Introduction to Algorithms

L3. Divide & Conquer

Instructor : Kilho Lee

Course Overview

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

Today's Outline

- Comparison sorting algorithms: selection sort, bubble sort
- Divide and Conquer I
    - Proving correctness with induction
    - Proving runtime with **recurrence relations**
        - How do we measure the runtime of a recursive algorithm?
    - *Problems: Comparison-sorting*
    - *Algorithms: Mergesort*
    - Reading: CLRS 2.3, 4.3-4.6

Comparison sortings

Overview

- Insertion sort (삽입정렬)
  - It finds a proper (sorted) position, and insert an element into there.

- Selection sort (선택정렬)
  - It picks a min/max element from the unsorted list, and put the element into the front/end of the sorted list.

- Bubble sort (버블정렬)
  - It pushes higher values to the end of the list. The highest element in the list will float toward the end of the list.

- Today, we will briefly cover the concept/intuition of them.

Selection sort

- Selection sort
  - It picks a min/max element from the unsorted list, and put the element into the front/end of the sorted list.

20	12	10	15	2
2	20	12	10	15
2	10	20	12	15
2	10	12	20	15
2	10	12	15	20

Bubble sort

- Bubble sort
  - It pushes higher values to the end of the list. The highest element in the list will float toward the end of the list.

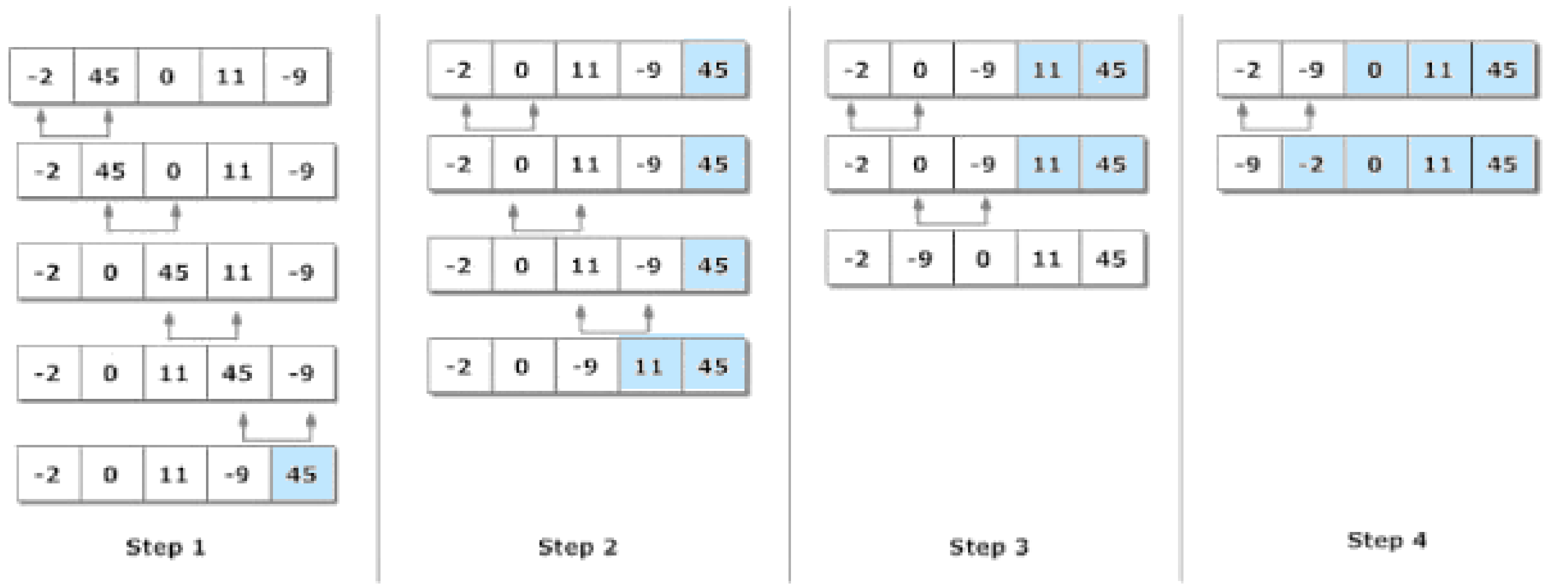


Figure: Working of Bubble sort algorithm

Correctness and Efficiency

- Identical to the insertion sort case:
  - Loop invariants
  - Count the number of operations and conduct the asymptotic analysis
  - *Leave it as homework*

Mergesort

Recall

- Insertion sort
  - **Does this actually work? Yes!**
    - We talked about loop invariants and proofs by induction.
  - **Is it fast? Eh, nah.**
    - We talked about worst-case, best-case, and average case analysis.
    - We talked about Big-O, Big-**Ω** and Big-**Θ** notation to describe upper-bounds, lower-bounds, and tight-bounds.
    - **Upper-bound for <u>worst-case</u> runtime $O(n^2)$**
    - **Lower-bound for <u>best-case</u> runtime $\Omega(n)$**

Another way of thinking

- Can we do better than insertion sort?
- Mergesort uses divide-and-conquer.
  - **Does this actually work?**
    - We will revisit proofs by induction.
  - **Is it fast?**
    - We will talk about recurrence relations.

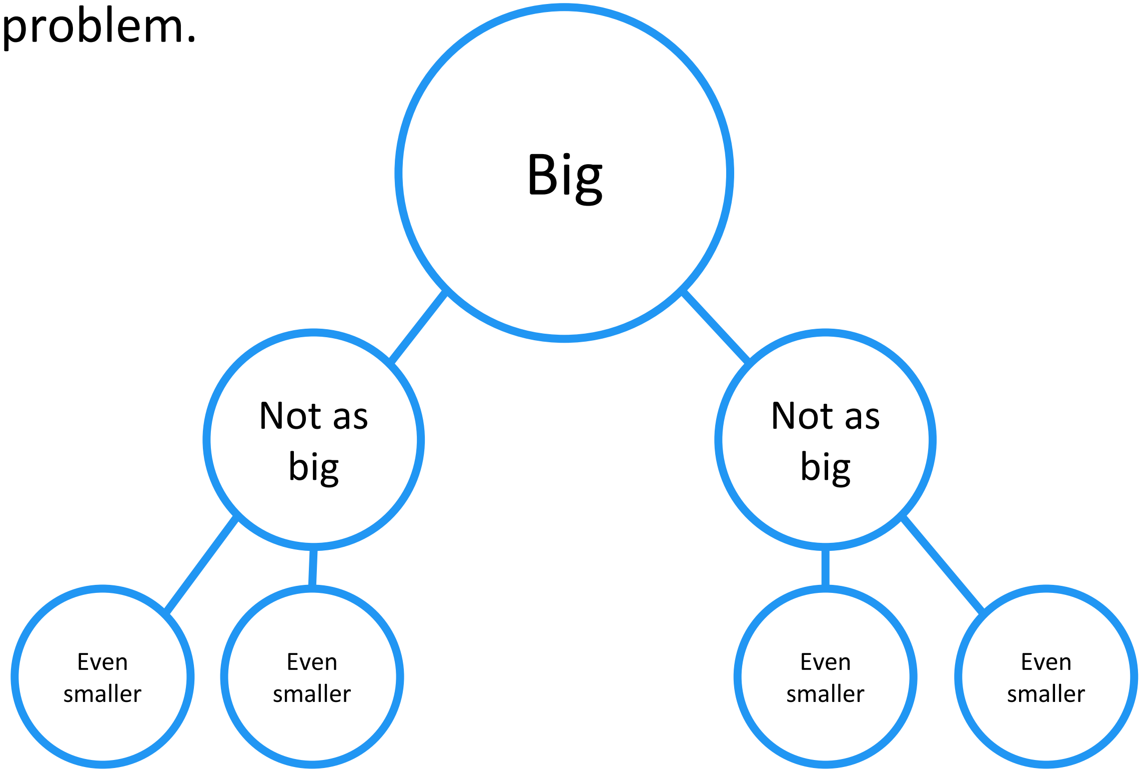
Another way of thinking

- Can we do better than insertion sort?
- Mergesort uses divide-and-conquer.
  - **Does this actually work?**
    - We will revisit proofs by induction.
  - **Is it fast?**
    - We will talk about recurrence relations.

These are the same questions we asked about insertion sort!

Divide and Conquer

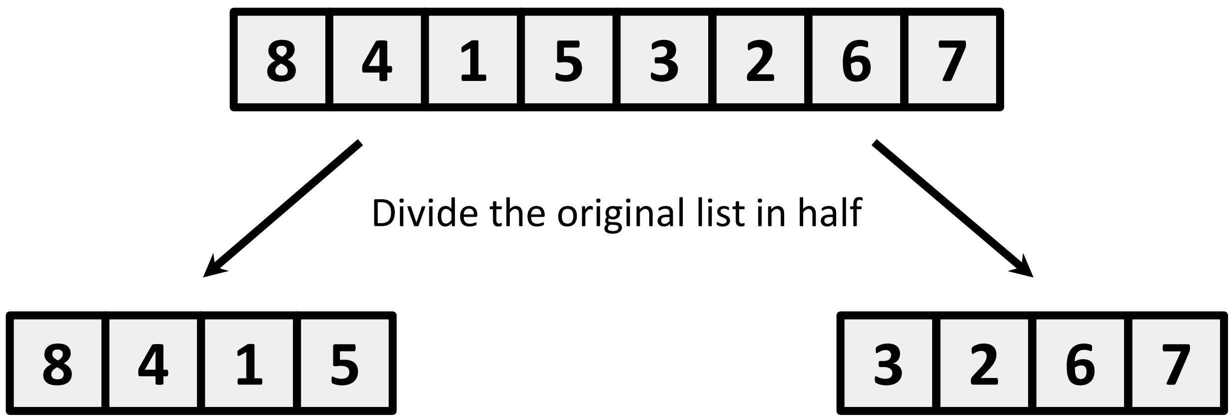
- 분할정복법
- **A soft of algorithmic paradigms**
- **Divide** Break the current problem into smaller (easier) sub-problems.
- **Conquer** Solve the smaller problems and collect the results to solve the current problem.



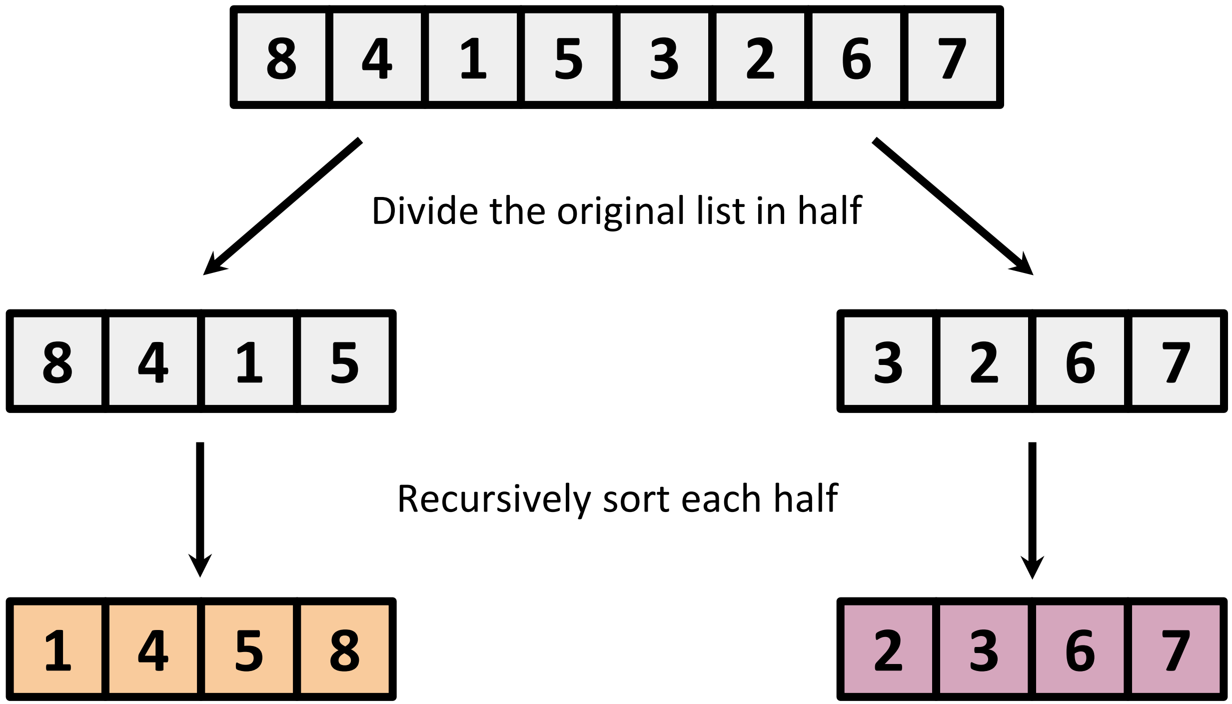
Mergesort (병합정렬)

8	4	1	5	3	2	6	7

Mergesort

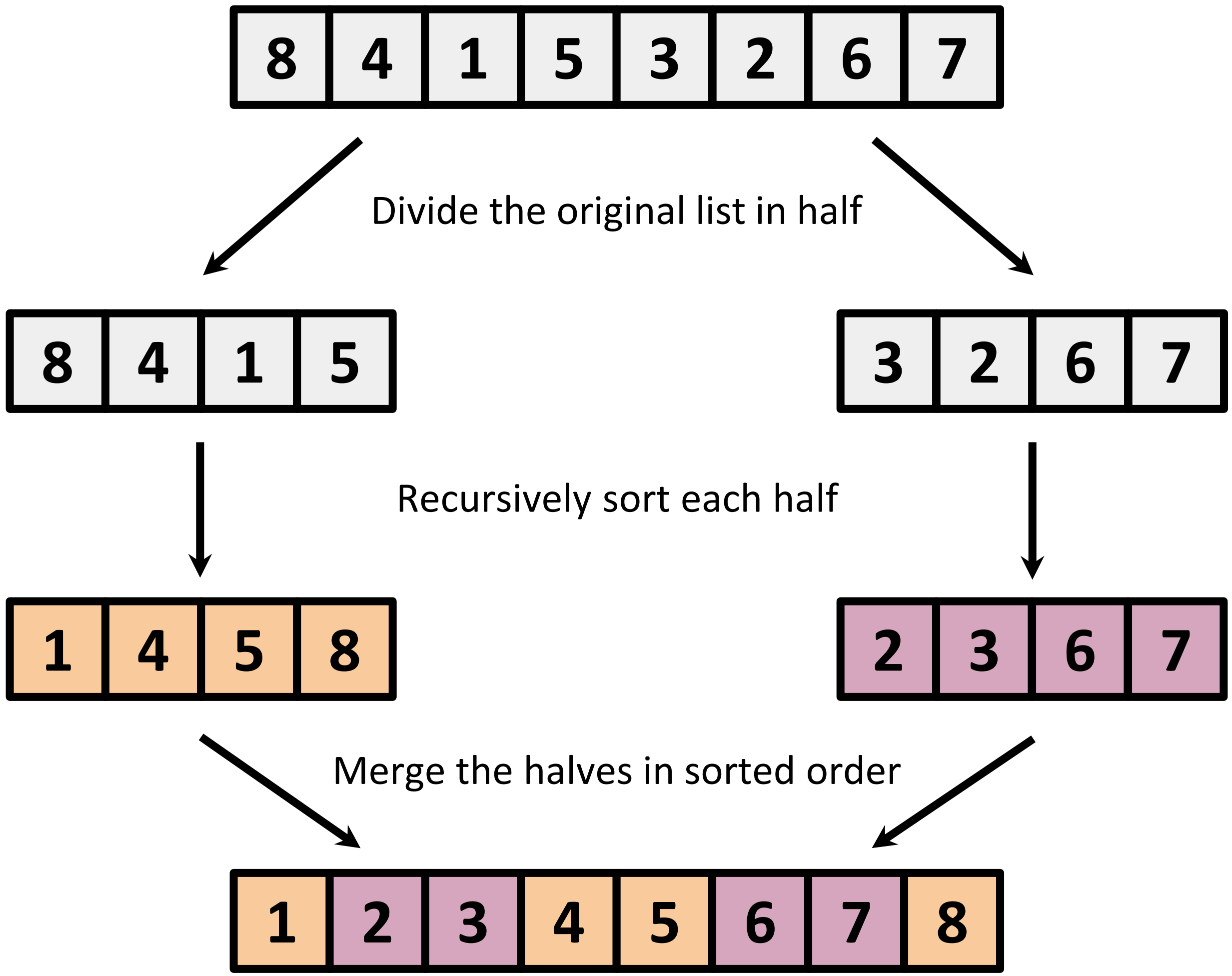


Mergesort



8 4 1 5 3 2 6 7
Divide the original list in half
8 4 1 5
3 2 6 7
Recursively sort each half
1 4 5 8
2 3 6 7

Mergesort


8 4 1 5 3 2 6 7
Divide the original list in half
8 4 1 5
3 2 6 7
Recursively sort each half
1 4 5 8
2 3 6 7
Merge the halves in sorted order
1 2 3 4 5 6 7 8

Mergesort

```
/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Mergesort

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray

    /* ...... */
```

Mergesort

```
/* … Continued … */
   while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    /* Copy the remaining elements of L[], if there  are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

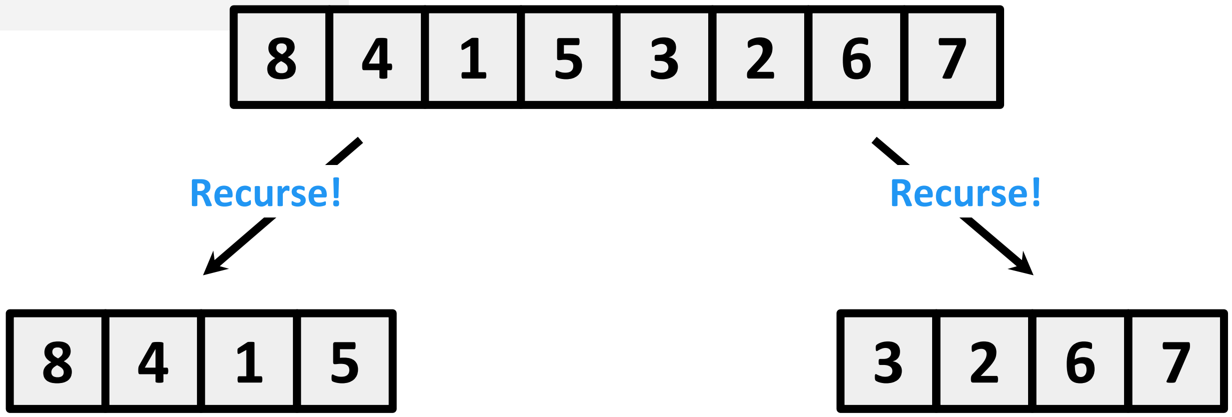
```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
            int m = (l+r)/2;
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

          merge(arr, l, m, r);
    }
}
```

Mergesort

8	4	1	5	3	2	6	7

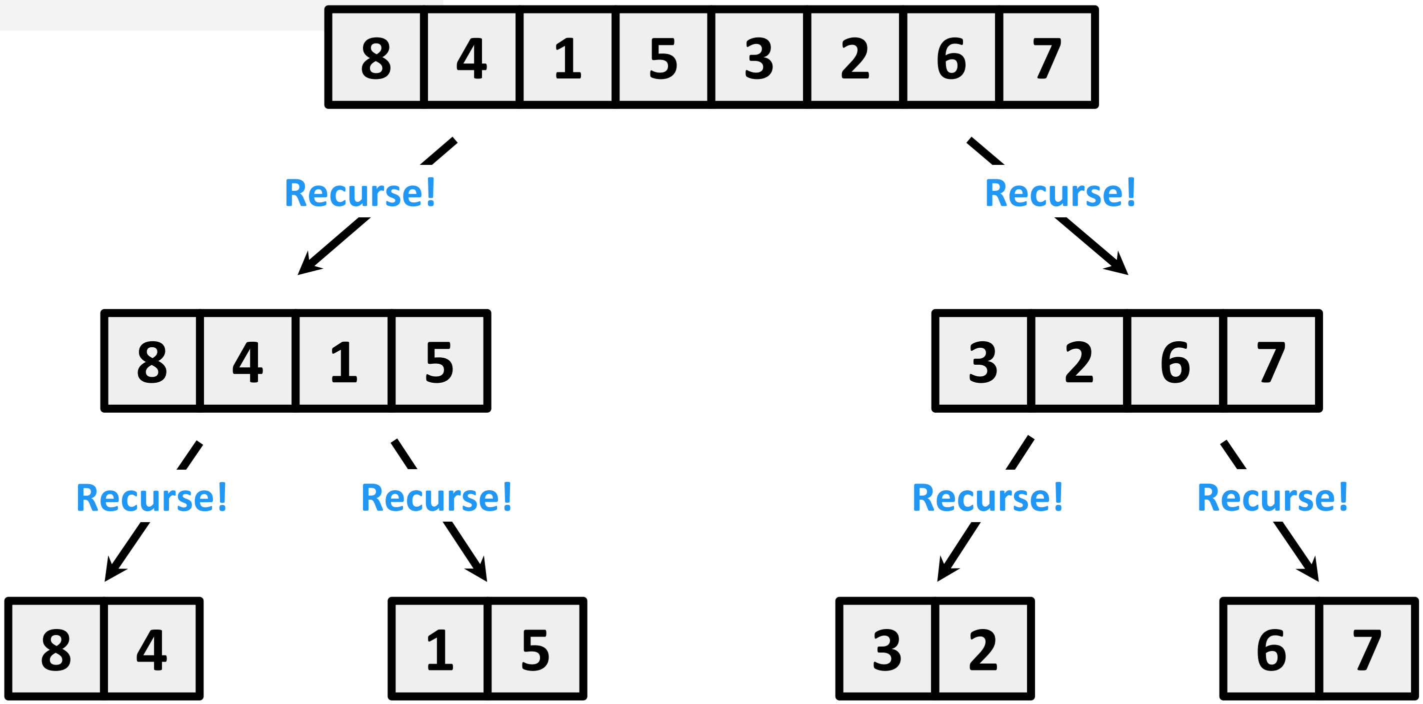
```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
            int m = (l+r)/2;
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

          merge(arr, l, m, r);
    }
}
```

Mergesort



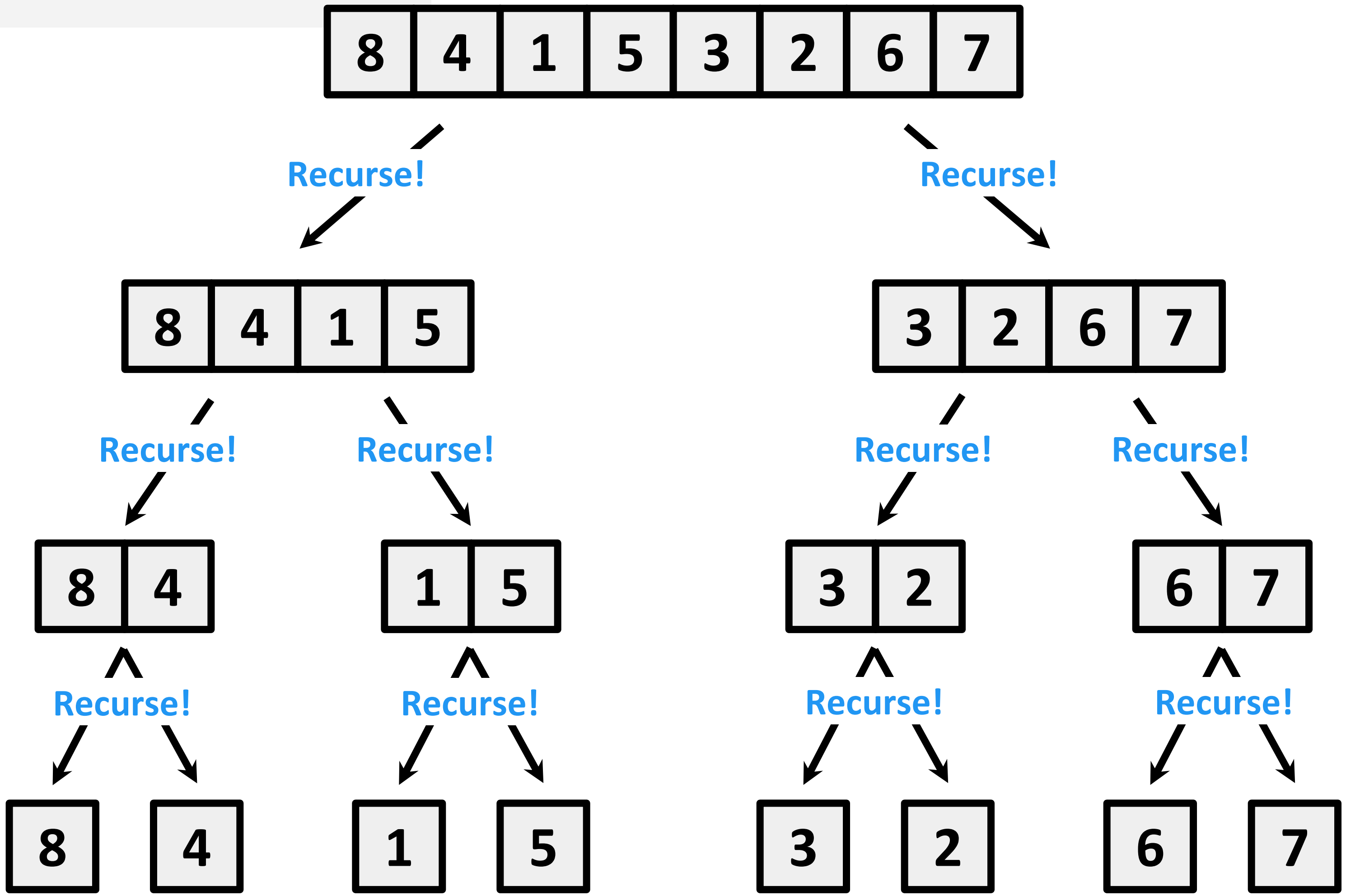
```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
            int m = (l+r)/2;
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

          merge(arr, l, m, r);
    }
}
```

Mergesort



```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
            int m = (l+r)/2;
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

         merge(arr, l, m, r);
    }
}
```

Mergesort



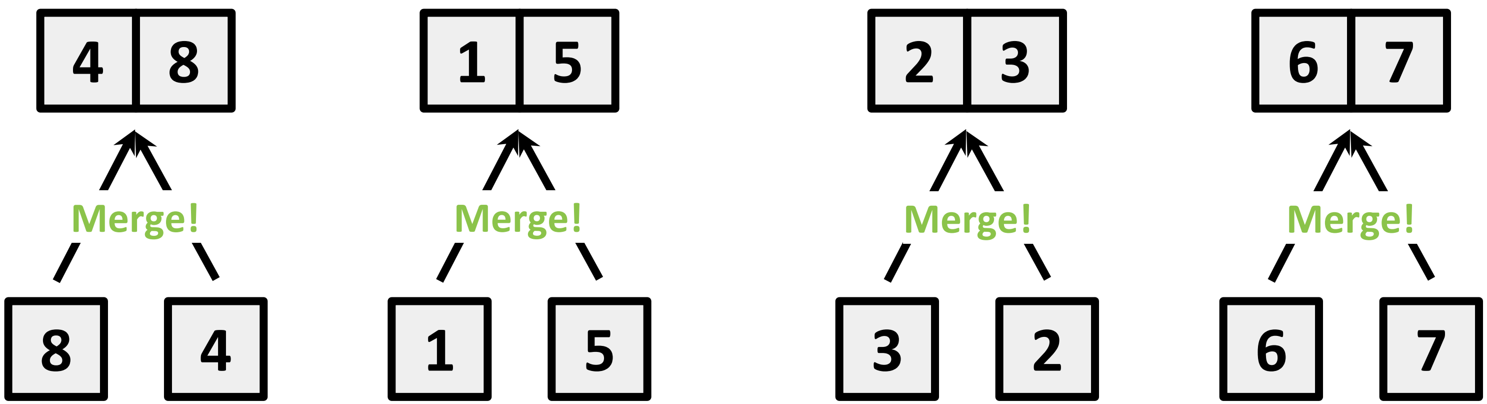
Mergesort

```
void merge(int arr[], int l, int m, int r)
{
   while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

8	4	1	5	3	2	6	7

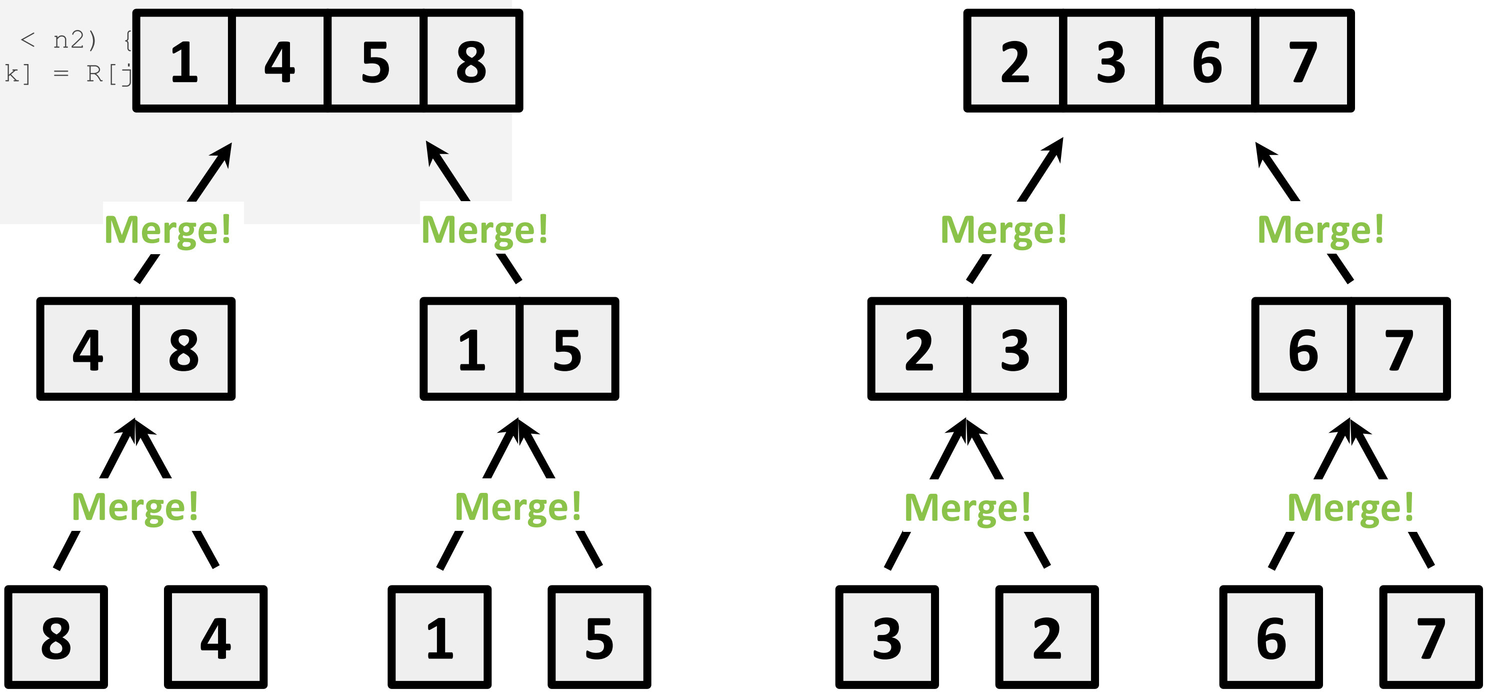
Mergesort

```
void merge(int arr[], int l, int m, int r)
{
   while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

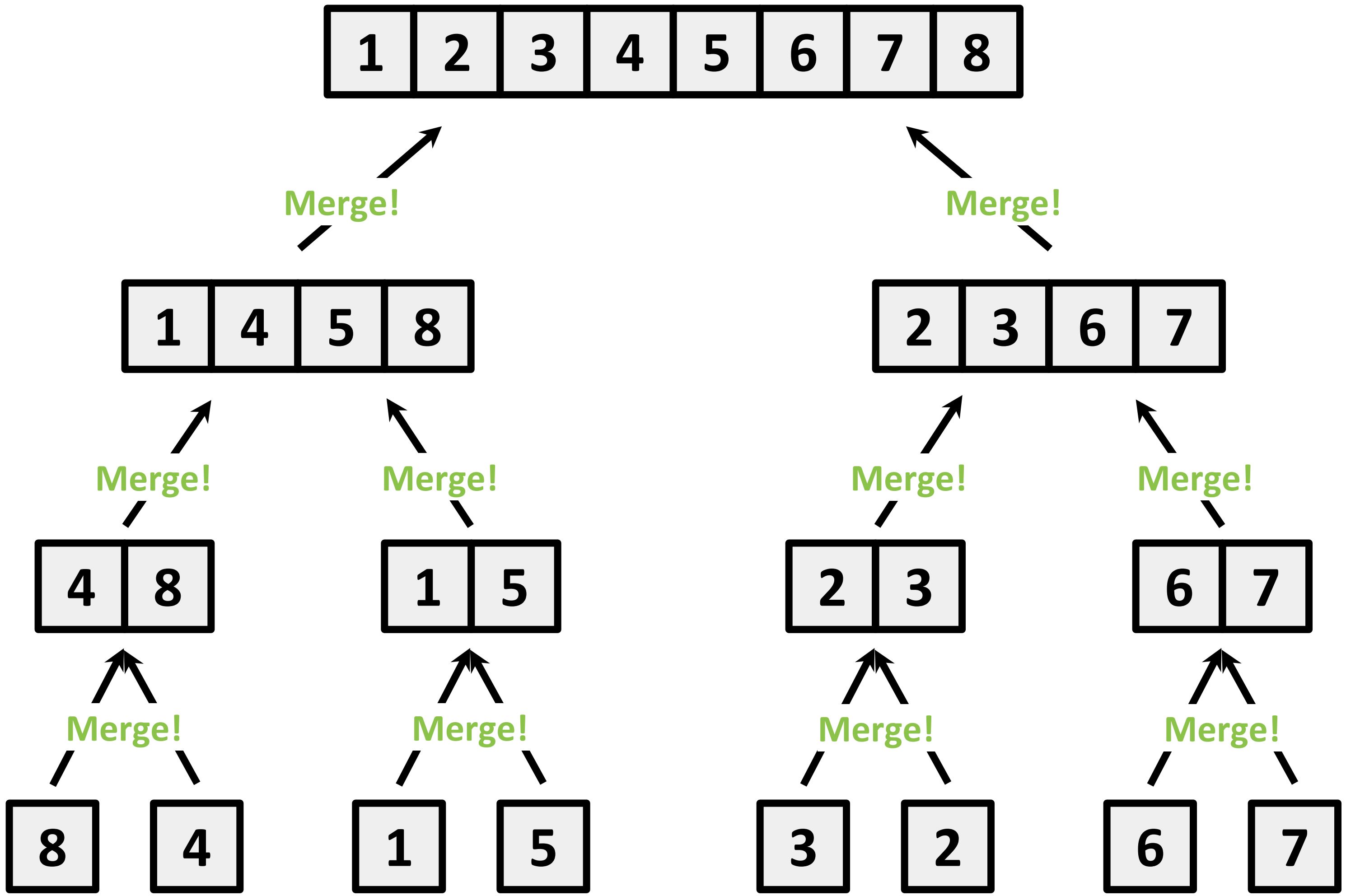


Mergesort

```
void merge(int arr[], int l, int m, int r)
{
   while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j
        j++;
        k++;
    }
}
```



Mergesort


1 2 3 4 5 6 7 8
Merge!
Merge!
1 4 5 8
2 3 6 7
Merge!
Merge!
Merge!
Merge!
4 8
1 5
2 3
6 7
Merge!
Merge!
Merge!
Merge!
8
4
1
5
3
2
6
7

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point…
  1. **Does this actually work?**
  2. **Is it fast?**

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point…
  1. **Does this actually work?**
  2. **Is it fast?**

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
         int m = (l+r)/2;
         mergeSort(arr, l, m);
         mergeSort(arr, m + 1, r);

         merge(arr, l, m, r);
    }
}
```

Mergesort

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

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
         int m = (l+r)/2;
         mergeSort(arr, l, m);
         mergeSort(arr, m + 1, r);

         merge(arr, l, m, r);
    }
}
```

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 `mergesort`...
  - **Inductive Hypothesis** `mergesort` correctly sorts input lists of length i. (In every recursive call, `mergesort` returns a sorted list.)
  - **Base case** (i=1) `mergesort` correctly sorts input lists of length 1; it returns a 1-element list, which is trivially sorted.
  - **Inductive step** Suppose the algorithm works on input lists of length 1 to i. Calling `mergesort` on an input list of length i+1 recursively calls `mergesort` on the left and right halves, which have lengths between 1 and i; therefore, `left` and `right` contain the elements originally in the left and right halves of the list, but sorted. Given two sorted lists, `merge` returns a single sorted list with all of the elements from the original two lists.
  - **Conclusion** The inductive hypothesis holds for all i. In particular, given an input list of any length n, mergesort returns a sorted version of that list!

Proving Correctness

- Formally, for **`mergesort`**...
  - **Inductive Hypothesis `mergesort`** correctly sorts input lists of length i.
    (길이가 1 to i 인 정렬된 리스트를 활용해 길이가 i+1인 리스트를 정렬할 수 있을 것이다)
  - **Base case** (i=1) **`mergesort`** 는 길이가 1인 리스트를 정렬한다. 길이가 1인 리스트는 자체적으로 정렬되어있음이 자명하다
  - **Inductive step** mergesort가 길이가 1 부터 i인 모든 리스트를 정렬할 수 있다고 하자. 이 때, 길이가 i+1인 리스트에 대해 mergesort 를 실행하면, mergesort는 좌측 부분과 우측 부분의 부분 리스트를 매개변수로하는 merge sort를 재귀호출하고, 이 때 좌측/우측 부분 리스트의 길이는 1부터 i사이이다. 따라서, 앞선 가정에 따라 좌측 부분과 우측 부분 리스트는 정렬되어 있는 상태이며, 정렬된 두 리스트에 대하여 merge 함수는 항상 정렬된 리스트를 만들어낸다. 곧, 길이가 i+1인 리스트에 대해서도 mergesort는 정렬된 리스트를 만든다.
  - **Conclusion** The inductive hypothesis holds for all i. In particular, given an input list of any length n, mergesort returns a sorted version of that list!

Proving Correctness

- Formally, for **`mergesort`**...
  - **Inductive Hypothesis** **`mergesort`** correctly sorts input lists of length i.
  - **Base case** **`mergesort`** correctly sorts input lists of length 1; it returns a 1-element list, which is trivially sorted.
  - **Inductive step** Suppose the algorithm works on input lists of length 1 to i. Calling **`mergesort`** on an input list of length i+1 recursively calls **`mergesort`** on the left and right halves, which have lengths between 1 and i; therefore, **`left`** and **`right`** contain the elements originally in the left and right halves of the list, but sorted. Given two sorted lists, **`merge`** returns a single sorted list with all of the elements from the original two lists.

    Proving this statement requires another proof by induction, with a loop invariant!

  - **Conclusion** The inductive hypothesis holds for all i. In particular, given an input list of any length n, mergesort returns a sorted version of that list!

Proving Correctness

```
def proof_of_correctness_helper(algorithm):
```

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)
```

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)
```

Today's Outline

- Divide and Conquer I
  - ~~Proving correctness with induction~~ **Done!**
  - Proving runtime with recurrence relations
  - Proving the Master method
  - *Problems: Comparison-sorting*
  - *Algorithms: Mergesort*
  - Reading: CLRS 2.3, 4.3-4.6

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point…
  1. **Does this actually work?**
  2. **Is it fast?**

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point…
  1. **Does this actually work? Yes!**
  2. **Is it fast?**

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
          int m = (l+r)/2;
          mergeSort(arr, l, m);
          mergeSort(arr, m + 1, r);

           merge(arr, l, m, r);
    }
}
```

Mergesort

- **Intuition** Divide the list into halves, recursively sort them, merge the sorted halves into a whole sorted list, and return this list.
- You might have two questions at this point…
  1. **Does this actually work? Yes!**
  2. **Is it fast?**

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
         int m = (l+r)/2;
         mergeSort(arr, l, m);
         mergeSort(arr, m + 1, r);

         merge(arr, l, m, r);
    }
}
```

Analyzing Runtime

2. Is it fast?

- Let T(n) represent the runtime of **`mergesort`** on a list of length n.
  - Extending this notation, T(n/2) is the runtime of **`mergesort`** on a list of length n/2 and T(1000) is the runtime of **`mergesort`** on a list of length 1000.
  - Calling **`mergesort`** on a list of length n calls **`mergesort`** once for each half, a total runtime of $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$.
  - What is the runtime of **`merge`**?

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Mergesort

```
void merge(int arr[], int l, int m, int r)
{
   while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    /* Copy the remaining elements of L[], if there  are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Mergesort

```
void merge(int arr[], int l, int m, int r)
{
   while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    /* Copy the remaining elements of L[], if there  are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

At most len(L) + len(R),
which is n iters

Analyzing Runtime

2. Is it fast?

- Let **T(n)** represent the runtime of **`mergesort`** on a list of length n.
  - Extending this notation, T(n/2) is the runtime of **`mergesort`** on a list of length n/2 and T(1000) is the runtime of **`mergesort`** on a list of length 1000.
  - Calling **`mergesort`** on a list of length n calls **`mergesort`** once for each half, a total runtime of **$T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$**.
  - What is the runtime of **`merge`**?

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Analyzing Runtime

2. Is it fast?

- Let **T(n)** represent the runtime of **`mergesort`** on a list of length n.
    - Extending this notation, T(n/2) is the runtime of **`mergesort`** on a list of length n/2 and T(1000) is the runtime of **`mergesort`** on a list of length 1000.
    - Calling **`mergesort`** on a list of length n calls **`mergesort`** once for each half, a total runtime of **$T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$**.
    - What is the runtime of **`merge`**? **$\Theta(n)$**.

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Analyzing Runtime

2. Is it fast?

- Let **T(n)** represent the runtime of **`mergesort`** on a list of length n.
  - Extending this notation, T(n/2) is the runtime of **`mergesort`** on a list of length n/2 and T(1000) is the runtime of **`mergesort`** on a list of length 1000.
  - Calling **`mergesort`** on a list of length n calls **`mergesort`** once for each half, a total runtime of **$T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$**.
  - What is the runtime of **`merge`**? **$\Theta(n)$**.
- Here's our first **recurrence relation**! (점화식)
  - $T(0) = \Theta(1)$
  - $T(1) = \Theta(1)$
  - $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.
    - A well-known recurrence relation defines the Fibonacci sequence: $T(n) = T(n-1) + T(n-2)$.

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.
    - A well-known recurrence relation defines the Fibonacci sequence: $T(n) = T(n-1) + T(n-2)$.
- Our recurrence relation for the runtime of **`mergesort`** isn't very useful unless we can determine the runtime as closed-form expression.

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.
  - A well-known recurrence relation defines the Fibonacci sequence: $T(n) = T(n-1) + T(n-2)$.
- Our recurrence relation for the runtime of **`mergesort`** isn't very useful unless we can determine the runtime as closed-form expression (일반항).
  - Let's learn how to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$!

Analyzing Runtime

- First, let's make a few simplifications.

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

Analyzing Runtime

- First, let's make a few simplifications.
  - **Simplification 1** Using the definition of Big-Θ, rewrite Θ(1) and Θ(n) terms.

$$T(0) = \Theta(1)$$

$$T(1) \leq c_1$$

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c_2 n$$

Analyzing Runtime

- First, let's make a few simplifications.
  - **Simplification 1** Using the definition of Big-Θ, rewrite Θ(1) and Θ(n) terms.
  - **Simplification 2** n is a power of 2.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) \le c_1$$

$$T(n) \le 2T(n/2) + c_2 n$$

Analyzing Runtime

- First, let's make a few simplifications.
  - **Simplification 1** Using the definition of Big-Θ, rewrite Θ(1) and Θ(n) terms.
  - **Simplification 2** n is a power of 2.
  - **Simplification 3** $c = \max\{c_1, c_2\}$.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

Analyzing Runtime

- First, let's make a few simplifications.
  - **Simplification 1** Using the definition of Big-Θ, rewrite Θ(1) and Θ(n) terms.
  - **Simplification 2** n is a power of 2.
  - **Simplification 3** $c = \max\{c_1, c_2\}$.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) \le c$$

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

Analyzing Runtime

- First, let's make a few simplifications.
  - **Simplification 1** Using the definition of Big-Θ, rewrite Θ(1) and Θ(n) terms.
  - **Simplification 2** n is a power of 2.
  - **Simplification 3** $c = \max\{c_1, c_2\}$.

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) \le c$$

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

How do we translate this simplified recurrence relation to a closed-form expression?

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

$$T(1) \le c$$

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

$$\Downarrow$$

$$T(n) = O(?)$$

Closed-form expression

Solving Recurrences

- There are a few different methods to translate a recurrence relation for $T(n)$ to a closed form expression for $T(n)$.
    - Recursion tree method
    - Iteration method
    - Master method
    - Substitution Method

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method**
  - Iteration method
  - Master method
  - Substitution Method

Recursion Tree Method

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$

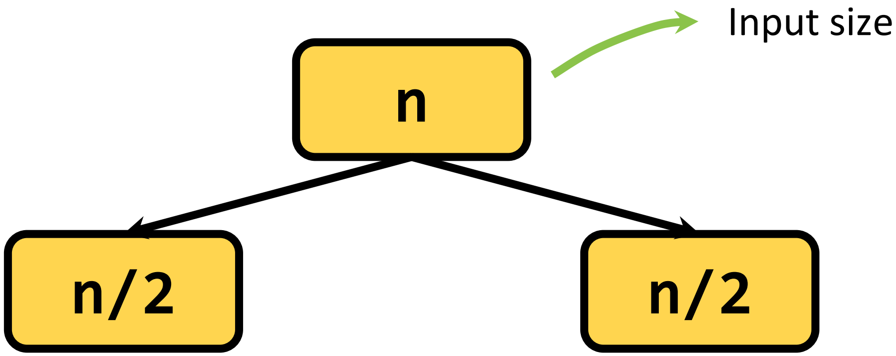
n

Input size

Recursion Tree Method

$T(1) \leq c$

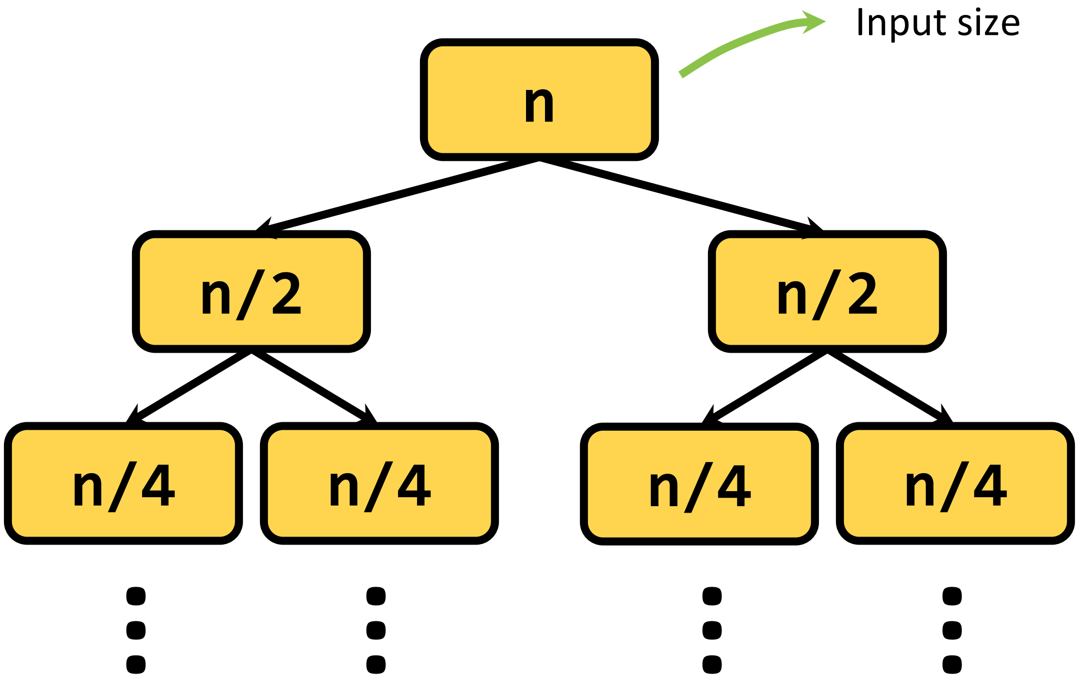
$T(n) \leq 2T(n/2) + cn$



$$T(1) \le c$$

Recursion Tree Method

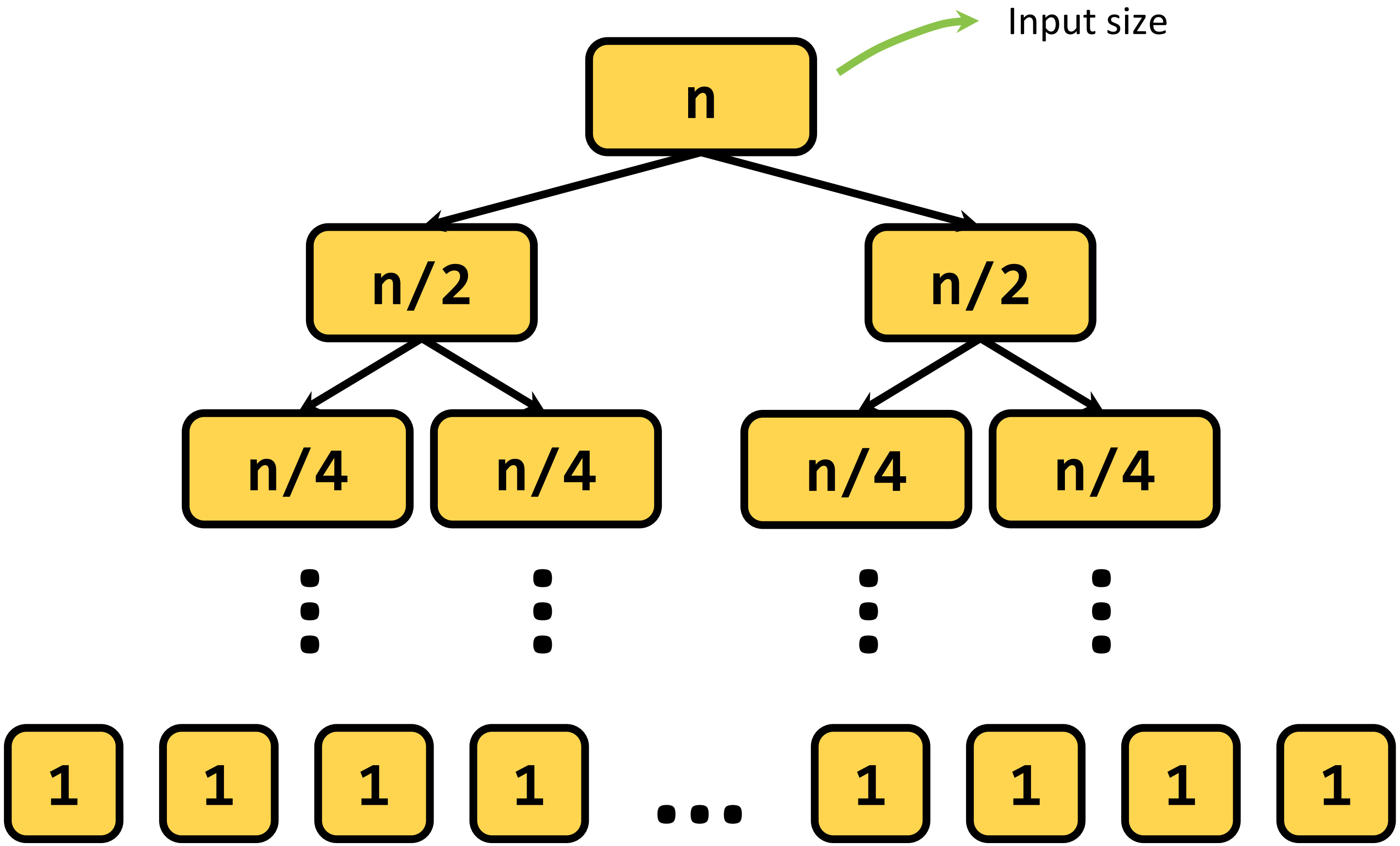
$$T(n) \le 2T(n/2) + cn$$



Recursion Tree Method

$T(1) \leq c$

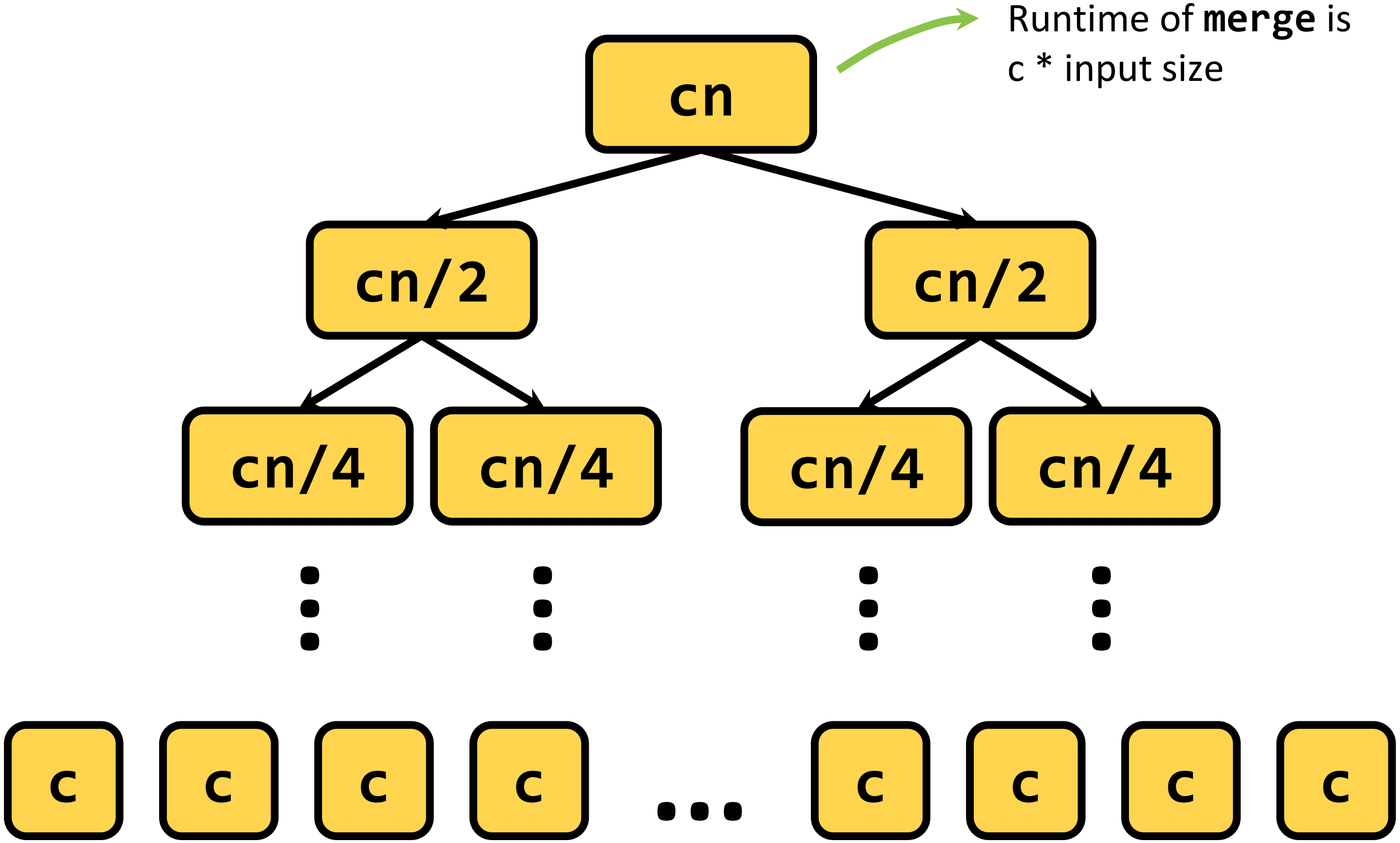
$T(n) \leq 2T(n/2) + cn$



Recursion Tree Method

$T(1) \leq c$

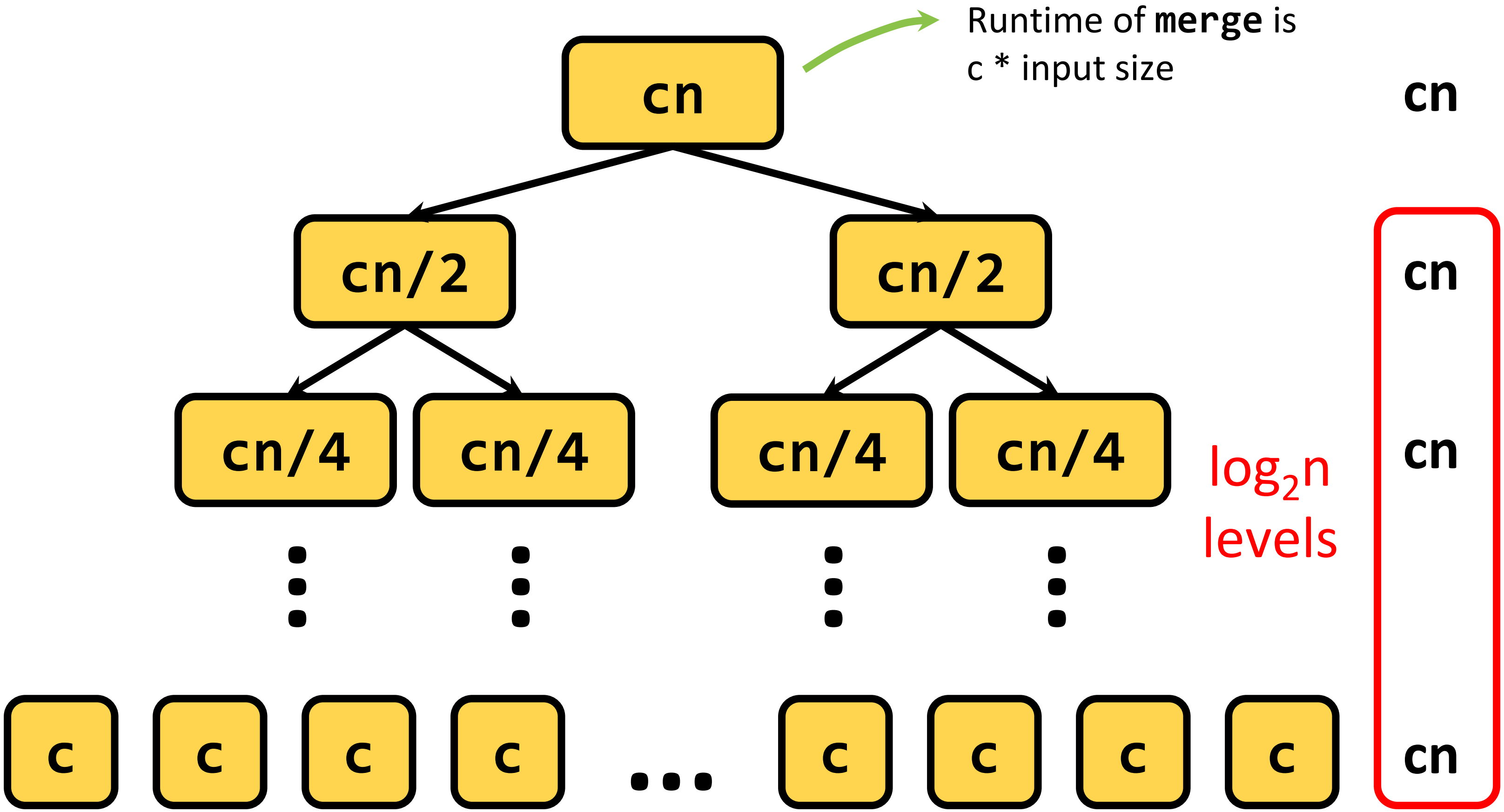
$T(n) \leq 2T(n/2) + cn$



$T(1) \le c$

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

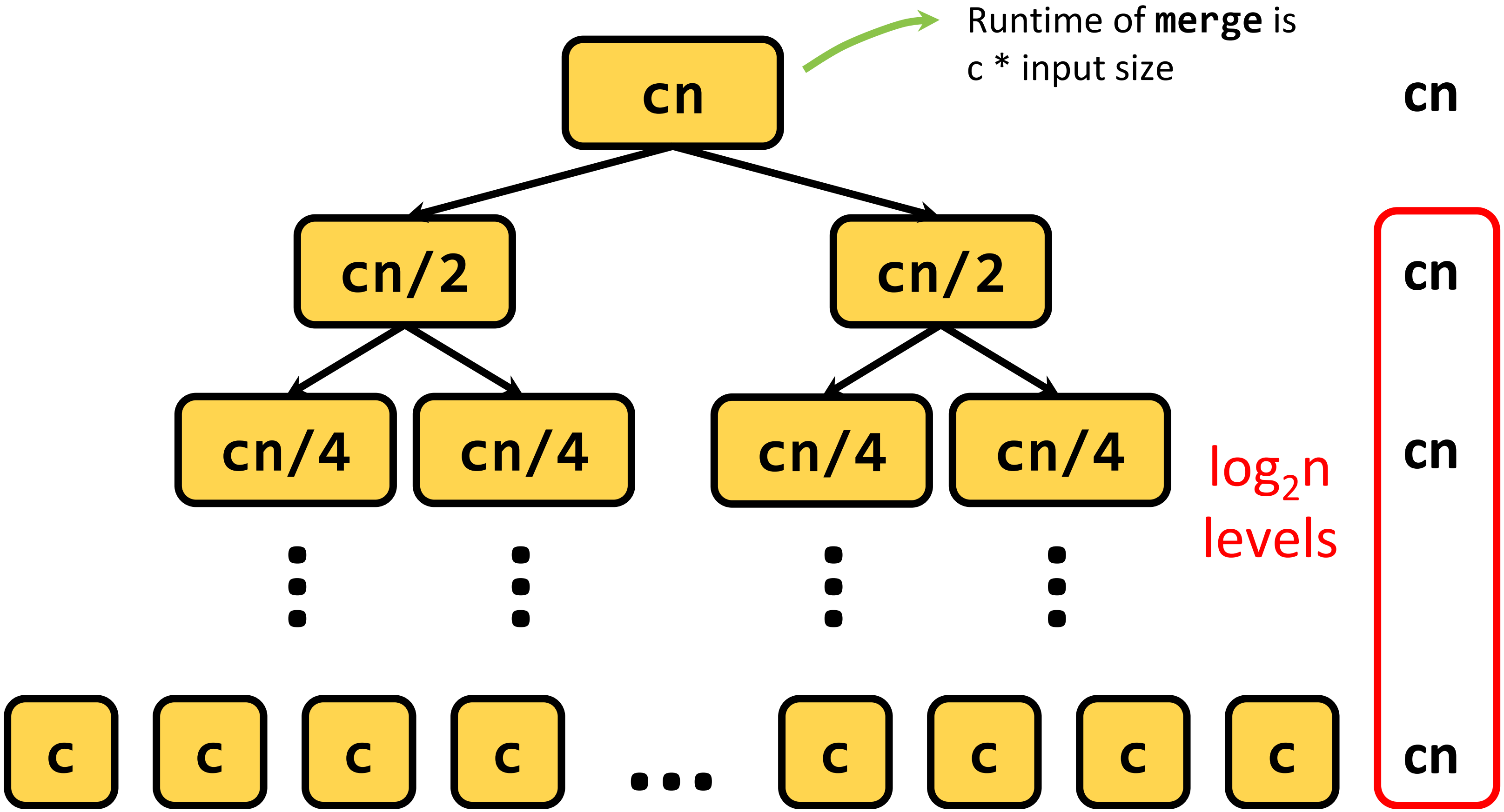
Recursion Tree Method



Recursion Tree Method

$T(1) \le c$

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

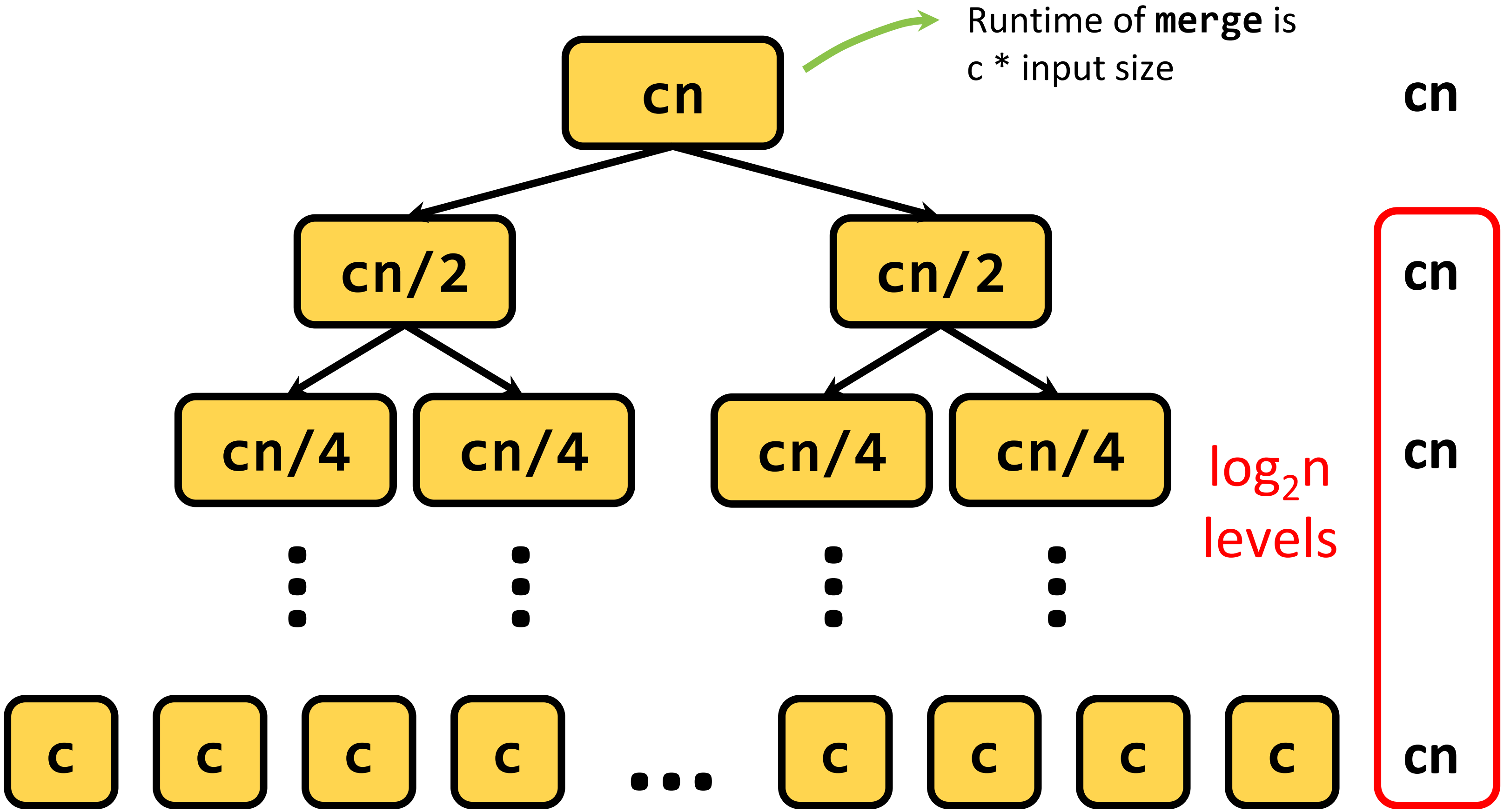


Runtime $cn \log_2 n + cn = O(n \log(n))$

Recursion Tree Method

$T(1) \le c$

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



Runtime $cn \log_2 n + cn = O(n \log(n))$

Use same reasoning for Ω

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method $\Theta(n \log(n))$**.
  - Iteration method
  - Master method
  - Substitution Method

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method** **$\Theta(n \log(n))$**.
  - **Iteration method**
  - Master method
  - Substitution Method

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$

Iteration Method

- Apply the relationship until you see a pattern.

  $T(n) \leq 2 \cdot T(n/2) + cn$

Iteration Method

$T(1) \le c$

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

- Apply the relationship until you see a pattern.

$$\begin{aligned} T(n) &\le 2\cdot T(n/2) + cn \\ &\le 2\cdot(2T(n/4) + cn/2) + cn \\ &= 4\cdot T(n/4) + 2cn \end{aligned}$$

Iteration Method

$T(1) \le c$

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

- Apply the relationship until you see a pattern.

$$
\begin{aligned}
T(n) &\le 2\cdot T(n/2) + cn \\
&\le 2\cdot(2T(n/4) + cn/2) + cn \\
&= 4\cdot T(n/4) + 2cn \\
&\le 4\cdot(2T(n/8) + cn/4) + 2cn \\
&= 8\cdot T(n/8) + 3cn
\end{aligned}
$$

$T(1) \le c$

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

Iteration Method

- Apply the relationship until you see a pattern.

$$
\begin{aligned}
T(n) &\le 2\cdot T(n/2) + cn \\
&\le 2\cdot(2T(n/4) + cn/2) + cn \\
&= 4\cdot T(n/4) + 2cn \\
&\le 4\cdot(2T(n/8) + cn/4) + 2cn \\
&= 8\cdot T(n/8) + 3cn \\
&\ldots \\
&\le 2^k T(n/2^k) + kcn
\end{aligned}
$$

Iteration Method

$T(1) \le c$

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

- Apply the relationship until you see a pattern.

$$
\begin{aligned}
T(n) &\le 2\cdot T(n/2) + cn \\
&\le 2\cdot(2T(n/4) + cn/2) + cn \\
&= 4\cdot T(n/4) + 2cn \\
&\le 4\cdot(2T(n/8) + cn/4) + 2cn \\
&= 8\cdot T(n/8) + 3cn \\
&\ldots \\
&\le 2^k T(n/2^k) + kcn
\end{aligned}
$$

- What is k?

$T(1) \le c$

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

Iteration Method

- Apply the relationship until you see a pattern.

$$
\begin{aligned}
T(n) &\le 2\cdot T(n/2) + cn \\
&\le 2\cdot(2T(n/4) + cn/2) + cn \\
&= 4\cdot T(n/4) + 2cn \\
&\le 4\cdot(2T(n/8) + cn/4) + 2cn \\
&= 8\cdot T(n/8) + 3cn \\
&\dots \\
&\le 2^k T(n/2^k) + kcn
\end{aligned}
$$

- What is k? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$T(n) \le 2^k T(n/2^k) + kcn$$

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$

Iteration Method

- Apply the relationship until you see a pattern.

$$
\begin{aligned}
T(n) &\leq 2\cdot T(n/2) + cn \\
&\leq 2\cdot(2T(n/4) + cn/2) + cn \\
&= 4\cdot T(n/4) + 2cn \\
&\leq 4\cdot(2T(n/8) + cn/4) + 2cn \\
&= 8\cdot T(n/8) + 3cn \\
&\dots \\
&\leq 2^k T(n/2^k) + kcn
\end{aligned}
$$

- What is k? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$
\begin{aligned}
T(n) &\leq 2^k T(n/2^k) + kcn \\
&= 2^{\text{log\_2(n)}} T(n/2^{\text{log\_2(n)}}) + cn\log_2 n
\end{aligned}
$$

Substitute $k = \log_2 n$

$T(1) \le c$

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

Iteration Method

- Apply the relationship until you see a pattern.

$$
\begin{aligned}
T(n) &\le 2\cdot T(n/2) + cn \\
&\le 2\cdot(2T(n/4) + cn/2) + cn \\
&= 4\cdot T(n/4) + 2cn \\
&\le 4\cdot(2T(n/8) + cn/4) + 2cn \\
&= 8\cdot T(n/8) + 3cn \\
&\dots \\
&\le 2^k T(n/2^k) + kcn
\end{aligned}
$$

- What is k? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$
\begin{aligned}
T(n) &\le 2^k T(n/2^k) + kcn \\
&= 2^{\log\_2(n)} T(n/2^{\log\_2(n)}) + cn\log_2 n && \text{Substitute } k = \log_2 n \\
&= nT(1) + cn\log_2 n && \text{Simplify}
\end{aligned}
$$

Iteration Method

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$

- Apply the relationship until you see a pattern.

$$
\begin{aligned}
T(n) &\leq 2\cdot T(n/2) + cn \\
&\leq 2\cdot(2T(n/4) + cn/2) + cn \\
&= 4\cdot T(n/4) + 2cn \\
&\leq 4\cdot(2T(n/8) + cn/4) + 2cn \\
&= 8\cdot T(n/8) + 3cn \\
&\dots \\
&\leq 2^k T(n/2^k) + kcn
\end{aligned}
$$

- What is k? It's the number of times to divide n by 2 to get 1. So $k = \log_2 n$.

$$
\begin{aligned}
T(n) &\leq 2^k T(n/2^k) + kcn \\
&= 2^{\log\_2(n)} T(n/2^{\log\_2(n)}) + cn\log_2 n && \text{Substitute } k = \log_2 n \\
&= nT(1) + cn\log_2 n && \text{Simplify} \\
&\leq cn + cn\log_2 n \\
&= \mathbf{O(n\log n)}
\end{aligned}
$$

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method $\Theta(n \log(n))$**.
  - **Iteration method $\Theta(n \log(n))$**.
  - Master method
  - Substitution Method

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method $\Theta(n \log(n))$**.
  - **Iteration method $\Theta(n \log(n))$**.
  - **Master method Next time!**
  - Substitution Method

Today's Outline

- ~~Comparison sorting algorithms: selection sort, bubble sort~~ **Done!**
- Divide and Conquer I
  - ~~Proving correctness with induction~~ **Done!**
  - ~~Proving runtime with **recurrence relations**~~ **Done!**
  - Proving runtime with the **Master method**

  - *Problems: Comparison-sorting*
  - *Algorithms: Mergesort*
  - Reading: CLRS 2.3, 4.3-4.6

Introduction to Algorithms

L3. Divide & Conquer. II

Instructor : Kilho Lee

Today's Outline

- ~~Comparison sorting algorithms: selection sort, bubble sort~~ **Done!**
- Divide and Conquer I
  - ~~Proving correctness with induction~~ **Done!**
  - ~~Proving runtime with **recurrence relations**~~ **Done!**
  - Proving runtime with the **Master method**
  - Proving runtime with the **Substitution method**

  - *Problems: Comparison-sorting*
  - *Algorithms: Mergesort*
  - Reading: CLRS 2.3, 4.3-4.6

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method** **$\Theta(n \log(n))$**.
  - **Iteration method** **$\Theta(n \log(n))$**.
  - **Master method** **Today!**
  - Substitution Method

Analyzing Runtime

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier or smaller values.

```
Void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = (l+r)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

$$T(1) \leq c$$

$$T(n) \leq 2T(n/2) + cn$$

$$\Downarrow$$

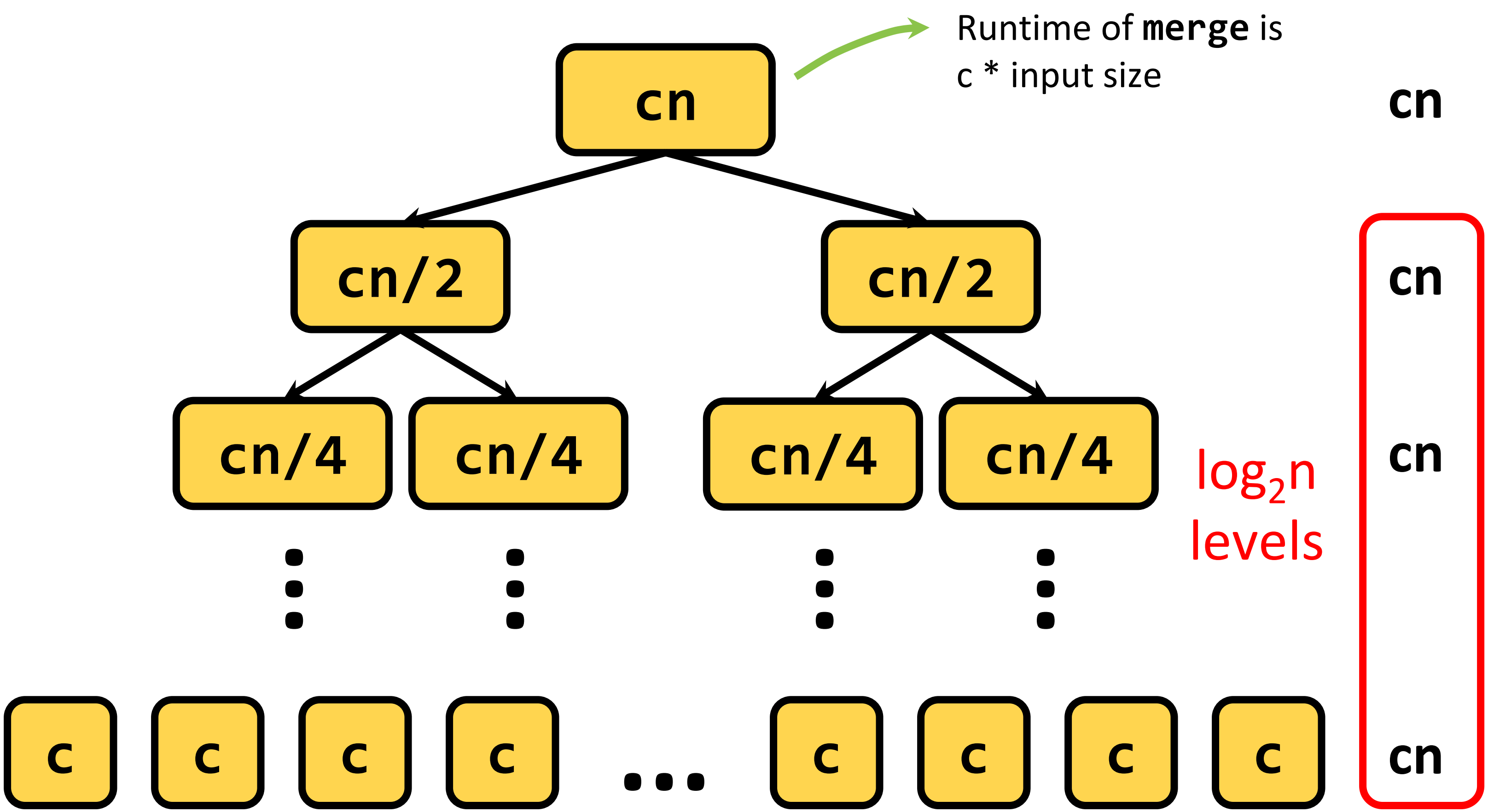
$$T(n) = O(?)$$

Closed-form expression

Recursion Tree Method

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$



Runtime $cn \log_2 n + cn = O(n \log(n))$

Use same reasoning for Ω

More examples

- Needlessly recursive integer multiplication
- $T(n) = 4\,T(n/2) + O(n)$

- Karatsuba integer multiplication
- $T(n) = 3\,T(n/2) + O(n)$

- MergeSort
- $T(n) = 2T(n/2) + O(n)$

- Another example
- $T(n) = 2T(n/2) + O(\,n^2\,)$

What's the pattern?!?!?!?!

A

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method** **Θ(n log(n))**.
  - **Iteration method** **Θ(n log(n))**.
  - **Master method**
  - Substitution Method

Master Method

- A formula that solves recurrences **when all of the sub-problems are the same size**
  - We will see an example later when not all problems are the same size.

A useful
formula it is.
You should know
why it works.

Jedi master Yoda

Master Method

- Suppose T(n) = a·T(n/b) + f(n).
  - $a \geq 1$ and $b > 1$ are constant
  - $f(n)$ is an asymptotically-positive (>0) function
- Compare $f(n)$ with $n^{\log_b a}$ (polynomially):
  - $n^{\log_b a}$ = number of leaves in the recursion tree

The Master method states:

Case	Condition	Regularity condition	Solution
1	$f(n) = \Theta(n^{\log_b a})$		$T(n) = \Theta(n^{\log_b a} \, log \, n)$
2	$f(n) = \Omega(n^{\log_b a+\epsilon})$ for a constant $\epsilon > 0$	$af\left(\frac{n}{b}\right) \leq cf(n)$ for a constant $c < 1$	$T(n) = \Theta(f(n))$
3	$f(n) = O(n^{\log_b a-\epsilon})$ for a constant $\epsilon > 0$		$T(n) = \Theta(n^{\log_b a})$

Master Method

- Suppose T(n) = a·T(n/b) + O(n^d).

The Master method states:

$\log_b a = d$

$T(1) \le c$

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

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

Three paremeters:

a : the number of subproblems

b : the factor by which the input size shrinks (n을 몇 개의 부분으로 나누었는지)

d : need to do n^d work to create all the subproblems and combine their solutions (n^d – subproblem들의 결과를 합치기 위한 runtime)

Master Method

- Suppose T(n) = a·T(n/b) + O(n^d).

We can also take n/b to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$ and the theorem is still true.

The Master method states:

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

T(1) ≤ c

T(n) ≤ 2T(n/2) + cn

a = 2, b = 2, d = 1

Three paremeters:

a : the number of subproblems

b : the factor by which the input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

Understanding the Master Theorem

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

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

- What do these three cases mean?

a : the number of subproblems

b : the factor by which the input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

The Eternal Struggle

Branching causes the number of problems to explode!
The most work is at the bottom of the tree!

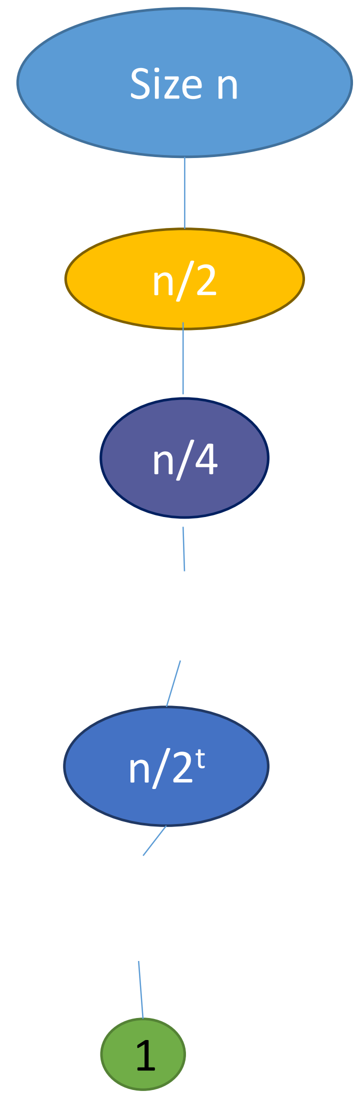
The problems lower in the tree are smaller!
The most work is at the top of the tree!

Consider Examples

1. $T(n) = T\left(\frac{n}{2}\right) + n$
2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$
3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n$

First example: tall and skinny tree

1. $T(n) = T\left(\frac{n}{2}\right) + n, \qquad \left(a < b^d\right)$



- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else.

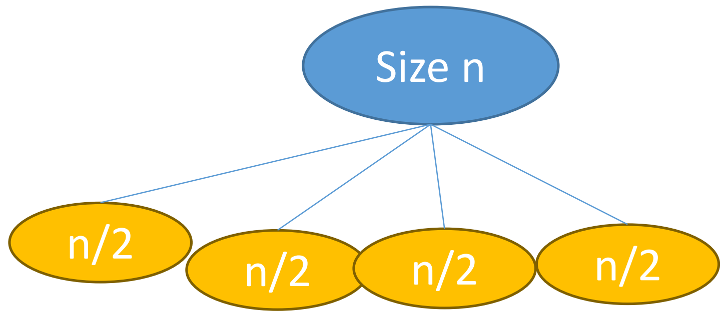
- T(n) = O(work at top) = O(n)

WINNER

Most work at the top of the tree!

Third example: bushy tree

3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \qquad \left(a > b^d\right)$

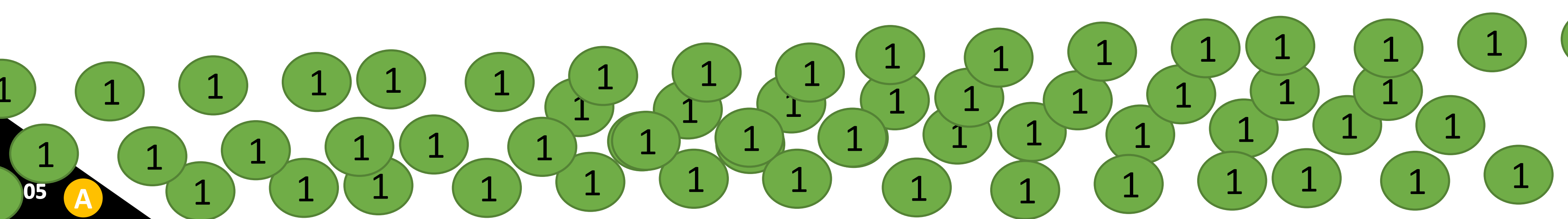


WINNER

Most work at the bottom of the tree!

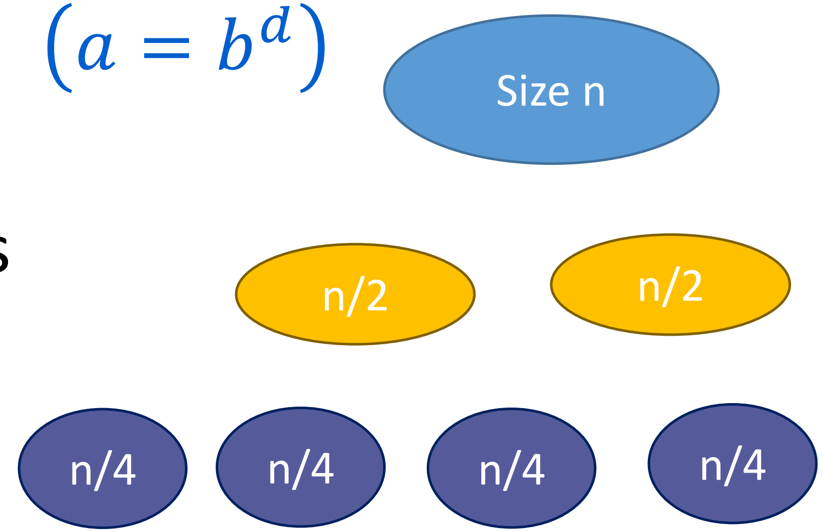
- There are a HUGE number of leaves, and the total work is dominated by the time to do work at these leaves.

- T(n) = O(work at bottom) = O($4^{\text{depth of tree}}$) = O(n^2)



Second example: just right

$$2.\quad T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \qquad \left(a = b^d\right)$$



- The branching **just** balances out the amount of work.
- The same amount of work is done at every level.
- T(n) = (number of levels) * (work per level)
- = log(n) * O(n) = O(nlog(n))

Master Method

- We can prove the Master Method by writing out a generic proof using a recursion tree.
  - Draw out the tree.
  - Determine the work per level.
  - Sum across all levels.
  - Details on ch 4.6.
- The three cases of the Master Method correspond to whether the recurrence is top heavy, balanced, or bottom heavy.

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method** **$\Theta(n \log(n))$**.
  - **Iteration method** **$\Theta(n \log(n))$**.
  - **Master method** **$\Theta(n \log(n))$**.
  - Substitution Method

Solving Recurrences

- There are a few different methods to translate a recurrence relation for T(n) to a closed form expression for T(n).
  - **Recursion tree method** **Θ(n log(n))**.
  - **Iteration method** **Θ(n log(n))**.
  - **Master method** **Θ(n log(n))**.
  - **Substitution Method** **Next time!**

Master Method

- Suppose $T(n) = a \cdot T(n/b) + O(n^d)$.

The Master method states:

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

Three paremeters:

a : the number of subproblems

b : the factor by which the input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

A powerful theorem it is…

Jedi master Yoda

The master theorem only works when all sub-problems are the same size.

Substitution Method

Substitution Method

Guess and verify

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:
  - $T(1) \leq 1$
  - $T(n) \leq 2T(n/2) + n$

$T(1) \leq 1$

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

Substitution Method

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

- Try solving 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...
  $T(n) = nT(1) + n \log(n) = \mathbf{n (\log(n) + 1)}$

$T(1) \leq 1$

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

Substitution Method

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

- **Inductive hypothesis** $T(k) \leq k(\log(k) + 1)$ for all $1 \leq k < n$.
- **Base case** $T(1) = 1 = 1(\log(1) + 1)$.
- **Inductive step**
  - Support that the hyp. holds when k/2
  - $T(k) = 2T(k/2) + k$ Substitute n/2 into inductive hyp.

$$\begin{aligned} &\leq 2((k/2)(\log(k/2) + 1) + k \\ &= 2((k/2)(\log(k) - 1 + 1) + k \\ &= 2((k/2) \log(k)) + k \\ &= k(\log(k) + 1) \end{aligned}$$

- **Conclusion** By induction, $T(n) \leq n(\log(n) + 1)$ for all $n > 0$.

Substitution Method

- So far, just seems like a different way of doing the same thing.
- But, 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 ≤ n ≤ 10

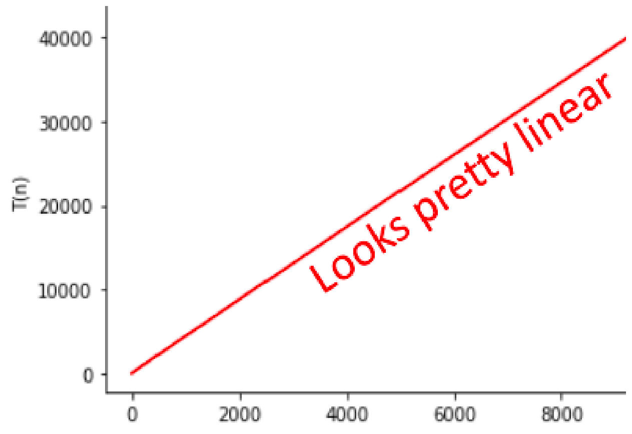
T(n) = 3n + T(n/5) + T(n/2), otherwise

1. **Guess what the answer is.**
   - **Try solving it**

     [Whiteboard] – Gets gross fast
   - **Try plotting it** → **Guess O(n)**
   - **What else do we know?**
     - $T(n) \leq 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$

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

       $= O(n\log(n))$
     - $T(n) \geq 3n$
     - **So the right answer is somewhere between O(n) and O(nlog(n)).**
   - **Let's guess O(n)**



Substitution Method

$$T(n) = 10n \text{ when } 1 \le n \le 10$$

$$T(n) = 3n + T(n/5) + T(n/2) \text{ otherwise}$$

Guess 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 $k \le 10$.
- **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

$$T(n) = 10n \text{ when } 1 \le n \le 10$$

$$T(n) = 3n + T(n/5) + T(n/2) \text{ otherwise}$$

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

Pretend like we knew it

- **Inductive hypothesis** $T(k) \le 10k$ for all $1 \le k < n$.
- **Base case** $T(k) \le 10k$ for all $k \le 10$.
- **Inductive step**
  - $T(n) = 3n + T(n/5) + T(n/2)$
    $\le 3n + 10(n/5) + 10(n/2)$
    $= 3n + (10/5)n + (10/2)n$
    $\le 10n$

Pretend we knew C = 10 all along.

- **Conclusion** For all $n > 1$, $T(n) \le 10n$. Therefore, $T(n) = O(n)$.

Substitution Method

- What have we learned?
  - The substitution method can work when the master theorem doesn't.
    - For example with different-sized sub-problems
  - Step 1: generate a guess
  - Step 2: try to prove that your guess is correct
    - Might need to leave some constants unspecified until the end
    - Then see what they need to be for the proof to work
  - Step 3: profit
    - Pretend you didn't do Steps 1 and 2 and write down a nice proof

Today's Outline

- Divide and Conquer I
  - ~~Proving correctness with induction~~ **Done!**
  - ~~Proving runtime with **recurrence relations**~~ **Done!**
    - How do we measure the runtime of a recursive algorithm?
  - ~~Proving the **Master method**~~ **Done!**
    - A useful theorem (do not have to answer the runtime from scratch)
  - ~~Learn the **Substitution method**~~ **Done!**
    - It can be used when the master method doesn't work
  - *Problems: Comparison-sorting*
  - *Algorithms: Mergesort*
  - Reading: CLRS 2.3, 4.3-4.6

Introduction to Algorithms

L3. Divide & Conquer. III

Instructor : Kilho Lee

Today's Outline

- Divide and Conquer I
  - ~~Proving correctness with induction~~ **Done!**
  - ~~Proving runtime with **recurrence relations**~~ **Done!**
  - ~~Proving the **Master method**~~ **Done!**
  - ~~Learn the **Substitution method**~~ **Done!**
- Divide and Conquer II
  - *Problems: k^{th} number selection*
  - *Algorithms: Linear-time selection*
  - Reading: CLRS 9.2 9.3

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

SELECT(A,k): return the k'th smallest element in A

SELECT(A,0)=0
SELECT(A,1)=1
SELECT(A,2)=4
SELECT(A,4)=16
SELECT(A,9)=81

SELECT(A,0)=MIN(A)
SELECT(A,n/2-1)=MEDIAN(A)
SELECT(A,n-1)=MAX(A)

- 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=[n/2]-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]
```

<u>Worst-case</u> runtime

Θ(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**.

1	64	9	49	16	4	0	25	36	81

Select a pivot at random (for now)

1	9	16	4	0	25	36	64	49	81

Partition around the pivot, such that all elements to the left are less than it and all elements to the right are greater than it
(Notice that the halves remain unsorted.)

Find element k=3 in this half since 36 occupies index 6 and k=3 < 6.

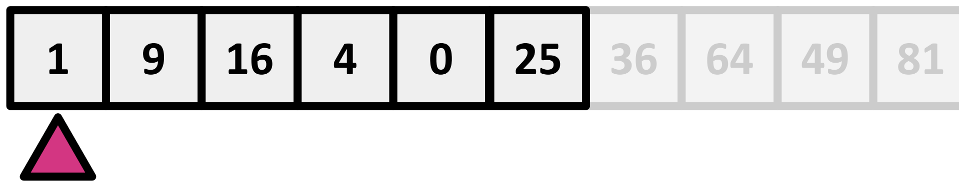
Linear-Time Selection

- **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

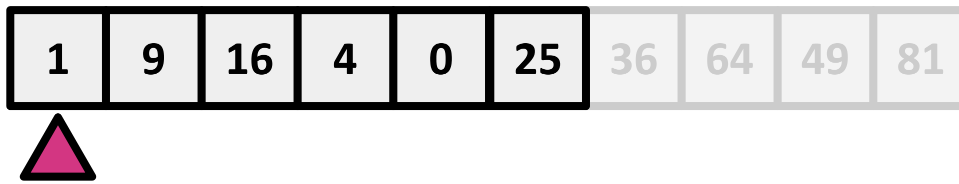
- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.



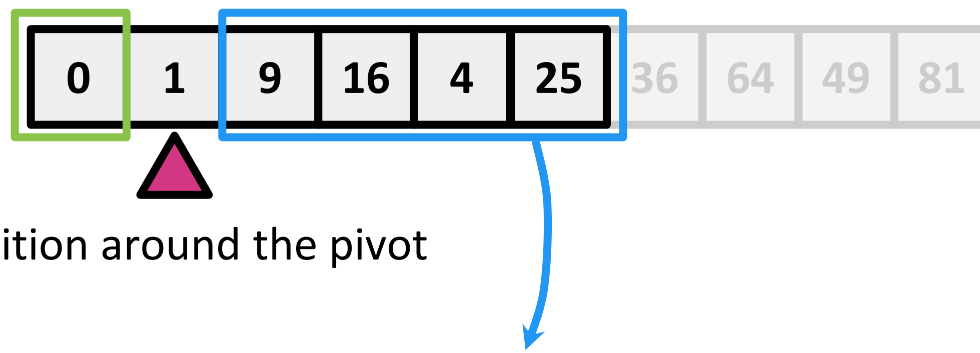
Select another pivot at random (for now)

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)

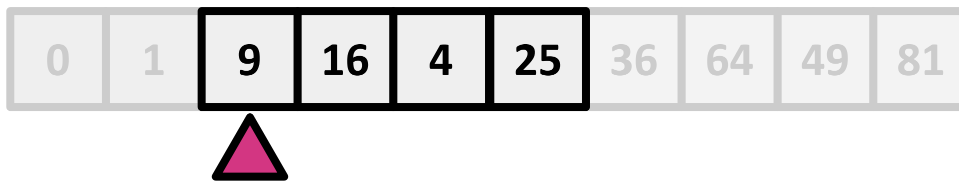


Partition around the pivot

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

Linear-Time Selection

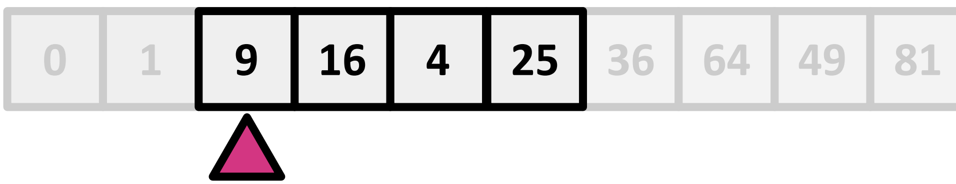
- **Key Insight** Select a pivot, partition around it, and recurse.
  - Suppose we want to find element **k=3**.



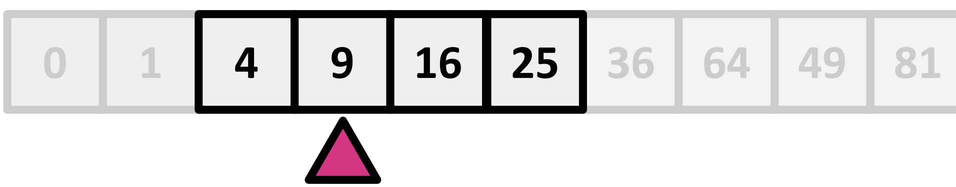
Select another pivot at random (for now)

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

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Linear-Time Selection

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

<u>“Worst-case”</u> runtime

$\Theta(n^2)$

Linear-Time Selection

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

"Worst-case" runtime

$\Theta(n^2)$

We'll discuss this runtime later...

Linear-Time Selection

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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

```
// It searches for x in arr[l..r], and partitions the array around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
           break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}
```

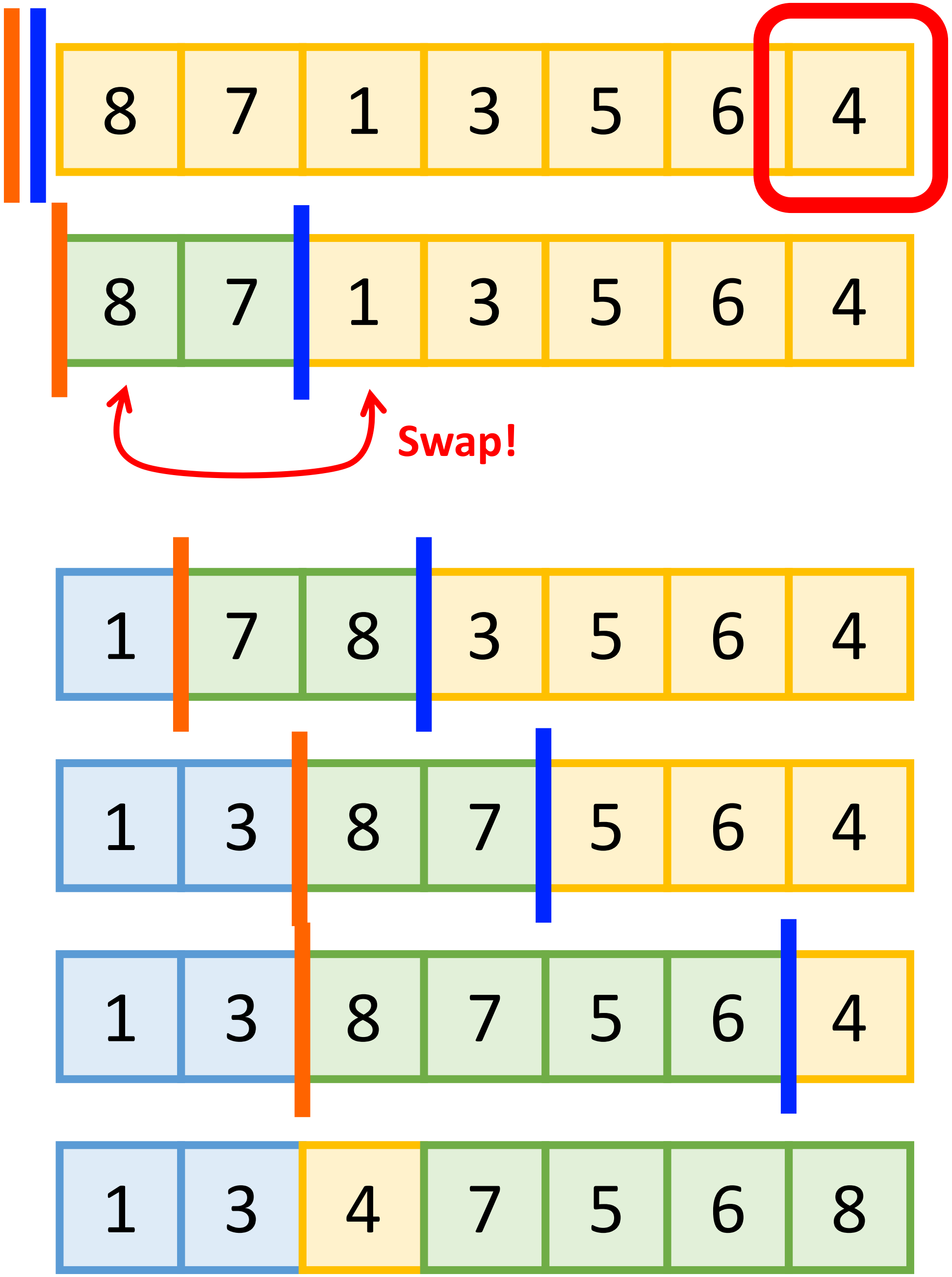
Linear-Time Selection

```
// It searches for x in arr[l..r], and partitions the array around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
           break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}
```

Worst-case runtime

Θ(n)

A better way to do Partition



Pivot

Choose it randomly, then swap it with the last one, so it's at the end.

Initialize and

Step forward.

When sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars.

Repeat till the end, then put the pivot in the right place.

See CLRS

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?**

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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?**

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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.

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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 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.

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 $(k - \text{len(left)} - 1)^{st}$-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
  - 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:

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

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

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

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

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

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

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

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

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

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

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

The runtime to partition about the chosen pivot

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

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

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

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!

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = arr[l+ rand() % n];
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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? $T(n) \leq T(n/2) + O(n)$
    - 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? $T(n) \leq T(n/2) + O(n)$
    - Then, $a = 1$, $b = 2$, $d = 1$ (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:

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

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

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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

Linear-Time Selection

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

<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:

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

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)` $< 7n/10$.
  - 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

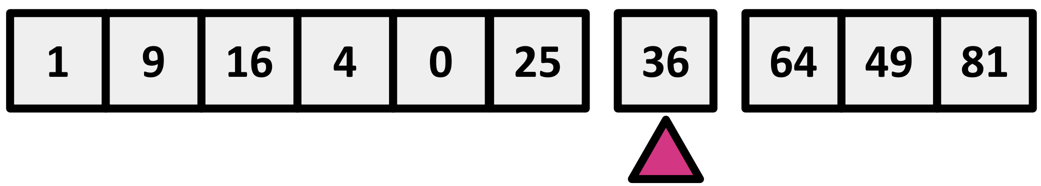
- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In a reasonable world, we split the input roughly in half, such that: $3n/10 <$ **`len(left)`**, **`len(right)`** $< 7n/10$.
  - Once again, we could use **Master Theorem**!
    - What's the recurrence?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

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

Analyzing Runtime in an ~~Ideal~~ Reasonable World

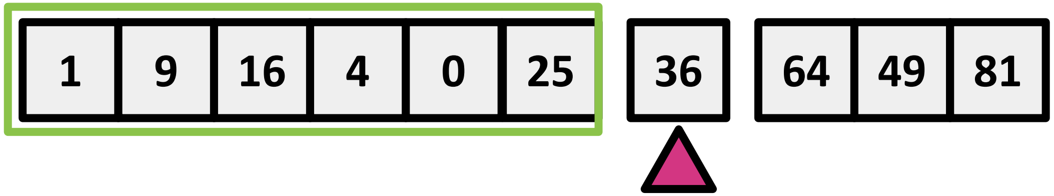
- **`len(left)`** and **`len(right)`** determine the runtime of the recursive calls to **`select`**.
  - In a reasonable world, we split the input roughly in half, such that: 3n/10 < **`len(left)`**, **`len(right)`** < 7n/10.



The **goal** is to pick a pivot such that

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.

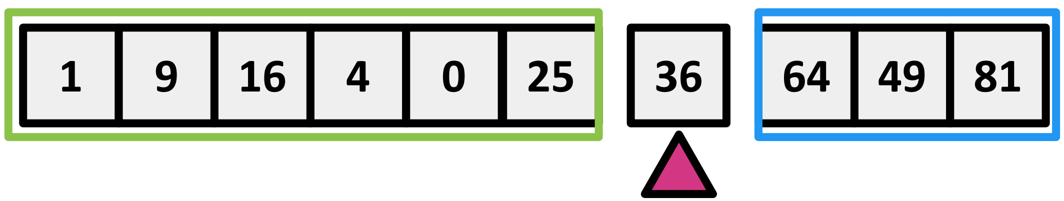


The **goal** is to pick a pivot such that

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?

A

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.**

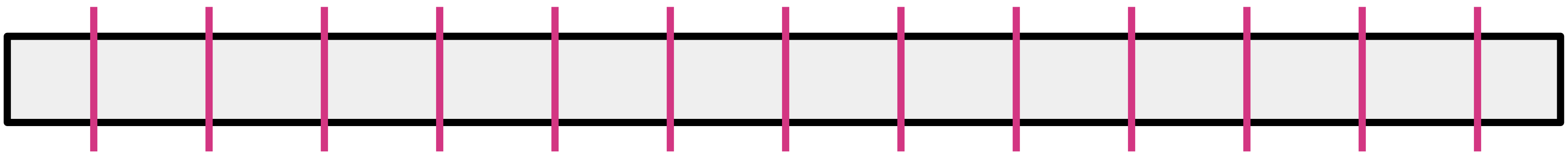
Pro tip: making the inductive hypothesis i.e. assuming correctness of the algorithm on smaller inputs is a helpful technique for designing divide and conquer algorithms.

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.

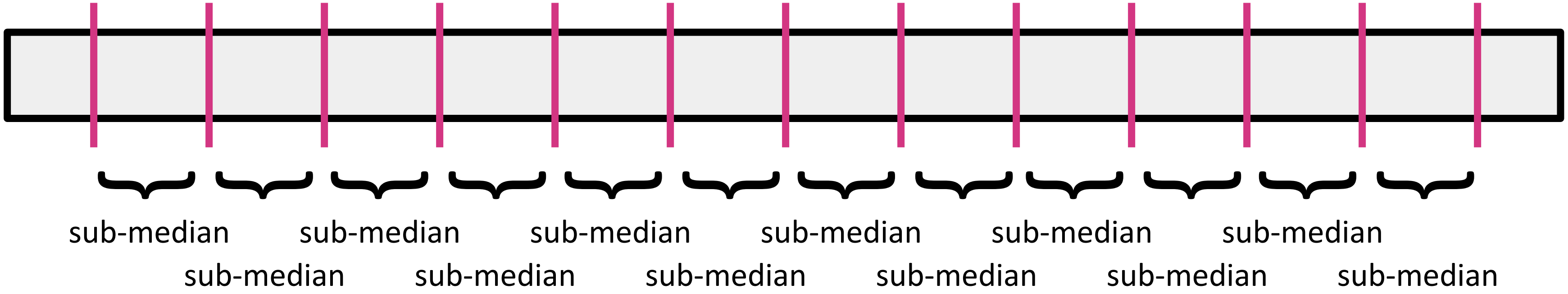
Another Divide and Conquer Algorithm

- **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.



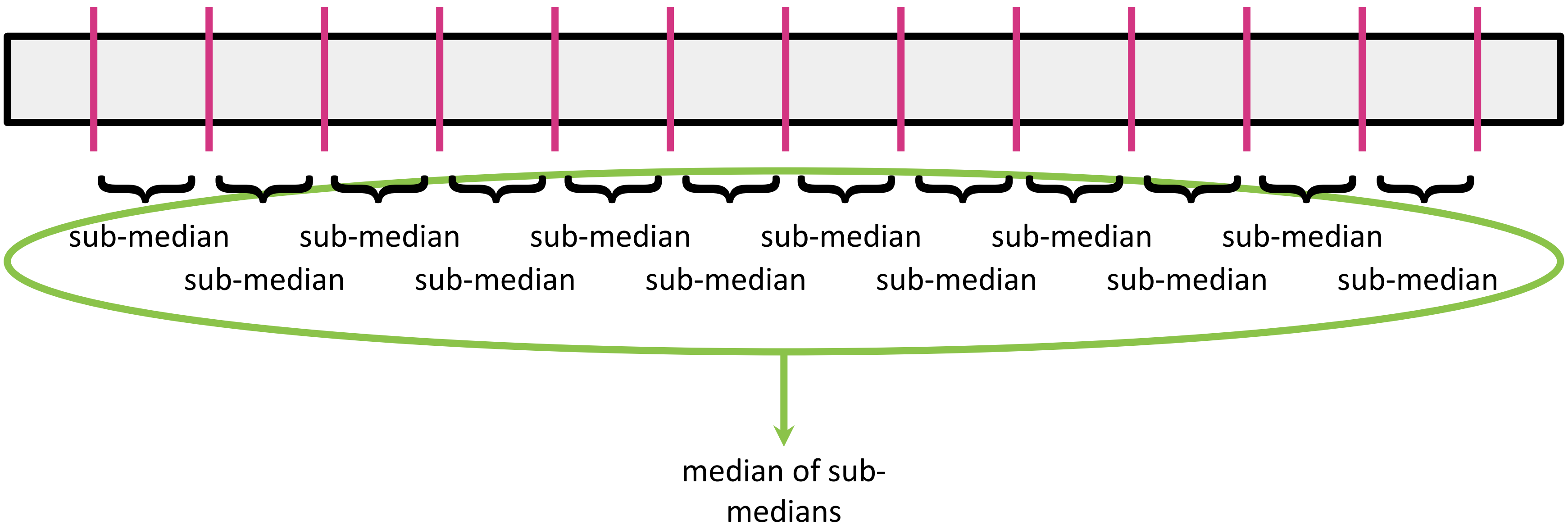
Another Divide and Conquer Algorithm

- **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.



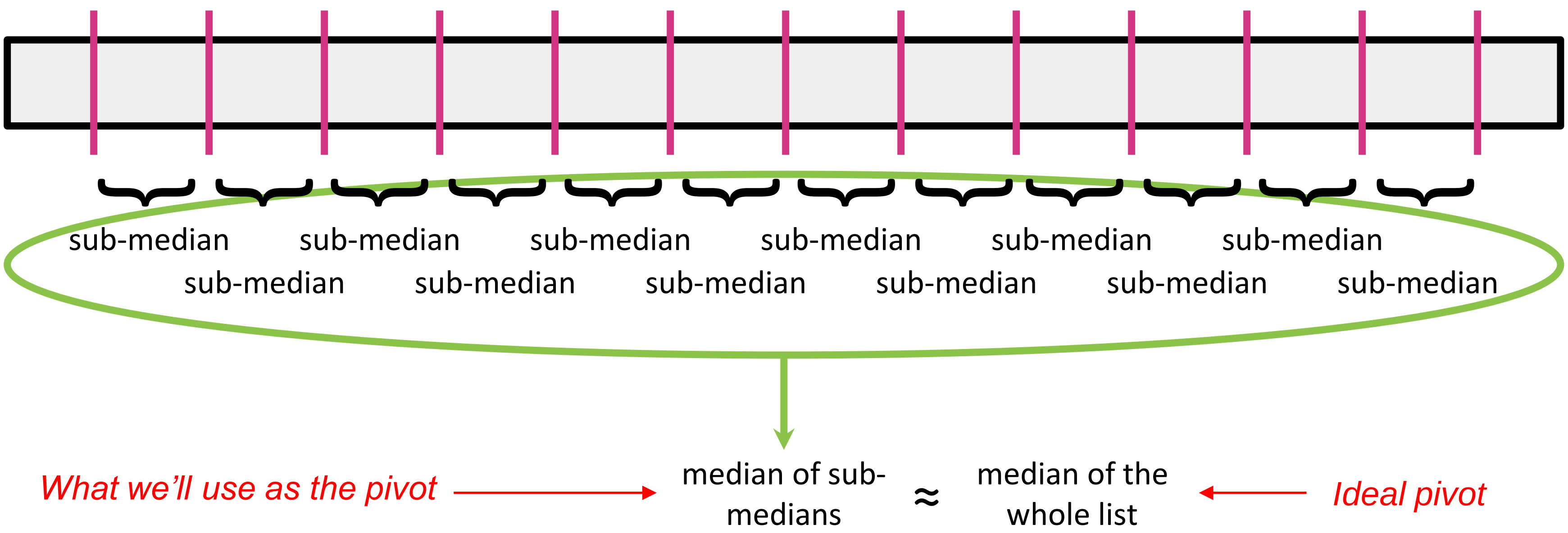
Another Divide and Conquer Algorithm

- **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.

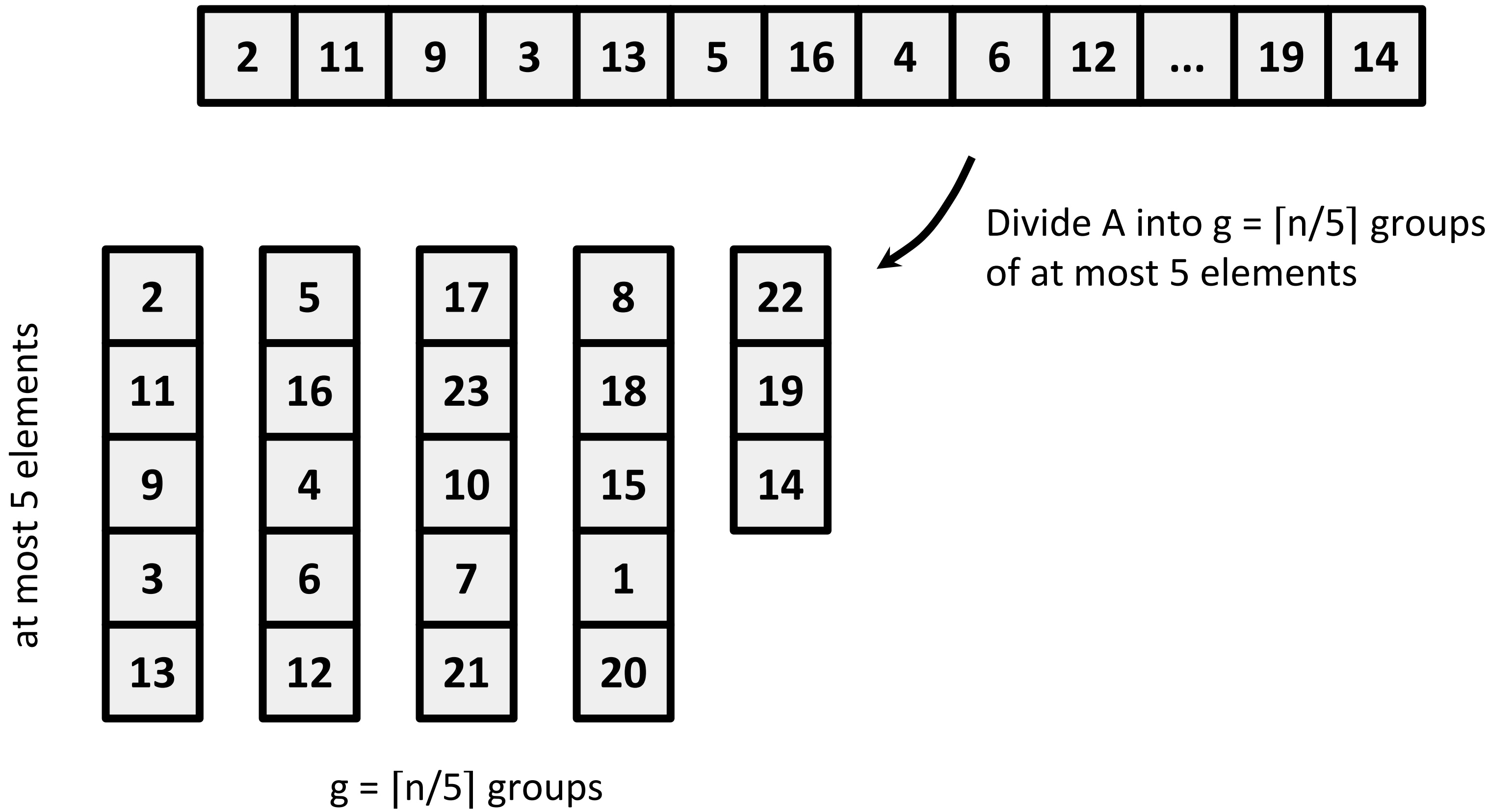


Lemma: The median of sub-medians is close to the median.

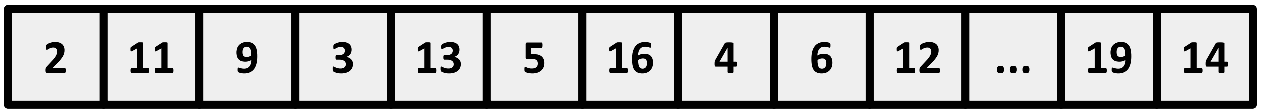
Another Divide and Conquer Algorithm

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

Another Divide and Conquer Algorithm

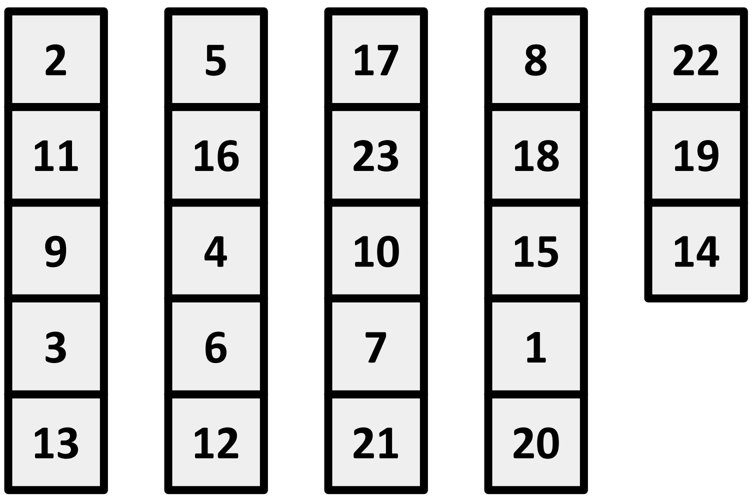


Another Divide and Conquer Algorithm



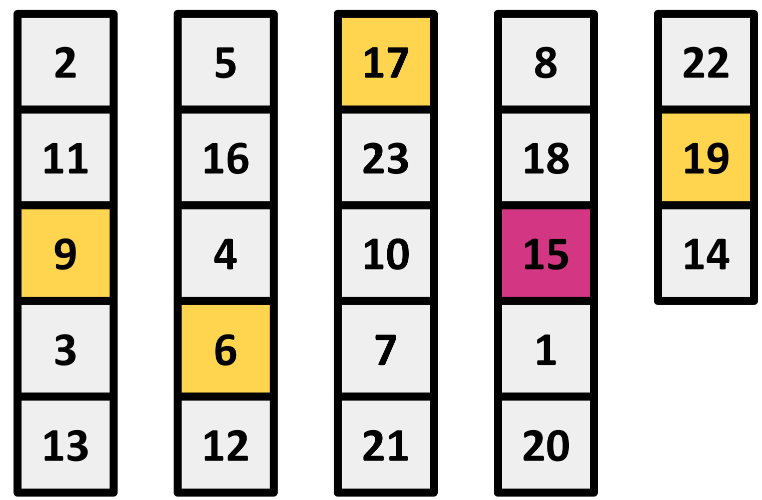
Divide A into g = ⌈n/5⌉ groups
of at most 5 elements

at most 5 elements



g = ⌈n/5⌉ groups

Find the sub-median of each of the groups (yellow) and recursively call `select` to find the median of these sub-medians (pink).



How to pick the pivot

- median_of_medians(A):
  - Split A into m $=\left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** p

This takes time O(1), for each group, since each group has size 5. So that's O(m) total in the for loop.

8	4	5	6	12

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

Pivot is SELECT(8 4 5 6 12 , 3) = 6:

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

PARTITION around that 6:

1	3	5	1	3	4	2	1	2	4	1	3	5	6	8	9	15	9	12	20	15	13	12	15	22

This part is L

This part is R: it's almost the same size as L.

Linear-Time Selection

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

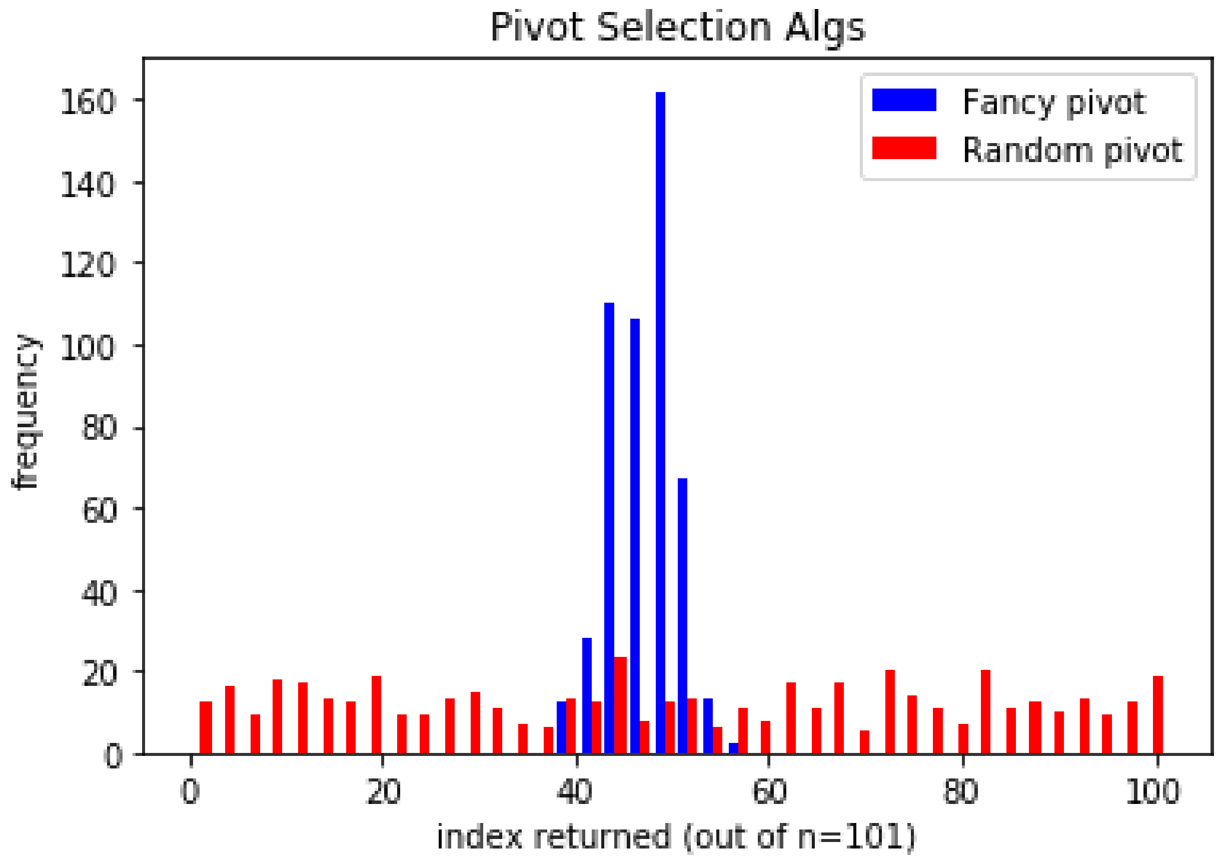
"Worst-case" **runtime** $\Theta(n^2)$

Linear-Time Selection

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = arr[l + rand() % n];    Median_of_median (arr, l, r);
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

- Emprically,



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.

2	5	17	8	22
11	16	23	18	19
9	4	10	15	14
3	6	7	1	
13	12	21	20	

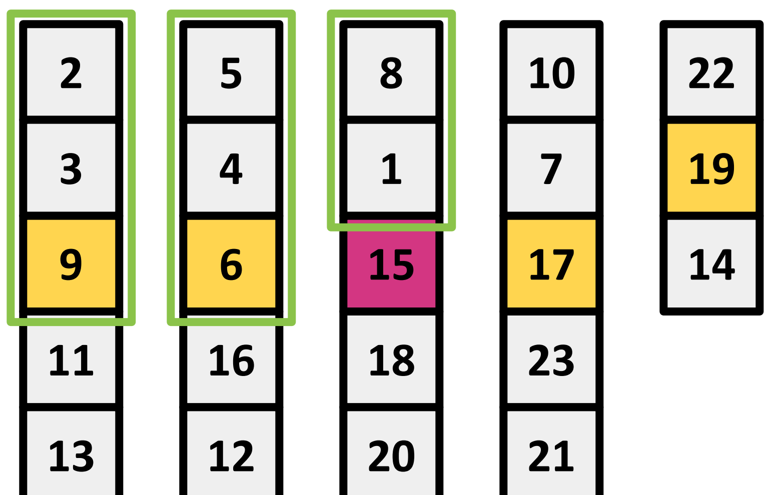
Analyzing Runtime

- 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?

2	5	8	10	22
3	4	1	7	19
9	6	15	17	14
11	16	18	23	
13	12	20	21	

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.

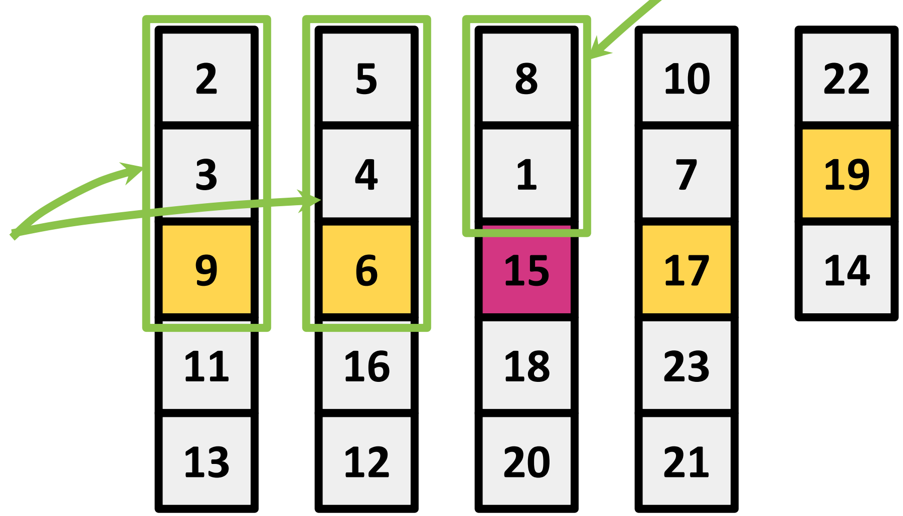


Analyzing Runtime

- 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.

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**.



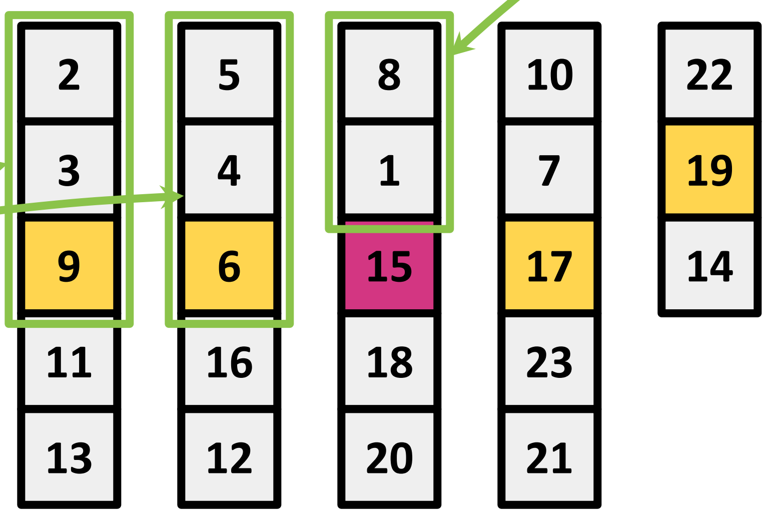
Analyzing Runtime

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

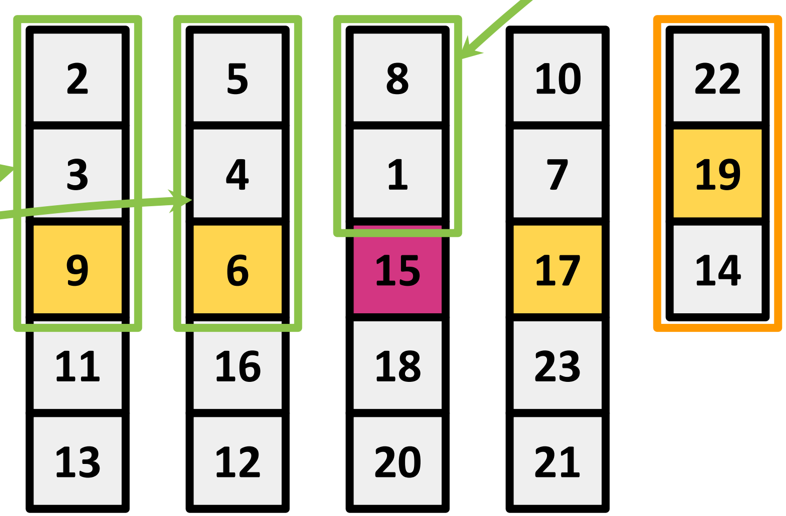
- As a function of n (the size of the original list), how many elements are guaranteed to be **<u>smaller</u>** than the median of medians?
  - Let $g = \lceil n/5 \rceil$ represent the number of groups.
  - **<u>At least</u>** $3 \cdot (\lceil g/2 \rceil - 1 - 1) + 2$ elements.

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**.

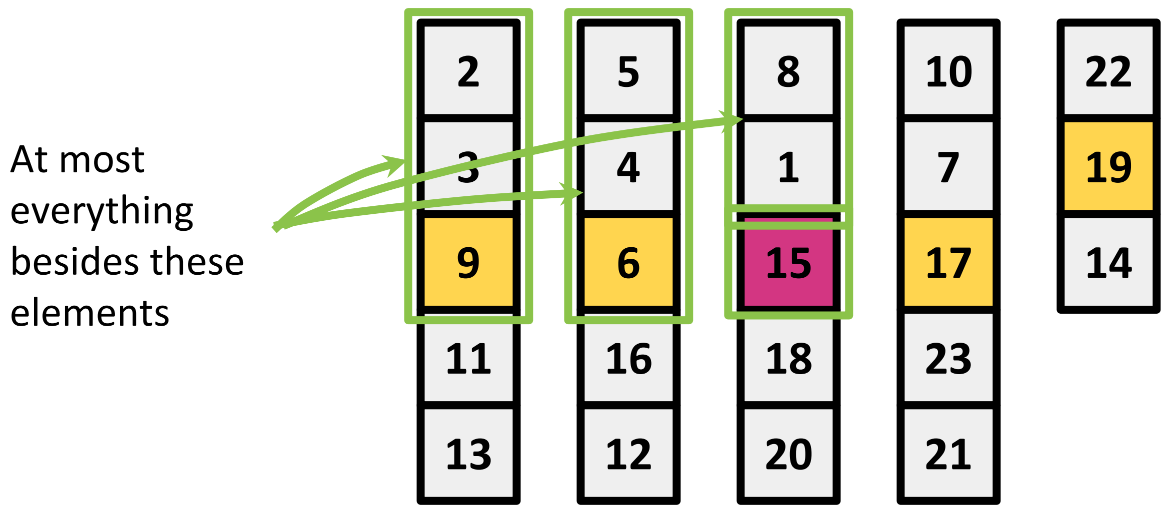
3 elements from each small group with a **median** smaller than the **median of medians**.



Analyzing Runtime

- If **at least** 3·([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)

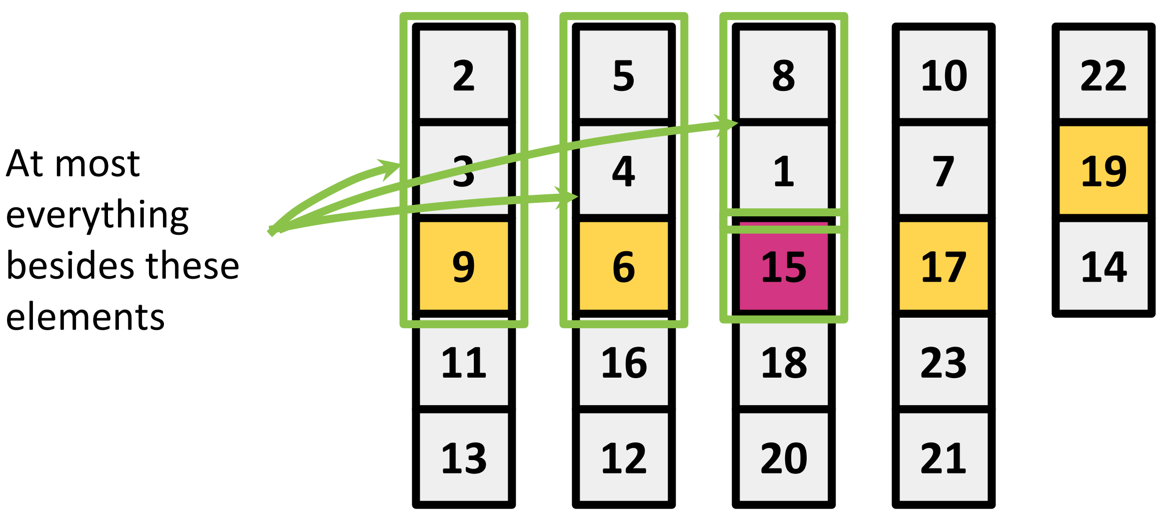
n-1 is for all of the elements except for the median of medians.



Analyzing Runtime

- If **<u>at least</u>** 3·([g/2] - 2) + 2 elements are guaranteed to be **<u>smaller</u>** than the median of medians, **<u>at most</u>** how many elements are **<u>larger</u>** than the median of medians?
  - **<u>At most</u>** n - 1 - (3·([g/2] - 2) + 2) ≤ $\boxed{\textbf{7n/10 + 3} \text{ elements.}}$

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



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)`** $\leq 7n/10 + 3$

$3n/10 - 4 \leq$ **`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.

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

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

- We can just as easily show the inverse.

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

Linear-Time Selection

```
// Returns k'th (k = 0, 1, ..) smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int select(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]
        int pivot = Median_of_median(arr, l, r);
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, pivot);

        // If position is same as k
        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)  // If position is more, recur for left
            return select(arr, l, pos-1, k);
        else // Else recur for right subarray
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

- What's the recurrence relation?

Analyzing Runtime

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

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = Median_of_median(arr, l, r);
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

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

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = Median_of_median(arr, l, r);
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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)**

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = Median_of_median(arr, l, r);
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

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

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = Median_of_median(arr, l, r);
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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**!

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = Median_of_median(arr, l, r);
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

Analyzing Runtime

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

```
int select(int arr[], int l, int r, int k)
{
    if (k >= 0 && k < r - l + 1)
    {
        int n = r-l+1;
        int pivot = Median_of_median(arr, l, r);
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k)
            return arr[pos];
        if (pos-l > k)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

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)

$\longrightarrow$ **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**
  - $T(n) = T(n/5) + T(7n/10) + dn$
    $\le C(n/5) + C(7n/10) + dn$
    $= (C/5)n + (7C/10)n + dn$
    $\le Cn$

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

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

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

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

- **Inductive hypothesis** $T(k) \le \max\{7, 10d\}k$ for all $1 \le k < n$.
- **Base case** $T(k) \le \max\{7, 10d\}k$ for all $k \le 100$.
- **Inductive step**
  - $$\begin{aligned} T(n) &= T(n/5) + T(7n/10) + dn \\ &\le \max\{7, 10d\}(n/5) + \max\{7, 10d\}(7n/10) + dn \\ &= (\max\{7, 10d\}/5)n + (7\max\{7, 10d\}/10)n + dn \\ &\le \max\{7, 10d\}n \end{aligned}$$
- **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.
1. 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
  - Linear-time selection
    - ~~Proving correctness~~ **Done!**
    - ~~Proving runtime with recurrence relations~~ **Done!**
  - *Problems: selection*
  - *Algorithms: Select*
  - Reading: CLRS 9

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.
  Runtime **O(nlog(n))**.

1	64	9	49	16	4	0	25	36	81

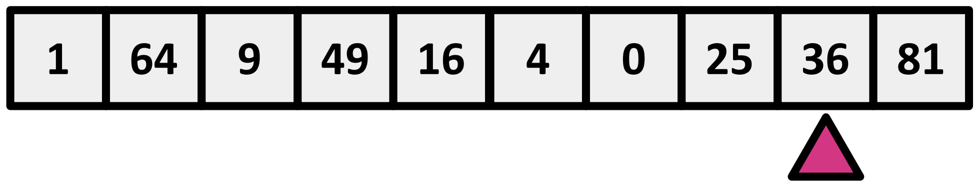
k=3

↓

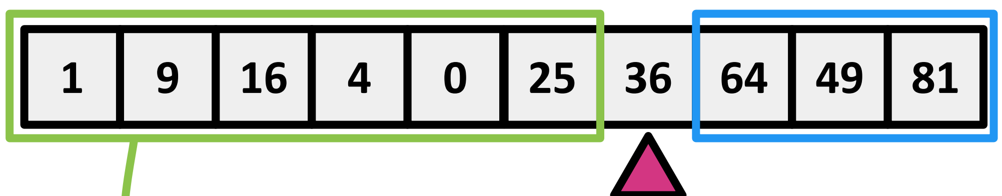
0	1	4	9	16	25	36	49	64	81

Linear-Time Selection

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



Select a pivot at random (for now)



Partition around the pivot, such that all elements to the left are less than it and all elements to the right are greater than it
(Notice that the halves remain unsorted.)

Find element k=3 in this half since 36 occupies index 6 and k=3 < 6.

Analyzing Runtime

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

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

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

```
int select(int arr[], int l, int r, int k)
{
    if (k > 0 && k <= r - l + 1)
    {
        int n = r-l+1;
        int pivot = median_of_medians(arr, l, r);
        int pos = partition(arr, l, r, pivot);

        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)
            return select(arr, l, pos-1, k);
        else
            return select(arr, pos+1, r, k-pos+l-1);
    }
}
```

How to pick the pivot?

- **Idea #1: choose a random pivot**
  - Unlucky case: **`len(left)`** = n - 1 and **`len(right)`** = 1 or vice versa
  - $T(n) \leq T(n-1) + O(n)$
  - Worst-case runtime $\mathbf{\Theta(n^2)}$
- **Idea #2: choose a pivot that divides the input list in half (the median)**
  - **`len(left)`** = **`len(right)`** = $(n-1)/2$
  - $T(n) \leq T(n/2) + O(n)$
  - Worst-case runtime $\mathbf{\Theta(n)}$
  - We do not know how to find the median in linear time
- **Idea #3: find a pivot “close enough” to median**
  - $3n/10 <$ **`len(left)`**, **`len(right)`** $< 7n/10$.
  - $T(n) \leq T(7n/10) + O(n)$
  - Worst-case runtime $\mathbf{\Theta(n)}$

Today's Outline

- Divide and Conquer I
  - ~~Proving correctness with induction~~ **Done!**
  - ~~Proving runtime with **recurrence relations**~~ **Done!**
  - ~~Proving the **Master method**~~ **Done!**
  - ~~Learn the **Substitution method**~~ **Done!**
- Divide and Conquer II
  - *Problems: k^{th} number selection*
  - ~~*Algorithms: Linear-time selection*~~ **Done!**
  - Reading: CLRS 9.2 9.3