Answer the Warm Up:
PollEv.com/champk

Lecture 6: Analyzing Recursive Code

CSE 373: Data Structures and Algorithms

Warm Up

Approach

-> start with basic operations, work inside out for control structures

- Each basic operation = +1
- Conditionals = test operations + appropriate branch
- Loop = iterations * loop body
- runtime of other methods roll into total sum

Answer the Warm Up:
PollEv.com/champk

Use asymptotic analysis AND case analysis to determine the upper and lower bounds of both the best case and the worst case for the following method

```
public void mystery(int n, HashSet<Integer> primes) {
    if (primes.contains(n)) {        O(1)
        return true;        +1
    } else {
        int toTest = 2;        +1
        while(toTest < Math.sqrt(n)){
            if(n % toTest == 0) {        +2
                return false;        +1
            } else {
                toTest++;        +1
            }
        }
        primes.add(n);  O(1)
        return true;        +1
    }
}
```

Ω(1)

O(n)

Best Case O(1)

Worst Case O(n) Ω(1)

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

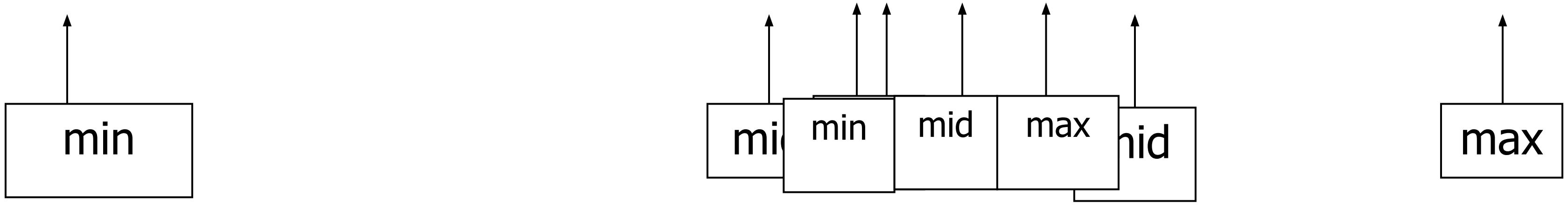
```
public int binarySearch(int[] arr, int toFind, int lo, int hi) {
   if( hi < lo ) {
      return -1;
   } if(hi == lo) {
      if(arr[hi] == toFind) {
         return hi;
      }
      return -1;
   }
   int mid = (lo+hi) / 2;
   if(arr[mid] == toFind) {
      return mid;
   } else if(arr[mid] <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



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 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".

$2^x = N$

$\mathbf{x = \log_2 N}$

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

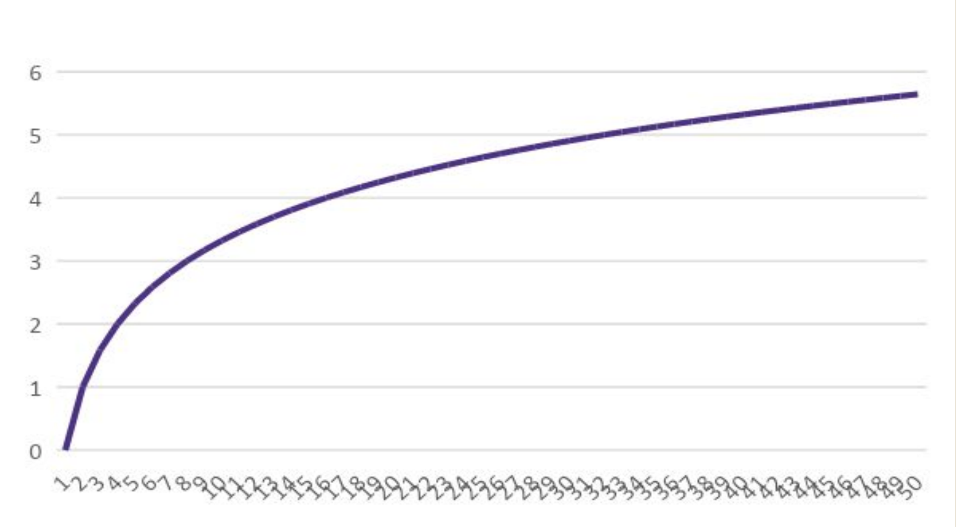
Logarithm – inverse of exponentials

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

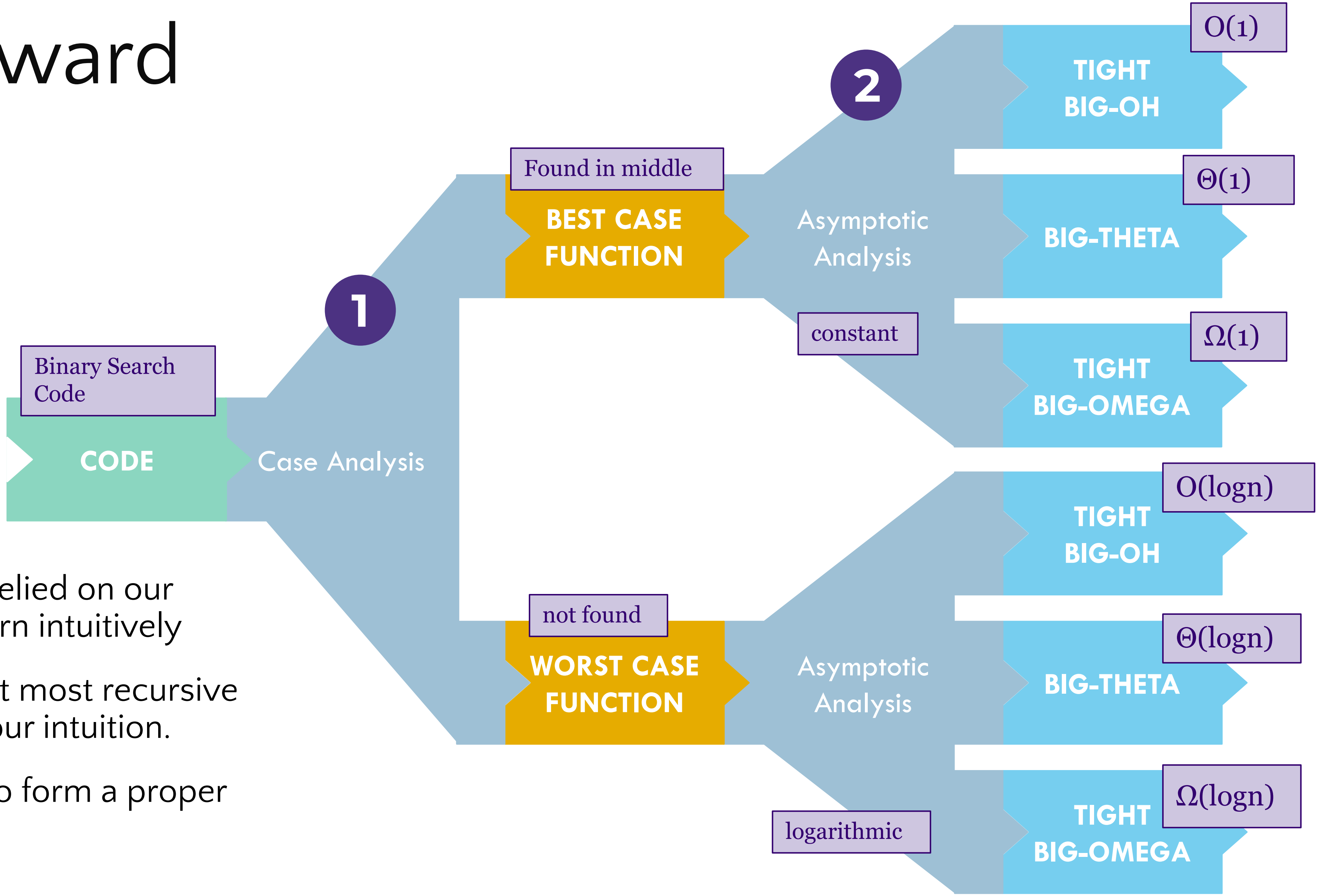
Examples:

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

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



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.

Binary Search Code Model

Approach
- Model the base case first (non recursive)
- Model any non-recursive work in the recursive case
- Write a recursive math function for recursive work

Let F(n) represent the worst case runtime for binary search

```
public int binarySearch(int[] arr, int toFind, int lo, int hi) {
    if( hi < lo ) {          +1
        return -1;  +1
    } if(hi == lo) {  +1
        if(arr[hi] == toFind) {  +2
            return hi;  +1
        }
        return -1;  +1
    }
    int mid = (lo+hi) / 2;  +3
    if(arr[mid] == toFind) {  +2
        return mid;  +1
    } else if(arr[mid] < toFind) {  +2
        return binarySearch(arr, toFind, mid+1, hi);  +?
    } else {
        return binarySearch(arr, toFind, lo, mid-1);  +?
    }
}
```

base case
best case: +2
worst case: +5

recursive case
best case: +6
worst case: +7 + ???

$$F(n) = \begin{cases} 5 & if\ (hi < lo) \\ F\left(\frac{n}{2}\right) + 7 & otherwise \end{cases}$$

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

$$F(n) = \begin{cases} base\ case & if\ BaseCaseTest \\ numRecursiveCalls * F(NReducingToBaseCase) + nonRecursiveWork & otherwise \end{cases}$$

Write a Recurrence

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

+1

+1

base case: +2

+1

*n

non-recursive work: n+2

recursive work: 2F(n/3)

+2

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

Recurrence to Big Θ Techniques

A recurrence is a mathematical function that includes itself in its definition

This makes it very difficult to find the dominating term that will dictate the asymptotic growth

Solving the recurrence or "finding the closed form" is the process of eliminating the recursive definition. So far, we've seen three methods to do so:

1. Apply Master Theorem
   - Pro: Plug and chug convenience
   - Con: only works for recurrences of a certain format
2. Unrolling
   - Pro: Least complicated setup
   - Con: requires intuitive pattern matching
3. Tree Method
   - Pro: Plug and chug
   - Con: Complex setup

Master Theorem

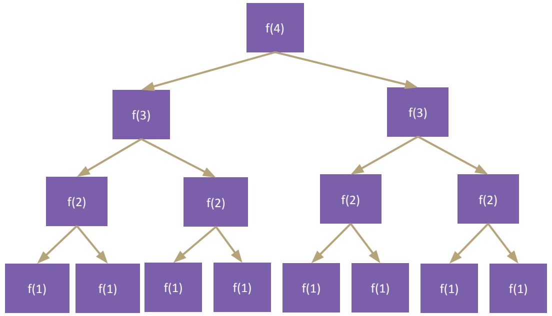
$$F(n) = \begin{cases} d & \textit{if } n \textit{ is at most some constant} \\ \boldsymbol{a}\, F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^c) & \textit{otherwise} \end{cases}$$

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

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



Recurrence to Big-Θ

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

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!

<u>Master Theorem</u>

$$F(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ \boldsymbol{a}\,F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{\boldsymbol{c}}) & \text{otherwise} \end{cases}$$

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

Master Theorem

$$F(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ \boldsymbol{a}\, F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{\boldsymbol{c}}) & \text{otherwise} \end{cases}$$

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 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 of 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 of a = c

- 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 of 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

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

+2

+1

base case: +3

n/2

n/2

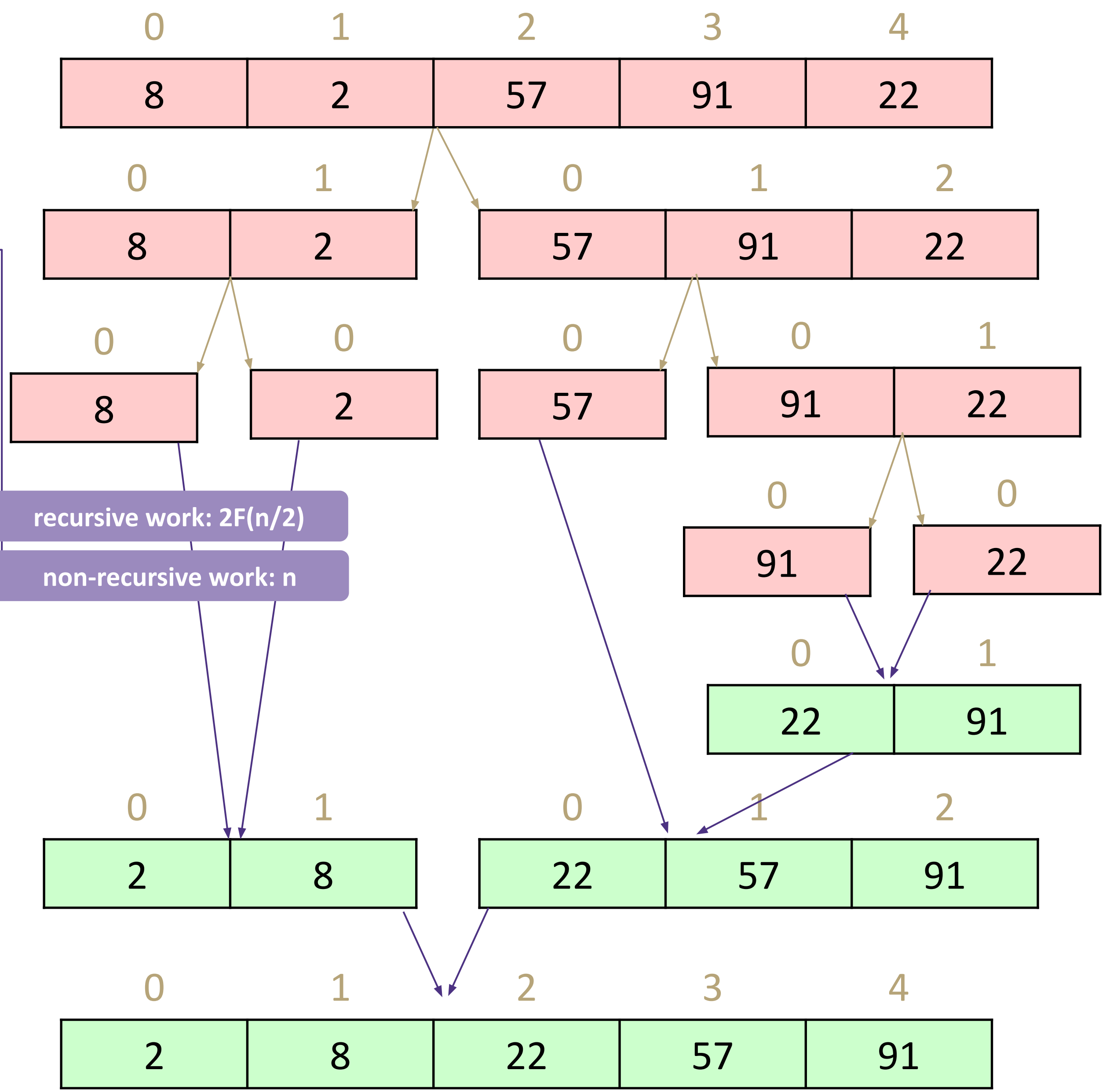
+n

recursive work: 2F(n/2)

non-recursive work: n

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

Pattern #2 – Constant size input and doing work



Merge Sort Recurrence to Big-Θ

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

Master Theorem

$$F(n) = \begin{cases} d & if\ n\ is\ at\ most\ some\ constant \\ \boldsymbol{a}\,F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{\boldsymbol{c}}) & otherwise \end{cases}$$

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

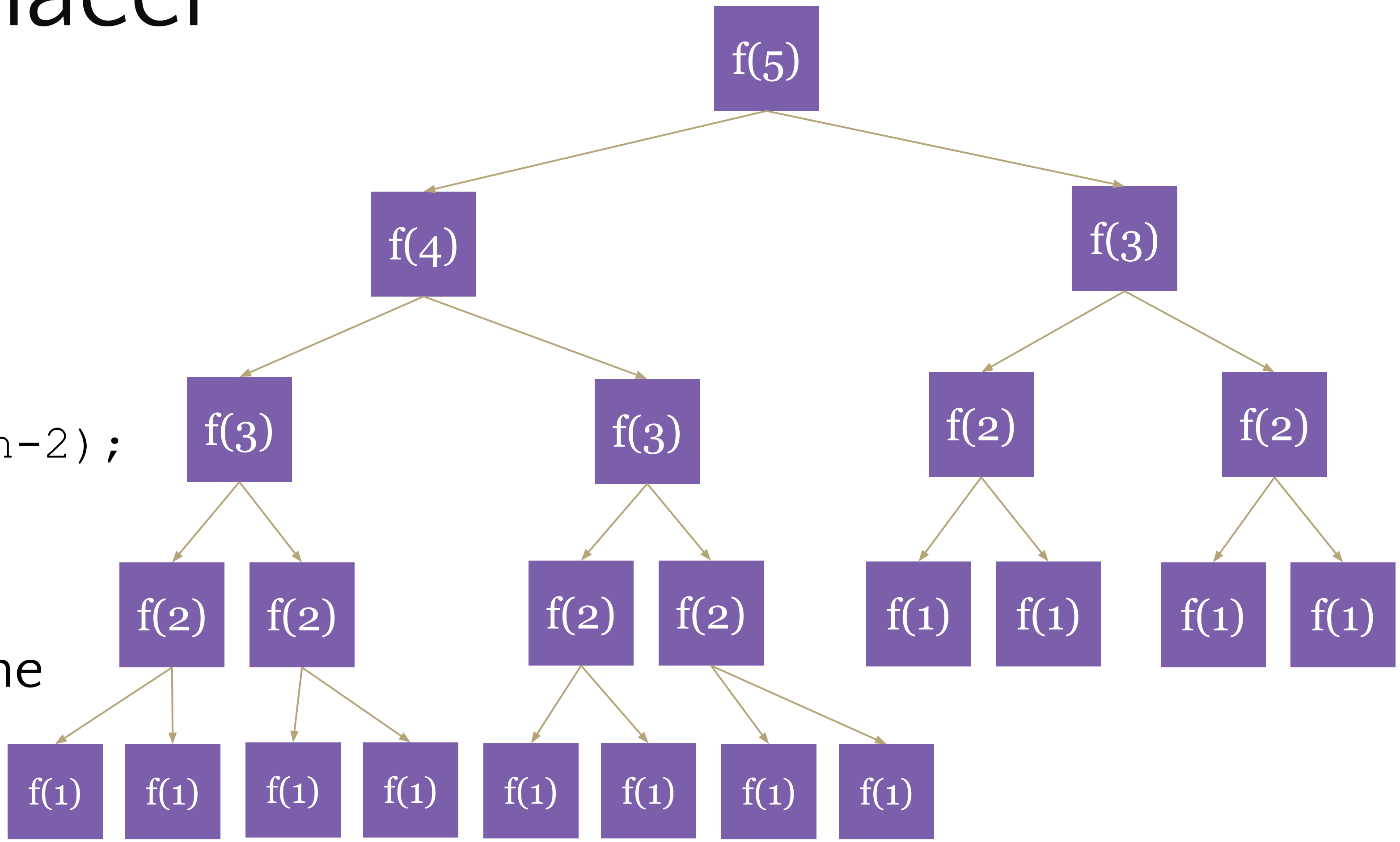
  Merge Sort – $\Theta(n \log n)$

- **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-2);
}
```



- 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 fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

C_0

$2T(n-C_1) + C_2$

$$F(n) = \begin{cases} C_0 & if\ n < 2 \\ 2F(n - C_1) + C_2 & otherwise \end{cases}$$

Apply Master Theorem...

Master Theorem

$$F(n) = \begin{cases} d & if\ n\ is\ at\ most\ some\ constant \\ \boldsymbol{a}\,F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{c}) & otherwise \end{cases}$$

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

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

Can we intuit a pattern? ("**unrolling**")

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!

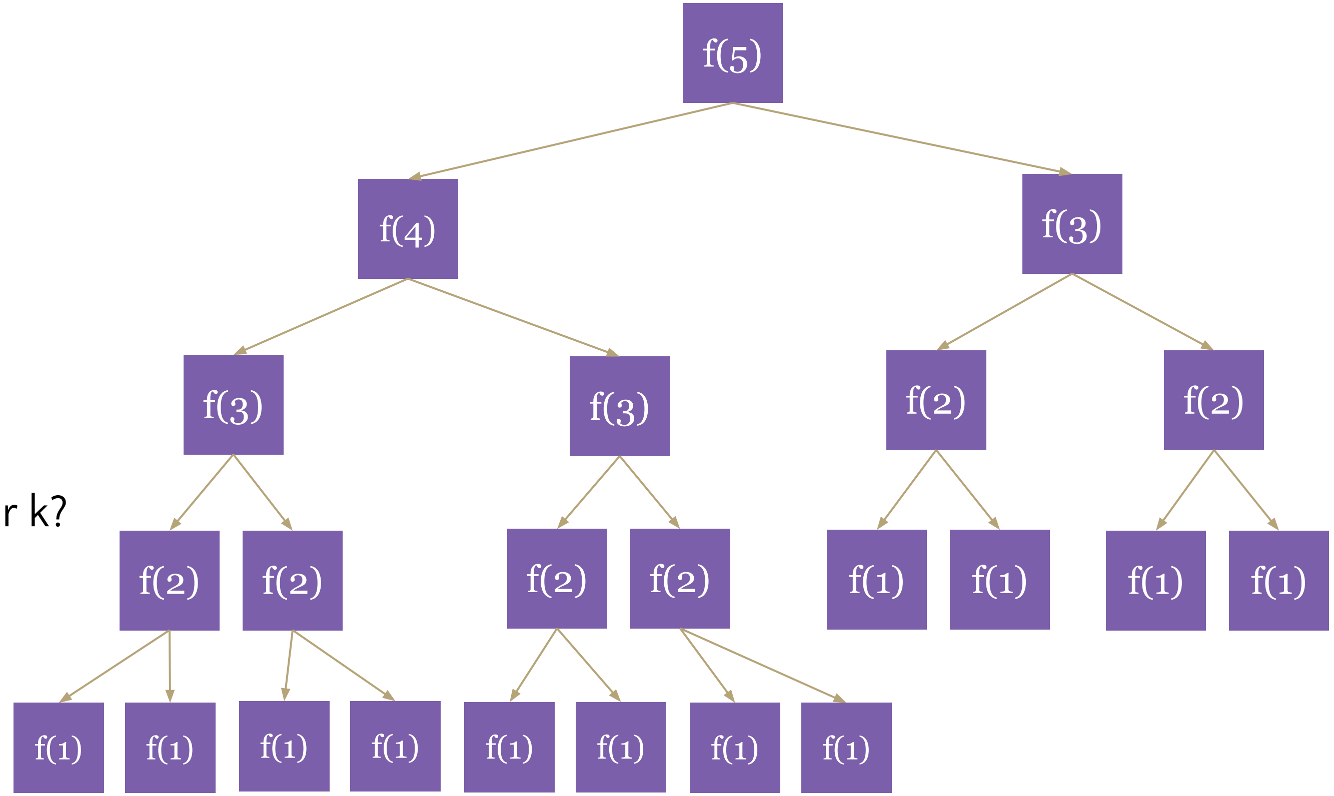
Calculating Fibonacci Recurrence to Big-Θ

$$F(n) = \begin{cases} C_0 & if\ n < 2 \\ 2F(n - C_1) + C_2 & otherwise \end{cases}$$

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?

2^{k-1}

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

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

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)

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

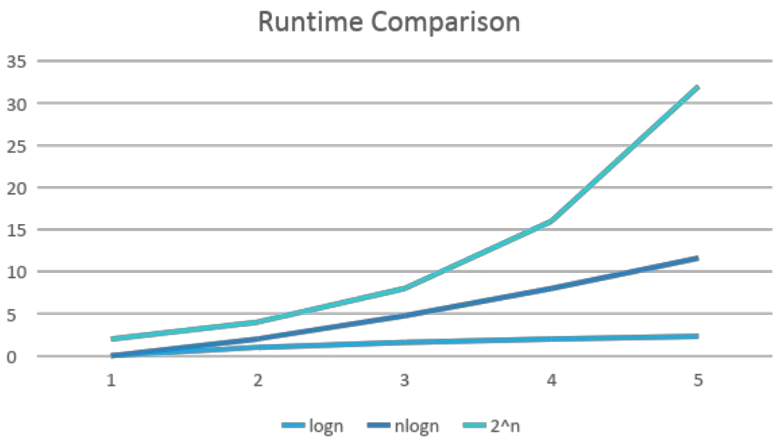
Summation Identity
Finite Geometric Series

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

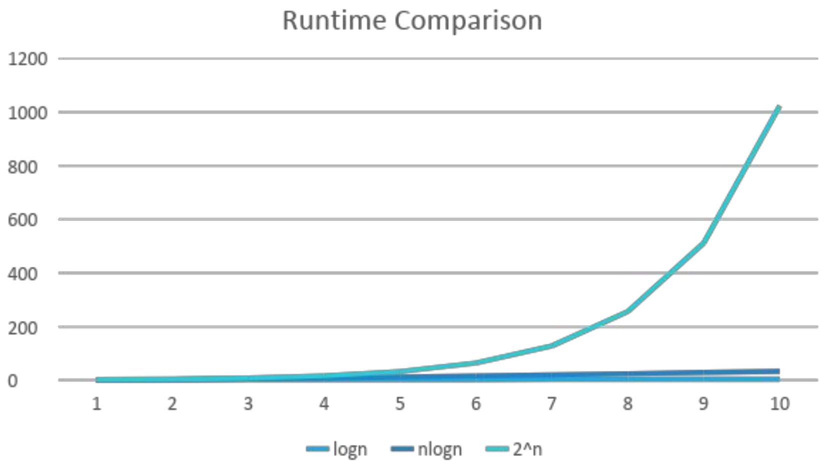
$$\boxed{\boldsymbol{T(n) = 2^n - 1 \in \Theta(2^n)}}$$

Recursive Patterns

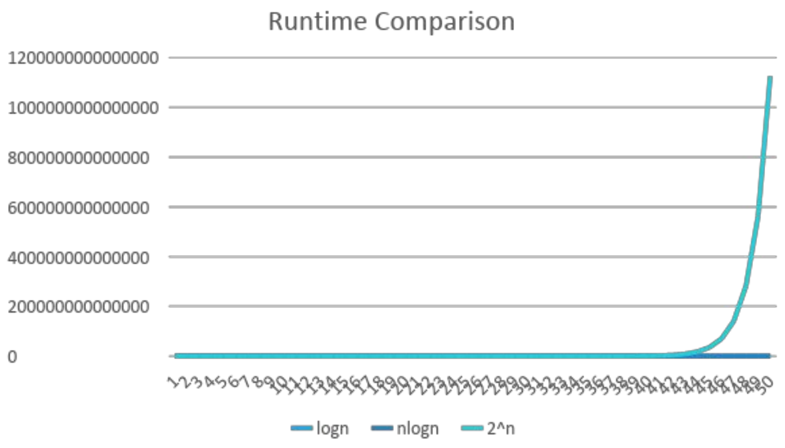
Pattern #1: Dividing the Input
Binary Search $\Theta(\log_2 n)$



Pattern #2: Constant input + work
Merge Sort $\Theta(n\log n)$



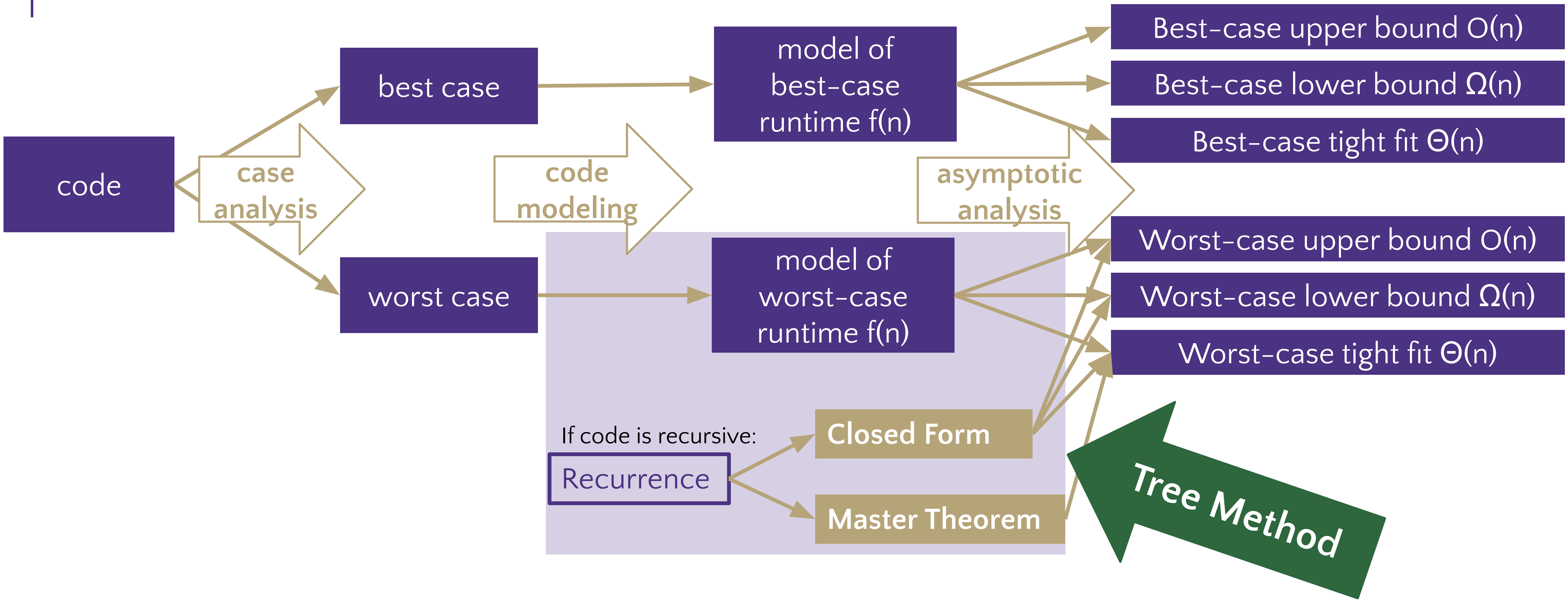
Pattern #3: Exponential Input
Calculating Fibonacci $\Theta(2^n)$



Appendix

Extra slides

Code Analysis Process



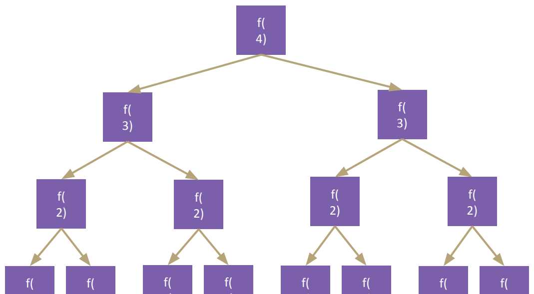
Recurrence to Big Θ Techniques

A recurrence is a mathematical function that includes itself in its definition

This makes it very difficult to find the dominating term that will dictate the asymptotic growth

Solving the recurrence or "finding the closed form" is the process of eliminating the recursive definition. So far, we've seen three methods to do so:

1. Apply Master Theorem
   - Pro: Plug and chug convenience
   - Con: only works for recurrences of a certain format
2. Unrolling
   - Pro: Least complicated setup
   - Con: requires intuitive pattern matching
3. Tree Method
   - Pro: Plug and chug
   - Con: Complex setup



Master Theorem

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

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
T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c
T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d +25c

Tree Method

Draw out call stack, what is the input to each call? How much work is done by each call?

How much work is done at each layer?

- 64 for this example -> n work at each layer
- Work is variable per layer, but across the entire layer work is constant – always n

How many layers are in our function call tree?

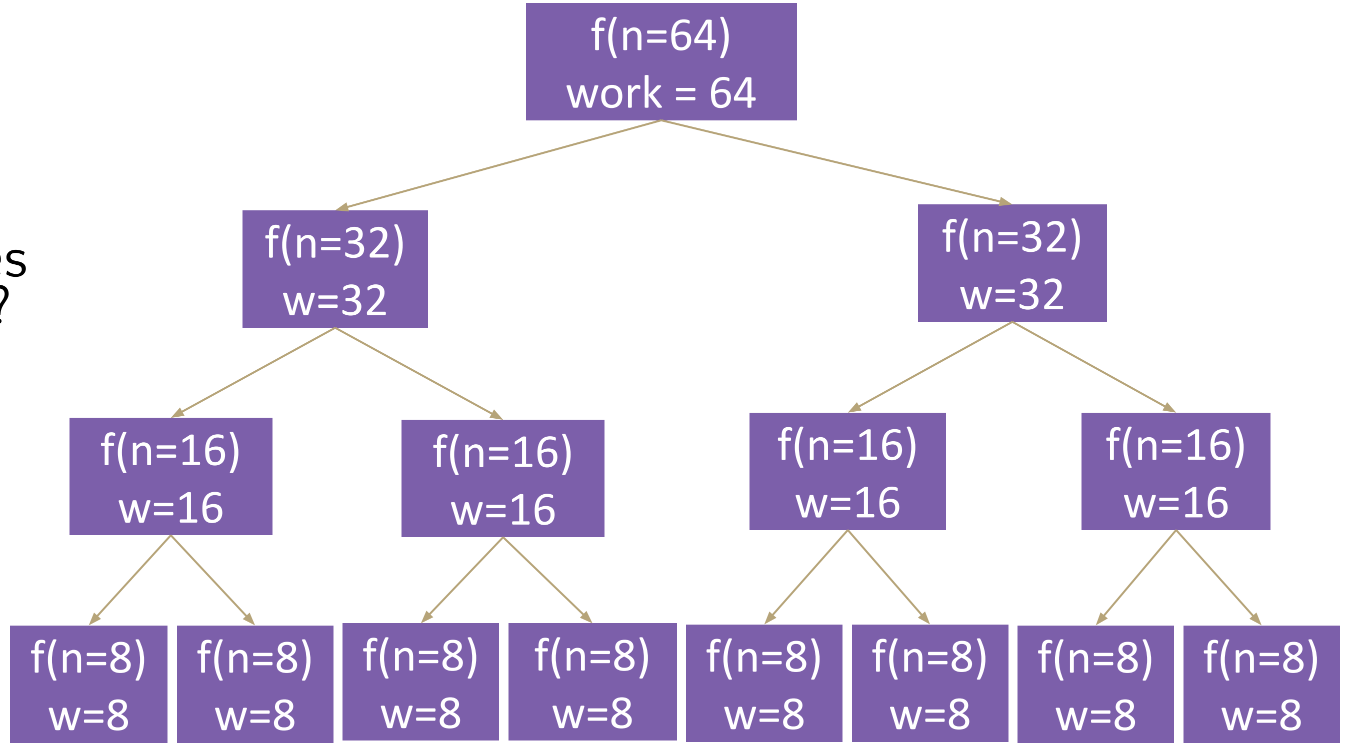
Hint: how many levels of recursive calls does it take *binary search* to get to the base case?

Height = $\log_2 n$

It takes $\log_2 n$ divisions by 2 for n to be reduced to the base case 1

$\log_2 64 = 6$ -> 6 levels of this tree

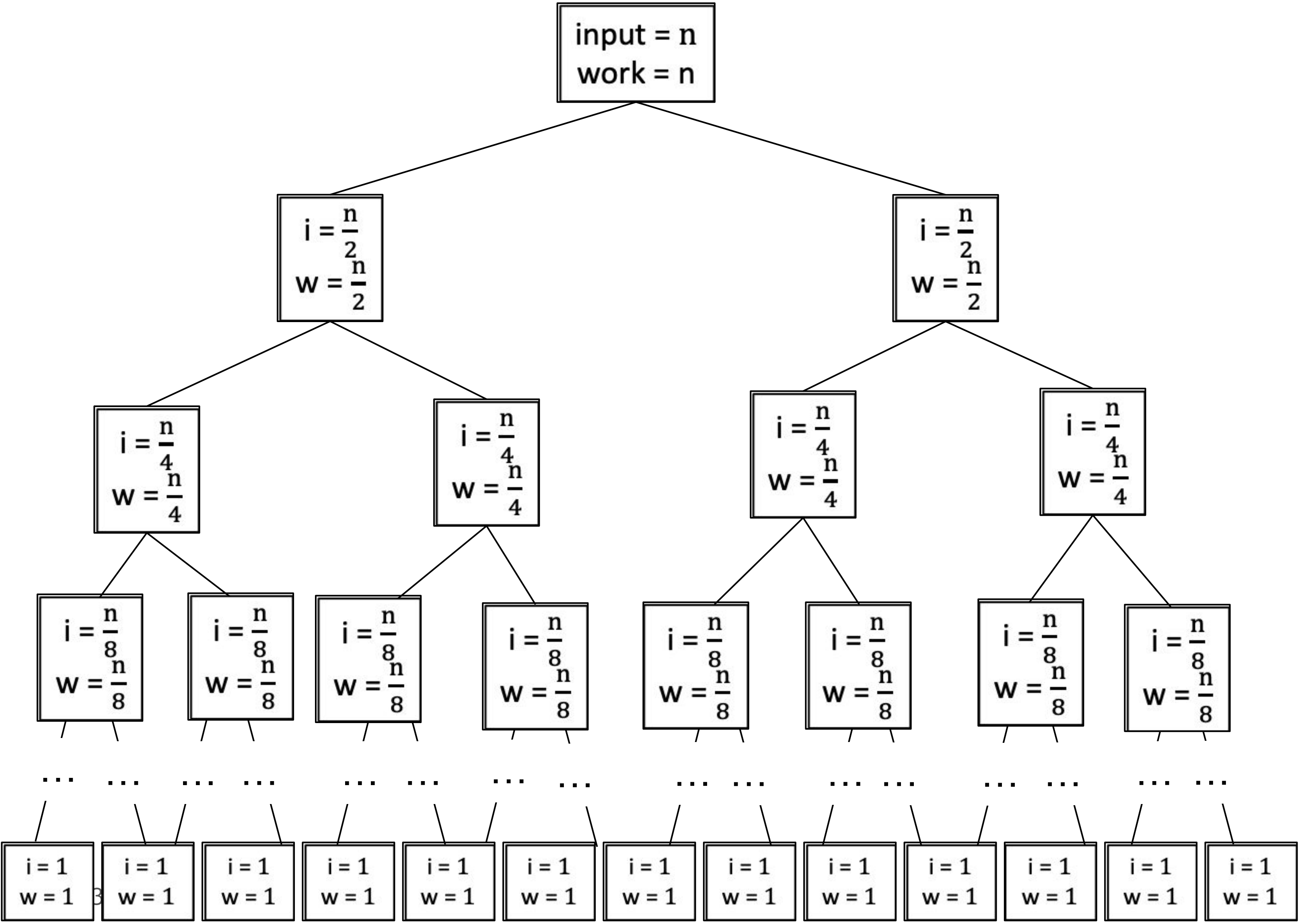
Merge Sort $T(n) = \begin{cases} 1 \text{ if } n \le 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases}$



… and so on…

Tree Method

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



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 Practice

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

1. What is the size of the input on level i? $\frac{n}{2^i}$

2. What is the work done by each node on the i^{th} recursive level? $\left(\frac{n}{2^i}\right)$

3. What is the number of nodes at level i? 2^i

4. What is the total work done at the i^{th} recursive level?

$$numNodes * workPerNode = 2^i\left(\frac{n}{2^i}\right) = n$$

5. What value of i does the last level occur?

$$\frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow i = \log_2 n$$

6. What is the total work across the base case level?

$$numNodes * workPerNode = 2^{log_2 n}(1) = n$$

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

Combining it all together…

$$T(n) = \sum_{i=0}^{\log_2 n - 1} n \;+ n \;= nlog_2 n + n \;= \Theta(\text{nlogn})$$

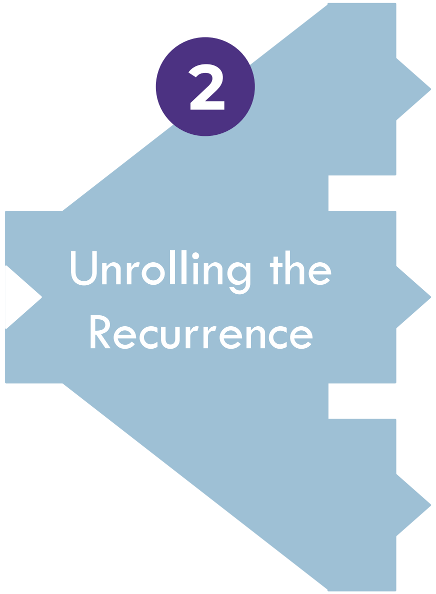
power of a log

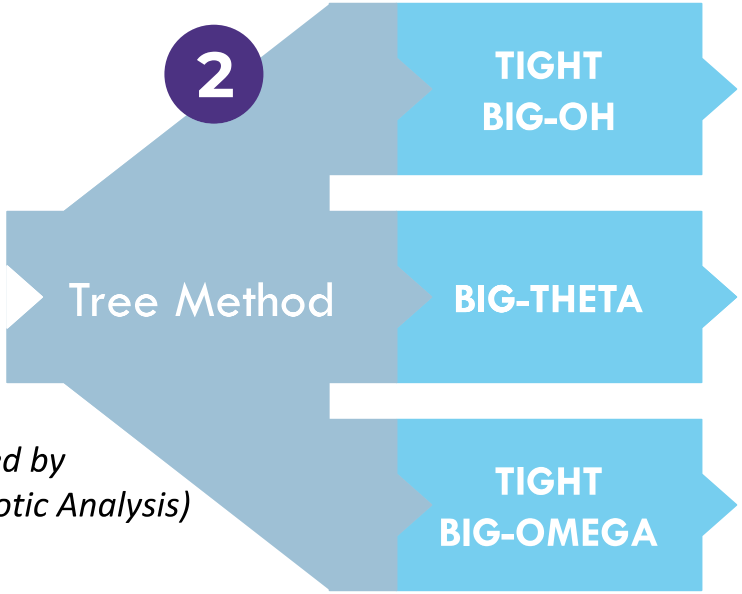
$$x^{\log_b y} = y^{\log_b x}$$

Summation of a constant

$$\sum_{i=0}^{n-1} c = cn$$

Recurrence to Big-Theta: Our Toolbox



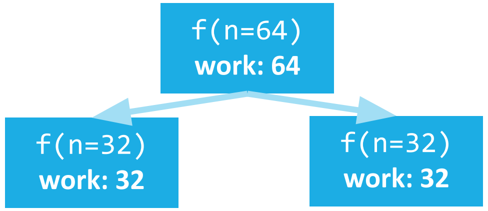


(followed by Asymptotic Analysis)

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

Case Analysis

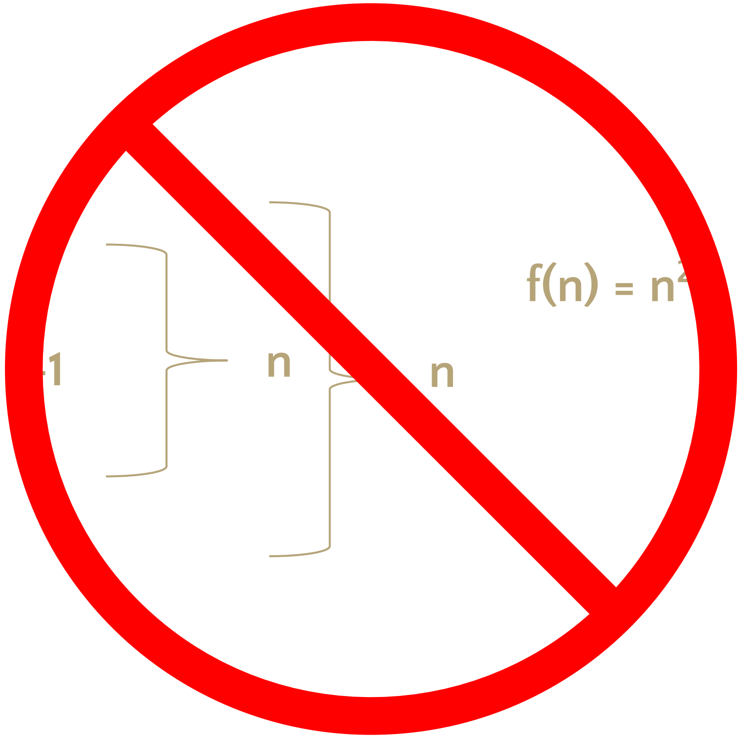
Modeling Recursive Code

Summations

Modeling Complex Loops

Write a mathematical model of the following code

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



Keep an eye on loop bounds!

Modeling Complex Loops

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

+1

0 + 1 + 2 + 3 +…+ i-1

n

$$T(n) = \underbrace{(0 + 1 + 2 + 3 + \ldots + i-1)}$$

How do we model this part?

Summations!

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

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

What is the Big O?

Definition: Summation

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

Simplifying Summations

Find closed form using summation identities (given on exams)

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

→ $T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1$ → closed form → simplified tight big O

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

Summation of a constant

$$\sum_{i=0}^{n-1} c = cn$$

Factoring out a constant

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

Gauss's Identity

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