Data Structures & Algo

Arrays

Arrays problem can be solved by many different approaches

30

1. Sliding window – (4) problem
2. Two pointer – (22) problem
3. fast & slow pointer – Used in Linked List (2)
   Array – merge 2 sorted array
4. Merge Interval –
5. cyclic sort –
6. Binary Search – (4) problem

1. Sliding window

Keyword: Sub-arrays, contiguous sub-array, Sub-lists, given range.

Q1. Given an array, find an Average of all contiguous Sub-array of size K.

Array: [1, 3, 2, 6, -1, 4, 1, 8, 2] K = 5

for 1st 5 number = (1+3+2+6+(-1))/5 = 2.2

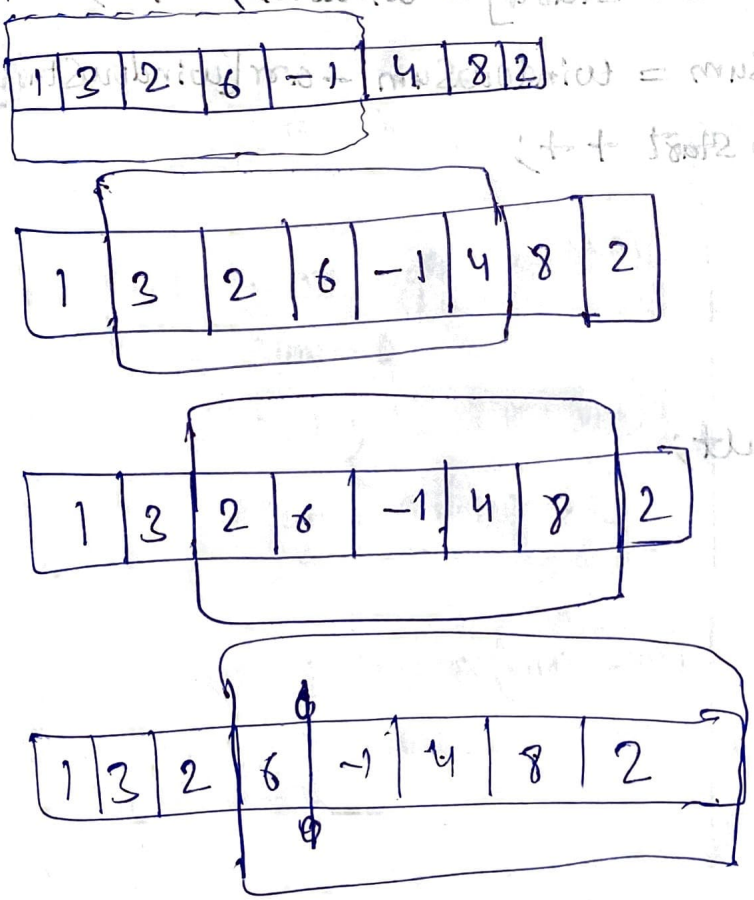
Next = 2.8

= 2.4

O/P: [2.2, 2.8, 2.4, 3.6, 2.8]

```
findAverage(int K, int[] arr) {
    double[] result = new double[arr.length-K+1];
    for(int i=0; i<arr.length-K; i++) {
        double sum=0
        for(j=i; j<i+K; j++){
            sum = sum + arr[j];
        result[i] = sum/K
    }
    return result
}
```

T.C = O(N*K)



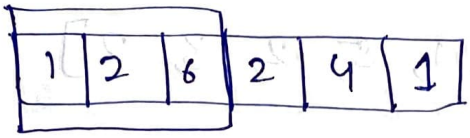
```
findAverage(int K, int[] arr) {
    double[] result = new double[arr.length - K + 1]
    double windowSum = 0
    int windowStart = 0
    for(int windowEnd = 0; windowEnd < arr.length;
                                windowEnd++) {
        windowSum = windowSum + arr[windowEnd];
        if(windowEnd >= K-1) {
        result[windowStart] = windowSum / K;
         windowSum = windowSum - arr[windowStart]
         windowStart++;
        }
    }
    return result;
}
```

Q2. find the maximum Sum of Continuous Sub-array of size K.

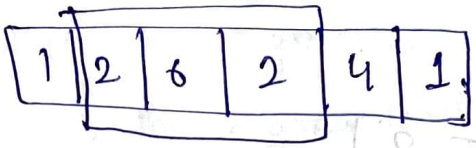
Array : [1, 2, 6, 2, 4, 1] K = 3

O/P : 12

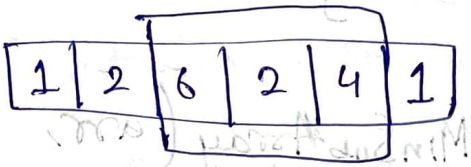
Explanation:



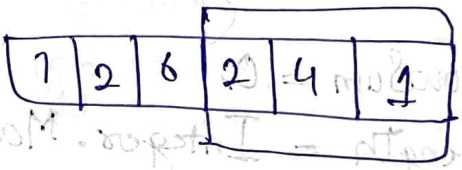
Sum = 9
maxSum = 9



Sum = 10
maxSum = 10



Sum = 12
maxSum = 12



Sum = 7
maxSum = 12

```
findMaxSum(arr, K){
    int max=0
    for(i=0; i<arr.length-K; i++){
        int sum=0
        for(j=i; j<i+K; j++){
            sum = sum+arr[j]
        }
        max= Math.max(max, sum)
    }
    return max;
}
```

T.C O(n * K)

```
findMaxSum(arr, K){
    int max=0,
        left = right=0;
        sum=0;
    while(right < arr.length){
        sum= sum + arr[right];
        if(right >= K){
            max= Math.max(max,
                          sum)
            sum= sum - arr[left];
            left++
        }
    }
    return max;
}
```

Q3. Smallest Sub-array with given Sum.

OR

Find the Smallest contiguous Subarray whose Sum is greater than or Equal or 's' else return zero

I/P : [2, 1, 5, 2, 3, 2]

S : 7

O/P : [5, 2]

```
function findMinSubArray(arr, s) {
    int windowSum = 0;
    int minLength = Integer.MAX_VALUE;
    int windowStart, windowEnd = 0;

    for(windowEnd = 0; windowEnd < arr.length;
                                windowEnd++) {
        windowSum = windowSum + arr[windowEnd];
        if(windowSum > s) {
            minLength = Math.min(minLength, windowEnd-
                                windowStart+1);
            windowSum = windowSum - arr[windowStart];
            windowStart++;
        }
    }
    return minLength == Integer.MaxValue ? 0 : minLength
}
```

Q4. find a Sub-array having the given Sum in an Array

I/P : [2, 6, 0, 9, 7, 3, 1, 4, 1, 10]

Sum : 15

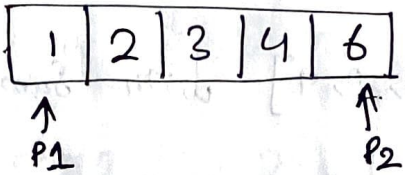
O/P : [6, 0, 9] with Sum 15

```
function findSubArray(array, sum) {
    int windowSum = 0, high = 0;
    for (int low = 0; low < array.length; low++) {
        while (windowSum < sum && high < array.length)
        {
            windowSum = windowSum + array[high]
            high++
        }
        if (windowSum == sum) {
            return [low, high - 1)
        }
        windowSum = windowSum - array[low];
    }
}
```

② Two-pointer

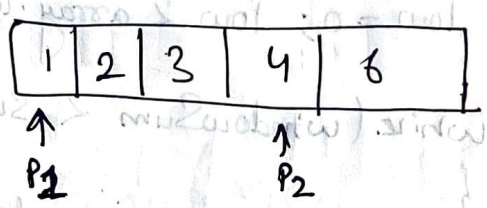
Keyword: Pair, triplet, Sub-array

target Sum = 6



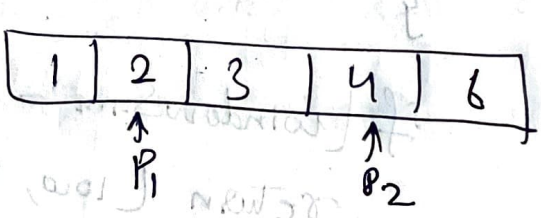
1 + 6 > 6

Decrement pointer P_2



1 + 4 < 6

Increment P1



2 + 4 = 6

found pair (2, 4)

Q1. find a pair in the array whose sum is Equal to the given target.

I/P : [1, 2, 3, 4, 6] target = 6

O/P : [1, 3] bcoz 2+4 = 6

```
function int[] Search(arr, target) {
    int left = 0, right = arr.length - 1;
    while(left < right) {
        int sum = arr[left] + arr[right];
        if(sum == target) {
            return new int[] {left, right};
        }
        else if(sum < target) {
            left++;
        } else {
            right--;
        }
        return new int[] {-1, -1};
    }
}
```

Note: If data is not sorted, Then we can use SET datastructure to optimize.

② Remove duplicate from sorted Array.

i/p: [1, 2, 2, 3, 3, 4, 4, 4, 5]

o/p: [1, 2, 3, 4, 5]

```
int j=0;
for(i=0; i<arr.length-1; i++){
    if(arr[i] != arr[i+1]){
        temp[j]=arr[i]
        j++;
    }
}
temp[j] = arr[n-1]
```

Note:- Here it is using Extra space which is temp array.

```
int j=0
for(i=0; i<n-1; i++){
    if(arr[i] != arr[i+1]){
        arr[j] = arr[i]
        j++
    }
}
arr[j] = arr[n-1]
```

③ Remove duplicate from un-sorted Array

4. 3 Sum (All triplet with zero sum)
5. 4 Sum
6. Merge String Alternately
7. Reverse String
8. Intersection of two Array
9. find duplicate number
10. Container with most water
11. Longest mountain in Array + 1 2
12. Rotate Array
13. Next Greater Element
14. Trapping rain water
15. Pair in Array whose Sum is closet to X.
16. Triplet that Sum to given Value.
17. find a triplet such that Sum of two Equal to third
18. ~~All triplets with zero sum.~~
19. Merge 2 Sorted Array.
20. Merge K Sorted Array.

(4) find all triplet with zero sum.

Bruteforce approach → 3 loops

T.C → O(n3)

~~Note: Array is sorted~~

Better approach is to use two pointers & sorting

```
findTriplets(arr, n) {
    Arrays.sort(arr);
    for(i=0; i<n-1; i++){
        int left = i+1;
        int right = n-1;
        int x = arr[i]
        while(left < right){
            if( x + arr[left] + arr[right] == 0){
                print(x, left, right)
                left++;
                right--;
            }
            else if (x + arr[left] + arr[right]<0){
                left++
            } else {
                right--
            }
        }
    }
}
```

⑤ 4 sum

⑥ Merge String alternately.

```
Public String mergeAlternately(String word1, String word2){
    StringBuilder sb = new StringBuilder();
    int p1 = 0, p2 = 0;
    while(p1 < word1.length() && p2 < word2.length()){
        sb.append(word1.charAt(p1++)
        sb.append(word2.charAt(p2++)
    }
    while(p1 < word1.length()) {
        sb.append(word1.charAt(p1++)
    }
    while(p2 < word2.length()){
        sb.append(word2.charAt(p2++)
    }
    return sb.toString();
}
```

⑦ Reverse String. or in-place

```
Public Static String Reverse(String s) {
    if(s == null || s.isEmpty()) {
        return s;
    }
    char[] characters = s.toCharArray();
    int i=0;
    int j = characters.length - 1;
    while(i < j) {
        Char temp = characters[i];
        characters[i] = characters[j];
        characters[j] = temp;
        i++;
        j--;
    }
}
```

swapping

⑧ Intersection of two - Array.

⑨ find Duplicate Element in an array, using O(n) time & O(1) Extra space.

OR

Given an array of n Element that contain an Element 0 to n-1, with any of those number appearing any number of times. find those repeating number in O(n) & using only constant space.

Soln

I/P : n = 7 & array = [1, 2, 3, 6, 3, 6, 1]

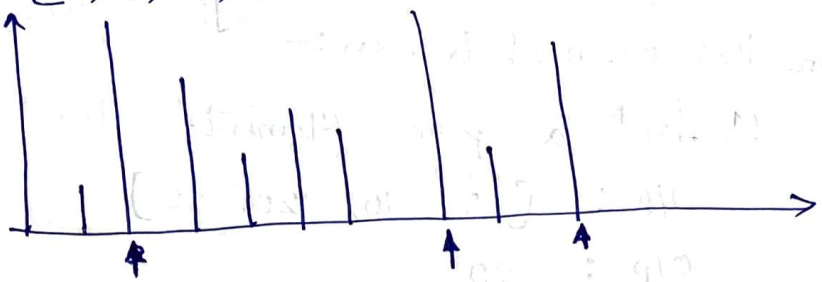
O/P : 1, 3, 6

Soln

```
function PrintRepeat(arr, size) {
    for(int i=0; i<size; i++){
        int j = Math.abs[arr[i]];
        if(arr[j] >= 0){
            arr[j] = -arr[j]
        } else {
            Print(j);
        }
    }
}
```

(10) Container with most water.

I/P : [1, 8, 6, 2, 5, 4, 8, 3, 7]



JS

```
var maxArea = function (H) {
    let ans = 0, i = 0, j = H.length - 1
    while (i < j) {
        ans = Math.max(ans, Math.min(H[i],
                H[j]) * (j - i))
        H[i] <= H[j] ? i++ : j--;
    }
    return ans
}
```

Java

```
maxArea(int[] H) {
    int ans=0, i=0, j= H.length -1, res = 0;
    while (i < j) {
        if( H[i] <= H[j]) {
            res = H[i] * (j-i);
            i++
        } else {
            res = H[j] * (j-i);
            j--
        }
        if (res > ans) ans = res;
    }
    return ans;
}
```

(11) Longest mountain in an array.

before this we need to learn :-

(1) find a peak element

i/p : [5, 10, 20, 15]

o/p : 20

Naive approach

Step 1: If 1st element is greater than second elem
OR
last element is greater than 2nd last
Print the element & terminate the pgm.

Step 2 : Else traverse the array from 2nd to second last.

Step 3 : If an element i is greater than both the neighbour than print & terminate.

Sol

```
findPeak(arr, n) {
    if (n==1) return 0;
    if (arr[0] >= arr[1]) return 0;
    if (arr[n-1] >= arr arr[n-2]) return n-1;
    for(i=1; i<n-1; i++) {
        if(arr[i] >= arr[i-1] &&
           arr[i] >= arr[i+1])
            return i;
    }
    return 0;
}
```

Best Approach : Divide & Conquer

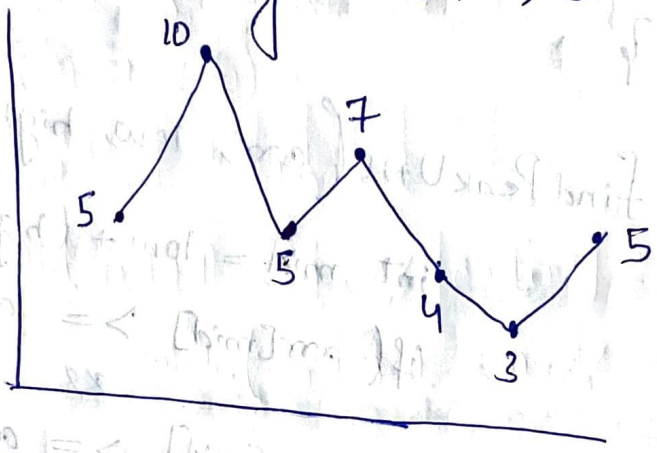
```
Class PeakElement {

    int findPeak(arr, n){
        return findPeakUtil(arr, 0, n-1, n);
    }

    findPeakUtil(arr, low, high, n) {
            int mid = low + (high-low)/2;
            if( arr[mid] >= arr[mid-1]
                        &&
            arr[mid] >= arr[mid + 1]){
                return mid;
            }
            else if( arr[mid] <= arr[mid-1]){
                return(findPeakUtil(arr, 0, mid-1,
                                    n)
            }
            else return findPeakUtil(arr, mid+1,
                                     high, n)

    }

}
```

② find All peaks & troughs in a Array.

i/p : [5, 10, 5, 7, 4, 3, 5]

o/p : Peak : 10, 7, 5

troughs : 5, 5, 3



Peak = 10, 7, 5

Troughs = 5, 5, 3

Soln

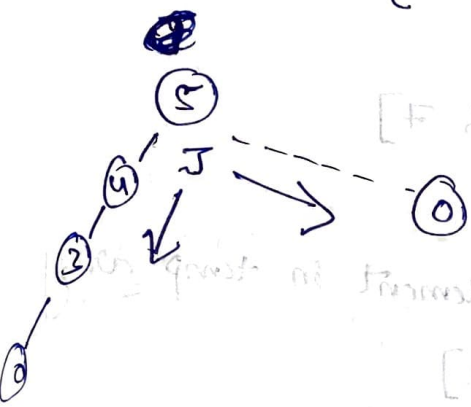
```
isPeak( arr, n, num, i, j) {
    if( i >=0 && arr[i] > num){
        return false
    }
    if( j<n && arr[j] > num){
        return false
    }
    return true
}
```

```
isTrough(arr, n, num, i, j) {
    if(i >=0 && arr[i] < num) {
        return false
    }
    if( j<n && arr[j] < num) {
        return false
    }
    return true;
}

PrintPeakTrough(arr, n) {
    System.out.Print("Peeks :")
    for(i=0 ; i<n ; i++) {
        if( isPeak(arr, n, arr[i], i-1, i+1))
        { Print arr[i] }
    }

    S.O.P ———— ("Trough :")
    for (i=0; i<n ; i++) {
        if( isTrough(arr, n, arr[i] , i-1, i+1))
        { Print arr[i] }
    }
}
```

Q: longestMountain

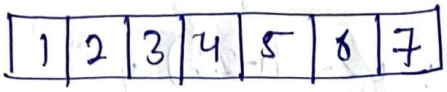
```
Public int Solution(arr) {

    int n = arr.length;
    int result = 0, start = 0;
    while(start < n) {

        int end = start;
        if( end+1 < n && arr[end] < arr[end+1]) {
            while( end + 1 < n && Arr[end] <
                                  arr[end+1]) {
                end++;
            }
        }

        if(end+1 < n && arr[end] > arr[end+1]) {
            while(end+1 < n && arr[end] > arr[end+1]){
                end++;
            }
        }

        result = Math.max(result, end- start+1);
    }
    return result
```

```
int HeighestMountain(arr) {
    int n = arr.size()
    int largest = 0;
    for(int i = 1; i <= n-2;) {
        if(arr[i] > arr[i-1] && arr[i] > arr[i+1])
        {
            // do some work
            int cnt = 1;
            int J = i;
            // count backward (left)
            while(J >= 1 && a[J] > a[J-1]) {
                J--
                cnt++
            }
            // count forward (right)
            while(i <= n-2 && a[i] > a[i+1]) {
                i++
                cnt++
            }
            Largest = max(largest, cnt);
        } else {
            i++;
        }
    }
    return largest
}
```

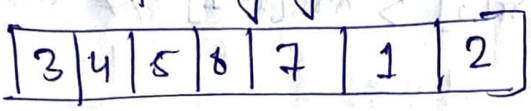


(12) Rotate Array

write a function rotate(arr, d, n) that rotate an arr[] of size n by d Elements.



Rotation of array by 2 will make



Method 1: Using temporary array

I/p : [1 2 3 4 5 6 7]

d : 2 n : 7

① Store the first d Element in temp array.

temp[] = [1 2]

② Shift remaining array

arr[] = [3 4 5 6 7]

③ Store back the temp array

arr[] = [3 4 5 6 7 1 2]

T/C : O(n) Space : O(d)

Method 2: Rotate by one

```
LeftRotateByOne(arr, n){
    int i, temp
    temp = arr[0]
    for(i=0; i<n-1; i++){
        arr[i] = arr[i+1];
    }
    arr[n-1] = temp;
}
```

```
LeftRotate(arr, d, n) {
    for(i=0; i<d; i++) {
        LeftRotateByOne(arr, n)
    }
}
```

Method 3: Reversal Algorithm

```
Rotate(arr[], d, n)
    reverse(arr, 1, d)
    reverse(arr, d+1, n)
    reverse(arr, 1, n)
```

Implementation

```
LeftRotate(arr, d) {
    n = arr.length
    reverseArray(arr, 0, d-1)
    reverseArray(arr, d, n-1)
    reverseArray(arr, 0, n-1)
}

reverseArray(arr, start, end) {
    int temp
    while(start < end) {
        temp = arr[start]
        arr[start] = arr[end]
        arr[end] = temp
        start++;
        end--;
    }
}
```

Method 4: two pointer logic

Cyclic Rotate an array by one

I/P : [1 2 3 4 5]

O/P : [5 1 2 3 4]

```
Rotate(arr, n) {
    int i=0, j= arr.length-1
    while(i != j){
    int temp = arr[i]
     arr[i] = arr[j]
     arr[j] = temp
       i++;
    }
}
```

(12.1) Search an Element in Sorted & Rotated Array.

Soln → find out pivot (point of rotation)

→ Divide into 2 halfves

→ call binary search on any 1 side.

→ return index of found Element

```
findPivot(arr, low, high) {
    int mid = (low+high)/2;
    if( mid < high && arr[mid] > arr[mid+1])
        return mid;
    if(mid > low && arr[mid] < arr[mid-1])
        return mid-1
```

```
    if( arr[low] >= arr[mid] ) {
        return findPivot(arr, low, mid-1)
    } else {
        return findPivot(arr, mid+1, high)
    }
}

BinarySearch(arr, low, high, key) {
    int mid = (low + high)/2
    if ( key == arr[mid]) {
        return mid
    }
    if ( key <= arr[mid]) {
        return BinarySearch(arr, low, mid-1, key)
    } else {
        return BinarySearch(arr, mid+1, high, key)
    }
}

PivotedBinarySearch(arr, n, key) {
    int pivot = findPivot(arr, 0, n-1);
    if (arr[pivot] == key) {
        return pivot
    }
    if(arr[0] <= key) {
        return BinarySearch(arr, 0, pivot-1, key)
    } else {
        return BinarySearch(arr, pivot+1, n, key)
    }
}
```

(12.2)

Find the Rotation Count.

→ Using Linear Search

No of rotation is Equal to index of minimum Element

T/C : O(n)

→ Binary Search

T/C : O(log n)

```
CountRotation(arr, low, high) {
    int mid = low + (high-low)/2
    if( mid < high && arr[mid] > arr[mid+1]){
            return mid+1
    }
    if(mid > low && arr[mid] < arr[mid-1]){
            return mid
    }
    if( arr[high] > arr[mid]) {
        return CountRotation(arr, low, mid-1)
    }
    else {
        return CountRotation(arr, mid+1, high)
    }
}
```

(12.3) find Minimum Element in a sorted & rotated array.

Soln use previous soln & return Element using the index pointed by mid.

(13) Next Greater Element

I/P: [4 5 2 25]

4 → 5
5 → 25
2 → -1
25 → -1

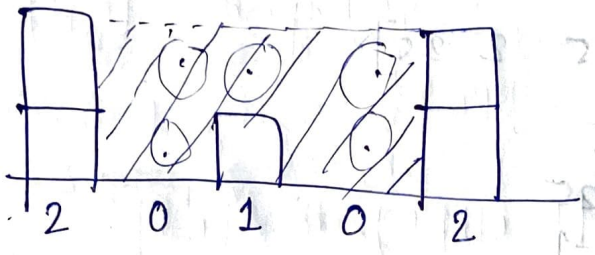
```
Print NGE(arr, n) {
    int next, i, j
    for(i=0; i<n; i++) {
        next = -1
        for(j= i+1; j<n; j++){
            if(arr[j] > arr[i]){
                next = arr[j]
                break;
            }
        }
        Print(arr[i] → next)
    }
}
```

Better solution is using STACK

(14) Trapping Rain water

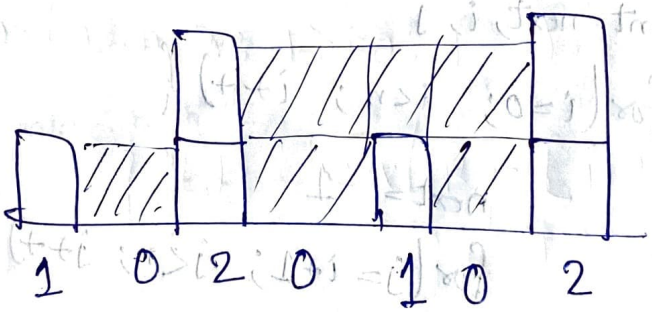
I/P : [2 0 1 0 2]

O/P : 5



I/P : [1 0 2 0 1 0 2]

O/P : 6



Solution Approach:

1. Using nested loop
2. Using Extra memory
3. Using Stack DS
4. Using 2 pointer approach.

① Using nested loop

```
RainWaterTrapping(height[], n) {
    int results = 0
    for(i = 0; i<n; i++){
        int left_maxHeight = 0
            right_maxHeight = 0
        for(k=i ; k>=0; k--){
            left_maxHeight = max(height[k],
                                 left_maxHeight)
        }
        for(j=i ; j<n ; j++){
            right_maxHeight = max(height[k],
                                  right_maxHeight)
        }
        results = results + min(left_max
                  Height , right_maxHeight]
                  - height[i]
    }
    return results
}
```

T/C : $O(n^2)$

S/C : $O(1)$

② Using Extra Memory

```
RainWaterTrapping(height, n) {
    int left[n], Right[n]
    left[0] = height[0]
    for(i=1 ; i<n ; i++){
        left[i] = max( left[i-1], height[i])
    }

    Right[n-1] = height[n-1]
    for(j=n-2; j>=0; j--){
        Right[j] = max( Right[j+1], height[j])
    }

    int result = 0
    for(i=0; i<n; i++){
        result = result + min( left[i], Right[i])
                        - height[i]
    }

    return result
}
```

T/C : O(n)

S/C : O(n)

(4) Two pointer approach

```
RainWaterTrapping(height[], n) {
    int results = 0
    int LeftMaxH = 0, RightMaxH = 0
    int left = 0, right = n-1
    while(left <= right) {
        if(height[left] < height[right]) {
            if(height[left] > leftMaxH) {
                leftMaxH = height[left]
            } else {
                results = results +
                    leftMaxH - height[left]
                left = left + 1
            }
        } else {
            if(height[right] > rightMaxH) {
                rightMaxH = height[right]
            } else {
                results = results +
                    rightMaxH - height[right]
                right = right - 1
            }
        }
    return results
}
```

(15) Pair in sorted Array whose Sum is closet to x.

```
ClosetPair(arr, size, x) {
    int resL = 0, resR = 0
    int l = 0, r = size - 1
    int diff = Integer.MaxValue()
    while(r > l) {
        if(Math.abs(arr[l] + arr[r] - x) < diff) {
            diff = Math.abs(arr[l] + arr[r] - x)
            resL = l
            resR = r
        }
        if(arr[l] + arr[r] > x) {
            r--
        }else {
            l++
        }
    }
    return(resL, resR)
}
```

(16) Triplet That Sum to given Value.

approach 1 : 3 loop

T/C = $O(n^3)$

S/C = $O(1)$

approach 2 : → fixed 1st Element

→ apply two pointer for remaing two Element.

Note: array need to be sorted before.

```
find Triplet(arr, size, sum) {

    arr.sort();
    for(i=0; i<size-2; i++){

        l = i+1
        r = size-1
        while(l < r){
            if(arr[l]+arr[r]+arr[i] == sum)
            {
                print(i,l,r)
                return(l, r, i)
            }
            else if(l+r+i < sum){
                l++
            else
                r--
            }
        }
    }
}
```

(17) Find triplet such that sum of two equal to third.

Soln Approach 1: Brute force

use 3 loops

```
for(i=0 to size-2) {
    for(j=i+1 to size-1) {
        for(k=j+1 to size) {
            if(i+j+k == sum){
                return i,j,k
            }
        }
    }
}
```

T/C : $O(n^3)$

Approach 2: use sorting & two pointer

```
data.sort()
for(i=0 to size-2) {
    left = i+1
    right = size-1
    while(left < right) {
        if(i+left+right == sum) {
            return i,left,right
        }
        else if(i+left+right < sum){
            left++
        }
        else {
            right--
        }
    }
}
```

T/C : $O(n^2)$

⑥ Binary Search

Whenever we are given a sorted Array, LinkedList or Matrix & we are asked to find a certain Element. the best solution is Binary Search.

① Recursive way:

```
BinarySearch(arr, L, R, x) {
    if(R >= L) {
        int mid = L + (R-L)/2
        if(arr[mid] == x){
            return mid
        }
        if(arr[mid] > x) {
            return BinarySearch(arr, L, mid-1, x)
        }
        else {
            return BinarySearch(arr, mid+1, r, x)
        }
    }
    return -1
}
```

② Iterative way:

```
BinarySearch (arr, L, r, x){
    while(L <= r){
        int mid = L + (r-L)/2
        if (arr[mid] == x)  return mid
        if (arr[mid] < x)   L = mid + 1
        else   r = mid - 1
    }
    return -1
}
```

(1) Write a function to return the index of the key if it is present in the Array otherwise return -1.

Note: - i) Array is sorted
ii) don't know if it is sorted Ascending/descen...

I/P : [4, 6, 10] key = 10

O/P : 2

```
Search(arr, key) {
    int start = 0, end = arr.length - 1
    boolean isAscending = arr[start] < arr[end]
    while(start <= end) {
        int mid = start + (end - start) / 2;
        if(key == arr[mid]) {
            return mid
        }
        if(isAscending) {
            if(key < arr[mid]) {
                end = mid - 1
            } else {
                start = mid + 1
            }
        }
```

```
        else if( ! isAscending) {
            if( key > arr[mid]) {
                end = mid - 1
            }else {
                start = mid + 1
            }
        }
    }
    return -1
}
```

(2) write a function to return the index of the ceiling of the key. OR minimum difference with the given key

I/P : [4, 6, 10] key = 6

O/P : 1

I/P : [1, 3, 8, 10, 15] key = 12

O/P : 4

```
CeilingOfNumber(arr, key) {
    if(key > arr[arr.length-1]) {
        return -1
    }
    int start = 0, end = arr.length - 1;
    while (start < end) {
        int mid = start + (end - start) / 2;
        if( key < arr[mid]) {
            end = mid - 1
        }else if( key > arr[mid]) {
            start = mid + 1
        }else {
            return mid;
```

```
    }
    return start
    // start == end+1
}
```

(3) Write a function to return the index of the floor of the key, if not return -1.

```
Searchfloor(arr, key) {
    if(key < arr[0]) {
        return -1
    }
    [ Binary search logic

    return end
    // start == end+1
}
```

(4) Write a function to return range of key:

I/P: [4, 6, 6, 6, 9] key = 6

O/P = [1, 3]

```
findRange(arr, key) {
    int result = new int[] {-1, -1}
    result[0] = Search(arr, key, false);
    if(result[0] != -1) {
        result[1] = Search(arr, key, true);
    }
    return result;
```

```
Search( arr, key, findmaxindex){
    int keyindex = -1;
    int start = 0, end = arr.length-1;
    while( start <= end){
        int mid = start + (end-start)/2
        if( key < arr[mid]){
            end = mid-1
        }else if( key > arr[mid]){
            start = mid+1
        }else{
            keyindex = mid;
            if( findmaxindex){
                start = mid+1
            }else{
                end = mid-1
            }
        }
    }
    return keyindex;
}
```