File Edit View Help

File Edit View Help

Searching, Sorting, Complexity

File Edit View Help

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

File Edit View Help

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

File Edit View Help

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

Searching Algorithms

- Problem:
  - Given a sorted array of integers, develop an algorithm that returns true or false depending on if a target value was found in the array.
- Linear Search
  1. Start from index 0
  2. If the value at that index matches the target value then return true
  3. Otherwise move to the next index
  4. If the next index is outside of the array then return false
  5. Repeat Step 2

Example

TRUE!

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

File Edit View Help

Searching Algorithms

- Binary Search
  1. Assume the Start Index is 0 and the End Index is Array.length - 1
  2. Calculate the Middle Index from the Start and End Indices (middle = (start+end)/2)
  3. If the Middle Index is outside of the array then return false
  4. If the value at that index matches the target value then return true
  5. If the value at the index is greater than the target value, then repeat Step 2 with the same Start Index to the Middle Index - 1
  6. If the value at the index is less than the target value, then repeat Step 2 starting with the Middle Index + 1 and the same End Index

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Searching Algorithms

- Binary Search
  1. Assume the Start Index is 0 and the End Index is Array.length - 1
  2. Calculate the Middle Index from the Start and End Indices (middle = (start+end)/2)
  3. If the Middle Index is outside of the array then return false
  4. If the value at that index matches the target value then return true
  5. If the value at the index is greater than the target value, then repeat Step 2 with the same Start Index to the Middle Index - 1
  6. If the value at the index is less than the target value, then repeat Step 2 starting with the Middle Index + 1 and the same End Index

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

Searching Algorithms

- Binary Search
  1. Assume the Start Index is 0 and the End Index is Array.length - 1
  2. Calculate the Middle Index from the Start and End Indices (middle = (start+end)/2)
  3. If the Middle Index is outside of the array then return false
  4. If the value at that index matches the target value then return true
  5. If the value at the index is greater than the target value, then repeat Step 2 with the same Start Index to the Middle Index - 1
  6. If the value at the index is less than the target value, then repeat Step 2 starting with the value, then repeat Step 2 starting with the Middle Index + 1 and the same End Index

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

Searching Algorithms

- Binary Search
  1. Assume the Start Index is 0 and the End Index is Array.length - 1
  2. Calculate the Middle Index from the Start and End Indices (middle = (start+end)/2)
  3. If the Middle Index is outside of the array then return false
  4. If the value at that index matches the target value then return true
  5. If the value at the index is greater than the target value, then repeat Step 2 with the same Start Index to the Middle Index - 1
  6. If the value at the index is less than the target value, then repeat Step 2 starting with the Middle Index + 1 and the same End Index

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

File Edit View Help

Searching Algorithms

- Binary Search
  1. Assume the Start Index is 0 and the End Index is Array.length - 1
  2. Calculate the Middle Index from the Start and End Indices (middle = (start+end)/2)
  3. If the Middle Index is outside of the array then return false
  4. If the value at that index matches the target value then return true
  5. If the value at the index is greater than the target value, then repeat Step 2 with the same Start Index to the Middle Index - 1
  6. If the value at the index is less than the target value, then repeat Step 2 starting with the Middle Index + 1 and the same End Index

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

File Edit View Help

Searching Algorithms

- Binary Search
  1. Assume the Start Index is 0 and the End Index is Array.length - 1
  2. Calculate the Middle Index from the Start and End Indices (middle = (start+end)/2)
  3. If the Middle Index is outside of the array then return false
  4. If the value at that index matches the target value then return true
  5. If the value at the index is greater than the target value, then repeat Step 2 with the same Start Index to the Middle Index - 1
  6. If the value at the index is less than the target value, then repeat Step 2 starting with the Middle Index + 1 and the same End Index

Example

TRUE!

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 11

File Edit View Help

Which is more Efficient?

Efficiency

- Efficiency
  - Producing desired results with little to no waste
  - Well organized and prevents wasteful use of a resource
- Resources
  - Time
  - Space
- How do we measure efficiency?
  - Algorithms do not require computers

File Edit View Help

Complexity

- Complexity
  - Classifies Computational Problems based on inherent difficulty
  - Relates problems to each other
  - Time and Space
- Asymptotic Analysis
  - A way to describe a *limiting* behavior / function
  - Limits in math are a value that a function *approaches* as the input *approaches* some value
  - Time and Space Complexity

File Edit View Help

Big O Notation

- Theoretical upper bound of an algorithm
- The “Worst Case” scenario
- Let f and g be functions defined on some subset of real numbers

$$f(n) = O(g(n))\ where\ n \in \mathbb{R}\ \text{as}\ n \to \infty$$

- Let M be a constant that’s sufficiently large then we can say

$$|f(n)| \leq M|g(n)|\ \ for\ all\ n \geq n_0$$

File Edit View Help

?

Complexity

- Theoretical Model of a Computer
  - Input is passed to the Computer
  - Computer Computes
  - Computer Outputs the result
- Time Complexity
  - For a given amount of data (n) how many operations will the algorithm take to complete?
- Space Complexity
  - For a given amount of data (n) how much space will the algorithm require to complete?

Theoretical Model of a Computer

Complexity

- Big O Time Complexity
  - For a given amount of data (n) **AT MOST** how many operations will the algorithm take to complete?
- Big O Space Complexity
  - For a given amount of data (n) **AT MOST** how much space will the algorithm require to complete?
- Assuming “AT MOST” means that we are viewing the “Worst Case” scenario
  - The case that would cause THE MOST operations to complete for time complexity
  - The case that would require THE MOST space to complete for space complexity

Theoretical Model of a Computer

File Edit View Help

Big O Notation

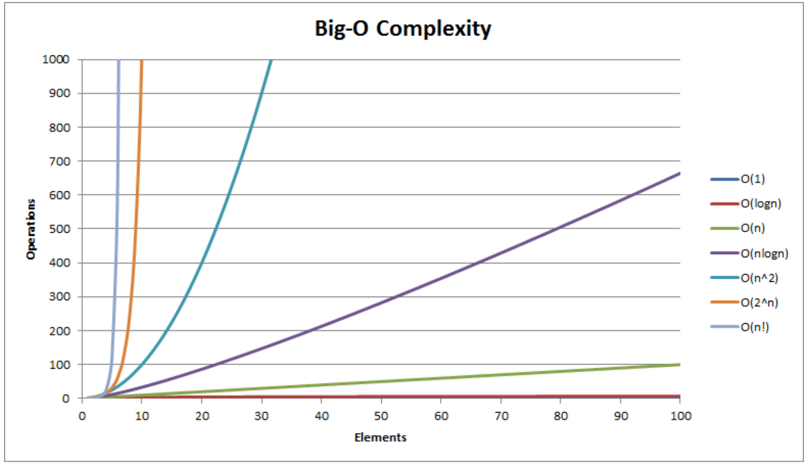
• We assign Big O function as a means to describe the time or space complexity given an input (n) as it approaches infinity • Big O is an inequality – Does not have to be exactly equal – As long as the function being assigned (g(n)) is always larger than given function (f(n)), given a big enough constant (M), then we consider it to be Big O of the given function (f(n) = O(g(n))	<u>Big O</u> $f(n) = O(g(n))$ $where\ n \in \mathbb{R}$ as $n \to \infty$ such that, $\lvert f(n)\rvert \leq M\lvert g(n)\rvert$ $for\ all\ n \geq n_0$

Big O Notation

- Plotting these functions on a graph
  - Amount of Data (n)
  - Number of Operations (for time) or amount of space (for space)
- If we plot these functions on a graph, then a function (g(n)) is Big O of another function (f(n)) if the second function is below or equal to the first function's curve
- Examples

$(f(n) = n) = O(n)$ $(f(n) = n) = O(n^2)$ $(f(n) = n^2+n+3) = O(n^3)$ $(f(n) = n^2) = O(n!)$	$(f(n) = n^2) \neq O(n)$ $(f(n) = n!) \neq O(n^2)$ $(f(n) = n) \neq O(1)$ $(f(n) = 2^n) \neq O(n)$

Big O

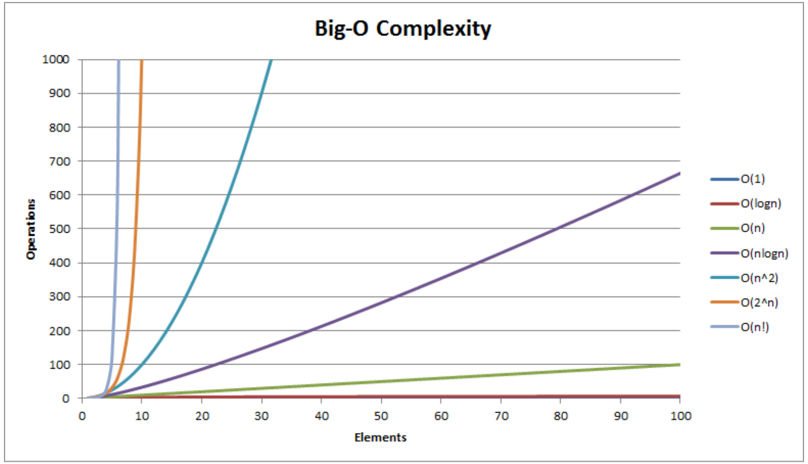


File Edit View Help

Big O Notation

- Common Big O Complexities
  - O(1) – Constant
  - O(log(n)) – Logarithmic
  - O(n) – Linear
  - O(nlogn) – Linearithmic
  - $O(n^2)$ – Quadratic
  - $O(2^n)$ – Exponential “Bad”
  - O(n!) – Factorial “Really Bad”

Big O



File Edit View Help

Applying this to the Examples

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Not Found

Target Value: 13

File Edit View Help

Searching Algorithms

- Linear Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)
  - 8 checks to determine it was not in the array
  - Assuming the array was size n then,

Linear Search
O(n)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Not Found

Target Value: 13

File Edit View Help

Searching Algorithms

- Binary Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

File Edit View Help

Searching Algorithms

- Binary Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

Searching Algorithms

- Binary Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

Searching Algorithms

- Binary Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Binary Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Target Value: 13

File Edit View Help

Searching Algorithms

- Binary Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)
  - 3 checks to determine it was not in the array
  - Assuming the array was size n then,

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Not Found

Target Value: 13

File Edit View Help

Searching Algorithms

- Binary Search
  - How many operations will this take in the worst case?
  - Array is size 8 (n=8)
  - 3 checks to determine it was not in the array
  - Assuming the array was size n then,

Binary Search

$O(\lg(n))$

$O(\log_2(n))$

Example

Index	0	1	2	3	4	5	6	7
Value	5	6	7	8	9	10	11	12

Not Found

Target Value: 13

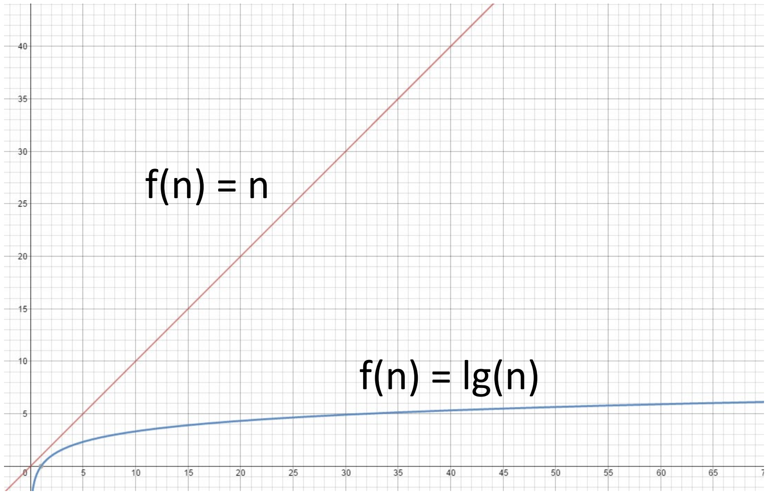
File Edit View Help

Which one is better?

Searching Algorithms

- Binary Search has a better time complexity than Linear Search
- Given the same amounts of data (n) Binary Search requires less steps to complete

<u>Example</u>



File Edit View Help

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method’s Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn’t have to be exact but close
  - An approximation that can be proven later

Example

```
public static boolean linearSearchIterative(int[] a, int value)
{
    for(int i=0;i<a.length;i++)
    {
        if(a[i] == value)
        {
            return true;
        }
    }
    return false;
}
```

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

O(1)

```
public static boolean linearSearchIterative(int[] a, int value)
{
    for(int i=0;i<a.length;i++)
    {
        if(a[i] == value)
        {
            return true;
        }
    }
    return false;
}
```

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

```
public static boolean linearSearchIterative(int[] a, int value)
{
    for(int i=0;i<a.length;i++)
    {
        if(a[i] == value)
        {
            return true;
        }
    }
    return false;
}
```

O(?)

File Edit View Help

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

<u>Example</u>

```
public static boolean linearSearchIterative(int[] a, int value)
{
    for(int i=0;i<a.length;i++)
    {
        if(a[i] == value)
        {
            return true;
        }
    }
    return false;
}
```

O(n)

File Edit View Help

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

```
public static boolean linearSearchIterative(int[] a, int value)
{
    for(int i=0;i<a.length;i++)
    {
        if(a[i] == value)
        {
            return true;
        }
    }
    return false;
}
```

O(n)

O(1)

Complexity =
O(n) + O(1)
or just
O(n)

File Edit View Help

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

```
public static boolean binarySearchRecursive(int[] a, int value, int minIndex, int maxIndex)
{
    int midIndex = (maxIndex + minIndex)/2;
    if(minIndex > maxIndex)
    {
        return false;
    }
    if(a[midIndex] == value)
    {
        return true;
    }

    else
    {
        if(value > a[midIndex])
        {
            return binarySearchRecursive(a,value,midIndex+1,maxIndex);
        }
        else
        {
            return binarySearchRecursive(a,value,minIndex,midIndex-1);
        }
    }
}
```

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

O(1)

```java
public static boolean binarySearchRecursive(int[] a, int value, int minIndex, int maxIndex)
{
    int midIndex = (maxIndex + minIndex)/2;
    if(minIndex > maxIndex)
    {
        return false;
    }
    if(a[midIndex] == value)
    {
        return true;
    }
    else
    {
        if(value > a[midIndex])
        {
            return binarySearchRecursive(a,value,midIndex+1,maxIndex);
        }
        else
        {
            return binarySearchRecursive(a,value,minIndex,midIndex-1);
        }
    }
}
```

File Edit View Help

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

```
public static boolean binarySearchRecursive(int[] a, int value, int minIndex, int maxIndex)
{
    int midIndex = (maxIndex + minIndex)/2;
    if(minIndex > maxIndex)
    {
        return false;
    }
    if(a[midIndex] == value)
    {
        return true;
    }

    else
    {
        if(value > a[midIndex])
        {
            return binarySearchRecursive(a,value,midIndex+1,maxIndex);
        }
        else
        {
            return binarySearchRecursive(a,value,minIndex,midIndex-1);
        }
    }
}
```

O(?)

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

```
public static boolean binarySearchRecursive(int[] a, int value, int minIndex, int maxIndex)
{
    int midIndex = (maxIndex + minIndex)/2;
    if(minIndex > maxIndex)
    {
        return false;
    }
    if(a[midIndex] == value)
    {
        return true;
    }

    else
    {
        if(value > a[midIndex])
        {
            return binarySearchRecursive(a,value,midIndex+1,maxIndex);
        }
        else
        {
            return binarySearchRecursive(a,value,minIndex,midIndex-1);
        }
    }
}
```

O(lg(n))

File Edit View Help

Applying this to Code

- Any simple statement is going to be O(1)
- Loops and recursion is where complexity increases
  - The number of times the loop runs or the number of times the recursive call is made
  - Observe how the inputted data (n) is being processed
- The largest complexity can be considered the method's Big O
  - Largest complexity is the function that approaches infinity faster
  - Doesn't have to be exact but close
  - An approximation that can be proven later

Example

```
public static boolean binarySearchRecursive(int[] a, int value, int minIndex, int maxIndex)
{
    int midIndex = (maxIndex + minIndex)/2;
    if(minIndex > maxIndex)
    {
        return false;
    }
    if(a[midIndex] == value)
    {
        return true;
    }

    else
    {
        if(value > a[midIndex])
        {
            return binarySearchRecursive(a,value,midIndex+1,maxIndex);
        }
        else
        {
            return binarySearchRecursive(a,value,minIndex,midIndex-1);
        }
    }
}
```

O(lg(n))

Complexity = O(1)'s + O(lg(n))
or just
O(lg(n))