# Divide and Conquer I: Sorting

CS/MCS 401: Computer Algorithms I
Lecture 5
Professor: Jonathan Liu

**Sit next to someone!** If you don't do it now, you'll have to move later.

**Introduce yourself!**
Icebreaker: How competitive are you?

# Lesson Goals

Today we're entering the first of our many paradigm-centered units.

The first unit is Divide and Conquer. Today you will learn what algorithms in this paradigm look like, and see how we write proofs of correctness about them.

# A Quick Course Overview

# Algorithm Paradigms

| Unit | Paradigm |
|---|---|
| 2 | Divide and Conquer |
| 3 | Graph Algorithms |
| MIDTERM | |
| 4 | Greedy Algorithms |
| 5 | Dynamic Programming |
| 6 | Intractable Problems |
| FINAL | |

We are here

# Algorithm Paradigms

For each paradigm, **the overarching goal is a skill: to be able to design and understand algorithms following the paradigm**.

**Lecture:** You will see a few well-known examples of algorithms that follow this paradigm.

**Homework:** You will mostly get problems where you use the paradigm to **design new algorithms**.

**Exams:** You will get questions asking you to **design or evaluate algorithms** following these paradigms.

# Mergesort

# Sorting

Sorting is important!

- We click “sort” all the time on all kinds of data: playlists, emails, really anything worth displaying
- We’ll see throughout this course that sorting can help us make better sense of the input data

# Sorting Exercise 1 – Find a partner!

When the next slide comes up, one person should sort the top row and the other should sort the bottom row. Decide now who will be each. When you're done sorting, raise your hand. Faster sorter wins!

Catch: You have to write down, in advance, which sorting algorithm you're going to use. Write out the whole pseudocode, and analyze its runtime.

# Sorting Exercise 1

S O F Y M C Q U H J

A N K Z W R G T D I

# Sorting Exercise 1

Now, compare your sorting algorithms. What was good about them for this task, and what was bad?

Did it work for letters? Most of the time, we're not sorting numbers!

# Sorting Exercise 2

When I say go, you're going to merge your two lists into one big *sorted* list.

Start by designing a plan together to do so, and analyze the runtime. *You may not look at each other's lists.*

# Sorting Exercise 2

Go!

# Sorting Exercise 2

What did you do?

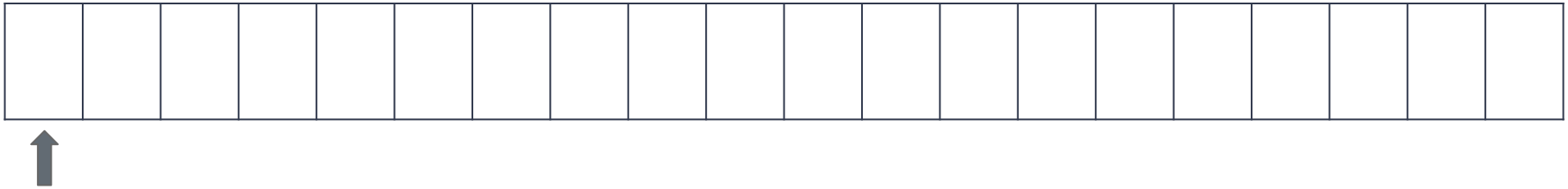
# Sorting Exercise 2

Merging sorted lists can be done as follows:

(1) Compare the first item in each list
(2) Add the “smaller” one to the combined list.
(3) Remove that item from its original list

C F H J M O Q S U Y

A D G I K N R T W Z



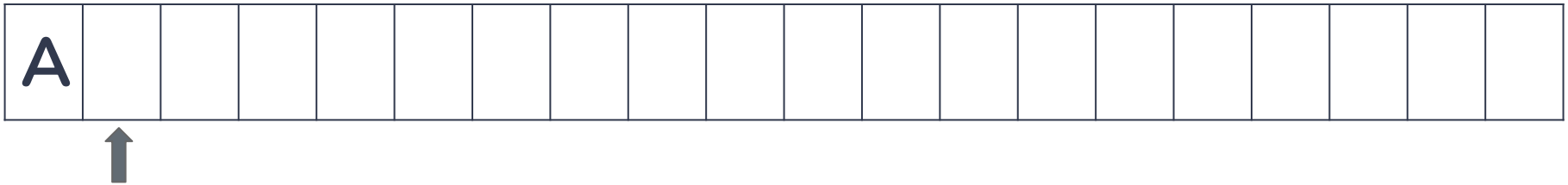
# Sorting Exercise 2

Merging sorted lists can be done as follows:

(1) Compare the first item in each list
(2) Add the “smaller” one to the combined list.
(3) Remove that item from its original list

C F H J M O Q S U Y

A D G I K N R T W Z



# Sorting Exercise 2

Merging sorted lists can be done as follows:

(1) Compare the first item in each list
(2) Add the “smaller” one to the combined list.
(3) Remove that item from its original list

C F H J M O Q S U Y

A D G I K N R T W Z

# Sorting Exercise 2

Merging sorted lists can be done as follows:

(1) Compare the first item in each list
(2) Add the “smaller” one to the combined list.
(3) Remove that item from its original list

C F H J M O Q S U Y

A D G I K N R T W Z

This takes n comparisons in total, one for each item on the list, so this is O(n) time!

| A | C | D | F | G | H | I | J | K | M | N | O | Q | R | S | T | U | W | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Sorting Exercise Takeaway

It’s easier to merge two sorted lists than it is to sort a whole list.

This implies the following approach for sorting:

(1) Divide the list into two halves.
(2) Sort each half.
(3) Merge the two sorted half-lists into a whole list.

# Sorting Exercise Takeaway

It’s easier to merge two sorted lists than it is to sort a whole list.

This implies the following approach for sorting:

(1) Divide the list into two halves.
(2) Sort each half.
(3) Merge the two sorted half-lists into a whole list.

**How should we sort each half?**

# Sorting Exercise Takeaway

We can sort them recursively! This implies the following pseudocode:

```
mergesort(A):
    left = mergesort(A[0:n/2])
    right = mergesort(A[n/2:n])
    L = merge(left, right)
    return L
```

**Q: What's wrong with this implementation?**

# Sorting Exercise Takeaway

We can sort them recursively! This implies the following pseudocode:

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

**It needed to be able to handle a base case!**

Mergesort was developed by John von Neumann in 1945.

# Recursive Algorithm Runtime Analysis

# But how long does this take?

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

O(1)

???

???

O(n)

# But how long does this take?

**Core Idea:** For recursive algorithms, we analyze runtime by writing a recurrence.

**Say mergesort takes T(n) time for an input of size n.**

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

O(1)

???

???

O(n)

# But how long does this take?

**Core Idea:** For recursive algorithms, we analyze runtime by writing a recurrence.

**Say mergesort takes T(n) time for an input of size n.**

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

O(1)

T(n/2)

T(n/2)

O(n)

# But how long does this take?

**Core Idea:** For recursive algorithms, we analyze runtime by writing a recurrence.

**Say mergesort takes T(n) time for an input of size n.**

Then mergesort follows the recurrence:

**T(n) = 2T(n/2) + O(n)**

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

O(1)

T(n/2)

T(n/2)

O(n)

# But how long does this take?

**Core Idea:** For recursive algorithms, we analyze runtime by writing a recurrence.

**Say mergesort takes T(n) time for an input of size n.**

Then mergesort follows the recurrence:

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

In this case, T(n) = O(n log n).

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

O(1)

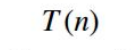
T(n/2)

T(n/2)

O(n)

# Proof 1: Recursion Tree

Claim: If T(n) = 2T(n/2) + O(n), then T(n) = n log n.

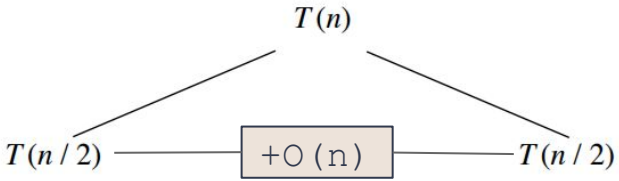



Disclaimer: We are assuming n is a power of 2.

# Proof 1: Recursion Tree

Claim: If T(n) = 2T(n/2) + O(n), then T(n) = n log n.

Merge Times

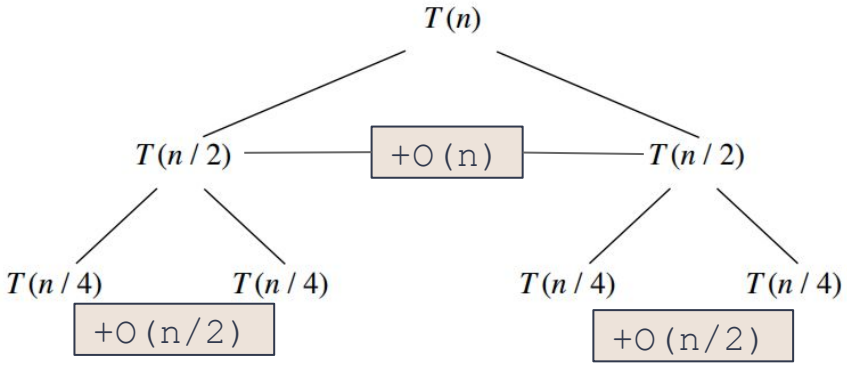



**TOTAL**

=O(n)

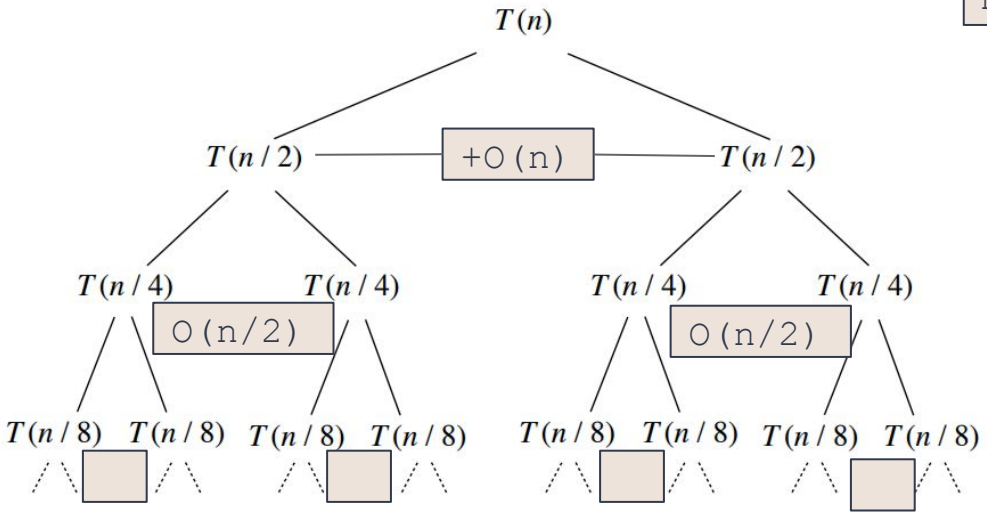
# Proof 1: Recursion Tree

Claim: If T(n) = 2T(n/2) + O(n), then T(n) = n log n.



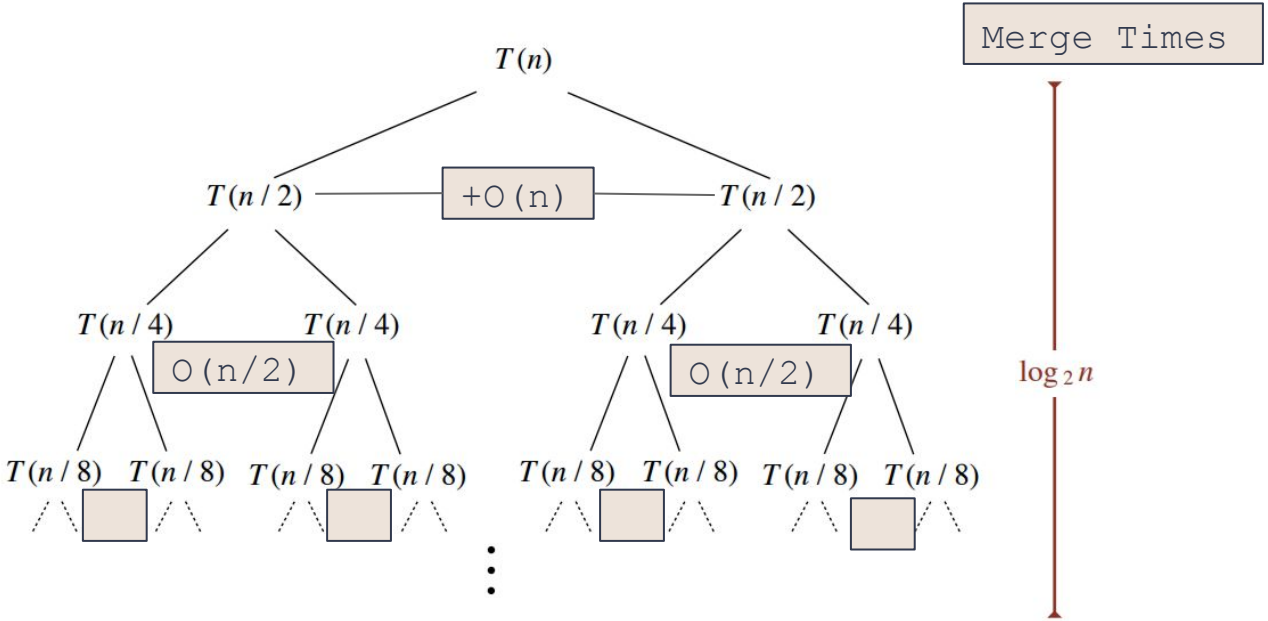
# Proof 1: Recursion Tree

Claim: If T(n) = 2T(n/2) + O(n), then T(n) = n log n.



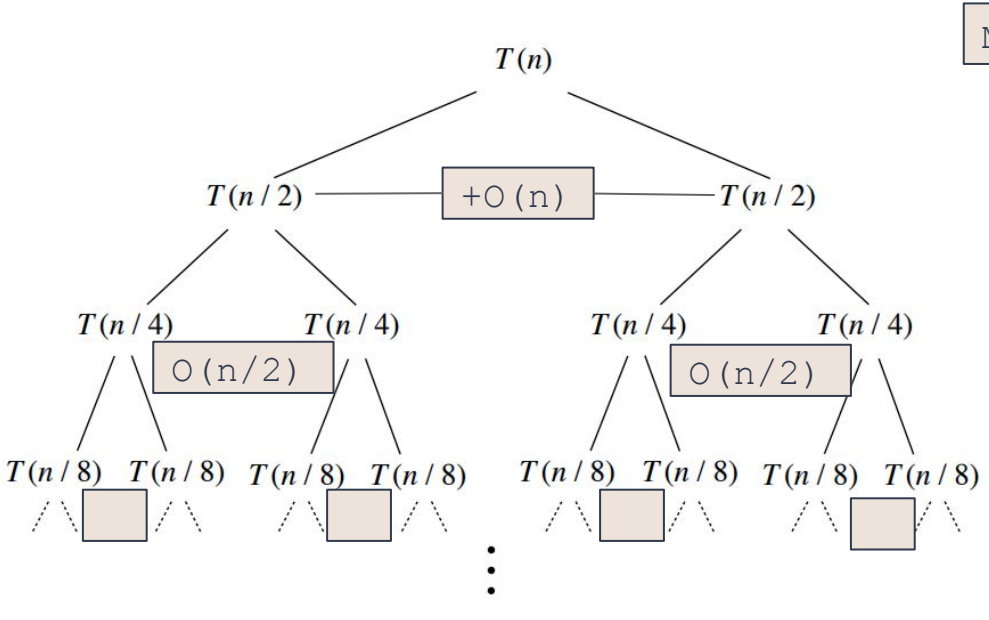
# Proof 1: Recursion Tree

Claim: If T(n) = 2T(n/2) + O(n), then T(n) = n log n.



# Proof 1: Recursion Tree

Claim: If T(n) = 2T(n/2) + O(n), then T(n) = n log n.




Merge Times

**TOTAL**

=O(n)

=O(n)

=O(n)

**= O(n log n)**

# Induction Recap

**Claim:** Some property holds for all non-negative integers.

**Proof by Induction:**
***Step 1: Base Case.*** Prove that the property holds for the base case(s).
***Step 2: Inductive Hypothesis.*** Assume that the statement is true for all integers less than or equal to some value n.
***Step 3: Inductive Step.*** Prove that if the inductive hypothesis is true then the statement must also be true for the next value of n.

We will see induction over and over (and over) in this course! Dig up your old notes if that'd be helpful, or come talk to me about it. You'll get some practice problems in the homework about it too.

# Proof 2: Induction

Claim: If $T(n) = 2T(n/2) + O(n)$, then $T(n) = O(n \log n)$.

Proof (by induction on d where $n=2^d$):

**Base Case:** d = 0. No algorithm is run here, so $T(1) = 1 \log 1 = 0$.
**Inductive Hypothesis:** Assume $T(n) = n \log n + O(n)$ for $n=2^d$.

**Inductive Step: Want to show $T(2n) = 2n \log(2n) + O(n)$.**

# Proof 2: Induction

Claim: If $T(n) = 2T(n/2) + O(n)$, then $T(n) = O(n \log n)$.

Proof (by induction on d where $n=2^d$):

**Base Case:** d = 0. No algorithm is run here, so $T(1) = 1 \log 1 = 0$.
**Inductive Hypothesis:** Assume $T(n) = n \log n + O(n)$ for $n=2^d$.

**Inductive Step: Want to show $T(2n) = 2n \log(2n) + O(n)$.**

$$T(2n) = 2T(n) + O(n).$$ By the recurrence.

$$2T(n) + O(n) = 2(n \log n) + O(n)$$ By the IH.

$$= 2n (\log 2n - \log 2) + O(n)$$ By log rules.

$$= 2n \log 2n - (2 \log 2) n + O(n)$$

$$= 2n \log(2n) + O(n)$$

So $T(n) = O(n \log n)$.

# What if n isn't a power of 2?

Say n is a power of 2, and (n/2)<m<n. You're running mergesort on a list of length m. Then:

(a) Then T(m) < O(n log n).
(b) Then T(m) > O(n log n).
(c) It depends.

# What if n isn't a power of 2?

Say n is a power of 2, and (n/2)<m<n. Then:

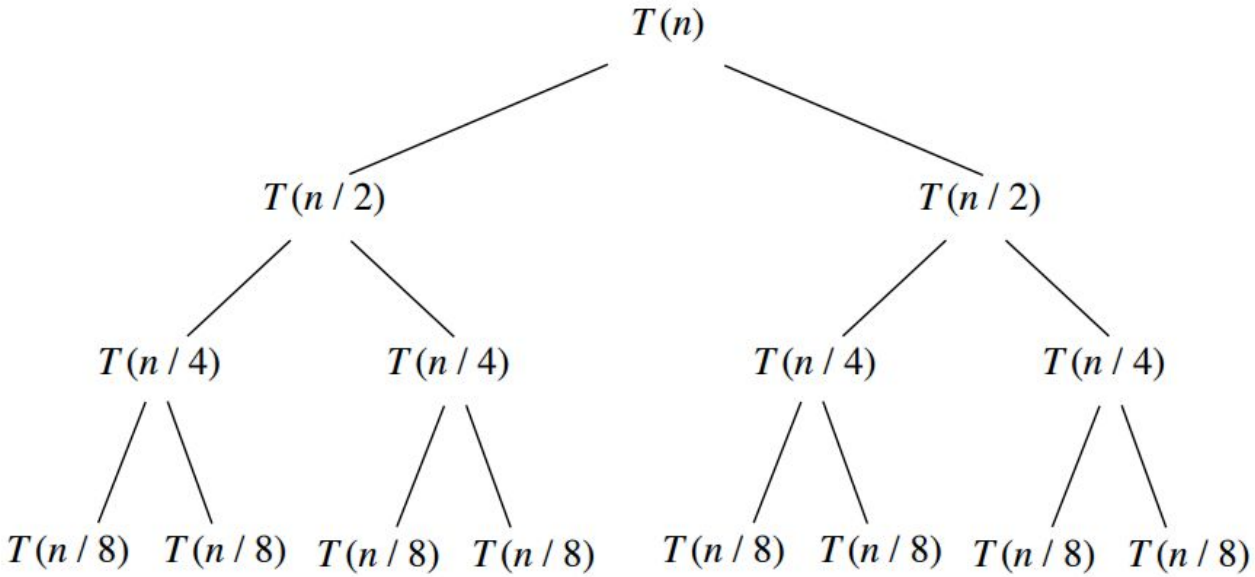
**(a) <u>Then T(m) < n log n.</u>**

(b) Then T(m) > n log n.

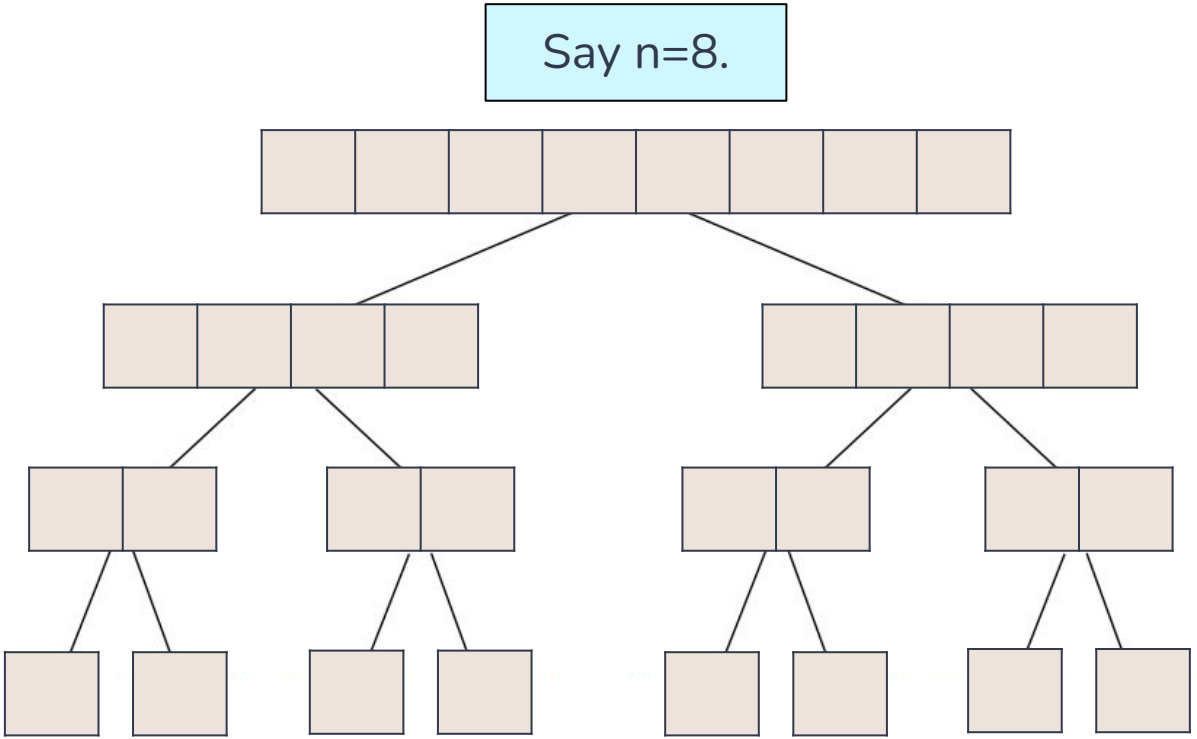
(c) It depends.

# What if n isn't a power of 2? (Proof by Picture)

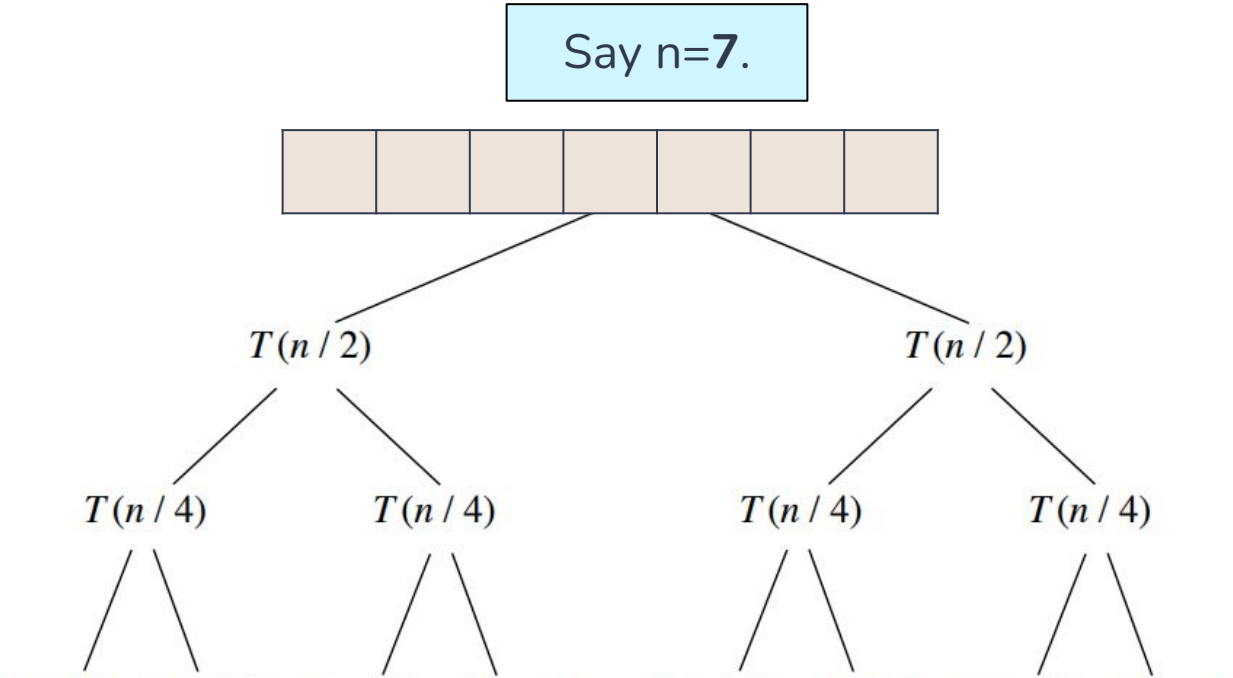
Say n=8.



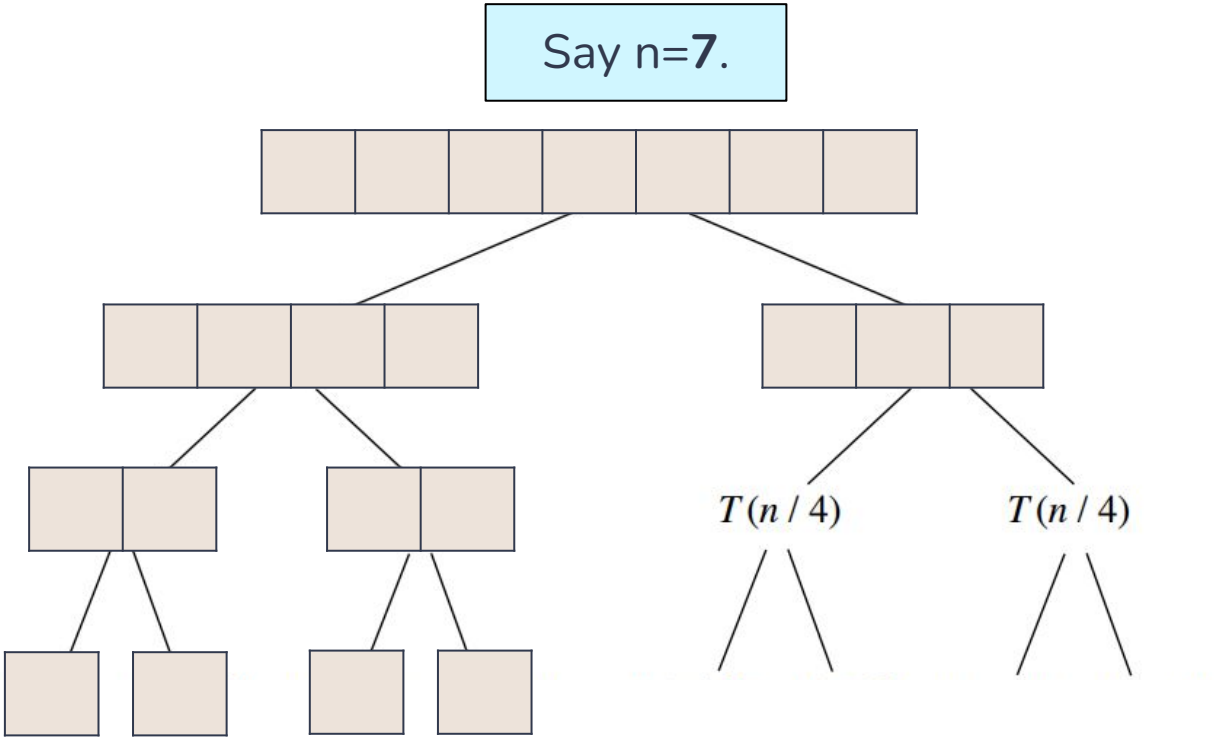
# What if n isn't a power of 2? (Proof by Picture)



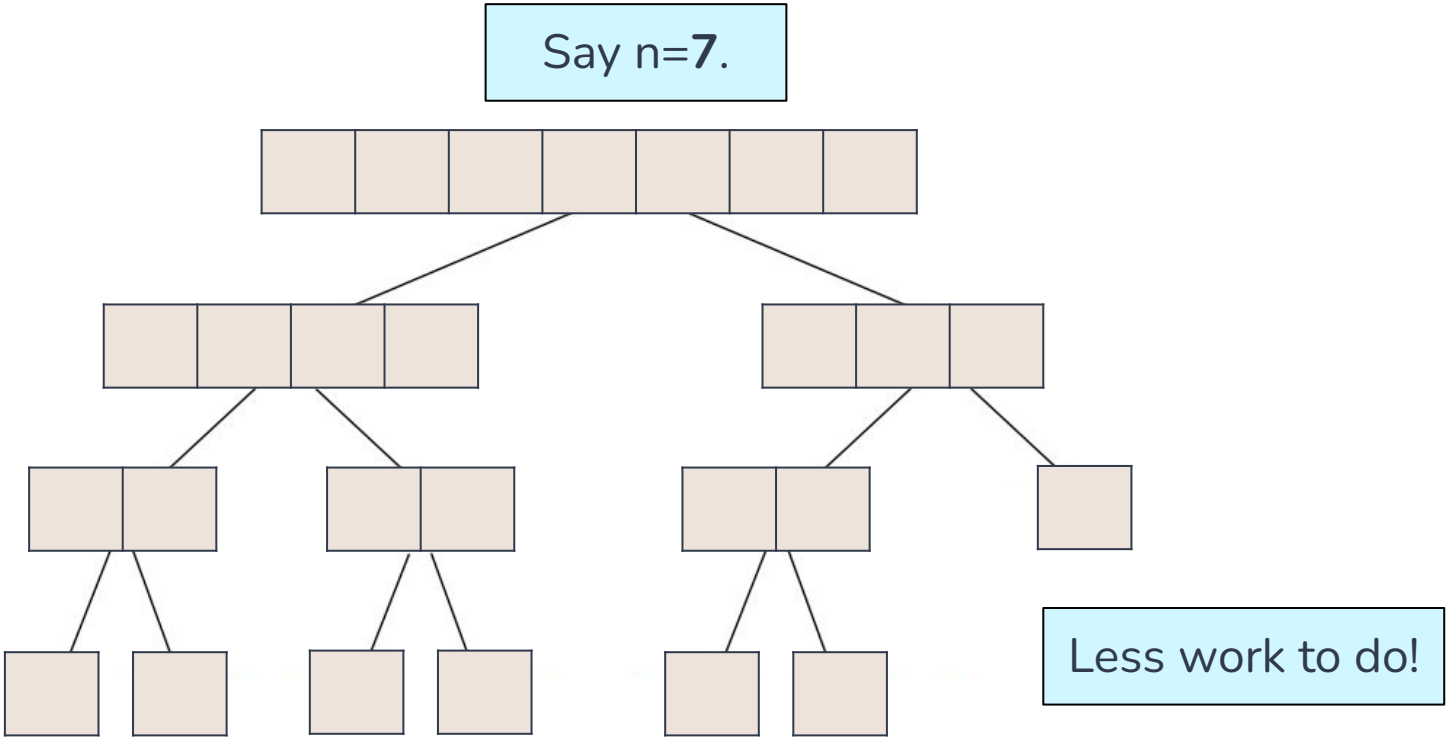
# What if n isn't a power of 2? (Proof by Picture)



# What if n isn't a power of 2? (Proof by Picture)



# What if n isn't a power of 2? (Proof by Picture)

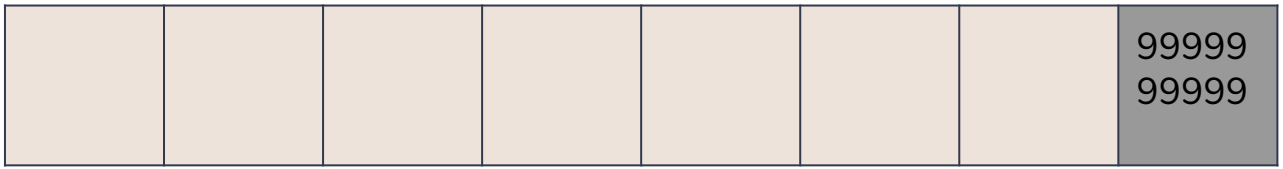


# What if n isn’t a power of 2? (Proof by Picture)

Say n=**7**.

# What if n isn't a power of 2? (Proof by Picture)

Say n=**7**.




If we add a junk number to the end, then it has become a list of length 8! So we can always make it run in exactly n log n time, so it'll never be worse than n log n.

# Divide and Conquer

# Divide and Conquer

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

# Divide and Conquer

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

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

# Divide and Conquer

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

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

# Divide and Conquer

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

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

# Divide and Conquer

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

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

# Summary

Today, we were introduced to the **Divide-and-Conquer** paradigm.

It follows a very simple idea: solving big problems is quite difficult, but solving small problems is easier.

These algorithms break the problem up into smaller pieces, recursively solve the smaller pieces, and find a efficient way to combine these solutions.