BEFORE WE START

Review: Which of the following are evidence that a Big-Theta exists?

a) Big-Oh == Big-Theta
b) We’re analyzing a function that can be fully expressed as a polynomial
c) There aren’t extra terms (e.g. $n^2 + n$)
d) Runtime isn’t affected by array contents
Announcements

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

Starting today, many lectures will have OPTIONAL review questions. Worth extra credit (no points, doing a lot of them can bump you up 0.1). Not worthwhile for credit, but may be helpful for your review!

Section Review Videos are provided if you can’t make it to section, or want to review! Often focus on specific worksheet problems.

Project 1 (Deques) due Wednesday 7/8 11:59pm PDT

Exercise 1 (written, individual) due Friday 7/10 11:59pm PDT
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

Best Case Function

\[ f(n) = 2 \]

Worst Case Function

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

Asymptotic Analysis

CASE

Best Case

Worst Case

Other Case

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
\[ f(n) \text{ is } O(g(n)) \text{ if there exist positive constants } c, n_0 \text{ such that for all } n \geq n_0, 
   f(n) \leq c \cdot g(n) \]

Big-Omega
\[ f(n) \text{ is } \Omega(g(n)) \text{ if there exist positive constants } c, n_0 \text{ such that for all } n \geq n_0, 
   f(n) \geq c \cdot g(n) \]

Big-Theta
\[ f(n) \text{ is } \Theta(g(n)) \text{ if } 
   f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n)). 
\text{(in other words: there exist positive constants } c_1, c_2, n_0 \text{ such that for all } n \geq n_0) 
   c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \]
# 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>$f(n) { n \text{ if } n \text{ is prime, } 1 \text{ otherwise} }$</td>
</tr>
</tbody>
</table>

- since in this case, the big-Oh (n) and the big-Omega (1) don't overlap at the same complexity class, there is no reasonable big-Theta and we couldn't use it here.
**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!
Can We Choose n=0 as the Best Case?
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**
Case Study: Binary Search

```java
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);
    }
}
```

Base Cases

```java
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><strong>42</strong></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:

<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>

What’s the Best Case?

**Element found at first index examined (index 8)** \(\Theta(1)\)

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.
  - $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...$, 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 $n$?
    - $1, 2, 4, 8, ..., \frac{n}{4}, \frac{n}{2}, n$
  - 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

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
        } 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); // Recursive Case
    } else {
        return binarySearch(arr, toFind, lo, mid-1); // Recursive Case
    }
}

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:

\[ T(n) = \begin{cases} 
5 & \text{if } n < 3 \\
2T\left(\frac{n}{2}\right) + 10 & \text{otherwise} 
\end{cases} \]
Writing Recurrences: Example 1

```java
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
}
```

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

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

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

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

    return val1 + val2; // Non-recursive Work: + n + 2 Recursive Work: + 2*\text{T}(n/3)
}
```

The recurrence relation for this algorithm is:

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

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

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

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

    return val1 + val2 + val3;  // Non-recursive Work: + n + 3
}
Our Goal: A Complete Toolchain

1. Case Analysis
   - Write a Recurrence

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

- TIGHT BIG-OMEGA
- BIG-THETA
- TIGHT BIG-OMEGA
- TIGHT BIG-OMEGA
- TIGHT BIG-OMEGA
- TIGHT BIG-OMEGA
- TIGHT BIG-OMEGA
- TIGHT BIG-OMEGA
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 \( c=1 \)

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

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

\[ \log_3 2 < 1 \]

**We’re in case 1**

\[ T(n) \in \Theta(n) \]
Aside 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**

<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>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>91</td>
<td>22</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<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>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>10</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>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)
}

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 3 \\
2T\left( \frac{n}{2} \right) + n & \text{otherwise} 
\end{cases} \]
Take a guess: What is the Big-Theta of worst-case Merge Sort?

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

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

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 \( c=1 \)

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

\[ \log_2 2 = x \Rightarrow 2^x = 2 \Rightarrow x = 1 \]

\[ \log_2 2 = 1 \]

**We’re in case 2**

\[ T(n) \in \Theta(n \log n) \]
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
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

Doubling the Input
Fibonacci Recurrence to Big-$\Theta$

Can we use the Master Theorem?

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

Can we intuit a pattern?

T(1) = $d$

T(2) = $2T(1) + c = 2d + c$

T(3) = $2T(2) + c = 4d + 3c$

T(4) = $2T(3) + c = 8d + 7c$

T(5) = $2T(4) + c = 16d + 25c$

Looks like something’s happening but it’s tough

Maybe geometry can help!
Fibonacci Recurrence to Big-Θ

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

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

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?

\[1 + 2 + 3 + 4 + ... + 2^{k-1}\]
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)

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

  \[ T(n) = 2^n - 1 \in \Theta(2^n) \]
# Fibonacci Recurrence to Big-Θ

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many layers in the function call tree?</td>
<td>n</td>
</tr>
<tr>
<td>How many function calls on layer k?</td>
<td>$2^{k-1}$</td>
</tr>
<tr>
<td>How many function calls TOTAL for a tree of k layers?</td>
<td>$1 + 2 + 4 + 8 + ... + 2^{k-1}$</td>
</tr>
<tr>
<td>Total runtime = (total function calls) * (runtime of each function call)</td>
<td>$(1 + 2 + 4 + 8 + ... + 2^{k-1}) \times \text{(constant work)}$</td>
</tr>
<tr>
<td>$1 + 2 + 4 + 8 + ... + 2^{k-1} = \sum_{i=1}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$</td>
<td>Summation Identity: Finite Geometric Series</td>
</tr>
<tr>
<td>$T(n) = 2^n - 1 \in \Theta(2^n)$</td>
<td></td>
</tr>
</tbody>
</table>
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
   - $\Theta (2^n)$