Lecture 7: Modeling Complex Code

CSE 373: Data Structures and Algorithms

Warm Up

Please fill out the Poll at- pollev.com/21sp373

Which of the following statements are true?

Select all options that apply.

- A Big-Theta bound will exist for every function.
- One possible Best Case for adding to **`ArrayDeque`** is when it is empty.
- We only use Big-Omega for Worst Case analysis
- If a function is $O(n^2)$ it can't also be $\Omega(n^2)$.

All false!

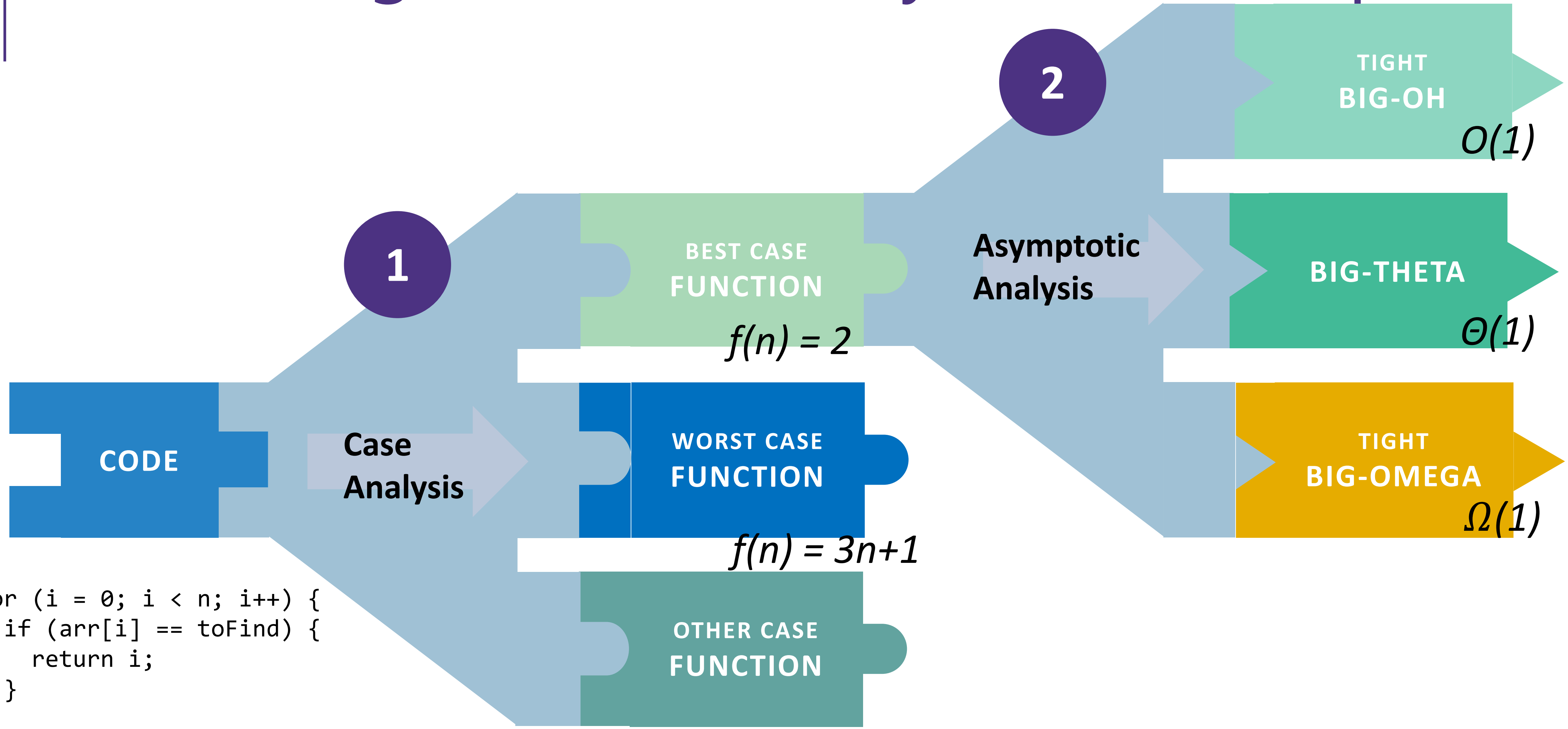
Announcements

Exercise 1 – Algorithm Analysis – Due Friday April 16th

Project 1 – Deques – Due Wednesday April 14th

Questions

Review Algorithmic Analysis Roadmap




```
for (i = 0; i < n; i++) {
  if (arr[i] == toFind) {
    return i;
  }
}
return -1;
```

Review Oh, and Omega, and Theta, oh my

Big-Oh is an **upper bound**

- My code takes at most this long to run

Big-Omega is a **lower bound**

- My code takes at least this long to run

Big Theta is **"equal to"**

- My code takes "exactly"* this long to run
- *Except for constant factors and lower order terms
- Only exists when Big-Oh == Big-Omega!

Big-Oh

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

Big-Theta

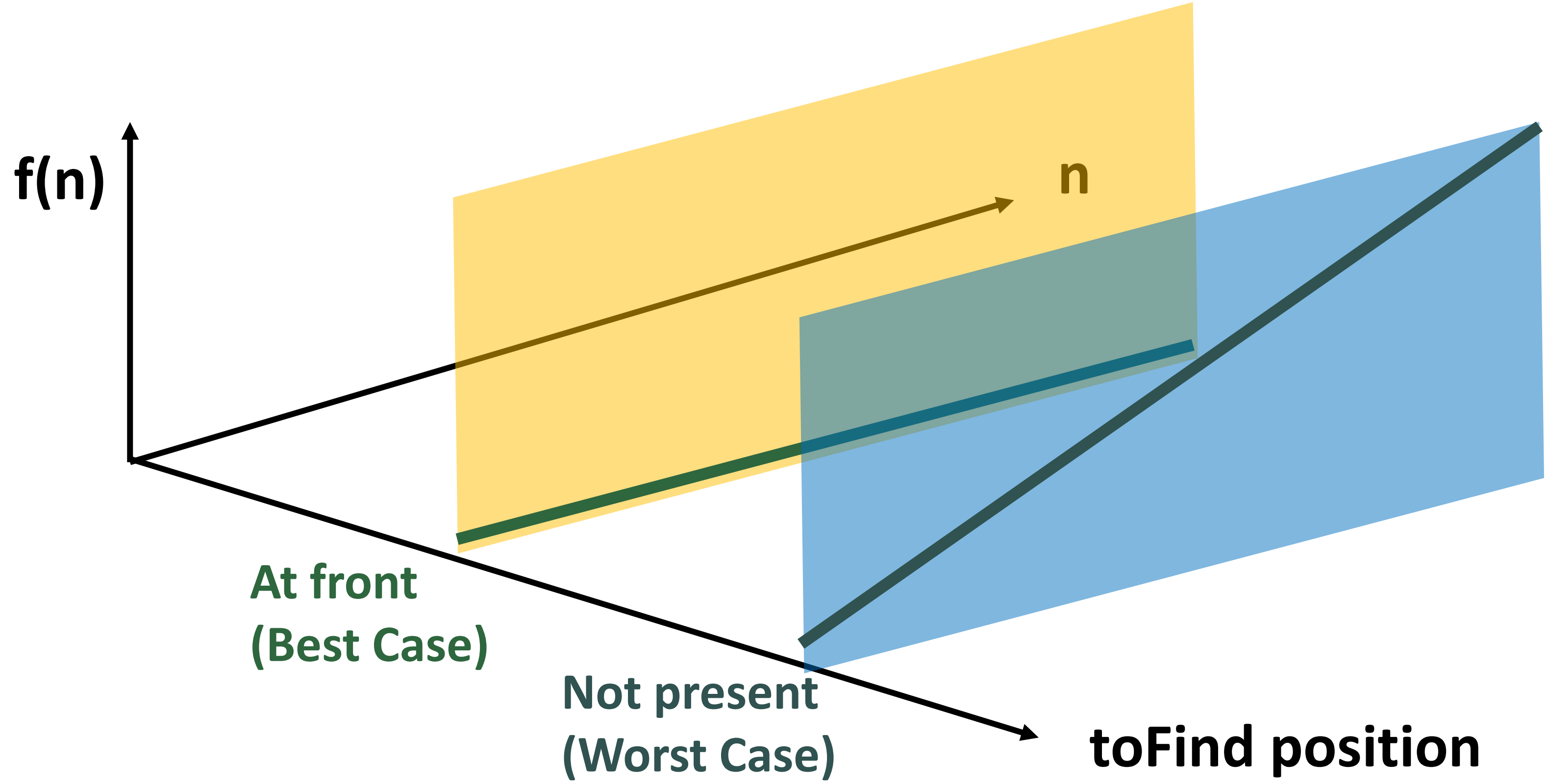
$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
(in other words: there exist positive constants $c1, c2, n_0$ such that for all $n \geq n_0$)

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Review When to do Case Analysis?

Imagine a 3-dimensional plot

- Which case we're considering is one dimension
- Choosing a case lets us take a "slice" of the other dimensions: n and f(n)
- We do asymptotic analysis on each slice in step 2



Modeling Recursive Code

Recursive Patterns

Modeling and analyzing recursive code is all about finding patterns in how the input changes between calls and how much work is done within each call

Let's explore some of the more common recursive patterns

Pattern #1: Halving the Input

Pattern #2: Constant size input and doing work

Pattern #3: Doubling the Input

Binary Search

```
public int binarySearch(int[] arr, int toFind, int lo, int hi) {
    if( hi < lo ) {
        return -1;
    } if(hi == lo) {
        if(arr[hi] == toFind) {
            return hi;
        }
        return -1;
    }
    int mid = (lo+hi) / 2;
    if(arr[mid] == toFind) {
        return mid;
    } else if(arr[mid] < toFind) {
        return binarySearch(arr, toFind, mid+1, hi);
    } else {
        return binarySearch(arr, toFind, lo, mid-1);
    }
}
```

Binary Search Runtime

binary search: Locates a target value in a *sorted* array or list by successively eliminating half of the array from consideration.

- Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	**42**	50	56	68	85	92	103

min mid max

How many elements will be examined?

- What is the best case?
  element found at index 8, 1 item examined, O(1)
- What is the worst case?
  element not found, ½ elements examined, then ½ of that...

Pattern #1 – Halving the input

Take 1 min to respond to activity

www.pollev.conm/cse373activity
Take a guess! What is the tight Big-O of worst case binary search?

Binary search runtime

For an array of size N, it eliminates ½ until 1 element remains.

N, N/2, N/4, N/8, ..., 4, 2, 1

- How many divisions does it take?

Think of it from the other direction:

- How many times do I have to multiply by 2 to reach N?

  1, 2, 4, 8, ..., N/4, N/2, N
- Call this number of multiplications "x".

$2^x = N$

$x = \log_2 N$

Binary search is in the **logarithmic** complexity class.

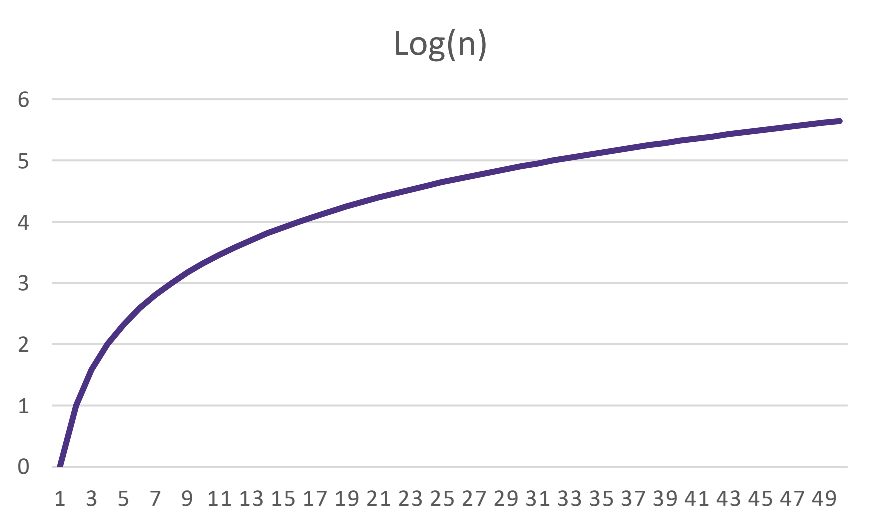
Logarithm – inverse of exponentials

$y = \log_b x \text{ is equal to } b^y = x$

Examples:

$2^2 = 4 \Rightarrow 2 = \log_2 4$

$3^2 = 9 \Rightarrow 2 = \log_3 9$

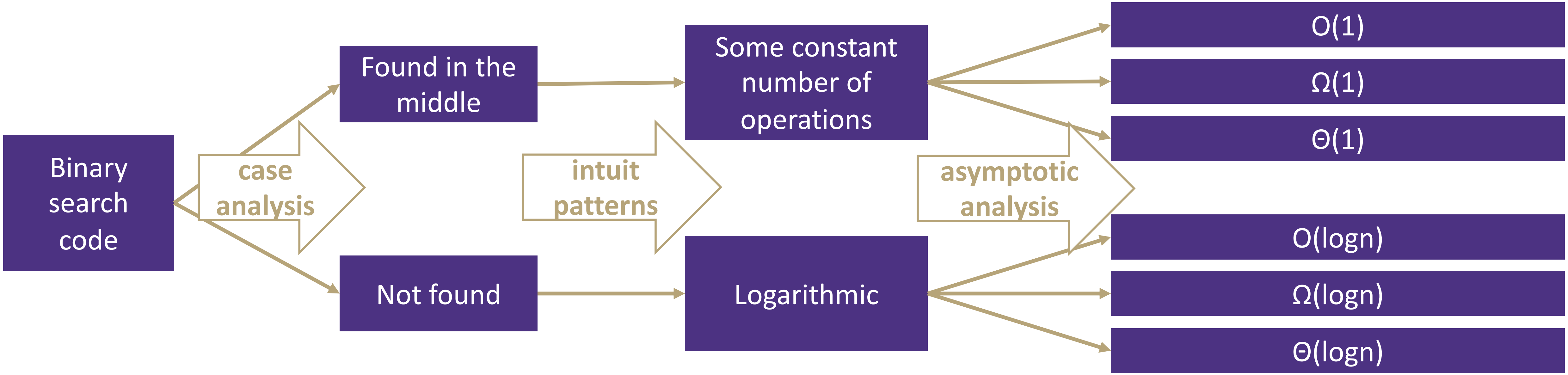


Moving Forward

While this analysis is correct it relied on our ability to think through the pattern intuitively

This works for binary search, but most recursive code is too complex to rely on our intuition.

We need more powerful tools to form a proper code model.



Model

Let's start by just getting a model. Let $F(n)$ be our model for the worst-case running time of binary search.

```
public int binarySearch(int[] arr, int toFind, int lo, int hi) {
    if( hi < lo ) {
        return -1;
    } if(hi == lo) {
        if(arr[hi] == toFind) {
            return hi;
        }
        return -1;
    }
    int mid = (lo+hi) / 2;
    if(arr[mid] == toFind) {
        return mid;
    } else if(arr[mid] < toFind) {
        return binarySearch(arr, toFind, mid+1, hi);
    } else {
        return binarySearch(arr, toFind, lo, mid-1);
    }
}
```

2

4

6

2 + ??

How do you model recursive calls?

With a recursive math function!

Meet the Recurrence

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s)

It's a lot like recursive code:

- At least one base case and at least one recursive case
- Each case should include the values for n to which it corresponds
- The recursive case should reduce the input size in a way that eventually triggers the base case
- The cases of your recurrence usually correspond exactly to the cases of the code

$$T(n) = \begin{cases} 5 & \text{if } n < 3 \\ 2T\left(\frac{n}{2}\right) + 10 & \text{otherwise} \end{cases}$$

Write a Recurrence

```
public int recursiveFunction(int n){
    if(n < 3) {
        return 3;
    }
    for(int int i=0; i < n; i++) { +n
        System.out.println(i);
    }
    int val1 = recursiveFunction(n/3);
    int val2 = recursvieFunction(n/3);
    return val1 * val2;
}
```

Base Case **2**

Recursive Case
Non-recursive work **n+2**
Recursive work **2*T(n/3)**

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

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* $b^y = x$

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

$\log_3 2 < 1$

We're in case 1

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

Understanding Master Theorem

<u>Master Theorem</u>

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

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 measures how many recursive calls are triggered by each method instance
- B measures the rate of change for input
- C measures the dominating term of the non recursive work within the recursive method
- D measures the work done in the base case

The $\log_b a < c$ case
- Recursive case does a lot of non recursive work in comparison to how quickly it divides the input size
- Most work happens in beginning of call stack
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case
- Recursive case evenly splits work between non recursive work and passing along inputs to subsequent recursive calls
- Work is distributed across call stack

The $\log_b a > c$ case
- Recursive case breaks inputs apart quickly and doesn't do much non recursive work
- Most work happens near bottom of call stack

Questions

Recursive Patterns

Pattern #1: Halving the Input

Binary Search $\Theta(\log n)$

Pattern #2: Constant size input and doing work

Merge Sort

Pattern #3: Doubling the Input

Merge Sort

Divide

0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	10	6	7	4

Conquer

0
8

0
8

Combine

0	1	2	3	4
2	8	22	57	91

5	6	7	8	9
1	4	6	7	10

0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

Merge Sort

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

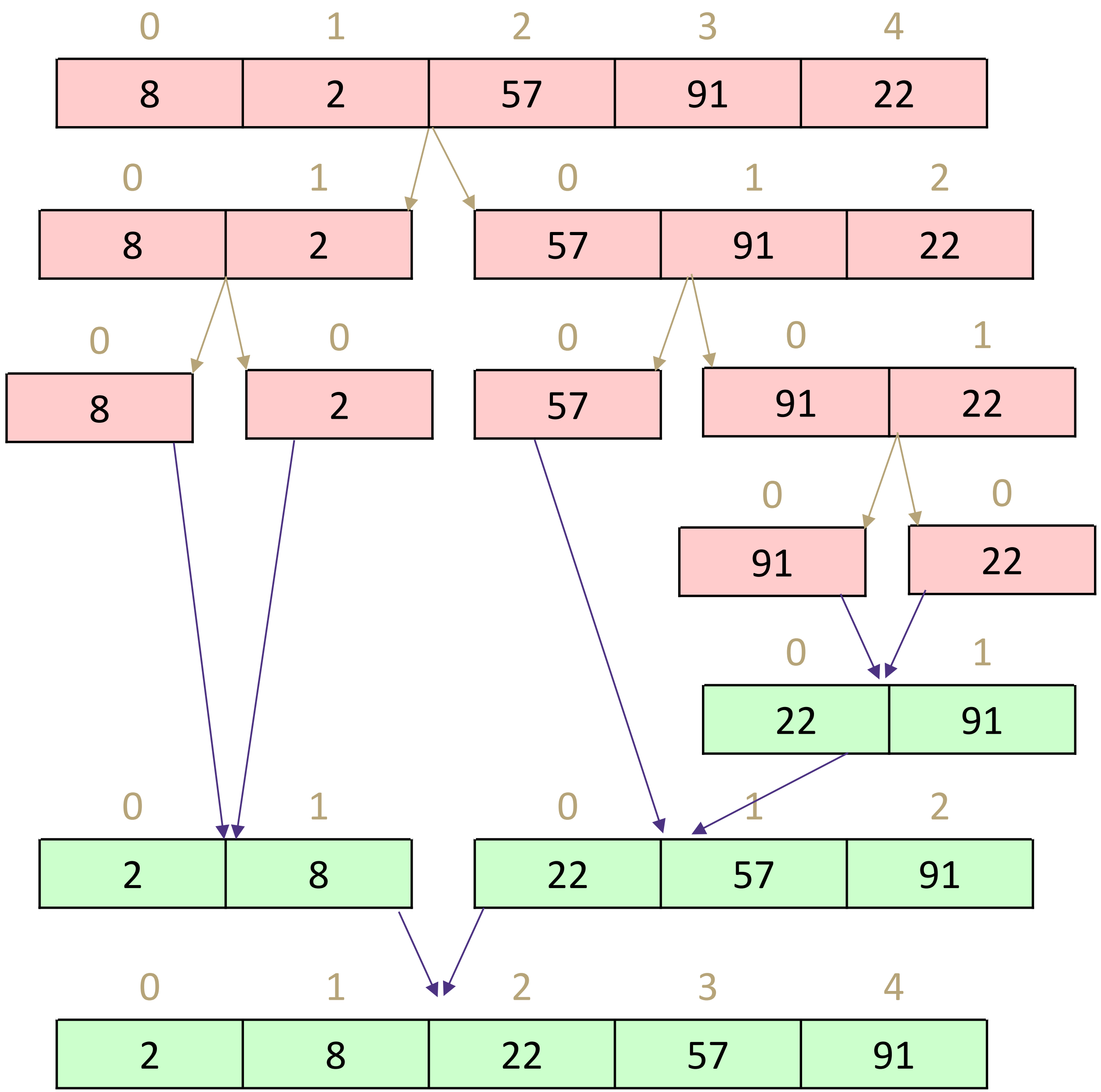
$$T(n) = \begin{cases} 1 \text{ if } n<= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases}$$

Pattern #2 – Constant size input and doing work

Take 1 min to respond to activity

www.pollev.conm/cse373activity

Take a guess! What is the Big-O of worst case merge sort?



Merge Sort Recurrence to Big-Θ

$$T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases}$$

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	$T(n) \in \Theta\left(n^{\log_b a}\right)$

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

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

Questions

Recursive Patterns

Pattern #1: Halving the Input

Binary Search $\Theta(\log n)$

Pattern #2: Constant size input and doing work

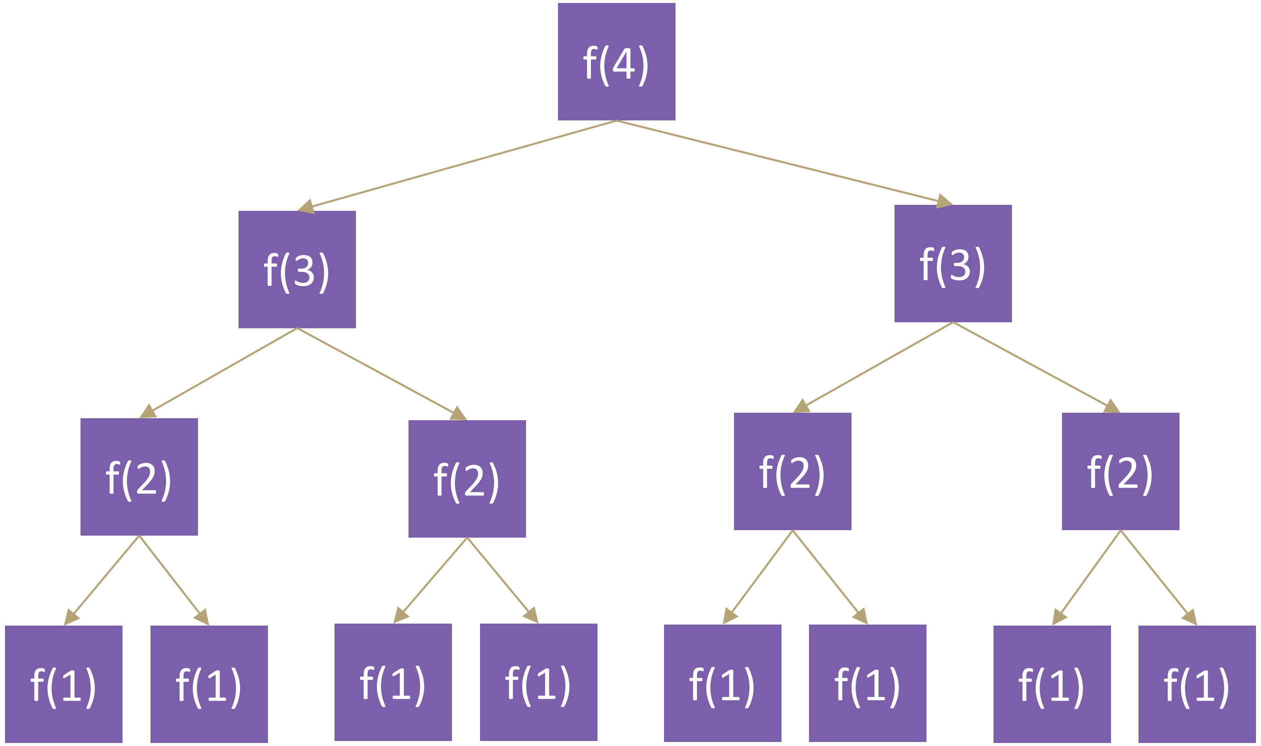
Merge Sort $\Theta(n \log n)$

Pattern #3: Doubling the Input

Calculating Fibonacci

Calculating Fibonacci

```
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

Pattern #3 – Doubling the Input Almost

Calculating Fibonacci Recurrence to Big-Θ

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

d

2T(n-1) + c

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

Take 1 min to respond to activity

www.pollev.conm/cse373activity
Finish the recurrence, what is the model for the recursive case?

Can we use 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 tough
  - Maybe geometry can help!

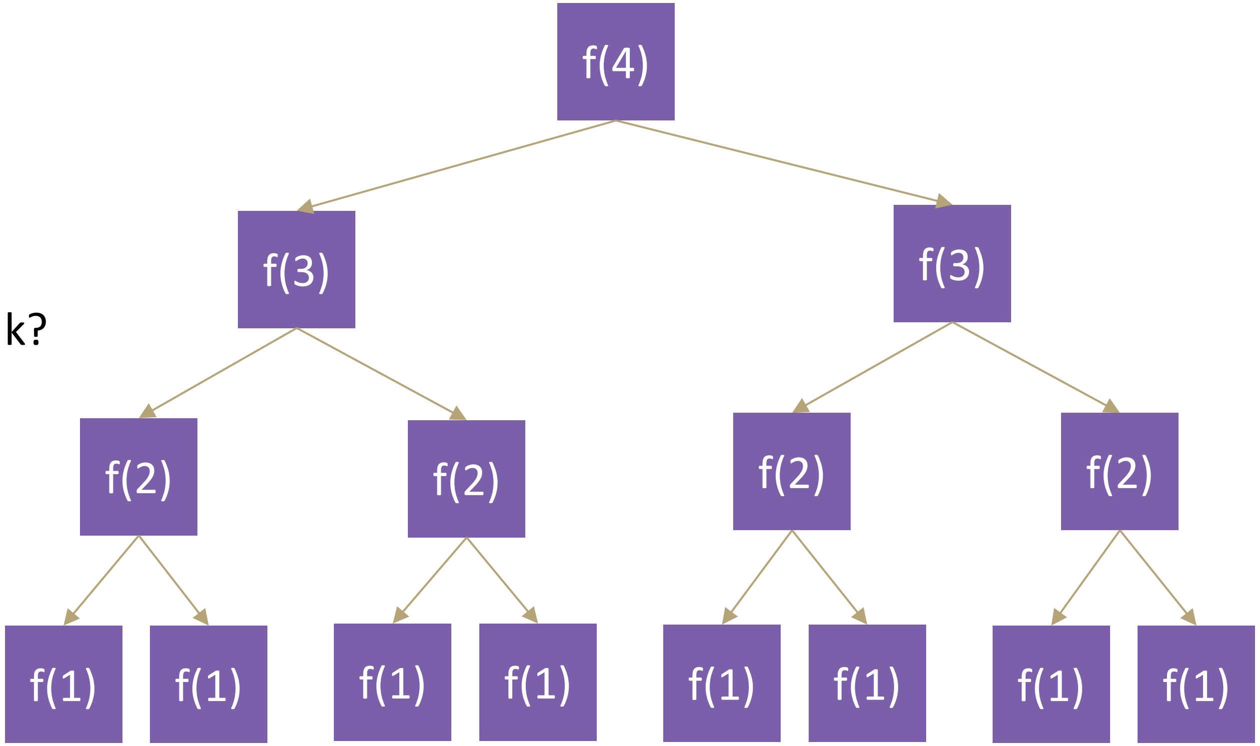
Calculating Fibonacci Recurrence to Big-Θ

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

How many layers in the function call tree?

How many layers will it take to transform "n" to the base case of "1" by subtracting 1

For our example, 4 -> Height = n



How many function calls per layer?

Layer	Function calls
1	1
2	2
3	4
4	8

How many function calls on layer k?

$\mathbf{2^{k-1}}$

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

$\mathbf{1 + 2 + 3 + 4 + \ldots + 2^{k-1}}$

Calculating Fibonacci Recurrence to Big-Θ

Patterns found:

How many layers in the function call tree? n

How many function calls on layer k? 2^{k-1}

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

$1 + 2 + 4 + 8 + \ldots + 2^{k-1}$

Total runtime = (total function calls) x (runtime of each function call)

Total runtime = $(1 + 2 + 4 + 8 + \ldots + 2^{k-1})$ x (constant work)

$$1 + 2 + 4 + 8 + \ldots + 2^{k-1} = \sum_{i=1}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$$

Summation Identity

Finite Geometric Series

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

$$\boldsymbol{T(n) = 2^n - 1 \in \Theta(2^n)}$$

Recursive Patterns

Pattern #1: Halving the Input

Binary Search Θ(logn)

Pattern #2: Constant size input and doing work

Merge Sort Θ(nlogn)

Pattern #3: Doubling the Input

Calculating Fibonacci Θ(2^n)