LEC 07

CSE 373

Recurrences II, Tree Method

BEFORE WE START

Review: Consider func (base case & non-recursive work omitted). How would you complete the recurrence?

```
int func (int n) {
  ...
  res += 2 * func(n / 4);
  res += 2 * func(n / 4);
  res += 2 * func(n / 4);
  return res;
}
```

$$T(n) = \begin{cases} 10 & \text{if } n = 0 \\ \boxed{??} + n & \text{otherwise} \end{cases}$$

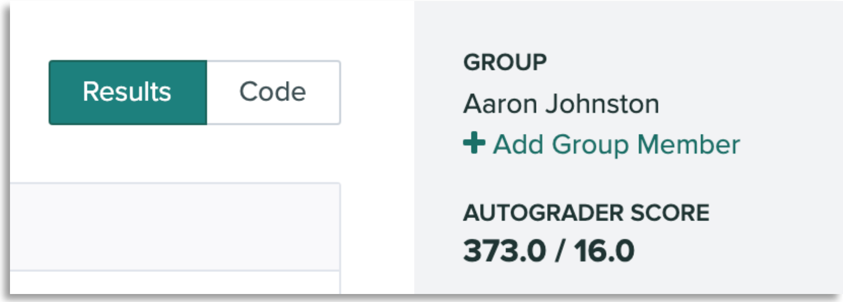
pollev.com/uwcse373

Instructor: Aaron Johnston

TAs: Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- P1 (Deques) due **TONIGHT 11:59pm PDT!**
  - Make sure to add your partner on your Gradescope submission!



  - Late Policy:
    - 7 penalty-free late days (24hr chunks) for the quarter
    - 5% deduction/day afterward
    - late assignment cutoff is 3 days later
  - Don't forget your writeup for the P1 experiments

- EX1 (Algo Analysis I) due Friday 7/10 11:59pm PDT
- P2 (Maps) and EX2 (Algo Analysis II) released Friday 7/10
- We'll see some summation identities in today's lecture
  - Summations Reference will be posted as a resource on the calendar

Announcements

- We're deeply disturbed by the recent US federal government announcement concerning F-1 visas in fall quarter requiring at least one in-person class
- UW and CSE are working on a response, hopefully updates soon
- If you need to focus on other things in light of this stressful news, please reach out – we can absolutely give accommodations
- Resources in the meantime:
  - Read Ana Mari Cauce's (UW President) statement:
    - https://www.washington.edu/president/2020/07/07/rule-ends-visa-waiver/
  - Allen School Advising & Diversity and Access Team
    - https://www.cs.washington.edu/academics/ugrad/advising
  - UW Counseling Center
    - https://www.washington.edu/counseling/
  - UW Samuel E. Kelly Ethnic Cultural Center
    - https://depts.washington.edu/ecc/
  - International Student Services
    - https://iss.washington.edu/travel-visas/coronavirus-information-for-f1-j1-students/
    - https://iss.washington.edu/

Learning Objectives

After this lecture, you should be able to...

1. ***Continued*** Describe the 3 most common recursive patterns and identify whether code belongs to one of them
2. Model a recurrence with the Tree Method and use it to characterize the recurrence with a bound
3. Use Summation Identities to find closed forms for summations (*Non-Objective:* come up with or explain Summation Identities)

Review Writing Recurrences

```
public int recurse(int n) {
  if (n < 3) {
    return 80;
  }

  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }

  int val1 = recurse(n / 3);
  int val2 = recurse(n / 3);
  int val3 = recurse(n / 3);

  return val1 + val2 + val3;
}
```

+2 Base Case

+n

+3

Recursive Case

Non-recursive Work: **+ n + 3**

Recursive Work: **+ 3*T(n/3)**

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 3T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

Review Why Include Non-Recursive Work?

```java
public int recurse(int n) {
  if (n < 3) {
    return 80;
  }
  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }
  int val1 = recurse(n / 3);
  int val2 = recurse(n / 3);
  int val3 = recurse(n / 3);

  return val1 + val2 + val3;
}
```

Base Case

Recursive Case

+n

+3

Think of it this way:

"work that happens if we enter base case"

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 3T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

"work that happens if we enter recursive case"

Non-recursive parts of recursive cases are sometimes where the bulk of the work takes place!

Review Master Theorem: Recurrence to Big-Θ

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

- It's still really hard to tell what the big-O is just by looking at it.
- But fancy mathematicians have a formula for us to use!

MASTER THEOREM

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

a=2 b=3 and c=1

$y = \log_b x$ *is equal to* $b^y = x$

$\log_3 2 = x$ $(3^x = 2 \Rightarrow x \cong 0.63)$

$\log_3 2 < 1$

We're in case 1

$T(n) \in \Theta(n)$

Lecture Outline

- Analyzing Recursive Code: Recursive Patterns

1	2	3
Halving the Input	**Constant size Input**	**Doubling the Input**
Binary Search Θ (log n)	Merge Sort	

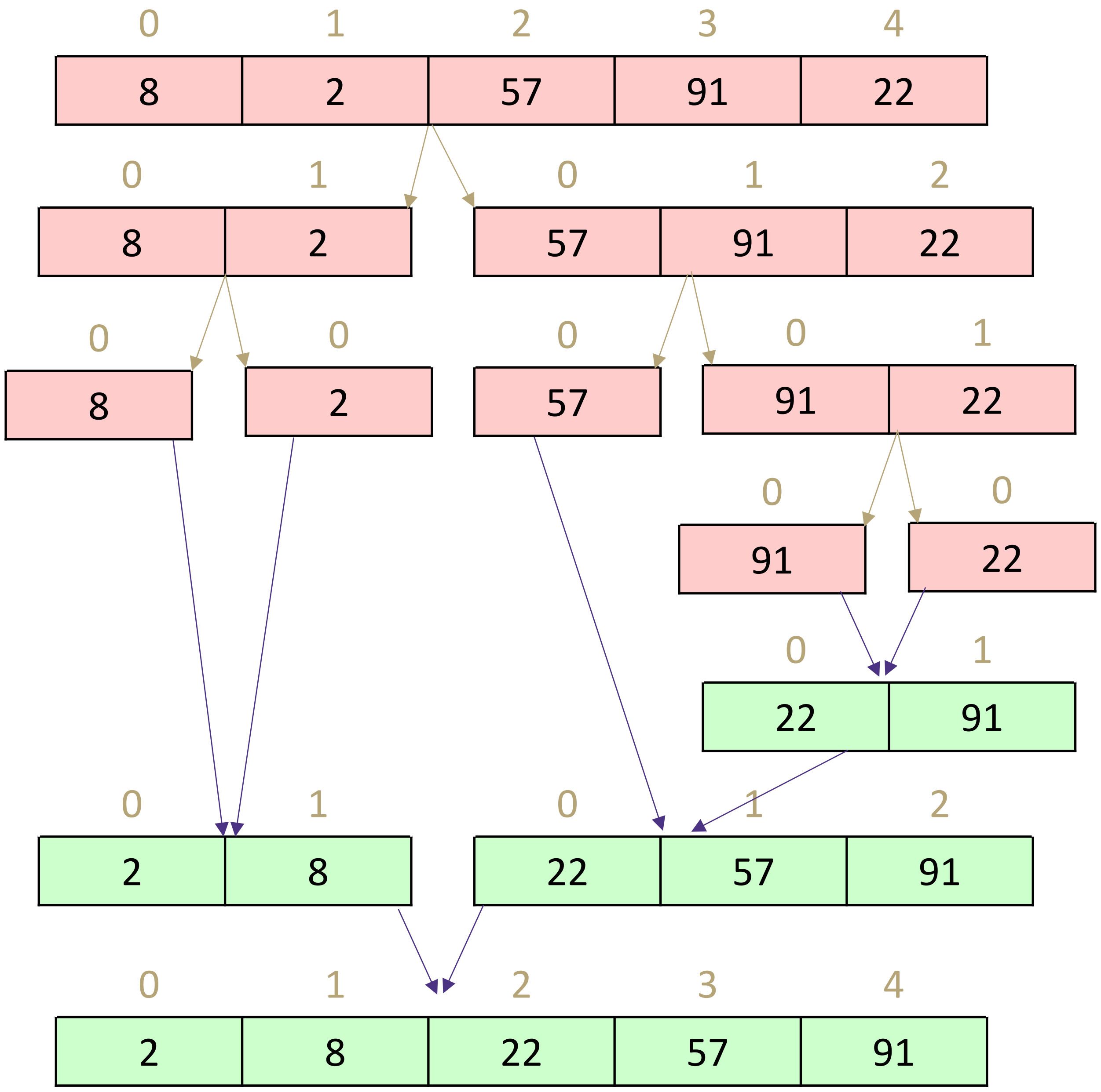
- Summations
- The Tree Method

Review Merge Sort

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

2 **Constant size Input**



Review Merge Sort Recurrence to Big-Θ

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

MASTER THEOREM

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta(n^{\log_b a})$

a=2 b=2 and c=1

$y = \log_b x$ *is equal to* $b^y = x$

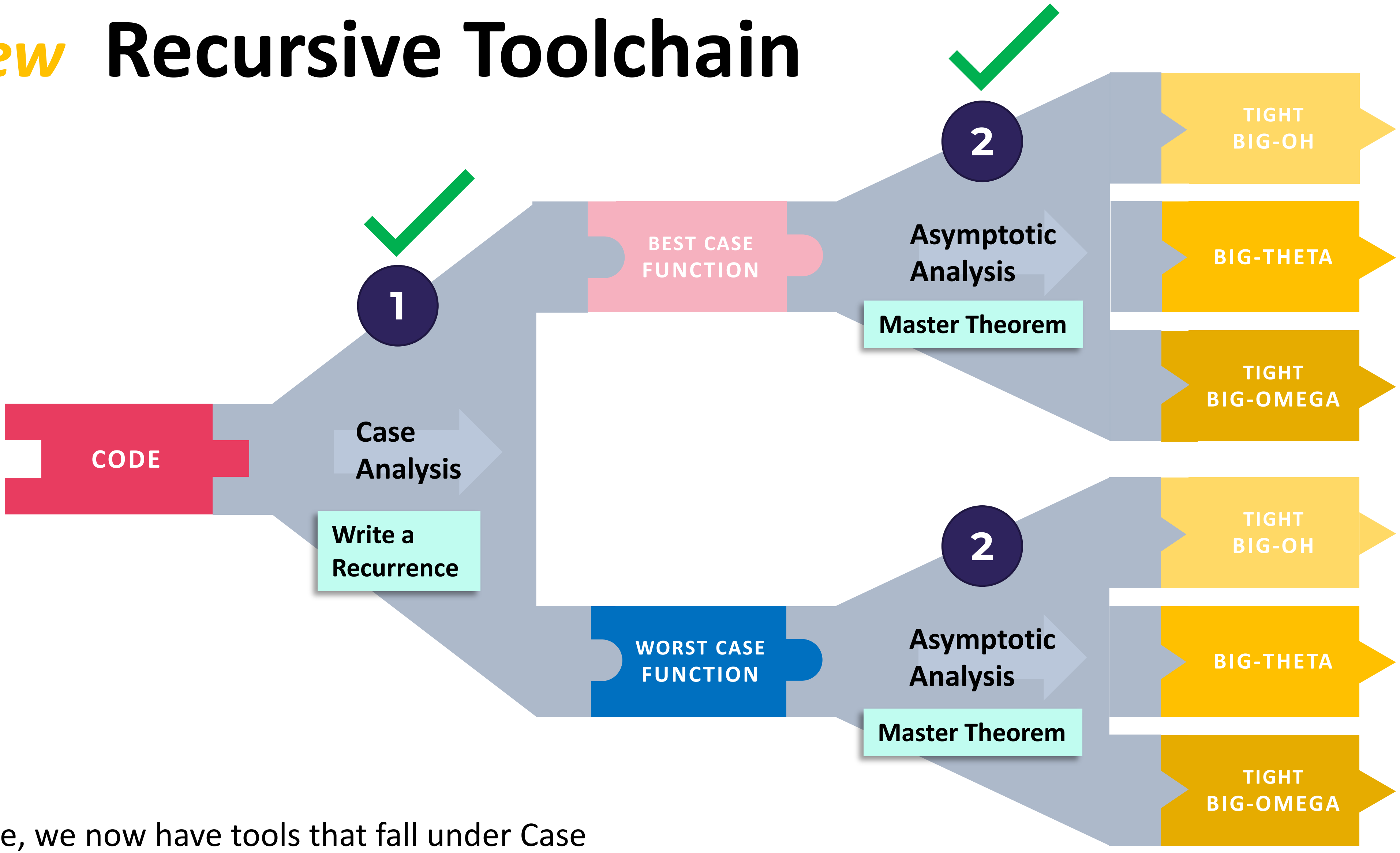
$\log_2 2 = x \Rightarrow 2^x = 2 \Rightarrow x = 1$

$\log_2 2 = 1$

We're in case 2

$T(n) \in \Theta(n \log n)$

Review Recursive Toolchain



For recursive code, we now have tools that fall under Case Analysis (Writing Recurrences) and Asymptotic Analysis (The Master Theorem).

Lecture Outline

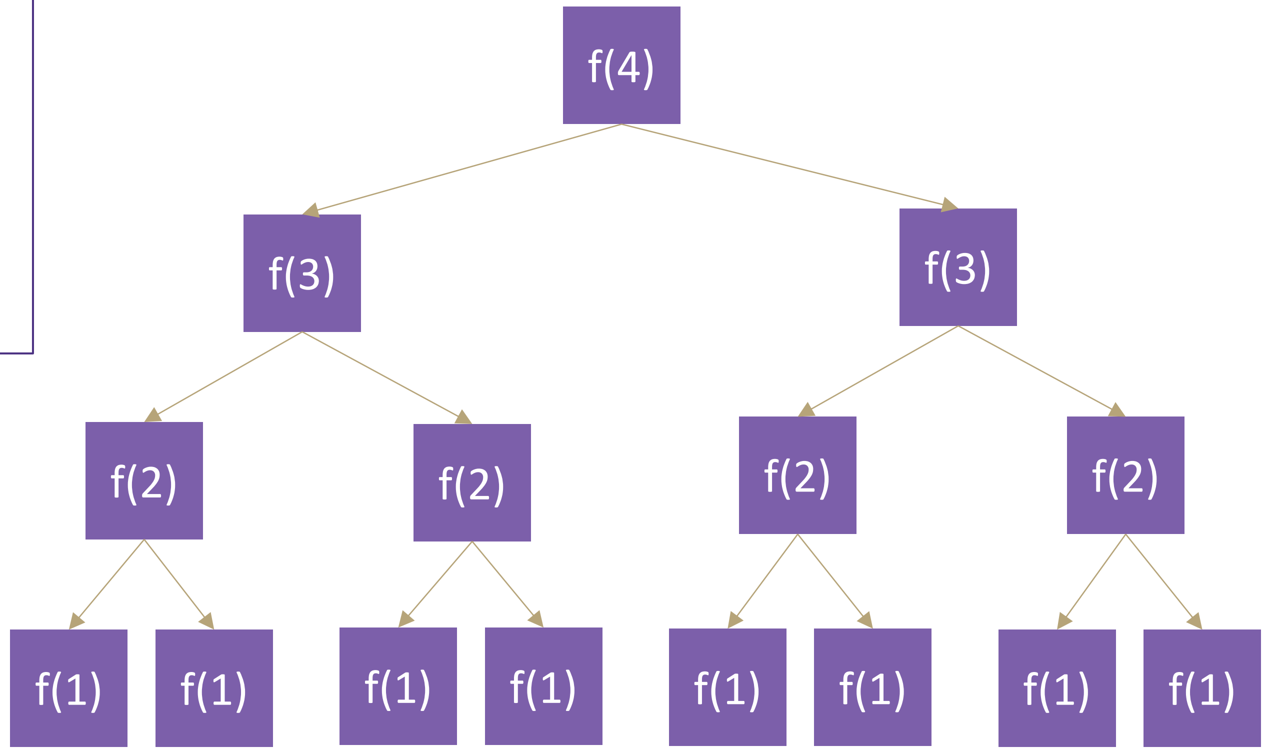
- Analyzing Recursive Code: Recursive Patterns

1	2	3
Halving the Input	**Constant size Input**	**Doubling the Input**
Binary Search Θ (log n)	Merge Sort Θ (n log n)	Fibonacci

- Summations
- The Tree Method

Calculating Fibonacci (ish)

```
public int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-1);
}
```



- Each call creates 2 more calls
- Each new call has a copy of the input, almost
- Almost doubling the input at each call

Almost

3 **Doubling the Input**

Fibonacci Recurrence to Big-Θ

```
public int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-1);
}
```

$\underbrace{\texttt{if (n <= 1) \{ return 1; \}}}_{d}$

$\underbrace{\texttt{return fib(n-1) + fib(n-1);}}_{2T(n-1) + c}$

$$T(n) = \begin{cases} d & \text{if } n \leq 1 \\ 2T(n-1) + c & \text{otherwise} \end{cases}$$

Can we use the Master Theorem?

MASTER THEOREM

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Uh oh, our model doesn't match that format…

Can we intuit a pattern?

T(1) = d

T(2) = 2T(2-1) + c = 2(d) + c

T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c

T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c

T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d +25c

Looks like something's happening, but it's hard to identify. Maybe geometry can help!

Fibonacci Recurrence to Big-Θ

CORRECTED AFTER LECTURE

$$T(n) = \begin{cases} d & \text{if } n \leq 1 \\ 2T(n-1) + c & \text{otherwise} \end{cases}$$

How many function calls per layer?

LAYER	FUNCTION CALLS
0	1 (= 2^0)
1	2 (= 2^1)
2	4 (= 2^2)
3	8 (= 2^3)

How many function calls on layer i?

2^i

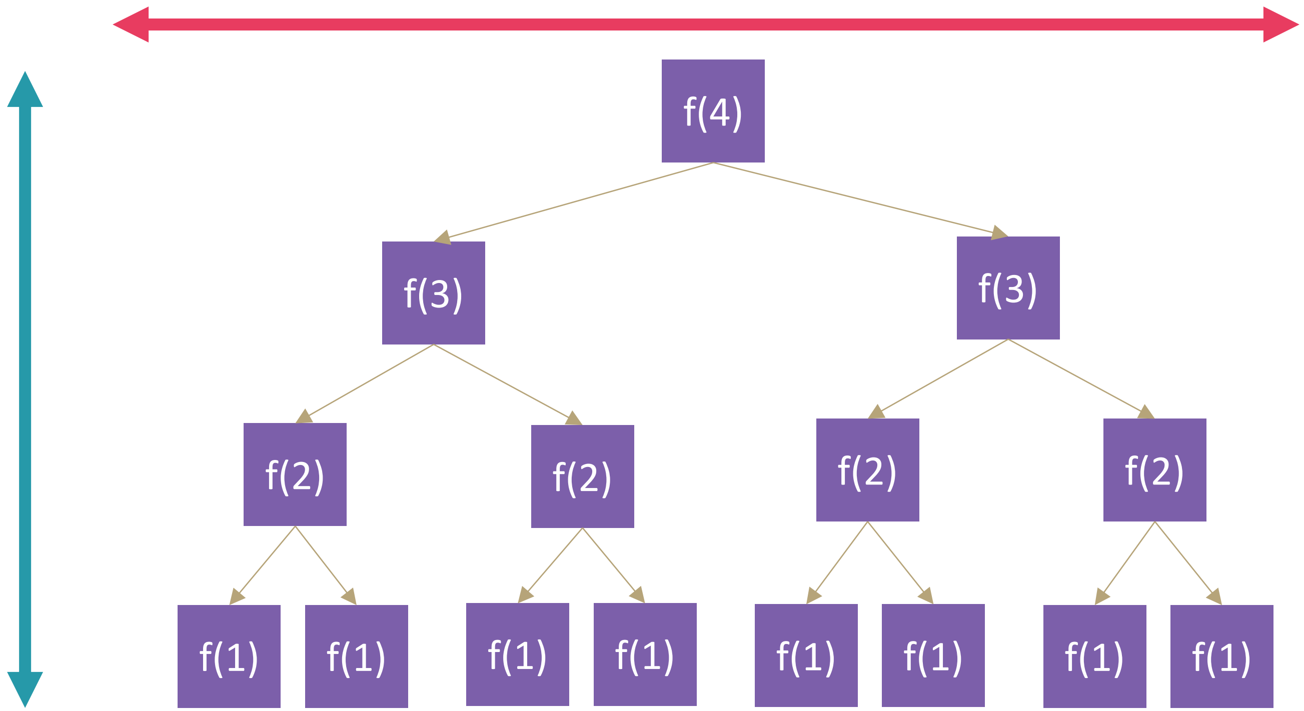
How many function calls TOTAL for a tree of k layers?

$1 + 2 + 4 + \ldots + 2^{k-1}$

How many layers in the function call tree?

How many steps to go from start value of n (4) to base case (1), subtracting 1 each time?

Height of function call tree: n



CORRECTED AFTER LECTURE

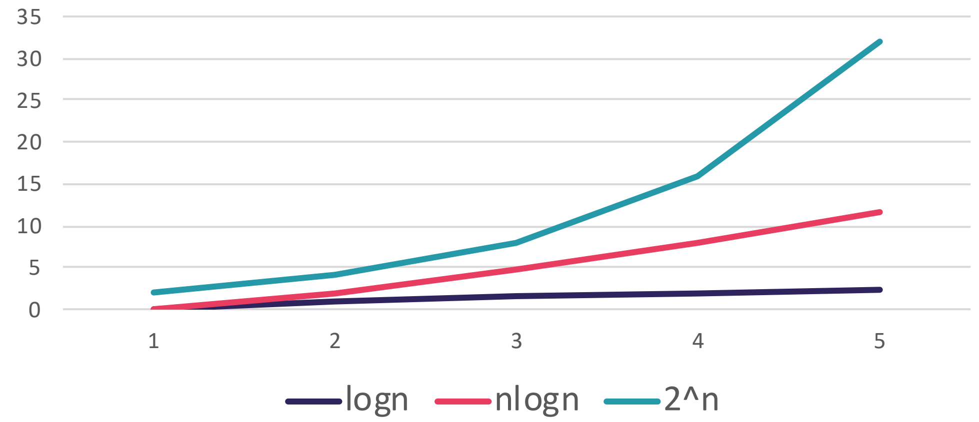
Fibonacci Recurrence to Big-Θ

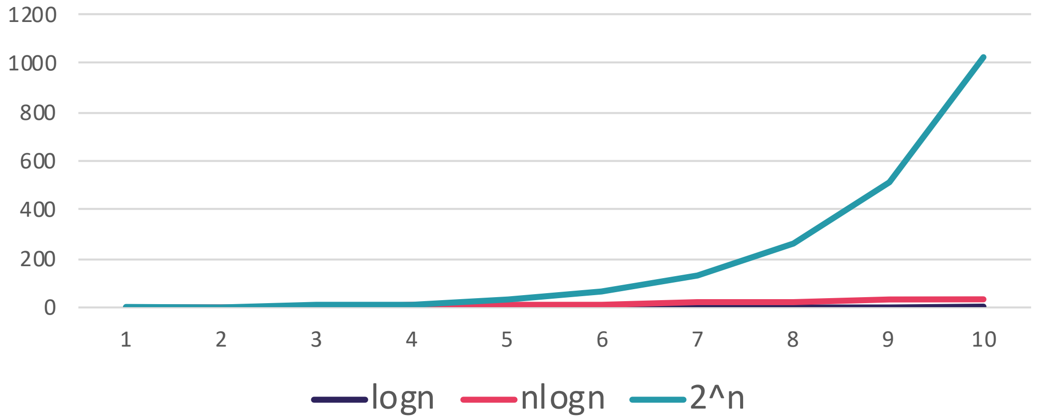
How many layers in the function call tree?	n
How many function calls on layer i?	2^i
How many function calls TOTAL for a tree of k layers?	$1 + 2 + 4 + 8 + \ldots + 2^{k-1}$
Total runtime = (total function calls) * (runtime of each function call)	$(1 + 2 + 4 + 8 + \ldots + 2^{k-1})$ x (constant work) $1 + 2 + 4 + 8 + \ldots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$ $\boldsymbol{T(n) = 2^n - 1 \in \Theta(2^n)}$

Summation Identity
Finite Geometric Series

$$\sum_{i=0}^{k-1} x^i = \frac{x^k - 1}{x - 1}$$

3 Patterns for Recursive Code





1

Halving the Input

Binary Search

Θ (log n)

2

Constant size Input

Merge Sort

Θ (n log n)

3

Doubling the Input

Fibonacci

Θ (2^n)

Lecture Outline

- Analyzing Recursive Code: Recursive Patterns

1	2	3
Halving the Input	**Constant size Input**	**Doubling the Input**
Binary Search	Merge Sort	Fibonacci
Θ (log n)	Θ (n log n)	Θ (2^n)

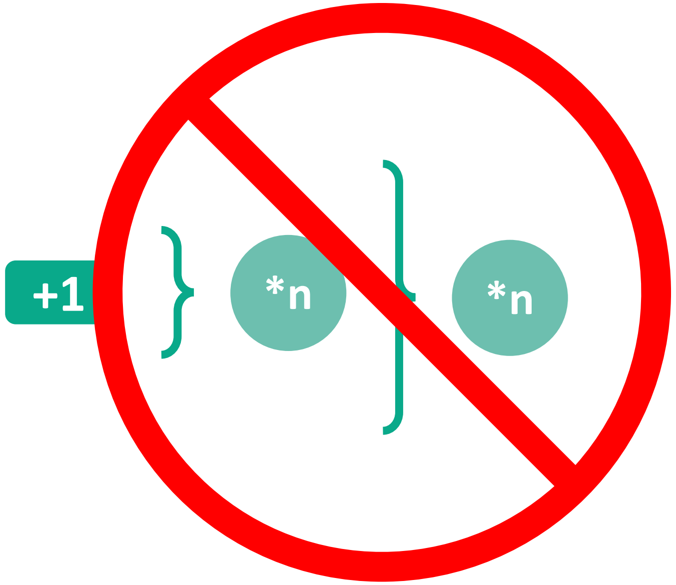
- **Summations**
- The Tree Method

Poll Everywhere

pollev.com/uwcse373

Which of these functions is a mathematical model for the runtime of this code?

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!");
    }
}
```



Keep an eye on the loop bounds!

a) $f(n) = 2n$

b) $f(n) = n + n$

c) $f(n) = n^2$

d) $f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1$

CORRECTED AFTER LECTURE

Modeling Complex Loops

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!");
    }
}
```

+1

*(0 + 1 + 2 + ... + i-1)

*n

Modeling the inner loop:

$$f(n) = \underbrace{(0 + 1 + 2 + \ldots + i-1)}_{\text{How do we model this part?}}$$

Summations!

$$1 + 2 + 3 + 4 + \ldots + n = \sum_{i=1}^{n} i$$

Definition: Summation

$$\sum_{i=a}^{b} f(i) = f(a) + f(a + 1) + f(a + 2) + \ldots + f(b-2) + f(b-1) + f(b)$$

Modeling the entire code snippet:

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1$$

What is the Big-Theta?

Simplifying Summations

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < i; j++) {
      System.out.println("Hello!");
   }
}
```

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1$$

closed form

simplified big-Theta

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \quad = \sum_{i=0}^{n-1} 1 \cdot i \quad = 1 \sum_{i=0}^{n-1} i \quad = \frac{n(n-1)}{2} \quad = \frac{1}{2}n^2 - \frac{1}{2}n \quad = \boldsymbol{\theta(n^2)}$$

Summation of a constant

$$\sum_{i=0}^{k-1} c = ck$$

Factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c \sum_{i=a}^{b} f(i)$$

Gauss's Identity

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

You don't have to come up with these or explain why! We'll publish a list of identities.

The code ***is*** $\Theta(n^2)$, but it ***is not*** correct to say $f(n) = n^2$ models its runtime!

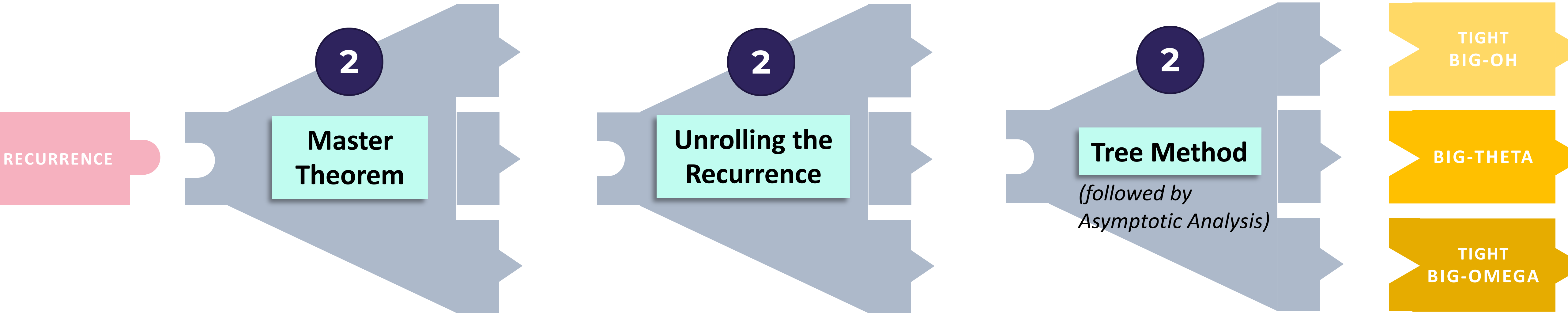
Lecture Outline

- Analyzing Recursive Code: Recursive Patterns

1	2	3
Halving the Input	**Constant size Input**	**Doubling the Input**
Binary Search Θ (log n)	Merge Sort Θ (n log n)	Fibonacci Θ (2^n)

- Summations
- **The Tree Method**

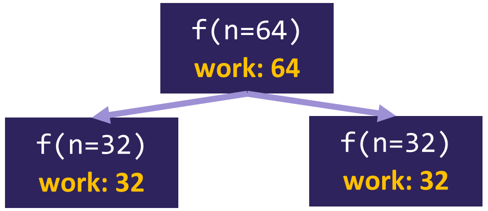
Recurrence to Big-Theta: Our Toolbox



MASTER THEOREM

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

T(1) = d
T(2) = 2T(2-1) + c = 2(d) + c
T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c



PROS: Convenient to plug ‘n’ chug
CONS: Only works for certain format of recurrences

PROS: Least complicated setup
CONS: Requires intuitive pattern matching, no formal technique

PROS: Convenient to plug ‘n’ chug
CONS: Complicated to set up for a given recurrence

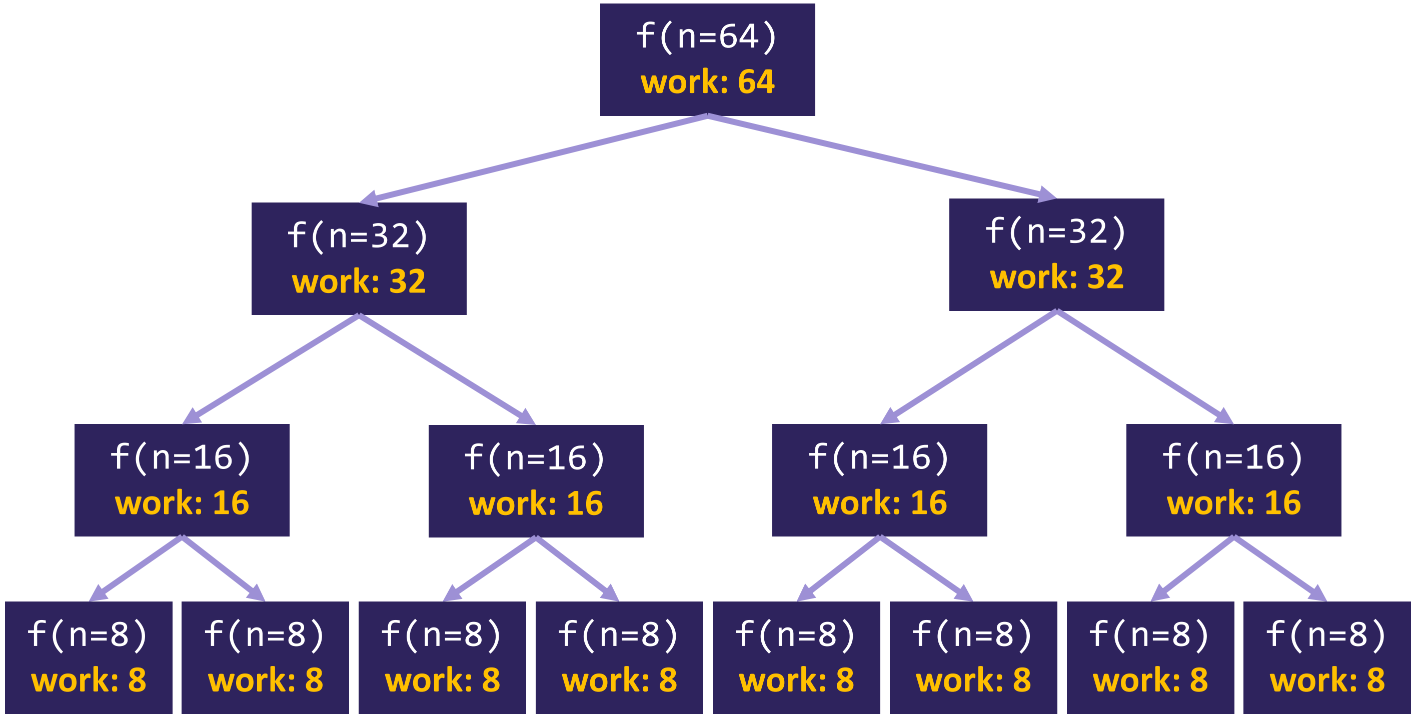
Tree Method (Generalizing from Fibonacci Example)

Draw out the function call tree. What's the input to each call? How much work is done in each call?

e.g. **Merge Sort:**

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```



...

Where's that work coming from?

A Θ(n) operation inside of Merge Sort that processes the entire input!

Tree Method

e.g. **Merge Sort:**

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

How many layers in the function call tree?

How many steps to go from start value of n to base case (1), dividing by 2 each time?

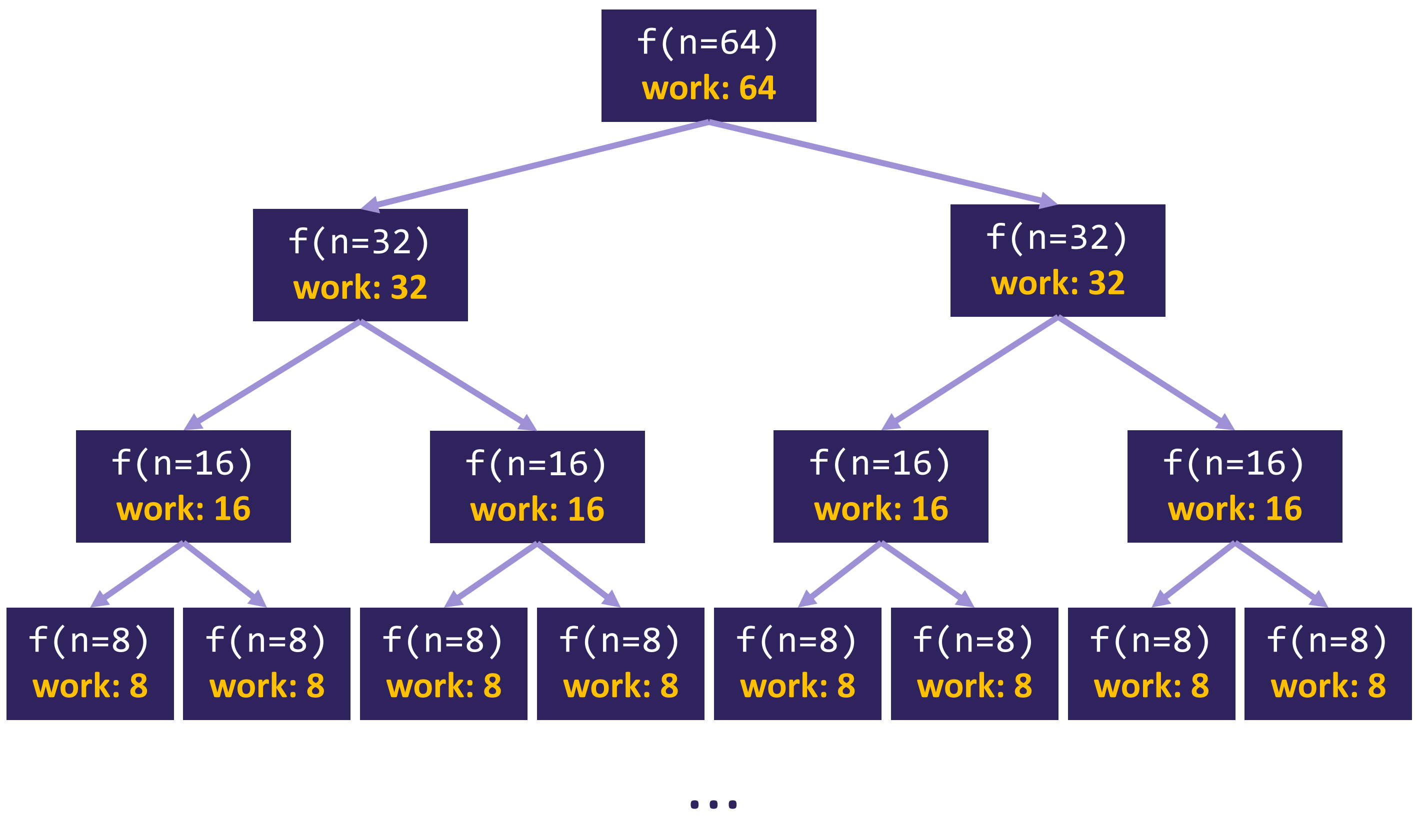
Think binary search – it takes $\log_2 n$ "halvings" to take n down to 1

Height of function call tree: $\log_2 n$

How much work done per layer?

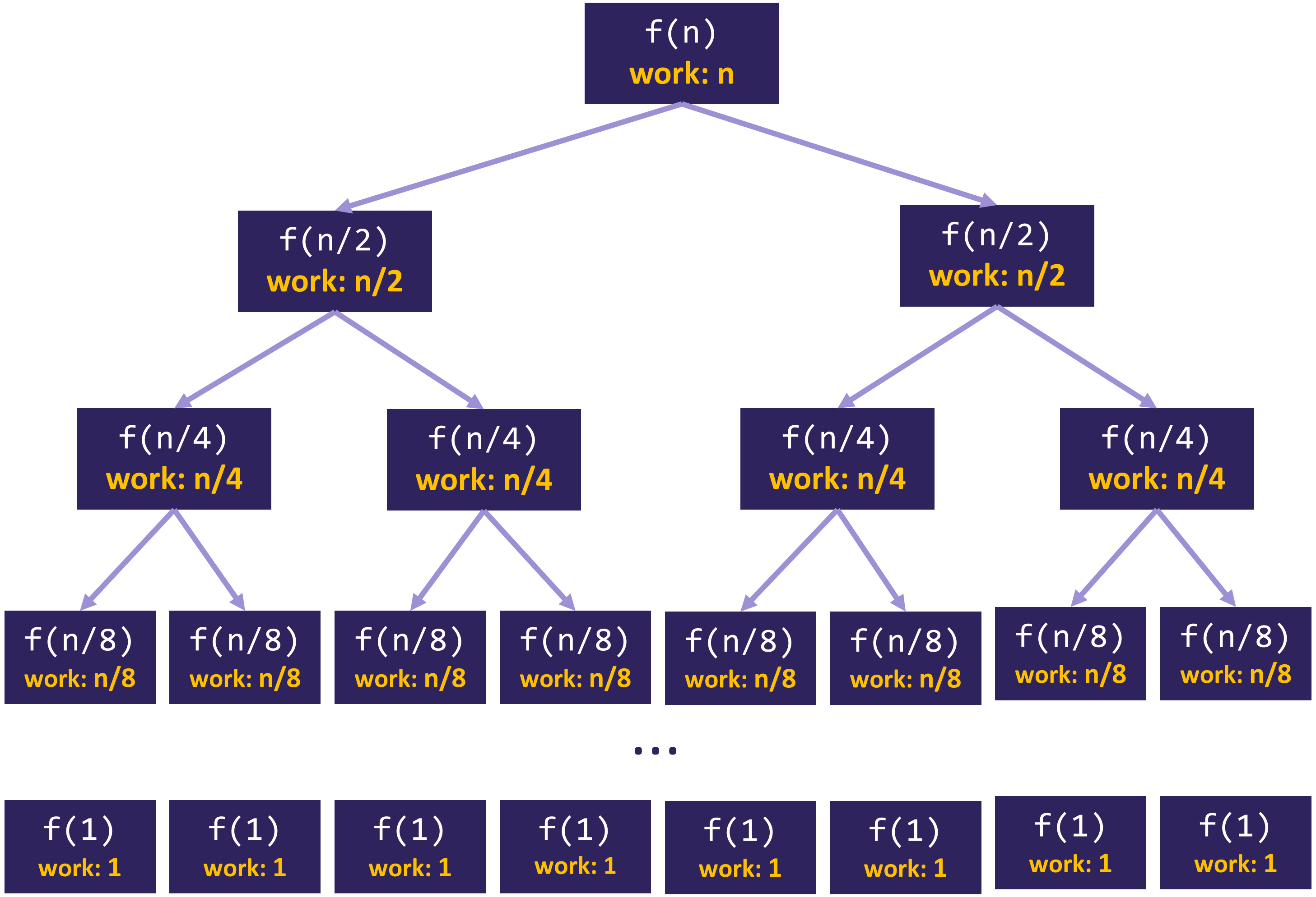
Amount of work varies by function call, but remains constant across entire layer

n work at each layer



Tree Method

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$



Recursive level	How many nodes at each level?	How much work done by each node?	How much work across each level?
0	1	n	n
1	2	$\frac{n}{2}$	n
2	4	$\frac{n}{4}$	n
3	8	$\frac{n}{8}$	n
logn	n	1	n

Tree Method Checklist

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

1	What's the size of the input per call on level i?	$\frac{n}{2^i}$
2	How much work done by each node on level i (recursive case)?	$(\frac{n}{2^i})$ 1
3	How many nodes at level i?	2^i
4	What's the total work done on level i (recursive case)?	$\overset{3}{numNodes} * \overset{2}{workPerNode}$ $= 2^i\left(\frac{n}{2^i}\right) = n$
5	On what value of i does the last level occur (base case)?	$\frac{n}{2^i} = 1$ $(n = 2^i \Rightarrow i = \log_2 n)$
6	How much work done by each node on last level (base case)?	1
7	What's the total work on the last level (base case)?	5 3 6 $numNodes * workPerNode$ $= 2^{\log_2 n}(1) = n$

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	n	n
1	2	$\frac{n}{2}$	n
2	4	$\frac{n}{4}$	n
3	8	$\frac{n}{8}$	n
$\log_2 n$	n	1	n

Tree Method Checklist

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

1	What's the size of the input per call on level i?	$\frac{n}{2^i}$
2	How much work done by each node on level i (recursive case)?	$(\frac{n}{2^i})$ 1
3	How many nodes at level i?	2^i
4	What's the total work done on level i (recursive case)?	3 2 $numNodes * workPerNode$ $= 2^i\left(\frac{n}{2^i}\right) = n$
5	On what value of i does the last level occur (base case)?	$\frac{n}{2^i} = 1$ $(n = 2^i \Rightarrow i = \log_2 n)$
6	How much work done by each node on last level (base case)?	1
7	What's the total work on the last level (base case)?	5 3 6 $numNodes * workPerNode$ $= 2^{log_2 n}(1) = n$

Putting it Together:

5 7

$$T(n) = \sum_{i=0}^{\log_2 n - 1} n + n$$

4

$$= n\ log_2 n + n$$

$$= \Theta(\text{n log n})$$

Summation of a Constant

$$\sum_{i=0}^{k-1} c = ck$$

Power of a Log

$$x^{\log_b y} = y^{\log_b x}$$

Next Stop: The Data Structures Part™

- We're now armed with a toolbox stuffed full of analysis tools
  - It's time to apply this theory to more practical topics!

- On Friday, we'll take our first deep dive using those tools on a data structure: Hash Maps!