**Binary search**

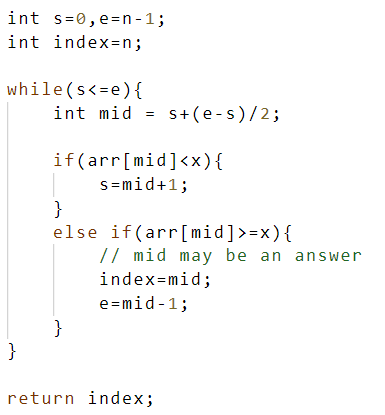
Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

**Conditions for when to apply Binary Search in a Data Structure:**

>>The data structure must be sorted.

>>Access to any element of the data structure takes constant time.

**Implement Lower Bound**



**Search Insert Position**

Same as lower bound

## Floor and Ceil in Sorted Array

Celi=lower bound

Floor=upper bound

**First and Last occurrence of an element**

Ap 1: naive approach

1. Traverse the array and find the fast and last occ..

Time complexity: O( log(N) ) Space complexity: O(1)

Ap 2: Binary search (optimal)

Time complexity: O( log(N) ) Space complexity: O(1)

public static int[] firstAndLastPosition(ArrayList<Integer> arr, int n, int k) {

        int first = -1;

        int last = -1;

        int si = 0;

        int ei = n - 1;

        // To find first position

        while (si <= ei) {

            int mid = si + (ei - si) / 2;

            if (arr.get(mid) == k) {

                first = mid;

                ei = mid - 1;

            } else if (arr.get(mid) < k) {

                si = mid + 1;

            } else {

                ei = mid - 1;

            }

        }

        si = 0;

        ei = n - 1;

        // To find last position

        while (si <= ei) {

            int mid = si + (ei - si) / 2;

            if (arr.get(mid) == k) {

                last = mid;

                si = mid + 1;

            } else if (arr.get(mid) < k) {

                si = mid + 1;

            } else {

                ei = mid - 1;

            }

        }

        return new int[]{first,last};

}

## Count Occurrences in Sorted Array

lastOcc - firstOcc + 1

**Peak index in mountain array**

Ap 1: linear scan

1. Create an integer variable i and initialize it to 0.
2. Using a while loop check if the current element pointed by i is smaller than the next element at index i + 1. If arr[i] < arr[i + 1], increment i by 1. Otherwise, if arr[i] > arr[i + 1], we return i.

Time complexity: O( N ) Space complexity: O(1)

Ap 2: Binary seach (optimal)

1. Create two integer variables l=0 and r = arr.length - 1.
2. While l < r:
   1. Get the index of the middle element using mid = (l + r) / 2.
   2. If arr[mid] < arr[mid + 1], it indicates peak index is greater than mid. As a result, we move to upper half of the range by setting left = mid + 1.
   3. Else, if arr[mid] >= arr[mid + 1], it indicates that the peak index is either mid or some index smaller than mid. As a result, we move to the lower half of the range by setting r = mid-1
3. Return mid

Time complexity: O( log(N) ) Space complexity: O(1)

public static int findPeakElement(ArrayList<Integer> arr) {

    // Get the size of the ArrayList

    int n = arr.size();

    // Check if the first element is greater than the second element, if so, it's a peak

    if (arr.get(0) > arr.get(1)) {

        return 0;

    }

    // Check if the last element is greater than the second to last element, if so, it's a peak

    if (arr.get(n - 1) > arr.get(n - 2)) {

        return n - 1;

    }

    // Binary search for the peak element within the array

    int start = 1; // Start index of the array

    int end = n - 2; // End index of the array

    while (start <= end) {

        int mid = start + (end - start) / 2; // Calculate the middle index

        // Check if the middle element is greater than its neighbors, indicating a peak

        if (arr.get(mid) > arr.get(mid - 1) && arr.get(mid) > arr.get(mid + 1)) {

            return mid;

        } else if (arr.get(mid) < arr.get(mid + 1)) {

            // If the middle element is less than its right neighbor, move towards the right

            start = mid + 1;

        } else {

            // If the middle element is less than its left neighbor, move towards the left

            end = mid - 1;

        }

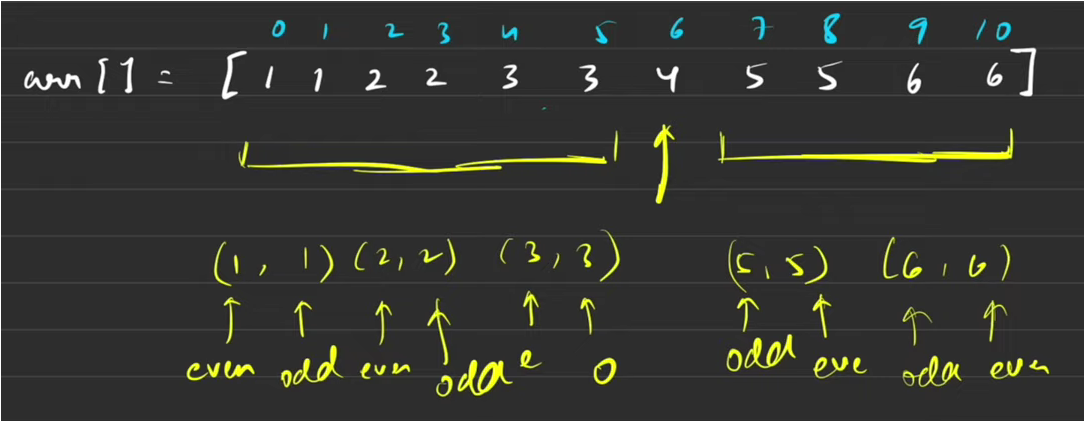
    }

    // If no peak is found, return -1

    return -1;

}

**Single Element in a Sorted Array**



public static int singleNonDuplicate(ArrayList<Integer> arr)

    {

        int n = arr.size(); // Size of the array.

        // Edge cases:

        if (n == 1)

            return arr.get(0);

        if (!arr.get(0).equals(arr.get(1)))

            return arr.get(0);

        if (!arr.get(n - 1).equals(arr.get(n - 2)))

            return arr.get(n - 1);

        int low = 1, high = n - 2;

        while (low <= high) {

            int mid = (low + high) / 2;

            // If arr[mid] is the single element:

            if (!arr.get(mid).equals(arr.get(mid + 1)) && !arr.get(mid).equals(arr.get(mid - 1))) {

                return arr.get(mid);

            }

            // We are in the left:

            if ((mid % 2 == 1 && arr.get(mid).equals(arr.get(mid - 1)))

                    || (mid % 2 == 0 && arr.get(mid).equals(arr.get(mid + 1)))) {

                // Eliminate the left half:

                low = mid + 1;

            }

            // We are in the right:

            else {

                // Eliminate the right half:

                high = mid - 1;

            }

        }

        // Dummy return statement:

        return -1;

}

**Search in Rotated and Sorted Array**

Though the array is rotated, we can clearly notice that for every index, one of the 2 halves will always be sorted. In the above example, the right half of the index mid is sorted.

So, to efficiently search for a target value using this observation, we will follow a simple two-step process.

1. First, we identify the sorted half of the array.
2. Once found, we determine if the target is located within this sorted half.
   * 1. If not, we eliminate that half from further consideration.
     2. Conversely, if the target does exist in the sorted half, we eliminate the other half.

public static int search(ArrayList<Integer> arr, int n, int k) {

        int low = 0, high = n - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            // if mid points to the target

            if (arr.get(mid) == k)

                return mid;

            // if left part is sorted

            if (arr.get(low) <= arr.get(mid)) {

                if (arr.get(low) <= k && k <= arr.get(mid)) {

                    // element exists

                    high = mid - 1;

                } else {

                    // element does not exist

                    low = mid + 1;

                }

            } else { // if right part is sorted

                if (arr.get(mid) <= k && k <= arr.get(high)) {

                    // element exists

                    low = mid + 1;

                } else {

                    // element does not exist

                    high = mid - 1;

                }

            }

        }

        return -1;

    }

**Find Minimum in Rotated Sorted Array**

public static int findMin(int []arr) {

        int low = 0, high = arr.length - 1;

        int ans = Integer.MAX\_VALUE;

        while (low <= high) {

            int mid = (low + high) / 2;

            //search space is already sorted

            //then arr[low] will always be

            //the minimum in that search space:

            if (arr[low] <= arr[high]) {

                ans = Math.min(ans, arr[low]);

                break;

            }

            //if left part is sorted:

            if (arr[low] <= arr[mid]) {

                // keep the minimum:

                ans = Math.min(ans, arr[low]);

                // Eliminate left half:

                low = mid + 1;

            } else { //if right part is sorted:

                // keep the minimum:

                ans = Math.min(ans, arr[mid]);

                // Eliminate right half:

                high = mid - 1;

            }

        }

        return ans;

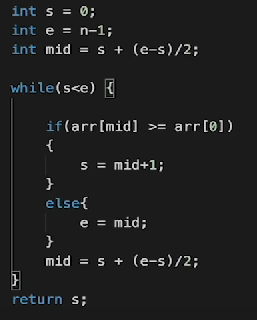
    }

**Rotation**

You are given an array 'arr' having 'n' distinct integers sorted in ascending order. The array is right rotated 'r' times.

Same as finding min in rotated sorted array but keep a track of index for min. That will be ans(r)

**Pivot element (min element) in sorted and rotated array**

****

**102\_Find a Fixed Point (Value equal to index) in a given array**

Time complexity: O( N ) Space complexity: O(1)

public int smallestEqual(int[] nums) {

        for(int i=0; i<nums.length; i++){

            if(i%10 == nums[i]){

                return i;

            }

        }

        return -1;

    }

**104\_square root of an integer**

Time complexity: O( log(N) ) Space complexity: O(1)

public int mySqrt(int x) {

        if(x==0) return 0;

        int low = 1,high = x,ans =0;

        while(low<=high){

            int mid =low + (high-low)/2;

            if(x/mid==mid) {

                return mid;

            }else if(x/mid<mid){

                high=mid-1;

            }

            else{  low = mid+1; ans = mid;  }

        }

        return ans;

}

**105\_Maximum and minimum of an array using minimum number of comparisons**

**114\_merge 2 sorted arrays**

Ap 1: using STL

1. Use built in sorting function to sort and return

Time complexity: O( (m+n)log(m+n) ) Space complexity: O(1)

Ap 2: using two pointers like merge sort (optimal)

1. Point i and j to two sizes of the arrays
2. Initialize another variable k to end of first array
3. While arr\_2 has elements
   * 1. Then we can start iterating from the end of the arrays i and j, and compare the elements at these positions. We will place the larger element in nums1 at position k, and decrement the corresponding pointer i or j accordingly.

Time complexity: O( m+n ) Space complexity: O(1)

public void merge(int[] nums1, int m, int[] nums2, int n) {

        int i=m-1;

        int j=n-1;

        int k=m+n-1;

        while(j>=0){

            if(i>=0 && (nums1[i]>nums2[j])){

                nums1[k--]=nums1[i--];

            }else{

                nums1[k--]=nums2[j--];

            }

        }

    }

**115\_print all subarrays with 0 sum**

**Ap 1 : nested loops**

1. Use a nested for loop to check for all subarrays sum == 0

Time complexity: O( N^2 ) Space complexity: O(1)

**Ap 2: Hashmap (optimal)**

The basic idea is to store the sum of the array while traversing the array. We store the sum of the elements traveled. Whenever we find a sum already present in a hashmap, we increase our count by the value stored in the hashmap.

For example:

If we have a subarray starting from index 0 and ending at index 2 has a sum of 10.

If there is another subarray starting from index 0 and ending at index 5 has a sum of 10.

Then the sum of elements from index 3 to index 5 should be 0.

In this way, we can find the number of subarrays having sum 0 by using the hashmap. For each sum found, we add it to our hashmap.

1. Create a variable (say, ‘COUNT’) to store the count of subarrays with 0 sum.
2. Create a hashmap (say, ‘MAP’) to store the sum count and initialize ‘MAP[0]’ to 1.
3. Create a variable (say, ‘localSum’) to store the sum of elements traveled so far and initialize it with 0.
4. Run a loop from 1 to ‘N’ (say, iterator ‘i’).
   1. Add ‘ARR[i]’ to ‘localSum’.
   2. Check if ‘localSum’ is present in ‘MAP’.
      1. Add ‘MAP[localSum]’ to ‘COUNT’.
   3. Add ‘localSum’ to ‘MAP’.
5. Return ‘COUNT’.

public static int countSubarrays(int n, int[] arr) {

        // To store the count.

        int count = 0;

        // To store the count sum.

        HashMap<Integer, Integer> map = new HashMap<>();

        map.put(0, 1);

        // To store the sum while traversing.

        int localSum = 0;

        // Find all subarrays.

        for (int i = 0; i < n; ++i) {

            // Update sum.

            localSum += arr[i];

            // Check if sum is already present.

            if (map.containsKey(localSum)) {

                // Update count.

                count += map.get(localSum);

            }

            // Update map.

            map.put(localSum, map.containsKey(localSum) ? map.get(localSum) + 1 : 1);        }

        return count;    }

Time complexity: O( N ) Space complexity: O( N )