Recurrences, Master Theorem

BEFORE WE START
# Announcements

**Project 1 (Deques)** due Wednesday 10/14 11:59pm PDT

**Exercise 1 (written, individual)** due Friday 10/16 11:59pm PDT

---

Remember you can submit **Anonymous Feedback**! Especially in this online world, we are extremely grateful for your insight!

---

<table>
<thead>
<tr>
<th>Week 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 10/05</td>
<td><strong>LEC 05</strong> Stacks, Queues, Maps</td>
</tr>
<tr>
<td>Lesson:</td>
<td>videos &gt; pdf &gt; pptx</td>
</tr>
<tr>
<td>Class Session:</td>
<td>zoom &gt; handout &gt; solution</td>
</tr>
<tr>
<td>Wed 10/07</td>
<td><strong>LEC 04</strong> Asymptotic Analysis</td>
</tr>
<tr>
<td>Lesson:</td>
<td>videos &gt; pdf &gt; pptx</td>
</tr>
<tr>
<td>Class Session:</td>
<td>zoom &gt; handout &gt; solution</td>
</tr>
<tr>
<td>Thu 10/08</td>
<td><strong>LEC 03</strong> Algorithmic Analysis</td>
</tr>
<tr>
<td>Worksheet:</td>
<td>blank &gt; solution</td>
</tr>
<tr>
<td>Resources:</td>
<td>slides</td>
</tr>
<tr>
<td>Fri 10/09</td>
<td><strong>LEC 02</strong> Case Analysis</td>
</tr>
<tr>
<td>Lesson:</td>
<td>videos &gt; pdf &gt; pptx</td>
</tr>
<tr>
<td>Class Session:</td>
<td>zoom &gt; handout &gt; solution</td>
</tr>
<tr>
<td>Week 3</td>
<td></td>
</tr>
<tr>
<td>Mon 10/12</td>
<td><strong>LEC 01</strong> Recurrences I, Master Theorem</td>
</tr>
<tr>
<td>Wed 10/14</td>
<td><strong>LEC 07</strong> Recurrences II, Tree Method</td>
</tr>
<tr>
<td>Thu 10/15</td>
<td><strong>LEC 05</strong> Recurrences, Master Theorem</td>
</tr>
<tr>
<td>Fri 10/16</td>
<td><strong>LEC 06</strong> Hash Maps</td>
</tr>
</tbody>
</table>

**Resources:** 373 prerequisite refresher
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!
Learning Objectives
After this lecture, you should be able to...

1. **Review** Distinguish between Asymptotic Analysis & Case Analysis, and apply both to code snippets

2. Describe the 3 most common recursive patterns and identify whether code belongs to one of them

3. Model recursive code using a recurrence relation (Step 1)

4. Use the Master Theorem to characterize a recurrence relation with Big-Oh/Big-Theta/Big-Omega bounds (Step 2)
Lecture Outline

• Review  Asymptotic Analysis & Case Analysis

• Analyzing Recursive Code
Review
Algorithmic Analysis Roadmap

1. Best Case Function
   $f(n) = 2$

2. Worst Case Function
   $f(n) = 3n + 1$

Asymptotic Analysis
- **Tight Big-O**: $O(1)$
- **Big-Theta**: $\Theta(1)$
- **Tight Big-Omega**: $\Omega(1)$

Case Analysis

```
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 \( c_1, c_2, n_0 \) such that for all \( n \geq n_0 \))
\[
   c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
\]
A Note on Asymptotic Analysis Tools

- We’ll generally use Big-Theta from here on out: most specific
- In industry, people often use Big-Oh to mean “Tight Big-Oh” and use it even when a Big-Theta exists

<table>
<thead>
<tr>
<th>When to use Big-Theta (most of the time):</th>
<th>When you have to use Big-Oh/Big-Omega:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for any function that's just the sum of its terms like f(n) = 2^n + 3n^3 + 4n - 5 we can always just do the approach of dropping constant multipliers / removing the lower order terms to find the big-Theta at a glance.</td>
<td>f(n) { n if n is prime, 1 otherwise} since in this case, the big-Oh (n) and the big-Omega (1) don't overlap at the same complexity class, there is no big-Theta and we couldn't use it here.</td>
</tr>
</tbody>
</table>
**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
Review  How to do Case Analysis

1. Are there significantly different cases?
   - Do other variables/parameters/fields affect the runtime, other than input size? For many algorithms, the answer is no.

2. Figure out how things could change depending on the input (excluding n, the input size)
   - Can you exit loops early?
   - Can you return early?
   - Are some branches much slower than others?

3. Determine what inputs could cause you to hit the best/worst parts of the code.
Other Useful Cases You Might See

• Overall Case:
  - Model code as a “cloud” that covers all possibilities across all cases. What’s the $O/\Omega/\Theta$ of that cloud?

• “Assume X Won’t Happen Case”:
  - E.g. Assume array won’t need to resize

• “Average Case”:
  - Assume random input
  - Lots of complications – what distribution of random?

• “In-Practice Case”:
  - Not a real term, but a useful idea
  - Make reasonable assumptions about how the world will work, then do worst-case analysis under those assumptions.
How Can You Tell if Best/Worst Cases Exist?

• Are there other possible models for this code?
• If \( n \) is given, are there still other factors that determine the runtime?

• Note: sometimes there aren’t significantly different cases! Sometimes we just want to model the code with a single function and go straight to asymptotic analysis!
In your own words, describe why we can or cannot use n=0 as the best case in our analysis.
Can We Choose n=0 as the Best Case?

• Remember that each case needs to be a “slice”: a function over n
  - The input to asymptotic analysis is a function over all of n, because we’re concerned with growth rate
  - Fixing n doesn’t work with our tools because it wouldn’t let us examine the bound asymptotically

• Think of it as “Best Case as n grows infinitely large”, not “Best Case of all inputs, including n”
Lecture Outline

• **Review**  Asymptotic Analysis & Case Analysis

• Analyzing Recursive Code

Recursive code usually falls into one of 3 common patterns:

1. **Halving the Input**  
   Binary Search

2. **Constant size Input**

3. **Doubling the Input**
public int binarySearch(int[] arr, int toFind, int lo, int hi) {
    if (hi < lo) {
        return -1;
    } else 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);
    }
}

Note: the parameters passed to recursive call reduce the size of the problem!
**Binary Search Runtime**

**Binary search**: An algorithm to find 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>

Let's consider the runtime of Binary Search

What’s the first step?
Binary Search Runtime

**Binary search**: An algorithm to find 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
```

What’s the Best Case?

**Element found at first index examined (index 8)**

\[ f(n) = 1 \quad \Rightarrow \Theta(1) \]

What’s the Worst Case?

**Element not found, cut input in half, then in half again...**

???

1 Halving the Input
**Binary Search Runtime**

- For an array of size $n$, eliminate $\frac{1}{2}$ until 1 element remains.
  - $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 4, 2, 1$
  - How many divisions does that 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**

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

Examples:

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

$3^2 = 9 \Rightarrow 2 = \log_3 9$
We Just Saw: A Leap of Intuition

- We identified the best and worst cases – a good start!
- But we didn’t do:
  - Step 1: model the code as a function
  - Step 2: analyze that function to find its bounds
Our Goal: A Complete Toolchain

We want to be able to use our algorithmic analysis tools
To do that, we need an essential intermediate: to model the code with runtime functions
Modeling Binary Search

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

How do we model a recursive call?

Fortunately, we have a tool for this!
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

A generic example of a recurrence:

\[ T(n) = \begin{cases} 
5 & \text{if } n < 3 \\
2T\left(\frac{n}{2}\right) + 10 & \text{otherwise}
\end{cases} \]
public int recurse(int n) {
    if (n < 3) {
        return 80;  // Base Case
    }

    int a = n * 2;  // Recursive Case
    int val1 = recurse(n / 3);
    int val2 = recurse(n / 3);
    return val1 + val2;  // Non-recursive Work:

    // Recursive Work:
    +2*T(n/3)

    T(n) = \begin{cases} 
          2 & \text{if } n < 3 \\
          2T\left(\frac{n}{3}\right) + 4 & \text{otherwise} 
    \end{cases}
Writing Recurrences: Example 2

```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);

    return val1 + val2;  // +2
}
```

Cost:

- Non-recursive Work: + n + 2
- Recursive Work: + 2*T(n/3)

**Recurrence:**

\[
T(n) = \begin{cases} 
2 & \text{if } n < 3 \\
2T\left(\frac{n}{3}\right) + n + 2 & \text{otherwise}
\end{cases}
\]
public int recurse(int n) {
    if (n < 3) {
        return 80; // Base Case
    }
    for (int i = 0; i < n; i++) {
        System.out.println(i); // Recursive Case
    }
    int val1 = recurse(n / 4); // Non-recursive Work: +n
    int val2 = recurse(n / 4); // Recursive Work: +3*T(n/4)
    int val3 = recurse(n / 4);
    return val1 + val2 * val3; // +3
}

\[
T(n) = \begin{cases} 
2 & \text{if } n < 3 \\
3T\left(\frac{n}{4}\right) + n + 3 & \text{otherwise}
\end{cases}
\]
Our Goal: A Complete Toolchain

1. Code
   - Case Analysis
   - Write a Recurrence

2. Asymptotic Analysis
   - Best Case Function
   - Worst Case Function

Asymptotic Analysis
- Big-O
- Big-Theta
- Big-Omega

Write a Recurrence
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-Θ 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 \( T(n) \in \Theta(n^{\log_b a}) \)

\[ a=2 \quad b=3 \quad c=1 \]

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

\[ \log_3 2 \approx 0.63 \]

\[ \log_3 2 < 1 \]

**We're in case 1**

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

**MASTER THEOREM**

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

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 \( \log_b a < c \) 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
Lecture Outline

• **Review**  Asymptotic Analysis & Case Analysis

• **Analyzing Recursive Code**

Recursive code usually falls into one of 3 common patterns:

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

2. **Constant size Input**
   - Merge Sort

3. **Doubling the Input**
Merge Sort

Divide

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
8 & 2 & 91 & 22 & 57 & 1 & 10 & 6 & 7 & 4 \\
\end{array}
\]

Conquer

\[
\begin{array}{c}
0 \\
8 \\
\end{array}
\]

Combine

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
2 & 8 & 22 & 57 & 91 & 1 & 4 & 6 & 7 & 10 \\
\end{array}
\]
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)
}
```

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

Constant size Input
What is the Big-Theta of worst-case Merge Sort?

\[ T(n) = \begin{cases} \frac{1}{2} \cdot \left( \frac{n}{2} \right) + n & \text{if } n \leq 3 \\ 2T\left( \frac{n}{2} \right) + n & \text{otherwise} \end{cases} \]

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

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 \( T(n) \in \Theta(n^{\log_b a}) \)
Merge Sort Recurrence to Big-Θ

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

MASTER THEOREM

\[ T(n) = \begin{cases} 
da & \text{if } n \text{ is at most some constant} \\
2T \left( \frac{n}{2} \right) + f(n) & \text{otherwise}
\end{cases} \]

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

\[ a=2 \quad b=2 \quad \text{and } c=1 \]
\[ \log_2 2 = 1 \]
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

- **Review**  Asymptotic Analysis & Case Analysis
- **Analyzing Recursive Code**

Recursive code usually falls into one of 3 common 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
Lecture Outline

• **Review**  Asymptotic Analysis & Case Analysis

• Analyzing Recursive Code

Recursive code usually falls into one of 3 common 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

NEXT LECTURE