Divide and Conquer I

Summer 2018 • Lecture 06/28

Announcements

- No tutorial this Friday.
- Homework 0
  - You might have noticed a *.tex file in **`hw0.zip`**.
  - Again, no submission required; worth 0% of your grade. But you should still attempt it before solutions are released next Tues 7/3.
- cs161.stanford.edu redirects to our website, finally!

Course Overview

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

Today's Outline

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

Recall Last Time...

- 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 about today...

- 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 about today...

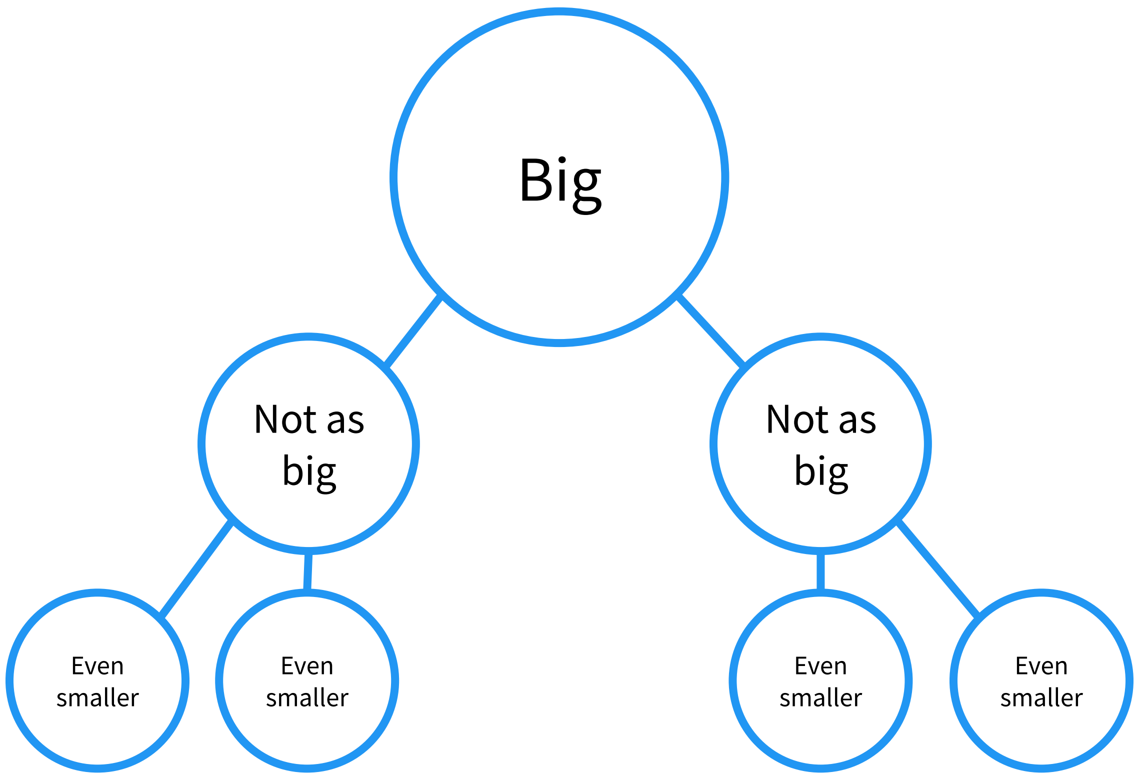
- Can we do better than insertion sort?
- Mergesort uses divide-and-conquer.
  - **Does this actually work?**
    - We will revisit proofs by induction.
  - **Is it fast?**
    - We will talk about recurrence relations.

These are the same questions we asked about insertion sort!

Mergesort

Divide and Conquer

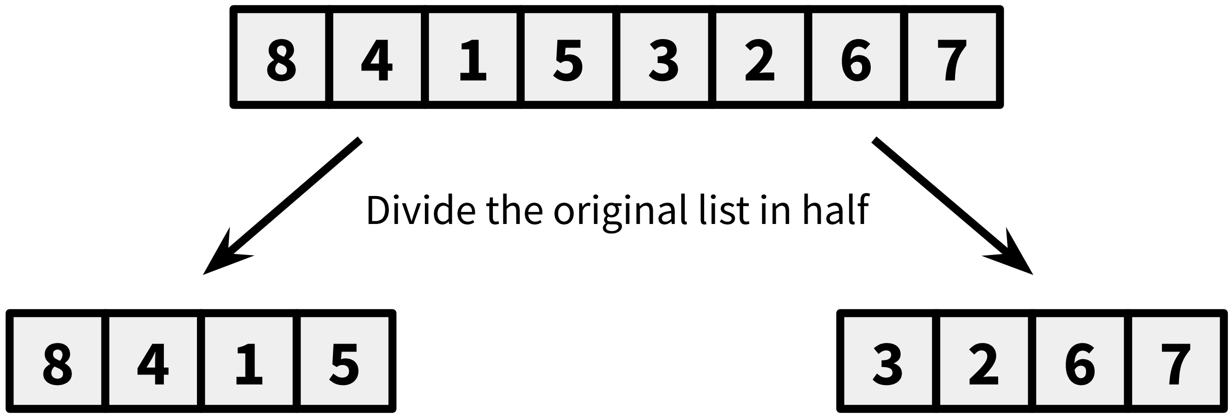
- **Divide** Break the current problem into smaller problems.
- **Conquer** Solve the smaller problems and collect the results to solve the current problem.



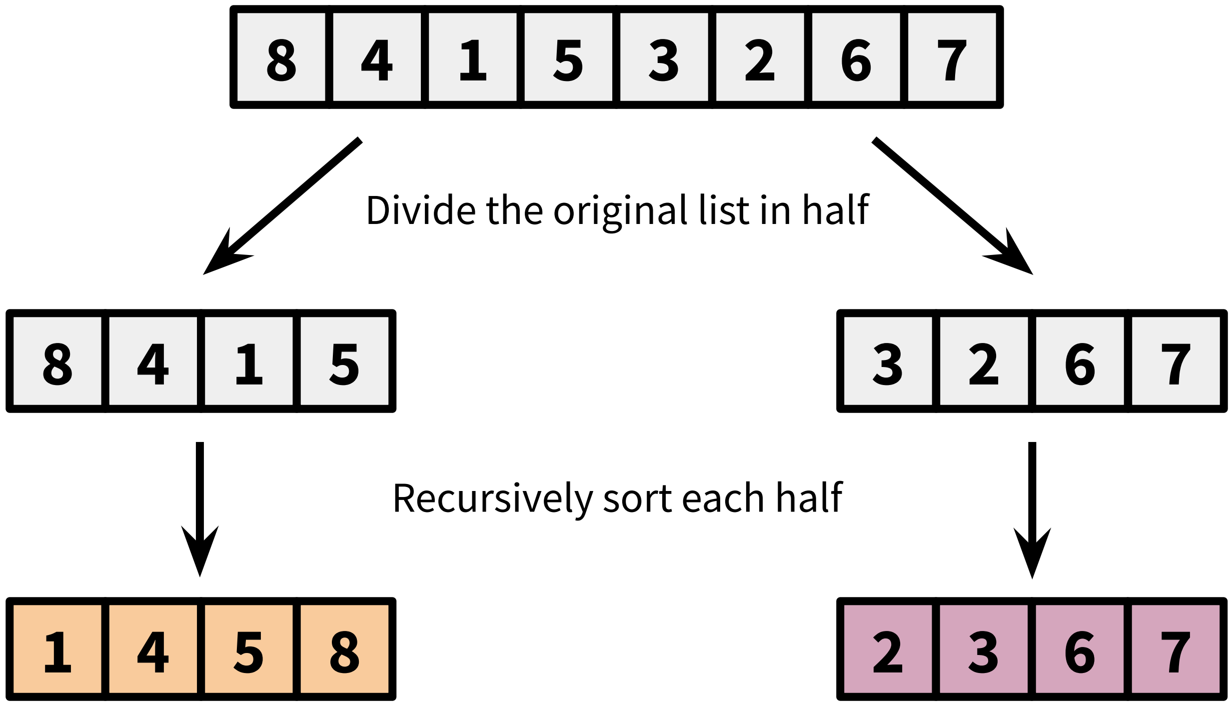
Mergesort

8	4	1	5	3	2	6	7

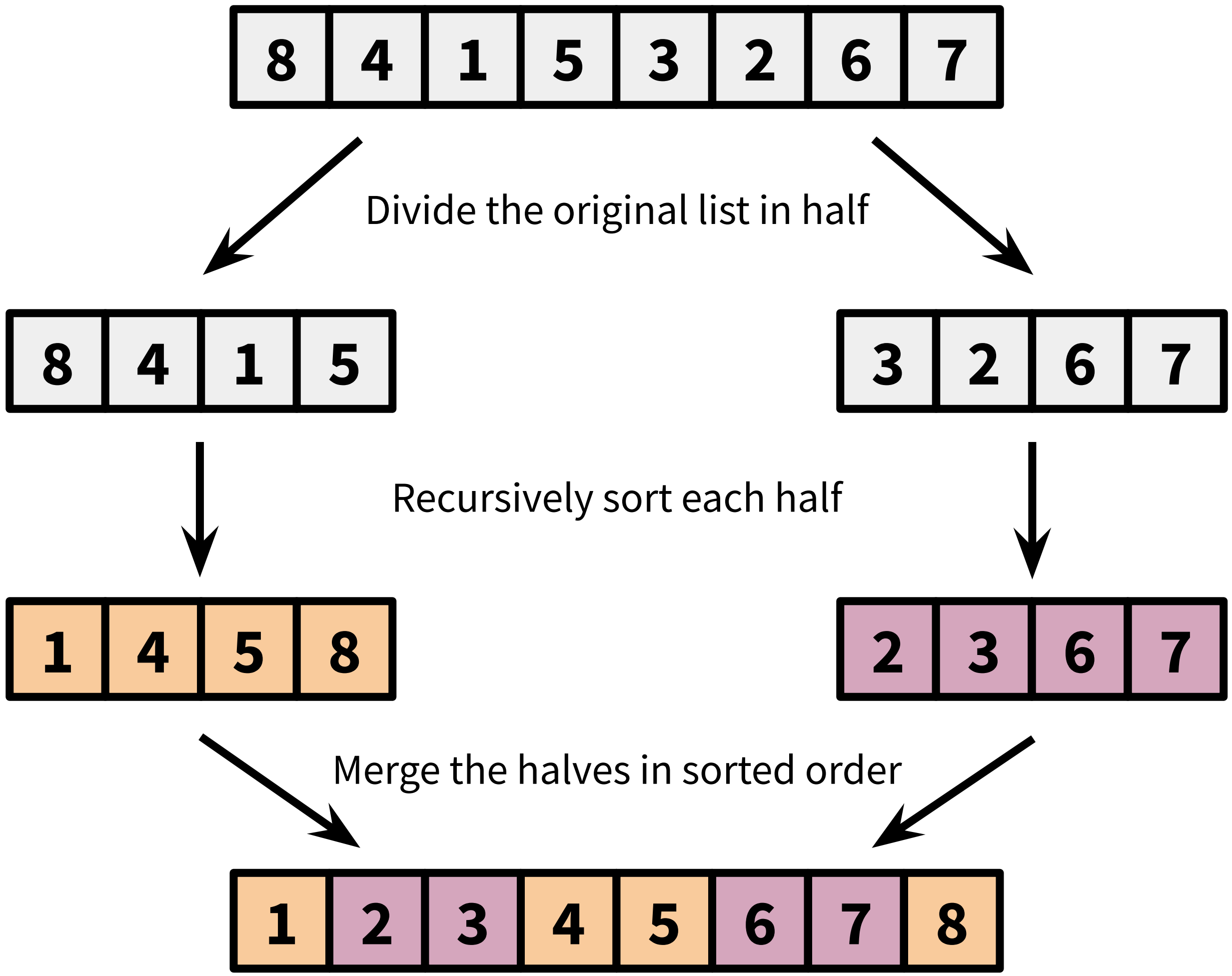
Mergesort



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

```
def mergesort(A):
```

Mergesort

```
def mergesort(A):
  if len(A) <= 1:
    return A
```

Mergesort

```
def mergesort(A):
  if len(A) <= 1:
    return A
  L = mergesort(A[0:n/2])
```

Mergesort

```
def mergesort(A):
  if len(A) <= 1:
    return A
  L = mergesort(A[0:n/2])
  R = mergesort(A[n/2:n])
```

Mergesort

```
def mergesort(A):
  if len(A) <= 1:
    return A
  L = mergesort(A[0:n/2])
  R = mergesort(A[n/2:n])
  return merge(L, R)
```

Mergesort

```
def merge(L, R):
```

Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
```

Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
  while l_idx < len(L) and r_idx < len(R):
```

Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
  while l_idx < len(L) and r_idx < len(R):
    if L[l_idx] < R[r_idx]:
      result.append(L[l_idx])
      l_idx += 1
```

Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
  while l_idx < len(L) and r_idx < len(R):
    if L[l_idx] < R[r_idx]:
      result.append(L[l_idx])
      l_idx += 1
    else:
      result.append(R[r_idx])
      r_idx += 1
```

Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
  while l_idx < len(L) and r_idx < len(R):
    if L[l_idx] < R[r_idx]:
      result.append(L[l_idx])
      l_idx += 1
    else:
      result.append(R[r_idx])
      r_idx += 1
  result.extend(L[l_idx:len(L)])
  result.extend(R[r_idx:len(R)])
```

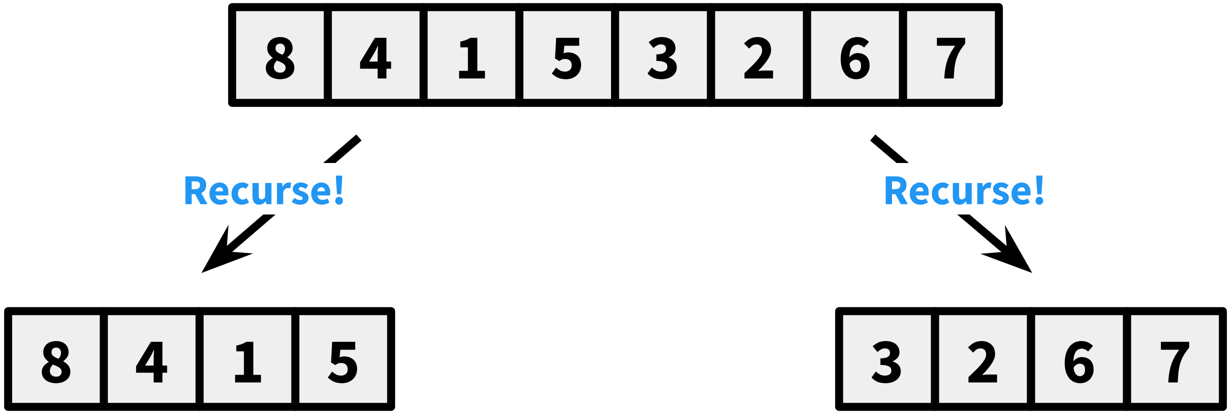
Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
  while l_idx < len(L) and r_idx < len(R):
    if L[l_idx] < R[r_idx]:
      result.append(L[l_idx])
      l_idx += 1
    else:
      result.append(R[r_idx])
      r_idx += 1
  result.extend(L[l_idx:len(L)])
  result.extend(R[r_idx:len(R)])
  return result
```

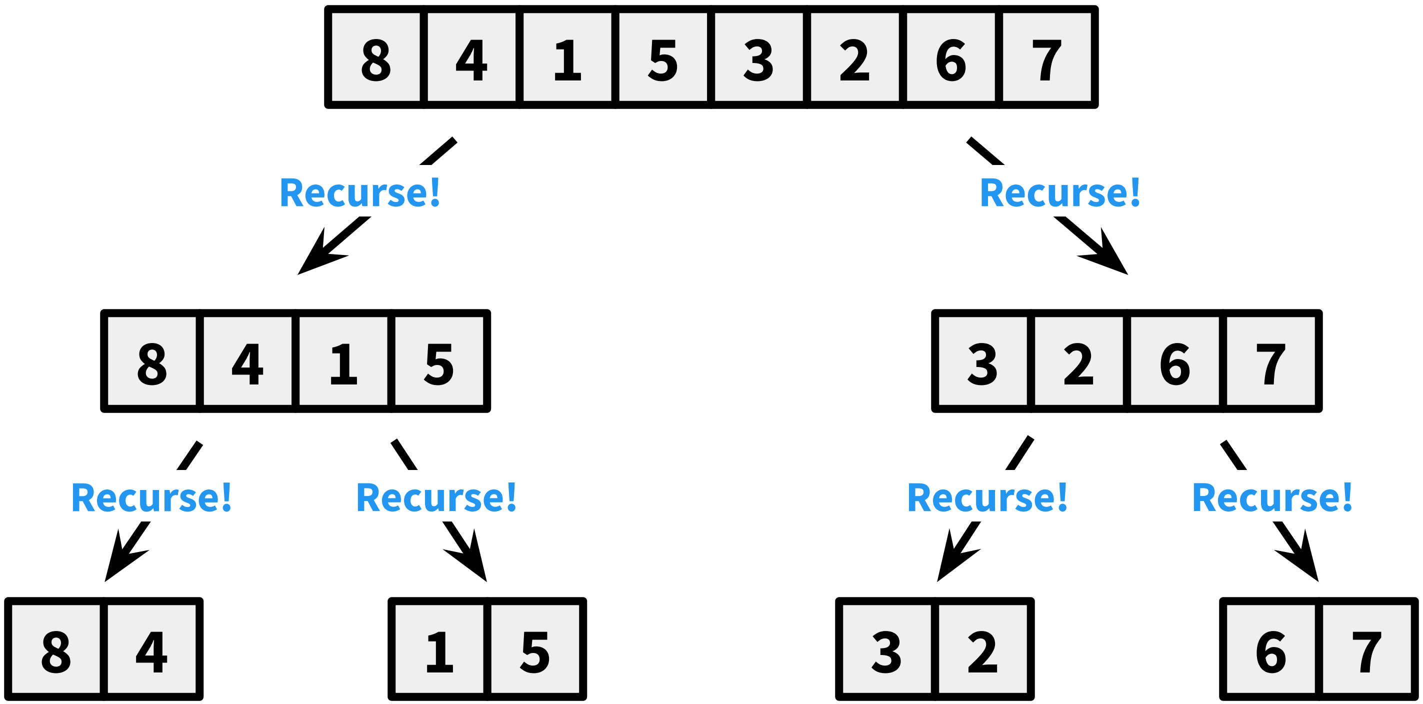
Mergesort

8	4	1	5	3	2	6	7

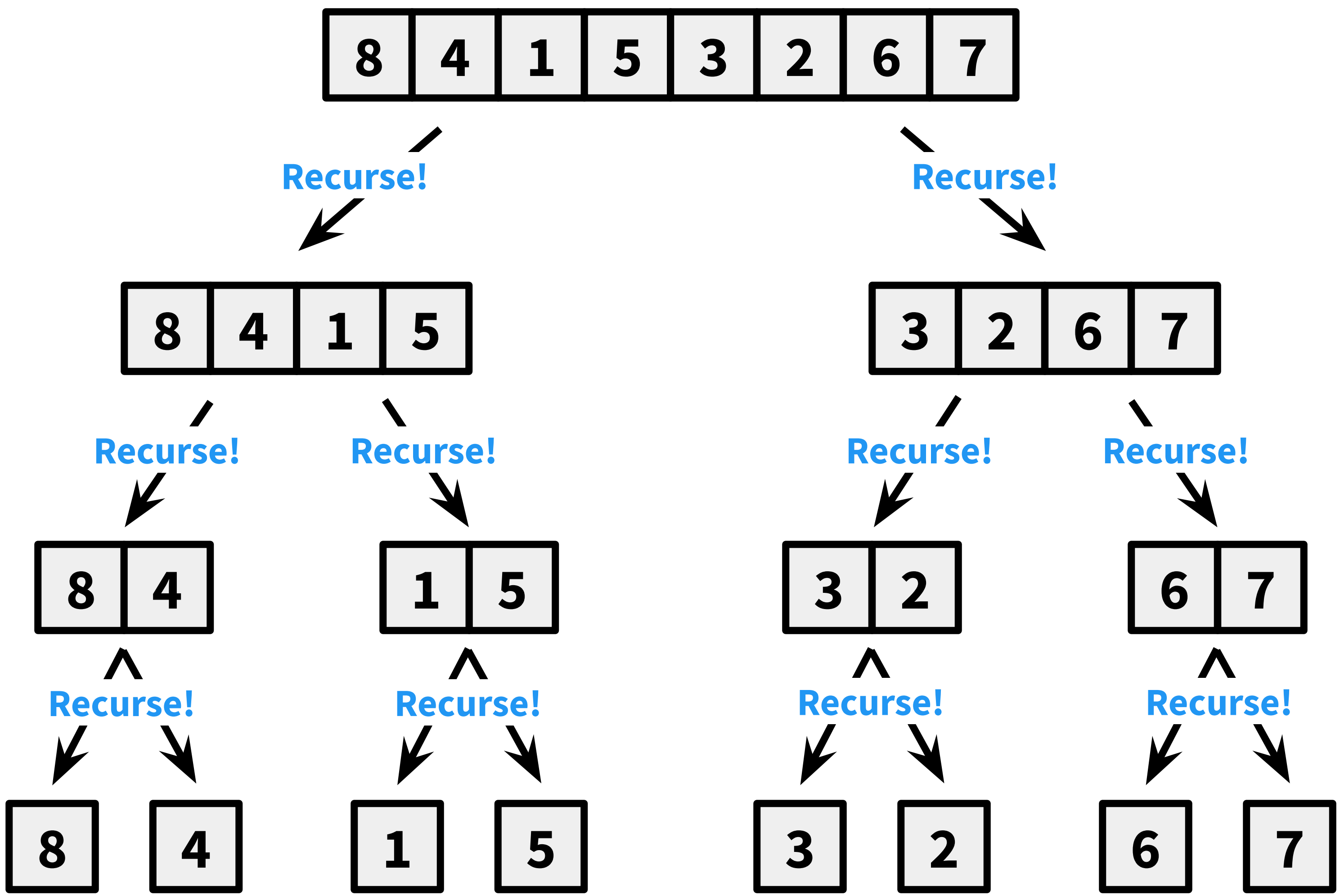
Mergesort



Mergesort


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

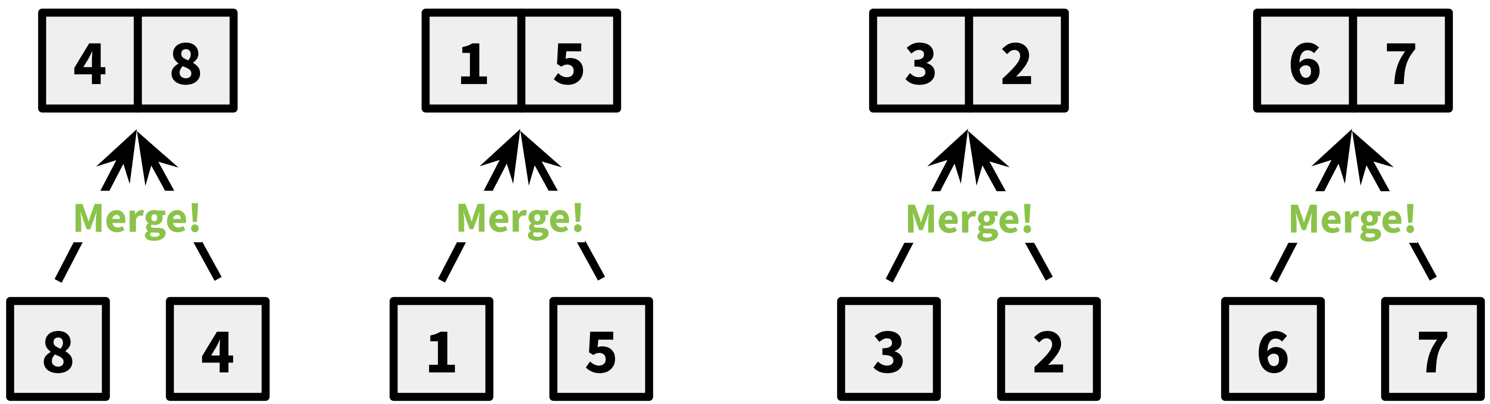
Mergesort


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

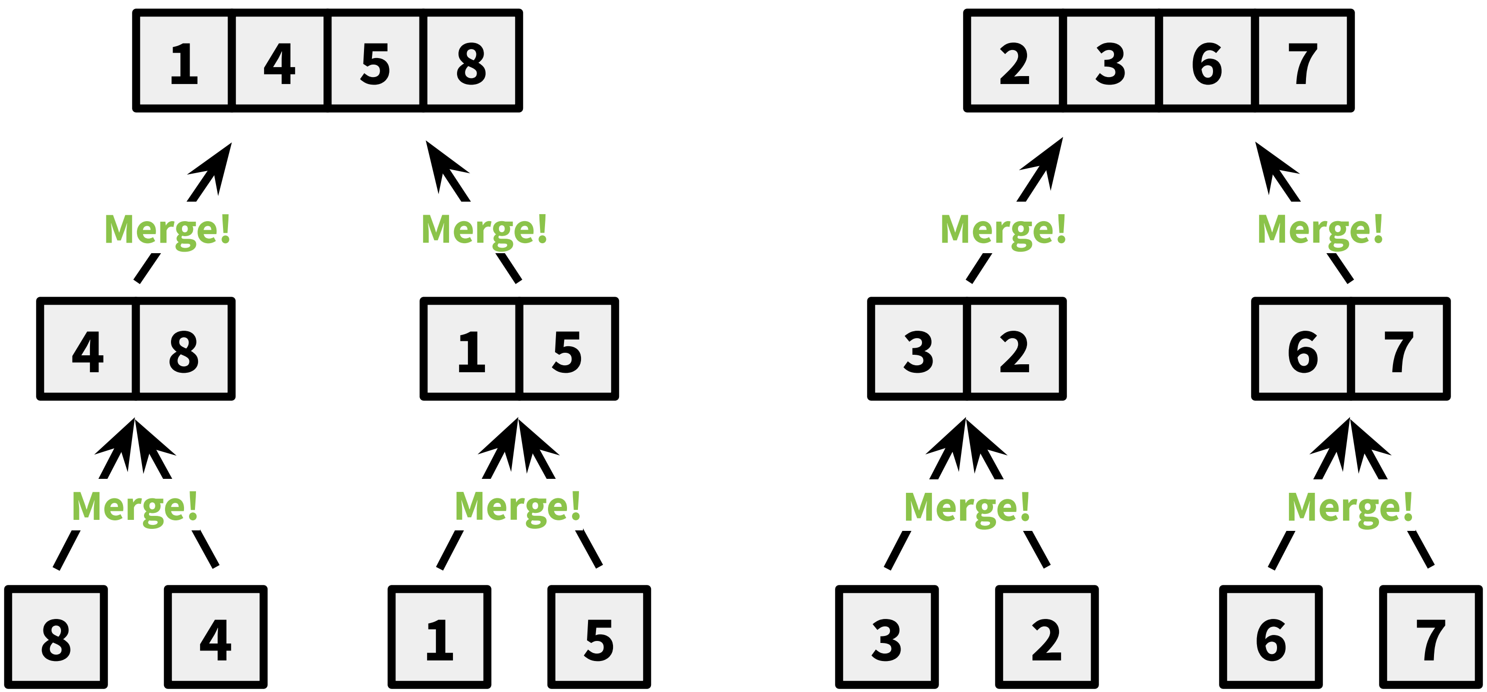
Mergesort

8	4	1	5	3	2	6	7

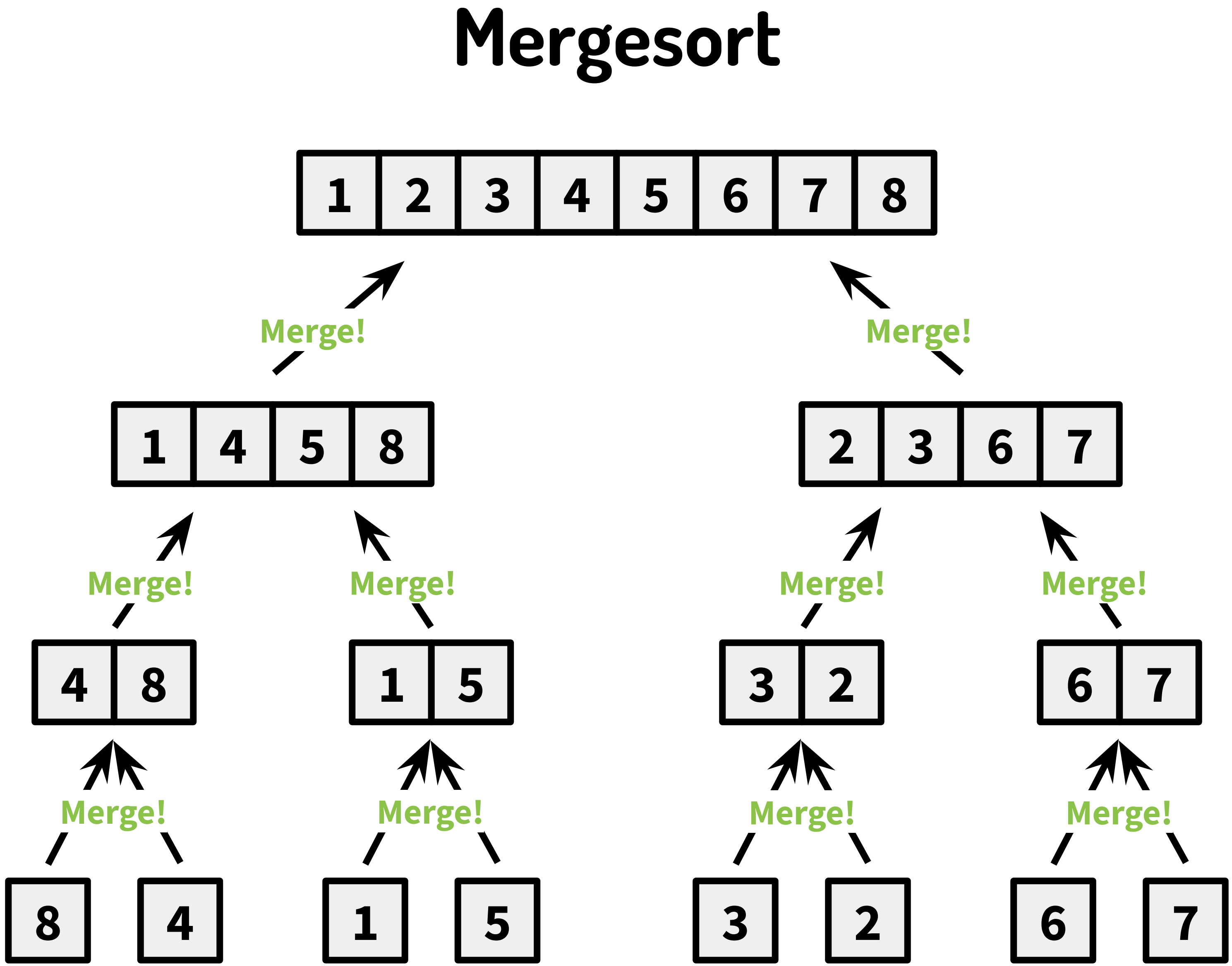
Mergesort



Mergesort



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


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
3 2
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?**

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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?**

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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.

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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.
  - **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.
  - **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)
```

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
```

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?**

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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?**

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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?**

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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`**?

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
  while l_idx < len(L) and r_idx < len(R):
    if L[l_idx] < R[r_idx]:
      result.append(L[l_idx])
      l_idx += 1
    else:
      result.append(R[r_idx])
      r_idx += 1
  result.extend(L[l_idx:len(L)])
  result.extend(R[r_idx:len(R)])
  return result
```

Mergesort

```
def merge(L, R):
  result = []
  l_idx, r_idx = (0, 0)
  while l_idx < len(L) and r_idx < len(R):
    if L[l_idx] < R[r_idx]:
      result.append(L[l_idx])
      l_idx += 1
    else:
      result.append(R[r_idx])
      r_idx += 1
  result.extend(L[l_idx:len(L)])
  result.extend(R[r_idx:len(R)])
  return result
```

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`**?

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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`**? **$\Theta(n)$**.

```
def mergesort(A):
  if len(A) <= 1:
    return A
  left = mergesort(A[0:n/2])
  right = mergesort(A[n/2:n])
  return merge(left, right)
```

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**!
  - $T(0) = \Theta(1)$
  - $T(1) = \Theta(1)$
  - $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

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.
  - 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 $\Theta(1)$ and $\Theta(n)$ 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** $c = \max\{c_1, c_2\}$.

$$\cancel{T(0) = \Theta(1)}$$
$$T(1) \leq c$$
$$T(n) \leq 2T(n/2) + cn$$

How do we translate this simplified recurrence relation to a 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

$$T(1) \leq c$$

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

Recursion Tree Method

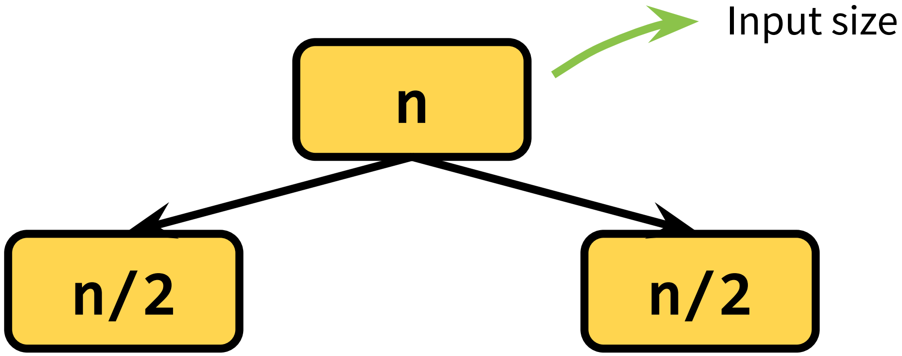
n

Input size

$T(1) \leq c$

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

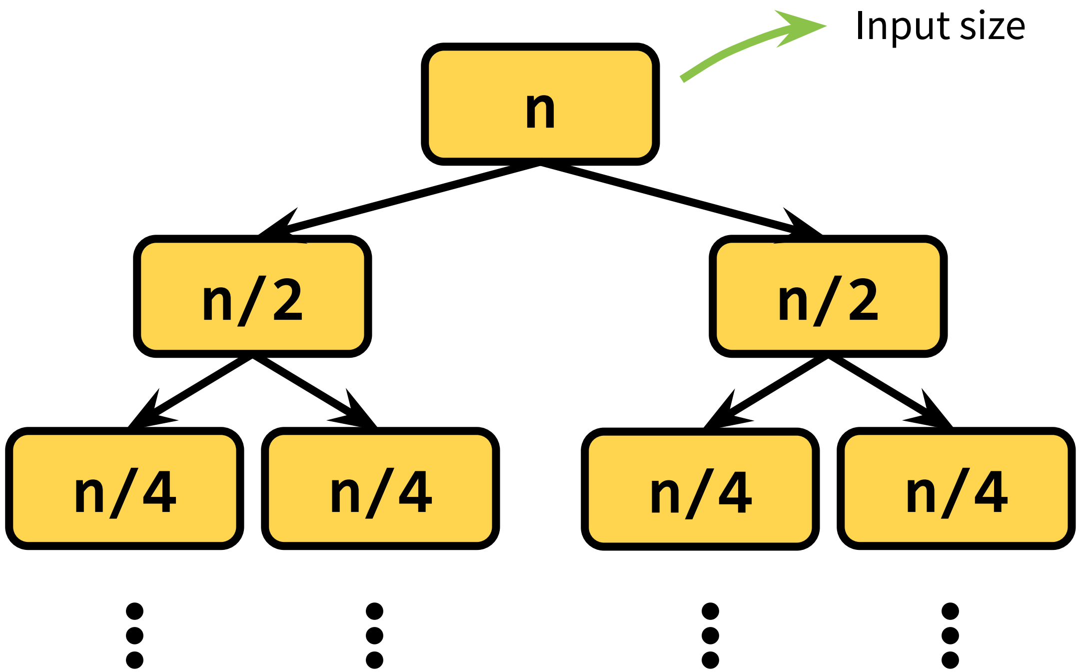
Recursion Tree Method



$T(1) \le c$

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

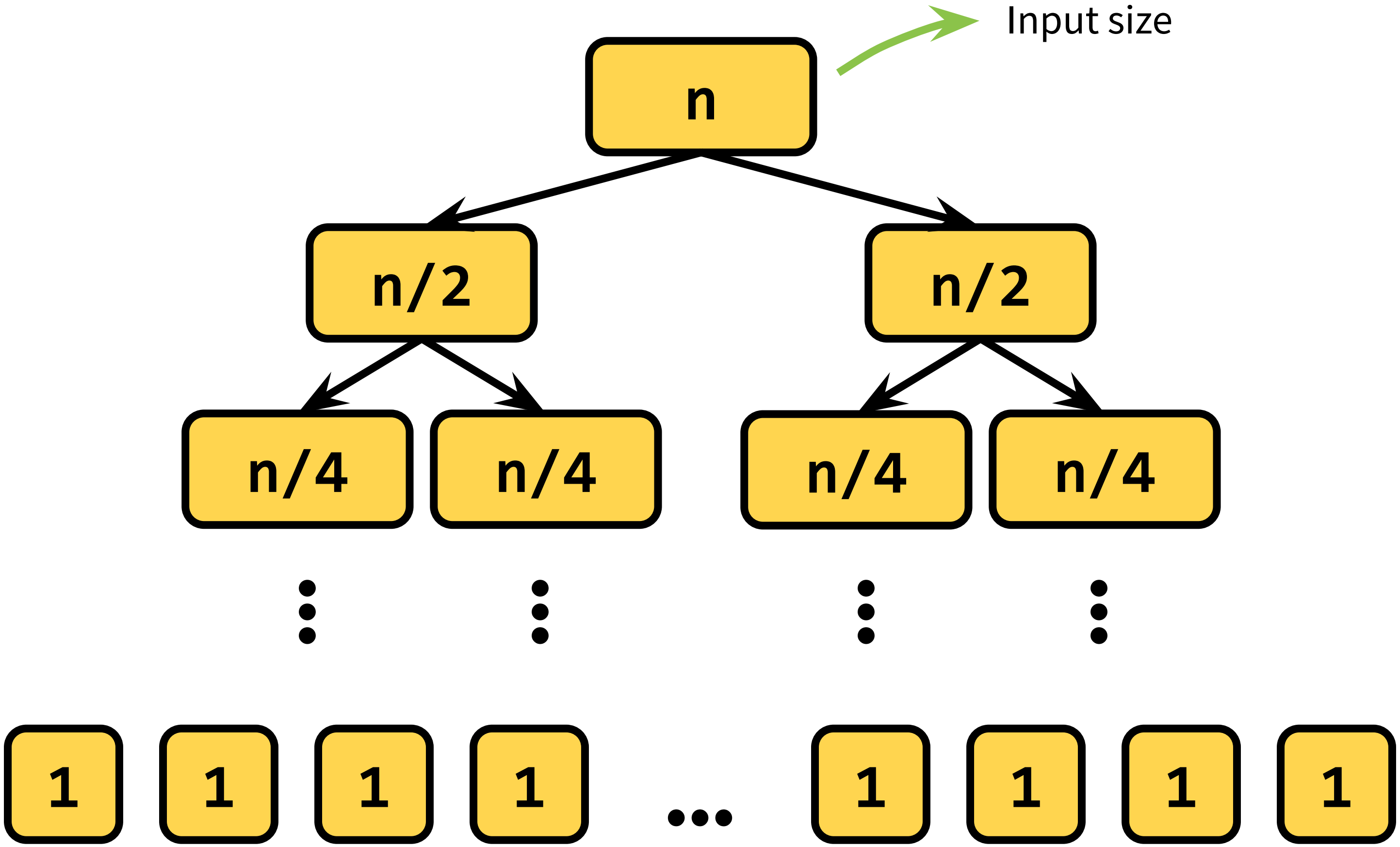
Recursion Tree Method



$T(1) \leq c$

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

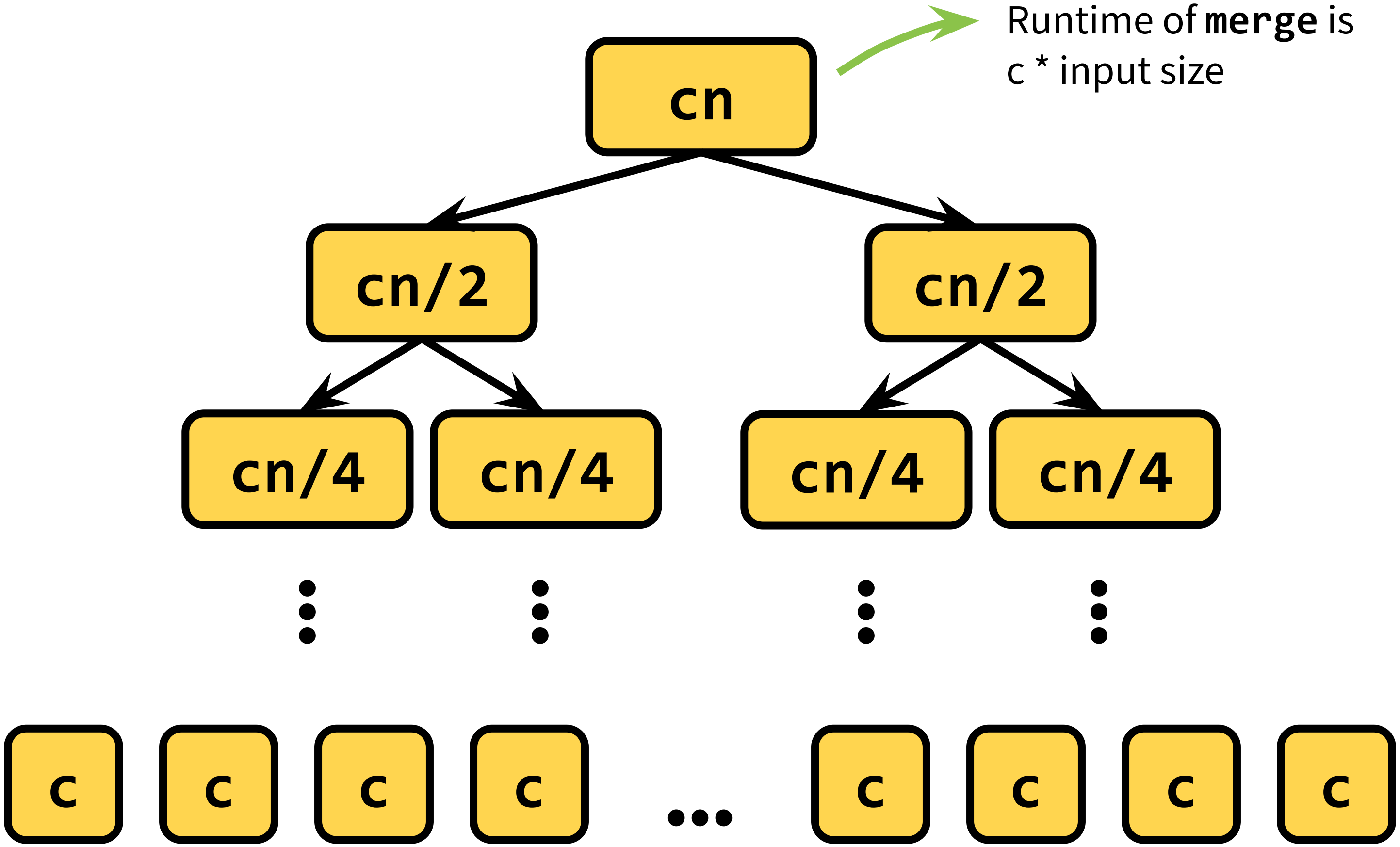
Recursion Tree Method



$T(1) \le c$

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

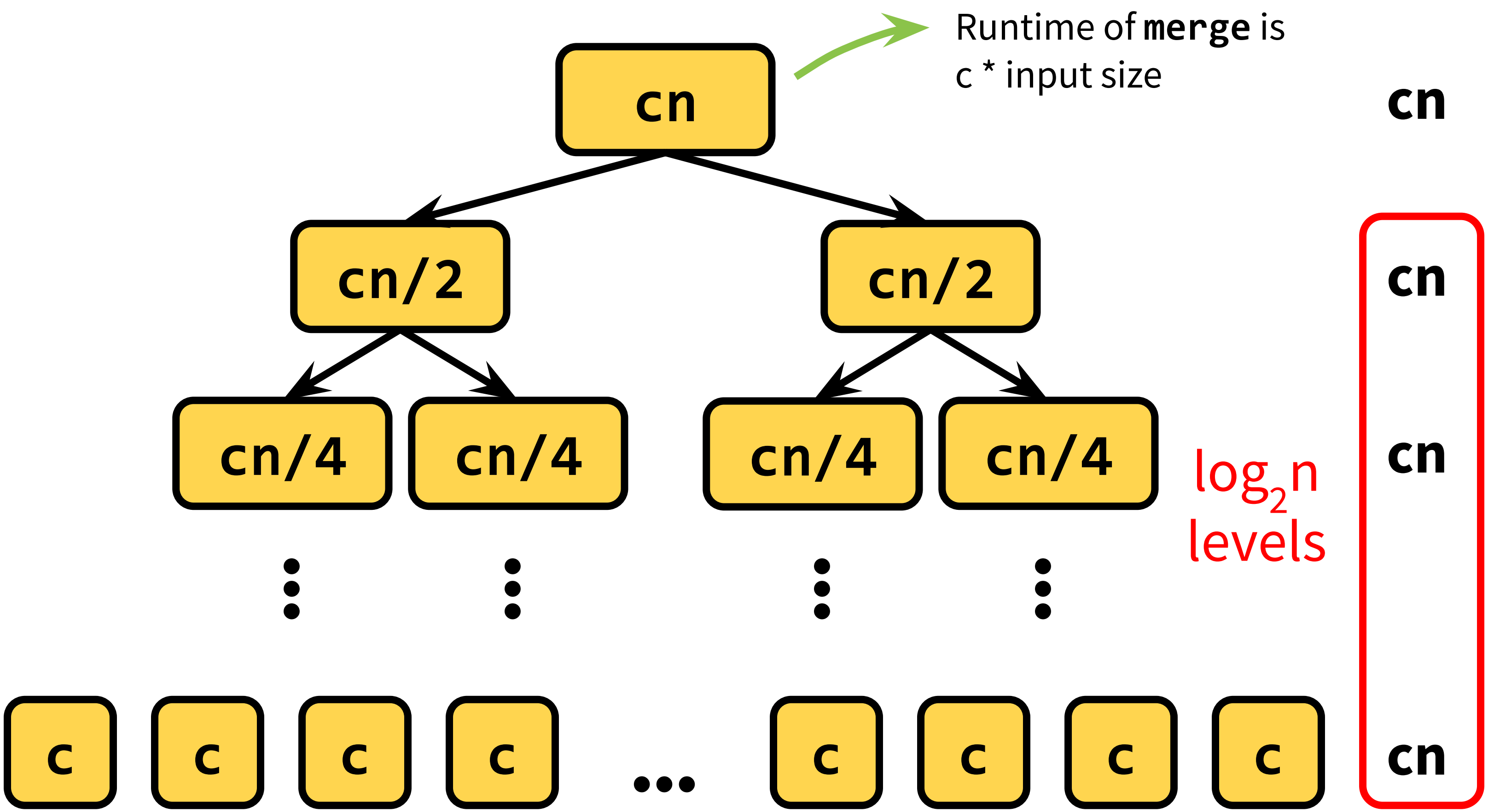
Recursion Tree Method



$T(1) \leq c$

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

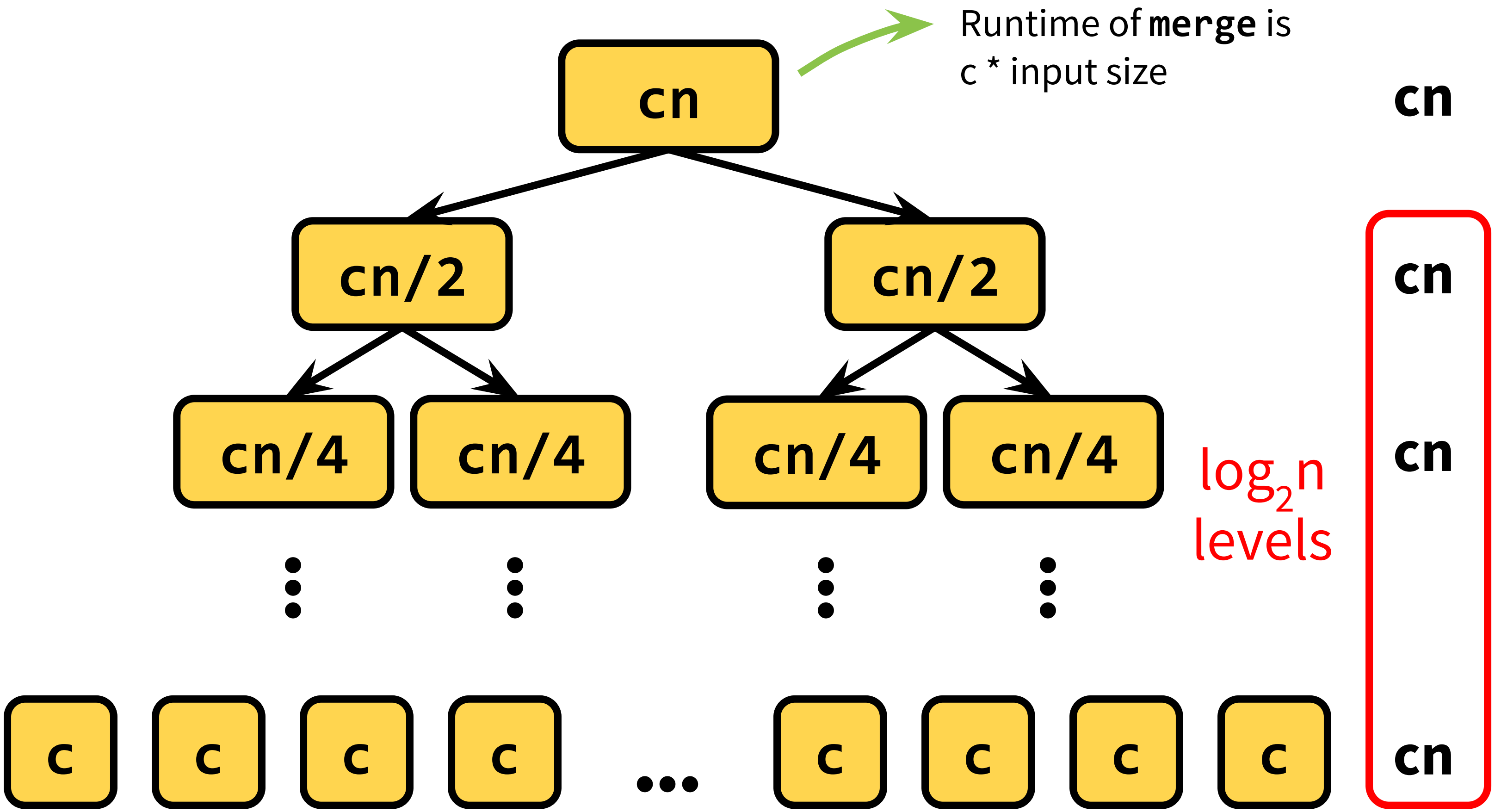
Recursion Tree Method



$T(1) \le c$

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

Recursion Tree Method

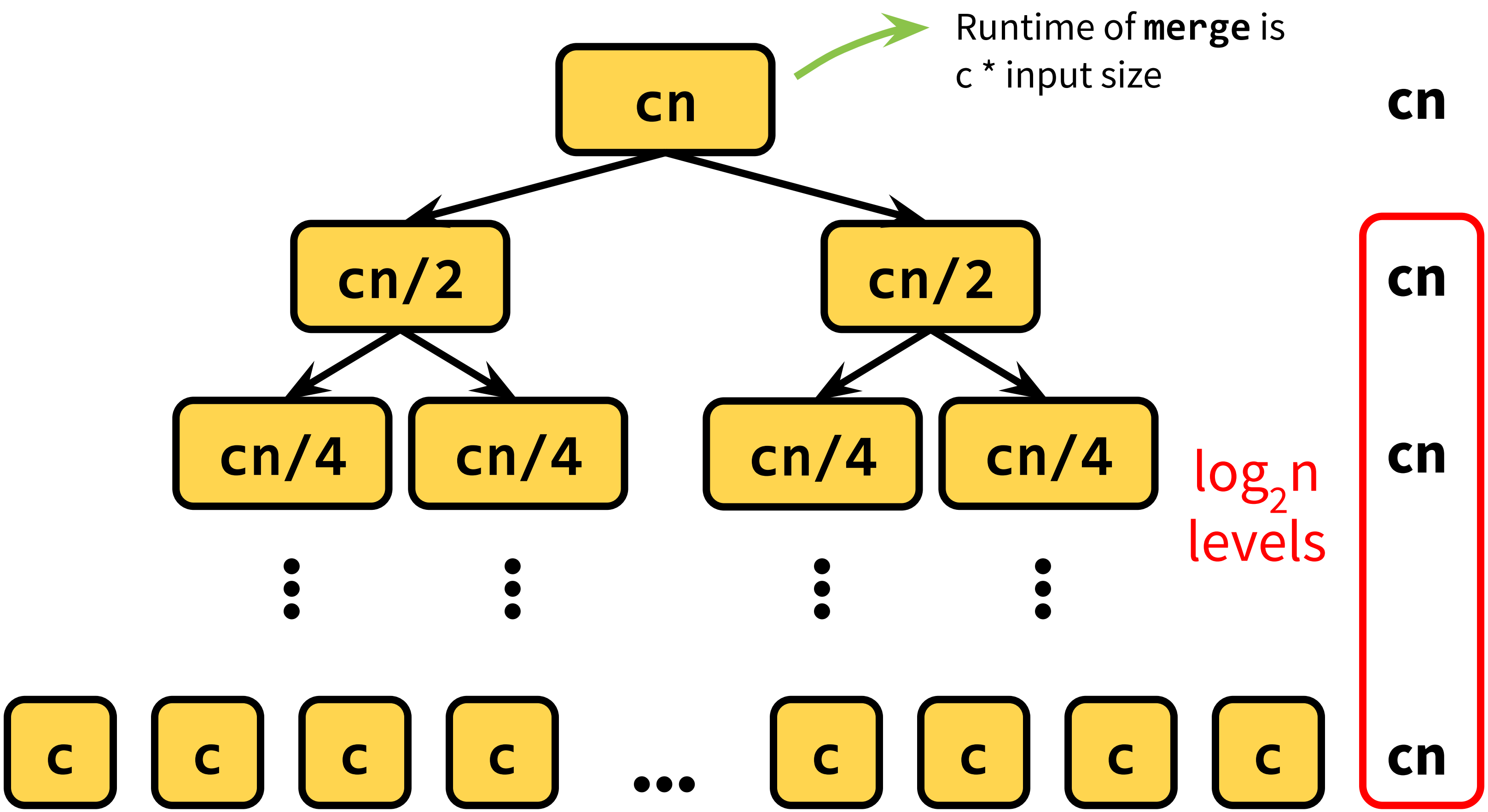


Runtime $cn \log_2 n + cn = O(n \log(n))$

$$T(1) \leq c$$

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

Recursion Tree Method



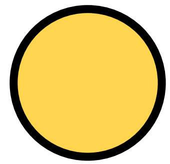
Runtime $cn \log_2 n + cn = \mathbf{O(n \log(n))}$

Use same reasoning for Ω

$T(1) \le c$

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

Recursion Tree Method



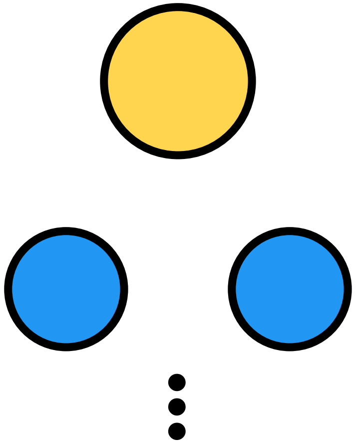
1 problem
of size cn

cn

$T(1) \leq c$

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

Recursion Tree Method



1 problem of size cn

cn

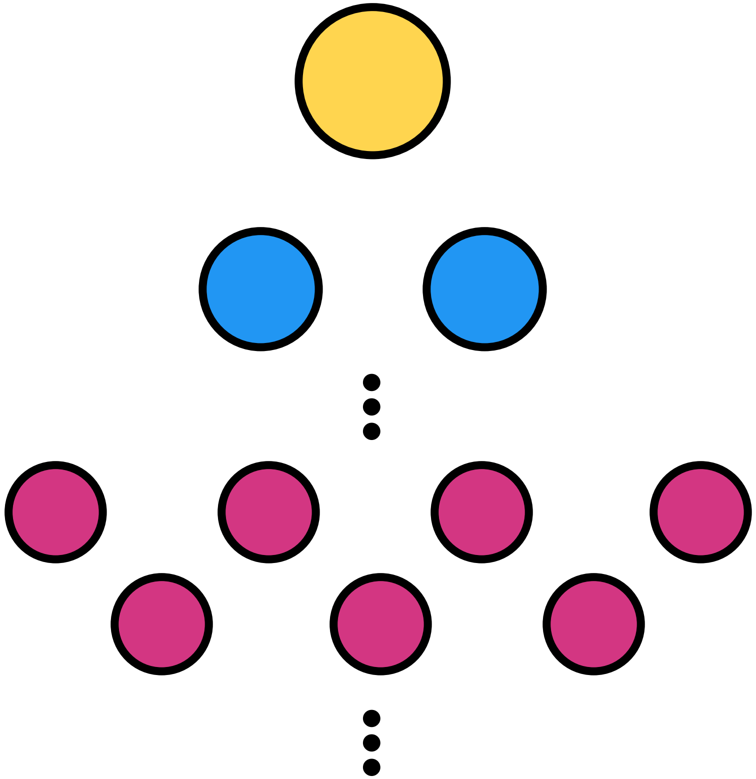
2 problems of size cn/2

cn

$T(1) \leq c$

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

Recursion Tree Method



1 problem
of size cn

cn

2 problems
of size cn/2

cn

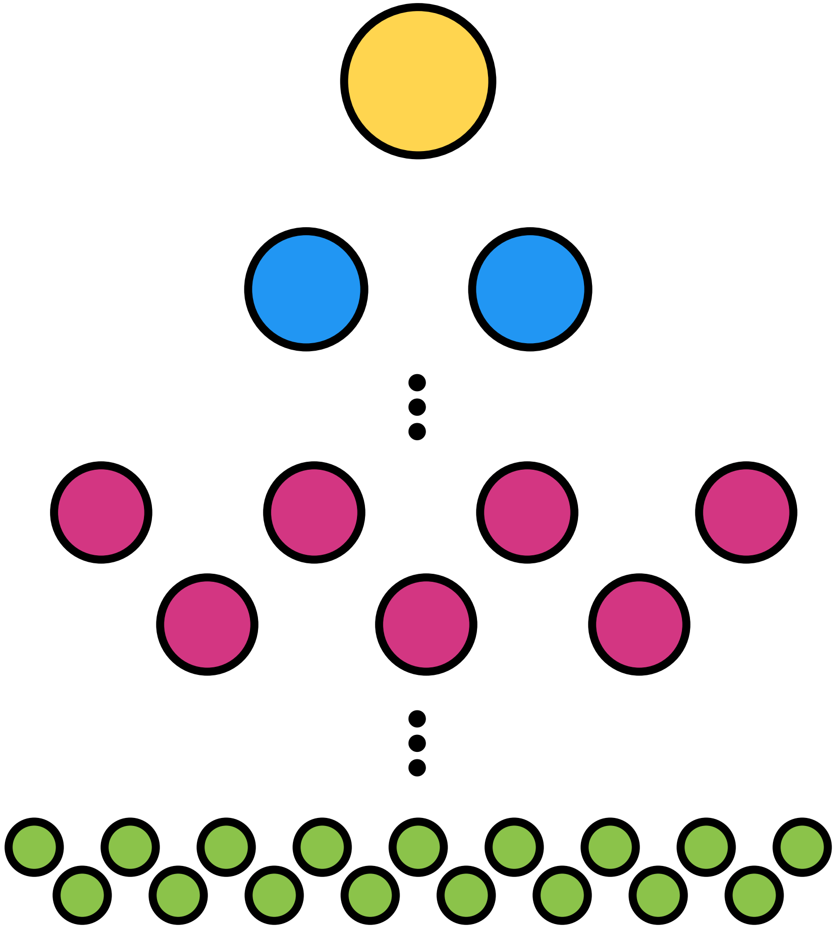
2^t problems
of size $cn/2^t$

cn

$T(1) \leq c$

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

Recursion Tree Method



1 problem of size cn — **cn**

2 problems of size cn/2 — **cn**

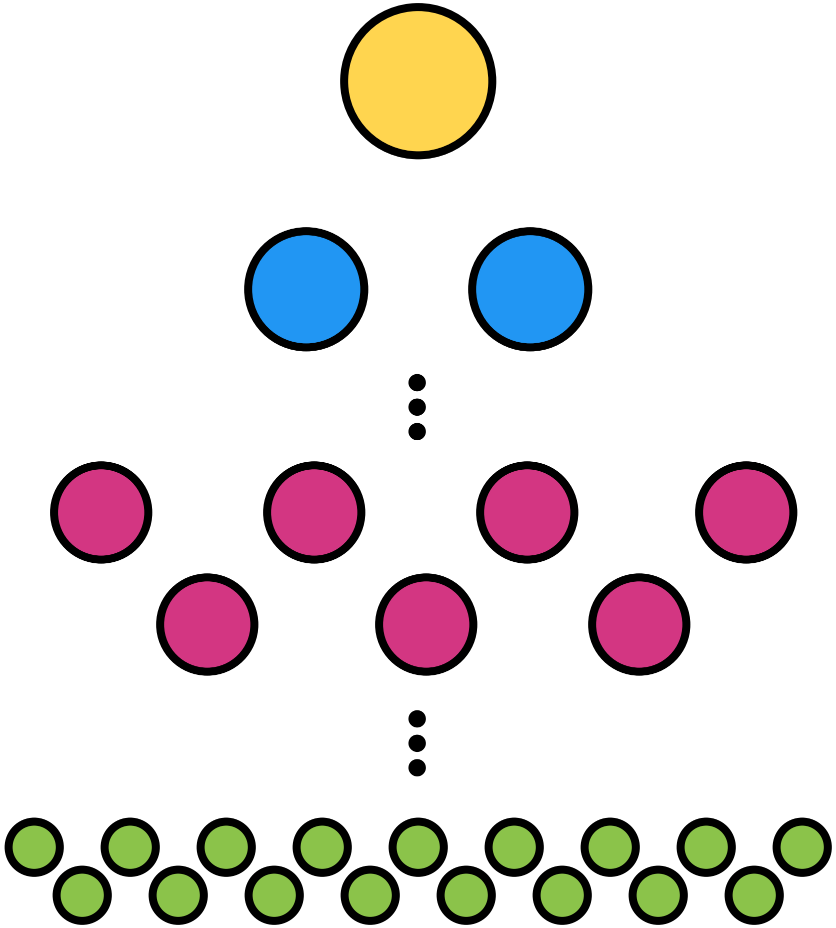
2^t problems of size $cn/2^t$ — **cn**

$2^{\log\_2(n)} = n$ problems of size c — **cn**

$T(1) \le c$

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

Recursion Tree Method



1 problem of size cn — **cn**

2 problems of size cn/2 — **cn**

2^t problems of size $cn/2^t$ — **cn**

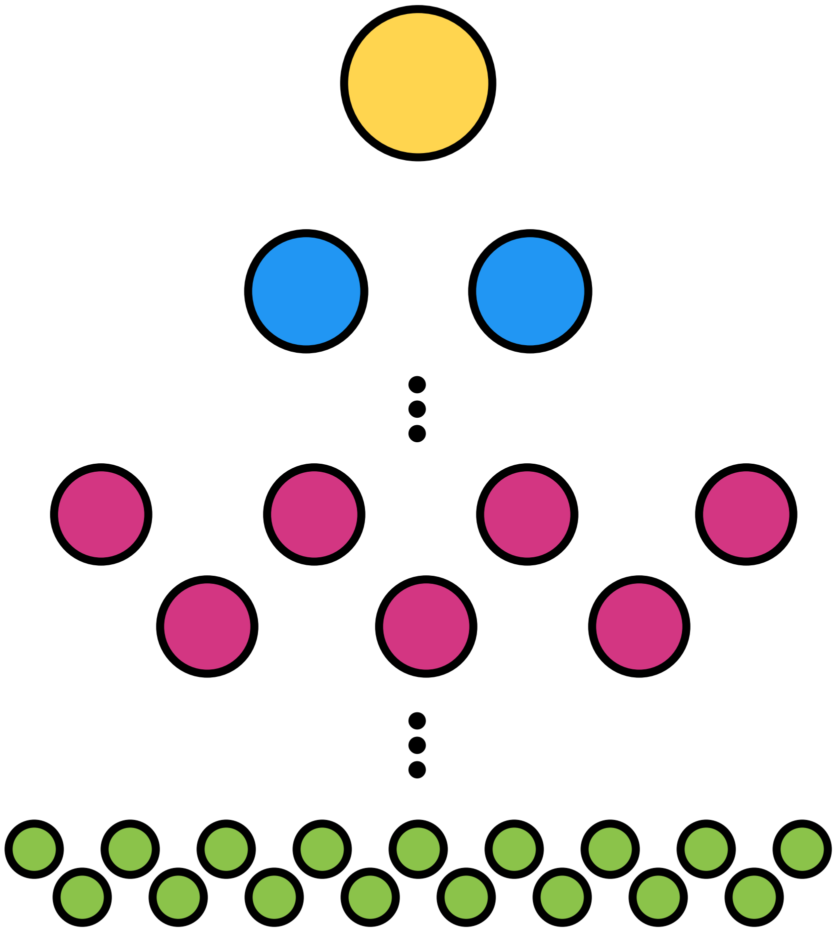
$2^{\log\_2(n)} = n$ problems of size c — **cn**

$\log_2 n$ levels

$T(1) \le c$

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

Recursion Tree Method



1 problem of size cn — **cn**

2 problems of size cn/2 — **cn**

2^t problems of size $cn/2^t$ — **cn**

$2^{\log\_2(n)} = n$ problems of size c — **cn**

$\log_2 n$ levels

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

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
  - 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

$T(1) \le c$

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

Iteration Method

- Apply the relationship until you see a pattern.

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

$$T(1) \le c$$

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

Iteration Method

- Apply the relationship until you see a pattern.

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

$$T(1) \le c$$

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

Iteration Method

- Apply the relationship until you see a pattern.

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

$T(1) \le c$

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

Iteration Method

- Apply the relationship until you see a pattern.

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

$T(1) \le c$

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

Iteration Method

- Apply the relationship until you see a pattern.

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

- What is k?

$T(1) \leq c$

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

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 $k = \log_2 n$.

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

$T(1) \le c$

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

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 $k = \log_2 n$.

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

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 $k = \log_2 n$.

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

$T(1) \le c$

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

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 $k = \log_2 n$.

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

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**
    - Substitution Method

5-min Break

Today's Outline

- Divide and Conquer I
  - ~~Proving correctness with induction~~ **Done!**
  - ~~Proving runtime with recurrence relations~~ **Done!**
  - Proving the Master 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**
  - Substitution Method

Master Method

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

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

$T(1) \le c$

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

where a is the number of subproblems,

b is the factor by which the input size shrinks, and

d parametrizes the runtime to create the subproblems and merge their solutions.

Master Method

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

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

$T(1) \leq c$

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

$a = 2, b = 2, d = 1$

where a is the number of subproblems,

b is the factor by which the input size shrinks, and

d parametrizes the runtime to create the subproblems and merge their solutions.

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.
- 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** **Θ(n log(n))**.
  - **Iteration method** **Θ(n log(n))**.
  - **Master method** **Θ(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 $\Theta(n \log(n))$.**
    - **Iteration method $\Theta(n \log(n))$.**
    - **Master method $\Theta(n \log(n))$.**
    - **Substitution Method Next time!**

Mergesort

```
def mergesort(A):
  if len(A) <= 1:
    return A
  L = mergesort(A[0:n/2])
  R = mergesort(A[n/2:n])
  return merge(L, R)
```

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

Mergesort

```
def mergesort(A):
  if len(A) <= 1:
    return A
  L = mergesort(A[0:n/2])
  R = mergesort(A[n/2:n])
  return merge(L, R)
```

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

It's the same as the worst-case runtime!

Today's Outline

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