute \(ac, bd, (a + b)(c + d)\). Then
Gauss technique for polynomials

\[ p(x) = ax + b \quad \text{and} \quad q(x) = cx + d. \]

\[ p(x)q(x) = acx^2 + (ad + bc)x + bd. \]
Gauss technique for polynomials

\[ p(x) = ax + b \quad \text{and} \quad q(x) = cx + d. \]

\[ p(x)q(x) = acx^2 + (ad + bc)x + bd. \]

\[ p(x)q(x) = acx^2 + (a + b)(c + d) - ac - bd)x + bd. \]
Improving the Running Time

\[ bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) \]
Improving the Running Time

\[ bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) \]
\[ = b_L c_L x^2 + (b_L c_R + b_R c_L)x + b_R c_R \]
Improving the Running Time

\[bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R)\]
\[= b_L c_L x^2 + (b_L c_R + b_R c_L)x + b_R c_R\]
\[= (b_L \cdot c_L)x^2 + \left( (b_L + b_R) \cdot (c_L + c_R) - b_L \cdot c_L - b_R \cdot c_R \right)x\]
\[+ b_R \cdot c_R\]
Improving the Running Time

\[ bc = b(x)c(x) = (b_Lx + b_R)(c_Lx + c_R) \]
\[ = b_Lc_Lx^2 + (b_Lc_R + b_Rc_L)x + b_Rc_R \]
\[ = (b_L * c_L)x^2 + \left((b_L + b_R) * (c_L + c_R) - b_L * c_L - b_R * c_R\right)x \]
\[ + b_R * c_R \]

Recursively compute only \( b_Lc_L, b_Rc_R, (b_L + b_R)(c_L + c_R) \).
Improving the Running Time

\[ bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) \]
\[ = b_L c_L x^2 + (b_L c_R + b_R c_L)x + b_R c_R \]
\[ = (b_L \ast c_L)x^2 + \left( (b_L + b_R) \ast (c_L + c_R) - b_L \ast c_L - b_R \ast c_R \right)x \]
\[ + b_R \ast c_R \]

Recursively compute only \( b_L c_L, b_R c_R, (b_L + b_R)(c_L + c_R) \).

**Time Analysis**
Running time is given by

\[ T(n) = 3T(n/2) + O(n) \]
\[ T(1) = O(1) \]

which means \( T(n) = O(n^{\log_2 3}) = O(n^{1.585}) \leq O(n^2) \)
State of the Art

Schönhage-Strassen 1971: $O(n \log n \log \log n)$ time using Fast-Fourier-Transform (FFT)

Martin Fürer 2007: $O(n \log n 2^{O(\log^* n)})$ time

Conjecture: There is an $O(n \log n)$ time algorithm.

CS/ECE 374-B, SP’24