Karatsuba’s Algorithm and Linear Time Selection

Lecture 11
We will learn

1. Last lecture
   1. How to think about recursion as a design paradigm
   2. How to analyze running time recurrences

2. More complicated recursion in action
   1. Fast multiplication (Karatsuba’s Algorithm)
   2. Linear Time Selection
Another way to think about it

*Reduce* a problem to *smaller* instances of the *same* problem.
Reduction

Reduction = Delegation

- Solve a problem using elementary operations + call a bunch of subroutines
- Subroutines = Black boxes
Example of Reduction: Distinct Elements Problem

**Problem** Given an array \( A \) of \( n \) integers, are there any *duplicates* in \( A \)?
Example of Reduction: Distinct Elements Problem

**Problem** Given an array $A$ of $n$ integers, are there any *duplicates* in $A$?

Naive algorithm:

```plaintext
DistinctElements(A[1..n])
    for $i = 1$ to $n - 1$ do
        for $j = i + 1$ to $n$ do
                return YES
        return NO
```

Running time: $O(n^2)$
Example of Reduction: Distinct Elements Problem

**Problem** Given an array $A$ of $n$ integers, are there any *duplicates* in $A$?

**Naive algorithm:**

```
DistinctElements(A[1..n])
    for $i = 1$ to $n - 1$ do
        for $j = i + 1$ to $n$ do
                return YES
        return NO
```

**Running time:**
Problem: Given an array $A$ of $n$ integers, are there any duplicates in $A$?

Naive algorithm:

```plaintext
DistinctElements(A[1..n])
    for i = 1 to n - 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
        return NO
```

Running time: $O(n^2)$
Reduction to Sorting

**DistinctElements**($A[1..n]$)

Sort $A$

for $i = 1$ to $n - 1$

if ($A[i] = A[i + 1]$) then

    return YES

return NO
Reduction to Sorting

DistinctElements(A[1..n])

Sort A
for i = 1 to n - 1 do
    if (A[i] = A[i + 1]) then
        return YES
    return NO

Running time: $O(n)$ plus time to sort an array of $n$ numbers

Important point: algorithm uses sorting as a black box
Recursion

It requires discipline to delegate

It is important to think of the recursive calls as black boxes, that is, subroutines taken care of by the recursion fairy.
Recursion

It requires discipline to delegate

It is important to think of the recursive calls as black boxes, that is, subroutines taken care of by the recursion fairy.

\[
\text{MergeSort}(A[1..n]):
\text{if } n > 1
\]
\[
m \leftarrow \lfloor n/2 \rfloor
\]
\[
\text{MergeSort}(A[1..m])
\]
\[
\text{MergeSort}(A[m+1..n])
\]
\[
\text{Merge}(A[1..n], m)
\]
Solving Recurrences

Two general methods:

1. Guess and Verify

2. Recursion tree method: At every level of recursion, how much *non-recursive* work you are doing.
Solving Recurrences

Two general methods:

1. Guess and Verify

2. Recursion tree method: At every level of recursion, how much *non-recursive* work you are doing.
   - Merge Sort: same amount of work at every level
Solving Recurrences

Two general methods:

1. Guess and Verify
2. Recursion tree method: At every level of recursion, how much \textit{non-recursive} work you are doing.
   
   1. Merge Sort: same amount of work at every level
   2. Increasing geometric series: count number of leaves. (Fast multiplication)
   3. Decreasing geometric series: summable, first level dominates. (Selection)
Part I

Fast Multiplication
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.

\[
\begin{align*}
3141 \\
\times 2718 \\
\hline
25128 \\
3141 \\
21987 \\
6282 \\
\hline
8537238
\end{align*}
\]
Time Analysis of Grade School Multiplication

1. Each partial product: $\Theta(n)$
2. Number of partial products: $\Theta(n)$
3. Addition of partial products: $\Theta(n^2)$
4. Total time: $\Theta(n^2)$
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

1. $x = x_{n-1}x_{n-2} \cdots x_0$ and $y = y_{n-1}y_{n-2} \cdots y_0$
2. $x = x_{n-1} \cdots x_{n/2}0 \cdots 0 + x_{n/2-1} \cdots x_0$
3. $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1} \cdots x_{n/2}$ and $x_R = x_{n/2-1} \cdots x_0$
4. Similarly $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1} \cdots y_{n/2}$ and $y_R = y_{n/2-1} \cdots y_0$
Example

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

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

1. $x = x_{n-1}x_{n-2} \cdots x_0$ and $y = y_{n-1}y_{n-2} \cdots y_0$

2. $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1} \cdots x_{n/2}$ and $x_R = x_{n/2-1} \cdots x_0$

3. $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1} \cdots y_{n/2}$ and $y_R = y_{n/2-1} \cdots y_0$

Therefore

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$

$$= 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R$$
Time Analysis

\[
x y = (10^{n/2} x_L + x_R)(10^{n/2} y_L + y_R)
\]
\[
= 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R
\]

4 recursive multiplications of size \( n/2 \) plus 3 additions and left shifts (adding enough 0’s to the right)
Time Analysis

\[ xy = (10^{n/2} x_L + x_R)(10^{n/2} y_L + y_R) \]
\[ = 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R \]

4 recursive multiplications of size \( n/2 \) plus 3 additions and left shifts (adding enough 0’s to the right)

\[ T(n) = 4 T(n/2) + O(n) \]
\[ T(1) = O(1) \]
\[ xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \]
\[ = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \]

4 recursive multiplications of size \( n/2 \) plus 3 additions and left shifts (adding enough 0’s to the right)

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

\[ T(n) = \Theta(n^2) \text{. No better than grade school multiplication!} \]
Recursion Tree
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

\[(ad + bc) = (a + b)(c + d) - ac - bd\]
Improving the Running Time

\[ xy = \left(10^{n/2} x_L + x_R\right) \left(10^{n/2} y_L + y_R\right) \]
\[ = 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R \]

Gauss trick: \[ x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R \]
Improving the Running Time

\[ xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \]
\[ = 10^n x_Ly_L + 10^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R \]

Gauss trick: \( x_Ly_R + x_Ry_L = (x_L + x_R)(y_L + y_R) - x_Ly_L - x_Ry_R \)

Recursively compute only \( x_Ly_L, x_Ry_R, (x_L + x_R)(y_L + y_R) \).
Improving the Running Time

\[
x y = (10^{n/2} x_L + x_R)(10^{n/2} y_L + y_R) \\
= 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R
\]

Gauss trick: \( x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R \)

Recursively compute only \( x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R) \).

**Time Analysis**

Running time is given by

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

which means
Improving the Running Time

\[ xy = (10^{n/2} x_L + x_R)(10^{n/2} y_L + y_R) \]
\[ = 10^n x_L y_L + 10^{n/2} (x_L y_R + x_R y_L) + x_R y_R \]

Gauss trick: \( x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R \)

Recursively compute only \( x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R) \).

### Time Analysis

Running time is given by

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

which means \( T(n) = O(n^{\log_2 3}) = O(n^{1.585}) \)
Analyzing the Recurrences

1. Basic divide and conquer: \( T(n) = 4T(n/2) + O(n) \), \( T(1) = 1 \). **Claim:** \( T(n) = \Theta(n^2) \).

2. Saving a multiplication: \( T(n) = 3T(n/2) + O(n) \), \( T(1) = 1 \). **Claim:** \( T(n) = \Theta(n^{\log_2 3}) \).
Analyzing the Recurrences

1. Basic divide and conquer: \( T(n) = 4T(n/2) + O(n) \), \( T(1) = 1 \). **Claim:** \( T(n) = \Theta(n^2) \).

2. Saving a multiplication: \( T(n) = 3T(n/2) + O(n) \), \( T(1) = 1 \). **Claim:** \( T(n) = \Theta(n^{\log_2 3}) \).

Use recursion tree method:

1. In both cases, depth of recursion \( L = \log n \).

2. Work at depth \( i \) is \( 4^i n/2^i \) and \( 3^i n/2^i \) respectively: number of children at depth \( i \) times the work at each child.

3. Total work is therefore \( n \sum_{i=0}^{L} 2^i \) and \( n \sum_{i=0}^{L} (3/2)^i \) respectively.
Part II

Selecting in Unsorted Lists
**Rank of element in an array**

**A**: an unsorted array of \( n \) integers

**Definition**

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

<table>
<thead>
<tr>
<th>Unsorted array</th>
<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>
<tbody>
<tr>
<td>Ranks</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Sort of array</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>19</td>
<td>20</td>
<td>34</td>
</tr>
</tbody>
</table>
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: $j = \left\lceil (n + 1)/2 \right\rceil$
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 I

1. Sort the elements in $A$
2. Pick $j$th element in sorted order

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

1. Sort the elements in $A$
2. 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?
Quicksort is another recursive sorting algorithm, discovered by Tony Hoare in 1962 and first published in 1961. In this algorithm, the hard work is splitting the array into smaller subarrays before recursion, so that merging the sorted subarrays is trivial.

Choose a pivot element from the array.

Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.

Recursively quicksort the first and last subarrays.

More detailed pseudocode is given in Figure 8. In the P subroutine, the input parameter p is the index of the pivot element in the unsorted array; the subroutine partitions the array and returns the new index of the pivot element. There are many different efficient partitioning algorithms; the one I'm presenting here is attributed to Nico Lomuto.

The variable `counts the number of items in the array that are `less than the pivot element.

```plaintext
QUICKSORT(A[1..n]):
    if (n > 1)
        Choose a pivot element A[p]
        r ← PARTITION(A, p)
        QUICKSORT(A[1..r - 1])  ⟨Recursively⟩
        QUICKSORT(A[r + 1..n])  ⟨Recursively⟩
```

Correctness

Just like mergesort, proving that QUICKSORT is correct requires two separate induction proofs: one to prove that PARTITION correctly partitions the array, and...
Algorithm II. One-armed Quick Sort

**Algorithm:** QuickSelect

`QuickSelect(A[1..n], k):`
- if `n = 1`
  - return `A[1]`
- else
  - Choose a pivot element `A[p]`
  - `r ← Partition(A[1..n], p)`
  - if `k < r`
    - return `QuickSelect(A[1..r-1], k)`
  - else if `k > r`
    - return `QuickSelect(A[r+1..n], k-r)`
  - else
    - return `A[r]`
Running Time Analysis

1 Partitioning step: $O(n)$ time to scan $A$

2 $T(n) = \max_{1 \leq k \leq n} \max (T(k - 1), T(n - k)) + O(n)$

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.
A Better Pivot

Suppose pivot is the $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$. That is pivot is *approximately* in the middle of $A$

Then $n/4 \leq |A_{less}| \leq n/2$ and $n/2 \leq |A_{greater}| \leq 3n/4$. If we apply recursion,
A Better Pivot

Suppose pivot is the $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$. That is pivot is *approximately* in the middle of $A$. Then $n/4 \leq |A_{\text{less}}| \leq n/2$ and $n/2 \leq |A_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!
A Better Pivot

Suppose pivot is the $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$. That is pivot is *approximately* in the middle of $A$.

Then $n/4 \leq |A_{\text{less}}| \leq n/2$ and $n/2 \leq |A_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!

How do we find such a pivot?
A Better Pivot

Suppose pivot is the $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$. That is pivot is *approximately* in the middle of $A$.

Then $n/4 \leq |A_{\text{less}}| \leq n/2$ and $n/2 \leq |A_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!

How do we find such a pivot? Randomly?
A Better Pivot

Suppose pivot is the \( \ell \)th smallest element where \( n/4 \leq \ell \leq 3n/4 \).
That is pivot is \textit{approximately} in the middle of \( A \).
Then \( n/4 \leq |A_{\text{less}}| \leq n/2 \) and \( n/2 \leq |A_{\text{greater}}| \leq 3n/4 \).
If we apply recursion,

\[
T(n) \leq T(3n/4) + O(n)
\]

Implies \( T(n) = O(n) \)!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.
A Better Pivot

Suppose pivot is the $\ell$th smallest element where $\frac{n}{4} \leq \ell \leq \frac{3n}{4}$. That is pivot is \textit{approximately} in the middle of $A$

Then $\frac{n}{4} \leq |A_{\text{less}}| \leq \frac{n}{2}$ and $\frac{n}{2} \leq |A_{\text{greater}}| \leq \frac{3n}{4}$. If we apply recursion,

$$T(n) \leq T\left(\frac{3n}{4}\right) + O(n)$$

Implies $T(n) = O(n)$!

How do we find such a pivot? Randomly? In fact works! Analysis a little bit later.

Can we choose pivot deterministically?
Divide and Conquer Approach
A game of medians

Idea

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

$$T(n) \leq O(n) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

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 \(\frac{n}{5}\) instead of \(\frac{n}{3}\), the running time recurrence would have been

$$T(n) \leq O(n) + T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right),$$

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

$$T(n) \leq 11\frac{n}{5} + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

The recursion tree method implies the upper bound

$$T(n) \leq 11\left(\frac{n}{5}\right)^\infty \sum_{i=0}^{\infty} \frac{9}{10} = 22n.$$

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.
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

\[ T(n) = \frac{11n}{5} + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right). \]

The recursion tree method implies the upper bound

\[ T(n) \leq \frac{11n}{5} \cdot \frac{9}{10} \cdot \frac{9}{10} \cdot \frac{9}{10} = \frac{22n}{25}. \]

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.
Median of median

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

\[ T(n) \leq \Theta(n) + T(n/5) + T(7n/10). \]

The recursion tree method implies the solution

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

the total work at each level of the recursion tree is at most \( 9/10 \) the total work at the previous level. If we had used blocks of size \( \approx \) instead of \( \approx \), the running time recurrence would have been

\[ T(n) \leq \Theta(n) + T(n/3) + T(2n/3), \]

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

Finer analysis reveals that the constant hidden by the \( \Theta() \) 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 \( 6n/5 \) comparisons to set up the recursive subproblem. We need another \( n \) comparisons to partition the array after the recursive call returns.

So a more accurate recurrence for the worst-case number of comparisons is

\[ T(n) \leq 11n/5 + T(n/5) + T(7n/10). \]

The recursion tree method implies the upper bound

\[ T(n) \leq 11n/5 \cdot 9/10 = 22n. \]

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.
Median of median

Lemma

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

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

The recursion tree method implies the solution $T(n) = O(n)$; the total work at each level of the recursion tree is at most $9/10$ the total work at the previous level. If we had used blocks of size $\frac{3}{5}$ instead of $\frac{2}{3}$, the running time recurrence would have been $T(n) \leq O(n) + T(n/3) + T(2n/3)$, 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 $6n/5$ comparisons to set up the recursive subproblem.

We need another $n$ comparisons to partition the array after the recursive call returns. So a more accurate recurrence for the worst-case number of comparisons is

\[ T(n) \leq 11n/5 + T(n/5) + T(7n/10) \]

The recursion tree method implies the upper bound

\[ T(n) \leq 11n/5 \times \left( \frac{9}{10} \right)^l \]

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.
Algorithm for Selection

A storm of medians

\[
\text{select}(A, j)
\]

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 \text{select}(A_{\text{less}}, j)
else
    return \text{select}(A_{\text{greater}}, j - |A_{\text{less}}|)
Algorithm for Selection
A storm of medians

\textbf{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

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

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

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

\textbf{else}

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

How do we find median of $B$?
Algorithm for Selection

A storm of medians

**select**(A, j):

1. Form lists \(L_1, L_2, \ldots, L_{\lceil n/5 \rceil}\) 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 \(B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}\)
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 \(\text{select}(A_{\text{less}}, j)\)
   
   else
     
     return \(\text{select}(A_{\text{greater}}, j - |A_{\text{less}}|)\)

How do we find median of \(B\)? Recursively!
Algorithm for Selection
A storm of medians

\texttt{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

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

$b = \text{select}(B, \lceil n/10 \rceil)$

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

else

return $\text{select}(A_{\text{greater}}, j - |A_{\text{less}}|)$
Running time of deterministic median selection

A dance with recurrences

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

A dance with recurrences

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

From Lemma,

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

and

\[ T(n) = O(1) \quad n < 10 \]
Why 5? How about 3?