Learning Objectives

Learners will be able to...

- **Define Search Algorithm**
- **Understand the algorithm behind Linear Search, Binary Search, and Jump Search**
- **Implement Common search algorithm**
- **Analyze the complexities of Common Search Algorithms**

Searching

Linear Search, Binary Search, and Jump Search

In the field of computer science, searching algorithms play a pivotal role in locating specified items among a collection of items. Efficient and effective search algorithms are at the core of many operations — be it in databases, data retrieval, or simply within software applications. We mentioned some of these search algorithm before, but here we are giving them individualized attention. This assignment aims to delve into three common searching algorithms: **linear search**, **binary search**, and **jump search**.

Each of these algorithms has its unique characteristics, advantages, and disadvantages. Throughout this assignment, we will explore how each algorithm works, analyze their time and space complexities, and understand their best and worst-case scenarios.

Goals of the Assignment

1. To understand the fundamental principles behind each searching algorithm.
2. To implement linear search, binary search, and jump search in Java.
3. To analyze the time and space complexities for each algorithm.
4. To compare and contrast the efficiency and practicality of each algorithm in various scenarios.
5. To draw conclusions regarding which algorithm is best suited for particular types of problems.

By the end of this assignment, you will have a comprehensive understanding of how each of these algorithms functions and where they are best applied. This will equip you with the skills necessary to make informed decisions when faced with problems that require searching operations.

Linear Search

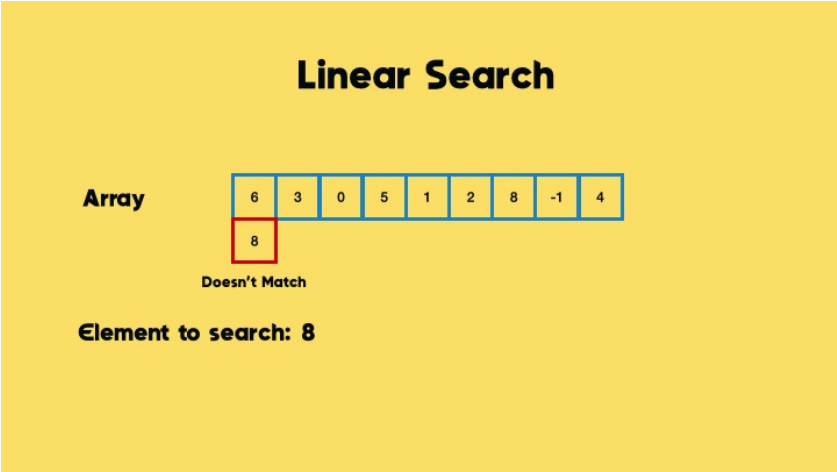
Linear Search: Overview, Steps, and Implementation

Linear search is the simplest form of searching algorithm. It is often the first searching algorithm taught in computer science courses due to its straightforward logic. The algorithm scans through each element of the collection sequentially, stopping when it finds the element that matches the target. While not the most efficient for large datasets, it's simple to understand and implement.

info

Searching Mechanics

The following gif demonstrates how a linear search works:



Steps

1. **Start at the Beginning**: Begin from the first element of the array.
2. **Sequential Scan**: Move from one element to the next in sequence, comparing each with the target value.
3. **Check for Match**: If an element matches the target, return its index.
4. **End of Array**: If the end of the array is reached without finding the target, return -1 to indicate that the item was not found.

Implementation

We have seen linear search several times now, so the implementation of its algorithm should not come as a surprise. Create the `linearSearch` method. It takes an array of integers and another integer representing the search target. The method should return an integer.

```
    public static int linearSearch(int[] arr, int target) {

    }
```

Starting at the first element, iterate over the array. Check each element in the array against the search target. If they match, return the index of the matching element. If the loop finishes without a match, that means the search target is not in the array. Return `-1`.

```
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // Target found, return index
            }
        }
        return -1; // Target not found
    }
```

The code in the IDE is sending the information below to the `linearSearch` method. Your code should return `6`.

```
int[] arr = {3, 9, 6, 4, 7, 1, 5, 8, 2};
int target = 5;
```

Linear search is an excellent algorithm for smaller datasets or for unsorted arrays where more advanced algorithms cannot be applied. It is a brute-force method, checking each element one-by-one, which makes it less suitable for larger, sorted datasets where other methods like Binary Search or Jump Search may offer better performance.

Linear Search Analysis

Time Complexity

- **Worst-Case**: $O(n)$ - In the worst-case scenario, the target element could be at the end of the array, necessitating a scan through every element.
- **Average-Case**: $O(n)$ - On average, we expect to search through half of the elements, but the time complexity still scales linearly with the size of the data set.
- **Best-Case**: $O(1)$ - In the best-case scenario, the target element is the first one in the array, and the search concludes immediately.

Space Complexity

The space complexity of linear search is $O(1)$, meaning it requires a constant amount of additional memory to perform the search. This is because the algorithm only needs to store a single element (the target) for comparison, regardless of the size of the array.

Practical Use-Cases

Linear search is practical when dealing with small arrays or lists, where its simplicity outweighs the speed benefits of more complex algorithms. Unlike algorithms such as binary search, linear search can be applied to unsorted datasets. In cases where the data is streaming or not all available at once, linear search can be useful as it processes elements sequentially.

Comparison with Other Algorithms

1. **Ease of Implementation**: Linear search is easier to implement compared to other algorithms like binary or jump search.
2. **Universality**: It works on sorted and unsorted data, unlike binary and jump search which require sorted arrays.
3. **Efficiency**: It is less efficient than other search algorithms like binary and jump search for larger datasets.

Linear search, though straightforward, is often inefficient for larger datasets, especially when compared to more advanced algorithms like binary search or jump search. However, its simplicity and universality make it a suitable choice for small or unsorted datasets. It also has the advantage of not requiring any additional memory, making it a “greedy” algorithm in terms of space.

Understanding the strengths and weaknesses of linear search is crucial for selecting the appropriate searching algorithm based on the problem requirements. In subsequent sections, we will delve deeper into more advanced searching algorithms to compare their efficiencies and practical applications.

Binary Search

Binary Search: Overview, Steps, and Implementation

Binary search is a highly efficient searching algorithm that works by repeatedly dividing the sorted array into halves until the target element is found. By taking advantage of the sorted nature of the array, binary search minimizes the number of comparisons needed, offering a substantial performance gain over algorithms like linear search.

info

Searching Mechanics

The following gif demonstrates how a binary search works:

Steps

1. **Initialize Pointers**: Set the low pointer to the first index and the high pointer to the last index of the array.
2. **Calculate Mid**: Calculate the middle index as `({low} + {high}) / 2)`.
3. **Check for Match**: Compare the middle element with the target. If it matches, return the middle index.
4. **Adjust Pointers**:
   - If the target is less than the middle element, set high to `({mid} - 1 )`.
   - If the target is greater than the middle element, set low to `({mid} + 1 )`.

5. **Iterate**: Repeat steps 2-4 until the low pointer is less than or equal to the high pointer. If not found, return -1.

Implementation

Binary search can be implemented in a variety of ways. We are going to use an iterative approach. Start by creating the `binarySearch` method. It should take an array of integers and an integer representing the search target. The method should return an integer. Create the variables `low` and `high`, and set them to the first and last elements respectively.

```
    public static int binarySearch(int[] arr, int target) {
        int low = 0, high = arr.length - 1;

    }
```

As long as `low` is less than or equal to `high`, find the midpoint between the two pointers. If the element at the midpoint is equal to the target, return the midpoint. If the element at the midpoint is less than the target, set `low` to the midpoint plus `1`. Otherwise set `high` to the midpoint minus `1`. If the loop completes without returning a value, then the search target is not in the array. Return `-1`.

```
    public static int binarySearch(int[] arr, int target) {
        int low = 0, high = arr.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == target) {
                return mid; // Target found, return index
            } else if (arr[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1; // Target not found
    }
```

The code in the IDE is sending the information below to the `binarySearch` method. Your code should return `5`.

```
int[] arr = {1, 3, 5, 7, 9, 11, 13, 15};
int target = 11;
```

Binary search offers a significant performance improvement over Linear Search for sorted arrays, particularly as the size of the dataset increases. However, it requires the array to be sorted beforehand, which could be a

limitation depending on the application.

Binary Search Analysis

Time Complexity

- **Worst-Case**: $O(logn)$ - Even in the worst-case scenario, binary search minimizes the number of comparisons through successive divisions of the array.
- **Average-Case**: $O(logn)$ - The average case also benefits from the logarithmic nature of the algorithm.
- **Best-Case**: $O(1)$ - In the best-case scenario, the target element is right at the center of the array, allowing the algorithm to find it in just one comparison.

Space Complexity

The space complexity for Binary Search is $O(1)$, meaning it requires a constant amount of additional memory for its operation. The algorithm only requires variables to store the high, low, and mid indices, which does not scale with the size of the input array.

▼ **Recursive Space Complexity**
You can also have a recursive solution to binary search. In this case, the space complexity would not be constant. Instead it would be $O(log n)$.

Practical Use-Cases

Binary search is ideal for search problems involving large datasets, provided that the data is sorted. It is frequently used in applications that require fast data retrieval times, such as databases. Binary search also provides version control, meaning you can quickly find specific versions in a sorted version history.

Comparison with Other Algorithms

- **Efficiency**: Binary search is significantly more efficient than linear search for large datasets, but less efficient than specialized algorithms like Fibonacci search or interpolation search for uniformly distributed datasets.
- **Preconditions**: Unlike linear search, binary search requires the input array to be sorted.
- **Simplicity**: It is slightly more complicated to understand and implement than linear search, but simpler than many other advanced searching algorithms.

Binary Search offers an excellent balance of efficiency and simplicity. Its logarithmic time complexity makes it a preferred choice for large, sorted datasets. However, the prerequisite of a sorted array can sometimes be a limitation.

In terms of efficiency, Binary Search performs exceptionally well, but it is crucial to remember its limitation that it can only operate on sorted arrays. Its performance and operational characteristics make it a popular choice in many applications that require efficient search capabilities.

Jump Search

Jump Search: Overview, Steps, and Implementation

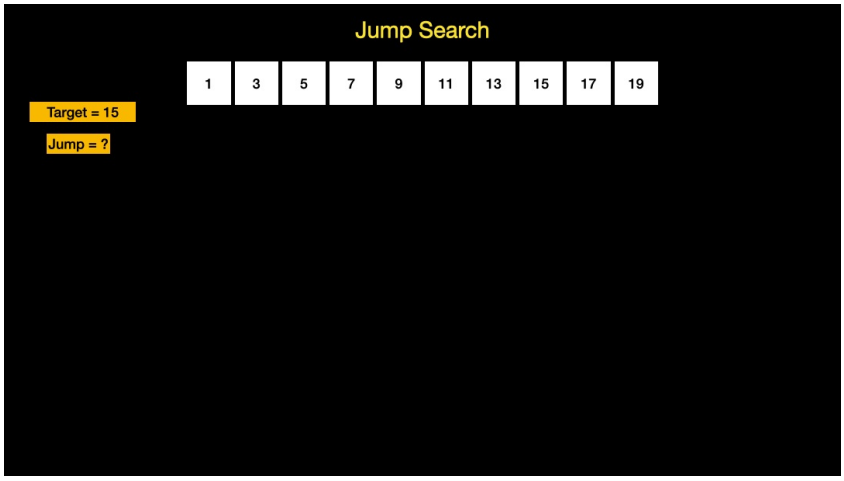
Overview:

Jump Search is an intermediate searching algorithm that falls between linear and binary search in terms of efficiency. It's designed to find a target element within a sorted array by skipping a predefined number of steps. If Jump Search goes too far (i.e., the current element is greater than the target), it takes a step back and performs a linear search to locate the target. This strategy capitalizes on a balance between skipping a large number of elements and refining the search space.

info

Searching Mechanics

The following gif demonstrates how a jump search works:



Steps

1. **Determine Jump Size**: Typically, the jump size is calculated as $\sqrt{n}$, where n is the length of the array.
2. **Initialize Pointers**: Set the initial position to 0.
3. **Jump Ahead**: Jump forward by the step size.
4. **Check Position**:

    - If the element at the current position is equal to the target, return the position.
    - If the element at the current position is greater than the target, move to the linear search phase.
    - If the element at the current position is less than the target, repeat the jump.
5. **Linear Search Phase**: If you have jumped too far (current element > target), backtrack to the previous jump and search linearly up to the current jump position.
6. **Iterate**: Continue this process until the target is found or you have searched the entire array. If you did not find the target, return -1.

Since this is our first time encountering this specific search algorithm, let's walk through a jump search together. Assume we are starting with the following information.

```
Array: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
Target: 15
```

First, find the step size by calculating $\sqrt{10}$, since the length of the array is `10`. The jumps are going to be an index, so we need to remove any decimals from the step size. In this example, we will be jumping by `3` elements.

1. **First Jump**: Start at index 0. Jump 3 steps to index `3`, `Array[3]` = 7 and 7 < 15.
2. **Second Jump**: Jump another 3 steps to index `6`, `Array[6]` = 13 and 13 < 15.
3. **Third Jump**: Jump another 3 steps to index `9`, `Array[9]` = 19 and 9 > 15.
4. **Linear Search Phase**: At this point, we've jumped too far. We backtrack to index `6` and perform a linear search between index `6` and index `9`.
    - **Linear Search 1**: Check index `6`, `Array[6]` = 13 and 13 < 15.
    - **Linear Search 2**: Check index `7`, `Array[7]` = 15 and 15 = 15.
5. **Found**: We find that the element at index `7` is the target value. The search stops and returns the index `7`.

Implementation

Before we begin programming, do note that a jump search commonly employs using a square root. This means you must import the `java.lang.Math` library. Start by creating the `jumpSearch` method. It takes an array of integers, and integer that represents the search target, and the method returns an integer. Calculate the length of the array, and use the length the calculate the step size for each jump. Since taking the square root of a number can return a decimal, use the `Math.floor` method to truncate any decimals. We also need the variable `prev` which we will use to represent the index prior to the jump.

```java
public static int jumpSearch(int[] arr, int target) {
    int n = arr.length;
    int step = (int) Math.floor(Math.sqrt(n));
    int prev = 0;

}
```

We need to be careful to not "jump out of" the array. We are going to do this with a while loop. By saying `arr[Math.min(step, n) - 1]`, we are telling Java to use the most recent jump as the index, unless it were to exceed the length of the array. In that case, use the last element in the array. This ensures that we do not have an index out of bound error.

Keep iterating as long as the index at the `step` value is less than the target. When this happens, set `prev` to `step` and increase the value of `step`. If `prev` is greater than the size of the array, that means the search target is not in the array. Return `-1`.

```java
public static int jumpSearch(int[] arr, int target) {
    int n = arr.length;
    int step = (int) Math.floor(Math.sqrt(n));
    int prev = 0;

    while (arr[Math.min(step, n) - 1] < target) {
        prev = step;
        step += (int) Math.floor(Math.sqrt(n));
        if (prev >= n) {
            return -1; // Target not found
        }
    }

}
```

If the value of `arr[step]` and `arr[n]` is greater than the search target, then we are going to switch to the linear search portion of the algorithm. Start the for loop at the value of `prev`. Continue iterating as long as the loop variable is less than the smallest value between `step` and `n`. If the element at the index of the loop variable is the search target, return the loop variable. If the for loop finishes without returning a value, that means the search target is not in the array. Return `-1`.

```
    public static int jumpSearch(int[] arr, int target) {
        int n = arr.length;
        int step = (int) Math.floor(Math.sqrt(n));
        int prev = 0;

        while (arr[Math.min(step, n) - 1] < target) {
            prev = step;
            step += (int) Math.floor(Math.sqrt(n));
            if (prev >= n) {
                return -1; // Target not found
            }
        }

        // Linear search phase
        for (int i = prev; i < Math.min(step, n); i++) {
            if (arr[i] == target) {
                return i; // Target found, return index
            }
        }
        return -1; // Target not found after linear search
    }
```

The code in the IDE is sending the information below to the `jumpSearch` method. Your code should return `3`.

```
int[] arr = {1, 3, 5, 7, 9, 11, 13, 15};
int target = 7;
```

Jump Search strikes a balance between skipping large portions of the data and ensuring no data is missed. By taking 'jumps' and then refining the search with linear methods, it offers a balance of speed and precision. However, similar to Binary Search, it requires a sorted array.

Jump Search Analysis

Time Complexity

- **Best-Case**: The best-case scenario occurs when the target element is the first element of the array. In this case, the time complexity is $O(1)$.
- **Worst-Case**: The worst-case scenario occurs when the target is either the last element in the array or not present at all. In this case, you'll have to do approximately $(\sqrt{n})$ jumps, and then up to $(\sqrt{n})$ linear searches, giving a worst-case time complexity of $O(\sqrt{n})$.
- **Average-Case**: On average, you can expect the algorithm to perform in $O(\sqrt{n})$ time.

Space Complexity

The space complexity for Jump Search is $O(1)$, as it only uses a few extra variables and does not require any additional data structures like stacks or queues.

Practical Use-Cases

Jump searches work well with large arrays, database indexing, and approximate searching. Approximate searching is when an exact match is not necessary. It is also a good choice for when you want a search that is faster than linear search, but has an algorithm that is a bit easier to understand than some more advanced searching solutions.

Comparison with Other Algorithms

- **Balance**: Jump search sit between linear and binary search in terms of performance and complexity.
- **Preconditions**: Like binary search, a jump search requires the input array to be sorted.
- **Distribution**: Jump searches perform better when elements are uniformly distributed, as each 'jump' can eliminate a consistent number of elements from consideration.
- **Jumping Too Far**: Switching to linear search after jumping too far ensures that the algorithm will not miss the target element. However, it does mean that the actual search range is not always cut in half or reduced by $(\sqrt{n})$, which contributes to its worst-case time complexity.

Jump Search offers a balanced compromise between linear and binary search in both implementation complexity and performance. It's particularly useful when you have a read-only memory, as it does not require the array to be divided like in Binary Search. Jump searches allow you to find a balance between speed and simplicity.

Formative Assessment 1

Formative Assessment 2