Lecture 6: Analyzing Recursive Code

CSE 373: Data Structures and Algorithms
Warm Up

Which of the following statements are FALSE?

Select all options that apply:

- A Big-Theta bound will exist for every function
- If a function is $O(n^2)$ it can’t also be $\Omega(n^2)$
- A piece of code whose model is $f(n) = 3n + 6$ has a simplified tight BigO of $O(n^2)$
- The tight upper and lower bound of piece of code’s runtime growth is always the same complexity class

All false!
Announcements

- Project 1 is out
  - Due Wednesday April 12th

- Exercise 1 is out
  - Due Monday April 10th
Questions?
Case Analysis

Modeling Recursive Code

Summations
Review: Algorithmic Analysis Roadmap

1. CODE
   - Code Modeling

2. RUNTIME FUNCTION
   - Asymptotic Analysis

TIGHT
- BIG-O

BIG-THETA
- TIGHT BIG-OMEGA
Case Study: Linear Search

```java
int linearSearch(int[] arr, int toFind) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == toFind)
            return i;
    }
    return -1;
}
```

The number of operations doesn’t depend just on \( n \).
Even once you fix \( n \) (the size of the array) there are still a number of cases to consider.

- If \( toFind \) is in \( arr[0] \), we’ll only need one iteration, \( f(n) = 4 \).
- If \( toFind \) is not in \( arr \), we’ll need \( n \) iterations. \( f(n) = 3n + 1 \).
And there are a bunch of cases in-between.
\( f(n) = 2 \)  

Best Case  
On Lucky Earth

\( f(n) = 3n + 1 \)  

Worst Case  
On Unlucky Earth (where it’s 2020 every year)

arr  
\[ \begin{bmatrix} 2 & 3 & 9 & 4 & 5 \end{bmatrix} \]

toFind 2

\( O(1) \quad \Theta(1) \quad \Omega(1) \)

After asymptotic analysis:

arr  
\[ \begin{bmatrix} 2 & 3 & 9 & 4 & 5 \end{bmatrix} \]

toFind 8

\( O(n) \quad \Theta(n) \quad \Omega(n) \)

After asymptotic analysis:
Case Analysis

**Case**: a description of inputs/state for an algorithm that is specific enough to build a code model (runtime function) whose only parameter is the input size

- Case Analysis is our tool for reasoning about **all variation other than n**!
- Occurs during the code function step instead of function $O/\Omega/\Theta$ step!

- **Best Case** = fastest that our code could finish on input of size $n$
- **Worst Case** = slowest that our code could finish on input of size $n$.
- Importantly, *any* position of `toFind` in `arr` could be its own case!
  - For this simple example, probably don’t care (they all still have bound $O(n)$)
  - But intermediate cases will be important later
Caution: Common Misunderstanding

Best/Worst case is based on all variation other than value of n

Incorrect – based on specific values of n

- “The best case is when n=1, worst is when n=infinity”
- “The best case is when front is null”
- “The best case is when overallRoot is null”
- “The best case is when n is an even number”

Correct – based on state of data structure regardless of n

- “The best case is when the node I’m looking for is at front, the worst is when it’s not in the list”
- “The best case is when the BST is perfectly balanced, the worst is when it’s a single line of nodes”
Other cases

“Assume X won’t happen” case
● Assuming our array won’t need to resize is the most common example

“Average” case
● Assuming your input is random
● Need to specify what the possible inputs are and how likely they are
● \( f(n) \) is now the average number of steps on a random input of size \( n \)

“In-practice” case
● This isn’t a real term (I just made it up)
● Making some reasonable assumptions about how the real-world is probably going to work
● We’ll tell you the assumptions and won’t ask you to come up with these assumptions on your own
● Then do worst-case analysis under those assumptions

All of these can be combined with any of \( O, \Omega, \) and \( \Theta \)
How to do case analysis

1. Look at the code, understand how thing could change depending on the state of input
   - How can you exit loops early?
   - Can you return (exit the method) early?
   - Are some if/else branches much slower than others?

2. Figure out what input values can cause you to hit the (best/worst) parts of the code.
   - not to be confused with number of inputs

3. Now do the analysis like normal!
Algorithmic Analysis Roadmap

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

Review Algorithmic Analysis Roadmap

Case Analysis

1. BEST CASE FUNCTION
   - f(n) = 2

2. WORST CASE FUNCTION
   - f(n) = 3n+1

OTHER CASE FUNCTION

Asymptotic Analysis

TIGHT BIG-O

BIG-THETA

TIGHT BIG-OMEGA

O(1)

Θ(1)

Ω(1)
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
Case Analysis
Modeling Recursive Code
Summations
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
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: 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:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

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 \( \frac{1}{2} \) until 1 element remains.

\[ N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2, 1 \]

○ 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 \( N \)?

\[ 1, 2, 4, 8, \ldots, \frac{N}{4}, \frac{N}{2}, N \]

○ 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:

\[ 2^2 = 4 \Rightarrow 2 = \log_2 4 \]
\[ 3^2 = 9 \Rightarrow 2 = \log_3 9 \]
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.

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

```java
public int recursiveFunction(int n) {
    if (n < 3) {
        return 3; // base case: +2
    }
    for (int i = 0; i < n; i++) {
        System.out.println(i); // non-recursive work: n+2
    }
    int val1 = recursiveFunction(n/3); // recursive work: 2T(n/3)
    int val2 = recursiveFunction(n/3);
    return val1 * val2; // T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T(\frac{n}{3}) + 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**

\[ T(n) = \begin{cases} 
d & \text{if } n \text{ is at most some constant} \\
\frac{a}{b} n + f(n) & \text{otherwise}
\end{cases} \]

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

\[ a=2 \quad b=3 \quad \text{and} \quad c=1 \]

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

\[ \log_3 2 = x \Rightarrow 3^x = 2 \Rightarrow x \approx 0.63 \]

\[ \log_3 2 < 1 \]

We’re in case 1

\[ T(n) \in \Theta(n) \]
Understanding Master Theorem

Master Theorem

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

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 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 $\Theta(n \log n)$

● Pattern #2: Constant size input and doing work

  Merge Sort

● Pattern #3: Doubling the Input
Merge Sort

Divide

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>91</td>
<td>22</td>
<td>57</td>
<td>1</td>
<td>10</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Conquer

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

Combine

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>22</td>
<td>57</td>
<td>91</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>22</td>
<td>57</td>
<td>91</td>
</tr>
</tbody>
</table>
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(n/2) + n & \text{otherwise}
\end{cases}

Pattern #2 – Constant size input and doing work

Take a guess! What is the Big-O of worst case merge sort?
Merge Sort Recurrence to Big-$\Theta$

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

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

$a=2 \ b=2 \ and \ c=1$

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

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

$\log_2 2 = 1$

We’re in case 2

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

```java
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
Calculating Fibonacci Recurrence to Big-$\Theta$

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

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

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? (**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-$\Theta$

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?

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

How many function calls on layer k?

$2^{k-1}$

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

$1 + 2 + 3 + 4 + ... + 2^{k-1}$

$T(n) = \begin{cases} 
  d & \text{when } n \leq 1 \\
  2T(n-1) + c & \text{otherwise}
\end{cases}$
Calculating Fibonacci Recurrence to Big-$\Theta$

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 $= (\text{total function calls}) \times (\text{runtime of each function call})$

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

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

$T(n) = 2^n - 1 \in \Theta(2^n)$
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 $\Theta(2^n)$
Questions?
That’s all!
Appendix

Extra slides
Code Analysis Process

- **Case Analysis**
  - **Best Case**
    - Code Modeling
    - Model of Best-Case Runtime $f(n)$
    - Asymptotic Analysis
    - Best-Case Upper Bound $O(n)$
    - Best-Case Lower Bound $\Omega(n)$
    - Best-Case Tight Fit $\Theta(n)$
  - **Worst Case**
    - Code Modeling
    - Model of Worst-Case Runtime $f(n)$
    - Asymptotic Analysis
    - Worst-Case Upper Bound $O(n)$
    - Worst-Case Lower Bound $\Omega(n)$
    - Worst-Case Tight Fit $\Theta(n)$
- If code is recursive:
  - **Recurrence**
  - **Closed Form**
  - **Master Theorem**
  - **Tree Method**
Recurrence to Big $\Theta$ 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

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

- $T(1) = d$
- $T(2) = 2T(2-1) + c = 2(d) + c$
- $T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c$
- $T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c$
- $T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d + 25c$
How much work is done at each layer?

- 64 for this example \( \rightarrow \) 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

\( \log_2 64 = 6 \rightarrow \) 6 levels of this tree

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]
Tree Method

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

Recursive level | How many nodes at each level? | How much work done by each node? | How much work done 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

...
Tree Method Practice

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?
   \[ \frac{n}{2^i} \]

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?
   \[
   \text{numNodes} \times \text{workPerNode} = 2^i \left( \frac{n}{2^i} \right) = n
   \]

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?
   \[
   \text{numNodes} \times \text{workPerNode} = 2^{\log_2 n} (1) = n
   \]

Combining it all together...

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

<table>
<thead>
<tr>
<th>Level (i)</th>
<th>Number of Nodes</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>( n/2 )</td>
<td>( n )</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>( n/4 )</td>
<td>( n )</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>( n/8 )</td>
<td>( n )</td>
</tr>
<tr>
<td>( \log_2 n )</td>
<td>( n )</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

\[
T(n) = \sum_{i=0}^{\log_2 n - 1} n + n = nlog_2 n + n = \Theta(n\log n)
\]

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

**MASTER THEOREM**

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

**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

**Recurrence Examples:**

- **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**

(followed by Asymptotic Analysis)
Questions?
Case Analysis
Modeling Recursive Code

Summations
Modeling Complex Loops

Write a mathematical model of the following code

```java
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

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

Summations!

\[ \sum_{i=1}^{n} 0 + 1 + 2 + 3 + \ldots + i-1 \]

Definition: Summation

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

How do we model this part?

\[ T(n) = \left( \sum_{i=1}^{n} i \right) \]

What is the Big O?

\[ T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \]
Simplifying Summations

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

\[ T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} 1 \cdot i = 1 \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2} n^2 - \frac{1}{2} n = O(n^2) \]

Find closed form using summation identities (given on exams)

closed form simplified tight big O