Recurrences II, Tree Method
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 after due date
  - Don’t forget your writeup for the P1 experiments

• EX1 (Algo Analysis I) due Friday 10/16 11:59pm PDT

• P2 (Maps) and EX2 (Algo Analysis II) released Friday 10/16
  - Partner 2 Pool form already out, due Thursday 10/15 at 11:59 pm

• We’ll see some summation identities in today’s lecture
  - Summations Reference will be posted as a resource on the calendar
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**

```java
public int recurse(int n) {
    if (n < 3) {
        return 80;  // +2 Base Case
    }

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

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

    return val1 + val2 + val3;  // +3 Recursive Work
}
```

Non-recursive Work: \(+ n + 3\)

Recursive Work: \(+ 3 \cdot T(n/3)\)

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


### Review Why Include Non-Recursive Work?

```java
class Solution {
    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;
    }
}
```

Think of it this way:

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

"work that happens if we enter base case"

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

---

**a=2 b=3 and c=1**

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

\[ (\log_3 2 \approx 0.63) < (c = 1) \]

*We’re in case 1*  
\( T(n) \in \Theta(n) \)
Lecture Outline

• Analyzing Recursive Code: Recursive Patterns

1. Halving the Input
   - Binary Search
   - Θ (log n)

2. Constant size Input
   - Merge Sort

3. Doubling the Input

• Summations
• The Tree Method
**Review** Merge Sort

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

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

### Constant size Input

- 2
**Review**  
Merge Sort Recurrence to Big-Θ

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

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

**Example**

- \( a=2 \), \( b=2 \), and \( c=1 \)

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

\[ \log_2 2 = 1 \]

**We’re in case 2**

\[ T(n) \in \Theta(n \log n) \]
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. Halving the Input
   - Binary Search
   - $\Theta (\log n)$

2. Constant size Input
   - Merge Sort
   - $\Theta (n \log n)$

3. Doubling the Input
   - Fibonacci

• Summations
• The Tree Method
Calculating Fibonacci (ish)

```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

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);
}
```

Can we use the Master Theorem?

<table>
<thead>
<tr>
<th>MASTER THEOREM</th>
</tr>
</thead>
</table>
| \( T(n) = \begin{cases} 
  d & \text{if } n \text{ is at most some constant} \\
  2T(n-1) + c & \text{otherwise} 
\end{cases} \) |

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

Can we intuit a pattern?
- \( T(1) = d \)
- \( T(2) = 2T(2-1) + c = 2(d) + c \)
- \( T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c \)
- \( T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c \)
- \( T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d + 15c \)

Looks like something’s happening, but it’s hard to identify. Maybe geometry can help!
Fibonacci Recurrence to Big-$\Theta$

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

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: $n$

<table>
<thead>
<tr>
<th>LAYER</th>
<th>FUNCTION CALLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 ($= 2^0$)</td>
</tr>
<tr>
<td>1</td>
<td>2 ($= 2^1$)</td>
</tr>
<tr>
<td>2</td>
<td>4 ($= 2^2$)</td>
</tr>
<tr>
<td>3</td>
<td>8 ($= 2^3$)</td>
</tr>
</tbody>
</table>

How many function calls per layer?
How many function calls on layer $i$?
$2^i$
How many function calls TOTAL for a tree of $k$ layers?
$1 + 2 + 4 + ... + 2^{k-1}$
# 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 n layers? & $1 + 2 + 4 + 8 + ... + 2^{n-1}$ \\

Total runtime = (total function calls) * (runtime of each function call) & $(1 + 2 + 4 + 8 + ... + 2^{n-1}) \times \text{(constant work)}$ 

\[
1 + 2 + 4 + 8 + ... + 2^{n-1} = \sum_{i=0}^{n-1} 2^i = \frac{2^n - 1}{2 - 1} = 2^n - 1
\]

\[
T(n) = 2^n - 1 \in \Theta(2^n)
\]
3 Patterns for Recursive Code

1. Halving the Input
   - Binary Search
   - $\Theta (\log n)$

2. Constant size Input
   - Merge Sort
   - $\Theta (n \log n)$

3. Doubling the Input
   - Fibonacci
   - $\Theta (2^n)$
Lecture Outline

• Analyzing Recursive Code: Recursive Patterns

1. Halving the Input
   - Binary Search
     \( \Theta (\log n) \)

2. Constant size Input
   - Merge Sort
     \( \Theta (n \log n) \)

3. Doubling the Input
   - Fibonacci
     \( \Theta (2^n) \)

• Summations
• The Tree Method
Which of these functions is a mathematical model for the runtime of this code?

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

a) \( f(n) = 2n \)
b) \( f(n) = n + n \)
c) \( f(n) = n^2 \)
d) \( f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \)

Keep an eye on the loop bounds!
Modeling Complex Loops

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

Modeling the inner loop:

\[ f(n) = (0 + 1 + 2 + \ldots + i-1) \]

How do we model this part?

Summations:

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

Modeling the entire code snippet:

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

What is the Big-Theta?

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

\[
\begin{align*}
\text{for (int } i = 0; i < n; i++) \{ \\
\quad \text{for (int } j = 0; j < i; j++) \{ \\
\quad\quad \text{System.out.println(“Hello!”);}
\quad \}
\}
\end{align*}
\]

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

**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. Halving the Input
   - Binary Search
     \( \Theta (\log n) \)

2. Constant size Input
   - Merge Sort
     \( \Theta (n \log n) \)

3. Doubling the Input
   - Fibonacci
     \( \Theta (2^n) \)

• Summations

• The Tree Method
Recurrence to Big-Theta: Our Toolbox

MASTER THEOREM

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

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

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

TIGHT BIG-OH
BIG-THETA
TIGHT BIG-Omega
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?**

<table>
<thead>
<tr>
<th>Recursive level</th>
<th>How many nodes at each level?</th>
<th>How much work done by each node?</th>
<th>How much work across each 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>( \frac{n}{2} )</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>( \frac{n}{4} )</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>( \frac{n}{8} )</td>
<td>n</td>
</tr>
<tr>
<td>( \log n )</td>
<td>( n )</td>
<td>1</td>
<td>n</td>
</tr>
</tbody>
</table>
## Tree Method Checklist

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)?**
   
   \[ \left( \frac{n}{2^i} \right) \]

3. **How many nodes at level i?**
   
   \[ 2^i \]

4. **What’s the total work done on level i (recursive case)?**

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

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)?**

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

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

<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>\frac{n}{2}</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>\frac{n}{4}</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>\frac{n}{8}</td>
<td>n</td>
</tr>
<tr>
<td>\log_2 n</td>
<td>n</td>
<td>1</td>
<td>n</td>
</tr>
</tbody>
</table>
# Tree Method Checklist

<table>
<thead>
<tr>
<th></th>
<th>What’s the size of the input per call on level i?</th>
<th>(\frac{n}{2^i})</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>How much work done by each node on level i (recursive case)?</td>
<td>(\left(\frac{n}{2^i}\right)^1)</td>
</tr>
<tr>
<td>3</td>
<td>How many nodes at level i?</td>
<td>(2^i)</td>
</tr>
<tr>
<td>4</td>
<td>What’s the total work done on level i (recursive case)?</td>
<td>(\text{numNodes} \times \text{workPerNode})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(= 2^i \left(\frac{n}{2^i}\right) = n)</td>
</tr>
<tr>
<td>5</td>
<td>On what value of i does the last level occur (base case)?</td>
<td>(\frac{n}{2^i} = 1) (\Rightarrow n = 2^i \Rightarrow i = \log_2 n)</td>
</tr>
<tr>
<td>6</td>
<td>How much work done by each node on last level (base case)?</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>What’s the total work on the last level (base case)?</td>
<td>(\text{numNodes} \times \text{workPerNode})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(= 2^{\log_2 n} (1) = n)</td>
</tr>
</tbody>
</table>

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

Putting it Together:

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

Summation of a Constant

\[
\sum_{i=0}^{k-1} c = ck
\]
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!