Searching
Overview

We often need to search for an item in a collection

- *Is this student in this recitation roster?*
- *Is this username in our user database?*
- *Is there any data point in our dataset that matches this description?*

In this module, we will learn about how to search for an element in an array.
Learning Objectives

- To be able to use linear search to find an element inside an array
- To be able to use binary search to find an element inside an array
- To be able to know when to use linear search and when to use binary search
Problem: Search

Formally, given a sequence of values and a target value, we want to determine if the target value is in the sequence, and if so, where it is located.

- in our case, the "sequence of values" is an array
- the "target value" is the value we are searching for
- the location is the index of the value in the array, or $-1$ if it's not present.
**Concept: The "Feasible Region"**

In any problem, the **feasible region** is the name for the set of possible values that might be a solution.

- In the context of search, the feasible region refers to the set of indices in the array that might contain the target value.
- A set of indices is functionally a region of the array where the target value might be found.

In array search, we repeatedly reduce the feasible region until we find the target value, or until we determine that the target value is not present in the array.
Linear Search

Used to search for a value (the target) in an **unsorted array**

- Use a loop to iterate over the values
- Start at the first element and move to the next element until the target is found
- Returns the position of the target if it was found in the array, or -1 if the target was not found in the array

With each iteration, we reduce the feasible region by one element.
Linear Search Example

Searching for 82 at position 0

(this image is a link)
Linear Search

public static int linearSearch(int[] A, int k) {
    // for each index in the array...
    for (int i = 0; i < A.length; i++) {
        // compare the current value (A[i]) to the target k
        if (A[i] == k) {
            return i; // if there's a match, stop & return THE INDEX
        }
    }
    // we only get to this line if we didn't find anything!
    return -1;
}
Linear Search: Thinking Critically

How many iterations of the for loop will we need if...

- the target is the first element in the array?
- the target is the 10th element in the array?
- the target is not in the array?
Linear Search: Thinking Critically

How many iterations of the for loop will we need if...

- the target is the first element in the array? 1
- the target is the 10th element in the array? 10
- the target is not in the array? A.length
**Binary Search**

Used to search for a target value in a **sorted array only**

- Compares the target with the value at the middle index (middle element)
  - If the middle element is the target element, then we're done!
  - If the target is less than the middle element, then we search for the target in the **left half of the array** (the positions before the middle element)
  - If the target is greater than the middle element, then we search the target in the **right half of the array** (the positions after the middle element)

- Repeat on the remaining search area of the array until
  - the element is found
  - there is no feasible search area left
**Binary Search**

Returns the position of the middle element if it is equal to the target

Returns -1 if the target was not found in the array
Binary Search

Searching for "Dustin" in the array names!

<table>
<thead>
<tr>
<th>Caryn</th>
<th>Debbie</th>
<th>Dustin</th>
<th>Elliot</th>
<th>Jacquie</th>
<th>Jon</th>
<th>Rich</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

middle = (low + high) / 2 = 3
names[middle] is "Elliot", which comes after "Dustin" alphabetically.
So, if "Dustin" is present, it must be between positions 0 and middle − 1.
Binary Search

Searching for "Dustin" in the array names!

<table>
<thead>
<tr>
<th>Caryn</th>
<th>Debbie</th>
<th>Dustin</th>
<th>Elliot</th>
<th>Jacque</th>
<th>Jon</th>
<th>Rich</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- middle = (low + high) / 2 = 1
- \text{names}[middle] \text{ is } "Debbie", which comes before "Dustin" alphabetically.
- So, if "Dustin" is present, it must be between positions middle + 1 and 2.
Binary Search

Searching for "Dustin" in the array names!

<table>
<thead>
<tr>
<th>Caryn</th>
<th>Debbie</th>
<th>Dustin</th>
<th>Elliot</th>
<th>Jacquie</th>
<th>Jon</th>
<th>Rich</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- middle = (low + high) / 2 = 2
- names[middle] is "Dustin", which is the target element! So, we return middle.
Binary Search: Searching for an Element not Present

Searching for "Drew" in the array names!

\[
\begin{array}{cccccccc}
\text{Caryn} & \text{Debbie} & \text{Dustin} & \text{Elliot} & \text{Jacquie} & \text{Jon} & \text{Rich} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\text{low} & \text{middle} & \text{high}
\end{array}
\]

- \( \text{middle} = (\text{low} + \text{high}) / 2 = 3 \)
- \( \text{names[middle]} \) is "Elliot", which comes after "Drew" alphabetically.
- So, if "Drew" is present, it must be between positions 0 and middle − 1.
Binary Search: Searching for an Element not Present

Searching for "Drew" in the array names!

<table>
<thead>
<tr>
<th>Caryn</th>
<th>Debbie</th>
<th>Dustin</th>
<th>Elliot</th>
<th>Jacque</th>
<th>Jon</th>
<th>Rich</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- middle = (low + high) / 2 = 1
- names[middle] is "Debbie", which comes before "Drew" alphabetically.
- So, if "Drew" is present, it must be between positions middle + 1 and 2.
Binary Search: Searching for an Element not Present

Searching for "Drew" in the array names!

<table>
<thead>
<tr>
<th>Caryn</th>
<th>Debbie</th>
<th>Dustin</th>
<th>Elliot</th>
<th>Jacque</th>
<th>Jon</th>
<th>Rich</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

low, middle, high

- middle = (low + high) / 2 = 2
- names[middle] is "Dustin", which comes after "Drew" alphabetically.
- So, if "Drew" is present, it must be between positions 2 and middle - 1.
Binary Search: Searching for an Element not Present

Searching for "Drew" in the array names!

<table>
<thead>
<tr>
<th>Caryn</th>
<th>Debbie</th>
<th>Dustin</th>
<th>Elliot</th>
<th>Jacquie</th>
<th>Jon</th>
<th>Rich</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- high is now less than low. The "feasible search area" is now totally empty.
- So, we return -1 to indicate that the target was not found in the array.
Binary Search, Interactive

(0 5 10 11 15 16 23 26 28 43 50 50 52 53 66 69 71 77 81 82 95)

Searching for 26 at position 10
Left bound at position 0, right bound at position 20

(this image is a link)
public static int binarySearch(int[] array, int target) {
    // to start, the search area is the entire array, [0, array.length - 1]
    int lowIndex = 0;
    int highIndex = array.length - 1;

    // there's a valid search area as long as lowIndex <= highIndex
    while (lowIndex <= highIndex) {
        int middleIndex = (lowIndex + highIndex) / 2;
        if (target < array[middleIndex]) {
            highIndex = middleIndex - 1; // throw away *right* half
        } else if (target > array[middleIndex]) {
            lowIndex = middleIndex + 1; // throw away *left* half
        } else { // happens when target == array[middleIndex]
            return middleIndex; // success!
        }
    }

    return -1; // couldn't find element, so return default -1
}
Linear Search vs. Binary Search

- Binary search is faster than linear search 😊ğini
  - Per iteration, binary search shrinks the feasible region by half the remaining elements, linear search only by one element.
  - On average, binary search requires fewer iterations of the search loop
  - (when is binary search not faster then linear search?)

- Binary search runs on sorted data 😞😞❗
  - why?

- Linear search runs on unsorted data 😊😊😊
Linear Search vs. Binary Search

Runtime analysis: how many comparisons will it take to determine that the target is not in the array?

<table>
<thead>
<tr>
<th>Length of the array</th>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>