Geeks for Geeks DS Questions:

Arrays:-

Array Rearrangement

- **Rearrange an array such that arr[i] = i**

Given an array of elements of length N, ranging from 0 to N – 1. All elements may not be present in the array. If the element is not present then there will be -1 present in the array. Rearrange the array such that A[i] = i and if i is not present, display -1 at that place.

Solution:

Approach 1 :- (Using HashSet) :

1) Store all the numbers present in the array into a HashSet
2) Iterate through the length of the array, if the corresponding position element is present in the HashSet, then set A[i] = i, else A[i] = -1

Time Complexity: O(n), Space Complexity: O(n)

Another Approach (Swap elements in Array) :

1) Iterate through elements in an array
2) If arr[i] >= 0 && arr[i] != i, put arr[i] at i (swap arr[i] with arr[arr[i]])

Time Complexity :- O(n) , Space Complexity: O(1)

- **Write a program to reverse an array or string**

1) *Initialize start and end indexes as start = 0, end = n-1*
2) *In a loop, swap arr[start] with arr[end] and change start and end as follows :*
   start = start +1, end = end – 1

- **Rearrange array such that arr[i] >= arr[j] if i is even and arr[i]<=arr[j] if i is odd and j < i**

Given an array of n elements. Our task is to write a program to rearrange the array such that elements at even positions are greater than all elements after it and elements at odd positions are greater than all elements before it.

```
Input : arr[] = {1, 2, 3, 4, 5, 6, 7}
Output : 4 5 3 6 2 7 1
```

```
Input : arr[] = {1, 2, 1, 4, 5, 6, 8, 8}
Output : 4 5 2 6 1 8 1 8
```

Solution :-

The basic idea is to create a copy of the input array, sort it and copy the numbers alternatively from sorted array to original array.

Total number of positions will be n/2 even and n – n/2 odd positions.

1) Copy the array in a new array -> Sort it

2) Start from middle element -1 and go towards left in sorted array and keep copying the element (+2) alternatively 1 chhod ke… and same process karo next half ke jo greater elements hai unke liye right move karke sorted array mai 1 chhod ke even positions mai daal do..

O(nlogn), O(n)

Code given below :-

```
public class GfG{
    // function to rearrange the array
    public static void rearrangeArr(int arr[],
                                               int n)
    {
        // total even positions
        int evenPos = n / 2;

        // total odd positions
        int oddPos = n - evenPos;

        int[] tempArr = new int [n];

        // copy original array in an
        // auxiliary array
        for (int i = 0; i < n; i++)
            tempArr[i] = arr[i];

        // sort the auxiliary array
        Arrays.sort(tempArr);

        int j = oddPos - 1;

        // fill up odd position in
        // original array
        for (int i = 0; i < n; i += 2) {
            arr[i] = tempArr[j];
            j--;
        }

        j = oddPos;

        // fill up even positions in
        // original array
        for (int i = 1; i < n; i += 2) {
            arr[i] = tempArr[j];
            j++;
        }

        // display array
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
    }

    // Driver function
    public static void main(String argc[]){
        int[] arr = new int []{ 1, 2, 3, 4, 5,
                                        6, 7 };
        int size = 7;
        rearrangeArr(arr, size); }}
```

Rearrange positive and negative numbers in O(n) time and O(1) extra space (Not maintain order of element)

An array contains both positive and negative numbers in random order. Rearrange the array elements so that positive and negative numbers are placed alternatively. Number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear in the end of the array.

For example, if the input array is [-1, 2, -3, 4, 5, 6, -7, 8, 9], then the output should be [9, -7, 8, -3, 5, -1, 2, 4, 6]

Note: The partition process changes relative order of elements. I.e., the order of the appearance of elements is not maintained with this approach. See this for maintaining order of appearance of elements in this problem.

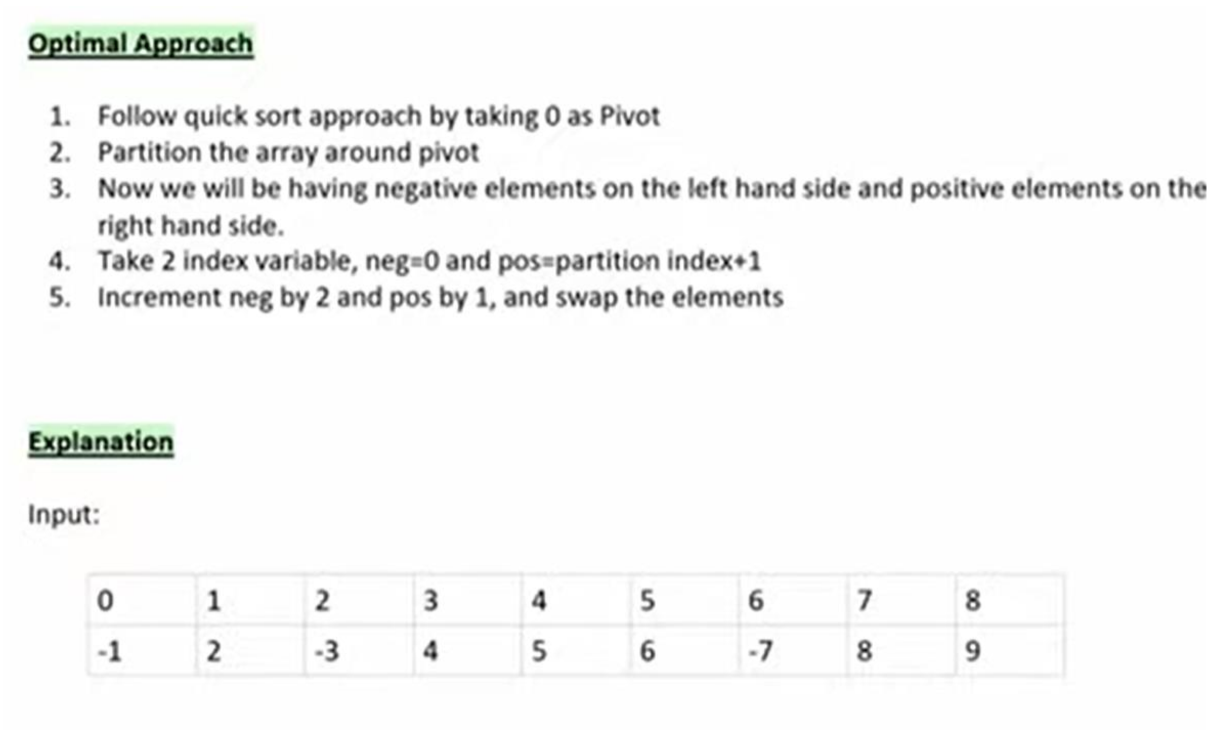
Solution :-

Approach 1 : To solve in O(nlogn) & Space O(n):-

- Sort the Array
- After sorting, use one pointer at beginning of array and one at the end of the array and keep storing in the new output array alternatively until start != high.

Approach 2 :- To solve in O(n) & O(1) :-

The solution is to first separate positive and negative numbers using partition process of QuickSort. In the partition process, consider 0 as value of pivot element so that all negative numbers are placed before positive numbers. Once negative and positive numbers are separated, we start from the first negative number and first positive number, and swap every alternate negative number with next positive number.



Optimal Approach

1. Follow quick sort approach by taking 0 as Pivot
2. Partition the array around pivot
3. Now we will be having negative elements on the left hand side and positive elements on the right hand side.
4. Take 2 index variable, neg=0 and pos=partition index+1
5. Increment neg by 2 and pos by 1, and swap the elements

Explanation

Input:

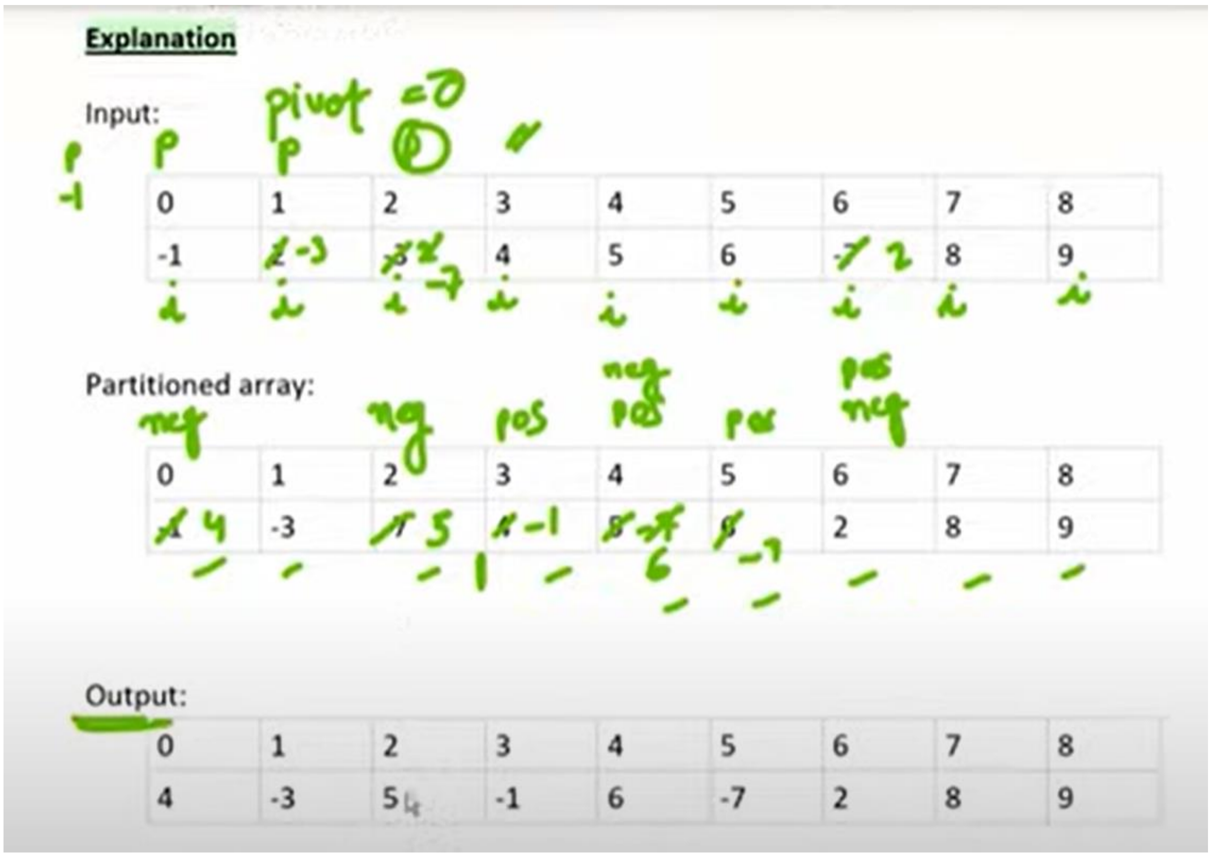
0	1	2	3	4	5	6	7	8
-1	2	-3	4	5	6	-7	8	9

Partitioned Array would be :

-1 , -3, -7, 4, 5, 6, 2, 8, 9

Logic :

- Initialize Pivot element at index -1.
- Iterate through the existing array from I = 0 to i<n , and for each element check if its <0 i.e, negative.
- If element is negative, Increment the Pivot element index and then swap it with ith element.
  After this all process, you will have a partitioned array, where elements on the left side will be negative, and elements on the right side will be all positive.
- Once you have the partitioned array, you can start from index 0 and 'index of Pivot currently + 1', and increment 0th index i.e., negative index value by 2 and positive index value by 1, and keep swapping each other until end of the array.



Explanation

Input:

0	1	2	3	4	5	6	7	8
-1	~~2~~ -3	~~-3~~ ~~2~~ -7	4	5	6	~~-7~~ 2	8	9

Partitioned array:

0	1	2	3	4	5	6	7	8
~~-1~~ 4	-3	~~-7~~ 5	~~4~~ -1	~~5~~ ~~-7~~ 6	~~6~~ -7	2	8	9

Output:

0	1	2	3	4	5	6	7	8
4	-3	5	-1	6	-7	2	8	9

```
class Alternate {

    // The main function that rearranges elements of given
    // array.  It puts positive elements at even indexes (0,
    // 2, ..) and negative numbers at odd indexes (1, 3, ..).
    static void rearrange(int arr[], int n)
    {

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

        // Now all positive numbers are at end and negative
numbers at
        // the beginning of array. Initialize indexes for
starting point
        // of positive and negative numbers to be swapped
        int pos = i+1, neg = 0;

        // Increment the negative index by 2 and positive
index by 1, i.e.,
        // swap every alternate negative number with next
positive number

 while (pos < n && neg < pos && arr[neg] < 0)
        {
            temp = arr[neg];
            arr[neg] = arr[pos];
            arr[pos] = temp;
            pos++;
            neg += 2;
        }
    }

    // A utility function to print an array
    static void printArray(int arr[], int n)
    {
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + "    ");
    }

    /*Driver function to check for above functions*/
    public static void main (String[] args)
    {
        int arr[] = {-1, 2, -3, 4, 5, 6, -7, 8, 9};
        int n = arr.length;
        rearrange(arr,n);
        System.out.println("Array after rearranging: ");
        printArray(arr,n);
    }
}
```

Move all zeroes to end of array :

```
Input :  arr[] = {1, 2, 0, 4, 3, 0, 5, 0};
Output : arr[] = {1, 2, 4, 3, 5, 0, 0};
```

Approach 1:- O(n) and O(1)

Traverse the given array ‘arr’ from left to right. While traversing, maintain count of non-zero elements in array. Let the count be ‘count’. For every non-zero element arr[i], put the element at ‘arr[count]’ and increment ‘count’. After complete traversal, all non- zero elements have already been shifted to front end and ‘count’ is set as index of first 0. Now all we need to do is that run a loop which makes all elements zero from ‘count’ till end of the array.

```
static void pushZerosToEnd(int arr[], int n)

{

        int count = 0; // Count of non-zero elements


        // Traverse the array. If element encountered is

        // non-zero, then replace the element at index 'count'

        // with this element

        for (int i = 0; i < n; i++)

                if (arr[i] != 0)

                        arr[count++] = arr[i]; // here count is

                                                            // incremented


        // Now all non-zero elements have been shifted to

        // front and 'count' is set as index of first 0.

        // Make all elements 0 from count to end.

        while (count < n)

                arr[count++] = 0;

}
```

Approach 2:- O(n) and O(1)

Start from i = 0 and j = n-1;

Iterate from i = 0 till it meets j

```
    Check if i != 0 then increment i
    Check if j == 0 then decrement j
    Check if i == 0 then swap arr[i] and arr[j] and decrement
j.
```

i = 0, j = n-1;

```
while (i!=j){
        if(a[i] != 0) i++;
        if(a[j]==0) j--;
        if(a[i]==0 && a[j] !=0)
                swap(a[i],a[j])
                i++; j--;
}
```

Approach 3 :- (Only single traversal would be required) O(n)

```
moveZerosToEnd(arr, n)
    Initialize count = 0
    for i = 0 to n-1
        if (arr[i] != 0) then
            swap(arr[count++], arr[i])
```

Rearrange array such that even positioned are greater than odd

Given an array A of n elements, sort the array according to the following relations :

```
Input : A[] = {1, 2, 2, 1}
Output :  1 2 1 2 Ex., Position of digit 1 is 0th, and positit ion of 2 is 1st
Explanation :
For 1st element, 1  1, i = 2 is even.
3rd element, 1  1, i = 4 is even.
```

Method 1 –
Observe that array consists of [n/2] even positioned elements. If we assign

the largest [n/2] elements to the even positions and the rest of the elements to the odd positions, our problem is solved. Because element at the odd position will always be less than the element at the even position as it is the maximum element and vice versa. Sort the array and assign the first [n/2] elements at even positions.
Below is the implementation of the above approach:

```
class GFG {

    static void assign(int a[], int n)

    {

        // Sort the array

        Arrays.sort(a);

        int ans[] = new int[n];

        int p = 0, q = n - 1;

        for (int i = 0; i < n; i++) {

            // Assign even indexes with maximum elements

            if ((i + 1) % 2 == 0)

                ans[i] = a[q--];

            // Assign odd indexes with remaining elements

            else

                ans[i] = a[p++];

        }

        // Print result

        for (int i = 0; i < n; i++)

            System.out.print(ans[i] + " ");
    }

    // Driver code

   public static void main(String args[])

    {

        int A[] = { 1, 3, 2, 2, 5 };

        int n = A.length;

        assign(A, n);

    }

}
```

Output:

```
 1 5 2 3 2
```

Time Complexity: O(n * log n)

Auxiliary Space: O(n)

Method 2 – O(n) O(1)

One other approach is to traverse the array from the second element and swap the element with the previous one if the condition is not satisfied. This is implemented as follows:

```
public static void rearrange(int[] arr, int n)
  {
    for(int i = 1; i < n; i++)
    {

      // if index is even
      if(i % 2 == 0)
      {
        if(arr[i] > arr[i - 1])
        {

          // swap two elements
          int temp = arr[i];
          arr[i] = arr[i - 1];
          arr[i - 1] = temp;
        }
      }

      // if index is odd
      else
      {
        if (arr[i] < arr[i - 1])
        {

          // swap two elements
          int temp = arr[i];
          arr[i] = arr[i - 1];
          arr[i - 1] = temp;
        }
      }
    }

    for (int i = 0; i < n; i++)
    {
      System.out.print(arr[i] + " ");
    }
  }
```

Rearrange an array in order – smallest, largest, 2nd smallest, 2nd largest

An **efficient solution** is to use sorting.
1. Sort the elements of array.
2. Take two variables say i and j and point them to the first and last index of the array respectively.
3. Now run a loop and store the elements in the array one by one by incrementing i and decrementing j.

Time Complexity : O(n Log n)
Auxiliary Space : O(n)

Double the first element and move zero to end

Method 1:-

For a given array of n integers and assume that ‘0’ as an invalid number and all other as a valid number. Convert the array in such a way that if both current and next element is valid then double current value and replace the next number with 0. After the modification, rearrange the array such that all

0’s shifted to the end.

Examples:

```
Input : arr[] = {2, 2, 0, 4, 0, 8}

Output : 4 4 8 0 0 0


Input : arr[] = {0, 2, 2, 2, 0, 6, 6, 0, 0, 8}

Output :  4 2 12 8 0 0 0 0 0 0
```

```
static void modifyAndRearrangeArr(int arr[], int n)
    {
        // if 'arr[]' contains a single element
        // only
        if (n == 1)
            return;

        // traverse the array
        for (int i = 0; i < n - 1; i++) {

            // if true, perform the required modification
            if ((arr[i] != 0) && (arr[i] == arr[i + 1]))
            {

                // double current index value
                arr[i] = 2 * arr[i];

                // put 0 in the next index
                arr[i + 1] = 0;

                // increment by 1 so as to move two
                // indexes ahead during loop iteration
                i++;
            }
        }

        // push all the zeros at
        // the end of 'arr[]'
        pushZerosToEnd(arr, n); // Refer Previous code in prev pages…
    }
```

Arrange given numbers to form the biggest number

In the used sorting algorithm, instead of using the default comparison, write a comparison function **myCompare()** and use it to sort numbers.

Given two numbers **X** and **Y**, how should **myCompare()** decide which number to put first – we compare two numbers XY (Y appended at the end of X) and YX (X appended at the end of Y). If **XY** is larger, then X should come before Y in output, else Y should come before. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.

```
static void printLargest(Vector<String> arr)
    {

        Collections.sort(arr, new Comparator<String>()
        {
            // A comparison function which is used by
            // sort() in printLargest()
            @Override public int compare(String X, String Y)
            {

                // first append Y at the end of X
                String XY = X + Y;

                // then append X at the end of Y
                String YX = Y + X;

                // Now see which of the two
                // formed numbers
                // is greater
                return XY.compareTo(YX) > 0 ? -1 : 1;
            }
        });

        Iterator it = arr.iterator();

        while (it.hasNext())
            System.out.print(it.next());
    }
```

O(nlogn), O(1)

Rearrange an array in maximum minimum form | Set 2 (O(1) extra space)

In this post a solution that requires O(n) time and O(1) extra space is discussed. The idea is to use multiplication and modular trick to store two elements at an index.

A simpler approach will be to observe indexing positioning of maximum elements and minimum elements. The even index stores maximum elements and the odd index stores the minimum elements. With every increasing index, the maximum element decreases by one and the minimum element increases by one. A simple traversal can be done and arr[] can be filled in again.
Note: This approach is only valid when elements of given sorted array are consecutive i.e., vary by one unit.
Below is the implementation of the above approach:

```
public static void rearrange(int arr[], int n)
    {
        // initialize index of first minimum and first
        // maximum element
        int max_ele = arr[n - 1];
        int min_ele = arr[0];
        // traverse array elements
        for (int i = 0; i < n; i++) {
            // at even index : we have to put maximum element
            if (i % 2 == 0) {
                arr[i] = max_ele;
                max_ele -= 1;
            }

            // at odd index : we have to put minimum element
            else {
                arr[i] = min_ele;
                min_ele += 1;
            }
        }
    }
```

Move all negative numbers to beginning and positive to end with constant extra space

Two Pointer Approach: The idea is to solve this problem with constant space and linear time is by using a two-pointer or two-variable approach where we simply take two variables like left and right which hold the 0 and N-1 indexes. Just need to check that :

1. Check If the left and right pointer elements are negative then simply increment the left pointer.
2. Otherwise, if the left element is positive and the right element is negative then simply swap the elements, and Simultaneously increment or decrement the left and right pointers.
3. Else if the left element is positive and the right element is also positive then simply decrement the right pointer.

4. Repeat the above 3 steps until the left pointer ≤ right pointer.

O(n) O(1)

Rearrange array such that even index elements are smaller and odd index elements are greater

An **efficient solution** is to iterate over the array and swap the elements as per the given condition.
If we have an array of length n, then we iterate from index 0 to n-2 and check the given condition.
At any point of time if i is even and arr[i] > arr[i+1], then we swap arr[i] and arr[i+1]. Similarly, if i is odd and
arr[i] < arr[i+1], then we swap arr[i] and arr[i+1].

O(n) and O(1)

Replace every array element by multiplication of previous and next

Given an array of integers, update every element with multiplication of previous and next elements with following exceptions.
a) First element is replaced by multiplication of first and second.
b) Last element is replaced by multiplication of last and second last.

Example:

```
Input: arr[] = {2, 3, 4, 5, 6}

Output: arr[] = {6, 8, 15, 24, 30}
```

Approach 1: To create an auxiliary array, copy contents of given array to auxiliary array. Finally traverse the auxiliary array and update given array using copied values. Time complexity of this solution is O(n), but it requires O(n) extra space.

Approach 2: To Keep track of previous element - An efficient solution can solve the problem in O(n) time and O(1) space. The idea is to keep track of previous element in loop.

Use two variables to keep track of previous element and to track the current element.

```
 static void modify(int arr[], int n)
    {
        // Nothing to do when array size is 1
        if (n <= 1)
            return;

        // store current value of arr[0] and update it
        int prev = arr[0];
        arr[0] = arr[0] * arr[1];

        // Update rest of the array elements
        for (int i=1; i<n-1; i++)
        {
            // Store current value of next interation
            int curr = arr[i];

            // Update current value using previos value
            arr[i] = prev * arr[i+1];

            // Update previous value
            prev = curr;
        }

        // Update last array element
        arr[n-1] = prev * arr[n-1];
    }
```

Segregate even and odd numbers | Set 3

Given an array of integers, segregate even and odd numbers in the array. All the even numbers should be present first, and then the odd numbers.

Solution : We can solve it using 2 pointer approach where one pointer will be pointing to the first element and other pointer will be pointing to the last of the element.

1. Maintain a pointer i = 0 and j = n-1;
2. Traverse the array until i != j, if even number is encountered on left i.e., a[i] then increment i.
3. Check for a[j] as well, if a[j] is odd then decrement j.

4. If a[i] is odd then don't increment, wait for a[j] to be even after decrement nd then swap with it..so main idea should be that the even numbers are in the beginning and odd numbers at the end.
5. Continue the traversal.

Time Complexity : *O(n)*
Auxiliary Space : *O(1)*

Positive elements at even and negative at odd positions (Relative order not maintained)

```
Input : arr[] = {1, -3, 5, 6, -3, 6, 7, -4, 9, 10}
Output : arr[] = {1, -3, 5, -3, 6, 6, 7, -4, 9, 10}
```

We take two pointers positive and negative. We set the positive pointer at start of the array and the negative pointer at 1st position of the array.
We move positive pointer two steps forward till it finds a negative element.
Similarly we move negative pointer forward by two places till it finds a positive value at its position.
If the positive and negative pointers are in the array then we will swap the values at these indexes otherwise we will stop executing the process.

Maximum sum of pairs with specific difference

O(nlogn)

Given an array of integers and a number k. We can pair two number of the array if the difference between them is strictly less than k. The task is to find the maximum possible sum of disjoint pairs. Sum of P pairs is the sum of all 2P numbers of pairs.

First we sort the given array in increasing order. Once array is sorted, we traverse the array. For every element, we try to pair it with its previous element first. Why do we prefer previous element? Let arr[i] can be paired with arr[i-1] and arr[i-2] (i.e. arr[i] – arr[i-1] < K and arr[i]-arr[i-2] < K). Since

the array is sorted, value of arr[i-1] would be more than arr[i-2]. Also, we need to pair with difference less than k, it means if arr[i-2] can be paired, then arr[i-1] can also be paired in a sorted array.
Now observing the above facts, we can formulate our dynamic programming solution as below,

```
class GFG {

    // Method to return maximum sum we can get by
    // finding less than K difference pairs
    static int maxSumPairWithDifferenceLessThanK(int arr[],
                                                 int N,
                                                 int k)
    {
        int maxSum = 0;

        // Sort elements to ensure every i and i-1 is
        // closest possible pair

        Arrays.sort(arr);

        // To get maximum possible sum,
        // iterate from largest
        // to smallest, giving larger
        // numbers priority over
        // smaller numbers.

        for (int i = N - 1; i > 0; --i)
        {
            // Case I: Diff of arr[i] and arr[i-1] is less
            // then K, add to maxSum
            // Case II: Diff between arr[i] and arr[i-1] is
            // not less then K, move to next i
            // since with sorting we know, arr[i]-arr[i-1] <
            // arr[i]-arr[i-2] and so on.

            if (arr[i] - arr[i - 1] < k)
            {
                // Assuming only positive numbers.
                maxSum += arr[i];
                maxSum += arr[i - 1];

                // When a match is found skip this pair
                --i;
            }
        }

        return maxSum;
    }

    // Driver code
    public static void main(String[] args)
    {

        int arr[] = { 3, 5, 10, 15, 17, 12, 9 };
        int N = arr.length;
        int K = 4;
        System.out.println(
            maxSumPairWithDifferenceLessThanK(arr, N, K));
    }
}
```

Replace every element with the greatest element on right side

A **tricky method** is to replace all elements using one traversal of the array. The idea is to start from the rightmost element, move to the left side one by one, and keep track of the maximum element. Replace every element with the maximum element.

O(n)

```
    static void nextGreatest(int arr[])
    {
        int size = arr.length;
        int max_from_right =  arr[size-1];

        arr[size-1] = -1;

        for (int i = size-2; i >= 0; i--)
        {
            int temp = arr[i];

            arr[i] = max_from_right;

            if(max_from_right < temp)
            max_from_right = temp;
        }
    }
```

Good Questions below for array rearrangement :-

Merge k sorted arrays

Input:
k = 3, n = 4
arr[][] = { {1, 3, 5, 7},
{2, 4, 6, 8},
{0, 9, 10, 11}} ;
Output: *0 1 2 3 4 5 6 7 8 9 10 11*

Ref : https://medium.com/outco/how-to-merge-k-sorted-arrays-c35d87aa298e

Ideally, if we have 2 arrays, we just need to compare 2 elements and store in the output array, but if we have k arrays, the comparison will be complex.

Other way is to concat all the elements of k arrays, and then sort them which will take the complexity of nk*log(n*k).

Basically, on each iteration, if we can find out the minimum element out of k arrays and store in output we can solve this kind of problem easily, here comes the concept of minheap, where can get the minimum element at root and the complexity will be O(1).

Check whether two strings are anagram of each other

Use only one count array instead of two. We can increment the value in count array for characters in str1 and decrement for characters in str2. Finally, if all count values are 0, then the two strings are anagram of each other

```
static boolean areAnagram(char[] str1,
                          char[] str2)
{

    // Create a count array and initialize
    // all values as 0
    int[] count = new int[NO_OF_CHARS];
    int i;

    // For each character in input strings,
    // increment count in the corresponding
    // count array
    for(i = 0; i < str1.length; i++)
    {
        count[str1[i] - 'a']++;
        count[str2[i] - 'a']--;
    }

    // If both strings are of different
    // length. Removing this condition
    // will make the program fail for
    // strings like "aaca" and "aca"
    if (str1.length != str2.length)
        return false;

    // See if there is any non-zero
    // value in count array
    for(i = 0; i < NO_OF_CHARS; i++)
        if (count[i] != 0)
        {
            return false;
        }
    return true;
}
```

Group anagrams together :-

The idea here is to have a Hashamap<HashMap<Character, Integer>, ArrayList<String>>, where inner hashamp will maintain a count of character and their count, and corresponding list in value.
Ref :- Pepcoding Link

```
private static ArrayList<ArrayList<String> >
    solver(
        ArrayList<String> list)
    {

        // Inner hashmap counts frequency
        // of characters in a string.
        // Outer hashmap for if same
        // frequency characters are present in
        // in a string then it will add it to
        // the arraylist.
        HashMap<HashMap<Character, Integer>,
                ArrayList<String> >
            map = new HashMap<HashMap<Character, Integer>,
                              ArrayList<String> >();
        for (String str : list) {
            HashMap<Character, Integer>
                tempMap = new HashMap<Character, Integer>();

            // Counting the frequency of the
            // characters present in a string
            for (int i = 0; i < str.length(); i++) {
                if (tempMap.containsKey(str.charAt(i))) {
                    int x = tempMap.get(str.charAt(i));
                    tempMap.put(str.charAt(i), ++x);
                }
                else {
                    tempMap.put(str.charAt(i), 1);
                }
            }

            // If the same frequency of chanracters
            // are alraedy present then add that
            // string into that arraylist otherwise
            // created a new arraylist and add that string
            if (map.containsKey(tempMap))
                map.get(tempMap).add(str);
            else {
                ArrayList<String>
                    tempList = new ArrayList<String>();
                tempList.add(str);
                map.put(tempMap, tempList);
            }
        }
        // Stores the result in a arraylist
        ArrayList<ArrayList<String> >
            result = new ArrayList<>();
        for (HashMap<Character, Integer>
                 temp : map.keySet())
            result.add(map.get(temp));
        return result;
    }
```

Remove minimum number of characters so that two strings become anagram

```
static int countDeletions(String str1, String str2) {
        int arr[] = new int[CHARS];
        for (int i = 0; i < str1.length(); i++) {
            arr[str1.charAt(i) - 'a']++;
        }

        for (int i = 0; i < str2.length(); i++) {
            arr[str2.charAt(i) - 'a']--;
        }

        int ans = 0;
        for (int i = 0; i < CHARS; i++) {
            ans += Math.abs(arr[i]);
        }
        return ans;
    }
```

Longest Common Anagram Subsequence

```
Input : str1 = "abdacp", str2 = "ckamb"

Output : 3

Subsequence of str1 = abc

Subsequence of str2 = cab

          OR

Subsequence of str1 = bac

Subsequence of str2 = cab


These are longest common anagram subsequences.


Input : str1 = "abbcfke", str2 = "fbaafbly"

Output : 4
```

```
static int longCommomAnagramSubseq(String str1,
                                       String str2,
                                       int n1, int n2)
    {
        // hash tables for
        // storing frequencies
        // of each character
        int []freq1 = new int[SIZE];
        int []freq2 = new int[SIZE];

        for(int i = 0; i < SIZE; i++)
        {
            freq1[i] = 0;
            freq2[i] = 0;
        }

        int len = 0;

        // calculate frequency
        // of each character of
        // 'str1[]'
        for (int i = 0; i < n1; i++)
            freq1[(int)str1.charAt(i) - (int)'a']++;

        // calculate frequency
        // of each character
        // of 'str2[]'
        for (int i = 0; i < n2; i++)
            freq2[(int)str2.charAt(i) - (int)'a']++;

        // for each character add
        // its minimum frequency
        // out of the two Strings
        // in 'len'
        for (int i = 0; i < SIZE; i++)
            len += Math.min(freq1[i],
                            freq2[i]);

        // required length()
        return len;
    }
```

Median of two sorted arrays of different sizes O(m+n), O(1)

1) Median : In case of odd number of elements, median will be the middle element, in case of even number of elements, median will be average of middle two elements of the list.

Algorithm:

1. Given two arrays are sorted. So they can be merged in O(m+n) time. Create a variable count to have a count of elements in the output array.

2. If the value of (m+n) is odd then there is only one median else the median is the average of elements at index (m+n)/2 and ((m+n)/2 – 1).
3. To merge the both arrays, keep two indices i and j initially assigned to 0. Compare the ith index of 1st array and jth index of second, increase the index of the smallest element and increase the count.
4. Check if the count reached (m+n) / 2 if (m+n) is odd and store the element, if even store the average of (m+n)/2 th and (m+n)/2 -1 th element and print it.

```
static int getMedian(int ar1[], int ar2[],
                     int n, int m)
{

    // Current index of input array ar1[]
    int i = 0;

    // Current index of input array ar2[]
    int j = 0;
    int count;
    int m1 = -1, m2 = -1;

    // Since there are (n+m) elements,
    // There are following two cases
    // if n+m is odd then the middle
    //index is median i.e. (m+n)/2
    if ((m + n) % 2 == 1)
    {
        for(count = 0;
            count <= (n + m) / 2;
            count++)
        {
            if (i != n && j != m)
            {
                m1 = (ar1[i] > ar2[j]) ?
                      ar2[j++] : ar1[i++];
            }
            else if (i < n)
            {
                m1 = ar1[i++];
            }

            // for case when j<m,
            else
            {
                m1 = ar2[j++];
            }
        }
        return m1;
    }

    // median will be average of elements
    // at index ((m+n)/2 - 1) and (m+n)/2
    // in the array obtained after merging
    // ar1 and ar2
    else
    {
        for(count = 0;
            count <= (n + m) / 2;
            count++)
        {
            m2 = m1;
            if (i != n && j != m)
            {
                m1 = (ar1[i] > ar2[j]) ?
                      ar2[j++] : ar1[i++];
            }
            else if (i < n)
            {
                m1 = ar1[i++];
            }

            // for case when j<m,
            else
            {
                m1 = ar2[j++];
            }
        }
        return (m1 + m2) / 2;
    }
}
```

Optimized Approach Min (O(Log n), O(Log M))

Reference : Youtube

```
class
Solution
{
               public double findMedianSortedArrays(int[] nums1, int[] nums2) {
                   if(nums2.length < nums1.length) return findMedianSortedArrays(nums2,
           nums1);
                   int n1 = nums1.length;
                   int n2 = nums2.length;
                   int low = 0, high = n1;

                   while(low <= high) {
                       int cut1 = (low+high) >> 1;
                       int cut2 = (n1 + n2 + 1) / 2 - cut1;


                       int left1 = cut1 == 0 ? Integer.MIN_VALUE : nums1[cut1-1];
                       int left2 = cut2 == 0 ? Integer.MIN_VALUE : nums2[cut2-1];

                       int right1 = cut1 == n1 ? Integer.MAX_VALUE :   nums1[cut1];
                       int right2 = cut2 == n2 ? Integer.MAX_VALUE : nums2[cut2];
```

```
            if(left1 <= right2 && left2 <= right1) {
                if( (n1 + n2) % 2 == 0 )
                    return (Math.max(left1, left2) + Math.min(right1, 
right2)) / 2.0;
                else
                    return Math.max(left1, left2);
            }
            else if(left1 > right2) {
                high = cut1 - 1;
            }
            else {
                low = cut1 + 1;
            }
        }
        return 0.0;
    }
}
```

Find position of an element in a sorted array of infinite numbers

Since array is sorted, the first thing clicks into mind is binary search, but the problem here is that we don't know size of array.
If the array is infinite, that means we don't have proper bounds to apply binary search. So in order to find position of key, first we find bounds and then apply binary search algorithm.
Let low be pointing to 1st element and high pointing to 2nd element of array, Now compare key with high index element,
->if it is greater than high index element then copy high index in low index and double the high index.
->if it is smaller, then apply binary search on high and low indices found.

```
class Test
{
    // Simple binary search algorithm
    static int binarySearch(int arr[], int l, int r, int 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);
            return binarySearch(arr, mid+1, r, x);
        }
        return -1;
    }

    // Method takes an infinite size array and a key to be
    // searched and returns its position if found else -1.
    // We don't know size of arr[] and we can assume size to be
    // infinite in this function.
    // NOTE THAT THIS FUNCTION ASSUMES arr[] TO BE OF INFINITE SIZE
    // THEREFORE, THERE IS NO INDEX OUT OF BOUND CHECKING
    static int findPos(int arr[],int key)
    {
        int l = 0, h = 1;
        int val = arr[0];

        // Find h to do binary search
        while (val < key)
        {
            l = h;      // store previous high
            //check that 2*h doesn't exceeds array
            //length to prevent ArrayOutOfBoundException
            if(2*h < arr.length-1)
               h = 2*h;
            else
               h = arr.length-1;

            val = arr[h]; // update new val
        }

        // at this point we have updated low
        // and high indices, thus use binary
        // search between them
        return binarySearch(arr, l, h, key);
    }

    // Driver method to test the above function
    public static void main(String[] args)
    {
        int arr[] = new int[]{3, 5, 7, 9, 10, 90,
                            100, 130, 140, 160, 170};
        int ans = findPos(arr,10);

        if (ans==-1)
            System.out.println("Element not found");
        else
            System.out.println("Element found at index " + ans);
    }
}
```

Let p be the position of element to be searched. Number of steps for finding high index ‘h’ is O(Log p). The value of ‘h’ must be less than 2*p. The number of elements between h/2 and h must be O(p). Therefore, time complexity of Binary Search step is also O(Log p) and overall time complexity is 2*O(Log p) which is O(Log p).

Find distinct palindromic substrings in a string

O(n3):

Algm :-

1) Run 2 for loops : outer for loop will run from 0 to length of string
    a. Inner for loop will run from i+1 to length of string
        i. Get substring from I to J and check if that substring is palindrome or not, if yes print it else ignore.

Reference

Approach 2: - O(n2)

Reference :-

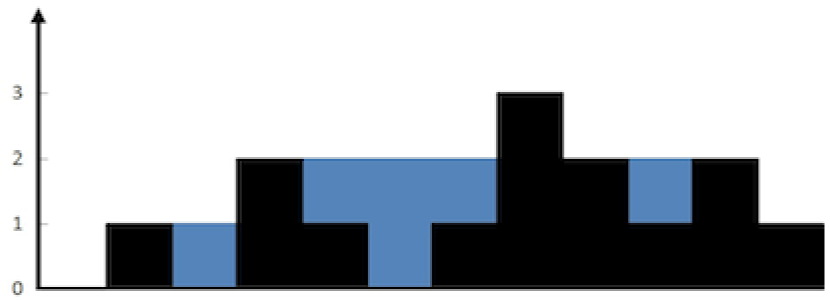
```
import java.io.*;

import java.util.*;


public class Main {

  public static void main(String[] args) throws Exception {

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    String str = br.readLine();


    boolean[][] dp = new boolean[str.length()][str.length()];

    int count = 0;

    for(int g = 0; g < dp.length; g++){

      for(int i = 0, j = g; j < dp[0].length; i++, j++){

        if(g == 0){

          dp[i][j] = true;

        } else if(g == 1){

          if(str.charAt(i) == str.charAt(j)){

            dp[i][j] = true;

          } else {

            dp[i][j] = false;

          }

        } else {

          if(str.charAt(i) == str.charAt(j)){

            dp[i][j] = dp[i + 1][j - 1];
```

```
                } else {
                    dp[i][j] = false;
                }
            }

            if(dp[i][j]){
                count++;
            }
        }
    }

    System.out.println(count);
  }
}
```

Trapping Rainwater :-

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:



```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6

Explanation: The above elevation map (black section) is represented by array
[0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are
being trapped.
```

Reference

Solution:-

Approach 1 :- For each index find **minimum(leftMaxsoFar,RightMaxAhead)** – currentHeight

We will have to use two for loop: For each index find left max and then right max and calculate above

Complexity : O(n2)

Better Approach :-

Use two arrays prefix and suffix seperatly

1. Run a loop to fill prefix array for each index by maxLeftSofar.
2. Run a loop to fill suffix array for each index by rightMaxSofar from last to end.
3. Then at last Run a final loop to add the values by using for any index I : Min(suffix[i],prefix[i]) – currentHeight

   Complexity : O(n), Space : O(2n)

Best Approach using two pointer technique :- O(n) O(1)

```
class Solution {
    public int trap(int[] height) {


        int n = height.length;
        int left=0; int right=n-1;
        int res=0;
        int maxleft=0, maxright=0;


        while(left<=right){


            if(height[left]<=height[right]){


                if(height[left]>=maxleft) maxleft=height[left];
                else res+=maxleft-height[left];


                left++;
            }
            else{

                if(height[right]>=maxright) maxright= height[right];
                else res+=maxright-height[right];

                right--;
            }
        }
        return res;
    }
}
```

Longest Word in Dictionary through Deleting

Reference

Given a string `s` and a string array `dictionary`, return *the longest string in the dictionary that can be formed by deleting some of the given string characters*. If there is more than one possible result, return the longest word with the smallest lexicographical order. If there is no possible result, return the empty string.

Example 1:

```
Input: s = "abpcplea", dictionary = ["ale","apple","monkey","plea"]
Output: "apple"
```

Example 2:

```
Input: s = "abpcplea", dictionary = ["a","b","c"]
Output: "a"
```

Approach :- For each string in dictionary, check if I'th string in dict is subsequence of the big String s.

To check for subsequence : Compare each character of both string, if there is a match increase both, if dict is not matching just increase pointer in big string, at last check if Dict string length is the current pointer , means it's a subsequence.

```
class Solution {
    public String findLongestWord(String s, List<String> d) {
        String result = "";

        for(String str : d) {
            if(isSubsequence(s, str) ) {
                //Length str > result || length is equal but str is lexicographically small.
                if(str.length() > result.length() || (str.length() == result.length() && str.compareTo(result) <
1))
                result = str;
            }
        }
        return result;
    }

    public boolean isSubsequence(String S, String D) {
        int i = 0, j = 0;
        if(D.length() > S.length()) return false;
        while(i < S.length() && j < D.length()) {
            if(S.charAt(i) == D.charAt(j)) j++;
            i++;
        }
        return j == D.length();
    }
}
```

//Complexity : O(NX) where N is number of strings in the list and X is the average number of characters in string

First Non-repeating characters in the String. O(n), O(n)

```
class Solution {
    public int firstUniqChar(String s) {
        HashMap<Character, Integer> count = new HashMap<Character, Integer>();
        int n = s.length();
        // build hash map : character and how often it appears
        for (int i = 0; i < n; i++) {
            char c = s.charAt(i);
            count.put(c, count.getOrDefault(c, 0) + 1);
        }

        // find the index
        for (int i = 0; i < n; i++) {
            if (count.get(s.charAt(i)) == 1)
                return i;
        }
        return -1;
    }
}
```

Find subarray with given sum

O(n), O(1) Ref

Efficient Approach: There is an idea if all the elements of the array are positive. If a subarray has sum greater than the given sum then there is no possibility that adding elements to the current subarray the sum will be *x* (given sum). Idea is to use a similar approach to a sliding window. Start with an empty subarray, add elements to the subarray until the sum is less than *x*. If the sum is greater than *x*, remove elements from the start of the current subarray.
Algorithm:

1. Create three variables, *l=0, sum = 0*
2. Traverse the array from start to end.
3. Update the variable sum by adding current element, *sum = sum + array[i]*
4. If the sum is greater than the given sum, update the variable sum as *sum = sum – array[l]*, and update l as, l++.
5. If the sum is equal to given sum, print the subarray and break the loop.
6.

Find subarray with given sum | Set 2 (Handles Negative Numbers)

https://www.geeksforgeeks.org/find-subarray-with-given-sum-in-array-of-integers/

Smallest subarray with sum greater than a given value

Approach : Two pointer technique

- Initialize left pointer to 0 sum to 0
- Iterate over the nums:
    - Add $\text{nums}[i]$ to sum
    - While sum is greater than or equal to s:
        - Update $\text{ans}=\min(\text{ans},i+1-\text{left})$, where $\text{i}+1-\text{left}$ is the size of current subarray
        - It means that the first index can safely be incremented, since, the minimum subarray starting with this index with $\text{sum}\geq s$ has been achieved
        - Subtract $\text{nums}[\text{left}]$ from sum and increment left

```
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int sum = 0;
        int result = Integer.MAX_VALUE;
        int left = 0;
        for(int i = 0;i<nums.length;i++){
            sum+=nums[i];
            while(sum >= target){
                result = Math.min(result, i+1-left);
                sum -= nums[left++];
            }
        }
        return result == Integer.MAX_VALUE ? 0 : result;
    }
}
```

Subsets of an Array ***Ref***

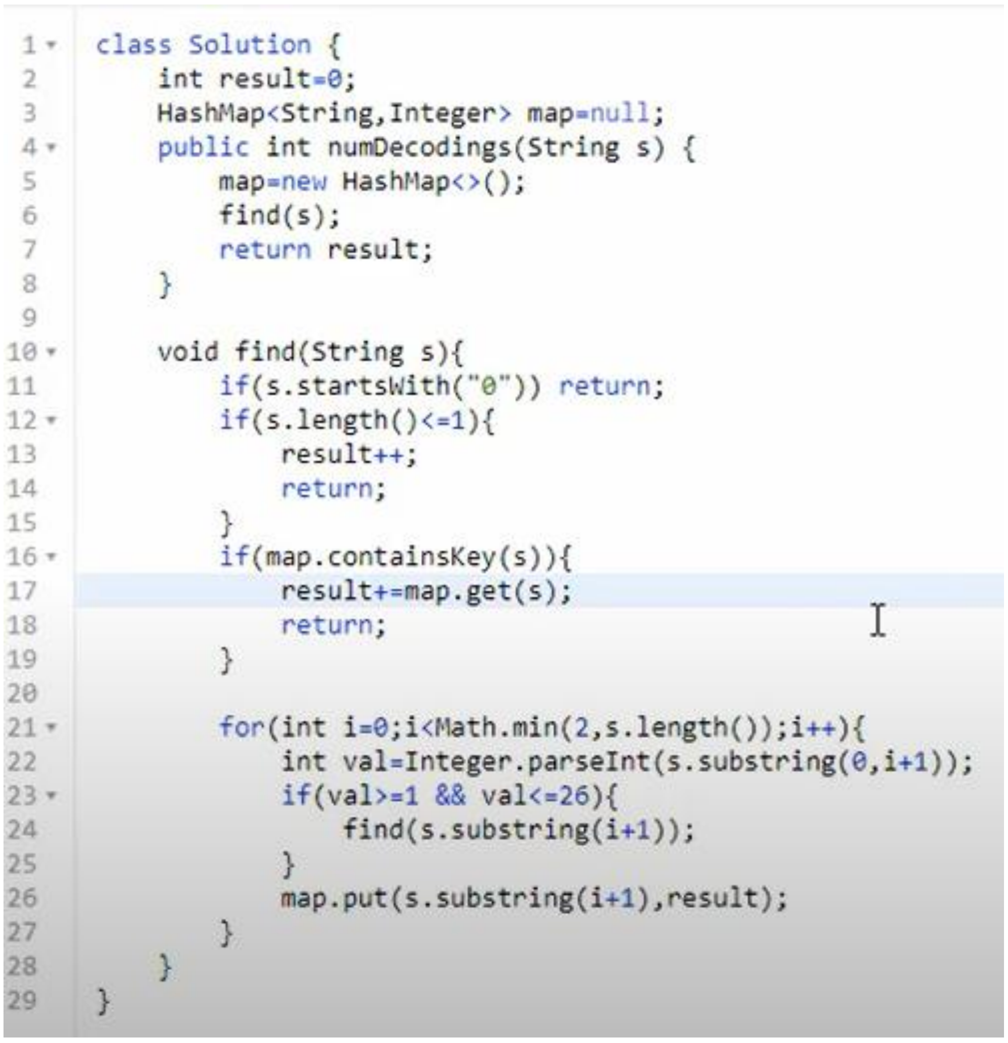
```
import java.io.*;
import java.util.*;
```

```
public class Main{

public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    int n = Integer.parseInt(br.readLine());
    int[] arr = new int[n];
    for(int i = 0; i < n; i++){
       arr[i] = Integer.parseInt(br.readLine());
    }

    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < (1 << arr.length); i++){
       int dec = i;
       String str = "";

       for(int j = 0; j < arr.length; j++){
          int r = dec % 2;
          dec = dec / 2;

          if(r == 0){
            str = "-\t" + str;
          } else {
            str = arr[arr.length - 1 - j] + "\t" + str;
          }
       }

       sb.append(str + "\n");
    }

    System.out.println(sb);
 }

}
```

Find number of positional elements :-

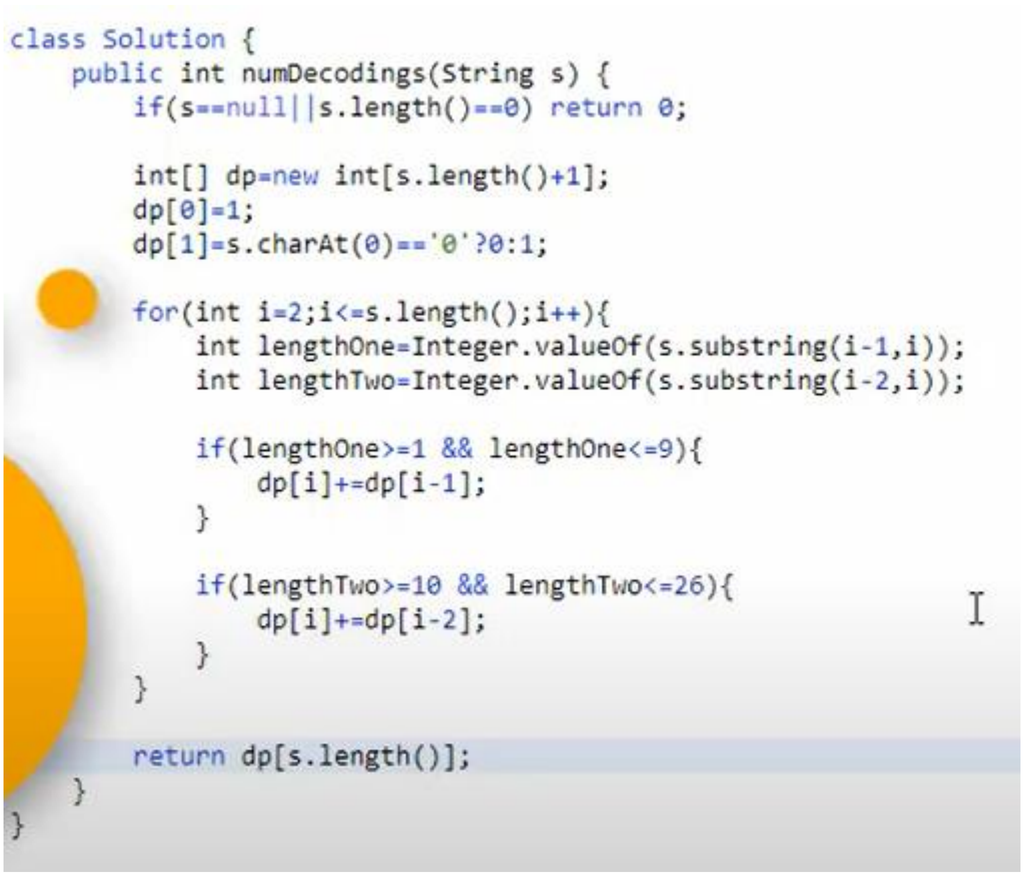
DECODE WAYS TO STRING

O(n), O(n) Ref



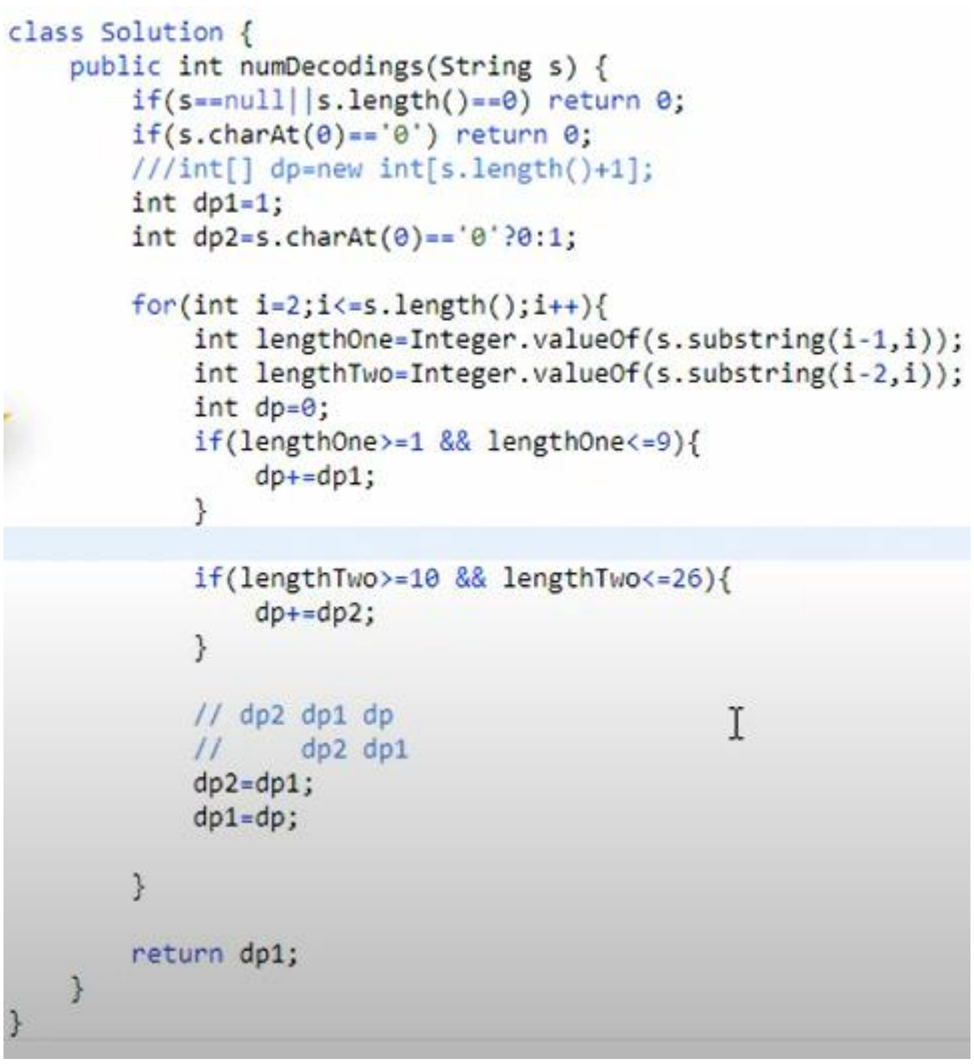
```java
class Solution {
    int result=0;
    HashMap<String,Integer> map=null;
    public int numDecodings(String s) {
        map=new HashMap<>();
        find(s);
        return result;
    }

    void find(String s){
        if(s.startsWith("0")) return;
        if(s.length()<=1){
            result++;
            return;
        }
        if(map.containsKey(s)){
            result+=map.get(s);
            return;
        }

        for(int i=0;i<Math.min(2,s.length());i++){
            int val=Integer.parseInt(s.substring(0,i+1));
            if(val>=1 && val<=26){
                find(s.substring(i+1));
            }
            map.put(s.substring(i+1),result);
        }
    }
}
```

Dyanamic programming solution O(n), O(1)



```
class Solution {
    public int numDecodings(String s) {
        if(s==null||s.length()==0) return 0;

        int[] dp=new int[s.length()+1];
        dp[0]=1;
        dp[1]=s.charAt(0)=='0'?0:1;

        for(int i=2;i<=s.length();i++){
            int lengthOne=Integer.valueOf(s.substring(i-1,i));
            int lengthTwo=Integer.valueOf(s.substring(i-2,i));

            if(lengthOne>=1 && lengthOne<=9){
                dp[i]+=dp[i-1];
            }

            if(lengthTwo>=10 && lengthTwo<=26){
                dp[i]+=dp[i-2];
            }
        }

        return dp[s.length()];
    }
}
```

Below is the solution for O(n) and O(1)



```java
class Solution {
    public int numDecodings(String s) {
        if(s==null||s.length()==0) return 0;
        if(s.charAt(0)=='0') return 0;
        ///int[] dp=new int[s.length()+1];
        int dp1=1;
        int dp2=s.charAt(0)=='0'?0:1;

        for(int i=2;i<=s.length();i++){
            int lengthOne=Integer.valueOf(s.substring(i-1,i));
            int lengthTwo=Integer.valueOf(s.substring(i-2,i));
            int dp=0;
            if(lengthOne>=1 && lengthOne<=9){
                dp+=dp1;
            }

            if(lengthTwo>=10 && lengthTwo<=26){
                dp+=dp2;
            }

            // dp2 dp1 dp
            //     dp2 dp1
            dp2=dp1;
            dp1=dp;

        }

        return dp1;
    }
}
```

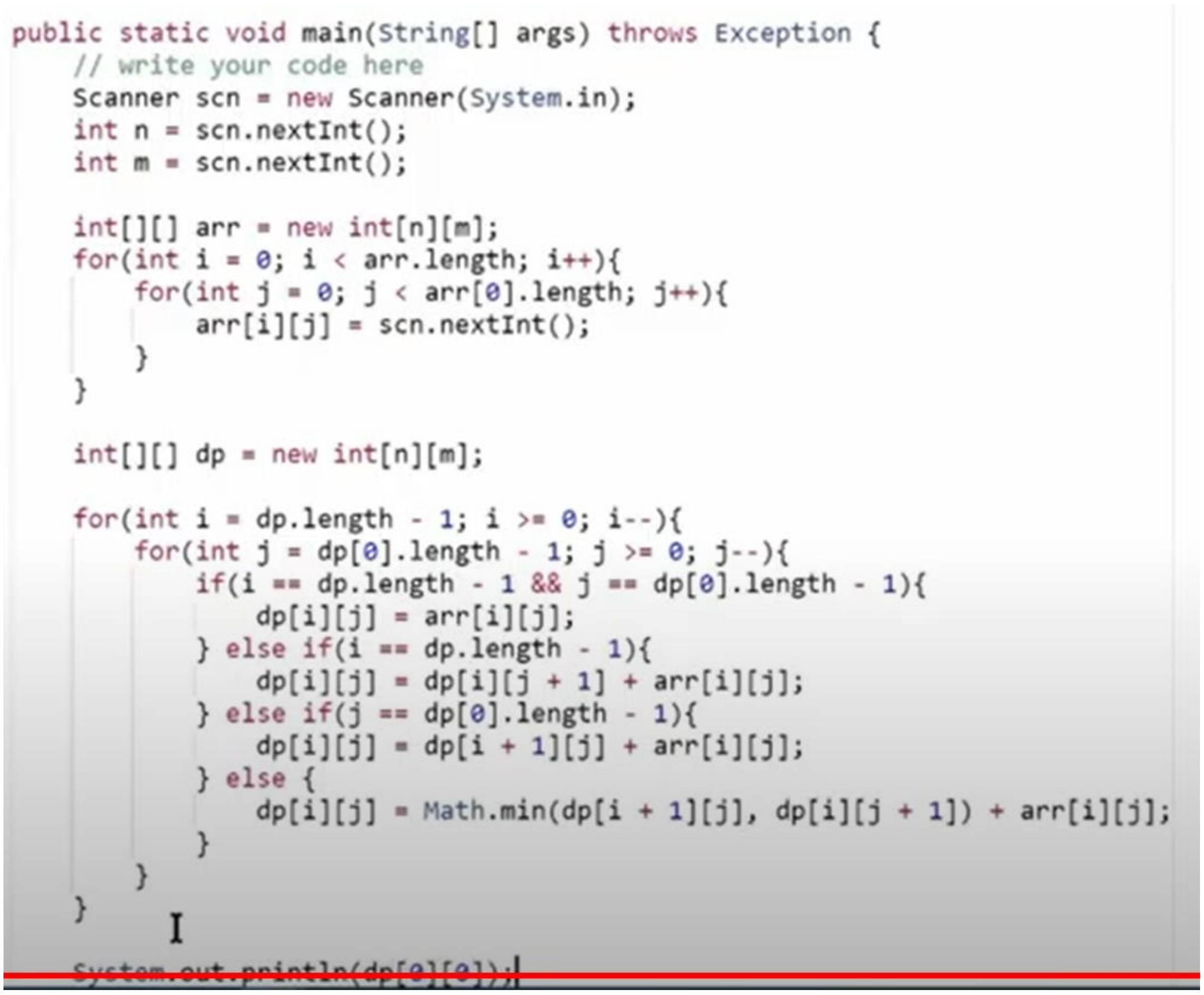
Find minimum cost to reach the last cell of a matrix from its first cell

Ref

Approach :

1) Fill the last cell by same value - last cell se last jaane ki cost
2) Fill the last row, by add each element to its subsequent element
3) Fill the last column by adding each element to its down element
4) For all other cells, take the minimum for horixontal next and bottom next element and add the curret element to it.

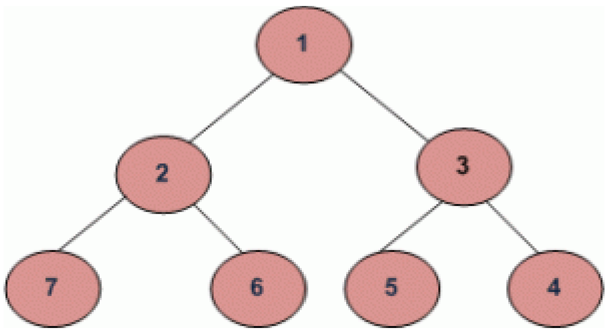
Compelxity O(m*n)



```
public static void main(String[] args) throws Exception {
    // write your code here
    Scanner scn = new Scanner(System.in);
    int n = scn.nextInt();
    int m = scn.nextInt();

    int[][] arr = new int[n][m];
    for(int i = 0; i < arr.length; i++){
        for(int j = 0; j < arr[0].length; j++){
            arr[i][j] = scn.nextInt();
        }
    }

    int[][] dp = new int[n][m];

    for(int i = dp.length - 1; i >= 0; i--){
        for(int j = dp[0].length - 1; j >= 0; j--){
            if(i == dp.length - 1 && j == dp[0].length - 1){
                dp[i][j] = arr[i][j];
            } else if(i == dp.length - 1){
                dp[i][j] = dp[i][j + 1] + arr[i][j];
            } else if(j == dp[0].length - 1){
                dp[i][j] = dp[i + 1][j] + arr[i][j];
            } else {
                dp[i][j] = Math.min(dp[i + 1][j], dp[i][j + 1]) + arr[i][j];
            }
        }
    }

    System.out.println(dp[0][0]);
```

ZigZag Tree Traversal

Write a function to print ZigZag order traversal of a binary tree. For the below binary tree the zigzag order traversal will be **1 3 2 7 6 5 4.**



This problem can be solved using two stacks. Assume the two stacks are current: **currentlevel and nextlevel.** We would also need a variable to keep track of the current level order(whether it is left to right or right to left). We pop from the currentlevel stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to the stack nextlevel. Since a stack is a LIFO(Last-In-First_out) structure, next time when nodes are popped off nextlevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, do-not forget to swap those two stacks at the end of each level(i.e., when current level is empty)
Below is the implementation of the above approach:

```
 class Node

{
int data;
Node leftChild;
Node rightChild;
Node(int data)
{
    this.data = data;
}
}

class BinaryTree {
Node rootNode;

// function to print the
// zigzag traversal
void printZigZagTraversal() {

    // if null then return
    if (rootNode == null) {
    return;
    }

    // declare two stacks
    Stack<Node> currentLevel = new Stack<>();
    Stack<Node> nextLevel = new Stack<>();

    // push the root
    currentLevel.push(rootNode);
    boolean leftToRight = true;

    // check if stack is empty
    while (!currentLevel.isEmpty()) {

    // pop out of stack
    Node node = currentLevel.pop();

    // print the data in it
    System.out.print(node.data + " ");

    // store data according to current
    // order.
    if (leftToRight) {
        if (node.leftChild != null) {
        nextLevel.push(node.leftChild);
        }
        if (node.rightChild != null) {
        nextLevel.push(node.rightChild);
        }
    }
    else {
        if (node.rightChild != null) {
        nextLevel.push(node.rightChild);
        }

        if (node.leftChild != null) {
        nextLevel.push(node.leftChild);
        }
    }

    if (currentLevel.isEmpty()) {
        leftToRight = !leftToRight;
        Stack<Node> temp = currentLevel;
        currentLevel = nextLevel;
        nextLevel = temp;
    }
    }
}
}

public class zigZagTreeTraversal {

// driver program to test the above function
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.rootNode = new Node(1);
    tree.rootNode.leftChild = new Node(2);
    tree.rootNode.rightChild = new Node(3);
    tree.rootNode.leftChild.leftChild = new Node(7);
    tree.rootNode.leftChild.rightChild = new Node(6);
    tree.rootNode.rightChild.leftChild = new Node(5);
    tree.rootNode.rightChild.rightChild = new Node(4);

    System.out.println("ZigZag Order traversal of binary tree is");
    tree.printZigZagTraversal();
}
}
```

Find maximum possible stolen value from houses

There are n houses build in a line, each of which contains some value in it. A thief is going to steal the maximal value of these houses, but he can’t steal in two adjacent houses because the owner of the stolen houses will tell his two neighbors left and right side. What is the maximum stolen value?
Examples:

```
Input: hval[] = {6, 7, 1, 3, 8, 2, 4}
Output: 19

Explanation: The thief will steal 6, 1, 8 and 4 from the house.
```

```
Input: hval[] = {5, 3, 4, 11, 2}
Output: 16

Explanation: Thief will steal 5 and 11
```

Approach :- Create an extra space *dp*, DP array to store the sub-problems.

1. Tackle some basic cases, if the length of the array is 0, print 0, if the length of the array is 1, print the first element, if the length of the array is 2, print maximum of two elements.
2. Update *dp[0]* as *array[0]* and *dp[1]* as maximum of *array[0]* and *array[1]*
3. Traverse the array from the second element (2nd index) to the end of array.
4. For every index, update *dp[i]* as maximum of *dp[i-2]* + *array[i]* and *dp[i-1]*, this step defines the two cases, if an element is selected then the previous element cannot be selected and if an element is not selected then the previous element can be selected.
5. Print the value *dp[n-1]*

```
class GFG
{
    // Function to calculate the maximum stolen value
    static int maxLoot(int hval[], int n)
    {
        if (n == 0)
        return 0;
        if (n == 1)
            return hval[0];
        if (n == 2)
            return Math.max(hval[0], hval[1]);

        // dp[i] represent the maximum value stolen
        // so far after reaching house i.
        int[] dp = new int[n];

        // Initialize the dp[0] and dp[1]
        dp[0] = hval[0];
        dp[1] = Math.max(hval[0], hval[1]);

        // Fill remaining positions
        for (int i = 2; i<n; i++)
            dp[i] = Math.max(hval[i]+dp[i-2], dp[i-1]);

        return dp[n-1];
    }

    // Driver program
    public static void main (String[] args)
    {
        int hval[] = {6, 7, 1, 3, 8, 2, 4};
        int n = hval.length;
        System.out.println("Maximum loot value : " + maxLoot(hval, n));
    }
}
```

Complexity Analysis:

- **Time Complexity:** .
  Only one traversal of original array is needed. So the time complexity is O(n)
- **Space Complexity:** .
  An array is required of size n, so space complexity is O(n).

LRU Cache Implementation :-

1. Using LinkedHashSet : O(1) O(1)

```
class LRUCache {

    Set<Integer> cache;
    int capacity;

    public LRUCache(int capacity)
    {
        this.cache = new LinkedHashSet<Integer>(capacity);
        this.capacity = capacity;
    }

    // This function returns false if key is not
    // present in cache. Else it moves the key to
    // front by first removing it and then adding
    // it, and returns true.
    public boolean get(int key)
    {
        if (!cache.contains(key))
            return false;
        cache.remove(key);
        cache.add(key);
        return true;
    }

    /* Refers key x with in the LRU cache */
    public void refer(int key)
    {
        if (get(key) == false)
           put(key);
    }

    // displays contents of cache in Reverse Order
    public void display()
    {
      LinkedList<Integer> list = new LinkedList<>(cache);

      // The descendingIterator() method of java.util.LinkedList
      // class is used to return an iterator over the elements
      // in this LinkedList in reverse sequential order
      Iterator<Integer> itr = list.descendingIterator();

      while (itr.hasNext())
            System.out.print(itr.next() + " ");
    }

    public void put(int key)
    {

      if (cache.size() == capacity) {
            int firstKey = cache.iterator().next();
            cache.remove(firstKey);
        }
        cache.add(key);
    }

    public static void main(String[] args)
    {
        LRUCache ca = new LRUCache(4);
        ca.refer(1);
        ca.refer(2);
        ca.refer(3);
        ca.refer(1);
        ca.refer(4);
        ca.refer(5);
        ca.display();
    }
}
```

2. Using Deque

```
public class LRUCache {

    // store keys of cache
    private Deque<Integer> doublyQueue;

    // store references of key in cache
    private HashSet<Integer> hashSet;

    // maximum capacity of cache
    private final int CACHE_SIZE;

    LRUCache(int capacity) {
        doublyQueue = new LinkedList<>();
        hashSet = new HashSet<>();
        CACHE_SIZE = capacity;
    }

    /* Refer the page within the LRU cache */
    public void refer(int page) {
        if (!hashSet.contains(page)) {
            if (doublyQueue.size() == CACHE_SIZE) {
                int last = doublyQueue.removeLast();
                hashSet.remove(last);
            }
        }
        else {/* The found page may not be always the last element, even
if it's an
              intermediate element that needs to be removed and added to
the start
              of the Queue */
            doublyQueue.remove(page);
        }
        doublyQueue.push(page);
        hashSet.add(page);
    }

    // display contents of cache
    public void display() {
        Iterator<Integer> itr = doublyQueue.iterator();
        while (itr.hasNext()) {
            System.out.print(itr.next() + " ");
        }
    }

    public static void main(String[] args) {
        LRUCache cache = new LRUCache(4);
        cache.refer(1);
        cache.refer(2);
        cache.refer(3);
        cache.refer(1);
        cache.refer(4);
        cache.refer(5);
        cache.refer(2);
        cache.refer(2);
        cache.refer(1);
        cache.display();
    }
}
```

Smarted and shortest LRU using LinkedHashMap:-

```
class LRUCache {
    private LinkedHashMap<Integer, Integer> map;
    private final int CAPACITY;
    public LRUCache(int capacity)
    {
        CAPACITY = capacity;
        map = new LinkedHashMap<Integer, Integer>(capacity, 0.75f, true)
{
            protected boolean removeEldestEntry(Map.Entry eldest)
            {
                return size() > CAPACITY;
            }
        };
    }

    // This method works in O(1)
    public int get(int key)
    {
        System.out.println("Going to get the value " +
                               "for the key : " + key);
        return map.getOrDefault(key, -1);
    }

    // This method works in O(1)
    public void set(int key, int value)
    {
        System.out.println("Going to set the (key, " +
             "value) : (" + key + ", " + value + ")");
        map.put(key, value);
    }
}
```

CIRCULAR TOUR PETROL TRUCK problem :-

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1-litre petrol, the truck can go 1 unit of distance.
For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where the truck can make a circular tour is 2nd petrol pump. Output should be "start = 1" (index of 2nd petrol pump).

```
public class Petrol
{
    // A petrol pump has petrol and distance to next petrol pump
    static class petrolPump
    {
        int petrol;
        int distance;

        // constructor
        public petrolPump(int petrol, int distance)
        {
            this.petrol = petrol;
            this.distance = distance;
        }
    }

    // The function returns starting point if there is a possible
solution,
    // otherwise returns -1
    static int printTour(petrolPump arr[], int n)
    {
        int start = 0;
        int end = 1;
        int curr_petrol = arr[start].petrol - arr[start].distance;

        // If current amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while(end != start || curr_petrol < 0)
        {

            // If current amount of petrol in truck becomes less than 0,
then
            // remove the starting petrol pump from tour
            while(curr_petrol < 0 && start != end)
            {
                // Remove starting petrol pump. Change start
                curr_petrol -= arr[start].petrol - arr[start].distance;
                start = (start + 1) % n;

                // If 0 is being considered as start again, then there is
no
                // possible solution
                if(start == 0)
                    return -1;
            }
            // Add a petrol pump to current tour
            curr_petrol += arr[end].petrol - arr[end].distance;
            end = (end + 1)%n;
        }

        // Return starting point
        return start;
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {

        petrolPump[] arr = {new petrolPump(6, 4),
                            new petrolPump(3, 6),
                            new petrolPump(7, 3)};

        int start = printTour(arr, arr.length);

        System.out.println(start == -1 ? "No Solution" : "Start = " +
start);

    }

}
```

N Queen Problem :-

Backtracking Algorithm : Reference
The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

```
1) Start in the leftmost column

2) If all queens are placed

    return true

3) Try all rows in the current column.

   Do following for every tried row.

    a) If the queen can be placed safely in this row

       then mark this [row, column] as part of the

       solution and recursively check if placing

       queen here leads to a solution.

    b) If placing the queen in [row, column] leads to

       a solution then return true.

    c) If placing queen doesn't lead to a solution then

       unmark this [row, column] (Backtrack) and go to

       step (a) to try other rows.

3) If all rows have been tried and nothing worked,

   return false to trigger backtracking.
```

Ex :- A queen is placed at 1,2 => It can be attacked by the queen in same row i.,e 1st row and it can be attacked by queen is same column i.e.,2nd column.

To check same row : dx is zero , To check same column dy = 0 and to check for diagonal dx!=dy

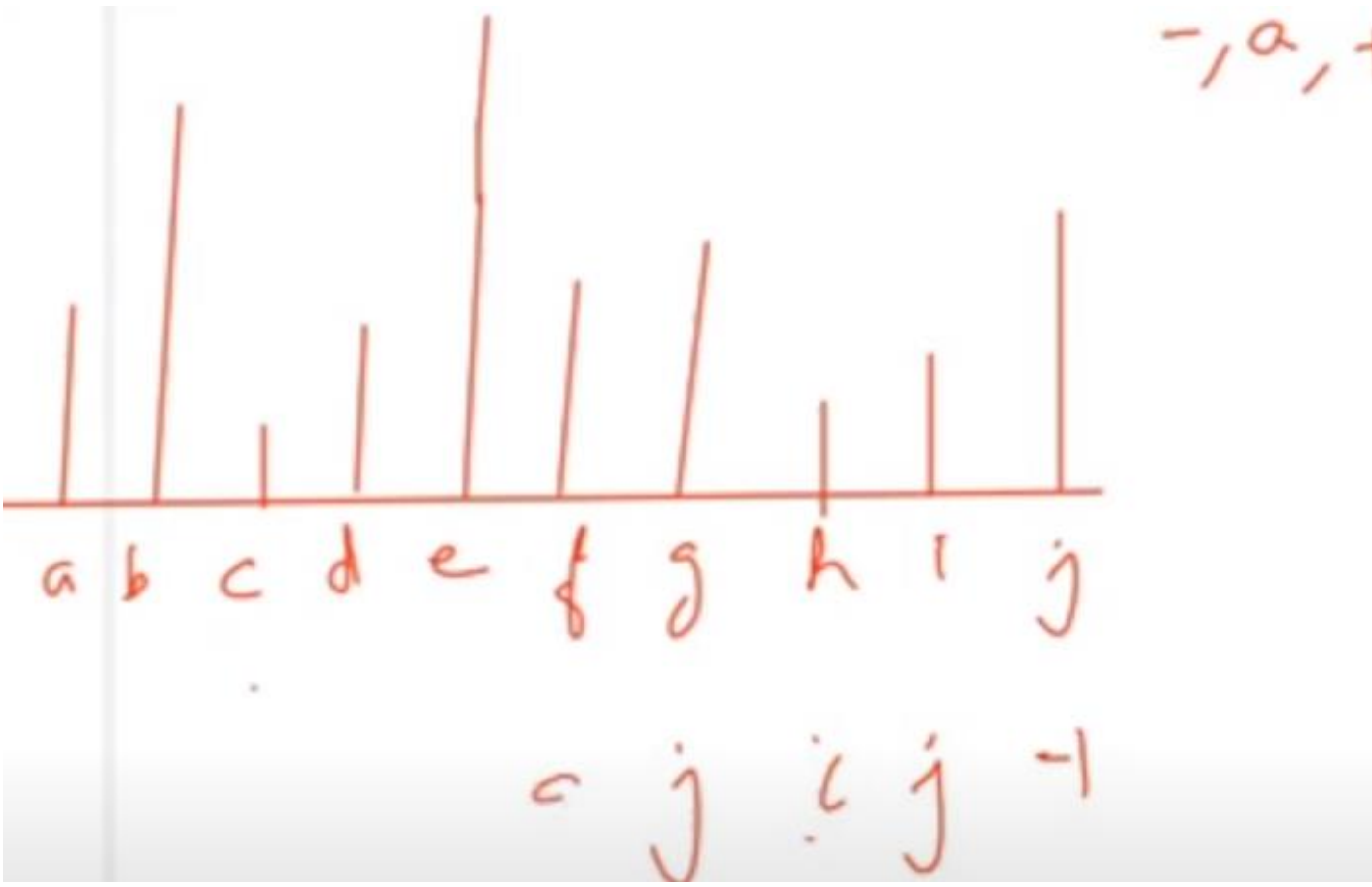
//Ref :-Algorithm made easy

```
class Solution {
    List<List<String>> result;
    public List<List<String>> solveNQueens(int n) {
        result = new ArrayList<>();
        char[][] board = new char[n][n];

        //Filled it as empty cells
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++) {
                board[i][j] = '.';
            }
        }

        List<int[]> queens = new ArrayList<>();
        dfs(board, 0, queens);
        return result;
    }

    private void dfs(char[][] board, int r, List<int[]> queens) {
        //Check if all queens are placed
        if(queens.size() == board.length) {
            //Construct output
            List<String> rows = new ArrayList<>();
            for(char[] row : board) {
                rows.add(new String(row));
            }
            result.add(rows);
        }
        //Try adding the queen
        for(int c = 0; c < board.length; c++) {
            if(canAddQueen(r,c,queens)) {
                board[r][c] = 'Q';
                queens.add(new int[]{r,c});
                dfs(board, r+1, queens);
                board[r][c] = '.';
                queens.remove(queens.size()-1);
            }
        }
    }

    private boolean canAddQueen(int r, int c, List<int[]> queens) {
        for(int[] q : queens) {
            int dx = Math.abs(r-q[0]);
            int dy = Math.abs(c-q[1]);
            if(dx==0 || dy==0 || dx==dy) return false;
        }
        return true;
    }
}
```

Next greater right element

Approach :-

Use a stack, iterate array from right, push last element in array -> and then for each element check in stack if the elements are smaller, keep popping the element until the top of stack is greater than current array element, and print it. And then push the current array element. Ref

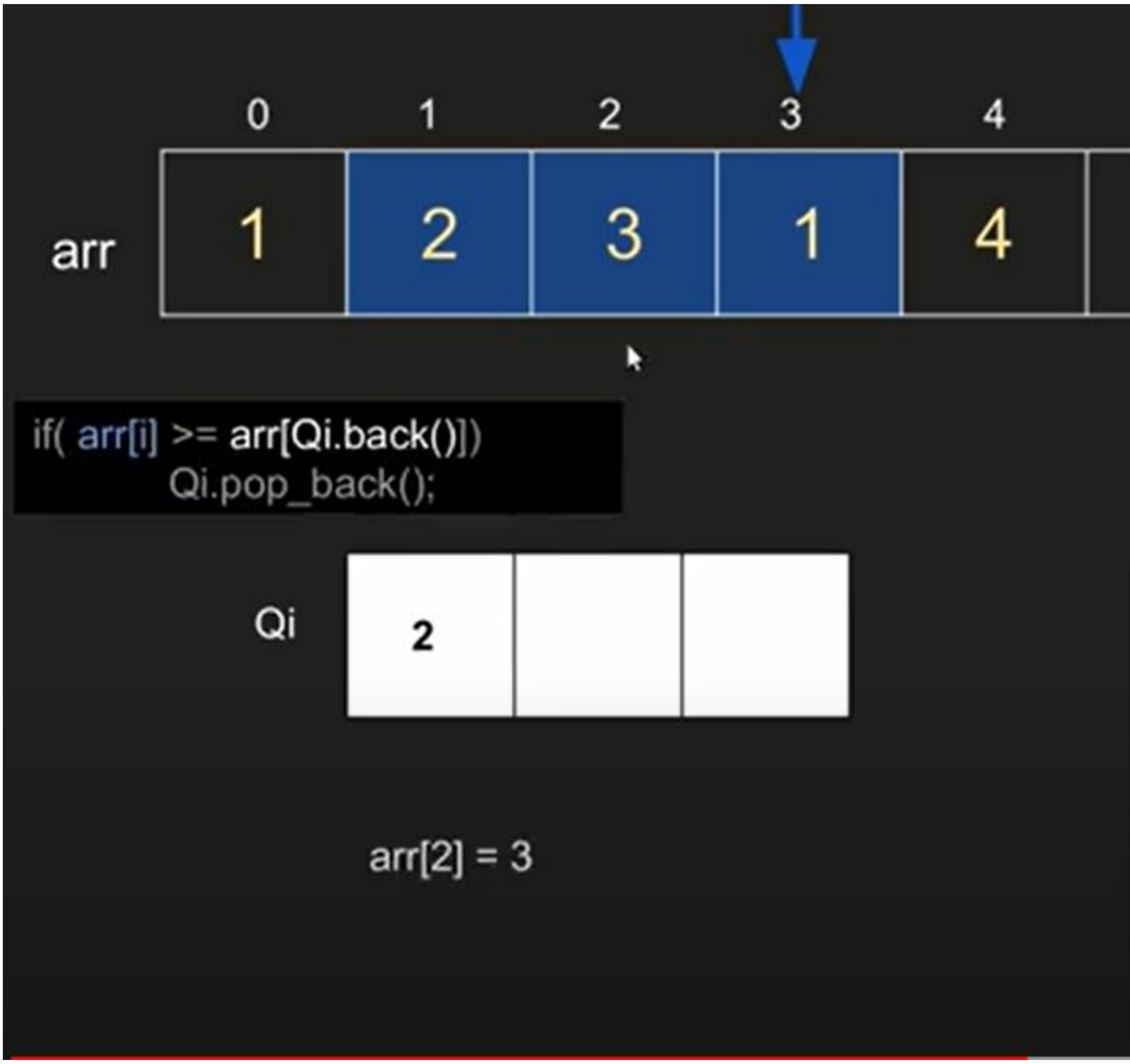


```
public static int[] solve(int[] arr) {
    int[] nge = new int[arr.length];

    Stack<Integer> st = new Stack<>();

    nge[arr.length - 1] = -1;
    st.push(arr[arr.length - 1]);
    for(int i = arr.length - 2; i >= 0; i--){
      while(st.size() > 0 && arr[i] >= st.peek()){
        st.pop();
      }
      if(st.size() > 0){
        nge[i] = st.peek();
      } else {
        nge[i] = -1;
      }

      st.push(arr[i]);
    }

    return nge;
  }
```

Sliding Window Maximum

find and print the maximum element in every window of size k.
e.g.
for the array [2 9 3 8 1 7 12 6 14 4 32 0 7 19 8 12 6] and k = 4, the answer is [9 9 8 12 12 14 14 32 32 32 32 19 19 19]
Using Deque :- (Ref)

```
 static void printMax(int arr[], int n, int k)
    {

        // Create a Double Ended Queue, Qi
        // that will store indexes of array elements
        // The queue will store indexes of
        // useful elements in every window and it will
        // maintain decreasing order of values
        // from front to rear in Qi, i.e.,
        // arr[Qi.front[]] to arr[Qi.rear()]
        // are sorted in decreasing order
        Deque<Integer> Qi = new LinkedList<Integer>();

        /* Process first k (or first window)
        elements of array */
        int i;
        for (i = 0; i < k; ++i)
        {

            // For every element, the previous
            // smaller elements are useless so
            // remove them from Qi
            while (!Qi.isEmpty() && arr[i] >=
                               arr[Qi.peekLast()])

                // Remove from rear
                Qi.removeLast();

            // Add new element at rear of queue
            Qi.addLast(i);
        }

        // Process rest of the elements,
        // i.e., from arr[k] to arr[n-1]
        for (; i < n; ++i)
        {

            // The element at the front of the
            // queue is the largest element of
            // previous window, so print it
            System.out.print(arr[Qi.peek()] + " ");

            // Remove the elements which
            // are out of this window
            while ((!Qi.isEmpty()) && Qi.peek() <=
                                            i - k)
                Qi.removeFirst();

            // Remove all elements smaller
            // than the currently
            // being added element (remove
            // useless elements)
            while ((!Qi.isEmpty()) && arr[i] >=
                              arr[Qi.peekLast()])
                Qi.removeLast();

            // Add current element at the rear of Qi
            Qi.addLast(i);
        }

        // Print the maximum element of last window
        System.out.print(arr[Qi.peek()]);
    }
```

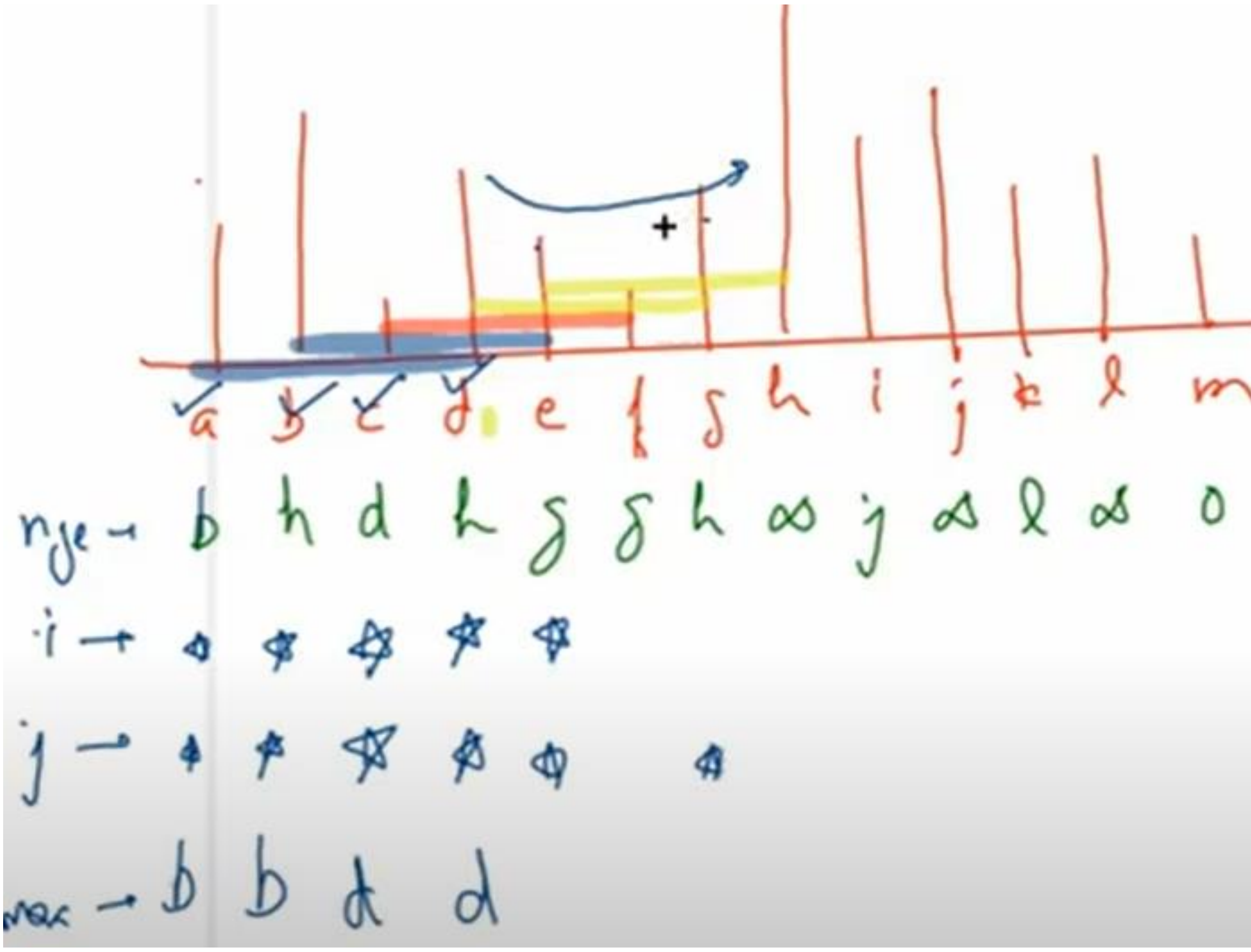


```
if( arr[i] >= arr[Qi.back()])
        Qi.pop_back();
```

```
static void printMax(int arr[], int n, int k)
   {

       // Create a Double Ended Queue, Qi
       // that will store indexes of array elements
       // The queue will store indexes of
       // useful elements in every window and it will
       // maintain decreasing order of values
       // from front to rear in Qi, i.e.,
       // arr[Qi.front[]] to arr[Qi.rear()]
       // are sorted in decreasing order
       Deque<Integer> Qi = new LinkedList<Integer>();

       /* Process first k (or first window)
       elements of array */
       int i;
```

```
        for (i = 0; i < k; ++i)
        {

            // For every element, the previous
            // smaller elements are useless so
            // remove them from Qi
            while (!Qi.isEmpty() && arr[i] >=
                           arr[Qi.peekLast()])

                // Remove from rear
                Qi.removeLast();

            // Add new element at rear of queue
            Qi.addLast(i);
        }

        // Process rest of the elements,
        // i.e., from arr[k] to arr[n-1]
        for (; i < n; ++i)
        {

            // The element at the front of the
            // queue is the largest element of
            // previous window, so print it
            System.out.print(arr[Qi.peek()] + " ");

            // Remove the elements which
            // are out of this window
            while ((!Qi.isEmpty()) && Qi.peek() <=
                                             i - k)
                Qi.removeFirst();

            // Remove all elements smaller
            // than the currently
            // being added element (remove
            // useless elements)
            while ((!Qi.isEmpty()) && arr[i] >=
                              arr[Qi.peekLast()])
                Qi.removeLast();

            // Add current element at the rear of Qi
            Qi.addLast(i);
        }

        // Print the maximum element of last window
        System.out.print(arr[Qi.peek()]);
    }
```

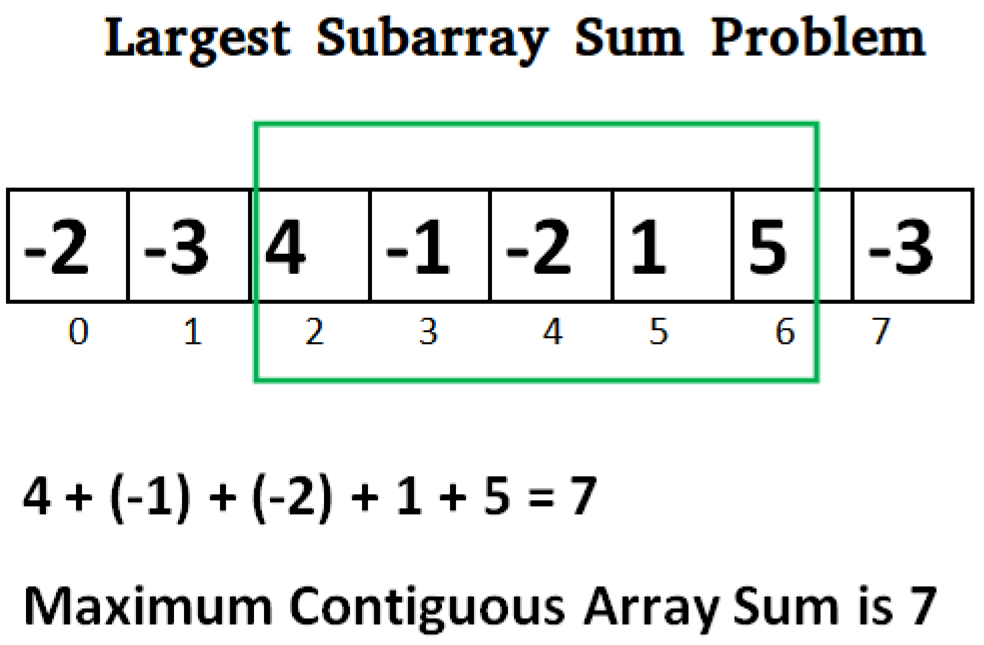
Using Stack (Pepcoding Ref)



```
public static void main(String[] args) throws Exception {
   BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

   int n = Integer.parseInt(br.readLine());
   int[] arr = new int[n];
   for(int i = 0; i < n; i++){
     arr[i] = Integer.parseInt(br.readLine());
   }
   int k = Integer.parseInt(br.readLine());

   // code
   // nge begin
   int[] nge = new int[arr.length];

   Stack<Integer> st = new Stack<>();
   st.push(arr.length - 1);
   nge[arr.length - 1] = arr.length;

   for(int i = arr.length - 2; i >= 0; i--){
     while(st.size() > 0 && arr[i] >= arr[st.peek()]){
       st.pop();
     }

     if(st.size() == 0){
       nge[i] = arr.length;
     } else {
       nge[i] = st.peek();
     }

     st.push(i);
   }

   // nge end

   int j = 0;
   for(int i = 0; i <= arr.length - k; i++){
     if(j < i){
       j = i;
     }

     while(nge[j] < i + k){
       j = nge[j];
     }
```

```
      System.out.println(arr[j]);
    }
  }
```

KADANE'S ALGORITHM :- (Maximum Sum Subarray)



Approach 1:-

```
 static void maxSubArraySum(int a[], int size)
    {
        int max_so_far = Integer.MIN_VALUE,
        max_ending_here = 0,start = 0,
        end = 0, s = 0;

        for (int i = 0; i < size; i++)
        {
            max_ending_here += a[i];

            if (max_so_far < max_ending_here)
            {
                max_so_far = max_ending_here;
                start = s;
                end = i;
            }

            if (max_ending_here < 0)
            {
                max_ending_here = 0;
                s = i + 1;
            }
        }
        System.out.println("Maximum contiguous sum is "
                           + max_so_far);
        System.out.println("Starting index " + start);
        System.out.println("Ending index " + end);
    }
```

Count Subarrays with sum equal K (Ref)

An efficient solution is while traversing the array, store sum so far in currsum. Also, maintain the count of different values of currsum in a map. If the value of currsum is equal to the desired sum at any instance increment count of subarrays by one. The value of currsum exceeds the desired sum by currsum – sum. If this value is removed from currsum then the desired sum can be obtained. From the map find the number of subarrays previously found having sum equal to currsum-sum. Excluding all those subarrays from the current subarray, gives new subarrays having the desired sum. So increase count by the number of such subarrays. Note that when currsum is equal to the desired sum then also check the number of subarrays previously having a sum equal to 0. Excluding those subarrays from the current subarray gives new subarrays having the desired sum. Increase count by the number of subarrays having sum 0 in that case.

```
 static int findSubarraySum(int arr[], int n, int sum)
    {
        // HashMap to store number of subarrays
        // starting from index zero having
        // particular value of sum.
        HashMap<Integer, Integer> prevSum = new HashMap<>();

        int res = 0;

        // Sum of elements so far.
        int currsum = 0;

        for (int i = 0; i < n; i++) {

            // Add current element to sum so far.
            currsum += arr[i];

            // If currsum is equal to desired sum,
            // then a new subarray is found. So
            // increase count of subarrays.
            if (currsum == sum)
                res++;

            // currsum exceeds given sum by currsum
            //  - sum. Find number of subarrays having
            // this sum and exclude those subarrays
            // from currsum by increasing count by
            // same amount.
            if (prevSum.containsKey(currsum - sum))
                res += prevSum.get(currsum - sum);

            // Add currsum value to count of
            // different values of sum.
            Integer count = prevSum.get(currsum);
            if (count == null)
```

```
                prevSum.put(currsum, 1);
            else
                prevSum.put(currsum, count + 1);
        }

        return res;
    }
```

Find the length of largest subarray with 0 sum

prefix(i) = arr[0] + arr[1] +…+ arr[i]
prefix(j) = arr[0] + arr[1] +…+ arr[j], j>i
if prefix(i) == prefix(j) then prefix(j) – prefix(i) = 0 that means arr[i+1] + .. + arr[j] = 0, So a sub-array has zero sum , and the length of that sub-array is j-i+1

Algorithm:

1. Create an extra space, an array of *length n (prefix)*, *a variable (sum)*, *length (max_len)*, and *a hash map (hm)* to store the sum-index pair as a key-value pair.
2. Move along the input array from the start to the end.
3. For every index, update the value of *sum = sum + array[i].*
4. Check every index, if the current sum is present in the hash map or not.
5. If present, update the value of *max_len* to a maximum difference of two indices (current index and index in the hash-map) and *max_len.*
6. Else, put the value *(sum)* in the hash map, with the index as a key-value pair.
7. Print the maximum length *(max_len).*

```
static int maxLen(int arr[])
    {
        // Creates an empty hashMap hM
        HashMap<Integer, Integer> hM = new HashMap<Integer, Integer>();

        int sum = 0; // Initialize sum of elements
        int max_len = 0; // Initialize result

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++) {
            // Add current element to sum
            sum += arr[i];

            if (arr[i] == 0 && max_len == 0)
                max_len = 1;

            if (sum == 0)
                max_len = i + 1;

            // Look this sum in hash table
            Integer prev_i = hM.get(sum);

            // If this sum is seen before, then update max_len
            // if required
            if (prev_i != null)
                max_len = Math.max(max_len, i - prev_i);
            else // Else put this sum in hash table
                hM.put(sum, i);
        }

        return max_len;
    }
```

Minimum number of swaps required to sort an array

Given an array of **n** distinct elements, find the minimum number of swaps required to sort the array.
Examples:

```
Input: {4, 3, 2, 1}

Output: 2

Explanation: Swap index 0 with 3 and 1 with 2 to

              form the sorted array {1, 2, 3, 4}.


Input: {1, 5, 4, 3, 2}

Output: 2
```

Approach 1 :- O(nlogn) O(n)

```
 public static int minSwaps(int[] nums)
    {
        int len = nums.length;
        HashMap<Integer, Integer> map = new HashMap<>();
        for(int i=0;i<len;i++)
            map.put(nums[i], i);

        Arrays.sort(nums);

          // To keep track of visited elements. Initialize
        // all elements as not visited or false.
        boolean[] visited = new boolean[len];
        Arrays.fill(visited, false);

          // Initialize result
        int ans = 0;
        for(int i=0;i<len;i++) {

              // already swapped and corrected or
            // already present at correct pos
            if(visited[i] || map.get(nums[i]) == i)
                continue;

            int j = i, cycle_size = 0;
            while(!visited[j]) {
                visited[j] = true;

                  // move to next node
                j = map.get(nums[j]);
                cycle_size++;
            }
```

```
                // Update answer by adding current cycle.
            if(cycle_size > 0) {
                ans += (cycle_size - 1);
            }
        }
        return ans;
    }
```

Approach 2 :-

We can still improve the complexity by using a hashmap. The main operation here is the indexOf method inside the loop, which costs us n*n. We can improve this section to O(n), by using a hashmap to store the indexes. Still, we use the sort method, so the complexity cannot improve beyond O(n Log n)

Method using HashMap:
Same as before, make a new array (called temp), which is the sorted form of the input array. We know that we need to transform the input array to the new array (temp) in the minimum number of swaps. Make a map that stores the elements and their corresponding index, of the input array.

So at each i starting from 0 to N in the given array, where N is the size of the array:

1. If i is not in its correct position according to the sorted array, then

2. We will fill this position with the correct element from the hashmap we built earlier. We know the correct element which should come here is temp[i], so we look up the index of this element from the hashmap.

3. After swapping the required elements, we update the content of the hashmap accordingly, as temp[i] to the ith position, and arr[i] to where temp[i] was earlier.

Below is the implementation of the above approach:

```
 // Return the minimum number
    // of swaps required to sort the array
    public int minSwaps(int[] arr, int N)
    {

        int ans = 0;
        int[] temp = Arrays.copyOfRange(arr, 0, N);

        // Hashmap which stores the
        // indexes of the input array
        HashMap<Integer, Integer> h
            = new HashMap<Integer, Integer>();

        Arrays.sort(temp);
        for (int i = 0; i < N; i++)
        {
            h.put(arr[i], i);
        }
        for (int i = 0; i < N; i++)
        {

            // This is checking whether
            // the current element is
            // at the right place or not
            if (arr[i] != temp[i])
            {
                ans++;
                int init = arr[i];

                // If not, swap this element
                // with the index of the
                // element which should come here
                swap(arr, i, h.get(temp[i]));

                // Update the indexes in
                // the hashmap accordingly
                h.put(init, h.get(temp[i]));
                h.put(temp[i], i);
            }
        }
        return ans;
    }
```

Without sorting : -array consisting of consecutive distinct integers https://sites.google.com/view/minswaps

TREES :

GFG Questions :- (Binary tree)

Introduction :- 1,2,3,6,7,8,,11,12,17

Traversals :- 1-7,12-17,28 → Construction :- 1-3,5,20,26,27,28,35

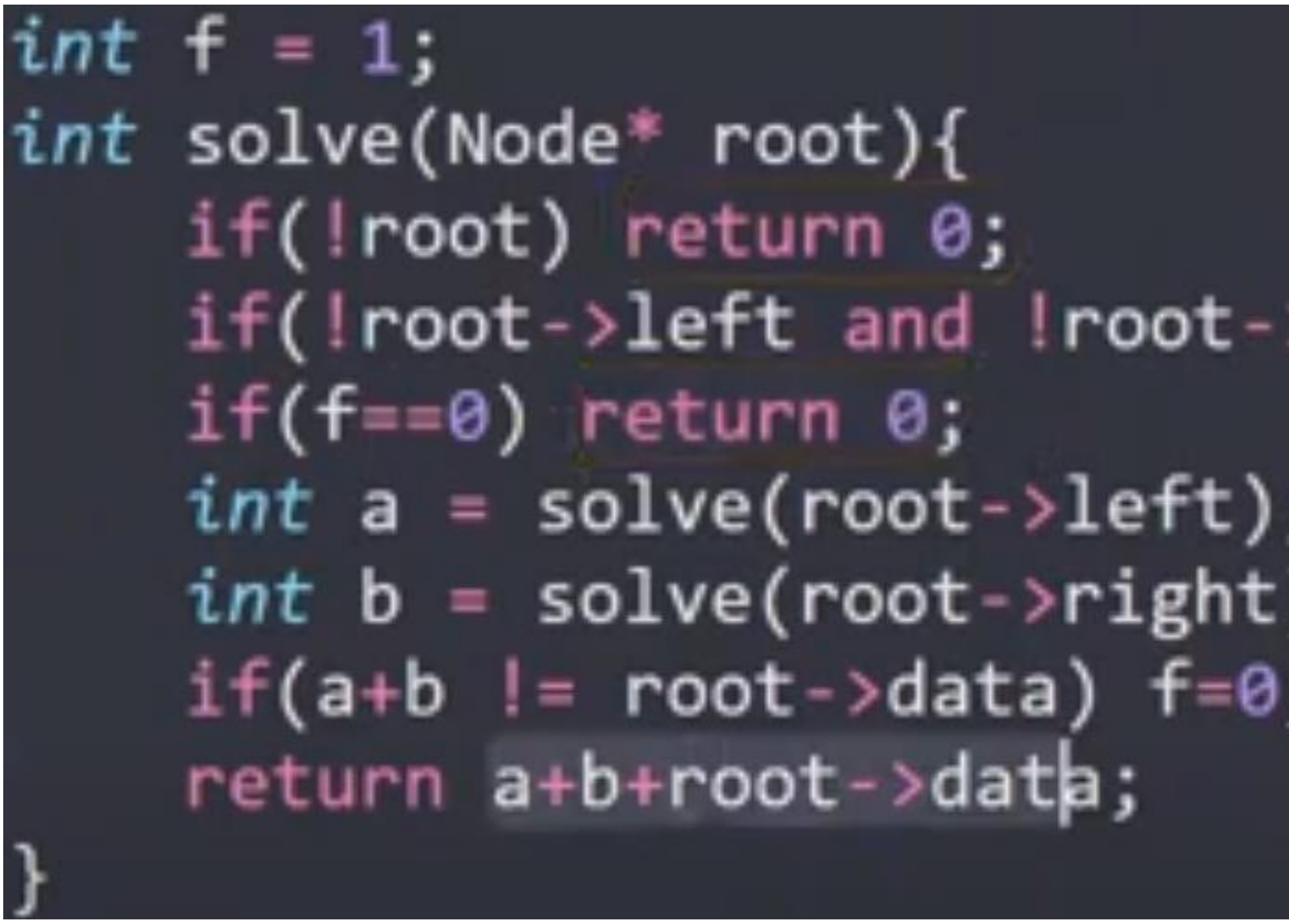
Checking & Printing :- 1,2,4,5,8,9,16,19,20,21,22,23,24,25,27,28,29,30,31,32,39,30,41,42

Summation :- 1-5,7,8,10,14,15,

LCA :- 1-3(Best approach either of 1), 4,5,10,11

Misc :- 5,7,8,10,1112,13,17,20,21,27,28,29,32,37,55,76

Check if Sumtree :-



```
int f = 1;
int solve(Node* root){
    if(!root) return 0;
    if(!root->left and !root-
    if(f==0) return 0;
    int a = solve(root->left)
    int b = solve(root->right
    if(a+b != root->data) f=0
    return a+b+root->data;
}
```

GFG Questions :- (Binary Search tree)

Basic 1-3,

Construction and conversion :- 1-6,8,10-12,17

Checking searching :- 1,4,5,6,9,10,11,12,16,17,23,30,31,32,33,41

Red Black Tree and Threaded Tree :- High Level 1-4,7

Misc :- 2,4,8,9

GFG Questions :- (Graphs)

Introduction : 1-8,10,12,15,20,28,29,39,42,48

Graph cycle : 1,2,5,6,9,13,14

Topological Sorting :- 1,2,6

Minimum spanning tree :- 1-4,6,9

BackTracking :- 1,2,3,4,5

Shortest Paths :- 1,2,3,4,6,7,9,12,18,

Connectivity :- 1,2,6,11,12,15,17,22,23

Hard Problems : 3,4,

GFG Questions :- (Heap)

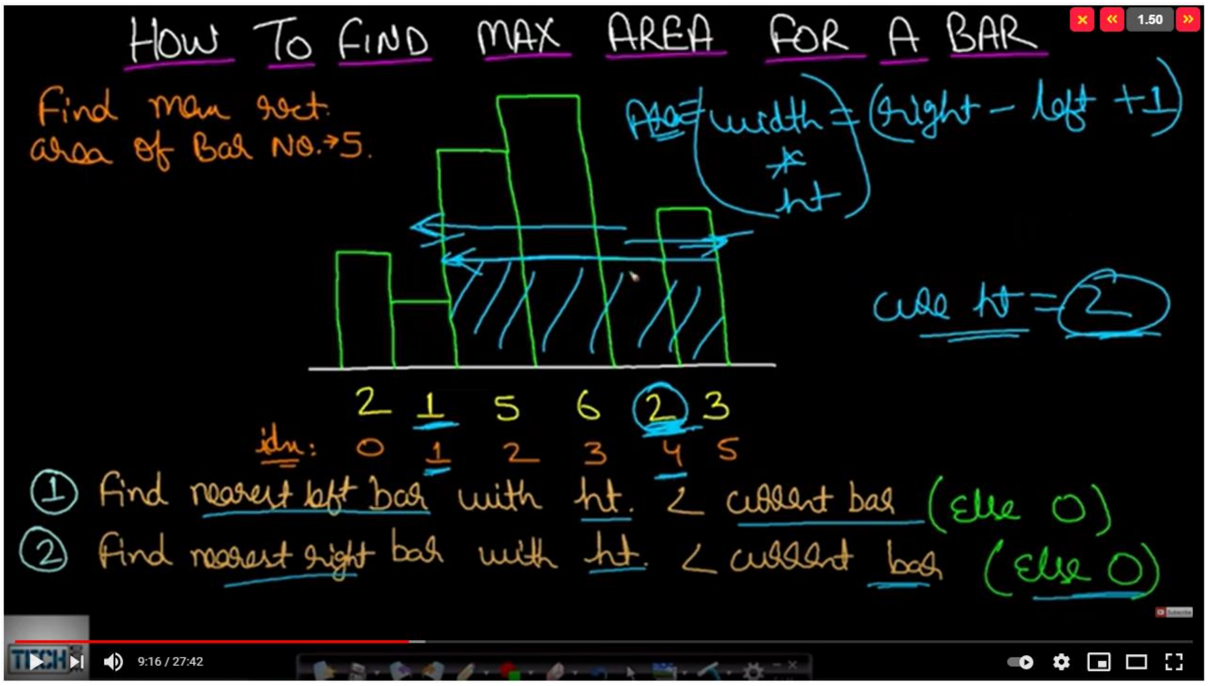
1,2,3,4,5,8,10,11,12,13,14,18,32,34,38,46,47,48

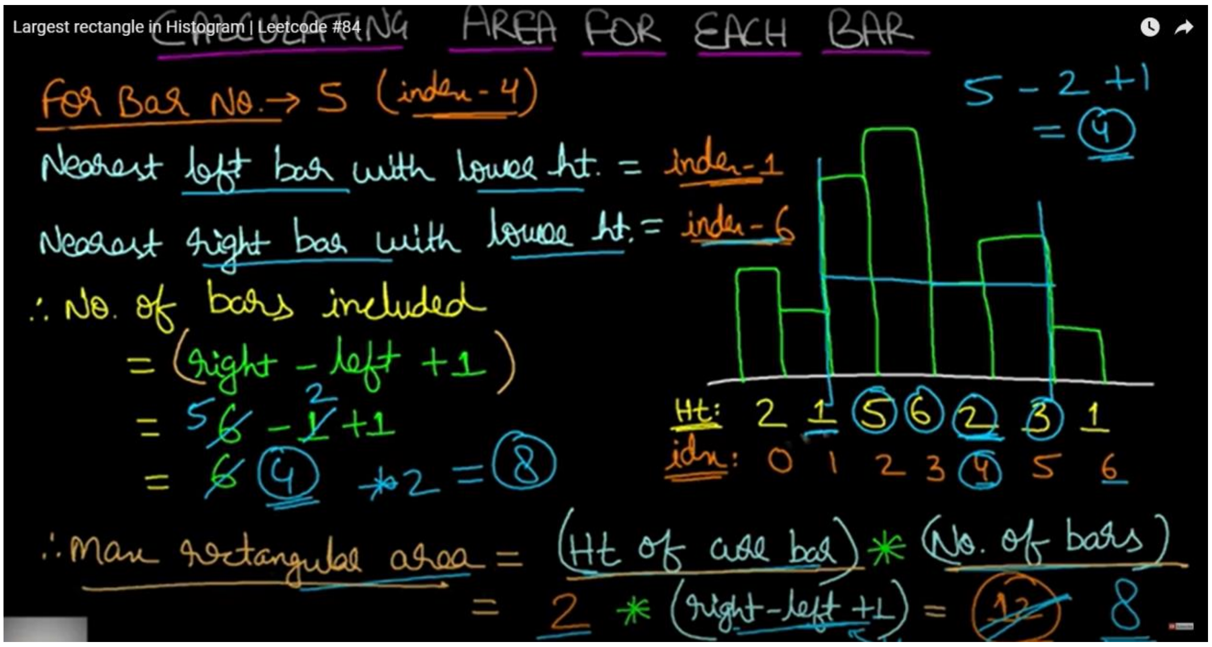
GFG Questions :- (Matrix)

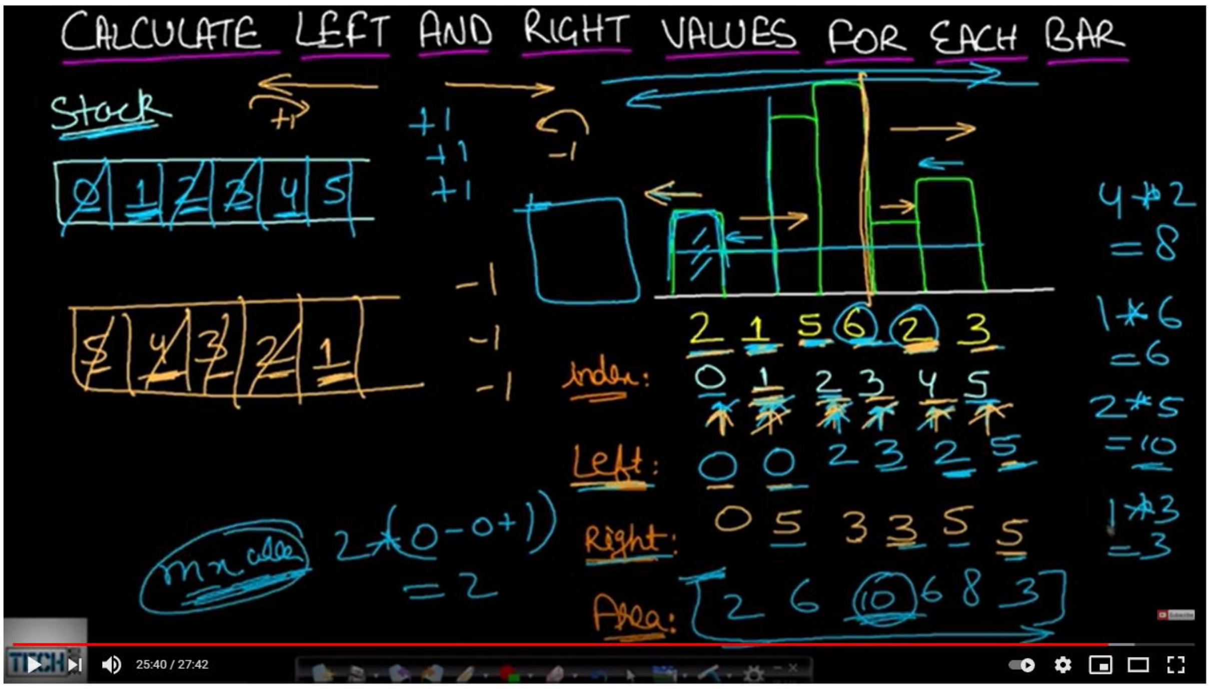
Luv Babbar sheet DSE

[Maximum Size Rectangle in histogram](https://www.geeksforgeeks.org/largest-rectangle-under-histogram/)

https://www.geeksforgeeks.org/largest-rectangle-under-histogram/


How To Find MAX AREA FOR A BAR
Find max rect. area of Bar No.→5.
Area = width * ht
width = (right − left + 1)
curr ht = 2
2 1 5 6 2 3
idx: 0 1 2 3 4 5
① Find nearest left bar with ht. < current bar (else 0)
② Find nearest right bar with ht. < current bar (else 0)
1.50
9:16 / 27:42


Largest rectangle in Histogram | Leetcode #84
CALCULATING AREA FOR EACH BAR
For Bar No. → 5 (index - 4)
Nearest left bar with lower ht. = index - 1
Nearest right bar with lower ht. = index - 6
∴ No. of bars included
= (right - left + 1)
= 5 6 - 1 2 + 1
= 6 4 * 2 = 8
∴ Max rectangular area = (Ht of cur bar) * (No. of bars)
= 2 * (right - left + 1) = 12 8
5 - 2 + 1
= 4
Ht: 2 1 5 6 2 3 1
idx: 0 1 2 3 4 5 6



```
public class LargestRectangle {
    public static void main(String[] args) {
        int[] input = {2, 1, 5, 6, 2, 3};
        System.out.println("Max rectangle area: " + largestRectangleArea(input));
    }

    public static int largestRectangleArea(int[] heights) {
        int arraySize =  heights.length;
        int[] left = new int[arraySize];
        int[] right = new int[arraySize];
        Stack<Integer> stack = new Stack<>();

        for (int i =0; i < arraySize; i++) {
            if (stack.isEmpty()) {
                left[i] = 0;
            } else {
                while (!stack.empty() && heights[stack.peek()] >= heights[i]) {
                    stack.pop();
                }
                left[i] = stack.empty() ? 0 : stack.peek() + 1;
            }
            stack.push(i);
        }

        stack.clear();

        for (int i = arraySize-1; i >= 0; i--) {
            if (stack.empty()) {
                right[i] = arraySize - 1;
            } else {
                while (!stack.isEmpty() && heights[stack.peek()] >= heights[i]){
                    stack.pop();
                }
                right[i] = stack.isEmpty() ? arraySize - 1 : stack.peek() - 1;
            }
            stack.push(i);
        }

        int maxArea = 0;
        for (int i =0; i < arraySize; i++) {
            int area = heights[i] * (right[i] - left[i] + 1);
            maxArea = Math.max(maxArea, area);
        }
        return maxArea;
    }
}
```

Maximum Rectangle in matrix :-

Uses the above idea only, for each row in the matrix and set of rows in the matrix check for histogram above logic and get maximum rectangle size.

```
class Solution {


  // Get the maximum area in a histogram given its heights

  public int leetcode84(int[] heights) {

    Stack < Integer > stack = new Stack < > ();

    stack.push(-1);

    int maxarea = 0;

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

      while (stack.peek() != -1 && heights[stack.peek()] >= heights[i])

        maxarea = Math.max(maxarea, heights[stack.pop()] * (i - stack.peek() - 1));

      stack.push(i);

    }
```

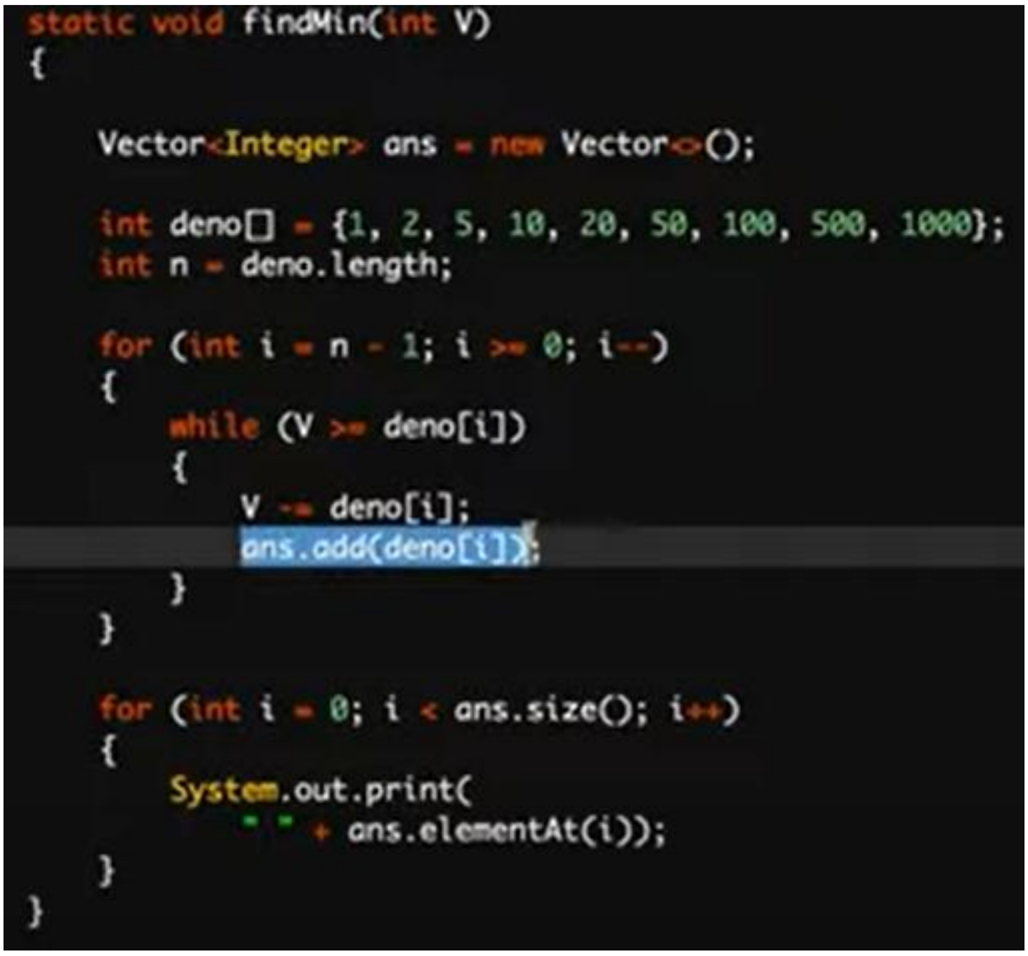
```
        while (stack.peek() != -1)
            maxarea = Math.max(maxarea, heights[stack.pop()] * (heights.length - stack.peek() -1));
        return maxarea;
    }

    public int maximalRectangle(char[][] matrix) {

        if (matrix.length == 0) return 0;
        int maxarea = 0;
        int[] dp = new int[matrix[0].length];

        for(int i = 0; i < matrix.length; i++) {
            for(int j = 0; j < matrix[0].length; j++) {

                // update the state of this row's histogram using the last row's histogram
                // by keeping track of the number of consecutive ones

                dp[j] = matrix[i][j] == '1' ? dp[j] + 1 : 0;
            }
            // update maxarea with the maximum area from this row's histogram
            maxarea = Math.max(maxarea, leetcode84(dp));
        } return maxarea;
    }
}
```

Graph (GFG – Luv Babbar sheet)

Stack & Queues, LinkedList(GFG – Luv Babbar sheet)

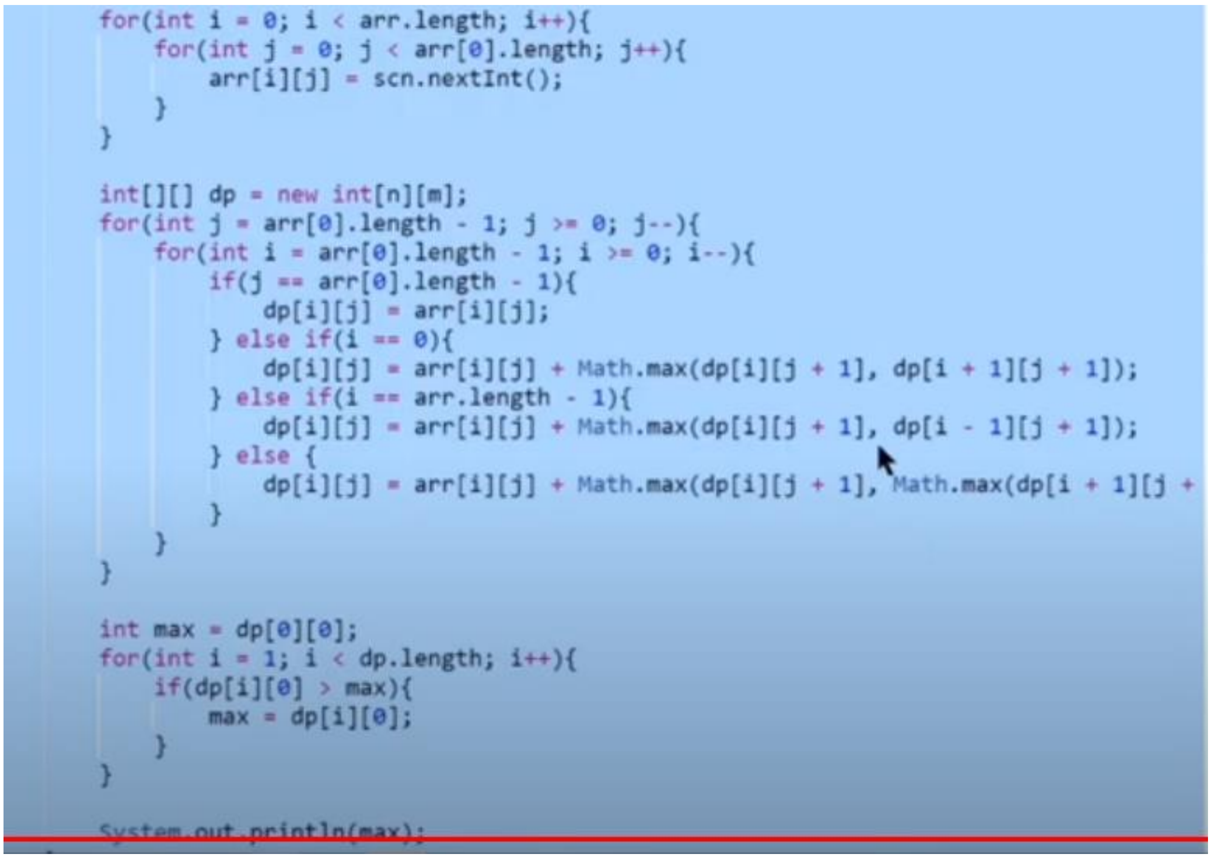
Greedy -> Abdul bari and Luv Babbar que.

1. Minimum coin required to make change



```java
static void findMin(int V)
{

    Vector<Integer> ans = new Vector<>();

    int deno[] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
    int n = deno.length;

    for (int i = n - 1; i >= 0; i--)
    {
        while (V >= deno[i])
        {
            V -= deno[i];
            ans.add(deno[i]);
        }
    }

    for (int i = 0; i < ans.size(); i++)
    {
        System.out.print(
            " " + ans.elementAt(i));
    }
}
```

Gold Mine problem :-



```
for(int i = 0; i < arr.length; i++){
    for(int j = 0; j < arr[0].length; j++){
        arr[i][j] = scn.nextInt();
    }
}

int[][] dp = new int[n][m];
for(int j = arr[0].length - 1; j >= 0; j--){
    for(int i = arr[0].length - 1; i >= 0; i--){
        if(j == arr[0].length - 1){
            dp[i][j] = arr[i][j];
        } else if(i == 0){
            dp[i][j] = arr[i][j] + Math.max(dp[i][j + 1], dp[i + 1][j + 1]);
        } else if(i == arr.length - 1){
            dp[i][j] = arr[i][j] + Math.max(dp[i][j + 1], dp[i - 1][j + 1]);
        } else {
            dp[i][j] = arr[i][j] + Math.max(dp[i][j + 1], Math.max(dp[i + 1][j +
        }
    }
}

int max = dp[0][0];
for(int i = 1; i < dp.length; i++){
    if(dp[i][0] > max){
        max = dp[i][0];
    }
}

System.out.println(max);
```