Divide and Conquer II

Summer 2018 • Lecture 07/03

Announcements

- Homework 0 solutions
- Homework 1
  - **`hw1.zip`** is live!
  - It's due next Tuesday 7/10, but start early!
  - You've learned most of the required material.
- Tutorial 2
  - Friday, 7/6 3:30-4:50 p.m. in STLC 115.
  - RSVP, so I can print enough copies for everyone: https://goo.gl/forms/MSGUGEVBnXaS21kR2 (requires Stanford email).

Course Overview

- Algorithmic Analysis
- **Divide and Conquer**
- Randomized Algorithms
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

- Divide and Conquer II
    - Substitution method
    - Linear-time selection
        - Proving correctness
        - Proving runtime with recurrence relations
    - *Problems: selection*
    - *Algorithms: Select*
    - Reading: CLRS 9

So far...

	Proving correctness	Proving runtime
Iterative	Induction on the iteration, leveraging a loop variant (e.g. insertion sort)	Intuition
Recursive	Induction on the input size (e.g. mergesort)	Defining and solving recurrence relations

So far...

	Proving correctness	Proving runtime
Iterative	Induction on the iteration, leveraging a loop variant (e.g. insertion sort)	Intuition
Recursive	Induction on the input size (e.g. mergesort)	**Defining and solving recurrence relations**

So far...

- **Divide-and-conquer algorithms via defining and solving recurrence relations**
  - After deriving the recurrence relation, we learned several methods to find the closed-form runtime expression: recursion-tree method, iteration method, Master method.
  - Today, I owe you another method: **substitution method**!

Substitution Method

Substitution Method

1. Guess what the answer is.
2. Formally prove that's what the answer is.

- Let's try it out with an example recurrence from last time:
    - $T(1) \leq 1$
    - $T(n) \leq 2T(n/2) + n$

$T(1) \le 1$

$T(n) \le 2T(n/2) + n$

Substitution Method

1. **Guess what the answer is.**

- Try unwinding it...
  $T(n) = 2T(n/2) + n$
  $T(n) = 2(2T(n/4) + n/2) + n$
  $T(n) = 4T(n/4) + 2n$
  $T(n) = 4(2T(n/8) + n/4) + 2n$
  $T(n) = 8T(n/8) + 3n$
  …
- Following the pattern…
  $T(n) = nT(1) + n\log(n) =$ **$n(\log(n) + 1)$**

$T(1) \le 1$

$T(n) \le 2T(n/2) + n$

Substitution Method

2. **Formally prove that's what the answer is.**

- **Inductive hypothesis** $T(k) \le k(\log(k) + 1)$ for all $1 \le k < n$.
- **Base case** $T(1) = 1 = 1(\log(1) + 1)$.
- **Inductive step**
  - $T(n) = 2T(n/2) + n$ Substitute n/2 into inductive hyp.
    $\le 2((n/2)(\log(n/2) + 1) + n$
    $= 2((n/2)(\log(n) - 1 + 1) + n$
    $= 2((n/2) \log(n)) + n$
    $= n(\log(n) + 1)$
- **Conclusion** By induction, $T(n) = n(\log(n) + 1)$ for all $n > 0$.

Substitution Method

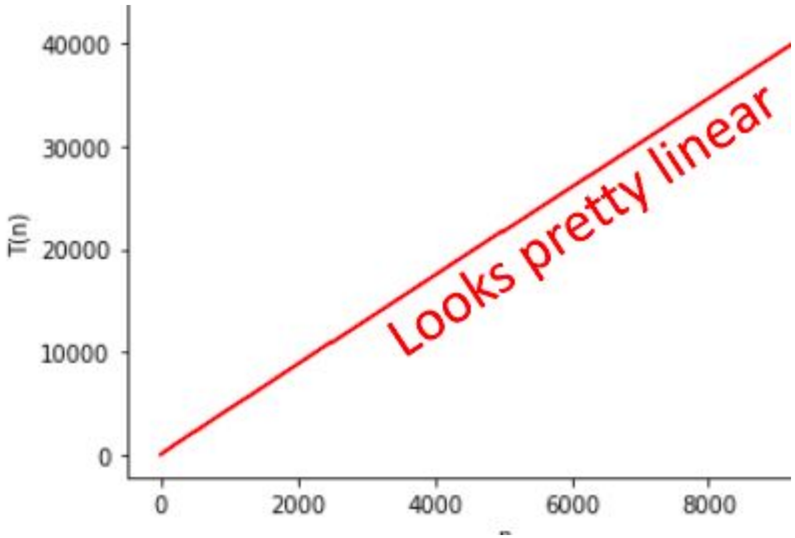
- Let's try it out with a new recurrence:
  - $T(n) = 10n$ when $1 \le n \le 10$
  - $T(n) = 3n + T(n/5) + T(n/2)$ otherwise

Substitution Method

$T(n) = 10n$ when $1 \leq n \leq 10$

$T(n) = 3n + T(n/5) + T(n/2)$ otherwise

1. **Guess what the answer is.**
   - Try unwinding it
     [Whiteboard] - Gets ugly fast.
   - Try plotting it



Guess O(n)

Substitution Method

$T(n) = 10n$ when $1 \le n \le 10$

$T(n) = 3n + T(n/5) + T(n/2)$ otherwise

2. **Formally prove that's what the answer is.**

- **Inductive hypothesis** $T(k) \le Ck$ for all $1 \le k < n$.
- **Base case** $T(k) \le Ck$ for all $k \le 10$.
- **Inductive step**
  - $T(n) = 3n + T(n/5) + T(n/2)$
    $\le 3n + C(n/5) + C(n/2)$
    $= 3n + (C/5)n + (C/2)n$
    $\le Cn$
- **Conclusion** There exists some C such that for all $n > 1$, $T(n) \le Cn$. Therefore, $T(n) = O(n)$.

C is some constant we'll have to fill in later!

C must be ≥ 10 since the recurrence states $T(k) = 10k$ when $1 \le k \le 10$

Solve for C to satisfy the inequality. $C = 10$ works.

Substitution Method

$T(n) = 10n$ when $1 \le n \le 10$

$T(n) = 3n + T(n/5) + T(n/2)$ otherwise

2. **Formally prove that's what the answer is.**
   - **Inductive hypothesis** $T(k) \le \mathbf{10}k$ for all $1 \le k < n$.
   - **Base case** $T(k) \le \mathbf{10}k$ for all $k \le 10$.
   - **Inductive step**
     - $$\begin{aligned} T(n) &= 3n + T(n/5) + T(n/2) \\ &\le 3n + \mathbf{10}(n/5) + \mathbf{10}(n/2) \\ &= 3n + (\mathbf{10}/5)n + (\mathbf{10}/2)n \\ &\le \mathbf{10}n \end{aligned}$$
   - **Conclusion** For all $n > 1$, $T(n) \le \mathbf{10}n$. Therefore, $T(n) = O(n)$.

Pretend we knew **C** = 10 all along.

Substitution Method

1. Guess what the answer is.
2. Formally prove that's what the answer is.
   - Might need to leave some constants unspecified until the end and see what they need to be for the proof to work.

Today's Outline

- Divide and Conquer II
    - ~~Substitution method~~ **Done!**
    - Linear-time selection
        - Proving correctness
        - Proving runtime with recurrence relations
    - *Problems: selection*
    - *Algorithms: Select*
    - Reading: CLRS 9

Linear-Time Selection

Linear-Time Selection

- **Task** Find the k^{th} smallest element in an unsorted list in O(n)-time.

1	**64**	**9**	**49**	**16**	**4**	**0**	**25**	**36**	**81**

  - Such an algorithm could find the min in O(n)-time if k=0 or the max if k=n-1.
  - Such an algorithm could find the median in O(n)-time if $k=\lceil n/2 \rceil-1$ (this definition allows the median of lists of even-length to always be elements of the list, as opposed to the average of two elements).

Linear-Time Selection

- **Finding the min and max** Iterate through the list and keep track of the smallest and largest elements. Runtime **O(n)**.
- **Finding the k^{th} smallest element (naive)** Sort the list and return the element in index k of the sorted list.

1	64	9	49	16	4	0	25	36	81

k=3

↓

0	1	4	9	16	25	36	49	64	81

Not Quite Linear-Time Selection

```
def naive_select(A, k):
  A = mergesort(A)
  return A[k]
```

Worst-case runtime $\Theta(n \log(n))$

Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.

1	**64**	**9**	**49**	**16**	**4**	**0**	**25**	**36**	**81**

Linear-Time Selection

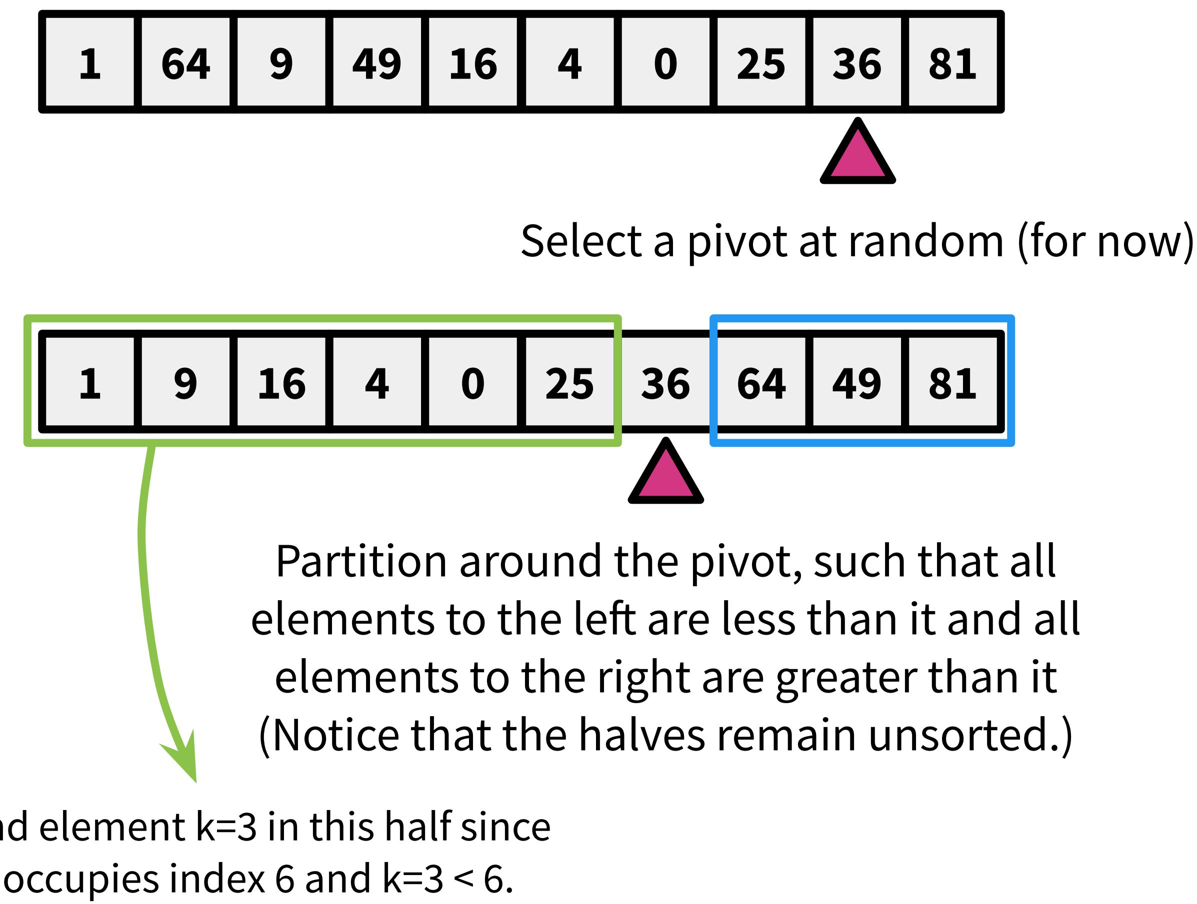
- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.

1	**64**	**9**	**49**	**16**	**4**	**0**	**25**	**36**	**81**

Select a pivot at random (for now)

Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.



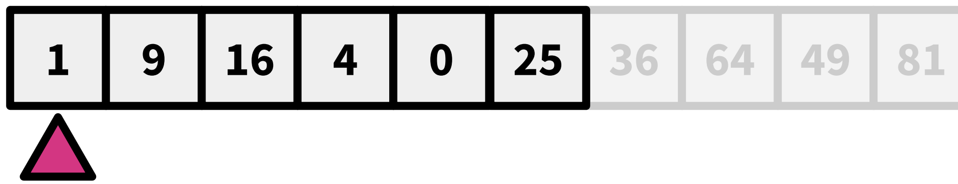
Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.

1	**9**	**16**	**4**	**0**	**25**	**36**	**64**	**49**	**81**

Linear-Time Selection

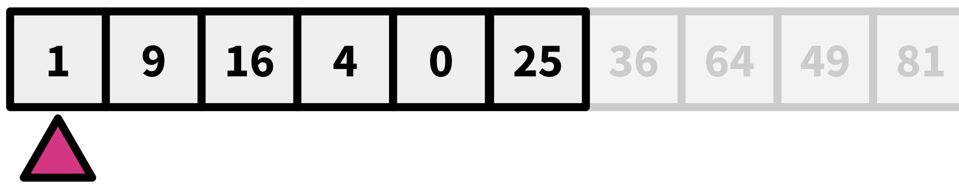
- **Key Insight** Select a pivot, partition around it, and recurse.
    - Suppose we want to find element **k=3**.



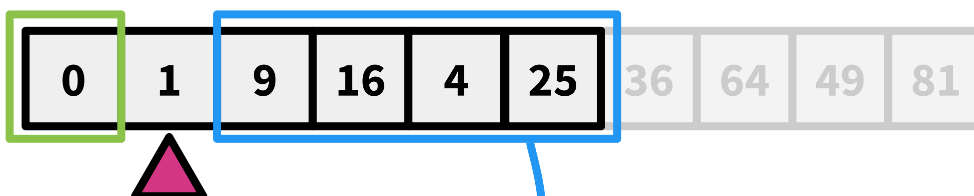
Select another pivot at random (for now)

Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.



Select another pivot at random (for now)

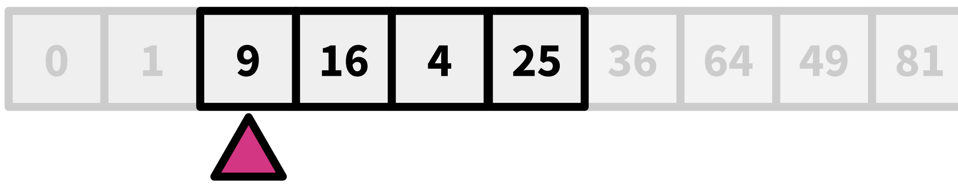


Partition around the pivot

Find element k=3-(1+1) in this half since
1 occupies index 1 and k=3 > 1.

Linear-Time Selection

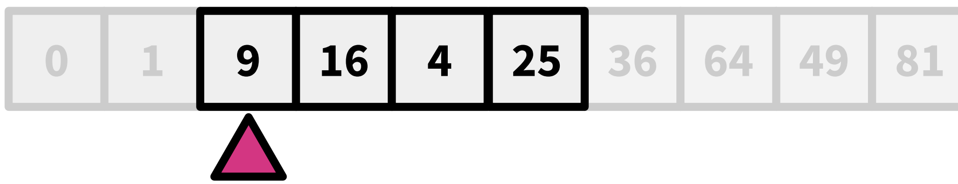
- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.



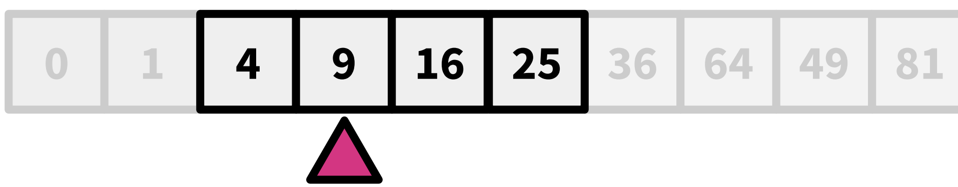
Select another pivot at random (for now)

Linear-Time Selection

- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.



Select another pivot at random (for now)



Partition around the pivot

We found the element!

Linear-Time Selection

```
def select(A, k, c=100):
```

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
```

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
```

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
```

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
```

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

“Worst-case” runtime $\Theta(n^2)$

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

<u>"Worst-case"</u> runtime $\Theta(n^2)$

We'll discuss this runtime later...

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

Note: this is different from the "worst-case" we saw for insertion sort (we'll revisit during Randomized Algs).

"Worst-case" runtime $\Theta(n^2)$

We'll discuss this runtime later...

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
  left, right = [], []
  for i in range(len(A)):
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
  left, right = [], []
  for i in range(len(A)):
    if A[i] == pivot: continue
```

Linear-Time Selection

```python
def partition_about_pivot(A, pivot):
  left, right = [], []
  for i in range(len(A)):
    if A[i] == pivot: continue
    elif A[i] < pivot:
      left.append(A[i])
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
  left, right = [], []
  for i in range(len(A)):
    if A[i] == pivot: continue
    elif A[i] < pivot:
      left.append(A[i])
    else:
      right.append(A[i])
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
  left, right = [], []
  for i in range(len(A)):
    if A[i] == pivot: continue
    elif A[i] < pivot:
      left.append(A[i])
    else:
      right.append(A[i])
  return left, right
```

Linear-Time Selection

```
def partition_about_pivot(A, pivot):
  left, right = [], []
  for i in range(len(A)):
    if A[i] == pivot: continue
    elif A[i] < pivot:
      left.append(A[i])
    else:
      right.append(A[i])
  return left, right
```

<u>Worst-case</u> runtime $\Theta(n)$

Linear-Time Selection

- **Intuition** Partition the list about a pivot selected at random, either return the pivot itself or recurse on the left or right sublists (but not both).
- You might have two questions at this point…
  1. **Does this actually work?**
  2. **Is it fast?**

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Linear-Time Selection

- **Intuition** Partition the list about a pivot selected at random, either return the pivot itself or recurse on the left or right sublists (but not both).
- You might have two questions at this point…
  1. **Does this actually work?**
  2. **Is it fast?**

```python
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Linear-Time Selection

1. **Does this actually work?** We've already seen an example!
   - Formally, similar to last time, we proceed by induction, inducting on the length of the input list.

```python
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Proving Correctness

```
def proof_of_correctness_helper(algorithm):
  if algorithm.type == "iterative":
    # 1) Find the loop invariant
    # 2) Define the inductive hypothesis
    #    (internal state at iteration i)
    # 3) Prove the base case (i=0)
    # 4) Prove the inductive step (i => i+1)
    # 5) Prove the conclusion (i=n => correct)
  elif algorithm.type == "recursive":
    # 1) Define the inductive hypothesis
    #    (correct for inputs of sizes 1 to i)
    # 2) Prove the base case (i < small constant)
    # 3) Prove the inductive step (i => i+1 OR
         {1,2,...,i} => i+1)
    # 4) Prove the conclusion (i=n => correct)
  # TODO
```

Proving Correctness

- Recall, there are four components in a proof by induction.
  - **Inductive Hypothesis** The algorithm works on input lists of length 1 to i.
  - **Base case** The algorithm works on input lists of length 1.
  - **Inductive step** If the algorithm works on input lists of length 1 to i, then it works on input lists of length i+1.
  - **Conclusion** If the algorithm works on input lists of length n, then it works on the entire list.

Proving Correctness

- Formally, for **`select`**...

Proving Correctness

- Formally, for **`select`**...
  - **Inductive Hypothesis** **`select(A,k)`** correctly finds the k^{th}-smallest element for inputs of length 1 to i.

Proving Correctness

- Formally, for `select`...
  - **Inductive Hypothesis** `select(A,k)` correctly finds the k^{th}-smallest element for inputs of length 1 to i.
  - **Base case** `select(A,k)` correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.

Proving Correctness

- Formally, for `select`...
  - **Inductive Hypothesis** `select(A,k)` correctly finds the k^{th}-smallest element for inputs of length 1 to i.
  - **Base case** `select(A,k)` correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.
  - **Inductive step** Suppose the algorithm works on input lists of length 1 to i. Calling `select(A,k)` on an input list of length i+1 selects a pivot, partitions around it, and compares the length of the left list to k. There are three cases:

Proving Correctness

- Formally, for **`select`**...
  - **Inductive Hypothesis `select(A,k)`** correctly finds the kth-smallest element for inputs of length 1 to i.
  - **Base case `select(A,k)`** correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.
  - **Inductive step** Suppose the algorithm works on input lists of length 1 to i. Calling **`select(A,k)`** on an input list of length i+1 selects a pivot, partitions around it, and compares the length of the left list to k. There are three cases:
    - **`len(left) == k`**: exactly k items less than the pivot, so return the pivot.
    - **`len(left) > k`**: More than k items less than the pivot, so return the kth-smallest element of the left half of the list.
    - **`len(left) < k`**: There are fewer than k items ≤ to the pivot, so return the (k - len(left) - 1)st-smallest element of the right half of the list.

Proving Correctness

- Formally, for **`select`**...
  - **Inductive Hypothesis `select(A,k)`** correctly finds the k^{th}-smallest element for inputs of length 1 to i.
  - **Base case `select(A,k)`** correctly finds the smallest element for inputs of length 1; it returns the element itself which is trivially the smallest.
  - **Inductive step** Suppose the algorithm works on input lists of length 1 to i. Calling **`select(A,k)`** on an input list of length i+1 selects a pivot, partitions around it, and compares the length of the left list to k. There are three cases:
    - **`len(left) == k`**: exactly k items less than the pivot, so return the pivot.
    - **`len(left) > k`**: More than k items less than the pivot, so return the k^{th}-smallest element of the left half of the list.
    - **`len(left) < k`**: There are fewer than k items ≤ to the pivot, so return the $(k - len(left) - 1)^{st}$-smallest element of the right half of the list.
  - **Conclusion** The inductive hypothesis holds for all i. In particular, given an input list of any length n, **`select(A,k)`** correctly finds the k^{th}-smallest element!

Today's Outline

- Divide and Conquer II
  - ~~Substitution method~~ **Done!**
  - Linear-time selection
    - ~~Proving correctness~~ **Done!**
    - Proving runtime with recurrence relations
  - *Problems: selection*
  - *Algorithms: Select*
  - Reading: CLRS 9

Analyzing Runtime

- Writing a recurrence relation for **`select`** gives:

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- Writing a recurrence relation for **`select`** gives:

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k} \\ T(\texttt{len(left)}) + O(n) & \texttt{len(left) > k} \\ T(\texttt{len(right)}) + O(n) & \texttt{len(left) < k} \end{cases}$$

```python
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- Writing a recurrence relation for **`select`** gives:

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k} \\ T(\texttt{len(left)}) + O(n) & \texttt{len(left) > k} \\ T(\texttt{len(right)}) + O(n) & \texttt{len(left) < k} \end{cases}$$

The runtime for the recursive call to **`select`**

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- Writing a recurrence relation for **`select`** gives:

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k} \\ T(\texttt{len(left)}) + O(n) & \texttt{len(left) > k} \\ T(\texttt{len(right)}) + O(n) & \texttt{len(left) < k} \end{cases}$$

The runtime for the recursive call to **`select`**

The runtime to partition about the chosen pivot

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- Writing a recurrence relation for **`select`** gives:

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k} \\ T(\texttt{len(left)}) + O(n) & \texttt{len(left) > k} \\ T(\texttt{len(right)}) + O(n) & \texttt{len(left) < k} \end{cases}$$

The runtime for the recursive call to **`select`**

The runtime to partition about the chosen pivot

len(left) and len(right) depend on how we pick the pivot!

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime in an Ideal World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In an ideal world, we split the input exactly in half, such that: **`len(left)`** = **`len(right)`** = (n-1)/2.
  - Then we could use **Master Theorem**!
    - What's the recurrence?

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an Ideal World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
  - In an ideal world, we split the input exactly in half, such that: `len(left)` = `len(right)` = (n-1)/2.
  - Then we could use **Master Theorem**!
    - What's the recurrence? $T(n) \le T(n/2) + O(n)$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an Ideal World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
  - In an ideal world, we split the input exactly in half, such that: `len(left)` = `len(right)` = (n-1)/2.
  - Then we could use **Master Theorem**!
    - What's the recurrence? $T(n) \le T(n/2) + O(n)$
    - Then, $a = 1$, $b = 2$, $d = 1$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an Ideal World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In an ideal world, we split the input exactly in half, such that: **`len(left)`** = **`len(right)`** = (n-1)/2.
  - Then we could use **Master Theorem**!
    - What's the recurrence? $T(n) \leq T(n/2) + O(n)$
    - Then, $a = 1$, $b = 2$, $d = 1$ (Case 2: $a < b^d$)

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an Ideal World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In an ideal world, we split the input exactly in half, such that: **`len(left)`** = **`len(right)`** = (n-1)/2.
  - Then we could use **Master Theorem**!
    - What's the recurrence? $T(n) \le T(n/2) + O(n)$
    - Then, $a = 1$, $b = 2$, $d = 1$ (Case 2: $a < b^d$)
    - $T(n) \le O(n^d) = \mathbf{O(n)}$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - If we get super unlucky, we split the input, such that: **`len(left)`** = n - 1 and **`len(right)`** = 1 or vice versa.
  - Then it would be a lot slower.
    - $T(n) \leq T(n-1) + O(n)$
    - Then, $O(n)$ levels of $O(n)$
    - $T(n) \leq O(n^2)$

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

<u>“Worst-case”</u> runtime $\Theta(n^2)$

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

<u>"Worst-case"</u> runtime $\Theta(n^2)$

We discussed this runtime from earlier!

Analyzing Runtime

- Recall **`pivot = random.choice(A)`** i.e. we randomly chose the pivot.
  - It's *possible* to get unlucky, thus leading to runtime of $\Theta(n^2)$.
  - We'll formalize this unluckiness when we study Randomized Algs.
- How might we pick a better pivot?
  - After all, it's called **linear-time** selection, which implies $\Theta(n)$-time.

Analyzing Runtime

- Recall in an ideal world, we split the input exactly in half, such that: **`len(left)`** = **`len(right)`** = (n-1)/2.
- **Key Insight** The ideal world requires us to pick the pivot that divides the input list in half

Analyzing Runtime

- Recall in an ideal world, we split the input exactly in half, such that: **`len(left)`** = **`len(right)`** = (n-1)/2.
- **Key Insight** The ideal world requires us to pick the pivot that divides the input list in half aka **the median**

Analyzing Runtime

- Recall in an ideal world, we split the input exactly in half, such that: **`len(left)`** = **`len(right)`** = (n-1)/2.
- **Key Insight** The ideal world requires us to pick the pivot that divides the input list in half aka **the median** aka **`select(A, k=⌈n/2⌉-1)`**.
- To approximate the ideal world, the linear-time select algorithm picks the pivot that divides the input list **approximately** in half aka **close to the median**.

Analyzing Runtime in an ~~Ideal~~ Reasonable World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
  - In a reasonable world, we split the input roughly in half, such that: $3n/10 <$ `len(left)`, `len(right)` $< 7n/10$.
  - Once again, we could use **Master Theorem**!
    - What's the recurrence?

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an ~~Ideal~~ Reasonable World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In a reasonable world, we split the input roughly in half, such that: $3n/10 <$ **`len(left)`**, **`len(right)`** $< 7n/10$.
  - Once again, we could use **Master Theorem**!
    - What's the recurrence? $T(n) \leq T(7n/10) + O(n)$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an ~~Ideal~~ Reasonable World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In a reasonable world, we split the input roughly in half, such that: $3n/10 <$ **`len(left)`**, **`len(right)`** $< 7n/10$.
  - Once again, we could use **Master Theorem**!
    - What's the recurrence? $T(n) \le T(7n/10) + O(n)$
    - Then, $a = 1$, $b = 10/7$, $d = 1$

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an ~~Ideal~~ Reasonable World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
    - In a reasonable world, we split the input roughly in half, such that: $3n/10 <$ **`len(left)`**, **`len(right)`** $< 7n/10$.
    - Once again, we could use **Master Theorem**!
        - What's the recurrence? $T(n) \leq T(7n/10) + O(n)$
        - Then, $a = 1$, $b = 10/7$, $d = 1$ (Case 2: $a < b^d$)

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Analyzing Runtime in an ~~Ideal~~ Reasonable World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In a reasonable world, we split the input roughly in half, such that: $3n/10 <$ **`len(left)`**, **`len(right)`** $< 7n/10$.
  - Once again, we could use **Master Theorem**!
    - What's the recurrence? $T(n) \leq T(7n/10) + O(n)$
    - Then, $a = 1$, $b = 10/7$, $d = 1$ (Case 2: $a < b^d$)
    - $T(n) \leq O(n^d) =$ **$O(n)$**

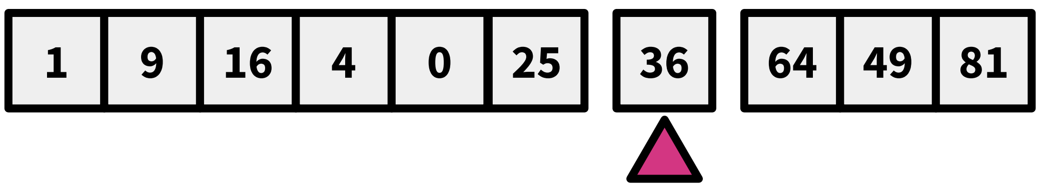
Suppose $T(n) = a \cdot T(n/b) + O(n^d)$. The Master method states:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log\_b(a)}) & \text{if } a > b^d \end{cases}$$

Reasonable

Analyzing Runtime in an ~~Ideal~~ World

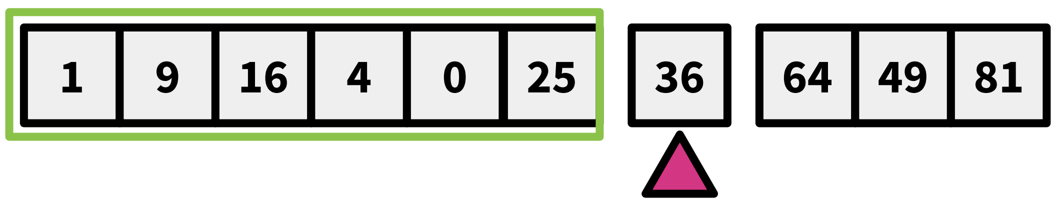
- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
    - In a reasonable world, we split the input roughly in half, such that: 3n/10 < **`len(left)`**, **`len(right)`** < 7n/10.



The goal is to pick a pivot such that

Analyzing Runtime in an ~~Ideal~~ Reasonable World

- `len(left)` and `len(right)` determine the runtime of the recursive calls to `select`.
  - In a reasonable world, we split the input roughly in half, such that: 3n/10 < `len(left)`, `len(right)` < 7n/10.

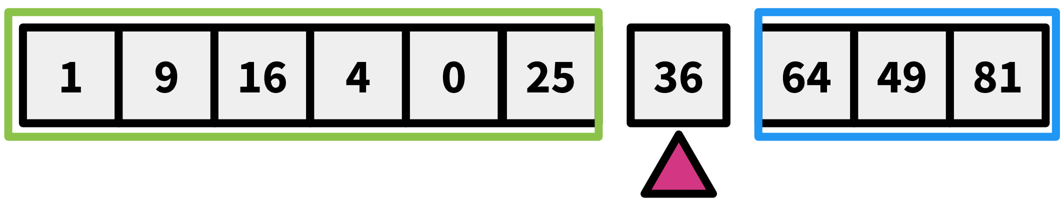


The goal is to pick a pivot such that

3n/10 < `len(left)` < 7n/10

Analyzing Runtime in an ~~Ideal~~ Reasonable World

- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In a reasonable world, we split the input roughly in half, such that: 3n/10 < **`len(left)`**, **`len(right)`** < 7n/10.



The goal is to pick a pivot such that

3n/10 < **`len(left)`** < 7n/10 and 3n/10 < **`len(right)`** < 7n/10

Another Divide and Conquer Algorithm

- We can't solve **`select(A,n/2)`** (yet).
- But we can solve **`select(B,m/2)`** for **`len(B)`** = m < n.
- How does having an algorithm that can find the median of smaller lists help us?

Another Divide and Conquer Algorithm

- We can’t solve **`select(A,n/2)`** (yet).
- But we can solve **`select(B,m/2)`** for **`len(B)`** = m < n.
- How does having an algorithm that can find the median of smaller lists help us?

Pro tip: making the inductive hypothesis i.e. assuming correctness of the algorithm on smaller inputs is a helpful technique for designing divide and conquer algorithms.

Another Divide and Conquer Algorithm

- We can't solve `select(A,n/2)` (yet).
- But we can solve `select(B,m/2)` for `len(B)` = m < n.
- How does having an algorithm that can find the median of smaller lists help us? **It can help us pick a pivot that's close to the median.**

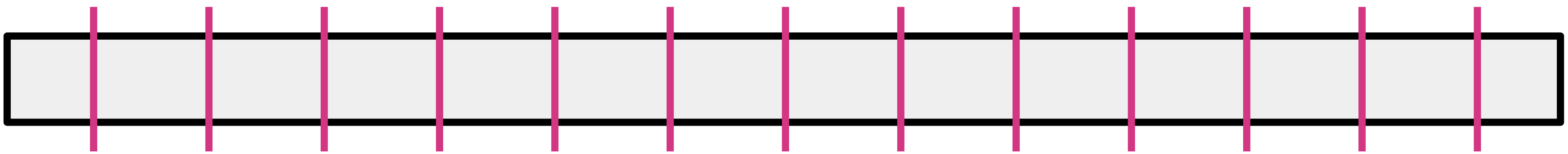
Pro tip: making the inductive hypothesis i.e. assuming correctness of the algorithm on smaller inputs is a helpful technique for designing divide and conquer algorithms.

Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.

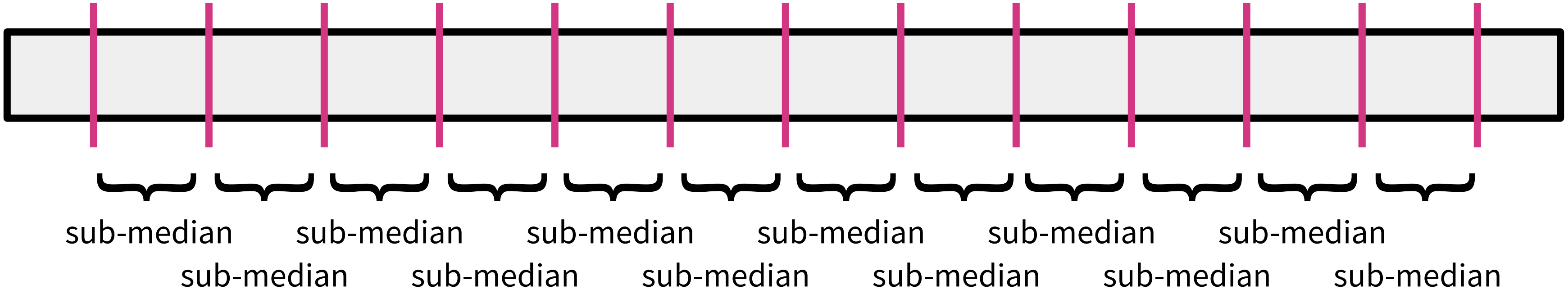
Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
    - Divide the original list into small groups.



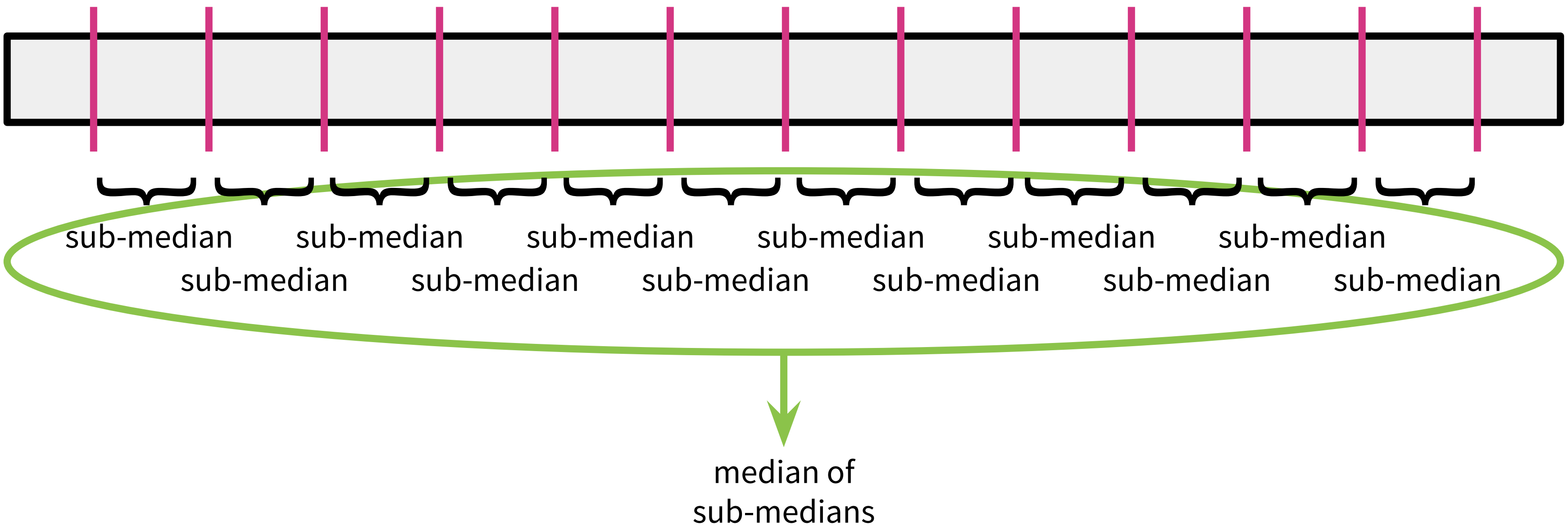
Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
  - Divide the original list into small groups.
  - Find the sub-median of each small group.



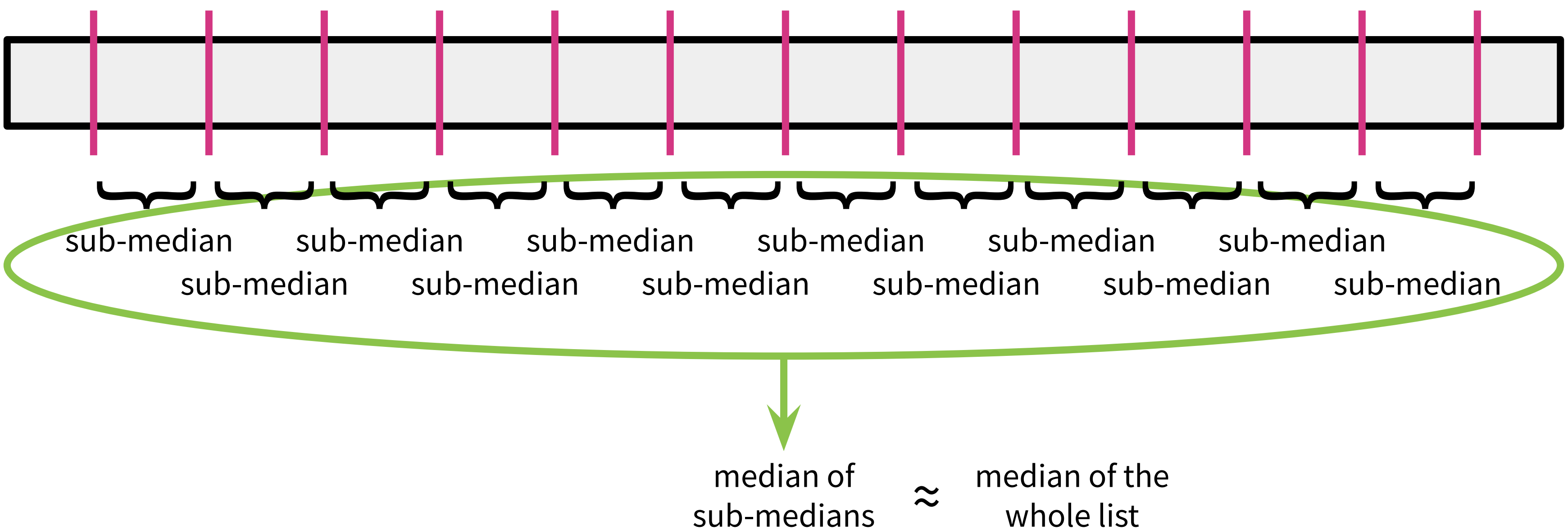
Another Divide and Conquer Algorithm

- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
  - Divide the original list into small groups.
  - Find the sub-median of each small group.
  - Find the median of all of the sub-medians.



Another Divide and Conquer Algorithm

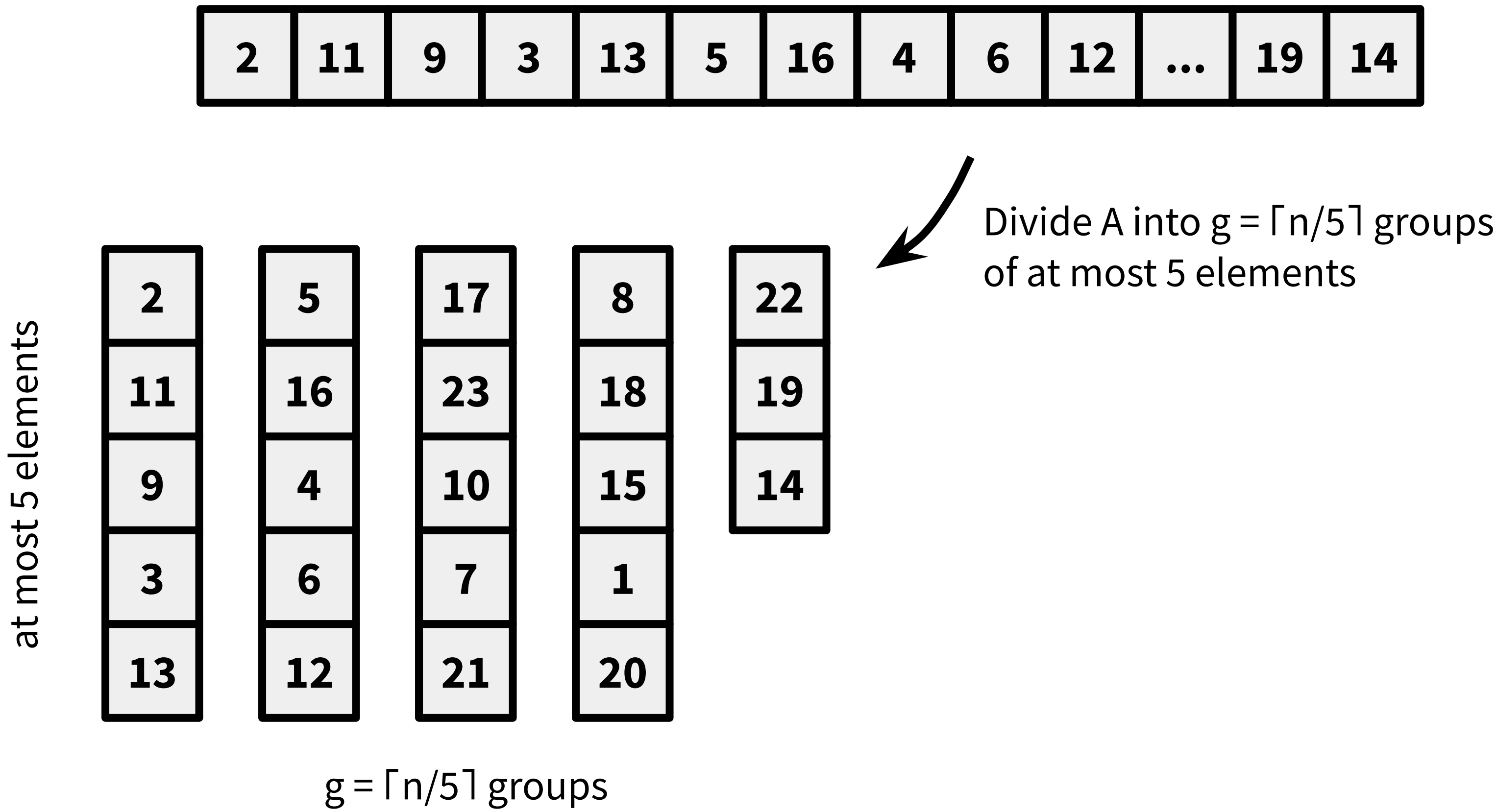
- **Goal:** Use an algorithm that can find the median of smaller lists to help pick a pivot that's close to the median of the original list.
  - Divide the original list into small groups.
  - Find the sub-median of each small group.
  - Find the median of all of the sub-medians.



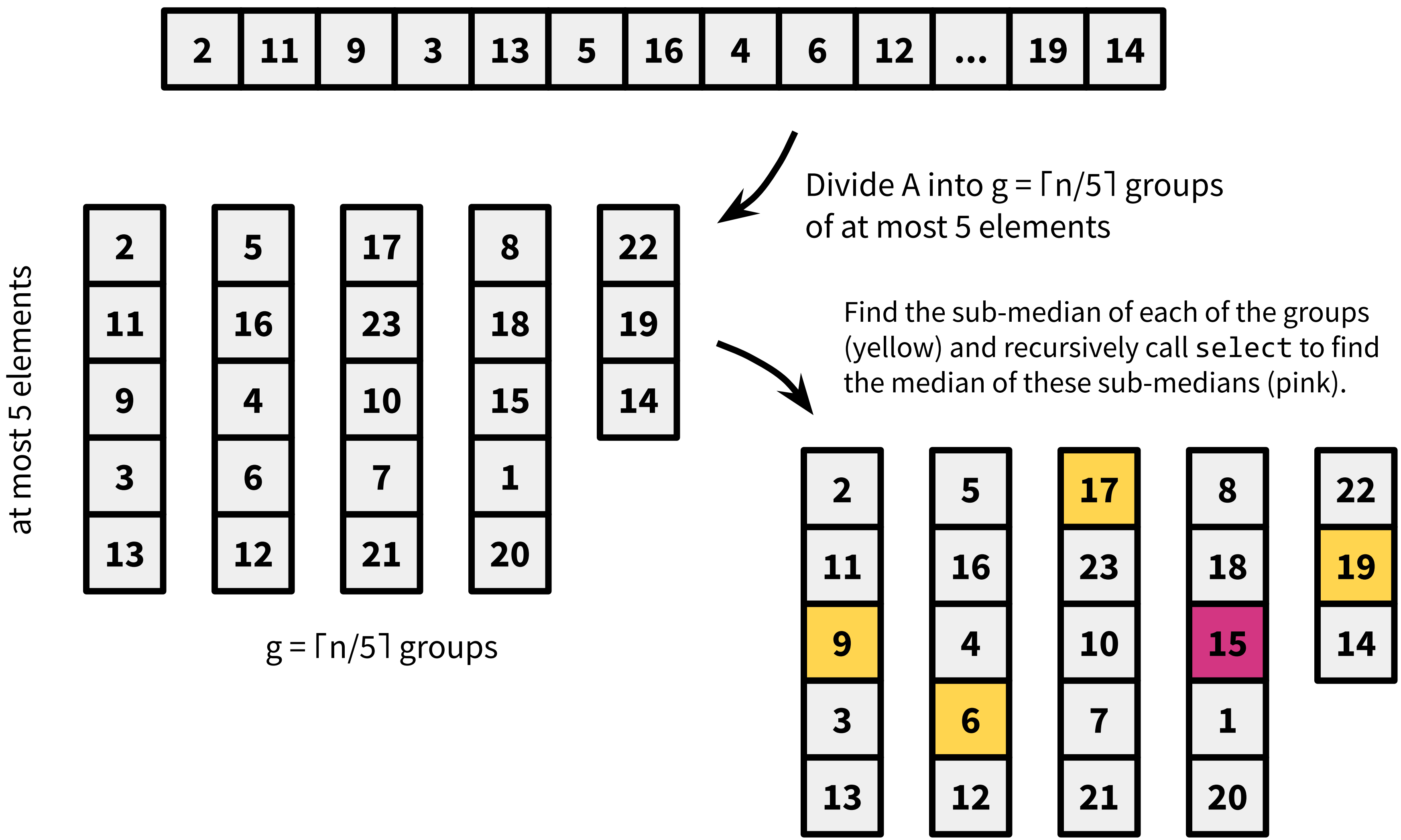
Another Divide and Conquer Algorithm

2	11	9	3	13	5	16	4	6	12	...	19	14

Another Divide and Conquer Algorithm



Another Divide and Conquer Algorithm



Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

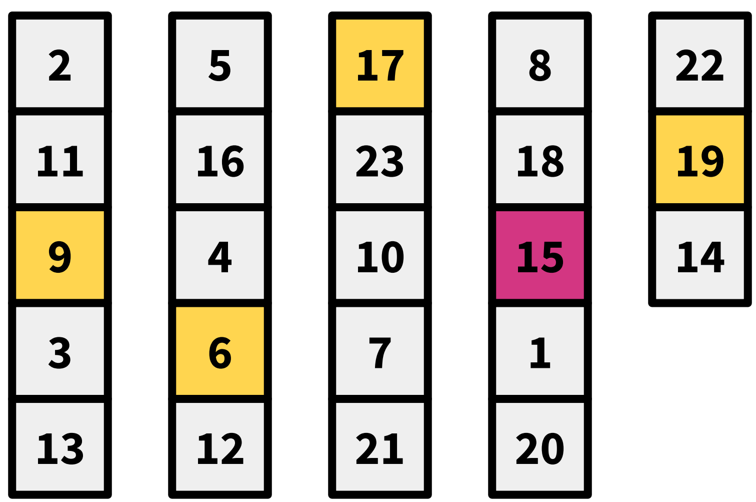
"Worst-case" runtime $\Theta(n^2)$

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A) median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

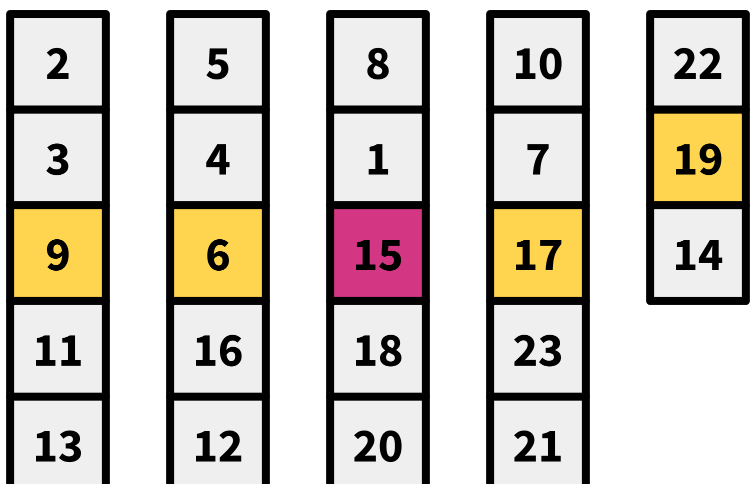
Analyzing Runtime

- Clearly, the median of medians (15) is not necessarily the actual median (12), but we claim that it's guaranteed to be pretty close.



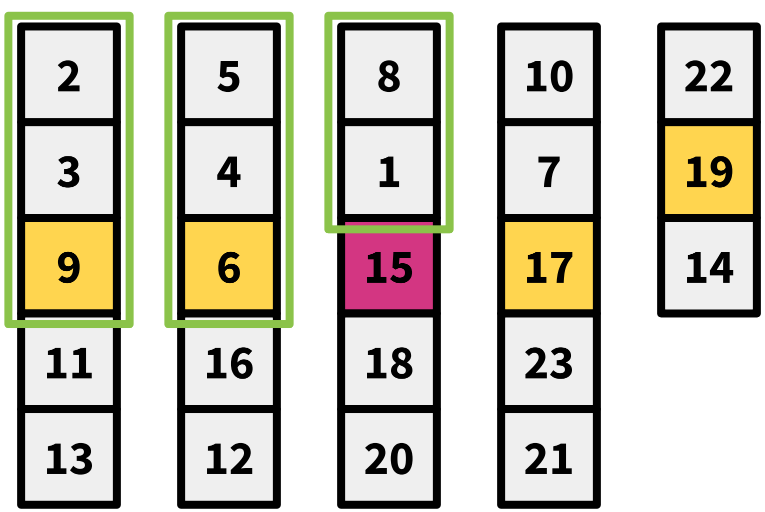
Analyzing Runtime

- To see why, partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.
  - **<u>At least</u>** how many elements are guaranteed to be **<u>smaller</u>** than the median of medians?



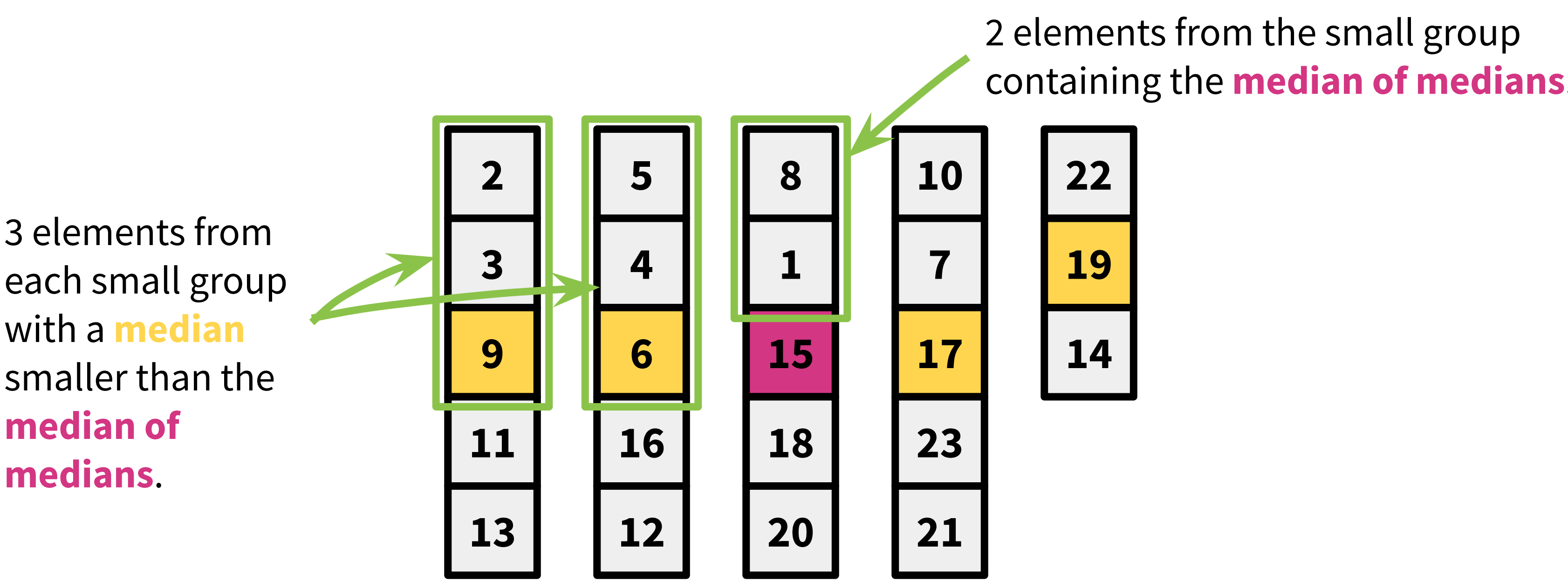
Analyzing Runtime

- To see why, partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.
  - **At least** how many elements are guaranteed to be **smaller** than the median of medians? **At least** **these (1, 2, 3, 4, 5, 6, 8, 9)**. There might be more (7, 11, 12, 13, 14), but we are *guaranteed* that at least these will be smaller.



Analyzing Runtime

- To see why, partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.
  - **<u>At least</u>** how many elements are guaranteed to be **<u>smaller</u>** than the median of medians? **<u>At least</u>** **these (1, 2, 3, 4, 5, 6, 8, 9)**. There might be more (7, 11, 12, 13, 14), but we are *guaranteed* that at least these will be smaller.



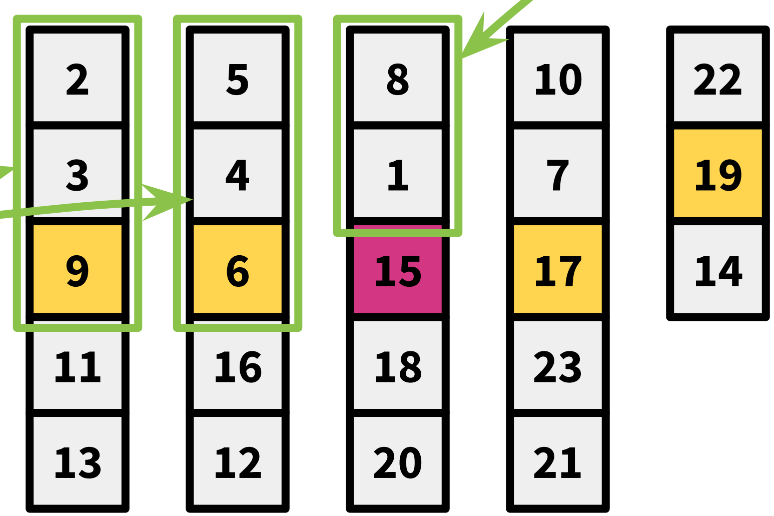
Analyzing Runtime

- As a function of n (the size of the original list), how many elements are guaranteed to be **smaller** than the median of medians?
  - Let $g = \lceil n/5 \rceil$ represent the number of groups.
  - **At least** $3 \cdot (\lceil g/2 \rceil - 1 - 1) + 2$ elements.

To exclude the list with the **median of medians**.

2 elements from the small group containing the **median of medians**.

3 elements from each small group with a **median** smaller than the **median of medians**.



Analyzing Runtime

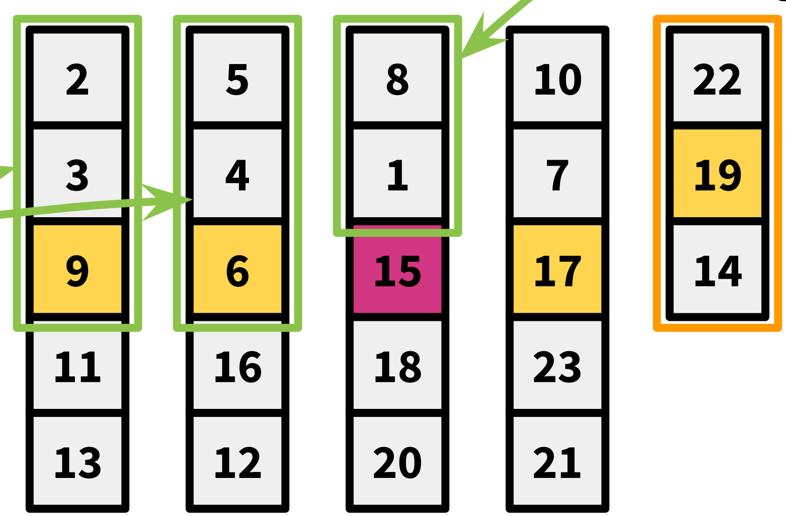
- As a function of n (the size of the original list), how many elements are guaranteed to be **smaller** than the median of medians?
  - Let $g = \lceil n/5 \rceil$ represent the number of groups.
  - **At least** $3 \cdot (\lceil g/2 \rceil - 1 - 1) + 2$ elements.

To exclude the list with the **median of medians**.

To exclude the list with the **leftovers**.

2 elements from the small group containing the **median of medians**.

3 elements from each small group with a **median** smaller than the **median of medians**.

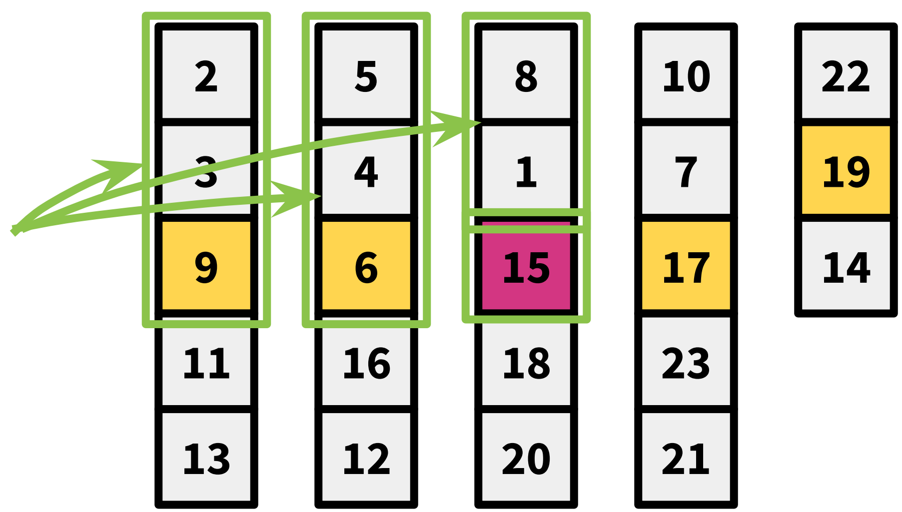


Analyzing Runtime

- If **at least** $3 \cdot (\lceil g/2 \rceil - 2) + 2$ elements are guaranteed to be **smaller** than the median of medians, **at most** how many elements are **larger** than the median of medians?
  - **At most** $n - 1 - (3 \cdot (\lceil g/2 \rceil - 2) + 2)$

n-1 is for all of the elements except for the median of medians.

At most everything besides these elements

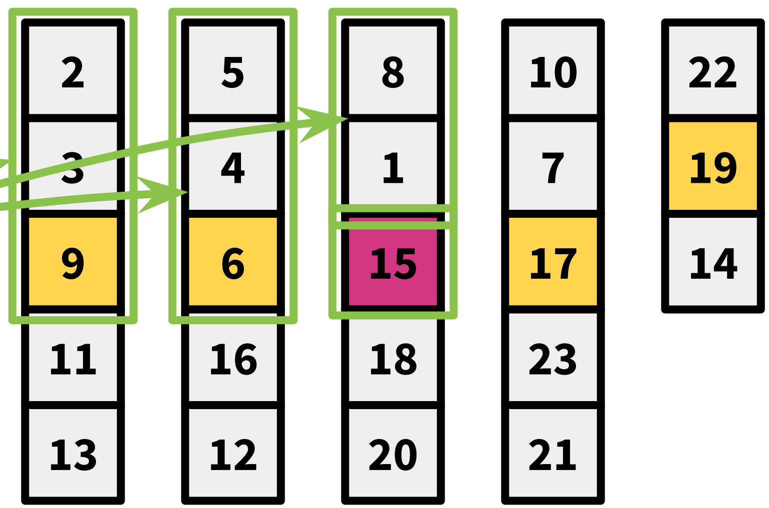


Analyzing Runtime

- If **at least** 3 · (⌈g/2⌉ - 2) + 2 elements are guaranteed to be **smaller** than the median of medians, **at most** how many elements are **larger** than the median of medians?
  - **At most** n - 1 - (3 · (⌈g/2⌉ - 2) + 2) ≤ **7n/10 + 3** elements.

n-1 is for all of the elements except for the median of medians.

At most everything besides these elements



Analyzing Runtime

- We just showed that …

median_of_medians will choose a pivot greater than at least $3 \cdot (\lceil g/2 \rceil - 2) + 2 \geq 3n/10 - 4$ elements.

$3n/10 - 4 \leq$ `len(left)`

`len(right)` $\leq 7n/10 + 3$

median_of_medians will choose a pivot less than at most $7n/10 + 3$ elements.

Analyzing Runtime

- We just showed that …

`median_of_medians` will choose a pivot greater than at least $3 \cdot (\lceil g/2 \rceil - 2) + 2 \geq 3n/10 - 4$ elements.

$$3n/10 - 4 \leq \texttt{len(left)} \leq 7n/10 + 3$$

$$3n/10 - 4 \leq \texttt{len(right)} \leq 7n/10 + 3$$

`median_of_medians` will choose a pivot less than at most $7n/10 + 3$ elements.

- We can just as easily show the inverse.

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = ~~random.choice(A)~~ median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
  - $T(n) = n\log(n)$ when $n \leq 100$
  - $T(n) \leq$ **$T(n/5)$** +

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
  - $T(n) = n\log(n)$ when $n \leq 100$
  - $T(n) \leq T(n/5) +$ **$T(7n/10)$** $+$

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
  - $T(n) = n\log(n)$ when $n \leq 100$
  - $T(n) \leq T(n/5) + T(7n/10) +$ **$O(n)$**

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
  - $T(n) = n\log(n)$ when $n \leq 100$
  - $T(n) \leq T(n/5) + T(7n/10) + O(n)$

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
  - $T(n) = n\log(n)$ when $n \leq 100$
  - $T(n) \leq T(n/5) + T(7n/10) + O(n)$
  - We can't use **Master Theorem**!

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Analyzing Runtime

- What's the recurrence relation?
  - $T(n) = n\log(n)$ when $n \leq 100$
  - $T(n) \leq T(n/5) + T(7n/10) + O(n)$
  - We can't use **Master Theorem**!
  - We use **substitution method**!

```python
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k: return pivot
  elif len(left) > k: return select(left, k, c)
  else: return select(right, k-len(left)-1, c)
```

Substitution Method

$T(n) = n\log(n)$ when $n \leq 100$

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

1. **Guess what the answer is.**
    - **Linear-time** select
    - Comparing to mergesort recurence, less than n log(n)

→ **Guess O(n)**

Substitution Method

$$T(n) = n\log(n) \text{ when } n \le 100$$

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

2. Formally prove that's what the answer is.

- **Inductive hypothesis** $T(k) \le Ck$ for all $1 \le k < n$.
- **Base case** $T(k) \le Ck$ for all $k \le 100$.
- **Inductive step**
  - $$\begin{aligned} T(n) &= T(n/5) + T(7n/10) + dn \\ &\le C(n/5) + C(7n/10) + dn \\ &= (C/5)n + (7C/10)n + dn \\ &\le Cn \end{aligned}$$

C is some constant we'll have to fill in later!

C must be ≥ log(n) for n ≤ 100, so C ≥ 7.

Solve for C to satisfy the inequality. C ≥ 10d works.

- **Conclusion** There exists some $C = \max\{7, 10d\}$ such that for all $n > 1$, $T(n) \le Cn$. Therefore, $T(n) = O(n)$.

Substitution Method

$T(n) = n\log(n)$ when $n \le 100$

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

2. **Formally prove that's what the answer is.**
   - **Inductive hypothesis** $T(k) \le$ **max{7, 10d}**k for all $1 \le k < n$.
   - **Base case** $T(k) \le$ **max{7, 10d}**k for all $k \le 100$.
   - **Inductive step**
     - $T(n) = T(n/5) + T(7n/10) + dn$
       $\le \max\{7, 10d\}(n/5) + \max\{7, 10d\}(7n/10) + dn$
       $= (\max\{7, 10d\}/5)n + (7\max\{7, 10d\}/10)n + dn$
       $\le \max\{7, 10d\}n$
   - **Conclusion** There exists some **C** = max{7, 10d} such that for all $n > 1$, $T(n) \le$ **max{7, 10d}**n. Therefore, $T(n) = O(n)$.

Substitution Method

1. Guess what the answer is.
2. Formally prove that's what the answer is.
   - Might need to leave some constants unspecified until the end and see what they need to be for the proof to work.

Today's Outline

- Divide and Conquer II
  - ~~Substitution method~~ **Done!**
  - Linear-time selection
    - ~~Proving correctness~~ **Done!**
    - ~~Proving runtime with recurrence relations~~ **Done!**
  - *Problems: selection*
  - *Algorithms: Select*
  - Reading: CLRS 9

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A) median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

<u>Worst-case</u> runtime $\Theta(n)$

Linear-Time Selection

```
def select(A, k, c=100):
  if len(A) <= c:
    return naive_select(A, k)
  pivot = random.choice(A) median_of_medians(A)
  left, right = partition_about_pivot(A, pivot)
  if len(left) == k:
    # The pivot is the kth smallest element!
    return pivot
  elif len(left) > k:
    # The kth smallest element is left of the pivot
    return select(left, k, c)
  else:
    # The kth smallest element is right of the pivot
    return select(right, k-len(left)-1, c)
```

Note: back to talking about the same worst-case we saw for insertion sort (we'll revisit during Randomized Algs).

Worst-case runtime $\Theta(n)$