Searching, Sorting, Complexity
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
• 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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
• 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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
• 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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11

TRUE!
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
• 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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
• 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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11
• 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

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 11

TRUE!
Which is more Efficient?
• 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
• 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
• Theoretical upper bound of an algorithm
• The “Worst Case” scenario
• Let $f$ and $g$ be functions defined on some subset of real numbers

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

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

$$|f(n)| \leq M|g(n)| \text{ for all } n \geq n_0$$
• 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?
• **Big O Time Complexity**
  – For a given amount of data \( n \) \textbf{AT MOST} how many operations will the algorithm take to complete?

• **Big O Space Complexity**
  – For a given amount of data \( n \) \textbf{AT MOST} how much space will the algorithm require to complete?

• Assuming “\textbf{AT MOST}” means that we are viewing the “\textbf{Worst Case}” scenario
  – The case that would cause \textbf{THE MOST} operations to complete for time complexity
  – The case that would require \textbf{THE MOST} space to complete for space complexity
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 \((f(n) = O(g(n))\)

<table>
<thead>
<tr>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f(n) = O(g(n))) where (n \in \mathbb{R}) as (n \to \infty) such that, (</td>
</tr>
</tbody>
</table>
• 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

| \( f(n) = n \) = \( O(n) \) | \( f(n) = n^2 \) ≠ \( O(n) \) |
| \( f(n) = n \) = \( O(n^2) \) | \( f(n) = n! \) ≠ \( O(n^2) \) |
| \( f(n) = n^2 + n + 3 \) = \( O(n^3) \) | \( f(n) = n \) ≠ \( O(1) \) |
| \( f(n) = n^2 \) = \( O(n!) \) | \( f(n) = 2^n \) ≠ \( O(n) \) |
• 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”
Applying this to the Examples
• Linear Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Linear Search

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

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
• Linear Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
• Linear Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
Linear Search
– How many operations will this take in the worst case?
– Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
• Linear Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)
• Linear Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
• Linear Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
Linear Search

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

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
• Linear Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13

Not Found
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,

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
• Binary Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Binary Search
- How many operations will this take in the worst case?
- Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
Binary Search

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

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
Binary Search
- How many operations will this take in the worst case?
- Array is size 8 (n=8)

Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Target Value: 13
• Binary Search
  – How many operations will this take in the worst case?
  – Array is size 8 (n=8)

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Example

Target Value: 13
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
  – 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,

\[
\text{Binary Search} \quad O(\log(n)) \quad O(\log_2(n))
\]
Which one is better?
• Binary Search has a better time complexity than Linear Search
• Given the same amounts of data (n) Binary Search requires less steps to complete

Example

\[ f(n) = n \]
\[ f(n) = \lg(n) \]
• 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

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

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

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

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

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

Complexity = $O(n) + O(1)$
or just $O(n)$
• 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**

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

```java
public static boolean binarySearchRecursive(int[] a, int value, int minIndex, int maxIndex)
{
    int midIndex = (minIndex + maxIndex)/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);
    }
}
```
• 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

```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);
        }
    }
}
```
• 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

```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);
        }
    }
}
```
• 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);
    }
}
```

Complexity = $O(1) + O(lg(n))$

or just

$O(lg(n))$