**EASY**

1. *Count Factors*

**Problem Description**

Given an integer **A**, you need to find the count of it's factors.  
Factor of a number is the number which divides it perfectly leaving no remainder.  
Example : 1, 2, 3, 6 are factors of 6

**Problem Constraints**

1 <= A <= 109  
**Input Format**

First and only argument is an integer A.  
**Output Format**

Return the count of factors of A.  
**Example Input**

Input 1:  
5

Input 2:

10  
**Example Output**

Output 1:  
2

Output 2:  
4

public class Solution {

public int solve(int A) {

int count = 0;

int num = (int) Math.sqrt(A);

for(int i=1;i<=num;i++){

if(A%i==0){

if(i!=A/i){

count+=2;

}else{

count+=1;

}

}

}

return count;

}

}

­­­

1. *Find Perfect Numbers*  
     
   **Input Format**

First and only argument contains a single positive integer **A**.

**Output Format**

Return 1 if A is a perfect number and 0 otherwise.  
  
**Example Input**

Input 1:

A = 4

Input 2:

A = 6

**Example Output**

Output 1:

0

Output 2:

1

public class Solution {

public int solve(int A) {

if (A==1){

return 0;

}

int num = (int) Math.sqrt(A);

int sum = 1;

for (int i = 2; i <= num; i++) {

if (A % i == 0) {

int div = A / i;

sum = sum + i + div;

}

}

return A == sum ? 1 : 0;

}

}

1. *Two number Sum*

**Problem Description**

Given an array A and an integer B. A pair(i, j) in the array is a good pair if i != j and (A[i] + A[j] == B). Return the indices if present.

// O(N) space | O(N) time

public int[] twoNumberSum(int[] array, int target) {

Set<Integer> nums = new HashSet<>();

for (int num : array) {

int match = target - num;

if (nums.contains(match)) {

return new int[] { match, num };

} else {

nums.add(num);

}

}

return new int[0];

}

1. *Reverse in a range*

**Problem Description**

Given an array **A** of **N** integers and also given two integers **B** and **C**. Reverse the elements of the array **A** within the given inclusive range **[B, C].**  
  
**Problem Constraints**

1 <= N <= 105  
1 <= A[i] <= 109  
0 <= B <= C <= N - 1  
  
**Input Format**

The first argument A is an array of integer.  
The second and third arguments are integers B and C  
**Output Format**

Return the array A after reversing in the given range.  
**Example Input**

Input 1:

A = [1, 2, 3, 4]

B = 2

C = 3

**Input 2:**

A = [2, 5, 6]

B = 0

C = 2  
**Example Output**

Output 1:

[1, 2, 4, 3]

Output 2:

[6, 5, 2]

public class Solution {

public int[] solve(int[] A, int B, int C) {

int i = B;

int j = C;

while (i < j) {

int t = A[i];

A[i] = A[j];

A[j] = t;

i++;

j--;

}

return A;

}

}

1. *Array Rotation*

**Problem Description**

Given an integer array **A** of size **N** and an integer **B**, you have to return the same array after rotating it **B** times towards the right.

**Problem Constraints**

1 <= N <= 105  
1 <= A[i] <=109  
1 <= B <= 109

**Input Format**

The first argument given is the integer array A.  
The second argument given is the integer B.

**Output Format**

Return the array A after rotating it B times to the right

**Example Input**

Input 1:

A = [1, 2, 3, 4]

B = 2

Input 2:

A = [2, 5, 6]

B = 1

**Example Output**

Output 1:

[3, 4, 1, 2]

Output 2:

[6, 2, 5]

public class Solution {

public int[] rotate(int[]A, int B, int C){

while(B<C){

int t = A[B];

A[B] = A[C];

A[C] = t;

B++;

C--;

}

return A;

}

public int[] solve(int[] A, int B) {

int len = A.length;

if(B>len) {

B = B%len;

}

A = rotate(A,0,A.length-1);

A = rotate(A,0,B-1);

A = rotate(A,B,A.length-1);

return A;

}

}

1. *MIN MAX sum in an array*

**Problem Description**

Given an array **A** of size **N**. You need to find the sum of **Maximum and Minimum element** in the given array.  
  
**Problem Constraints**

1 <= N <= 105  
-109 <= A[i] <= 109  
**Input Format**

First argument A is an integer array.  
**Output Format**

Return the sum of maximum and minimum element of the array  
**Example Input**

Input 1:

A = [-2, 1, -4, 5, 3]

Input 2:

A = [1, 3, 4, 1]  
**Example Output**

Output 1:

1

Output 2:

5

public class Solution {

public int solve(int[] A) {

int max = Integer.MIN\_VALUE;

int min = Integer.MAX\_VALUE;

for (int i : A) {

if (i > max) {

max = i;

}

if (i < min) {

min = i;

}

}

return max + min;

}

}

1. *Range Sum Query*

**Problem Description**

You are given an integer array **A** of length **N**.  
You are also given a 2D integer array **B** with dimensions **M x 2**, where each row denotes a [L, R] query.  
For each query, you have to find the sum of all elements from L to R indices in A (0 - indexed).  
More formally, find A[L] + A[L + 1] + A[L + 2] +... + A[R - 1] + A[R] for each query.  
**Problem Constraints**

1 <= N, M <= 105  
1 <= A[i] <= 109  
0 <= L <= R < N  
**Input Format**

The first argument is the integer array A.  
The second argument is the 2D integer array B.  
**Output Format**

Return an integer array of length M where ith element is the answer for ith query in B.

**Example Input**

Input 1:

A = [1, 2, 3, 4, 5]

B = [[0, 3], [1, 2]]

Input 2:

A = [2, 2, 2]

B = [[0, 0], [1, 2]]

**Example Output**

Output 1:

[10, 5]

Output 2:

[2, 4]

TC:O(N+M) ||SC:O(1)

public class Solution {

public long[] rangeSum(int[] A, int[][] B) {

long pf[] = new long[A.length];

pf[0] = A[0];

for (int i = 1; i < A.length; i++) {

pf[i] = pf[i - 1] + A[i];

}

long arr[] = new long[B.length];

for (int j = 0; j < B.length; j++) {

int l = B[j][0];

int r = B[j][1];

long sum = 0;

if (l == 0) {

sum += pf[r];

} else {

sum += pf[r] - pf[l - 1];

}

arr[j] = sum;

}

return arr;

}

}

1. *Maximum subarray*

You are given an integer array **C** of size **A**. Now you need to find a subarray (contiguous elements) so that the sum of contiguous elements is maximum.  
But the sum must not exceed **B**.

**Problem Constraints**

1 <= A <= 103  
1 <= B <= 109  
1 <= C[i] <= 106  
**Input Format**

The first argument is the integer A.  
The second argument is the integer B.  
The third argument is the integer array C.  
**Output Format**

Return a single integer which denotes the maximum sum.  
**Example Input**

Input 1:

A = 5

B = 12

C = [2, 1, 3, 4, 5]

Input 2:

A = 3

B = 1

C = [2, 2, 2]

**Example Output**

Output 1:

12

Output 2:

0

public int maxSubarray(int A, int B, int[] C) {

int tmax = Integer.MIN\_VALUE;

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

for (int j = i; j < A; j++) {

int sum = 0;

for (int k = i; k <= j; k++) {

sum += C[k];

if (sum <= B) {

tmax = Math.max(tmax, sum);

}

}

}

}

return tmax;

}

1. *Sum of all subarrays*

**Problem Description**

You are given an integer array **A** of length**N.**  
You have to find the sum of all subarray sums of A.  
More formally, a subarray is defined as a contiguous part of an array which we can obtain by deleting zero or more elements from either end of the array.  
A subarray sum denotes the sum of all the elements of that subarray.

**Note :**Be careful of integer overflow issues while calculations. Use appropriate datatypes.

**Problem Constraints**

1 <= N <= 105  
1 <= Ai <= 10 4  
  
**Input Format**

The first argument is the integer array A.  
  
**Output Format**

Return a single integer denoting the sum of all subarray sums of the given array.  
  
**Example Input**

Input 1:

A = [1, 2, 3]

Input 2:

A = [2, 1, 3]  
  
**Example Output**

Output 1:

20

Output 2:

19

TC:O(N) || SC:O(1)

public long subarraySum(int[] A) { //Contribution technique

int len = A.length;

long ans = 0;

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

ans = ans + (long) A[i] \* (i + 1) \* (len - i);

}

return ans;

}

1. *Subarray with given sum and length*

**Problem Description**

Given an array **A** of length **N**. Also given are integers **B** and **C**.

Return 1 if there exists a subarray with length **B** having sum **C** and 0 otherwise

**Problem Constraints**

1 <= N <= 105

1 <= A[i] <= 104

1 <= B <= N

1 <= C <= 109

**Input Format**

First argument A is an array of integers.

The remaining arguments B and C are integers  
  
**Output Format**

Return 1 if such a subarray exist and 0 otherwise  
  
**Example Input**

Input 1:

A = [4, 3, 2, 6, 1]

B = 3

C = 11

Input 2:

A = [4, 2, 2, 5, 1]

B = 4

C = 6  
  
**Example Output**

Output 1:

1

Output 2:

0

public int solve(int[] A, int B, int C) {

int i=0;

int j=i+B;

while(j<=A.length) {

int sum = sum(A,i,j);

if(sum == C) {

return 1;

}

i++;

j++;

}

return 0;

}

public int sum(int A[], int l, int r) {

int sum = 0;

for(int i=l;i<r;i++) {

sum+=A[i];

}

return sum;

}

1. *Minimum Swaps*

**Problem Description**

Given an array of integers **A** and an integer **B**, find and return the minimum number of swaps required to bring all the numbers less than or equal to **B** together.

**Note:** It is possible to swap any two elements, not necessarily consecutive.  
  
**Problem Constraints**

1 <= length of the array <= 100000  
-109 <= A[i], B <= 109  
**Input Format**

The first argument given is the integer array A.  
The second argument given is the integer B.  
  
**Output Format**

Return the minimum number of swaps.  
  
**Example Input**

Input 1:

A = [1, 12, 10, 3, 14, 10, 5]

B = 8

Input 2:

A = [5, 17, 100, 11]

B = 20

**Example Output**

Output 1:

2

Output 2:

1

public int solve(int[] A, int B) {

int minSwap = 0;

int window = countNumberLessThanEqualtoB(A, B);

if (window <= 1) {

return 0;

} else {

int l = 0, r = 0, count = 0;

while (r < window) {

if (A[r] > B) {

count++;

}

r++;

}

minSwap = count;

while (r < A.length) {

if (A[r] > B) {

count++;

}

if (A[l] > B) {

count--;

}

minSwap = Math.min(minSwap, count);

l++;

r++;

}

}

return minSwap;

}

public int countNumberLessThanEqualtoB(int[] A, int B) {

int count = 0;

for (int a : A) {

if (a <= B) {

count++;

}

}

return count;

}

1. *Column Sum of a 2D matrix*

**Problem Description**

You are given a 2D integer matrix A, return a 1D integer array containing column-wise sums of original matrix.  
  
**Problem Constraints**

1 <= A.size() <= 103

1 <= A[i].size() <= 103

1 <= A[i][j] <= 103  
  
**Input Format**

First argument is a 2D array of integers.(2D matrix).  
  
**Output Format**

Return an array containing column-wise sums of original matrix.

**Example Input**

Input 1:

[1,2,3,4]

[5,6,7,8]

[9,2,3,4]  
  
**Example Output**

Output 1:

{15,10,13,16}

public int[] solve(int[][] A) {

int arr[] = new int[A[0].length];

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

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

if (arr[j] == 0) {

arr[j] = A[i][j];

} else {

arr[j] += A[i][j];

}

}

}

return arr;

}

1. *Main diagonal sum of a 2D matrix*

**Problem Description**

You are given a **N X N** integer matrix. You have to find the sum of all the main diagonal elements of **A**.

Main diagonal of a matrix **A** is a collection of elements **A[i, j]** such that **i = j**.

**Problem Constraints**

1 <= **N** <= 103

-1000 <= **A[i][j]** <= 1000

**Input Format**

There is 1 line in the input. First 2 integers R, C are the number of rows and columns. Then R \* C integers follow corresponding to the row wise numbers in the 2D array **A**.  
  
**Output Format**

Return an integer denoting the sum of main diagonal elements.  
  
**Example Input**

Input 1:

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

Input 2:

2 2 3 2 2 3  
  
**Example Output**

Output 1:

15

Output 2:

6

public int solve(final int[][] A) {

int sum = 0;

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

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

if (i == j) {

sum += A[i][j];

}

}

}

return sum;

}

1. *Matrix Transpose*

**Problem Constraints**

1 <= A.size() <= 1000

1 <= A[i].size() <= 1000

1 <= A[i][j] <= 1000  
  
**Input Format**

First argument is a 2D matrix of integers.  
  
**Output Format**

You have to return the Transpose of this 2D matrix.  
  
**Example Input**

Input 1:

A = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]

Input 2:

A = [[1, 2],[1, 2],[1, 2]]  
  
**Example Output**

Output 1:

[[1, 4, 7], [2, 5, 8], [3, 6, 9]]

Output 2:

[[1, 1, 1], [2, 2, 2]]

public int[][] solve(int[][] A) {

int tx[][] = new int[A[0].length][A.length];

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

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

tx[i][j] = A[j][i];

}

}

return tx;

}

1. *Matrix Rotation*

**Problem Description**

You are given a n x n 2D matrix **A** representing an image.

Rotate the image by 90 degrees (clockwise).

You need to do this in place.

**Note:** If you end up using an additional array, you will only receive partial score.  
  
**Problem Constraints**

1 <= n <= 1000  
  
**Input Format**

First argument is a 2D matrix A of integers  
  
**Output Format**

Return the 2D rotated matrix.  
  
**Example Input**

Input 1:

[

[1, 2],

[3, 4]

]

Input 2:

[

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

**Example Output**

Output 1:

[

[3, 1],

[4, 2]

]

Output 2:

[

[7, 4, 1],

[8, 5, 2],

[9, 6, 3]

]

public void solve(int[][] A) {

int temp[][] = transpose(A);

int rotate[][] = reverseRow(temp);

}

public int[][] transpose(int[][] A) {

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

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

int temp = A[i][j];

A[i][j] = A[j][i];

A[j][i] = temp;

}

}

return A;

}

public int[][] reverseRow(int[][] A) {

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

for (int j = 0; j < A[i].length / 2; j++) {

int temp = A[i][j];

A[i][j] = A[i][A.length - 1 - j];

A[i][A.length - 1 - j] = temp;

}

}

return A;

}

1. *Find all anti diagonals of a 2D matrix*

**Problem Description**

Give a **N \* N** square matrix **A**, return an array of its anti-diagonals. Look at the example for more details.  
  
**Problem Constraints**

1<= **N** <= 1000  
1<= **A[i][j]** <= 1e9  
  
**Input Format**

Only argument is a 2D array **A** of size **N** **\*** **N**.  
  
**Output Format**

Return a 2D integer array of size (2 \* **N**-1) **\*** **N**, representing the anti-diagonals of input array **A**.  
The vacant spaces in the grid should be assigned to 0.  
  
**Example Input**

Input 1:

1 2 3

4 5 6

7 8 9

Input 2:

1 2

3 4

**Example Output**

Output 1:

1 0 0

2 4 0

3 5 7

6 8 0

9 0 0

Output 2:

1 0

2 3

4 0

public int[][] diagonal(int[][] A) {

int totalRows = A[0].length + A.length - 1;

int arr[][] = new int[totalRows][A[0].length];

int ROW = 0;

// looping over the 1st row

for (int col = 0; col < A[0].length; col++) {

int r = 0, c = col, i = 0;

while (r < A.length && c >= 0) {

arr[ROW][i] = A[r][c];

r++;

c--;

i++;

}

ROW++;

}

// looping over the last column

for (int row = 1; row < A.length; row++) {

int r = row, c = A[0].length - 1, i = 0;

while (r < A.length && c >= 0) {

arr[ROW][i] = A[r][c];

r++;

c--;

i++;

}

ROW++;

}

return arr;

}

1. *Elements Removal*

**Problem Description**

Given an integer array **A** of size **N**. You can **remove** any element from the array in one operation.  
The cost of this operation is the **sum of all elements** in the array present **before this operation**.

Find the **minimum cost** to remove all elements from the array.

**Problem Constraints**

0 <= N <= 1000  
1 <= A[i] <= 103  
  
**Input Format**

First and only argument is an integer array A.  
**Output Format**

Return an integer denoting the total cost of removing all elements from the array.  
**Example Input**

Input 1:

A = [2, 1]

Input 2:

A = [5]  
  
**Example Output**

Output 1:

4

Output 2:

5

public int solve(int[] A) {

int sum=0;

Arrays.sort(A);

int ps[] = new int[A.length];

ps[0] = A[0];

for(int i=1;i<A.length;i++) {

ps[i] = ps[i-1]+A[i];

}

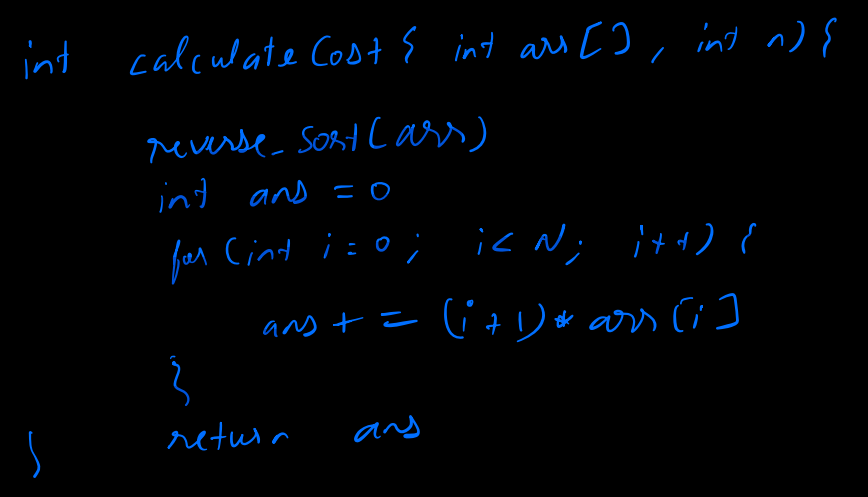
for(int j=0;j<ps.length;j++) {

sum+=ps[j];

}

return sum;

}



1. *Nobel Integer*

**Problem Description**

Given an integer array **A**, find if an integer **p** exists in the array such that the number of integers greater than **p** in the array equals **p**.  
**Problem Constraints**

1 <= |A| <= 2\*105  
-108 <= A[i] <= 108  
**Input Format**

First and only argument is an integer array A.  
**Output Format**

Return 1 if any such integer p is present else, return -1.  
**Example Input**

Input 1:

A = [3, 2, 1, 3]

Input 2:

A = [1, 1, 3, 3]

**Example Output**

Output 1:

1

Output 2:

-1

public int solve(int[] A) {

Arrays.sort(A);

int len = A.length;

for (int i = 0; i < len-1; i++) {

if (A[i] == len - i - 1 && A[i] != A[i + 1]) {

return 1;

}

}

if(A[len-1] == 0) {

return 1;

}

return -1;

}

1. *Search in a row wise and column wise sorted matrix*

**Problem Description**

Given a matrix of integers **A** of size **N x M** and an integer **B**.

In the given matrix every row and column is sorted in non-decreasing order. Find and return the position of **B** in the matrix in the given form:

* If A[i][j] = B then return (i \* 1009 + j)
* If B is not present return -1.

**Note 1:** Rows are numbered from top to bottom and columns are numbered from left to right.  
**Note 2:** If there are multiple B in A then return the smallest value of i\*1009 +j such that A[i][j]=B.  
**Note 3:** Expected time complexity is linear  
**Note 4:** Use 1-based indexing

**Problem Constraints**

1 <= N, M <= 1000  
-100000 <= A[i] <= 100000  
-100000 <= B <= 100000  
**Input Format**

The first argument given is the integer matrix A.  
The second argument given is the integer B.  
**Output Format**

Return the position of B and if it is not present in A return -1 instead.  
**Example Input**

Input 1:-

A = [[1, 2, 3]

[4, 5, 6]

[7, 8, 9]]

B = 2

Input 2:-

A = [[1, 2]

[3, 3]]

B = 3  
**Example Output**

Output 1:-

1011

Output 2:-

2019

public int solve(int[][] A, int B) { //TC:O(N+M) || SC:O(1)

int r = 0, c = A[0].length - 1;

int val = -1;

while (r < A.length && c >= 0) {

if (B < A[r][c]) {

c--;

} else if (B > A[r][c]) {

r++;

} else if (B == A[r][c]) {

int tVal = (r + 1) \* 1009 + (c + 1);

val = val == -1 ? tVal : Math.min(tVal, val);

c--;

}

}

return val;

}

1. *Spiral Order Matrix*

**Problem Description**

Given an integer **A**, generate a square matrix filled with elements from 1 to A2 in spiral order and return the generated square matrix.  
  
**Problem Constraints**

1 <= **A** <= 1000  
  
**Input Format**

First and only argument is integer A  
  
**Output Format**

Return a 2-D matrix which consists of the elements added in spiral order.  
  
**Example Input**

Input 1:

1

Input 2:

2

Input 3:

5  
  
**Example Output**

Output 1:

[ [1] ]

Output 2:

[ [1, 2],   
 [4, 3] ]

Output 2:

[ [1, 2, 3, 4, 5],   
 [16, 17, 18, 19, 6],   
 [15, 24, 25, 20, 7],   
 [14, 23, 22, 21, 8],   
 [13, 12, 11, 10, 9] ]

public int[][] generateMatrix(int A) { // TC:O(A^2) || SC:O(1)

int val = 1;

int r = 0, c = 0;

int arr[][] = new int[A][A];

while (A > 1) {

for (int i = 0; i < A - 1; i++) {

arr[r][c] = val++;

c++;

}

for (int i = 0; i < A - 1; i++) {

arr[r][c] = val++;

r++;

}

for (int i = 0; i < A - 1; i++) {

arr[r][c] = val++;

c--;

}

for (int i = 0; i < A - 1; i++) {

arr[r][c] = val++;

r--;

}

r++;

c++;

A -= 2;

}

if (A == 1) {

arr[r][c] = val;

}

return arr;

}

1. *Row with maximum number of ones*

**Problem Description**

Given a binary sorted matrix **A** of size **N x N**. Find the row with the **maximum** number of **1**.

**NOTE:**

* If two rows have the maximum number of 1 then return the row which has a **lower index**.
* Rows are numbered from top to bottom and columns are numbered from left to right.
* Assume **0-based** indexing.
* Assume each row to be sorted by values.
* Expected time complexity is O(rows + columns).

**Problem Constraints**

1 <= N <= 1000

0 <= A[i] <= 1  
  
**Input Format**

The only argument given is the integer matrix A.  
  
**Output Format**

Return the row with the maximum number of 1.  
  
**Example Input**

Input 1:

A = [ [0, 1, 1]

[0, 0, 1]

[0, 1, 1] ]

Input 2:

A = [ [0, 0, 0, 0]

[0, 0, 0, 1]

[0, 0, 1, 1]

[0, 1, 1, 1] ]  
  
**Example Output**

Output 1:

0

Output 2:

3

public int solve(int[][] A) { // TC: O(N+M) || SC:O(1)

int row = 0, col = A[0].length - 1;

int ans = 0;

while (row < A.length && col >= 0) {

while (col >= 0 && A[row][col] == 1) {

col--;

ans = row;

}

row++;

}

return ans;

}

1. *Minor Diagonal Sum*

**Problem Description**

You are given a **N X N** integer matrix. You have to find the sum of all the minor diagonal elements of **A**.

Minor diagonal of a **M X M** matrix **A** is a collection of elements **A[i, j]** such that **i + j = M + 1** (where **i, j** are 1-based).  
**Problem Constraints**

1 <= **N** <= 103

-1000 <= **A[i][j]** <= 1000  
**Input Format**

First and only argument is a 2D integer matrix **A**.  
**Output Format**

Return an integer denoting the sum of minor diagonal elements.  
  
**Example Input**

Input 1:

A = [[1, -2, -3],

[-4, 5, -6],

[-7, -8, 9]]

Input 2:

A = [[3, 2],

[2, 3]]  
**Example Output**

Output 1:

-5

Output 2:

4

public int solve(final int[][] A) { TC:O(N^2) || SC: O(1)

int M = A.length;

int sum = 0;

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

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

if ((i + 1) + (j + 1) == M + 1) {

sum += A[i][j];

}

}

}

return sum;

}

1. *Row Sum*

**Problem Constraints**

1 <= A.size() <= 103

1 <= A[i].size() <= 103

1 <= A[i][j] <= 103  
  
**Input Format**

First argument A is a 2D array of integers.(2D matrix).  
  
**Output Format**

Return an array containing row-wise sums of original matrix.  
**Example Input**

Input 1:

[1,2,3,4]

[5,6,7,8]

[9,2,3,4]  
**Example Output**

Output 1:

[10,26,18]

public int[] solve(int[][] A) { TC:O(N^2) || SC:O(1)

int arr[] = new int[A.length];

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

int t = 0;

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

t += A[i][j];

}

arr[i] = t;

}

return arr;

}

1. *Add the matrices*

**Problem Description**

You are given two matrices A & B of same size, you have to return another matrix which is the sum of A and B.  
**Note:** Matrices are of same size means the number of **rows** and number of **columns** of both matrices are equal.

**Problem Constraints**

1 <= A.size(), B.size() <= 1000 1 <= A[i].size(), B[i].size() <= 1000 1 <= A[i][j], B[i][j] <= 1000

**Input Format**

The first argument is the 2D integer array A The second argument is the 2D integer array B

**Output Format**

You have to return a vector of vector of integers after doing required operations.

**Example Input**

Input 1:

A = [[1, 2, 3],

[4, 5, 6],

[7, 8, 9]]

B = [[9, 8, 7],

[6, 5, 4],

[3, 2, 1]]

Input 2:

A = [[1, 2, 3],

[4, 1, 2],

[7, 8, 9]]

B = [[9, 9, 7],

[1, 2, 4],

[4, 6, 3]]

**Example Output**

Output 1:

[[10, 10, 10],

[10, 10, 10],

[10, 10, 10]]

Output 2:

[[10, 11, 10],

[5, 3, 6],

[11, 14, 12]]

1. *Matrix Scalar Product*

**Problem Description**

You are given a **matrix** **A** and and an integer **B**, you have to perform scalar multiplication of matrix A with an integer B.  
  
**Problem Constraints**

1 <= A.size() <= 1000

1 <= A[i].size() <= 1000

1 <= A[i][j] <= 1000

1 <= B <= 1000  
  
**Input Format**

First argument is 2D array of integers A representing matrix.

Second argument is an integer B.  
  
**Output Format**

You have to return a 2D array of integers after doing required operations.  
  
**Example Input**

Input 1:

A = [[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]]

B = 2

Input 2:

A = [[1]]

B = 5

public int[][] solve(int[][] A, int B) {

int arr[][] = new int[A.length][A[0].length];

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

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

arr[i][j] = A[i][j] \* B;

}

}

return arr;

}

1. *Longest Common Prefix*

**Problem Description**

Given the array of strings **A**, you need to find the longest string **S,** which is the prefix of **ALL** the strings in the array.

The longest common prefix for a pair of strings **S1** and **S2** is the longest string **S** which is the prefix of both **S1** and **S2**.

**Example:** the longest common prefix of "abcdefgh" and "abcefgh" is "abc".

**Problem Constraints**

0 <= sum of length of all strings <= 1000000  
  
**Input Format**

The only argument given is an array of strings A.

**Output Format**

Return the longest common prefix of all strings in A.  
  
**Example Input**

Input 1:

A = ["abcdefgh", "aefghijk", "abcefgh"]

Input 2:

A = ["abab", "ab", "abcd"];  
  
**Example Output**

Output 1:

"a"

Output 2:

"ab"

public String longestCommonPrefix(String[] A) {

StringBuilder sb = new StringBuilder();

String common = A[0];

for (int i = 1; i < A.length; i++) {

String temp = A[i];

int j = 0;

if (common.isEmpty()) {

return common;

}

while (j < common.length() && j < temp.length()) {

if (temp.charAt(j) == common.charAt(j)) {

sb.append(temp.charAt(j));

j++;

} else {

break;

}

}

common = sb.toString();

sb = new StringBuilder();

}

return common;

}

1. *Continuous Sum Query*

**Problem Description**

There are **A** beggars sitting in a row outside a temple. Each beggar initially has an empty pot. When the devotees come to the temple, they donate some amount of coins to these beggars. Each devotee gives a fixed amount of coin(according to their faith and ability) to some **K** beggars sitting next to each other.  
Given the amount **P** donated by each devotee to the beggars ranging from **L** to **R** index, where **1** <= **L** <= **R** <= **A**, find out the final amount of money in each beggar's pot at the end of the day, provided they don't fill their pots by any other means.  
For ith devotee B[i][0] = L, B[i][1] = R, B[i][2] = P, Given by the 2D array **B**  
**Problem Constraints**

1 <= A <= 2 \* 105  
1 <= L <= R <= A  
1 <= P <= 103  
0 <= len(B) <= 105  
**Input Format**

The first argument is a single integer A.  
The second argument is a 2D integer array B.  
**Output Format**

Return an array(0 based indexing) that stores the total number of coins in each beggars pot.  
**Example Input**

Input 1:-

A = 5

B = [[1, 2, 10], [2, 3, 20], [2, 5, 25]]  
**Example Output**

Output 1:-

10 55 45 25 25

//TC:O(B+A) || SC:O(1)

public int[] solve(int A, int[][] B) {

int arr[] = new int[A];

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

int l = B[i][0] - 1;

int r = B[i][1] - 1;

int p = B[i][2];

arr[l] += p;

if (r + 1 < A) {

arr[r + 1] -= p;

}

}

for (int j = 1; j < arr.length; j++) {

arr[j] += arr[j - 1];

}

return arr;

}

1. *Max Sum Contiguous Subarray*

**Problem Description**

Find the maximum sum of **contiguous non-empty subarray** within an array **A** of length **N**.  
  
**Problem Constraints**

1 <= N <= 1e6  
-1000 <= A[i] <= 1000  
  
**Input Format**

The first and the only argument contains an integer array, A.  
  
**Output Format**

Return an integer representing the maximum possible sum of the contiguous subarray.  
  
**Example Input**

Input 1:

A = [1, 2, 3, 4, -10]

Input 2:

A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]   
  
**Example Output**

Output 1:

10

Output 2:

6

public int maxSubArray(final int[] A) { // TC:O(n) || SC:O(1)

int cSum = 0;

int mSum = Integer.MIN\_VALUE;

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

cSum += A[i];

mSum = Math.max(mSum, cSum);

if (cSum < 0) {

cSum = 0;

}

}

return mSum;

}

**Approach** : Keep two variables ‘curSum’ and ‘maxSum’ which denotes the current sum ending at the given position and the maximum sum of a subarray respectively.  
Iterate through the array , at every index we will add the current element to our curSum , after this we can update the maxSum as max(maxSum,curSum), After this we can just check if curSum is less than 0 , if it is then just replace curSum with 0.

1. *Add One To Number*

**Problem Description**

Given a **non-negative** number represented as an array of digits, add **1** to the number ( increment the number represented by the digits ).

The digits are stored such that the most significant digit is at the head of the list.

**NOTE:** Certain things are intentionally left unclear in this question which you should practice asking the interviewer. For example: for this problem, the following are some good questions to ask :

* **Q:** Can the input have **0's** before the most significant digit. Or, in other words, is **0 1 2 3** a valid input?
* **A:** For the purpose of this question, **YES**
* **Q:** Can the output have **0's** before the most significant digit? Or, in other words, is **0 1 2 4** a valid output?
* **A:** For the purpose of this question, **NO**. Even if the input has zeroes before the most significant digit.  
    
  **Problem Constraints**

1 <= size of the array <= 1000000

**Input Format**

First argument is an array of digits.  
  
**Output Format**

Return the array of digits after adding one.

**Example Input**

Input 1:

[1, 2, 3]  
  
**Example Output**

Output 1:

[1, 2, 4]

public int[] plusOne(int[] A) {

A = removeLeadingZeros(A);

int i = A.length - 1;

int[] B = new int[A.length];

B[A.length - 1] = 1;

int j = B.length - 1;

int carry = 0;

List<Integer> list = new ArrayList<>();

while (i >= 0 || j >= 0 || carry > 0) {

int sum = 0;

int digit;

if (i >= 0) {

sum += A[i];

i--;

}

if (j >= 0) {

sum += B[j];

j--;

}

sum += carry;

digit = sum % 10;

carry = sum / 10;

list.add(digit);

}

int arr[] = new int[list.size()];

int k = 0;

for (int ii = list.size() - 1; ii >= 0; ii--) {

arr[k] = list.get(ii);

k++;

}

return arr;

}

public int[] removeLeadingZeros(int[]A) {

int idx = -1;

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

if(A[i]!=0) {

idx = i;

break;

}

}

if(idx==-1) {

return new int[] {0};

}

int B[] = new int[A.length-(idx)];

int k=0;

for(int i=idx;i<A.length;i++) {

B[k] = A[i];

k++;

}

return B;

}

1. *Check palindrome using recursion*

**Problem Description**

Write a **recursive** function that checks whether string **A** is a palindrome or Not.  
Return 1 if the string A is a palindrome, else return 0.

**Note:** A palindrome is a string that's the same when read forward and backward.  
**Problem Constraints**

1 <= |A| <= 50000

String A consists only of lowercase letters.  
  
**Input Format** - The first and only argument is a string A.  
**Output Format -** Return 1 if the string A is a palindrome, else return 0.

**Example Input**

Input 1:

A = "naman"

Input 2:

A = "strings"  
  
**Example Output**

Output 1:1

Output 2:0

public int solve(String A) {

int n = A.length();

if (n == 0) {

return 1;

}

return isPalCheck(A, 0, n - 1);

}

public int isPalCheck(String A, int s, int e) {

if (s == e) {

return 1;

}

if (A.charAt(s) != A.charAt(e)) {

return 0;

}

if (s < e + 1) {

return isPalCheck(A, s + 1, e - 1);

}

return 1;

}

1. *Find Fibonacci Using Recursion*

**Problem Description**

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation:

**Fn = Fn-1 + Fn-2**

Given a number **A**, find and return the **Ath** Fibonacci Number using recursion.

Given that F0 = 0 and F1 = 1.  
  
**Problem Constraints**

0 <= A <= 20  
  
**Input Format**

First and only argument is an integer **A**.  
  
**Output Format**

Return an integer denoting the **Ath** term of the sequence.  
  
**Example Input**

Input 1:

A = 2

Input 2:

A = 9  
  
**Example Output**

Output 1:

1

Output 2:

34

public int findAthFibonacci(int A) {

if (A == 0) {

return 0;

} else if (A == 1) {

return 1;

}

return findAthFibonacci(A - 1) + findAthFibonacci(A - 2);

}

1. *Find Factorial! Using Recursion*

**Problem Description**

Write a program to find the factorial of the given number **A** using recursion.

Note: The factorial of a number **N** is defined as the **product** of the numbers from 1 to N.  
  
**Problem Constraints**

0 <= A <= 12  
  
**Input Format**

First and only argument is an integer **A**.  
  
**Output Format**

Return an integer denoting the factorial of the number **A**.  
  
**Example Input**

Input 1:

A = 4

Input 2:

A = 1  
  
**Example Output**

Output 1:

24

Output 2:

1

public int fact(int A) {

if(A==1) {

return 1;

}

else {

return A \* fact(A-1);

}

}

1. *Check Palindrome Using Recursion*

**Problem Description**

Write a **recursive** function that checks whether string **A** is a palindrome or Not.  
Return 1 if the string A is a palindrome, else return 0.

**Note:** A palindrome is a string that's the same when read forward and backward.  
  
**Problem Constraints**

1 <= |A| <= 50000

String A consists only of lowercase letters.  
  
**Input Format**

The first and only argument is a string A.  
  
**Output Format**

Return 1 if the string A is a palindrome, else return 0.  
  
**Example Input**

Input 1:

A = "naman"

Input 2:

A = "strings"

**Example Output**

Output 1:

1

Output 2:

0

public int solve(String A) {

int n = A.length();

if (n == 0) {

return 1;

}

return isPalCheck(A, 0, n - 1);

}

public int isPalCheck(String A, int s, int e) {

if (s == e) {

return 1;

}

if (A.charAt(s) != A.charAt(e)) {

return 0;

}

if (s < e + 1) {

return isPalCheck(A, s + 1, e - 1);

}

return 1;

}

1. *Sum of Digits!**Using Recursion*

**Problem Description**

Given a number **A**, we need to find the sum of its digits using recursion.  
  
**Problem Constraints**

1 <= A <= 109  
  
**Input Format**

The first and only argument is an integer **A**.  
**Output Format**

Return an integer denoting the sum of digits of the number **A**.  
**Example Input**

Input 1:

A = 46

Input 2:

A = 11   
  
**Example Output**

Output 1:

10

Output 2:

2

public int solve(int A) {

return digitSum(A,0);

}

public int digitSum(int A, int sum) {

if(A<1) {

return sum;

}

int s = sum;

int t = A%10;

s+=t;

A = A/10;

return digitSum(A,s);

}

1. *Count Occurrences*

**Problem Description**

Find the number of occurrences of **bob** in string **A** consisting of lowercase English alphabets.  
  
**Problem Constraints**

1 <= |A| <= 1000  
  
**Input Format**

The first and only argument contains the string A, consisting of lowercase English alphabets.  
**Output Format**

Return an integer containing the answer.  
  
**Example Input**

Input 1:

"abobc"

Input 2:

"bobob"

public int solve(String A) {

int count = 0;

int i = 0, j = 2;

while (j < A.length()) {

if (A.substring(i, j + 1).equals("bob")) {

count++;

}

i++;

j++;

}

return count;

}

1. *Amazing Subarrays*

You are given a string **S**, and you have to find all the **amazing substrings** of **S**.

An amazing Substring is one that starts with a **vowel** (a, e, i, o, u, A, E, I, O, U).

**Input**

Only argument given is string S.

**Output**

Return a single integer X mod 10003, here X is the number of Amazing Substrings in given the string.

**Constraints**

1 <= length(S) <= 1e6

S can have special characters

**Example**

Input

ABEC

Output

6

Explanation

Amazing substrings of given string are :

1. A

2. AB

3. ABE

4. ABEC

5. E

6. EC

here number of substrings are 6 and 6 % 10003 = 6.

public int solve(String A) {

// ABEC

int count = 0;

int len = A.length();

int mod = 10003;

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

if (A.charAt(i) == 'u' || A.charAt(i) == 'o' || A.charAt(i) == 'i' || A.charAt(i) == 'e'

|| A.charAt(i) == 'a' || A.charAt(i) == 'U' || A.charAt(i) == 'O' || A.charAt(i) == 'I'

|| A.charAt(i) == 'E' || A.charAt(i) == 'A') {

//(a+b)%m = (a%m + b%m)%m

int sub = (len - i)%mod;

count = (count + sub)%mod;

}

}

return count % 10003;

}

1. *Best Time to Buy and Sell Stocks I*

**Problem Description**

Say you have an array, **A**, for which the **i**th element is the price of a given stock on day **i**.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Return the **maximum** possible profit.  
  
**Problem Constraints**

0 <= **A.size()** <= 700000

1 <= **A[i]** <= 107  
  
**Input Format**

The first and the only argument is an array of integers, A.

**Output Format**

Return an integer, representing the maximum possible profit.  
  
**Example Input**

Input 1:

A = [1, 2]

Input 2:

A = [1, 4, 5, 2, 4]  
  
**Example Output**

Output 1:

1

Output 2:

4

public int maxProfit(final int[] A) {

if (A.length < 1) {

return 0;

}

int maxProfit = 0;

int max = A[A.length - 1];

for (int i = A.length - 2; i >= 0; i--) {

max = Math.max(max, A[i]);

maxProfit = Math.max(maxProfit, max - A[i]);

}

return maxProfit;

}

1. *Pick from both sides*

**Problem Description**

You are given an integer array **A** of size **N**.

You have to perform **B** operations. In one operation, you can remove either the leftmost or the rightmost element of the array **A**.

Find and return the **maximum possible sum** of the **B elements** that were removed after the **B** operations.

**NOTE:** Suppose **B = 3**, and array A contains 10 elements, then you can:

* Remove 3 elements from front and 0 elements from the back, OR
* Remove 2 elements from front and 1 element from the back, OR
* Remove 1 element from front and 2 elements from the back, OR
* Remove 0 elements from front and 3 elements from the back.

**Problem Constraints**

1 <= N <= 105

1 <= B <= N

-103 <= A[i] <= 103  
  
**Input Format**

First argument is an integer array **A**.

Second argument is an integer **B**.  
  
**Output Format**

Return an integer denoting the maximum possible sum of elements you removed.  
  
**Example Input**

Input 1:

A = [5, -2, 3 , 1, 2]

B = 3

Input 2:

A = [ 2, 3, -1, 4, 2, 1 ]

B = 4  
  
**Example Output**

Output 1:

8

Output 2:

9

public int solve(int[] A, int B) {

int ans = 0;

int ps[] = new int[A.length];

int ss[] = new int[A.length];

ps[0] = A[0];

for (int i = 1; i < A.length; i++) {

ps[i] = ps[i - 1] + A[i];

}

ss[A.length - 1] = A[A.length - 1];

for (int j = A.length - 2; j >= 0; j--) {

ss[j] = ss[j + 1] + A[j];

}

ans = Math.max(ps[B - 1], ss[A.length - B]);

int sum = 0;

for (int i = 1; i < B; i++) {

sum = ps[i - 1] + ss[A.length - B + i];

ans = Math.max(ans, sum);

}

return ans;

}

1. *Sub-array with 0 sum*

**Problem Description**

Given an array of integers **A**, find and return whether the given array contains a **non-empty** subarray with a sum equal to 0.

If the given array contains a sub-array with sum zero return 1, else return 0.  
  
**Problem Constraints**

1 <= |A| <= 100000

-10^9 <= A[i] <= 10^9  
  
**Input Format**

The only argument given is the integer array A.  
  
**Output Format**

Return whether the given array contains a subarray with a sum equal to 0.  
  
**Example Input**

Input 1:

A = [1, 2, 3, 4, 5]

Input 2:

A = [4, -1, 1]  
  
**Example Output**

Output 1:

0

Output 2:

1

public int solve(ArrayList<Integer> A) {

long sum = 0;

Set<Long> set = new HashSet<>();

for (Integer a : A) {

sum += a;

if(sum == 0) {

return 1;

}

if (set.contains(sum)) {

return 1;

} else {

set.add(sum);

}

}

return 0;

}

1. *Count Subarray Zero Sum*

**Problem Description**

Given an array **A**of **N** integers.  
  
Find the count of the subarrays in the array which sums to zero. Since the answer can be very large, return the remainder on dividing the result with 109+7  
  
**Problem Constraints**

1 <= N <= 105  
-109 <= A[i] <= 109  
  
**Input Format**

Single argument which is an integer array A.  
  
**Output Format**

Return an integer.

**Example Input**

Input 1:

A = [1, -1, -2, 2]

Input 2:

A = [-1, 2, -1]  
  
**Example Output**

Output 1:

3

Output 2:

1

public int solve(ArrayList<Integer> A) { //TC:O(N) || SC:O(N)

int count=0;

long[] ps = new long[A.size()];

ps[0] = A.get(0);

for(int i=1;i<A.size();i++){

ps[i] = ps[i-1]+A.get(i);

}

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

map.put(Long.valueOf(0),1);

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

if(map.containsKey(ps[i])){

int val = map.get(ps[i]);

count+= val;

map.put(ps[i],val+1);

}else{

map.put(ps[i],1);

}

}

return count%1000000007;

}

1. *Count Pair Difference*

**Problem Description**

You are given an array **A** of **N** integers and an integer **B**.  
Count the number of pairs (i,j) such that A[i] - A[j] = B and i ≠ j.  
  
Since the answer can be very large, return the remainder after dividing the count with 109+7.  
  
**Problem Constraints**

1 <= N <= 105  
1 <= A[i] <= 109  
1 <= B <= 109  
  
**Input Format**

First argument A is an array of integers and second argument B is an integer.  
  
**Output Format**

Return an integer.

**Example Input**

Input 1:

A = [3, 5, 1, 2]

B = 4

Input 2:

A = [1, 2, 1, 2]

B = 1

**Example Output**

Output 1:

1

Output 2:

4

public int solve(ArrayList<Integer> A, int B) { //TC:O(N) || SC:O(N)

int ans = 0;

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

for (Integer num : A) {

ans += map.getOrDefault(num - B, 0);

ans += map.getOrDefault(num + B, 0);

ans %= 1000000007;

map.put(num, map.getOrDefault(num, 0) + 1);

}

return ans;

}

1. *Single Element in Sorted Array*

**Problem Description**

Given a sorted array of integers **A** where every element appears twice except for one element which appears once, find and return this single element that appears only once.

Elements which are appearing twice are adjacent to each other.

**NOTE:**Users are expected to solve this in O(log(N)) time.  
  
**Problem Constraints**

1 <= **|A|** <= 100000

1 <= A[i] <= 10^9

**Input Format**

The only argument given is the integer array A.

**Output Format**

Return the single element that appears only once.

**Example Input**

Input 1:

A = [1, 1, 7]

Input 2:

A = [2, 3, 3]

**Example Output**

Output 1:

7

Output 2:

2

s

public int solve(ArrayList<Integer> A) {

int len = A.size();

if (len == 1) {

return A.get(0);

}

if (!A.get(0).equals(A.get(1))) {

return A.get(0);

}

if (!A.get(len - 1).equals(A.get(len - 2))) {

return A.get(len - 1);

}

int low = 0, high = A.size() - 1;

while (low < high) {

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

if (mid % 2 == 0) {

if (A.get(mid).equals(A.get(mid + 1))) {

low = mid + 2;

} else {

high = mid;

}

} else {

if (A.get(mid).equals(A.get(mid - 1))) {

low = mid + 1;

} else {

high = mid;

}

}

}

return A.get(low);

}

1. *Sorted Insert Position*

**Problem Description**

You are given a sorted array A of size N and a target value B.  
Your task is to find the index (0-based indexing) of the target value in the array.

* If the target value is **present**, return its index.
* If the target value is **not found**, return the index of least element greater than equal to B.

Your solution should have a time complexity of O(log(N)).  
  
**Problem Constraints**

1 <= N <= 106  
1 <= A[i] <= 105  
1 <= B <= 105  
  
**Input Format**

The first argument is an integer array A of size N.  
The second argument is an integer B.

**Output Format**

Return an integer denoting the index of target value.  
  
**Example Input**

Input 1:

A = [1, 3, 5, 6]

B = 5

Input 2:

A = [1, 4, 9]

B = 3  
  
**Example Output**

Output 1:

2

Output 2:

1

public int searchInsert(ArrayList<Integer> A, int B) {

int l = 0, r = A.size() - 1;

int idx = -1;

int max = Integer.MIN\_VALUE;

while (l <= r) {

int mid = l + (r - l) / 2;

if (A.get(mid) == B) {

return mid;

} else if (A.get(mid) < B) {

int num = A.get(mid);

if (num > max) {

max = num;

idx = mid;

}

l = mid + 1;

} else {

r = mid - 1;

}

}

return idx + 1;

}

1. *Maximum height of staircase*

**Problem Description**

Given an integer **A** representing the number of square blocks. **The height of each square block is 1**. The task is to create a staircase of max-height using these blocks.

The first stair would require only one block, and the second stair would require two blocks, and so on.

Find and return the maximum height of the staircase.  
  
**Problem Constraints**

0 <= A <= 109  
  
**Input Format**

The only argument given is integer A.  
  
**Output Format**

Return the maximum height of the staircase using these blocks.  
  
**Example Input**

Input 1:

A = 10

Input 2:

A = 20  
  
**Example Output**

Output 1:

4

Output 2:

5

public int solve1(int A) {

int count = 1;

int blocksUsed = 0;

int maxHeight = Integer.MIN\_VALUE;

if (A <= 0) {

return 0;

} else {

while (blocksUsed <= A) {

blocksUsed += count;

if (blocksUsed > A) {

return maxHeight;

}

maxHeight = Math.max(count, maxHeight);

count++;

}

}

return maxHeight;

}

public int solve(int A) {

int height = A>0?1:0;

int i=1;

while(i\*(i+1) <= 2\*A){

height=i;

i++;

}

return height;

}

1. *Matrix Search*

**Problem Description**

Given a matrix of integers **A** of size **N x M** and an integer **B**. Write an efficient algorithm that searches for integer **B** in matrix **A**.

This matrix A has the following properties:

1. Integers in each row are **sorted** from left to right.
2. The first integer of each row is greater than or equal to the last integer of the previous row.

Return **1** if **B** is present in **A**, else return **0**.

**NOTE:** Rows are numbered from top to bottom, and columns are from left to right.  
  
**Problem Constraints**

1 <= N, M <= 1000  
1 <= A[i][j], B <= 106  
  
**Input Format**

The first argument given is the integer matrix A.  
The second argument given is the integer B.  
  
**Output Format**

Return 1 if B is present in A else, return 0.  
  
**Example Input**

Input 1:

A = [

[1, 3, 5, 7]

[10, 11, 16, 20]

[23, 30, 34, 50]

]

B = 3

Input 2:

A = [

[5, 17, 100, 111]

[119, 120, 127, 131]

]

B = 3  
  
**Example Output**

Output 1:

1

Output 2:

0

public int searchMatrix(int[][] A, int B) {

int r = 0, c = A[0].length - 1;

while (r < A.length && c >= 0) {

if (A[r][c] == B) {

return 1;

} else if (A[r][c] > B) {

c--;

} else {

r++;

}

}

return 0;

}

1. *Subarray with given sum*

**Problem Description**

Given an array of positive integers **A** and an integer **B**, find and return first continuous subarray which adds to **B**.

If the answer does not exist **return an array** with a single integer "**-1**".

First sub-array means the sub-array for which starting index in minimum.  
  
**Problem Constraints**

1 <= length of the array <= 100000  
1 <= A[i] <= 109  
1 <= B <= 109  
  
**Input Format**

The first argument given is the integer array A.

The second argument given is integer B.

**Output Format**

Return the **first continuous sub-array** which adds to B and if the answer does not exist **return an array** with a single integer "**-1**".  
  
**Example Input**

Input 1:

A = [1, 2, 3, 4, 5]

B = 5

Input 2:

A = [5, 10, 20, 100, 105]

B = 110  
  
**Example Output**

Output 1:

[2, 3]

Output 2:

[-1],

public ArrayList<Integer> solve(ArrayList<Integer> A, int B) {

ArrayList<Integer> arr = new ArrayList<>();

int l = 0, r = 0;

int sum = A.get(l);

while (r < A.size()) {

if (sum < B) {

r++;

if (r < A.size()) {

sum += A.get(r);

}

} else if (sum > B) {

sum = sum - A.get(l);

l++;

} else if (sum == B) {

for (int i = l; i <= r; i++) {

arr.add(A.get(i));

}

break;

}

}

if (arr.size() == 0) {

arr.add(-1);

}

return arr;

}

1. *Pairs with Given Difference*

**Problem Description**

Given an one-dimensional integer array **A** of size **N** and an integer **B**.

**Count all distinct pairs with difference equal to B**.

Here a pair is defined as an integer pair (x, y), where x and y are both numbers in the array and their absolute difference is **B**.  
  
**Problem Constraints**

1 <= N <= 104

0 <= A[i], B <= 105  
  
**Input Format**

First argument is an one-dimensional integer array **A** of size **N**.

Second argument is an integer **B**.  
  
**Output Format**

Return an integer denoting the count of all distinct pairs with difference equal to **B**.  
  
**Example Input**

Input 1:

A = [1, 5, 3, 4, 2]

B = 3

Input 2:

A = [8, 12, 16, 4, 0, 20]

B = 4

Input 3:

A = [1, 1, 1, 2, 2]

B = 0  
  
**Example Output**

Output 1:

2

Output 2:

5

Output 3:

2

public int solve(ArrayList<Integer> A, int B) {

Collections.sort(A);

int count = 0;

int l = 0, r = 1;

while(r<A.size()) {

int diff = A.get(r) - A.get(l);

if(l==r) {

r++;

}else if(diff == B) {

count++;

r++;

while(r<A.size() && A.get(r).equals(A.get(r-1))) {

r++;

}

}

else if(diff>B) {

l++;

}else {

r++;

}

}

return count;

}

1. *Another Coin Problem*

**Problem Description**

The monetary system in DarkLand is really simple and systematic. The locals-only use coins. The coins come in different values. The values used are:

1, 5, 25, 125, 625, 3125, 15625, ...

Formally, for each **K >= 0** there are coins worth **5K**.

Given an integer **A** denoting the cost of an item, find and return the **smallest** number of coins necessary to pay exactly the cost of the item (assuming you have a sufficient supply of coins of each of the types you will need).

**Problem Constraints**

1 <= A <= 2×109  
  
**Input Format**

The only argument given is integer A.

**Output Format**

Return the smallest number of coins necessary to pay exactly the cost of the item.

**Example Input**

Input 1:

A = 47

Input 2:

A = 9  
  
**Example Output**

Output 1:

7

Output 2:

5

public int solve(int A) {

int total = 0;

int num = A;

while (num > 0) {

int n = num % 5;

total += n;

num = num / 5;

}

return total;

}

1. *Assign Mice to Holes*

**Problem Description**

N Mice and **N** holes are placed in a straight line. Each hole can accommodate only one mouse.

The positions of Mice are denoted by array **A,** and the position of holes is denoted by array **B**.

A mouse can stay at his position, move one step right from x to x + 1, or move one step left from x to x − 1. Any of these moves consume 1 minute.

Assign mice to holes so that the time when the last mouse gets inside a hole is **minimized**.

**Problem Constraints**

1 <= N <= 105

-109 <= A[i], B[i] <= 109

**Input Format**

The first argument is an integer array A.

The second argument is an integer array B.

**Output Format**

Return an integer denoting the minimum time when the last nouse gets inside the holes.

**Example Input**

Input 1:

A = [-4, 2, 3]

B = [0, -2, 4]

Input 2:

A = [-2]

B = [-6]

**Example Output**

Output 1:

2

Output 2:

4

public int mice(int[] A, int[] B) {

Arrays.sort(A);

Arrays.sort(B);

int total = 0;

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

int x = Math.abs(A[i] - B[i]);

total = Math.max(total, x);

}

return total;

}

1. *Balanced Paranthesis*

**Problem Description**

Given an expression string **A**, examine whether the pairs and the orders of “{“,”}”, ”(“,”)”, ”[“,”]” are correct in **A**.

Refer to the examples for more clarity.

**Problem Constraints**

1 <= |A| <= 100

**Input Format**

The first and the only argument of input contains the string A having the parenthesis sequence.

**Output Format**

Return 0 if the parenthesis sequence is not balanced.

Return 1 if the parenthesis sequence is balanced.

**Example Input**

Input 1:

A = {([])}

Input 2:

A = (){

Input 3:

A = ()[]

**Example Output**

Output 1:

1

Output 2:

0

Output 3:

1

public int solve(String A) { //TC:O(N) || SC:O(N)

HashMap < Character, Character > mp = new HashMap < Character, Character > ();

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

mp.put(')', '(');

mp.put('}', '{');

mp.put(']', '[');

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

char c = A.charAt(i);

if (c == '(' || c == '[' || c == '{') {

// push any opening bracket into the stack

st.push(c);

} else if (st.empty() || st.peek() != mp.get(c)) {

// check if the last unpaired opening bracket is of the same type

// as the current closing bracket

return 0;

} else {

st.pop();

}

}

// checks if all the opening brackets are paired

if (st.empty())

return 1;

return 0;

}

1. *Passing game*

**Problem Description**

There is a football event going on in your city. In this event, you are given **A** passes and players having ids between **1** and **106**.

Initially, some player with a given id had the ball in his possession. You have to make a program to display the id of the player who possessed the ball after exactly A passes.

There are two kinds of passes:

1) ID

2) 0

For the first kind of pass, the player in possession of the ball passes the ball "forward" to the player with id = ID.

For the second kind of pass, the player in possession of the ball passes the ball back to the player who had forwarded the ball to him.

In the second kind of pass "0" just means Back Pass.

Return the ID of the player who currently possesses the ball.

**Problem Constraints**

1 <= A <= 100000

1 <= B <= 100000

|C| = A

**Input Format**

The first argument of the input contains the number A.

The second argument of the input contains the number B ( id of the player possessing the ball in the very beginning).

The third argument is an array C of size A having (ID/0).

**Output Format**

Return the "ID" of the player who possesses the ball after A passes.

**Example Input**

Input 1:

A = 10

B = 23

C = [86, 63, 60, 0, 47, 0, 99, 9, 0, 0]

Input 2:

A = 1

B = 1

C = [2]

**Example Output**

Output 1:

63

Output 2:

2

public int solve(int A, int B, int[] C) {

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

st.push(B);

for (int x: C) {

// pop from stack

if (x == 0) st.pop();

// push the given ID to stack

else st.push(x);

}

return st.peek();

}

**MEDIUM**

1. *Prime Number*

**Problem Description**

Given a number **A**. Return 1 if **A** is prime and return 0 if not.

**Note :**   
The value of **A** can cross the range of Integer.

**Problem Constraints**

1 <= **A** <= 109  
**Input Format**

The first argument is a single integer A.  
**Output Format**

Return 1 if A is prime else return 0.

public int solve(int A) {

int count = 0;

int num = (int) Math.sqrt(A);

for (int i = 1; i <= num; i++) {

if (A % i == 0) {

if (i != A / i) {

count += 2;

} else {

count += 1;

}

}

}

return count==2?1:0;

}

1. *Count of Primes*

**Problem Description**

You will be given an integer n. You need to return the count of prime numbers less than or equal to n.  
**Problem Constraints**

0 <= n <= 10^3  
**Input Format**

Single input parameter n in function.  
**Output Format**

Return the count of prime numbers less than or equal to n.  
**Example Input**

Input 1:

19

Input 2:

1  
**Example Output**

Output 1:

8

Output 2:

0

public class Solution {

public int solve(int A) {

int primeCount = 0;

for (int num = 1; num <= A; num++) {

int rootOfNum = (int) Math.sqrt(num);

int count = 0;

for (int i = 1; i <= rootOfNum; i++) {

if (num % i == 0) {

if (i != num / i) {

count += 2;

} else {

count += 1;

}

}

}

if (count == 2) {

primeCount += 1;

}

}

return primeCount;

}

}

1. *Special Index*

**Problem Description**

Given an array, **arr[]** of size **N**, the task is to find the count of array indices such that removing an element from these indices makes the sum of even-indexed and odd-indexed array elements equal.

**Problem Constraints**

1 <= n <= 105

-105 <= A[i] <= 105

**Input Format**

First argument contains an array A of integers of size N

**Output Format**

Return the count of array indices such that removing an element from these indices makes the sum of even-indexed and odd-indexed array elements equal.

**Example Input**

Input 1:

A=[2, 1, 6, 4]

Input 2:

A=[1, 1, 1]

**Example Output**

Output 1:

1

Output 2:

3

public class Solution {

// **TC:O(N) || SC : O(N)**

public int solve(int[] A) {

int[] pfe = new int[A.length];

int[] pfo = new int[A.length];

pfe[0] = A[0];

for (int i = 1; i < A.length; i++) {

if (i % 2 == 0) {

pfe[i] = pfe[i - 1] + A[i];

} else {

pfe[i] = pfe[i - 1];

}

}

pfo[0] = 0;

for (int i = 1; i < A.length; i++) {

if (i % 2 != 0) {

pfo[i] = pfo[i - 1] + A[i];

} else {

pfo[i] = pfo[i - 1];

}

}

int se, so;

int count = 0;

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

if (i == 0) {

se = pfe[A.length - 1] - pfe[i];

so = pfo[A.length - 1] - pfo[i];

} else {

se = pfe[i - 1] + pfo[A.length - 1] - pfo[i];

so = pfo[i - 1] + pfe[A.length - 1] - pfe[i];

}

if (se == so) {

count += 1;

}

}

return count;

}

}

1. *Good Subarrays*

**Problem Description**

Given an array of integers **A**, a subarray of an array is said to be good if it fulfills any one of the criteria:  
1. Length of the subarray is be even, and the sum of all the elements of the subarray must be less than **B**.  
2. Length of the subarray is be odd, and the sum of all the elements of the subarray must be greater than **B**.  
Your task is to find the count of good subarrays in A.  
  
**Problem Constraints**

1 <= **len(A)** <= 103  
1 <= **A[i]** <= 103  
1 <= **B** <= 107  
**Input Format**

The first argument given is the integer array A.  
The second argument given is an integer B.  
  
**Output Format**

Return the count of good subarrays in A.

**Example Input**

Input 1:

A = [1, 2, 3, 4, 5]

B = 4

Input 2:

A = [13, 16, 16, 15, 9, 16, 2, 7, 6, 17, 3, 9]

B = 65  
  
**Example Output**

Output 1:

6

Output 2:

36

public int solve(int[] A, int B) {

int count = 0;

int ps[] = new int[A.length];

ps[0] = A[0];

for (int i = 1; i < A.length; i++) {

ps[i] = ps[i - 1] + A[i];

}

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

for (int j = i; j < A.length; j++) {

int size = j - i + 1;

int sum = i == 0 ? ps[j] : ps[j] - ps[i - 1];

if (size % 2 == 0 && sum < B) {

count++;

} else if (size % 2 != 0 && sum > B) {

count++;

}

}

}

return count;

}

1. *Counting Subarrays*

**Problem Description**

Given an array **A**of **N** non-negative numbers and a non-negative number**B,**  
you need to find the **number of subarrays in A with a sum less than B.**  
We may assume that there is no overflow.  
  
**Problem Constraints**

1 <= N <= 103

1 <= A[i] <= 1000

1. <= B <= 107  
   **Input Format**

First argument is an integer array **A**.

Second argument is an integer **B**.

**Output Format**

Return an integer denoting the **number of subarrays in A having sum less than B**.

**Example Input**

Input 1:

A = [2, 5, 6]

B = 10

Input 2:

A = [1, 11, 2, 3, 15]

B = 10

**Example Output**

Output 1:

4

Output 2:

4

public int solve(int[] A, int B) {

int count = 0;

int[] ps = new int[A.length];

ps[0] = A[0];

for (int i = 1; i < A.length; i++) {

ps[i] = ps[i - 1] + A[i];

}

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

for (int j = i; j < A.length; j++) {

int sum = i==0?ps[j]:ps[j] - ps[i - 1];

if (sum < B) {

count++;

}

}

}

return count;

}

1. *Subarray with least average*

**Problem Description**

Given an array of size **N**, find the subarray of size **K** with the least average.  
  
**Problem Constraints**

1<=k<=N<=1e5

1e5<=A[i]<=1e5  
  
**Input Format**

First argument contains an array A of integers of size N.

Second argument contains integer k  
  
**Output Format**

Return the index of the first element of the subarray of size k that has least average.  
Array indexing starts from 0.  
  
**Example Input**

Input 1:

A=[3, 7, 90, 20, 10, 50, 40]

B=3

Input 2:

A=[3, 7, 5, 20, -10, 0, 12]

B=2  
  
**Example Output**

Output 1:

3

Output 2:

4

public int solve(int[] A, int B) {

int left = 0, right = B - 1;

float min = Integer.MAX\_VALUE;

float sum = 0;

int idx = -1;

for (int i = left; i <= right; i++) {

sum += A[i];

}

float ave = sum / B;

if (ave < min) {

min = ave;

idx = left;

}

left++;

right++;

while (right < A.length) {

sum = sum + A[right] - A[left - 1];

ave = sum / B;

if (ave < min) {

min = ave;

idx = left;

}

left++;

right++;

}

return idx;

}

1. *Row to Column Zero*

**Problem Description**

You are given a 2D integer matrix A, make all the elements in a row or column zero if the A[i][j] = 0. Specifically, make entire ith row and jth column zero.

**Problem Constraints**

1 <= A.size() <= 103

1 <= A[i].size() <= 103

0 <= A[i][j] <= 103

**Input Format**

First argument is a 2D integer matrix A.

**Output Format**

Return a 2D matrix after doing required operations.  
  
**Example Input**

Input 1:

[1,2,3,4]

[5,6,7,0]

[9,2,0,4]

**Example Output**

Output 1:

[1,2,0,0]

[0,0,0,0]

[0,0,0,0]

public int[][] solve(int[][] A) { // TC:O(N^2) || SC:O(1)

// go through the rows

for (int row = 0; row < A.length; row++) {

int flag = 0;

for (int col = 0; col < A[row].length; col++) {

if (A[row][col] == 0) {

flag = 1;

}

}

if (flag == 1) {

for (int col = 0; col < A[row].length; col++) {

if (A[row][col] != 0) {

A[row][col] = -1;

}

}

}

}

// go through the columns

for (int col = 0; col < A[0].length; col++) {

int flag = 0;

for (int row = 0; row < A.length; row++) {

if (A[row][col] == 0) {

flag = 1;

}

}

if (flag == 1) {

for (int row = 0; row < A.length; row++) {

if (A[row][col] != 0) {

A[row][col] = -1;

}

}

}

}

// go through the entire matrix and replace -1 with 0

for (int row = 0; row < A.length; row++) {

for (int col = 0; col < A[row].length; col++) {

if (A[row][col] == -1) {

A[row][col] = 0;

}

}

}

return A;

}

1. *Longest Palindromic Substring*

**Problem Description**

Given a string A of size N, find and return the **longest palindromic substring** in A.

Substring of string A is A[i...j] where 0 <= i <= j < len(A)

**Palindrome string:**  
A string which reads the same backwards. More formally, A is palindrome if reverse(A) = A.

**Incase of conflict**, return the substring which occurs first ( with the least starting index).

**Problem Constraints**

1 <= N <= 6000  
  
**Input Format**

First and only argument is a string A.  
  
**Output Format**

Return a string denoting the longest palindromic substring of string A.  
  
**Example Input**

Input 1:

A = "aaaabaaa"

Input 2:

A = "abba  
  
**Example Output**

Output 1:

"aaabaaa"

Output 2:

"abba"

public String longestPalindromicSubstring(String A) // { TC:O(N^2) || SC:O(1)

int[] currentLongest = { 0, 1 };

for (int i = 1; i < A.length(); i++) {

int[] odd = getLongestPalindrome(A, i - 1, i + 1);

int[] even = getLongestPalindrome(A, i - 1, i);

int[] longest = odd[1] - odd[0] > even[1] - even[0] ? odd : even;

if(currentLongest[1] - currentLongest[0] < longest[1] - longest[0]) {

currentLongest = longest;

}else if(currentLongest[1] - currentLongest[0] == longest[1] - longest[0] && currentLongest[0]>longest[0]) {

currentLongest = longest;

}

}

return A.substring(currentLongest[0], currentLongest[1]);

}

public int[] getLongestPalindrome(String str, int lIdx, int rIdx) {

while (lIdx >= 0 && rIdx < str.length()) {

if (str.charAt(lIdx) != str.charAt(rIdx)) {

break;

}

lIdx--;

rIdx++;

}

return new int[] { lIdx + 1, rIdx };

}

1. *Rain Water Trapped*

**Problem Description**

Given a vector **A** of non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.  
  
**Problem Constraints**

1 <= **|A|** <= 100000  
  
**Input Format**

First and only argument is the vector **A**  
  
**Output Format**

Return one integer, the answer to the question  
  
**Example Input**

Input 1:

A = [0, 1, 0, 2]

Input 2:

A = [1, 2]  
  
**Example Output**

Output 1:

1

Output 2:

0

public int trapBrute(final int[] A) { // TC:O(N^2) || SC:O(1)

int ans = 0;

for (int i = 1; i < A.length - 1; i++) {

int maxL = findMax(A, 0, i - 1);

int maxR = findMax(A, i + 1, A.length - 1);

int water = Math.min(maxL, maxR) - A[i];

if (water > 0) {

ans += water;

}

}

return ans;

}

public int findMax(int[] A, int s, int e) {

int max = Integer.MIN\_VALUE;

for (int i = s; i <= e; i++) {

if (A[i] > max) {

max = A[i];

}

}

return max;

}

public int trapTimeOptimised(final int[] A) { // TC:O(N) || SC:O(N)

int ans = 0;

int lMax[] = new int[A.length];

lMax[0] = 0;

int rMax[] = new int[A.length];

rMax[0] = 0;

for (int i = 1; i < A.length - 1; i++) {

lMax[i] = Math.max(lMax[i - 1], A[i - 1]);

}

for (int i = A.length - 2; i >= 0; i--) {

rMax[i] = Math.max(rMax[i + 1], A[i + 1]);

}

for (int i = 1; i < A.length - 1; i++) {

int water = Math.min(lMax[i], rMax[i]) - A[i];

if (water > 0) {

ans += water;

}

}

return ans;

}

public int trapSpaceOptimised(final int[] A) { // TC:O(N) || SC:O(1)

int l = 0, r = A.length - 1;

int lMax = A[0];

int rMax = A[A.length - 1];

int ans = 0;

while (l < r) {

int water = 0;

if (lMax < rMax) {

l++;

water = lMax - A[l];

lMax = Math.max(lMax, A[l]);

} else {

r--;

water = rMax - A[r];

rMax = Math.max(rMax, A[r]);

}

if (water > 0) {

ans += water;

}

}

return ans;

}

1. *Equilibrium index of an array*

**Problem Description**

You are given an array **A**of integers of size **N**.

Your task is to find the equilibrium index of the given array

The equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes.

If there are no elements that are at lower indexes or at higher indexes, then the corresponding sum of elements is considered as 0.

**Note:**

* Array indexing starts from 0.
* If there is no equilibrium index then return -1.
* If there are more than one equilibrium indexes then return the minimum index.

**Problem Constraints**

1 <= N <= 105

-105 <= A[i] <= 105  
**Input Format**

First arugment is an array A .  
**Output Format**

Return the equilibrium index of the given array. If no such index is found then return -1.  
**Example Input**

Input 1:

A = [-7, 1, 5, 2, -4, 3, 0]

Input 2:

A = [1, 2, 3]  
**Example Output**

Output 1:

3

Output 2:

-1

Approach:

1) Initialize leftsum as 0

2) Get the total sum of the array as sum

3) Iterate through the array and for each index i, do following.

a) Update sum to get the right sum.

sum = sum - arr[i]

// sum is now right sum

b) If leftsum is equal to sum, then return current index.

// update leftsum for next iteration.

c) leftsum = leftsum + arr[i]

4) return -1

// If we come out of loop without returning then

// there is no equilibrium index

public int solve(int[] A) {

int sum = 0, rightSum = 0;

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

rightSum += A[i];

}

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

rightSum -= A[i];

if (sum == rightSum) {

return i;

}

sum += A[i];

}

return -1;

}

1. *N/3 Repeat Number*

**Problem Description**

You're given a read-only array of **N** integers. Find out if any integer occurs more than N/3 times in the array in linear time and constant additional space.

If so, return the integer. If not, return **-1**.

If there are multiple solutions, return any one.

Note: Read-only array means that the input array should not be modified in the process of solving the problem  
  
**Problem Constraints**

1 <= N <= 7\*105  
1 <= A[i] <= 109  
  
**Input Format**

The only argument is an integer array A.

**Output Format**

Return an integer.  
  
**Example Input**

Input 1:

[1 2 3 1 1]

Input 2:

[1 2 3]  
  
**Example Output**

Output 1:

1

Output 2:

-1

//TC:O(N) || SC:O(1)

public int N3majorityElement(final List<Integer> a) {

if (a.size() == 0) {

return -1;

} else if (a.size() == 1) {

return a.get(0);

} else if (a.size() == 2) {

return a.get(0);

} else {

int first = a.get(0), second = a.get(1);

int count1 = 1, count2 = 1;

for (int i = 2; i < a.size(); i++) {

if (a.get(i) == first) {

count1++;

} else if (a.get(i) == second) {

count2++;

} else if (count1 == 0) {

first = a.get(i);

} else if (count2 == 0) {

second = a.get(i);

} else {

count1--;

count2--;

}

}

count1 = 0;

count2 = 0;

for (int i = 0; i < a.size(); i++) {

if (first == a.get(i)) {

count1++;

} else if (second == a.get(i)) {

count2++;

}

}

if (count1 > a.size() / 3) {

return first;

} else if (count2 > a.size() / 3) {

return second;

} else {

return -1;

}

}

}

1. *Length of longest consecutive ones*

Given a binary string **A**. It is allowed to do at most one swap between any 0 and 1. Find and return the length of the longest consecutive 1’s that can be achieved.

**Input Format**

The only argument given is string A.

**Output Format**

Return the length of the longest consecutive 1’s that can be achieved.

**Constraints**

1 <= length of string <= 1000000

A contains only characters 0 and 1.

**For Example**

Input 1:

A = "111000"

Output 1:

3

Input 2:

A = "111011101"

Output 2:

7

public int solve(String A) { // TC:O(N) || SC:O(N)

int len = A.length();

// count total number of 1

int count\_1 = 0;

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

if (A.charAt(i) == '1') {

count\_1++;

}

}

// have 2 arrays, left and right and get the 1s to the left and right of idx

int[] left = new int[len];

int[] right = new int[len];

if (A.charAt(0) == '1') {

left[0] = 1;

}

if (A.charAt(len - 1) == '1') {

right[len - 1] = 1;

}

for (int i = 1; i < len; i++) {

if (A.charAt(i) == '1') {

left[i] = left[i - 1] + 1;

}

}

for (int i = len - 2; i >= 0; i--) {

if (A.charAt(i) == '1') {

right[i] = right[i + 1] + 1;

}

}

// find max 1 to the left and right of each idx

int maxCount = 0;

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

maxCount = Math.max(maxCount, Math.max(left[i], right[i]));

}

// get the max count

for (int i = 1; i < len - 1; i++) {

if (A.charAt(i) == '0') {

int count = left[i - 1] + right[i + 1];

if (count < count\_1) {

count++;

}

maxCount = Math.max(maxCount, count);

}

}

return maxCount;

}

1. **Next Permutation**

**Problem Description**

Implement the **next permutation**, which rearranges numbers into the numerically next greater permutation of numbers for a given array **A** of size **N**.

If such arrangement is not possible, it must be rearranged as the lowest possible order, i.e., sorted in ascending order.

**NOTE:**

* The replacement must be in-place, do **not** allocate extra memory.
* DO NOT USE LIBRARY FUNCTION FOR NEXT PERMUTATION. Use of Library functions will disqualify your submission retroactively and will give you penalty points.

**Problem Constraints**

1 <= N <= 5 \* 105

1 <= A[i] <= 109  
  
**Input Format**

The first and the only argument of input has an array of integers, A.  
  
**Output Format**

Return an array of integers, representing the next permutation of the given array.

**Example Input**

Input 1:

A = [1, 2, 3]

Input 2:

A = [3, 2, 1]  
  
**Example Output**

Output 1:

[1, 3, 2]

Output 2:

[1, 2, 3]

public int[] nextPermutation(int[] A) {

// find x such that A[i] < A[i+1]

int nums[] = A;

int ind1 = -1;

int ind2 = -1;

// step 1 find breaking point

for (int i = A.length - 2; i >= 0; i--) {

if (nums[i] < nums[i + 1]) {

ind1 = i;

break;

}

}

// if there is no breaking point

if (ind1 == -1) {

return reverse(nums, 0);

}

else {

// step 2 find next greater element and swap with ind2

for (int i = nums.length - 1; i >= 0; i--) {

if (nums[i] > nums[ind1]) {

ind2 = i;

break;

}

}

nums = swap(nums, ind1, ind2);

// step 3 reverse the rest right half

return reverse(nums, ind1 + 1);

}

}

int[] swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

return nums;

}

int[] reverse(int[] nums, int start) {

int i = start;

int j = nums.length - 1;

while (i < j) {

swap(nums, i, j);

i++;

j--;

}

return nums;

}

1. *First Missing Integer*

**Problem Description**

Given an unsorted integer array, **A** of size **N**. Find the first missing positive integer.

**Note:**Your algorithm should run in O(n) time and use constant space.  
  
**Problem Constraints**

1 <= N <= 1000000

-109 <= A[i] <= 109  
  
**Input Format**

First argument is an integer array A.  
  
**Output Format**

Return an integer denoting the first missing positive integer.

**Example Input**

Input 1:

[1, 2, 0]

Input 2:

[3, 4, -1, 1]

Input 3:

[-8, -7, -6]  
  
**Example Output**

Output 1:

3

Output 2:

2

Output 3:

1

public int firstMissingPositive(int[] A) { //TC:O(N) || SC:O(1)

int ans = 1;

if (A.length == 1) {

if (A[0] == 1) {

return ans + 1;

}

return ans;

}

ans = A.length + 1;

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

while (A[i] != i + 1 && A[i] < A.length && A[i] > 0 && A[i] != A[A[i] - 1]) {

int t = A[i];

A[i] = A[t - 1];

A[t - 1] = t;

}

}

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

if (A[i] != i + 1) {

ans = i + 1;

break;

}

}

return ans;

}

1. *Flip*

**Problem Description**

You are given a binary string **A**(i.e., with characters **0** and **1**) consisting of characters **A1**, **A2**, ..., **AN**. In a single operation, you can choose two indices, **L** and **R,** such that **1** ≤ **L** ≤ **R** ≤ **N** and flip the characters **AL**, **AL+1**, ..., **AR**. By flipping, we mean changing character **0** to **1** and vice-versa.

Your aim is to perform **ATMOST** one operation such that in the final string number of **1s** is maximized.

If you don't want to perform the operation, return an **empty** array. Else, return an array consisting of two elements denoting **L** and **R**. If there are multiple solutions, return the **lexicographically smallest** pair of **L** and **R**.

**NOTE:**Pair **(a, b)** is lexicographically smaller than pair **(c, d)** if **a** < **c** or, if **a** == **c** and **b** < **d**.  
  
**Problem Constraints**

1 <= size of string <= 100000  
  
**Input Format**

First and only argument is a string A.  
  
**Output Format**

Return an array of integers denoting the answer.  
  
**Example Input**

Input 1:

A = "010"

Input 2:

A = "111"

**Example Output**

Output 1:

[1, 1]

Output 2:

[]

public int[] flip(String A) { //Follow Kadens Algo //TC:O(N) || SC:O(1)

int l = 0, r = 0, cSum = 0, mSum = 0;

int[] ans = new int[2];

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

char ch = A.charAt(i);

if (ch == '1') {

cSum--;

} else {

cSum++;

}

if (cSum > mSum) {

mSum = cSum;

ans[0] = l + 1;

ans[1] = r + 1;

} else if (cSum < 0) {

cSum = 0;

l = i + 1;

r = i + 1;

} else {

r++;

}

}

return mSum == 0?new int[] {}:ans;

}

1. *Number of Digit One*

**Problem Description**

Given an integer **A**, find and return the total number of digit **1** appearing in all non-negative integers less than or equal to **A**.  
  
**Problem Constraints**

0 <= A <= 109  
  
**Input Format**

The only argument given is the integer A.  
  
**Output Format**

Return the total number of digit 1 appearing in all non-negative integers less than or equal to A.  
  
**Example Input**

Input 1:

A = 10

Input 2:

A = 11  
  
**Example Output**

Output 1:

2

Output 2:

4

public int solve(int A) { // TC:O(1) || SC:O(1)

int ans = 0;

int temp = A;

int i = 1;

while (temp / i > 0) {

int num = (A / (i \* 10)) \* i + Math.min(Math.max((A % (i \* 10) - (i - 1)), 0), i);

ans += num;

i \*= 10;

}

return ans;

}

1. *Subarray Sum Equals K*

**Problem Description**

Given an array of integers **A** and an integer **B**.  
Find the total number of subarrays having sum equals to **B**.  
  
**Problem Constraints**

1 <= length of the array <= 50000  
-1000 <= A[i] <= 1000

**Input Format**

The first argument given is the integer array A.

The second argument given is integer B.  
  
**Output Format**

Return the total number of subarrays having sum equals to B.  
  
**Example Input**

Input 1:

A = [1, 0, 1]

B = 1

Input 2:

A = [0, 0, 0]

B = 0  
  
**Example Output**

Output 1:

4

Output 2:

6

public int solve(ArrayList<Integer> A, int B) { // TC:O(N) || SC:O(N)

int psum = 0;

int count = 0;

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

map.put(0, 1);

for(Integer num : A){

psum = psum + num;

int sum2 = psum - B;

if(map.containsKey(sum2)){

count += map.get(sum2);

}

if(map.containsKey(psum)){

map.put(psum, map.get(psum) + 1);

}

else{

map.put(psum , 1);

}

}

return count;

}

1. *Peak Element*

**Problem Description**

Given an **array** of integers **A**, find and return the **peak element** in it.  
An array element is considered a peak if it is not smaller than its neighbors. For corner elements, we need to consider only one neighbor.  
 **NOTE**:

* It is guaranteed that the array contains only a single peak element.
* Users are expected to solve this in O(log(N)) time. The array may contain duplicate elements.

**Problem Constraints**

1 <= **|A|** <= 100000

1 <= A[i] <= 109  
  
**Input Format**

The only argument given is the integer array A.  
  
**Output Format**

Return the peak element.  
  
**Example Input**

Input 1:

A = [1, 2, 3, 4, 5]

Input 2:

A = [5, 17, 100, 11]  
  
**Example Output**

Output 1:

5

Output 2:

100

public int solve(ArrayList<Integer> A) {

int len = A.size();

if (len == 1) {

return A.get(0);

}

if (A.get(0) >= A.get(1)) {

return A.get(0);

}

if (A.get(len - 1) >= A.get(len - 2)) {

return A.get(len - 1);

}

int l = 1, h = len - 2;

while (l <= h) {

int mid = l + (h - l) / 2;

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

return A.get(mid);

}

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

l = mid + 1;

} else if (A.get(mid) < A.get(mid - 1) && A.get(mid) > A.get(mid + 1)) {

h = mid - 1;

}

}

return -1;

}

1. *Minimum Difference*

**Problem Description**

You are given a 2-D matrix **C** of size **A × B**.  
You need to build a new 1-D array of size **A** by taking **one** element from each row of the **2-D matrix** in such a way that the **cost** of the newly built array is **minimized**.

**The cost** of an array is the **minimum** possible value of the **absolute difference** between any two **adjacent** elements of the array.

So if the newly built array is **X**, the element picked from row 1 will become X[1], element picked from row 2 will become X[2], and so on.

Determine the **minimum cost** of the newly built array.  
  
**Problem Constraints**

2 <= A <= 1000  
2 <= B <= 1000  
1 <= C[i][j] <= 106  
  
**Input Format**

The first argument is an integer A denoting number of rows in the 2-D array.  
The second argument is an integer B denoting number of columns in the 2-D array.  
The third argument is a 2-D array C of size A x B.  
  
**Output Format**

Return an integer denoting the minimum cost of the newly build array.  
  
**Example Input**

Input 1:

A = 2

B = 2

C = [ [8, 4]

[6, 8] ]

Input 2:

A = 3

B = 2

C = [ [7, 3]

[2, 1]

[4, 9] ]  
  
**Example Output**

Output 1:

0

Output 2:

1

public int solve(int A, int B, int[][] C) { //TC:O(A\*B\*logB) || SC:O(1)

int ans = Integer.MAX\_VALUE;

// Sort each row of the matrix.

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

Arrays.sort(C[i]);

}

// For each matrix element

for (int i = 0; i < A - 1; i++) {

for (int j = 0; j < B; j++) {

// Search smallest element in the

// next row which is greater than

// or equal to the current element

int p = bsearch(0, B - 1, C[i][j], C[i + 1]);

ans = Math.min(ans, Math.abs(C[i + 1][p] - C[i][j]));

// largest element which is smaller than the current

// element in the next row must be just before

// smallest element which is greater than or equal

// to the current element because rows are sorted.

if (p - 1 >= 0) {

ans = Math.min(ans, Math.abs(C[i][j] - C[i + 1][p - 1]));

}

}

}

return ans;

}

// Return smallest element greater than

// or equal to the current element.

public int bsearch(int l, int r, int n, int[] A) {

int ans = -1;

while (l <= r) {

int mid = (l + r) / 2;

if (A[mid] == n) {

return mid;

} else if (A[mid] < n) {

l = mid + 1;

} else {

ans = mid;

r = mid - 1;

}

}

// If the target n, is greater than all the array elements, the last element

// in the sorted array is the closest element to n. Only in this case, ans is

// unchanged. In this case, l gets incremented till length of array.

return ans == -1 ? l - 1 : ans;

}

1. *Distinct Numbers in Window*

**Problem Description**

You are given an array of **N** integers, **A1, A2 ,..., AN** and an integer **B**. Return the of count of distinct numbers in all windows of size **B**.

Formally, return an array of size **N-B+1** where **i'th** element in this array contains number of distinct elements in sequence **Ai, Ai+1 ,..., Ai+B-1.**

**NOTE:**if **B** > **N**, return an empty array.  
  
**Problem Constraints**

1 <= **N** <= 1061 <= **A[i]** <= 109  
  
**Input Format**

First argument is an integer array A  
Second argument is an integer B.  
  
**Output Format**

Return an integer array.  
  
**Example Input**

Input 1:

A = [1, 2, 1, 3, 4, 3]

B = 3

Input 2:

A = [1, 1, 2, 2]

B = 1  
  
**Example Output**

Output 1:

[2, 3, 3, 2]

Output 2:

[1, 1, 1, 1]

public ArrayList<Integer> dNums(ArrayList<Integer> A, int B) { //TC:O(N) || SC:O(B)

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

ArrayList<Integer> arr = new ArrayList<>();

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

map.put(A.get(i), map.getOrDefault(A.get(i), 0)+1);

}

arr.add(map.size());

int l=1, r = B;

while(r<A.size()) {

int remove = A.get(l-1);

map.put(remove, map.getOrDefault(remove, 0)-1);

if(map.get(remove)<1) {

map.remove(remove);

}

map.put(A.get(r), map.getOrDefault(A.get(r), 0)+1);

arr.add(map.size());

l++;

r++;

}

return arr;

}

1. *Set Bit – Bit Manipulation*

**Problem Description**

You are given two integers **A** and **B**.  
Set the A-th bit and B-th bit in 0, and return output in decimal Number System.  
  
**Note:**  
The bit positions are 0-indexed, which means that the least significant bit (LSB) has index 0.  
  
**Problem Constraints**

0 <= A <= 30  
0 <= B <= 30

**Input Format**

First argument A is an integer.  
Second argument B is an integer.  
  
**Output Format**

Return an integer.  
  
**Example Input**

Input 1:

A = 3

B = 5

Input 2:

A = 4

B = 4  
  
**Example Output**

Output 1:

40

Output 2:

16

public int solve(int A, int B) { //TC:O(1) || SC:O(1)

int n = 0;

n = (n | (1 << A));

n = (n | (1 << B));

return n;

}

//--------------------//

The value of the A-th bit is 2^A and that

of the B-th bit is 2^B.

We have to set the A-th bit and the B-th bit

in 0. This is similar to directly adding 2^A

and 2^B to 0.

If A = B, then we can just add 2^A to 0.

1. *Unset i-th bit – Bit Manipulation*

**Problem Description**

You are given two integers **A** and **B**.

* If B-th bit in A is set, make it unset.
* If B-th bit in A is unset, leave as it is.

Return the updated A value.  
  
**Note:**  
The bit position is 0-indexed, which means that the least significant bit (LSB) has index 0.  
  
**Problem Constraints**

1 <= A <= 109  
0 <= B <= 30  
  
**Input Format**

First argument A is an integer.  
Second argument B is an integer.  
  
**Output Format**

Return an integer.

**Example Input**

Input 1:

A = 4

B = 1

Input 2:

A = 5

B = 2  
  
**Example Output**

Output 1:

4

Output 2:

1

**Example Explanation**

For Input 1:

Given N = 4 which is 100 in binary. The 1-st bit is already unset

For Input 2:

Given N = 5 which is 101 in binary. We unset the 2-nd bit  
It becomes 001 which is 1 in Decimal.

public int solve(int A, int B) { //TC:O(1) || SC:O(1)

if ((A & (1 << B)) != 0) {

A = A ^ (1 << B);

return A;

}

return A;

}

//--------------------------------------------//

We can find if the B-th bit is set in A by performing

bitwise AND of A and 2^B. If the result is non-zero then

we subtract 2^B from A. If the bitwise AND is zero that means

the B-th bit is already unset. So, then we return A as it is.

1. *Check bit – Bit Manipulation*

**Problem Description**

You are given two integers **A** and **B**.

* Return 1 if B-th bit in A is set
* Return 0 if B-th bit in A is unset

**Note:**  
The bit position is 0-indexed, which means that the least significant bit (LSB) has index 0.  
**Problem Constraints**

1 <= A <= 109  
0 <= B <= 30

**Input Format**

First argument A is an integer.  
Second argument B is an integer.

**Output Format**

Return an integer.

**Example Input**

Input 1:

A = 4

B = 1

Input 2:

A = 5

B = 2

**Example Output**

Output 1:

0

Output 2:

1

**Example Explanation**

For Input 1:

Given N = 4 which is 100 in binary. The 1-st bit is unset

so we return 0

For Input 2:

Given N = 5 which is 101 in binary. The 2-nd bit is set

so we return 1

public int solve(int A, int B) { //TC:O(1) || SC:O(1)

if((A & (1<<B))!=0) {

return 1;

}

return 0;

}

//----------------------------------------//

The value of the B-th bit is 2^B

To find the B-th bit in A, we can directly

perform bitwise AND operation between A and 2^B.

If the bit was unset we get 0 as the result of the

biwise AND and if the bit was set then the result is

2^B

1. *Number of 1 Bits – Bit Manipulation*

**Problem Description**

Write a function that takes an integer and returns the number of 1 bits present in its binary representation.  
  
**Problem Constraints**

1 <= A <= 109  
  
**Input Format**

First and only argument contains integer A  
  
**Output Format**

Return an integer  
  
**Example Input**

**Input 1:**

11

**Input 2:**

6  
  
**Example Output**

**Output 1:**

3

**Output 2:**

2

**Example Explanation**

**Explaination 1:**

11 is represented as 1011 in binary.

**Explaination 2:**

6 is represented as 110 in binary.

public int numSetBits(int A) {

int ans = 0;

while(A>0) {

if((A & 1)==1) {

ans++;

}

A = A>>1;

}

return ans;

}

//----------------------------------------------//

Bruteforce:

Iterate 32 times, each time determining if the ith bit is a ’1′ or not.

This is probably the easiest solution, and the interviewer would probably not be too happy about it.

This solution is also machine dependent (You need to be sure that an unsigned integer is 32-bit).

In addition, this solution is not very efficient too because you need to iterate 32 times no matter what.

A better solution:

This special solution uses a trick which is normally used in bit manipulations.

Notice what x - 1 does to bit representation of x.

x - 1 would find the first set bit from the end, and then set it to 0, and set all the bits following it.

Which means if x = 10101001010100

^

|

|

|

First set bit from the end

Then x - 1 becomes 10101001010(011)

All other bits in x - 1 remain unaffected.

This means that if we do (x & (x - 1)), it would just unset the last set bit in x (which is why x&(x-1) is 0 for powers of 2).

1. *Help From Sam – Bit Manipulation*

**Problem Description**

Alex and Sam are good friends. Alex is doing a lot of programming these days. He has set a target score of **A** for himself.  
Initially, Alex's score was zero. Alex can double his score by doing a question, or Alex can seek help from Sam for doing questions that will contribute **1** to Alex's score. Alex wants his score to be precisely **A**. Also, he does not want to take much help from Sam.  
  
Find and return the minimum number of times Alex needs to take help from Sam to achieve a score of **A**.

**Problem Constraints**

0 <= A <= 10^9

**Input Format**

The only argument given is an integer A.

**Output Format**

Return the minimum number of times help taken from Sam.

**Example Input**

Input 1:

A = 5

Input 2:

A = 3

**Example Output**

Output 1:

2

Output 2:

2

**Example Explanation**

Explanation 1:

Initial score : 0

Takes help from Sam, score : 1

Alex solves a question, score : 2

Alex solves a question, score : 4

Takes help from Sam, score: 5

Explanation 2:

Initial score : 0

Takes help from Sam, score : 1

Alex solves a question, score : 2

Takes help from Sam, score : 3

public int solve(int A) { // Time Complexity : O( log(A) )

int count = 0;

for(int i=0;i<32;i++) {

if((A&(1<<i))!=0) {

count++;

}

}

return count;

}

//--------------------------------------------------//

Claim :The number of times we would require help from sam is the number of bits that are set in A.

Lets try to build an intuition for this. Instead of going from 0 to A , we will go in the reverse direction i.e. from A to 0.

First initialise a cnt variable to 0 which is the number of times we took help from sam.Now we would follow this approach until A becomes 0.

If A is an even number we can divide it by 2

otherwise if A is an odd number we can subtract 1 from it and increment the cnt.

Since every time we divide by 2 if its an even number , it is same as doing a left shift. The number of times A would become odd is the number of set bits of A.

Lets take an example to make this more clear:

A=17

0) 17 ,cnt=0

1) 17 -> 16 , cnt=1

2) 16 -> 8 , cnt=1

3) 8 -> 4 , cnt=1

4) 4 -> 2 , cnt=1

5) 2 -> 1 , cnt=1

6) 1 -> 0 , cnt=2

binary representation of 17 = 10001 , and we can see that the number of set bits is the same as cnt that we obtained in the above approach.

Therefore it is enough to just find the number of set bits of A.

We can do this as follows:

cnt=0

for i from 0 to 31

if A&(1<<i) != 0

cnt++

return cnt

1. *Toggle i-th bit – Bit Manipulation*

**Problem Description**

You are given two integers **A** and **B**.

* If B-th bit in A is set, make it unset
* If B-th bit in A is unset, make it set

Return the updated A value  
  
**Problem Constraints**

1 <= A <= 109  
0 <= B <= 30

**Input Format**

First argument A is an integer.  
Second argument B is an integer.  
  
**Output Format**

Return an integer.  
  
**Example Input**

Input 1:

A = 4

B = 1

Input 2:

A = 5

B = 2  
  
**Example Output**

Output 1:

6

Output 2:

1  
  
**Example Explanation**

For Input 1:

Given N = 4 which is 100 in binary. The 1-st bit is unset

so we make it set

For Input 2:

Given N = 5 which is 101 in binary. The 2-nd bit is set

so we make it unset

public int solve(int A, int B) { //TC:O(1) || SC:O(1)

A = (A^(1<<B));

return A;

}

//------------------------------------------------//

The value of the B-th bit is 2^B

To toggle the B-th bit in A, we can directly

perform bitwise XOR operation between A and 2^B.

If the bit was set, this will unset the bit and

if the bit was unset, then this will set that

bit

1. *Unset x bits from right - Bit Manipulation*

**Problem Description**

Given an integer **A**. Unset **B** bits from the right of **A** in binary.  
  
For example, if A = 93 and B = 4, the binary representation of A is 1011101.  
If we unset the rightmost **4 bits**, we get the binary number 1010000, which is equal to the decimal value**80**.  
  
**Problem Constraints**

1 <= A <= 1018  
1 <= B <= 60  
  
**Input Format**

The first argument is a single integer A.  
The second argument is a single integer B.

**Output Format**

Return the number with B unset bits from the right.

**Example Input**

Input 1:-

A = 25

B = 3

Input 2:-

A = 37

B = 3  
  
**Example Output**

Output 1:-

24

Output 2:-

32

**Example Explanation**

Explanation 1:-

A = 11001 to 11000

Explantio 2:-

A = 100101 to 100000

public Long solve(Long A, int B) {

long ans = A;

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

if ((A & (1 << i)) != 0) {

ans -= 1 << i;

}

}

return ans;

}

//-----------------------------------------------------//

Try using for loops iterate from 0 to B-1 and do the needful.

Initialise a variable ans = A

Iterate from i = 0 to i = B - 1. If ith bit was set in A that is (A & (1<<i)) != 0, ans -= (1<<i), unset it from the answer.

1. *Find nth Magic Number – Bit Manipulation*

**Problem Description**

Given an integer **A**, find and return the **Ath** magic number.

A magic number is defined as a number that can be expressed as a power of 5 or a sum of **unique** powers of 5.

First few magic numbers are 5, 25, 30(5 + 25), 125, 130(125 + 5), ….  
  
**Problem Constraints**

1 <= A <= 5000

**Input Format**

The only argument given is integer A.

**Output Format**

Return the Ath magic number.  
  
**Example Input**

Example Input 1:

A = 3

Example Input 2:

A = 10

**Example Output**

Example Output 1:

30

Example Output 2:

650  
  
**Example Explanation**

Explanation 1:

Magic Numbers in increasing order are [5, 25, 30, 125, 130, ...]

3rd element in this is 30

Explanation 2:

In the sequence shown in explanation 1, 10th element will be 650.

public int solve(int A) {

int ans = 0;

for(int i=0;i<32;i++) {

if((A&(1<<i))!=0) {

ans = (int) (ans + Math.pow(5, i+1));

}

}

return ans;

}

//---------------------------------------//

As we know \*\*5n > 51 + 52 + ... + 5n-1\*\*

So, we can find the sum of all subsets of the first 13 power of 5.

since no element will overlap, we will have 2^13 - 1 elements or 8000 elements.

Simply sort them and answer the query in O(1).

Time Complexity: O(A\*logA).

Else we can use a much faster approach.

We can represent A in its binary representation.

If the ith bit(1 based indexing) is set we will add 5i to our answer.

Time Complexity:- O(log(A))

1. *Pairs with given sum II*

**Problem Description**

Given a sorted array of integers (not necessarily distinct) **A** and an integer **B**, find and return how many pair of integers ( A[i], A[j] ) such that i != j have sum equal to B.

Since the number of such pairs can be very large, return number of such pairs modulo (109 + 7).

**Problem Constraints**

1 <= **|A|** <= 100000

1 <= A[i] <= 10^9

1 <= B <= 10^9

**Input Format**

The first argument given is the integer array A.

The second argument given is integer B.  
  
**Output Format**

Return the number of pairs for which sum is equal to B modulo (10^9+7).  
  
**Example Input**

Input 1:

A = [1, 1, 1]

B = 2

Input 2:

A = [1, 5, 7, 10]

B = 8  
  
**Example Output**

Output 1:

3

Output 2:

1

public int solve(ArrayList<Integer> A, int B) { //TC:(n) || SC:O(1)

int l = 0, r = A.size() - 1;

long ans = 0, mod = 1000000007;

while (l < r) {

int sum = A.get(l) + A.get(r);

if (sum == B) {

if (A.get(l).equals(A.get(r))) {

long elements = r - l + 1;

ans = (ans + (elements \* (elements - 1) / 2) % mod) % mod;

break;

}

int leftElementCount = findFrq(l, A, 'l', A.get(l));

int rightElementCount = findFrq(r, A, 'r', A.get(r));

ans = (ans + (1L \* leftElementCount \* rightElementCount) % mod) % mod;

l += leftElementCount;

r -= rightElementCount;

} else if (sum < B) {

l++;

} else {

r--;

}

}

return (int) ans;

}

public int findFrq(int idx, ArrayList<Integer> A, char dir, int k) {

int frq = 0;

if (dir == 'l') {

while (idx < A.size() && A.get(idx) == k) {

frq++;

idx++;

}

} else {

while (idx >= 0 && A.get(idx) == k) {

frq++;

idx--;

}

}

return frq;

}

}

1. *Connect ropes*

**Problem Description**

You are given an array **A** of integersthat represent the lengths of ropes.

You need to connect these ropes into one rope. The cost of joining two ropes equals the **sum** of their lengths.

Find and return the **minimum cost** to connect these ropes into one rope.

**Problem Constraints**

1 <= length of the array <= 100000  
1 <= A[i] <= 1000  
  
**Input Format**

The only argument given is the integer array A.  
**Output Format**

Return an integer denoting the minimum cost to connect these ropes into one rope.  
**Example Input**

Input 1:

A = [1, 2, 3, 4, 5]

Input 2:

A = [5, 17, 100, 11]  
**Example Output**

Output 1:

33

Output 2:

182

public int solve(int[] A) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

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

minHeap.add(A[i]);

}

int ans = 0;

while (minHeap.size() > 1) {

int x = minHeap.poll();

int y = minHeap.poll();

int z = x + y;

ans += z;

minHeap.add(z);

}

return ans;

}

1. *Maximum array sum after B negations*

**Problem Description**

Given an array of integers **A** and an integer **B**. You must modify the array exactly **B** number of times. In a single modification, we can replace any one array element **A[i]** by **-A[i]**.

You need to perform these modifications in such a way that after **exactly B modifications**, sum of the array must be **maximum**.

**NOTE:** You can perform the modification on the same element multiple times.  
  
**Problem Constraints**

1 <= length of the array <= 5\*105  
1 <= B <= 5 \* 106  
-100 <= A[i] <= 100  
  
**Input Format**

The first argument given is an integer array A.  
The second argument given is an integer B.  
  
**Output Format**

Return an integer denoting the maximum array sum after B modifications.  
  
**Example Input**

Input 1:

A = [24, -68, -29, -9, 84]

B = 4

Input 2:

A = [57, 3, -14, -87, 42, 38, 31, -7, -28, -61]

B = 10  
  
**Example Output**

Output 1:

196

Output 2:

362

public int solve(int[] A, int B) {

PriorityQueue<Integer> p = new PriorityQueue<>();

for (int a : A) {

p.add(a);

}

int i = 0;

while (i < B) {

int x = p.poll();

x = ~(x - 1); //to convert + to – and vice versa

p.add(x);

i++;

}

int sum = 0;

while (p.size() > 0) {

sum += p.poll();

}

return sum;

}

1. *Merge K Sorted Lists*

**Problem Description**

Given a list containing head pointers of **N** sorted linked lists.  
Merge thesegiven sorted linked lists and return them as one sorted list.

**Problem Constraints**

1 <= total number of elements in given linked lists <= 100000

**Input Format**

The first and only argument is a list containing N head pointers.

**Output Format**

Return a pointer to the head of the sorted linked list after merging all the given linked lists.

**Example Input**

Input 1:

1 -> 10 -> 20

4 -> 11 -> 13

3 -> 8 -> 9

Input 2:

10 -> 12

13

5 -> 6

**Example Output**

Output 1:

1 -> 3 -> 4 -> 8 -> 9 -> 10 -> 11 -> 13 -> 20

Output 2:

5 -> 6 -> 10 -> 12 ->13

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* public int val;

\* public ListNode next;

\* ListNode(int x) { val = x; next = null; }

\* }

\*/

public class Solution { // this is with O(N) space

public ListNode mergeKLists(ArrayList<ListNode> a) {

PriorityQueue<Integer> pq = new PriorityQueue<>();

for (ListNode n : a) {

while (n != null) {

int x = n.val;

pq.add(x);

n = n.next;

}

}

ListNode head = new ListNode(pq.poll());

ListNode temp = head;

while(pq.size()>0) {

ListNode n = new ListNode(pq.poll());

temp.next = n;

temp = n;

}

return head;

}

}

import java.util.Comparator;

import java.util.PriorityQueue;

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* public int val;

\* public ListNode next;

\* ListNode(int x) { val = x; next = null; }

\* }

\*/

public class Solution { // this is with O(1) space

public ListNode mergeKLists(ArrayList<ListNode> a) {

// Custom comparator to compare ListNode based on their values

Comparator<ListNode> comparator = new Comparator<ListNode>() {

@Override

public int compare(ListNode n1, ListNode n2) {

return Integer.compare(n1.val, n2.val);

}

};

// Priority queue to store ListNode based on their values

PriorityQueue<ListNode> minHeap = new PriorityQueue<>(a.size(), comparator);

// Add the head of each linked list to the minHeap

for (ListNode node : a) {

if (node != null) {

minHeap.add(node);

}

}

ListNode dummy = new ListNode(0);

ListNode current = dummy;

// Process the minHeap until it is empty

while (!minHeap.isEmpty()) {

// Get the smallest element from the minHeap

ListNode smallest = minHeap.poll();

current.next = smallest;

current = current.next;

// Move to the next node in the linked list of the smallest element

if (smallest.next != null) {

minHeap.add(smallest.next);

}

}

return dummy.next;

}

}

1. *Misha and Candies*

**Problem Description**

Misha loves eating candies. She has been given **N** boxes of Candies.

She decides that every time she will choose a box having the **minimum** number of candies, eat **half** of the candies and put the remaining candies in the **other box** that has the **minimum** number of candies.  
Misha does not like a box if it has the number of candies **greater than B** so she won't eat from that box. Can you find how many candies she will eat?

**NOTE 1:** If a box has an odd number of candies then Misha will eat the **floor (odd / 2)**.

**NOTE 2:** The same box will not be chosen again.  
  
**Problem Constraints**

1 <= N <= 105

1 <= A[i] <= 105

1 <= B <= 106  
  
**Input Format**

The first argument is A an Array of Integers, where A[i] is the number of candies in the ith box.  
The second argument is B, the maximum number of candies Misha like in a box.

**Output Format**

Return an integer denoting the number of candies Misha will eat.

**Example Input**

Input 1:

A = [3, 2, 3]

B = 4

Input 2:

A = [1, 2, 1]

B = 2

**Example Output**

Output 1:

2

Output 2:

1

public int solve(int[] A, int B) {

PriorityQueue<Integer> boxes = new PriorityQueue<>();

for (int box = 0; box < A.length; box++) {

boxes.add(A[box]);

}

int candies\_eaten = 0;

while (boxes.size() > 0 && boxes.peek() <= B) {

int candies = boxes.poll();

int candies\_pick = candies / 2;

candies\_eaten += candies\_pick;

if (boxes.size() != 0) {

int min\_candies = boxes.poll();

min\_candies += candies - candies\_pick;

boxes.add(min\_candies);

}

}

return candies\_eaten;

}

1. *Build a Heap*

**Problem Description**

Given an array **A** of **N** integers, convert that array into a min heap and return the array.

NOTE: A min heap is a binary tree where every node has a value less than or equal to its children.  
  
**Problem Constraints**

1 ≤ N ≤ 105

1 ≤ A[i] ≤ 109  
  
**Input Format**

First and only argument of input contains a single integer array A of length N.  
  
**Output Format**

Return the reordered array A such that it forms a min heap.  
  
**Example Input**

Input:

A = [5, 13, -2, 11, 27, 31, 0, 19]

**Example Output**

Output:

A = [-2 11 0 13 27 31 5 19]

public int[] buildHeap(int[] A) { //TC:O(n) || SC:O(1)

int n = A.length;

for (int i = (n / 2) - 1; i >= 0; i--) {

heapify(A, i);

}

return A;

}

public void heapify(int[] A, int i) {

int n = A.length;

while (i < n) {

int leftChild = (2 \* i) + 1;

int rightChild = (2 \* i) + 2;

int x = A[i];

if (leftChild < n) {

x = Math.min(x, A[leftChild]);

}

if (rightChild < n) {

x = Math.min(x, A[rightChild]);

}

if (x == A[i]) {

return;

}

if (leftChild < n && x == A[leftChild]) {

swap(A, i, leftChild);

i = leftChild;

}

if (rightChild < n && x == A[rightChild]) {

swap(A, i, rightChild);

i = rightChild;

}

}

}

public void swap(int[] A, int x, int y) {

int temp = A[x];

A[x] = A[y];

A[y] = temp;

}

1. *Heap Queries*

**Problem Description**

You have an empty min heap. You are given an array **A** consisting of **N** queries. Let P denote A[i][0] and Q denote A[i][1]. There are two types of queries:  
  
P = 1, Q = -1 : Pop the minimum element from the heap.  
P = 2, 1 <= Q <= 109 : Insert Q into the heap.

Return an integer array containing the answer for all the extract min operation. If the size of heap is 0, then extract min should return -1.  
  
**Problem Constraints**

1 <= N <= 105

1 <= A[i][0] <= 2

1 <= A[i][1] <= 109 or A[i][1] = -1  
  
**Input Format**

The only argument A is a 2D integer array

**Output Format**

Return an integer array  
  
**Example Input**

Input 1:

A = [[1, -1], [2, 2], [2, 1], [1, -1]]

Input 2:

A = [[2, 5], [2, 3], [2, 1], [1, -1], [1, -1]]  
  
**Example Output**

Output 1:

[-1, 1]

Output 2:

[1, 3]

public int[] solve(int[][] A) {

PriorityQueue<Integer> pq = new PriorityQueue<>();

List<Integer> arr = new ArrayList<>();

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

if (A[i][0] == 1 && A[i][1] == -1) {

if (pq.size() < 1) {

arr.add(-1);

} else {

arr.add(pq.poll());

}

} else if (A[i][0] == 2) {

pq.add(A[i][1]);

}

}

return arr.stream().mapToInt(i -> i).toArray();

}

1. *Ath largest element*

**Problem Description**

Given an integer array **B** of size **N**.

You need to find the **Ath** largest element in the **subarray [1 to i],** where **i** varies from **1** to **N**. In other words, find the **Ath** largest element in the sub-arrays **[1 : 1], [1 : 2], [1 : 3], ...., [1 : N]**.

**NOTE:** If any subarray [1 : i] has less than A elements, then the output should be **-1** at the **ith** index.  
  
**Problem Constraints**

1 <= N <= 100000  
1 <= A <= N  
1 <= B[i] <= INT\_MAX  
  
**Input Format**

The first argument is an integer A.  
The second argument is an integer array B of size N.

**Output Format**

Return an integer array C, where C[i] (1 <= i <= N) will have the Ath largest element in the subarray [1 : i].

**Example Input**

Input 1:

A = 4

B = [1 2 3 4 5 6]

Input 2:

A = 2

B = [15, 20, 99, 1]  
  
**Example Output**

Output 1:

[-1, -1, -1, 1, 2, 3]

Output 2:

[-1, 15, 20, 20]

public int[] solve(int A, int[] B) { //TC : O(KlogK) + (N-K)logK = O(NlogK) || SC:O(K)

int arr[] = new int[B.length];

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

arr[i] = -1;

}

PriorityQueue<Integer> pq = new PriorityQueue<>();

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

pq.add(B[i]);

}

arr[A - 1] = pq.peek();

for (int i = A; i < B.length; i++) {

int minInHeap = pq.peek();

int nextElement = B[i];

if (nextElement > minInHeap) {

pq.poll();

pq.add(B[i]);

arr[i] = pq.peek();

} else {

arr[i] = pq.peek();

}

}

return arr;

}

1. *K Places Apart*

**Problem Description**

**N** people having different priorities are standing in a queue.

The queue follows the **property** that each person is standing at most **B** places away from its position in the sorted queue.

Your **task** is to sort the queue in the increasing order of priorities.

**NOTE:**

* No two persons can have the same priority.
* Use the property of the queue to sort the queue with complexity O(NlogB).

**Problem Constraints**

1 <= N <= 100000  
0 <= B <= N  
  
**Input Format**

The first argument is an integer array A representing the priorities and initial order of N persons.  
The second argument is an integer B.  
  
**Output Format**

Return an integer array representing the sorted queue.  
  
**Example Input**

Input 1:

A = [1, 40, 2, 3]

B = 2

Input 2:

A = [2, 1, 17, 10, 21, 95]

B = 1  
  
**Example Output**

Output 1:

[1, 2, 3, 40]

Output 2:

[1, 2, 10, 17, 21, 95]

public int[] solve(int[] A, int B) { // TC:O(nlogk) || SC:O(k)

PriorityQueue<Integer> pq = new PriorityQueue<>();

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

pq.add(A[i]);

}

int ans[] = new int[A.length];

ans[0] = pq.poll();

int idx = 1;

for (int i = B + 1; i < A.length; i++) {

pq.add(A[i]);

ans[idx++] = pq.poll();

}

while (pq.size() > 0) {

ans[idx++] = pq.poll();

}

return ans;

}

1. *Product of 3*

**Problem Description**

Given an integer array **A** of size **N**.

You have to find the product of the **three** largest integers in array A from range **1** to **i**, where **i** goes from **1** to **N**.

Return an array **B** where **B[i]** is the product of the largest **3** integers in range **1** to **i** in array **A**. If **i < 3**, then the integer at index **i**in**B** should be **-1**.  
  
**Problem Constraints**

1 <= N <= 105  
0 <= A[i] <= 103  
  
**Input Format**

First and only argument is an integer array A.

**Output Format**

Return an integer array B. B[i] denotes the product of the largest 3 integers in range 1 to i in array A.  
  
**Example Input**

Input 1:

A = [1, 2, 3, 4, 5]

Input 2:

A = [10, 2, 13, 4]  
  
**Example Output**

Output 1:

[-1, -1, 6, 24, 60]

Output 2:

[-1, -1, 260, 520]

public int[] solve(int[] A) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

int mul = 1;

int ans[] = new int[A.length];

for (int i = 0; i < 3; i++) {

minHeap.add(A[i]);

mul \*= A[i];

ans[i] = -1;

if (i == 2) {

ans[i] = mul;

}

}

for (int i = 3; i < A.length; i++) {

if (A[i] < minHeap.peek()) {

ans[i] = mul;

} else {

int min = minHeap.poll();

if (min != 0) {

mul = mul / min;

}

minHeap.add(A[i]);

mul \*= A[i];

ans[i] = mul;

}

}

return ans;

}

1. *Kth Smallest Element in a Sorted Matrix*

**Problem Description**

Given a sorted matrix of integers **A** of size **N x M** and an integer **B**.

Each of the rows and columns of matrix **A** is sorted in ascending order, find the **Bth smallest** element in the matrix.

**NOTE:** Return The **Bth** smallest element in the sorted order, not the **Bth** distinct element.  
  
**Problem Constraints**

1 <= N, M <= 500

1 <= A[i] <= 109

1 <= B <= N \* M  
  
**Input Format**

The first argument given is the integer matrix A.  
The second argument given is an integer B.  
  
**Output Format**

Return the B-th smallest element in the matrix.  
  
**Example Input**

Input 1:

A = [ [9, 11, 15],

[10, 15, 17] ]

B = 6

Input 2:

A = [ [5, 9, 11],

[9, 11, 13],

[10, 12, 15],

[13, 14, 16],

[16, 20, 21] ]

B = 12  
  
**Example Output**

Output 1:

17

Output 2:

16

public int solve(int[][] A, int B) { //Time Complexity: O(N×M×logB)

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

int count = 0;

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

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

if (count < B) {

maxHeap.add(A[i][j]);

count++;

} else if (count == B) {

if (maxHeap.peek() > A[i][j]) {

maxHeap.poll();

maxHeap.add(A[i][j]);

}

}

}

}

return maxHeap.peek();

}

We will use Max-Heap to solve this problem.

Create a Max-Heap of size B and process the element of matrix in it.

If the size of the heap is less than B, then push the element inside it.

Once the size of the heap is equal to B, then if the top element in the heap is greater than the element of the matrix, pop the element from the heap and insert the element of the matrix in the Heap.

The size of the heap still remains the same, i.e., B.

In the end, Return the top element of the Heap.

Time Complexity: O(N×M×logB)

1. *Free Cars*

**Problem Description**

Given two arrays, **A** and **B** of size **N**. **A[i]** represents the time by which you can buy the **ith** car without paying any money.

**B[i]** represents the profit you can earn by buying the **ith** car. It takes **1** minute to buy a car, so you can only buy the **ith** car when the **current time <= A[i] - 1**.

Your task is to find the **maximum profit** one can earn by buying cars considering that you can only buy **one car at a time.**

**NOTE:**

* You can start buying from time = 0.
* Return your answer modulo 109 + 7.

**Problem Constraints**

1 <= N <= 105  
1 <= A[i] <= 109  
0 <= B[i] <= 109  
  
**Input Format**

The first argument is an integer array A represents the deadline for buying the cars.  
The second argument is an integer array B represents the profit obtained after buying the cars.

**Output Format**

Return an integer denoting the maximum profit you can earn.

**Example Input**

Input 1:

A = [1, 3, 2, 3, 3]

B = [5, 6, 1, 3, 9]

Input 2:

A = [3, 8, 7, 5]

B = [3, 1, 7, 19]

**Example Output**

Output 1:

20

Output 2:

30

public class FreeCars { //TC:O(NlogN) || SC:O(N)

class Pair {

int time;

int profit;

Pair(int time, int profit) {

this.time = time;

this.profit = profit;

}

}

public int solve(int[] A, int[] B) {

ArrayList<Pair> cars = new ArrayList<Pair>();

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

cars.add(new Pair(A[i], B[i]));

}

Collections.sort(cars, new Comparator<Pair>() {

public int compare(Pair p1, Pair p2) {

if (p1.time == p2.time) {

return Integer.compare(p2.profit, p1.profit);

}

return Integer.compare(p1.time, p2.time);

}

});

PriorityQueue<Integer> pq = new PriorityQueue<>();

int t = 0;

for (int i = 0; i < cars.size(); i++) {

if (t < cars.get(i).time) {

pq.add(cars.get(i).profit);

t++;

} else {

int min = pq.peek();

if (cars.get(i).profit > min) {

pq.poll();

pq.add(cars.get(i).profit);

}

}

}

int sum = 0;

while (!pq.isEmpty()) {

sum = (sum + pq.poll()) % 1000000007;

}

return sum;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

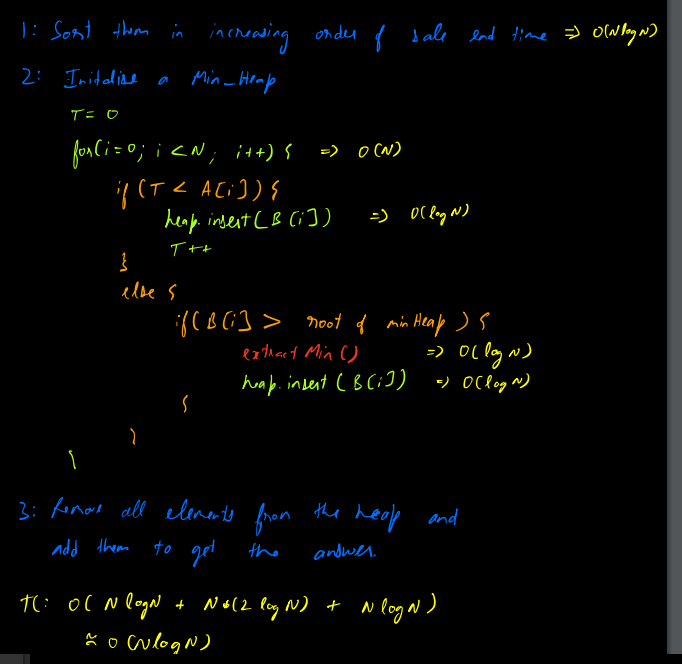
FreeCars f = new FreeCars();

System.out.println(f.solve(new int[] { 1, 3, 2, 3, 3 }, new int[] { 5, 6, 1, 3, 9 }));

System.out.println(f.solve(new int[] { 3, 8, 7, 5 }, new int[] { 3, 1, 7, 19 }));

}

}



1. *Finish Maximum Jobs*

**Problem Description**

There are **N** jobs to be done, but you can do only one job at a time.

Given an array **A** denoting the start time of the jobs and an array **B** denoting the finish time of the jobs.

Your aim is to select jobs in such a way so that you can finish the **maximum** number of jobs.  
  
Return the **maximum** number of jobs you can finish.

**Problem Constraints**

1 <= N <= 105

1 <= A[i] < B[i] <= 109  
  
**Input Format**

The first argument is an integer array A of size N, denoting the start time of the jobs.  
The second argument is an integer array B of size N, denoting the finish time of the jobs.

**Output Format**

Return an integer denoting the maximum number of jobs you can finish.

**Example Input**

Input 1:

A = [1, 5, 7, 1]

B = [7, 8, 8, 8]

Input 2:

A = [3, 2, 6]

B = [9, 8, 9]  
  
**Example Output**

Output 1:

2

Output 2:

1

public class FinishMaximumJobs { TC:O(NlogN) || SC:O(N)

class Pair {

int sTime;

int eTime;

Pair(int sTime, int eTime) {

this.sTime = sTime;

this.eTime = eTime;

}

}

public int solve(int[] A, int[] B) {

List<Pair> jobs = new ArrayList<>();

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

jobs.add(new Pair(A[i], B[i]));

}

Collections.sort(jobs, new Comparator<Pair>() {

public int compare(Pair p1, Pair p2) {

if (p1.eTime == p2.eTime) {

return Integer.compare(p2.sTime, p1.sTime);

}

return Integer.compare(p1.eTime, p2.eTime);

}

});

int jCount = 1;

int lastEndTime = jobs.get(0).eTime;

for (int i = 1; i < jobs.size(); i++) {

if (jobs.get(i).sTime >= lastEndTime) {

jCount++;

lastEndTime = jobs.get(i).eTime;

}

}

return jCount;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

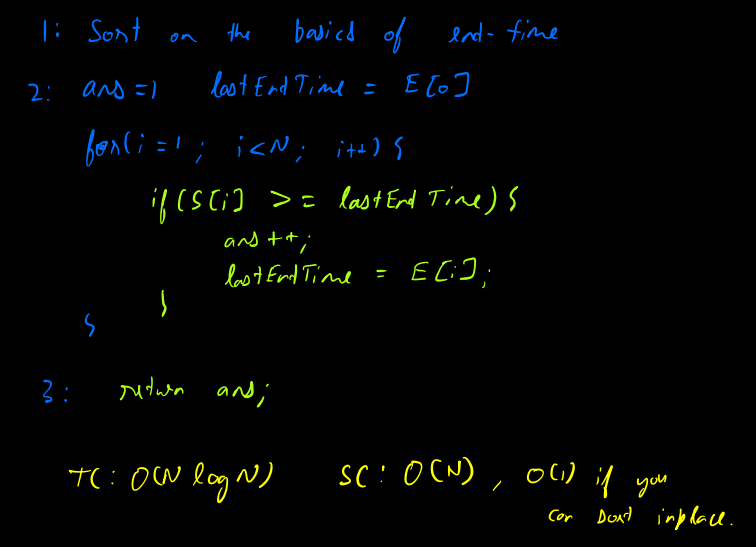
FinishMaximumJobs f = new FinishMaximumJobs();

System.out.println(f.solve(new int[] { 1, 5, 7, 1 }, new int[] { 7, 8, 8, 8 }));

System.out.println(f.solve(new int[] { 3, 2, 6 }, new int[] { 9, 8, 9 }));

}

}



1. *Running Median.*

**Problem Description**

Given an array of integers, **A** denoting a stream of integers. New arrays of integer **B** and **C** are formed.  
Each time an integer is encountered in a stream, append it at the end of **B** and append the median of array **B** at the **C**.

Find and return the array **C**.

**NOTE:**

* If the number of elements is **N** in **B** and **N** is odd, then consider the median as **B[N/2]** ( B must be in sorted order).
* If the number of elements is **N** in **B** and **N** is even, then consider the median as **B[N/2-1].** ( B must be in sorted order).

**Problem Constraints**

1 <= length of the array <= 100000  
1 <= A[i] <= 109

**Input Format**

The only argument given is the integer array A.

**Output Format**

Return an integer array C, C[i] denotes the median of the first i elements.

**Example Input**

Input 1:

A = [1, 2, 5, 4, 3]

Input 2:

A = [5, 17, 100, 11]

**Example Output**

Output 1:

[1, 1, 2, 2, 3]

Output 2:

[5, 5, 17, 11]

public class Solution {

static PriorityQueue<Integer> max\_heap;

static PriorityQueue<Integer> min\_heap;

public int[] solve(int[] A) {

min\_heap = new PriorityQueue<>();

max\_heap = new PriorityQueue<>(Collections.reverseOrder());

int n = A.length;

int ans[] = new int[n];

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

add(A[i]);

ans[i] = get\_median();

}

return ans;

}

public static int get\_median() {

int total = min\_heap.size() + max\_heap.size();

int ret;

if (total % 2 == 1) {

if (max\_heap.size() > min\_heap.size())

ret = max\_heap.peek();

else

ret = min\_heap.peek();

} else {

ret = Integer.MAX\_VALUE;

if (max\_heap.size() != 0)

ret = Math.min(ret, max\_heap.peek());

if (min\_heap.size() != 0)

ret = Math.min(ret, min\_heap.peek());

}

return ret;

}

public static void add(int a) {

if (max\_heap.size() != 0 && (a >= max\_heap.peek()))

min\_heap.offer(a);

else

max\_heap.offer(a);

if (Math.abs(max\_heap.size() - min\_heap.size()) > 1) {

if (max\_heap.size() > min\_heap.size()) {

int temp = max\_heap.peek();

max\_heap.poll();

min\_heap.offer(temp);

} else {

int temp = min\_heap.peek();

min\_heap.poll();

max\_heap.offer(temp);

}

}

}

}

// Logic

two heaps, a max-heap (max\_heap) and a min-heap (min\_heap), to efficiently calculate the running median as elements are added to the stream. Here's the logic and intuition behind each part of the code:

Logic:

Initialization:

Two heaps are used to keep track of the smaller and larger halves of the stream of integers. max\_heap is used as a max-heap, and min\_heap is used as a min-heap.

The CustomComp class implements a comparator for max\_heap to make it behave as a max-heap.

Method solve(int[] A):

Initializes the heaps and an array ans to store the results.

Iterates through the elements of the input array A.

For each element, it adds the element to the heaps and calculates the running median using the get\_median() method.

Method get\_median():

Calculates the running median based on the elements in max\_heap and min\_heap.

If the total number of elements is odd, the median is the top element of the larger heap. If it's even, the median is the average of the tops of both heaps.

Special handling for cases where one heap is empty (Integer.MAX\_VALUE is used to ensure it doesn't interfere in calculations).

Method add(int a):

Adds an element a to either max\_heap or min\_heap based on certain conditions.

If max\_heap is not empty and a is greater than or equal to the top element of max\_heap, a is added to min\_heap. Otherwise, it is added to max\_heap.

Balances the heaps by moving an element from the larger heap to the smaller heap if the size difference between the two heaps becomes greater than 1.

Custom Comparator CustomComp:

Implements a comparator for max\_heap to make it behave as a max-heap.

Intuition:

The idea behind using two heaps is to efficiently maintain the smaller and larger halves of the stream, allowing for quick access to the median.

The max-heap (max\_heap) is used to store the smaller half of the elements, and the min-heap (min\_heap) is used to store the larger half.

By maintaining the balance between the two heaps, the running median can be efficiently calculated without sorting the entire array.

Adding elements to the heaps and ensuring their balance ensures that the median can be determined in constant time.

The use of heaps provides a more efficient approach than sorting the entire array for each calculation, especially when dealing with a continuous stream of elements.

1. *Distribute Candy*

**Problem Description**

N children are standing in a line. Each child is assigned a **rating** value.

You are giving candies to these children subjected to the following requirements:

1. Each child must have at least one candy.
2. Children with a higher rating get more candies than their neighbors.

What is the minimum number of candies you must give?

**Problem Constraints**

1 <= N <= 105  
-109 <= A[i] <= 109

**Input Format**

The first and only argument is an integer array A representing the rating of children.

**Output Format**

Return an integer representing the minimum candies to be given.

**Example Input**

Input 1:

A = [1, 2]

Input 2:

A = [1, 5, 2, 1]

**Example Output**

Output 1:

3

Output 2:

7

public int candy(int[] A) { //TC:O(N) || SC:O(1)

int n = A.length;

int[] c = new int[n];

**// 1. Distribute 1 candy to everyone**

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

c[i] = 1;

}

**// 2. Go from left to right - compare with left neighbour and change the candies**

// accordingly.

for (int i = 1; i < n; i++) {

if (A[i] > A[i - 1]) {

c[i] = c[i - 1] + 1;

}

}

// **3. Go from right to left - compare with right neighbour and change the**

**// candies accordingly.**

for (int i = n - 2; i >= 0; i--) {

if (A[i] > A[i + 1] && c[i] <= c[i + 1]) {

c[i] = c[i + 1] + 1;

}

}

// **4. Total of array candies is the answer**

int sum = 0;

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

sum += c[i];

}

return sum;

}

1. *Fibonacci Number (Dynamic Programming)*

**Problem Description**

Given a positive integer **A**, write a program to find the **Ath** Fibonacci number.

In a Fibonacci series, each term is the sum of the previous two terms and the first two terms of the series are **0** and **1**. i.e. **f(0) = 0** and **f(1) = 1**. Hence, **f(2) = 1, f(3) = 2, f(4) = 3** and so on.

**NOTE:** 0th term is 0. 1th term is 1 and so on.

**Problem Constraints**

0 <= A <= 44

**Input Format**

First and only argument is an integer A.

**Output Format**

Return an integer denoting the **Ath** Fibonacci number.

**Example Input**

Input 1:

A = 4

Input 2:

A = 6  
  
**Example Output**

Output 1:

3

Output 2:

8

public class Main { //TC:O(N) || SC:O(N)

public int fibo(int A, int []arr) {

if(A<=1) {

return A;

}

if(arr[A]!=-1) {

return arr[A];

}

arr[A] = fibo(A-1,arr) + fibo(A-2,arr);

return arr[A];

}

public static void main(String[] args) {

// YOUR CODE GOES HERE

// Please take input and print output to standard input/output (stdin/stdout)

// DO NOT USE ARGUMENTS FOR INPUTS

// E.g. 'Scanner' for input & 'System.out' for output

Scanner sc = new Scanner(System.in);

int A=sc.nextInt();

int arr[] = new int[A+1];

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

arr[i] = -1;

}

Main f = new Main();

System.out.println(f.fibo(A,arr));

}

}

1. *Stairs (Dynamic Programming)*

**Problem Description**

You are climbing a staircase and it takes **A** steps to reach the top.

Each time you can either climb **1** or **2** steps. In how many **distinct ways** can you climb to the top?

Return the number of distinct ways modulo 1000000007  
  
**Problem Constraints**

1 <= A <= 105

**Input Format**

The first and the only argument contains an integer A, the number of steps.

**Output Format**

Return an integer, representing the number of ways to reach the top.

**Example Input**

Input 1:

A = 2

Input 2:

A = 3

**Example Output**

Output 1:

2

Output 2:

3

public int climbStairs(int A) { //TC:O(N) || SC:O(1)

int prev = 1, prevToPrev = 1;

if (A < 2) {

return 1;

}

int curr = 0;

for (int i = 2; i <= A; i++) {

curr = (prev + prevToPrev) % 1000000007;

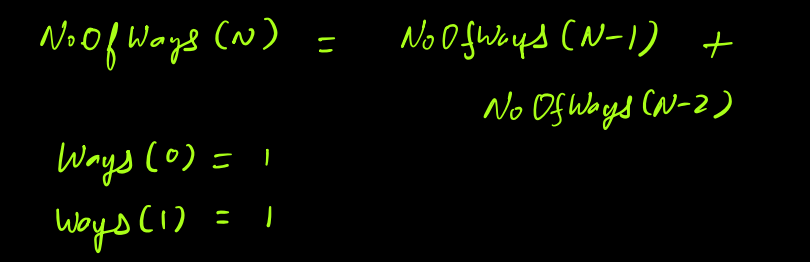
prevToPrev = prev;

prev = curr;

}

return curr;

}

 this is actually a Fibonacci sequence.

1. *Minimum Number of Squares (Dynamic Programming)*

**Problem Description**

Given an integer **A**. Return **minimum** count of numbers, sum of whose **squares** is equal to **A**.

**Problem Constraints**

1 <= A <= 105

**Input Format**

First and only argument is an integer A.

**Output Format**

Return an integer denoting the minimum count.

**Example Input**

Input 1:

A = 6

Input 2:

A = 5

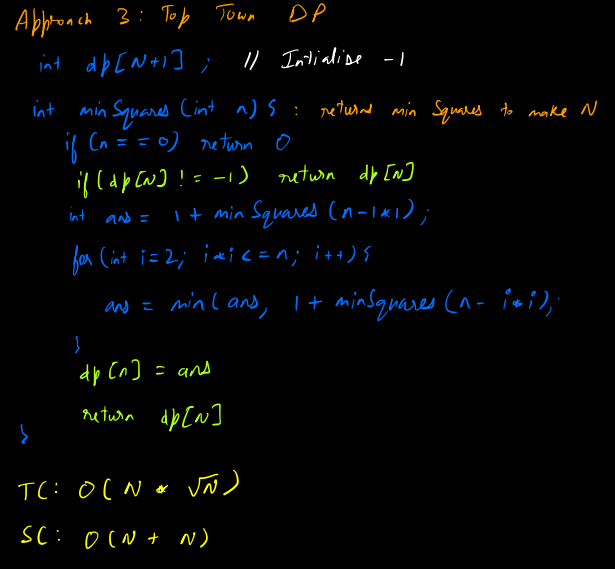
**Example Output**

Output 1:

3

Output 2:

2



//Bottom Up

public int countMinSquares(int A) { //TC:O(N \* sqrtN) || SC:O(N)

int arr[] = new int[A + 1];

arr[0] = 0;

for (int i = 1; i <= A; i++) {

arr[i] = i;

for (int j = 1; j \* j <= i; j++) {

arr[i] = Math.min(arr[i], 1 + arr[i - j \* j]);

}

}

return arr[A];

}

1. *Max Sum Without Adjacent Elements (Dynamic Prog)*

**Problem Description**

Given a **2 x N** grid of integers, **A**, your task is to choose numbers from the grid such that **sum** of these numbers is **maximized**.   
However, you cannot choose two numbers that are adjacent horizontally, vertically, or diagonally.   
  
Return the maximum possible sum.  
  
**Note:** You are allowed to choose more than 2 numbers from the grid.

**Problem Constraints**

1 <= N <= 20000  
1 <= A[i] <= 2000

**Input Format**

The first and the only argument of input contains a 2d matrix, A.

**Output Format**

Return an integer, representing the maximum possible sum.

**Example Input**

Input 1:

A = [

[1]

[2]

]

Input 2:

A = [

[1, 2, 3, 4]

[2, 3, 4, 5]

]

**Example Output**

Output 1:

2

Output 2:

8

public int adjacent(int[][] A) {

int arr[] = new int[A[0].length];

int max = Integer.MIN\_VALUE;

// Since we know we have only 2 row, so it can be safe to say, from one column,

// only the max value can be considered. SO we convert the 2D array to 1D array containing

// the max value of all the columns

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

max = Math.max(A[0][j], A[1][j]);

arr[j] = max;

}

// From the 1D array created, the max value till a position i can be either

// 1. the max value till the last position (i-1)

// OR

// 2. the max value till i-2 poition + the current value.

// Once we find the max value for all positions i, the value at the last position

// will the max value that can be created from the array and hence the output.

int maxV[] = new int[A[0].length];

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

if (i == 0) {

maxV[i] = arr[i];

} else if (i == 1) {

maxV[i] = Math.max(arr[i - 1], arr[i]);

} else {

maxV[i] = Math.max(maxV[i - 1], maxV[i - 2] + arr[i]);

}

}

return maxV[A[0].length - 1];

}

1. *Dungeon Princess (Dynamic Programming)*

**Problem Description**

The demons had captured the **princess** and imprisoned her in the **bottom-right** corner of a dungeon. The dungeon consists of **M x N** rooms laid out in a 2D grid. Our valiant **knight** was initially positioned in the **top-left** room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only **rightward or downward** in each step.

Given a 2D array of integers **A** of size **M x N**. Find and return the knight's **minimum** initial health so that he is able to rescue the princess.

**Problem Constraints**

1 <= M, N <= 500

-100 <= A[i] <= 100  
  
**Input Format**

First and only argument is a 2D integer array A denoting the grid of size M x N.

**Output Format**

Return an integer denoting the knight's **minimum** initial health so that he is able to rescue the princess.

**Example Input**

Input 1:

A = [

[-2, -3, 3],

[-5, -10, 1],

[10, 30, -5]

]

Input 2:

A = [

[1, -1, 0],

[-1, 1, -1],

[1, 0, -1]

]

**Example Output**

Output 1:

7

Output 2:

1

public int calculateMinimumHP(int[][] A) { // TC:O(N\*M) || SC:O(N\*M)

int row = A.length;

int col = A[0].length;

int dp[][] = new int[row][col];

for (int i = row - 1; i >= 0; i--) {

for (int j = col - 1; j >= 0; j--) {

if (i == row - 1 && j == col - 1) {

if (A[i][j] < 0) {

dp[i][j] = Math.abs(A[i][j]) + 1;

} else {

dp[i][j] = 1;

}

} else if (i == row - 1) {

dp[i][j] = (dp[i][j + 1] - A[i][j]) > 0 ? (dp[i][j + 1] - A[i][j]) : 1;

} else if (j == col - 1) {

dp[i][j] = (dp[i + 1][j] - A[i][j]) > 0 ? (dp[i + 1][j] - A[i][j]) : 1;

} else {

dp[i][j] = (Math.min(dp[i + 1][j], dp[i][j + 1]) - A[i][j]) > 0

? (Math.min(dp[i + 1][j], dp[i][j + 1]) - A[i][j])

: 1;

}

}

}

return dp[0][0];

}

1. *Unique Paths in a Grid (Dynamic Programming)*

**Problem Description**

Given a grid of size **n \* m**, lets assume you are starting at **(1,1)** and your goal is to reach **(n, m)**.   
At any instance, if you are on **(x, y)**, you can either go to **(x, y + 1)** or **(x + 1, y)**.  
  
Now consider if some obstacles are added to the grids.   
Return the total number unique paths from (1, 1) to (n, m).  
  
**Note:**   
1. An obstacle is marked as **1** and empty space is marked **0** respectively in the grid.  
2. Given Source Point and Destination points are 1-based index.

**Problem Constraints**

1 <= n, m <= 100  
A[i][j] = 0 or 1

**Input Format**

Firts and only argument A is a 2D array of size n \* m.

**Output Format**

Return an integer denoting the number of unique paths from (1, 1) to (n, m).

**Example Input**

Input 1:

A = [

[0, 0, 0]

[0, 1, 0]

[0, 0, 0]

]

Input 2:

A = [

[0, 0, 0]

[1, 1, 1]

[0, 0, 0]

]

**Example Output**

Output 1:

2

Output 2:

0

public int uniquePathsWithObstacles(int[][] A) { //TC:O(N\*M) ||SC:O(N\*M)

int dp[][] = new int[A.length][A[0].length];

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

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

if(A[i][j] == 1) {

dp[i][j] = 0;

}else {

if(i==0) {

dp[i][j] = j==0?1:dp[i][j-1];

}else if(j==0) {

dp[i][j] = i==0?1:dp[i-1][j];

}else {

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

}

}

}

}

return dp[A.length-1][A[0].length-1];

}

1. *Reverse Linked List*

**Problem Description**

You are given a singly linked list having head node **A**. You have to reverse the linked list and return the head node of that reversed list.

**NOTE:** You have to do it **in-place** and in **one-pass**.

**Problem Constraints**

1 <= Length of linked list <= 105

Value of each node is within the range of a 32-bit integer.

**Input Format**

First and only argument is a linked-list node **A**.

**Output Format**

Return a linked-list node denoting the head of the reversed linked list.

**Example Input**

Input 1:

A = 1 -> 2 -> 3 -> 4 -> 5 -> NULL

Input 2:

A = 3 -> NULL

**Example Output**

Output 1:

5 -> 4 -> 3 -> 2 -> 1 -> NULL

Output 2:

3 -> NULL

public ListNode reverseList(ListNode A) { //TC:O(N) || SC:O(1)

ListNode head = A;

if (head == null || head.next == null) {

return head;

}

ListNode curr = head;

ListNode prev = null;

while (curr != null) {

ListNode next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

head = prev;

return head;

}

1. *Palindrome List*

**Problem Description**

Given a singly linked list **A**, determine if it's a palindrome. Return **1** or **0,** denoting if it's a palindrome or not, respectively.

**Problem Constraints**

1 <= |A| <= 105

**Input Format**

The first and the only argument of input contains a pointer to the head of the given linked list.

**Output Format**

Return 0, if the linked list is not a palindrome.

Return 1, if the linked list is a palindrome.

**Example Input**

Input 1:

A = [1, 2, 2, 1]

Input 2:

A = [1, 3, 2]

**Example Output**

Output 1:

1

Output 2:

0

**public int lPalin(ListNode A)** { //TC:O(N) || SC:O(1)

int len = 0;

ListNode temp = A;

while (temp.next != null) {

temp = temp.next;

len++;

}

int mid = (len + 1) / 2;

ListNode tempHead = A;

for (int i = 1; i < mid; i++) {

tempHead = tempHead.next;

}

ListNode head1 = A;

ListNode head2 = reverseList(tempHead.next);

while (head1 != null && head2 != null) {

if (head1.val != head2.val) {

ListNode head3 = reverseList(head2);

tempHead.next = head3;

return 0;

}

head1 = head1.next;

head2 = head2.next;

}

ListNode head3 = reverseList(head2);

tempHead.next = head3;

return 1;

}

**public ListNode reverseList(ListNode A)** {

ListNode head = A;

if (head == null || head.next == null) {

return head;

}

ListNode curr = head;

ListNode prev = null;

while (curr != null) {

ListNode next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

head = prev;

return head;

}

1. *Middle element of linked list*

**Problem Description**

Given a linked list of integers, find and return the middle element of the linked list.

**NOTE:** If there are **N** nodes in the linked list and N is even then return the (N/2 + 1)th element.

**Problem Constraints**

1 <= length of the linked list <= 100000

1 <= Node value <= 109

**Input Format**

The only argument given head pointer of linked list.

**Output Format**

Return the middle element of the linked list.

**Example Input**

Input 1:

1 -> 2 -> 3 -> 4 -> 5

Input 2:

1 -> 5 -> 6 -> 2 -> 3 -> 4

**Example Output**

Output 1:

3

Output 2:

2

public int solve(ListNode A) {

if (A.next == null) {

return A.val;

}

ListNode i = A, j = A;

while (j != null && j.next != null) {

i = i.next;

j = j.next.next;

}

return i.val;

}

1. *Merge Two Sorted Lists*

**Problem Description**

Merge two sorted linked lists, **A** and **B,** and return it as a new list.

The new list should be made by splicing together the nodes of the first two lists and should also be sorted.

**Problem Constraints**

0 <= |A|, |B| <= 105

**Input Format**

The first argument of input contains a pointer to the head of linked list A.

The second argument of input contains a pointer to the head of linked list B.

**Output Format**

Return a pointer to the head of the merged linked list.

**Example Input**

Input 1:

A = 5 -> 8 -> 20

B = 4 -> 11 -> 15

Input 2:

A = 1 -> 2 -> 3

B = Null

**Example Output**

Output 1:

4 -> 5 -> 8 -> 11 -> 15 -> 20

Output 2:

1 -> 2 -> 3

public ListNode mergeTwoLists(ListNode A, ListNode B) { //TC:O(N+M) || SC:O(1)

ListNode ansHead = null;

if (A == null) {

return B;

} else if (B == null) {

return A;

} else {

ListNode curr = null;

while (A != null && B != null) {

if (A.val < B.val) {

ansHead = A;

A = A.next;

} else {

ansHead = B;

B = B.next;

}

curr = ansHead;

while (A != null && B != null) {

if (A.val < B.val) {

curr.next = A;

A = A.next;

} else {

curr.next = B;

B = B.next;

}

curr = curr.next;

}

if (A != null) {

curr.next = A;

}

if (B != null) {

curr.next = B;

}

}

}

return ansHead;

}

1. *Sort List*

**Problem Description**

Sort a linked list, **A** in **O(n log n)** time.

**Problem Constraints**

0 <= |A| = 105

**Input Format**

The first and the only arugment of input contains a pointer to the head of the linked list, A.

**Output Format**

Return a pointer to the head of the sorted linked list.

**Example Input**

Input 1:

A = [3, 4, 2, 8]

Input 2:

A = [1]

**Example Output**

Output 1:

[2, 3, 4, 8]

Output 2:

[1]

**public ListNode sortList(ListNode A)** { //TC:O(nlogn) || SC:O(logn)

if (A == null || A.next == null) {

return A;

}

ListNode middle = findMiddle(A);

ListNode nextToMiddle = middle.next;

middle.next = null;

ListNode left = sortList(A);

ListNode right = sortList(nextToMiddle);

ListNode sortedList = mergeTwoLists(left, right);

return sortedList;

}

**public ListNode findMiddle(ListNode A)** {

if (A.next == null) {

return A;

}

// start j from second element

// so that it will get handled in both cases, when number of nodes is either odd

// or even and it gets split exactly around mid

ListNode i = A, j = A.next;

while (j != null && j.next != null) {

i = i.next;

j = j.next.next;

}

return i;

}

**public ListNode mergeTwoLists(ListNode A, ListNode B)** {

ListNode ansHead = null;

if (A == null) {

return B;

} else if (B == null) {

return A;

} else {

ListNode curr = null;

while (A != null && B != null) {

if (A.val < B.val) {

ansHead = A;

A = A.next;

} else {

ansHead = B;

B = B.next;

}

curr = ansHead;

while (A != null && B != null) {

if (A.val < B.val) {

curr.next = A;

A = A.next;

} else {

curr.next = B;

B = B.next;

}

curr = curr.next;

}

if (A != null) {

curr.next = A;

}

if (B != null) {

curr.next = B;

}

}

}

return ansHead;

}

1. *Remove Loop from Linked List*

**Problem Description**

You are given a linked list that contains a loop.  
You need to find the node, which creates a loop and break it by making the node point to NULL.

**Problem Constraints**

1 <= number of nodes <= 1000

**Input Format**

The first of the input contains a LinkedList, where the first number is the number of nodes N, and the next N nodes are the node value of the linked list.  
The second line of the input contains an integer which denotes the position of node where cycle starts.

**Output Format**

return the head of the updated linked list.

**Example Input**

Input 1:

1 -> 2

^ |

| - -

Input 2:

3 -> 2 -> 4 -> 5 -> 6

^ |

| |

- - - - - -

**Example Output**

Output 1:

1 -> 2 -> NULL

Output 2:

3 -> 2 -> 4 -> 5 -> 6 -> NULL

public ListNode solve(ListNode A) { //TC:O(N) || SC:O(1)

if (A == null || A.next == null) {

return A;

}

ListNode slow = A;

ListNode fast = A;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) {

// Break the loop

ListNode start = A;

while (start.next != slow.next) {

start = start.next;

slow = slow.next;

}

slow.next = null;

return A;

}

}

return A;

}

1. *LRU Cache (HARD – Doubly LL)*

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

* get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
* set(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting the new item.

The LRUCache will be initialized with an integer corresponding to its capacity. Capacity indicates the maximum number of unique keys it can hold at a time.

**Definition of "least recently used"** : An access to an item is defined as a get or a set operation of the item. "Least recently used" item is the one with the oldest access time.

**NOTE:** If you are using any global variables, make sure to clear them in the constructor.

**Example :**

Input :

capacity = 2

set(1, 10)

set(5, 12)

get(5) returns 12

get(1) returns 10

get(10) returns -1

set(6, 14) this pushes out key = 5 as LRU is full.

get(5) returns -1

**Expected Output**

Enter your input as per the following guideline:

There are 1 lines in the input Line 1 ( Corresponds to arg 1 ) : The line starts with a pair of number numOperations, capacity. capacity is the number your constructor is initialized with. Then numOperation operations follow. Each operation is either : \* G : This corresponds to a function call get() \* S : This corresponds to a function call set(num1, num2) Note that the function calls are made in order.

public class LRUCache { //All operation (Insert, Delete, Get) TC : O(1) || SC:O(N)

class Node {

int key;

int val;

Node next;

Node prev;

Node (int k, int v) {s

this.key = k;

this.val = v;

prev = null;

next = null;

}

}

Node head = new Node(-1, -1);

Node tail = new Node(-1, -1);

Map<Integer, Node> hm;

static int capacity;

**public LRUCache(int capacity)** {

head.next = tail;

tail.prev = head;

this.capacity = capacity;

hm = new HashMap<>();

}

**public int get(int key)** {

if (hm.containsKey(key)) {

Node d = hm.get(key);

remove(d);

addToHead(d);

return d.val;

} else {

return -1;

}

}

**public void remove(Node x)** {

Node nx = x.next;

Node np = x.prev;

np.next = nx;

nx.prev = np;

}

**public void addToHead(Node x)** {

x.next = head.next;

x.prev = head;

head.next = x;

x.next.prev = x;

}

**public void set(int key, int value)** {

if (hm.containsKey(key)) {

Node x = hm.get(key);

x.val = value;

remove(x);

addToHead(x);

} else {

if (hm.size() == capacity) {

hm.remove(tail.prev.key);

remove(tail.prev);

}

Node newNode = new Node(key, value);

hm.put(key, newNode);

addToHead(newNode);

}

}

}

1. *Copy List (Hard)*

**Problem Description**

* You are given a **linked list A**
* Each node in the linked list contains two pointers: a **next** pointer and a **random** pointer
* The **next** pointer points to the **next node** in the **list**
* The **random** pointer can point to **any node** in the **list**, or it can be **NULL**
* Your task is to create a **deep copy** of the **linked list A**
* The copied list should be a completely separate linked list from the original list, but with the same **node values** and **random** pointer connections as the original list
* You should create a **new linked list B**, where each **node** in **B** has the same value as the corresponding **node** in**A**
* The **next** and **random** pointers of each **node** in **B** should point to the corresponding **nodes** in **B** (***rather than A***)

**Problem Constraints**

0 <= |A| <= 106

**Input Format**

The first argument of input contains a pointer to the head of linked list **A**.

**Output Format**

Return a pointer to the head of the required linked list.

**Example Input**

Given list

1 -> 2 -> 3

with random pointers going from

1 -> 3

2 -> 1

3 -> 1

**Example Output**

1 -> 2 -> 3

with random pointers going from

1 -> 3

2 -> 1

3 -> 1

**Example Explanation**

You should return a deep copy of the list. The returned answer should not contain the same node as the original list, but a copy of them. The pointers in the returned list should not link to any node in the original input list.

public class CopyList { //TC:O(N) || SC:O(1)

class RandomListNode {

int label;

RandomListNode next, random;

RandomListNode(int x) {

this.label = x;

}

}

public RandomListNode copyRandomList(RandomListNode head) {

*// Step 1: Duplicate each node and insert it after the original node.*

RandomListNode newHead = head;

while (newHead != null) {

RandomListNode newNode = new RandomListNode(newHead.label);

// *Connecting newNode to the next node of newHead*.

newNode.next = newHead.next;

newHead.next = newNode;

newHead = newNode.next;

}

*// Step 2: Update the random pointers for the duplicated nodes.*

RandomListNode t1 = head;

RandomListNode t2 = head.next;

while (t1 != null) {

if (t1.random != null) {

t2.random = t1.random.next;

}

t1 = t2.next;

if (t1 != null) {

t2 = t1.next;

}

}

*// Step 3: Separate the original and duplicated nodes.*

t1 = head;

t2 = head.next;

RandomListNode copiedListHead = t2;

while (t1 != null) {

t1.next = t2.next;

t1 = t1.next;

if (t1 != null) {

t2.next = t1.next;

}

t2 = t2.next;

}

return copiedListHead;

}

}