**What is an Array?**

An **array** is a data structure in programming that can store multiple elements of the same type in a contiguous block of memory. Arrays provide an efficient way to manage and access a collection of data. Each element in an array is identified by an **index**.

**Key Features of Arrays:**

1. **Fixed Size**: The size of the array is defined at the time of declaration and cannot be changed.
2. **Index-Based Access**: Each element in the array can be accessed using its index, starting from 0.
3. **Homogeneous Elements**: All elements in an array must be of the same data type (e.g., integers, strings, floats, objects).
4. **Continuous Memory Allocation**: Elements of an array are stored in adjacent memory locations.

**Declaration and Initialization of Arrays**

1. **Declaration**:

dataType[] arrayName; // Recommended syntax

dataType arrayName[]; // Also valid

1. **Initialization**:
   * **Static Initialization**: Define size and values.

int[] numbers = {10, 20, 30, 40};

* + **Dynamic Initialization**: Define size only, then assign values later.

int[] numbers = new int[4]; // Array of size 4

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

**Example in Java:**

**1. Basic Array Example**

|  |
| --- |
| public class ArrayExample {  public static void main(String[] args) {  // Declare and initialize the array  int[] numbers = {5, 10, 15, 20, 25};  // Access and print elements  System.out.println("First element: " + numbers[0]); // 5  System.out.println("Second element: " + numbers[1]); // 1  // Iterate through the array  System.out.println("All elements:");  for (int i = 0; i < numbers.length; i++) {  System.out.println(numbers[i]);  }  }  } |

**Output**:

|  |
| --- |
| First element: 5  Second element: 10  All elements:  5  10  15  20  25 |

**Multi-Dimensional Arrays**

An array can have multiple dimensions, such as a 2D array, which is essentially an array of arrays.

**Example: 2D Array**

|  |
| --- |
| public class MultiDimArrayExample {  public static void main(String[] args) {  // Declare and initialize a 2D array  int[][] matrix = {  {1, 2, 3},  {4, 5, 6},  {7, 8, 9}  };    // Access elements  System.out.println("Element at (0, 1): " + matrix[0][1]); // 2    // Iterate through the 2D array  System.out.println("Matrix elements:");  for (int i = 0; i < matrix.length; i++) {  for (int j = 0; j < matrix[i].length; j++) {  System.out.print(matrix[i][j] + " ");  }  System.out.println();  }  }  } |

**Output**:

|  |
| --- |
| Element at (0, 1): 2  Matrix elements:  1 2 3  4 5 6  7 8 9 |

**Common Operations on Arrays**

1. **Finding the Length**:

|  |
| --- |
| 1. int[] arr = {10, 20, 30}; 2. System.out.println("Array length: " + arr.length); // Output: 3 |

1. **Searching for an Element**:

|  |
| --- |
| 1. int[] arr = {10, 20, 30}; 2. int target = 20; 3. for (int i = 0; i < arr.length; i++) { 4. if (arr[i] == target) { 5. System.out.println("Found at index: " + i); 6. } 7. } |

1. **Sorting an Array** (Using Arrays utility class):

|  |
| --- |
| 1. import java.util.Arrays; 2. public class SortArray { 3. public static void main(String[] args) { 4. int[] arr = {30, 10, 20, 50}; 5. Arrays.sort(arr); // Sorts in ascending order 6. System.out.println("Sorted Array: " + Arrays.toString(arr)); 7. } 8. } |

**Output**:

|  |
| --- |
| Sorted Array: [10, 20, 30, 50] |

**Advantages of Arrays:**

1. **Fast Access**: Constant time complexity O(1)O(1)O(1) for accessing elements using an index.
2. **Efficient Memory Usage**: Contiguous memory allocation reduces overhead.
3. **Simple Implementation**: Easy to use for fixed-size collections.

**Disadvantages of Arrays:**

1. **Fixed Size**: Cannot dynamically resize after creation (use ArrayList for this).
2. **Insertion/Deletion Overhead**: Adding or removing elements can be inefficient due to shifting.
3. **Homogeneity**: All elements must be of the same type.

This covers the basics of arrays with examples and practical implementations. Let me know if you need help with more advanced concepts!

**Searching technique in java with Examples using both iterative and recursive technique.**

**Linear Search**

Linear search is one of the simplest searching algorithms. It works by checking every element in a list or array sequentially to find the target value. If the target is found, it returns the index of the element. If the element is not found, it returns an indication that the target is not present (often -1).

**1. Iterative Linear Search**

In the iterative version of linear search, we use a loop to check each element of the array or list until we find the target or exhaust the list.

**Steps:**

1. Start from the first element.
2. Compare the current element with the target.
3. If a match is found, return the index of the element.
4. If the target is not found after checking all elements, return -1.

**Example:** Consider an array of integers: [4, 2, 7, 5, 1, 3] and we want to find the target 5.

|  |
| --- |
| public class LinearSearch {  public static int iterativeLinearSearch(int[] arr, int target) {  // Iterate through each element in the array  for (int i = 0; i < arr.length; i++) {  if (arr[i] == target) {  return i; // Return the index if the target is found  }  }  return -1; // Return -1 if the target is not found  }  public static void main(String[] args) {  int[] arr = {4, 2, 7, 5, 1, 3};  int target = 5;  int index = iterativeLinearSearch(arr, target);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 3

**2. Recursive Linear Search :** In the recursive version, the search function calls itself, reducing the size of the problem (by checking one fewer element on each recursive call), until either the target is found or the list is exhausted.

**Steps:**

1. Start by comparing the first element of the array with the target.
2. If it matches, return the index.
3. If it doesn’t, make a recursive call to search the rest of the array (starting from the next element).
4. If the target is not found after all recursive calls, return -1.

**Example:** Consider the same array: [4, 2, 7, 5, 1, 3] and we want to find the target 5.

|  |
| --- |
| public class LinearSearch {  public static int recursiveLinearSearch(int[] arr, int target, int index) {  // Base case: If index exceeds the array length, return -1  if (index >= arr.length) {  return -1;  }  // Check if the current element matches the target  if (arr[index] == target) {  return index; // Return the index if the target is found  }  // Recursive case: Move to the next element  return recursiveLinearSearch(arr, target, index + 1);  }  public static void main(String[] args) {  int[] arr = {4, 2, 7, 5, 1, 3};  int target = 5;  int index = recursiveLinearSearch(arr, target, 0);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 3

**Comparison of Iterative and Recursive Linear Search**

* **Time Complexity**: Both the iterative and recursive versions of linear search have the same time complexity of **O(n)**, where n is the number of elements in the array.
* **Space Complexity**:
  + **Iterative**: O(1), as it uses only a fixed amount of space for variables like the loop counter and the target.
  + **Recursive**: O(n), as each recursive call uses additional space in the call stack. If the array is very large, this may lead to a stack overflow.

**Binary Search**

Binary search is a more efficient algorithm for finding a target value within a **sorted** array. Unlike linear search, which checks every element sequentially, binary search repeatedly divides the search space in half. This makes binary search much faster, with a time complexity of **O(log n)**, where n is the number of elements in the array.

Binary search works by comparing the target value to the middle element of the array:

* If the target is equal to the middle element, the search ends.
* If the target is smaller, the search continues in the left half of the array.
* If the target is larger, the search continues in the right half of the array.

Binary search can be implemented in two ways:

1. **Iterative Binary Search**: Uses a loop to reduce the search space.
2. **Recursive Binary Search**: Uses recursion to divide the array and search for the target.

**1. Iterative Binary Search:** In the iterative version of binary search, we maintain two pointers (left and right) that track the current search range. The middle element is calculated at each step, and depending on the comparison, the search space is reduced.

**Steps:**

1. Initialize left to 0 and right to the last index of the array.
2. Calculate the middle index mid = (left + right) / 2.
3. Compare the middle element to the target:
   * If the middle element is equal to the target, return the index.
   * If the middle element is smaller than the target, move the left pointer to mid + 1.
   * If the middle element is larger than the target, move the right pointer to mid - 1.
4. Repeat the process until the target is found or the left pointer exceeds the right pointer (indicating the target is not in the array).

**Example:** Consider an array: [1, 2, 4, 5, 7, 8, 9] and we want to find the target 5.

|  |
| --- |
| public class BinarySearch {  public static int iterativeBinarySearch(int[] arr, int target) {  int left = 0;  int right = arr.length - 1;  while (left <= right) {  int mid = left + (right - left) / 2; // To avoid potential overflow    if (arr[mid] == target) {  return mid; // Return the index if the target is found  }  if (arr[mid] < target) {  left = mid + 1; // Target is on the right half  } else {  right = mid - 1; // Target is on the left half  }  }  return -1; // Return -1 if the target is not found  }  public static void main(String[] args) {  int[] arr = {1, 2, 4, 5, 7, 8, 9};  int target = 5;  int index = iterativeBinarySearch(arr, target);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 3. in this example, the target 5 is found at index 3.

**2. Recursive Binary Search:** In the recursive version, the function calls itself with a reduced search space. The base case occurs when the target is found or the search space is exhausted.

**Steps:**

1. If the left pointer is greater than the right pointer, the target is not in the array, and we return -1.
2. Calculate the middle index mid = (left + right) / 2.
3. Compare the middle element to the target:
   * If the middle element equals the target, return the index.
   * If the middle element is smaller, recursively search the right half (left = mid + 1).
   * If the middle element is larger, recursively search the left half (right = mid - 1).

**Example:** Consider the same array: [1, 2, 4, 5, 7, 8, 9] and we want to find the target 5.

|  |
| --- |
| public class BinarySearch {  public static int recursiveBinarySearch(int[] arr, int target, int left, int right) {  if (left > right) {  return -1; // Return -1 if the target is not found  }  int mid = left + (right - left) / 2; // To avoid potential overflow    if (arr[mid] == target) {  return mid; // Return the index if the target is found  }  if (arr[mid] < target) {  return recursiveBinarySearch(arr, target, mid + 1, right); // Search right half  } else {  return recursiveBinarySearch(arr, target, left, mid - 1); // Search left half  }  }  public static void main(String[] args) {  int[] arr = {1, 2, 4, 5, 7, 8, 9};  int target = 5;  int index = recursiveBinarySearch(arr, target, 0, arr.length - 1);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 3

**Comparison of Iterative and Recursive Binary Search**

* **Time Complexity**: Both the iterative and recursive versions of binary search have the same time complexity of **O(log n)**, where n is the number of elements in the array.
* **Space Complexity**:
  + **Iterative**: O(1), as it uses only a fixed amount of space for the pointers.
  + **Recursive**: O(log n), as each recursive call adds a new stack frame to the call stack.

**Ternary Search**

Ternary search is a divide-and-conquer algorithm similar to binary search but instead of dividing the array into two parts, it divides the array into three parts. Ternary search is used for finding the maximum or minimum of a unimodal function (i.e., a function that is either increasing then decreasing or decreasing then increasing). In Ternary Search, the search space is divided into three equal parts, and the middle two points are evaluated. Based on the comparison, the search space is reduced to the region that could contain the target, and this process is repeated recursively or iteratively.

**Key Properties of Ternary Search:**

1. The array must be **sorted**.
2. It divides the array into three segments, unlike binary search, which divides it into two.
3. Ternary search is often used for finding a peak element in an array or for optimization problems.

**Steps for Ternary Search:**

1. Find two middle points mid1 and mid2 such that:
   * mid1 = left + (right - left) / 3
   * mid2 = right - (right - left) / 3
2. Compare the target element with arr[mid1] and arr[mid2]:
   * If the target equals arr[mid1], return mid1.
   * If the target equals arr[mid2], return mid2.
   * If the target is smaller than arr[mid1], narrow the search to the left part (left to mid1 - 1).
   * If the target is greater than arr[mid2], narrow the search to the right part (mid2 + 1 to right).
   * If the target is between arr[mid1] and arr[mid2], narrow the search to the middle part (mid1 + 1 to mid2 - 1).

Ternary search can be implemented both **iteratively** and **recursively**.

**1. Iterative Ternary Search** The iterative approach uses a loop to keep dividing the search space until the target is found or the search space becomes empty.

**Steps:**

1. Start with the entire array (left = 0 and right = n-1).
2. In each iteration, calculate mid1 and mid2.
3. Narrow the search range based on the comparisons of the target with arr[mid1] and arr[mid2].

**Example:** Consider an array: [1, 3, 5, 7, 9, 11, 13, 15] and we want to find the target 9.

|  |
| --- |
| public class TernarySearch {  public static int iterativeTernarySearch(int[] arr, int target) {  int left = 0;  int right = arr.length - 1;  while (left <= right) {  int mid1 = left + (right - left) / 3;  int mid2 = right - (right - left) / 3;  if (arr[mid1] == target) {  return mid1; // Return the index if the target is found at mid1  }  if (arr[mid2] == target) {  return mid2; // Return the index if the target is found at mid2  }  // Narrow the search based on the target's position  if (target < arr[mid1]) {  right = mid1 - 1; // Search in the left part  } else if (target > arr[mid2]) {  left = mid2 + 1; // Search in the right part  } else {  left = mid1 + 1; // Search in the middle part  right = mid2 - 1;  }  }  return -1; // Return -1 if the target is not found  }  public static void main(String[] args) {  int[] arr = {1, 3, 5, 7, 9, 11, 13, 15};  int target = 9;  int index = iterativeTernarySearch(arr, target);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 4. in this example, the target 9 is found at index 4 of the array.

**2. Recursive Ternary Search:** In the recursive version, the function calls itself with updated search ranges, dividing the array into three parts at each level of recursion.

**Steps:**

1. If left > right, return -1, indicating the target is not found.
2. Otherwise, calculate the two middle points mid1 and mid2.
3. Compare the target with arr[mid1] and arr[mid2]:
   * If the target equals arr[mid1], return mid1.
   * If the target equals arr[mid2], return mid2.
4. Based on the target's comparison with mid1 and mid2, recursively search the relevant part of the array.

**Example:** Consider the same array: [1, 3, 5, 7, 9, 11, 13, 15] and we want to find the target 9.

|  |
| --- |
| public class TernarySearch {  public static int recursiveTernarySearch(int[] arr, int target, int left, int right) {  if (left > right) {  return -1; // Base case: target not found  }  int mid1 = left + (right - left) / 3;  int mid2 = right - (right - left) / 3;  if (arr[mid1] == target) {  return mid1; // Return the index if the target is found at mid1  }  if (arr[mid2] == target) {  return mid2; // Return the index if the target is found at mid2  }  // Recursively search based on the target's value  if (target < arr[mid1]) {  return recursiveTernarySearch(arr, target, left, mid1 - 1); // Search in the left part  } else if (target > arr[mid2]) {  return recursiveTernarySearch(arr, target, mid2 + 1, right); // Search in the right part  } else {  return recursiveTernarySearch(arr, target, mid1 + 1, mid2 - 1); // Search in the middle part  }  }  public static void main(String[] args) {  int[] arr = {1, 3, 5, 7, 9, 11, 13, 15};  int target = 9;  int index = recursiveTernarySearch(arr, target, 0, arr.length - 1);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 4, In this example, the target 9 is found at index 4 of the array.

**Comparison of Iterative and Recursive Ternary Search**

* **Time Complexity**:
  + Both the iterative and recursive versions of ternary search have the same time complexity of **O(log₃ n)**, where n is the number of elements in the array. This is because each iteration (or recursion) reduces the search space to a third of its previous size.
* **Space Complexity**:
  + **Iterative**: O(1), as it only uses a constant amount of space for the variables (left, right, mid1, mid2).
  + **Recursive**: O(log₃ n), because each recursive call adds a new stack frame, and the maximum depth of recursion is proportional to the logarithm of the array size base 3.

**Jump Search**

Jump Search is an algorithm designed for searching in **sorted arrays** or lists. It works by dividing the array into smaller blocks (or "jump sizes") and then performing a linear search within the block that is likely to contain the target. Jump Search combines the efficiency of linear search with the benefits of skipping ahead, hence reducing the number of comparisons compared to a brute-force linear search. Jump search has the following key features:

* It works best for **sorted** arrays or lists.
* It divides the array into **blocks** and jumps through them.
* The ideal jump size is typically the square root of the array length, i.e., **√n**, which minimizes the number of comparisons.

**Key Properties:**

1. **Jump Size**: The array is divided into blocks of size √n. This jump size balances the trade-off between linear search and the time it takes to find the correct block.
2. **Complexity**:
   * **Time Complexity**: The time complexity of Jump Search is **O(√n)**, where n is the number of elements in the array. This is because we jump in increments of √n, and once we find a block that might contain the target, we perform a linear search on it.
   * **Space Complexity**: The space complexity of Jump Search is **O(1)**, since it only requires a constant amount of additional space.

**Steps for Jump Search:**

1. **Set jump size**: Calculate the block size as blockSize = √n, where n is the number of elements in the array.
2. **Jump ahead**: Start at index 0 and jump forward in increments of blockSize until you find an element that is greater than or equal to the target element.
3. **Linear search**: Once the jump size is found, perform a linear search backward within that block to find the target.

**1. Iterative Jump Search:** In the iterative approach, we calculate the block size (√n), jump ahead, and then perform a linear search within the block if necessary.

**Example:** Consider the array: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21] and the target 15.

|  |
| --- |
| public class JumpSearch {  // Iterative Jump Search  public static int iterativeJumpSearch(int[] arr, int target) {  int n = arr.length;  int blockSize = (int) Math.sqrt(n); // Calculate block size  int left = 0;  int right = 0;  // Jump ahead in blocks of size 'blockSize'  while (right < n && arr[right] < target) {  left = right;  right += blockSize;  if (right > n - 1) right = n; // Prevent out of bounds  }  // Linear search within the block  for (int i = left; i < right; i++) {  if (arr[i] == target) {  return i; // Return index if target is found  }  }  return -1; // Return -1 if target is not found  }  public static void main(String[] args) {  int[] arr = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21};  int target = 15;  int index = iterativeJumpSearch(arr, target);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 7, In this example, the target 15 is found at index 7 of the array.

**Explanation:**

1. **Block Size**: The block size is calculated as √11 = 3.
2. **Jumping**: We jump ahead by 3 each time. First jump from index 0 to 3 (value 7), then from 3 to 6 (value 13), and from 6 to 9 (value 17).
3. **Linear Search**: After jumping to the block containing 15, we perform a linear search within the block [13, 15, 17] and find the target 15 at index 7.

**2. Recursive Jump Search:** In the recursive version of Jump Search, we use recursion to perform the jumping and linear search instead of using a loop.

**Example:** Consider the same array: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21] and the target 15.

|  |
| --- |
| public class JumpSearch {  // Recursive Jump Search  public static int recursiveJumpSearch(int[] arr, int target, int left, int right) {  int n = arr.length;  int blockSize = (int) Math.sqrt(n); // Calculate block size  if (right >= n) right = n - 1; // Bound check for right index  // Jump ahead in blocks of size 'blockSize'  if (arr[right] < target) {  left = right;  right = Math.min(right + blockSize, n); // Move right by block size  if (right >= n) right = n; // Prevent out of bounds  return recursiveJumpSearch(arr, target, left, right); // Recursive call  }  // Linear search within the block  for (int i = left; i <= right; i++) {  if (arr[i] == target) {  return i; // Return index if target is found  }  }  return -1; // Return -1 if target is not found  }  public static void main(String[] args) {  int[] arr = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21};  int target = 15;  int index = recursiveJumpSearch(arr, target, 0, 0);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 7

**Explanation:**

1. **Recursive Call**: The function recursiveJumpSearch calls itself with updated left and right indices, jumping ahead by the block size (√n).
2. **Linear Search**: After jumping to the block containing the target, the algorithm performs a linear search within the block to find the target.

**Key Points for Recursive vs. Iterative:**

* **Iterative Jump Search** is simpler to implement in most cases and avoids the overhead of recursive function calls.
* **Recursive Jump Search** is more elegant and can be useful in scenarios where recursion is preferred for problem-solving, but the depth of recursion must be carefully controlled to avoid stack overflow in large arrays.

**Comparison of Iterative and Recursive Jump Search**

* **Time Complexity**: Both iterative and recursive Jump Search algorithms have the same time complexity of **O(√n)**, where n is the number of elements in the array. This is because the number of jumps is reduced to √n and then the linear search within the block takes up to √n comparisons.
* **Space Complexity**:
  + **Iterative Jump Search**: **O(1)**, since it uses a constant amount of additional memory (no recursion stack).
  + **Recursive Jump Search**: **O(log n)**, because recursion depth is proportional to the number of jumps made, leading to a logarithmic depth.

**Exponential Search**

Exponential Search is a search algorithm designed for sorted arrays. It is a hybrid of **binary search** and a growing search step. It works by finding the range where the target value might exist using exponential steps, and then performs a **binary search** in that range. This algorithm is particularly efficient when the target value is expected to be located closer to the beginning of the array.

**Steps for Exponential Search**

1. **Start at the first element** and check if it matches the target.
2. **Grow exponentially**: Double the range (indices) to find a boundary where the target might exist. For example, check indices 1, 2, 4, 8, 16, etc., until the value at the current index exceeds or equals the target, or the index goes out of bounds.
3. **Perform binary search**: Once the range is identified (from the previous step), apply binary search within that range.

**Time Complexity**

* **Best Case**: O(1)O(1)O(1), if the target is the first element.
* **Average Case**: O(log⁡i+log⁡n)O(\log i + \log n)O(logi+logn), where iii is the range boundary and nnn is the size of the array.
* **Worst Case**: O(log⁡n)O(\log n)O(logn), because the doubling step has a logarithmic complexity, and binary search within the range is also logarithmic.

**Space Complexity**

* **Iterative Approach**: O(1)O(1)O(1)
* **Recursive Approach**: O(log⁡n)O(\log n)O(logn), due to the recursive calls in binary search.

**1. Iterative Exponential Search**

**Example:** Given a sorted array: arr = [2, 3, 4, 10, 40, 50, 80, 100, 120] and a target 40.

|  |
| --- |
| import java.util.Arrays;  public class ExponentialSearch {  // Iterative Binary Search  public static int binarySearchIterative(int[] arr, int left, int right, int target) {  while (left <= right) {  int mid = left + (right - left) / 2;  if (arr[mid] == target) {  return mid; // Target found  } else if (arr[mid] < target) {  left = mid + 1; // Search in the right half  } else {  right = mid - 1; // Search in the left half  }  }  return -1; // Target not found  }  // Iterative Exponential Search  public static int exponentialSearchIterative(int[] arr, int target) {  int n = arr.length;  // f the first element is the target  if arr[0] == target) return 0;  // Exponentially find the range where the target might exist  int i = 1;  while (i < n && arr[i] <= target) {  i = i \* 2; // Double the range  }  // Perform binary search within the identified range  return binarySearchIterative(arr, i / 2, Math.min(i, n - 1), target);  }  public static void main(String[] args) {  int[] arr = {2, 3, 4, 10, 40, 50, 80, 100, 120};  int target = 40;  int index = exponentialSearchIterative(arr, target);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 4

**2. Recursive Exponential Search:** In the recursive implementation, we use recursion for both exponential search and binary search.

**Example:** Using the same array: arr = [2, 3, 4, 10, 40, 50, 80, 100, 120] and target 40.

|  |
| --- |
| public class ExponentialSearch {  // Recursive Binary Search  public static int binarySearchRecursive(int[] arr, int left, int right, int target) {  if (left > right) {  return -1; // Base case: target not found  }  int mid = left + (right - left) / 2;  if (arr[mid] == target) {  return mid; // Target found  } else if (arr[mid] < target) {  return binarySearchRecursive(arr, mid + 1, right, target); // Search in the right half  } else {  return binarySearchRecursive(arr, left, mid - 1, target); // Search in the left half  }  }  // Recursive Exponential Search  public static int exponentialSearchRecursive(int[] arr, int target, int i) {  if (i >= arr.length || arr[i] > target) {  // Perform binary search in the range  return binarySearchRecursive(arr, i / 2, Math.min(i, arr.length - 1), target);  }  // Recursive step: double the range  return exponentialSearchRecursive(arr, target, i \* 2);  }  public static void main(String[] args) {  int[] arr = {2, 3, 4, 10, 40, 50, 80, 100, 120};  int target = 40;  // Start exponential search with index 1  int index = exponentialSearchRecursive(arr, target, 1);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 4

**Explanation**

1. **Exponential Step**:
   * Start with the first index.
   * Double the index (i = 1, 2, 4, 8, ...) until either:
     + arr[i]>target\text{arr}[i] > \text{target}arr[i]>target, or
     + iii exceeds the array bounds.
2. **Binary Search**:
   * Perform a binary search within the identified range [i/2, min(i, n-1)].

### Fibonacci Search

### Fibonacci Search is an efficient searching algorithm for sorted arrays. It uses the Fibonacci sequence to divide the array into smaller segments, leveraging the property that a Fibonacci number can define an offset index for partitioning the array. It is similar to Binary Search but differs in how it divides the search range.

**Steps for Fibonacci Search**

1. **Initialize Fibonacci numbers**:
   * Find the smallest Fibonacci number that is greater than or equal to the size of the array.
2. **Set up indices**:
   * Use the Fibonacci sequence to find offsets for dividing the array.
3. **Search for the target**:
   * Compare the target value with the value at the current index.
   * Narrow down the range based on the comparison:
     + If the target is smaller, move left.
     + If larger, move right.
4. **Continue until the target is found**:
   * Reduce the range by updating the Fibonacci offsets.

**Fibonacci Numbers**

The Fibonacci sequence is: F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2), with base cases F(0)=0F(0) = 0F(0)=0 and F(1)=1F(1) = 1F(1)=1.

**Time Complexity**

* **Best Case**: O(1)O(1)O(1) (target is found at the first comparison).
* **Average Case**: O(log⁡n)O(\log n)O(logn), proportional to the logarithmic growth of the Fibonacci sequence.
* **Worst Case**: O(log⁡n)O(\log n)O(logn).

**Space Complexity**

* **Iterative Approach**: O(1)O(1)O(1).
* **Recursive Approach**: O(log⁡n)O(\log n)O(logn), due to recursion stack.

**1. Iterative Fibonacci Search Example:** Given a sorted array: arr = [10, 22, 35, 40, 45, 50, 80, 82, 90, 100] and a target 45.

|  |
| --- |
| public class FibonacciSearch {  public static int fibonacciSearchIterative(int[] arr, int target) {  int n = arr.length;  // Initialize Fibonacci numbers  int fib2 = 0; // F(n-2)  int fib1 = 1; // F(n-1)  int fib = fib2 + fib1; // F(n)  // Find the smallest Fibonacci number >= n  while (fib < n) {  fib2 = fib1;  fib1 = fib;  fib = fib2 + fib1;  }  // Marks the eliminated range from front  int offset = -1;  // While there are elements to be inspected  while (fib > 1) {  // Check if fib2 is a valid index  int i = Math.min(offset + fib2, n - 1);  // Target found  if (arr[i] == target) {  return i;  }  // If target is greater, move right  if (arr[i] < target) {  fib = fib1;  fib1 = fib2;  fib2 = fib - fib1;  offset = i; // Update offset  }  // If target is smaller, move left  else {  fib = fib2;  fib1 = fib1 - fib2;  fib2 = fib - fib1;  }  }  // Compare the last element with target  if (fib1 == 1 && arr[offset + 1] == target) {  return offset + 1;  }  // Target not found  return -1;  }  public static void main(String[] args) {  int[] arr = {10, 22, 35, 40, 45, 50, 80, 82, 90, 100};  int target = 45;  int index = fibonacciSearchIterative(arr, target);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 4

**2. Recursive Fibonacci Search:** In the recursive implementation, the Fibonacci numbers and the offset are updated recursively.

**Example:** Using the same array: arr = [10, 22, 35, 40, 45, 50, 80, 82, 90, 100] and target 45.

|  |
| --- |
| public class FibonacciSearch {  public static int fibonacciSearchRecursive(int[] arr, int target, int fib2, int fib1, int fib, int offset) {  // Base case: if there are no more elements to inspect  if (fib <= 1) {  return (fib1 == 1 && arr[offset + 1] == target) ? offset + 1 : -1;  }  // Calculate the current index  int i = Math.min(offset + fib2, arr.length - 1);  // If target is found  if (arr[i] == target) {  return i;  }  // If target is greater, search in the right subarray  if (arr[i] < target) {  return fibonacciSearchRecursive(arr, target, fib1, fib - fib1, fib1, i);  }  // If target is smaller, search in the left subarray  else {  return fibonacciSearchRecursive(arr, target, fib2, fib1 - fib2, fib2, offset);  }  }  public static int fibonacciSearchRecursive(int[] arr, int target) {  // Initialize Fibonacci numbers  int fib2 = 0; // F(n-2)  int fib1 = 1; // F(n-1)  int fib = fib2 + fib1; // F(n)  // Find the smallest Fibonacci number >= n  while (fib < arr.length) {  fib2 = fib1;  fib1 = fib;  fib = fib2 + fib1;  }  // Start the recursive search  return fibonacciSearchRecursive(arr, target, fib2, fib1, fib, -1);  }  public static void main(String[] args) {  int[] arr = {10, 22, 35, 40, 45, 50, 80, 82, 90, 100};  int target = 45;  int index = fibonacciSearchRecursive(arr, target);  if (index != -1) {  System.out.println("Target found at index: " + index);  } else {  System.out.println("Target not found.");  }  }  } |

**Output**: Target found at index: 4

**Explanation**

1. **Fibonacci Initialization**:
   * Generate Fibonacci numbers to define search offsets.
2. **Divide and Conquer**:
   * Use the offset fib2\text{fib2}fib2 to determine the partition of the array.
3. **Adjust Search Range**:
   * If the target is greater, search the right part by reducing Fibonacci numbers.
   * If the target is smaller, search the left part similarly.