# Array Product

Problem Statement:

Given an array of numbers nums of size n, find an array of numbers products of size n, such that products[i] is the product of all numbers nums[j], where j != i.

Input/Output Format For The Function:

Input Format:

There is only one argument: nums, denoting input array.

Output Format:

Return an array of numbers products, denoting the required product array where products[i] is the (product of all numbers nums[j]) % (10^9 + 7), where j != i.

Input/Output Format For The Custom Input:

Input Format:

The first line of the input should contain a single integer n denoting the size of input array.

In the next n lines, each line should contain a number Ai, denoting ith number of the input array A, (0<=i<n).

If n = 5 and nums = [1, 2, 3, 4, 5], then input should be:

5

1

2

3

4

5

Output Format:

There will be n lines, each line containing a number Pi, denoting ith number of the resultant product array P.

For input n = 5 and nums = [1, 2, 3, 4, 5], output will be:

120

60

40

30

24

Constraints:

* You can't use division anywhere in solution.
* 2 <= n <= 100000
* -10^9 <= nums[i] <= 10^9, i = 0, 1, 2, … , n-1
* products[i] >=0, i = 0, 1, 2, ... , n-1
* You are allowed to use only constant extra space and resultant product array will not be considered as extra space.

Notes:

* Usage of resultant products array will not be considered as extra space used.
* Without using division is the key constraint to remember.

Sample Test Cases:

Sample Input 1:

5

1

2

3

4

5

Sample Output 1:

120

60

40

30

24

Explanation 1:

Resultant Product array products = [products[0], products[1], products[2], products[3], products[4]]

= [(nums[1]\*nums[2]\*nums[3]\*nums[4]), (nums[0]\*nums[2]\*nums[3]\*nums[4]), (nums[0]\*nums[1]\*nums[3]\*nums[4]), (nums[0]\*nums[1]\*nums[2]\*nums[4]), (nums[0]\*nums[1]\*nums[2]\*nums[3])]

= [(2\*3\*4\*5), (1\*3\*4\*5), (1\*2\*4\*5), (1\*2\*3\*5), (1\*2\*3\*4)]

= [120, 60, 40, 30, 24]

Sample Input 2:

3

4

9

10

Sample Output 2:

90

40

36

Explanation 2:

Resultant Product array products = [products[0], products[1], products[2]]

= [(nums[1]\*nums[2]), (nums[0]\*nums[2]), (nums[0]\*nums[1])]

= [(9\*10), (4\*10), (4\*9)]

= [90, 40, 36]

Suggestions:

﻿﻿﻿﻿﻿﻿﻿

Suggested time in interview: 20 minutes.

The “Suggested Time” is the time expected to complete this question during a real-life interview, not now in homework i.e. For the first attempt of a given homework problem, the focus should be to understand what the problem is asking, what approach you are using, coding it, as well as identifying any gaps that you can discuss during a TC session. Take your time, but limit yourself to 2 one hour sessions for most problems.

Are you getting wrong answer for some of the test cases, but still think your logic is correct?

Check for the overflow.

If a = 10^9, b = 10^9 and we do, int c = (a \* b) % (10^9 + 7), then it will overflow.

Instead, we should use something like int c = (a \* (long long int) 1 \* b) % (10^9 + 7), to avoid overflow. By multiplying with (long long int) 1, we make sure that the calculation is done in long long int, instead of int.

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution.java

Description:

A naive approach would be, to find ith element of output array (i.e. products[i]), iterate over an entire input array nums to get the product of all elements nums[j], such that j != i.

Time Complexity:

O(n\*n) where n is length of input array nums.

As we are iterating over complete array to find products[i] and as i can be 0<=i<=n-1. Each calculation of element of products array will take O(n) so total complexity will be O(n\*n).

Auxiliary Space Used:

O(1).

As we are not storing anything extra and excluding space used to store output array products.

Space Complexity:

O(n) where n is length of input array nums.

As storing input array nums of length n will take O(n) and auxiliary space used is O(1). So, O(n) + O(1) → O(n).

2) optimal\_solution.java

Description:

An optimal approach would be as follows:

Notice that for products[i], product of all input array elements other than ith element is nothing but (product of all elements nums[j], 0 <= j <= (i-1)) \* (product of all elements nums[j], (i+1) <= j <= (n-1)) = (nums[0]\*nums[1]\*...\*nums[i-1]) \* (nums[i+1]\*nums[i+2]\*...\*nums[n-1]).

So, iterate input array nums twice to fill output array products, once for updating products[i] with (nums[0] \* nums[1] \*...\*nums[i-1]), and next one for updating products[i] with (nums[i+1] \* nums[i+2] \* … \* nums[n-1]).

Please see the commented code for detailed implementation of optimal approach.

Time Complexity:

O(n) where n is length of input array nums.

As we are iterating over input array nums two times it will take O(n).

Auxiliary Space Used:

O(1).

As we are not storing anything extra and excluding space used to store output array products.

Space Complexity:

O(n) where n is length of input array nums.

As storing input array nums of length n will take O(n) and auxiliary space used is O(1). So, O(n) + O(1) → O(n).

## Solutions Provided By Us

/\*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PROBLEM DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Given an array of numbers, nums, return an array of numbers products, where products[i]is the product of all nums[j],

\* j != i.

\*

\* Input : [1, 2, 3, 4, 5]

\* Output: [(2\*3\*4\*5), (1\*3\*4\*5), (1\*2\*4\*5), (1\*2\*3\*5), (1\*2\*3\*4)]

\* = [120, 60, 40, 30, 24]

\* You must do this in O(n) time, and constant space, without using division. Usage of products array is not considered

\* extra space.

\*

\* Without using division is the key constraint to remember.

\*/

import java.io.\*;

import java.util.\*;

class Result {

// ============================ Start ============================

static int mod = (int)Math.pow(10, 9) + 7;

static int[] getProductArray(int[] nums) {

// Size of output array is same as that of input array

int[] products = new int[nums.length];

// For finding value of products[i], product of all nums elements

// other than ith element is nothing but

// (product of all nums[j], 0<=j<=(i-1)) \* (product of all nums[j], (i+1)<=j<=(nums.length-1))

// i.e. (nums[0]\*nums[1]\*...\*nums[i-1]) \* (nums[i+1]\*nums[i+2]\*...\*nums[nums.length-1])

int leftProduct = 1;

// Filling products, such that products[i] contains

// product of all elements nums[j], 0<=j<=(i-1)

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

// Here, leftProduct contains product of all elements

// nums[j], 0<=j<=(currentIndex-1)

products[currentIndex] = leftProduct;

// After this updation of leftProduct, leftProduct contains product of all

// elements nums[j], 0<=j<=currentIndex

nums[currentIndex] = nums[currentIndex]>0?nums[currentIndex]:(mod+nums[currentIndex])%mod;

leftProduct = (int)((leftProduct \* 1l \* nums[currentIndex])%mod);

}

int rightProduct = 1;

// Updating products, such that products[i] contains new value

// ((products[i]) \* (product of all elements nums[j], 0<=j<=(i-1)))

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

// Here, rightProduct contains product of all elements

// nums[j], (currentIndex+1)<=j<=(nums.length-1)

products[currentIndex] = (int)((products[currentIndex] \* 1l \* rightProduct)%mod);

// after this updation of rightProduct, rightProduct contains product of all

// elements nums[j], currentIndex<=j<=(nums.length-1)

rightProduct = (int)((rightProduct \* 1l \* nums[currentIndex])%mod);

}

return products;

}

// ============================= End ==============================

}

class Solution {

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int n = Integer.parseInt(bufferedReader.readLine().trim());

int nums[] = new int[n];

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

nums[i] = Integer.parseInt(bufferedReader.readLine().trim());

}

int products[] = Result.getProductArray(nums);

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

bufferedWriter.write(products[i]+"\n");

}

bufferedReader.close();

bufferedWriter.close();

}

}

/\*\*

\* Time complexity: O(n)

\* Auxiliary space used: O(1)

\* Space complexity: O(n)

\*/

# Merge Overlapping Intervals

Problem Statement:

Given an array of time intervals(in any order) inputArray, of size n, merge all overlapping intervals into one and return the resulting array outputArray, such that no two intervals in outputArray are overlapping. In other words, result array should contain only mutually exclusive intervals. Hence, in outputArray, no pair of intervals i and j exists, such that

outputArray[i][0] <= outputArray[j][0] <= outputArray[i][1].

(In this problem, you should consider all the intervals as closed intervals. i.e. endpoints of intervals are inclusive.)

Input/Output Format For The Function:

Input Format:

There is only one argument: inputArray, denoting input array of time intervals, where inputArray is 2D array of n\*2 size, denoting inputArray[i][0] as start point of ith interval, and inputArray[i][1] as end point of ith interval.

Output Format:

Return an array of time intervals outputArray, denoting the required array of merged time intervals, where outputArray is 2D array of len\*2 size, denoting outputArray[i][0] as start point of ith interval, and outputArray[i][1] as end point of ith interval.

(Order of intervals in outputArray doesn't matter.)

Input/Output Format For The Custom Input:

Input Format:

First line should contain a number n, denoting number of intervals in inputArray. Next line should contain 2, unconditionally, as inputArray is 2D array of n\*2. In next n lines, ith line should contain two space separated numbers starti and endi, denoting start point and end point of ith interval respectively.

If n = 4, inputArray = [[1, 3], [5, 7], [2, 4], [6, 8]], then input should be:

4

2

1 3

5 7

2 4

6 8

Output Format:

Let say len\*2 is the size of resultant 2D array outputArray. Then, there will be len lines, where ith line contains two space separated integers starti and endi, denoting start point and end point of ith interval in outputArray respectively.

For input n = 4, inputArray = [[1, 3], [5, 7], [2, 4], [6, 8]], output will be:

1 4

5 8

Constraints:

* 1 <= n <= 10^5
* -10^9 <= inputArray[i][0] <= inputArray[i][1] <= 10^9,   i=0, 1, ..., (n-1)

Sample Test Cases:

Sample Input 1:

4

2

1 3

5 7

2 4

6 8

Sample Output 1:

1 4

5 8

Explanation 1:

The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1,4}.

Similarly {5,7} and {6,8} should be merged and become {5,8}.

Sample Input 2:

7

2

100 154

13 47

1 5

2 9

7 11

51 51

47 50

Sample Output 2:

1 11

13 50

51 51

100 154

Explanation 2:

The intervals {1,5} and {2,9} overlap with each other, so they should be merged and become {1,9}.

Also, {1,9} and {7,11} overlap with each other, so they should be merged and become {1,11}

Similarly, The intervals {13,47} and {47,50} should be merged and become {13,50}.

Intervals {51,51} and {100,154} are kept as it is as they are not overlapping with any other intervals.

Suggestions:

Suggested time in interview: 20 minutes.

The “Suggested Time” is the time expected to complete this question during a real-life interview, not now in homework i.e. For the first attempt of a given homework problem, the focus should be to understand what the problem is asking, what approach you are using, coding it, as well as identifying any gaps that you can discuss during a TA session. Take your time, but limit yourself to 2 one hour sessions for most problems.

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution.java

Description:

A naive approach would be that iterating over inputArray,

For 0<=i<=n-1, Check if inputArray[i] is a removed interval.

1. If it’s a removed interval continue.
2. If it's not a removed interval, compare inputArray[i] with all other intervals for overlapping. Let say it overlaps with interval inputArray[k], then remove inputArray[k] from array and merge it into the inputArray[i].

For removing an interval from array, one way is to make the interval invalid (i.e. start>end), so that later we can

check if it is removed or not. See implementation for better understanding.

Time Complexity:

O(n\*n) where n is length of inputArray.

As we have to iterate entire input interval array for each interval, time complexity will be O(n\*n).

Auxiliary Space Used:

O(1).

Here, all updation can be done in inputArray only. So, no extra space is used.

Space Complexity:

O(n) where n is length of inputArray.

For inputArray, it takes O(n) and auxiliary space used is O(1). So, O(n) + O(1) → O(n).

2) other\_solution.java

Description:

An efficient approach would be as follows:

Sort the interval array in increasing order of start point. Once we have sorted intervals, we can combine all intervals in a linear traversal.

Following is the detailed step by step algorithm.

1. Sort the intervals based on increasing order of starting time.
2. Push the first interval on to a stack.
3. For each interval do the following
4. If the current interval does not overlap with the stack top, push it.
5. If the current interval overlaps with stack top and ending time of current interval is more than that of stack top, update stack top with the ending  time of current interval.
6. At the end stack contains the merged intervals.

Time Complexity:

O(n\*log(n)) where n is length of inputArray.

As we have to sort the interval array, followed by linear traversal, time complexity will be

O(n\*log(n)) + O(n) → O(n\*log(n)).

Auxiliary Space Used:

O(n) where n is length of inputArray.

Here we have used Stack. So, extra space is used other than that of inputArray. So, auxiliary space used is O(n). (Here we are ignoring auxiliary space used by inbuilt sort function for sorting n elements of inputArray because it is language specific like in java, it will take O(1) but in python it will take O(n)).

Space Complexity:

O(n) where n is length of inputArray.

For inputArray, it takes O(n) and auxiliary space used is O(n). So, O(n) + O(n) → O(n).

3) optimal\_solution.java

Description:

Auxiliary space used in above approach is O(n). It can be reduced.

The idea remains same as discussed in previous approach. Sort the interval array in increasing order of starting point.

Once you have sorted intervals, you can combine all intervals in a linear traversal.

Following is the detailed step by step algorithm:

Let last be the last interval of non overlapping intervals. last=0.

Iterating over inputArray, starting from second interval (1<=i<=n-1)

1. Check if inputArray[i] is overlapping with inputArray[last]
2. If overlapping, merge inputArray[i] and inputArray[last], For merging them, it is sufficient to update only endpoint of inputArray[last] as it is guaranteed that starting point of inputArray[last] <= starting point of inputArray[i][0] as array is sorted by starting point.
3. If non overlapping, we increment last and moving on, inputArray[i] is the new interval under test of overlapping with following intervals.
4. repeat step 1 for i=i+1.

Time Complexity:

O(n\*log(n)) where n is length of inputArray.

As we have to sort the interval array, followed by linear traversal, time complexity will be

O(n\*log(n)) + O(n) → O(n\*log(n)).

Auxiliary Space Used:

O(1).

Here, all updation can be done in inputArray only. So, no extra space is used. (Here we are ignoring auxiliary space used by inbuilt sort function for sorting n elements of inputArray because it is language specific like in java, it will take O(1) but in python it will take O(n)).

Space Complexity:

O(n) where n is length of inputArray.

For inputArray, it takes O(n) and auxiliary space used is O(1). So, O(n) + O(1) → O(n).

## Solutions Provided By Us

/\*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PROBLEM DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which

\* should have only mutually exclusive intervals.

\*

\* e.g. for this input: {{1,3}, {2,4}, {5,7}, {6,8} }.

\* The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1, 4}.

\* Similarly {5, 7} and {6, 8} should be merged and become {5, 8}.

\*

\* Write a function which produces the set of merged intervals for the given set of intervals.

\*/

import java.io.\*;

import java.util.\*;

class Result {

// -------------------- START ----------------------

static int[][] getMergedIntervals(int[][] inputArray) {

// Sorting the input interval array by their starting points in increasing order

Arrays.sort(inputArray, (int object1[], int object2[]) -> {

if (object1[0] != object2[0]){

return object1[0] - object2[0];

}

return object1[1] - object2[1]; }

);

int last = 0;

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

// Checking if inputArray[last] and inputArray[i] are overlapping or not

if (inputArray[last][1] >= inputArray[i][0]) {

// If overlapping, then merge inputArray[i] into inputArray[last]

// For merging them, it is sufficient to update only endpoint of inputArray[last] as

// it is guaranteed that inputArray[last][0]<=inputArray[i][0] , last<i

inputArray[last][1] = Math.max(inputArray[last][1], inputArray[i][1]);

} else {

// inputArray[last] and inputArray[i] are found non-overlapping.

// Moving on, inputArray[i] is the new interval under test of overlapping with following intervals.

last++;

inputArray[last] = inputArray[i];

}

}

// From index 0 to last of inputArray will contain all non overlapping intervals

return Arrays.copyOfRange(inputArray, 0, last+1);

}

// -------------------- END ----------------------

}

class Solution{

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int n = Integer.parseInt(bufferedReader.readLine().trim());

int m = Integer.parseInt(bufferedReader.readLine().trim());

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

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

String[] str = bufferedReader.readLine().split(" ");

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

inputArray[i][j] = Integer.parseInt(str[j].trim());

}

}

int[][] res = Result.getMergedIntervals(inputArray);

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

bufferedWriter.write(res[i][0]+" "+res[i][1]+"\n");

}

bufferedReader.close();

bufferedWriter.close();

}

}

/\*\*

\* Time complexity: O(N\*log(N))

\* Auxiliary Space complexity: O(1)

\*/

# 2D Array Search

Problem Statement:

You are given a sorted 2D array arr of size r \* c, where all the numbers (integers) in the array are in non decreasing order from left to right and top to bottom. ( i.e. arr[i][j] <= arr[i+1][j] and arr[i][j] <= arr[i][j+1] for all i = 0,1,...,(r - 2) and j = 0,1,...,(c - 2) )

Check if a given number x exists in arr or not.

Given an arr, you have to answer q such queries.

Input/Output Format For The Function:

Input Format:

There are two arguments, arr and x, denoting input 2D array and a number to be searched as mentioned in problem statement respectively

Output Format:

Return string "present" if x is present in arr, string "not present" otherwise.

Input/Output Format For The Custom Input:

Input Format:

The first line of the input should contain a single integer r, denoting the no. of rows of input array arr. Second line should contain a single integer c, denoting the no. of columns of arr.

In the next r lines, each line should contain c space separated numbers. jth number in ith line of these r lines is arr[i][j], denoting the number at ith row of the jth column of arr.

Next line should contain q, denoting no of queries to be answered. In next q lines, each line should contain xi, denoting x for ith query

If r = 3, c = 4, arr = [[1, 2, 3, 12], [4, 5, 6, 45], [7, 8, 9, 78]], q = 3, x for 1st query = 6, x for 2nd query = 7 and x for 3rd query = 23, then input should be:

3

4

1 2 3 12

4 5 6 45

7 8 9 78

3

6

7

23

Output Format:

There will be q lines, ith line of which contains the result for ith query as "present" or "not present"

For input r = 3, c = 4, arr = [[1, 2, 3, 12], [4, 5, 6, 45], [7, 8, 9, 78]], q = 3, x for 1st query = 6, x for 2nd query = 7 and x for 3rd query = 23, output will be:

present

present

not present

Constraints:

* 1 <= r <= 10^3
* 1 <= c <= 10^3
* 1 <= q <= 10^4
* -10^9 <= arr[i][j] <= 10^9, (i = 0,1,...,(r - 1) and j = 0,1,...,(c - 1))
* -10^9 <= x <= 10^9

Sample Test Cases:

Sample Input 1:

arr = [[1, 2, 3, 12], [4, 5, 6, 45], [7, 8, 9, 78]]

queries = [6, 7, 23]

Sample Output 1:

result = [“present”, “present”, “not present”]

Explanation 1:

Given number x=6 is present at arr[1][2] and x=7 is present at arr[2][0]. Hence, "present" returned for them, while

x=23 is not present in arr, hence "not present" returned

Sample Input 2:

arr = [[3, 4], [5, 10]]

queries = [12, 32]

Sample Output 2:

result = [“not present”, “not present”]

Explanation 2:

Given number x=12 and x=32 are not present in arr. Hence, "not present" returned for both of the queries

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution.java

Description:

A naive approach would be, to iterate over entire input array arr to check if x is present or not.

Time Complexity:

O(r\*c\*q) where r denotes number of rows of arr, c denotes number of columns of arr and q denotes number of queries.

As we are iterating over entire array for each query, time complexity will be O(r\*c) (for each query) and as there are q queries so total time complexity will be O(r\*c\*q).

Auxiliary Space Used:

O(1).

As we are not storing anything extra.

Space Complexity:

O(r\*c) where r denotes number of rows of arr and c denotes number of columns of arr.

To store input, it would take O(r\*c), auxiliary space used is O(1).

So, total space complexity will be O(r\*c).

2) optimal\_solution.java

Description:

An optimal approach would be as follows:

Steps:

1) Start with top right element arr[0][c-1]

2) Loop: compare this element arr[i][j] with x

   -> If arr[i][j] == x, then return "present"

   -> If arr[i][j] < x then move to next row (i.e. arr[i+1][j])

   -> If arr[i][j] > x then move to column to its left (i.e. arr[i][j-1])

3) repeat the steps in #2 till you find element and return "present" OR if out of bound of matrix then break and return "not present"

Let say x is not present in first (i-1) rows.

Let's say in ith row, arr[i][j] is largest number smaller than or equal to x.

-> If it is equal to x, then problem solved, directly return “present”.

-> If arr[i][j] < x, it can be implied that x cannot be present at arr[l][m], i < l and j < m as array is row wise and column wise sorted (ascending). So, moving on to next row, (i+1)th row, we can start checking from jth column (i.e. arr[i+1][j]).

-> If arr[i][j] > x, means element x can be present in left side of column jth as row and column are sorted in ascending order. So, we start checking it with arr[i][j-1].

Time Complexity:

O((r+c)\*q) where r denotes number of rows of arr, c denotes number of columns of arr and q denotes number of queries.

As for each query maximum iteration over array can be of O(r+c) and as there can be q queries so, total complexity will be O((r+c)\*q).

Auxiliary Space Used:

O(1).

As we are not storing anything extra.

Space Complexity:

O(r\*c) where r denotes number of rows of arr and c denotes number of columns of arr.

To store input, it would take O(r\*c), auxiliary space used is O(1).

So, total space complexity will be O(r\*c).

## Solutions Provided By Us

/\*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PROBLEM DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* You are given a sorted 2D array arr of size r\*c, where all the numbers (integers) in the array are in increasing

\* order from left to right and top to bottom i.e. arr[i][j]<=arr[i+1][j] and arr[i][j]<=arr[i][j+1].

\* Check if a given number x exists in it or not.

\*/

import java.io.\*;

import java.util.\*;

class Result {

// -------------------- START ----------------------

static String isPresent(int arr[][], int x) {

int r = arr.length;

int c = arr[0].length;

int rowIndex = 0;

int colIndex = c - 1;

// Starting from 0th row, find first element from right in current row, let say a[l][m], such

// that a[l][m] <= x.

while(rowIndex <= (r-1) && colIndex >= 0){

// arr[rowIndex][colIndex] is the first element from right in current row rowIndex.

if (arr[rowIndex][colIndex] == x){

return "present";

}

// As arr is sorted row wise and column wise in increasing order,

// we can say that x can't be present at arr[l][m], rowIndex<l and colIndex<m

// Also, in current row rowIndex, x can't be present as arr[rowIndex][colIndex] < x and

// all elements to its left are even smaller than arr[rowIndex][colIndex] and

// we have already checked all elements to its right. So moving on to next row.

// Notice that you can start to check at current column j (stored in colIndex) in next row as x can't

// be present at arr[l][m], l>rowIndex and m>colIndex

if (arr[rowIndex][colIndex] < x){

rowIndex++;

}

// As arr is sorted row wise and column wise in increasing order,

// we can say that if x < arr[rowIndex][colIndex] means x can be present

// on left side of colIndex in same row rowIndex.

else if(arr[rowIndex][colIndex] > x){

colIndex--;

}

}

return "not present";

}

// -------------------- END ----------------------

}

class Solution{

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int arrRows = Integer.parseInt(bufferedReader.readLine().trim());

int arrColumns = Integer.parseInt(bufferedReader.readLine().trim());

int[][] arr = new int[arrRows][arrColumns];

for (int arrRowItr = 0; arrRowItr < arrRows; arrRowItr++) {

String[] arrRowItems = bufferedReader.readLine().split(" ");

for (int arrColumnItr = 0; arrColumnItr < arrColumns; arrColumnItr++) {

int arrItem = Integer.parseInt(arrRowItems[arrColumnItr].trim());

arr[arrRowItr][arrColumnItr] = arrItem;

}

}

int q = Integer.parseInt(bufferedReader.readLine().trim());

int x;

String res;

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

x = Integer.parseInt(bufferedReader.readLine().trim());

res = Result.isPresent(arr, x);

bufferedWriter.write(res+"\n");

}

bufferedReader.close();

bufferedWriter.close();

}

}

/\*\*

\* Time complexity: O((r+c)\*q)

\* Space complexity: O(1)

\*/

Mine

#

# Complete the isPresent function below.

#

def isPresent(a, x):

#

# Write your code here.

#

r = len(arr)

c = len(arr[0])

i = 0

j = c - 1

while (i < r and j >= 0):

cur = a[i][j]

if (x == cur):

return "present"

elif (x < cur):

j = j - 1

elif (x > cur):

i = i + 1

else:

break

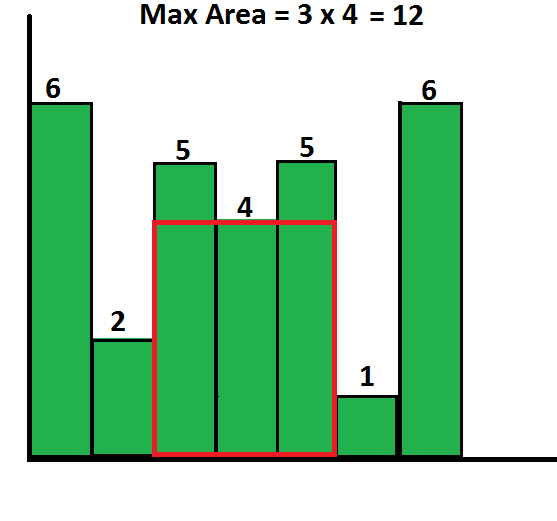
return "not present"

# Area under histogram

Problem Statement:

You will be given an array arr of height of bars, of size n. You have to answer q queries, where in each query, you will be given left index l and right index r. For each query, return largest rectangular area possible in a histogram formed using (right-left+1) bars with array of heights as [arr[left], arr[left+1], arr[left+2], ..., arr[right]]. Largest rectangle can be made of a number of contiguous bars. For simplicity, you can assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights [6, 2, 5, 4, 5, 1, 6]. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red).



(source: https://goo.gl/fTcCTK)

Input/Output Format For The Function:

Input Format:

There are three arguments: arr, denoting input array of height of bars, l denoting left and r denoting right as explained in problem statement.

Output Format:

Return a number maxArea, denoting maximum rectangular area possible in a histogram formed as explained in problem statement.

Input/Output Format For The Custom Input:

Input Format:

The first line of the input should contain a single integer n, denoting the size of input array arr. In the next n lines, ith line should contain a number arr[i], denoting ith number of the input array arr, i=(1,2,...,n).

Next line should contain q, denoting no. of queries that need to be answered. In next 2\*q lines, (2\*i-1)th line should contain left value for ith query and (2\*i)th line should contain right value for ith query, i=(1,2,...,q), i.e. 1st and 2nd line should contain left and right values for 1st query, 3rd and 4th line should contain left and right values for 2nd query, and so on...

If n = 5, arr = [2, 4, 6, 5, 8], q = 2, for 1st query: l = 0 and r = 4 and for 2nd query: l = 3 and r = 3, then input should be:

5

2

4

6

5

8

2

0

4

3

3

Output Format:

There will be q lines, where ith line contains a number maxArea[i], denoting result of ith query.

For input n = 5, arr = [2, 4, 6, 5, 8], q = 2, for 1st query: l = 0 and r = 4 and for 2nd query: l = 3 and r = 3, output will be:

16

5

Constraints:

* 1 <= n <= 2\*10^5
* 1 <= q <= 10
* 1 <= arr[i] <= 10^9, i=(0,1,2,3,...,n-1)
* 0 <= l <= r < n for each query.

Sample Test Cases:

Sample Input 1:

arr = [6, 2, 5, 4, 5, 1, 6]

q = 1

For 1st query: l = 0 and r = 6.

Sample Output 1:

12

Explanation 1:

1st query: A rectangle of area 12 can be formed using 2nd to 4th bar (0-based indexing) and has maximum area possible in histogram out of all possible rectangles that can be formed using contiguous bar with given array of heights [arr[0],…,arr[6]] = [6, 2, 5, 4, 5, 1, 6] as l=0 and r=6.

Sample Input 2:

arr = [2, 4, 6, 5, 8]

q = 2

For 1st query: l = 0 and r = 4.

For 2nd query: l = 3 and r = 3.

Sample Output 2:

16

5

Explanation 2:

1st query: A rectangle of area 16 can be formed using 1st to 4th bar (0-based indexing) and has maximum area possible in histogram out of all possible rectangles that can be formed using contiguous bar with given array of heights [arr[0], …, arr[4]] = [2, 4, 6, 5, 8] as l=0 and r=4.

2nd query: A rectangle of area 5 can be formed using 3rd to 3rd bar (0-based indexing) and has maximum area possible in histogram out of all possible rectangles that can be formed using contiguous bar with given array of heights [arr[3]] = [5] as l=3 and r=3.

Suggestions:

Suggested time in interview: 30 minutes.

The “Suggested Time” is the time expected to complete this question during a real-life interview, not now in homework i.e. For the first attempt of a given homework problem, the focus should be to understand what the problem is asking, what approach you are using, coding it, as well as identifying any gaps that you can discuss during a TA session.

Take your time, but limit yourself to 2 one hour sessions for most problems.

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution.java

Description:

A naive approach would be to check area of all possible rectangles that can be made using bars in given histogram and find the max area rectangle.

To implement this approach, iterate over all bars j for each bar i, j>=i, find the smallest height bar 'hsmall' from (i, i+1, i+2, ..., j)th bars. Then area of largest rectangle made using (i, i+1, i+2 , ..., j)th bars would be currentArea = (hsmall \* (j - i + 1)). Compare with max area 'maxArea' found till now and replace it if maxArea < currentArea.

Time Complexity:

O(n\*n) where n is length of input array arr.

As we are iterating over all possible subarrays, it will take O(n\*n).

This time complexity is for function findMaxPossibleArea.

Auxiliary Space Used:

O(1).

As we are not storing anything extra.

This auxiliary is for function findMaxPossibleArea.

Space Complexity:

O(n) where n is length of input array arr.

To store array it will take O(n) and as auxiliary space used is O(1).

So, O(n) + O(1) → O(n).

This space complexity is for function findMaxPossibleArea.

2) optimal\_solution1.java

Description:

The optimal would be as follows:

As we are calling this function with given l and r.

For each element arr[i+l], find the first element smaller than it on its left (only consider upto index l) and that on its right (only consider upto index r) and build arrays leftSmaller and rightSmaller.

Thus, leftSmaller[i] = j, implies that j is greatest index smaller than i+l such that arr[i+l]>arr[j] and rightSmaller[i] = j, implies that j is smallest index greater than i+l such that arr[i+l]>arr[j].

Notice that all elements in subarray arr[leftSmaller[i]+1], arr[leftSmaller[i]+2],..., arr[rightSmaller[i]-1] are greater than or equal to arr[i+l]. So, the max area a rectangle can have such that it must contain ith bar is (arr[i+l]\*((rightSmaller[i]-1) - (leftSmaller[i]+1) + 1).

Time Complexity:

O(n) where n is length of input array arr.

Arrays leftSmaller and rightSmaller can be generated in a single traversal of arr input array, one traversal of arr array for each using stack.

So, to create leftSmaller and rightSmaller array it will take O(n) and to find result it will take O(n).

So, O(n) + O(n) → O(n).

This time complexity is for function findMaxPossibleArea.

Auxiliary Space Used:

O(n) where n is length of input array arr.

To store leftSmaller and rightSmaller array in generating them stack used which will take O(n) space.

This auxiliary is for function findMaxPossibleArea.

Space Complexity:

O(n) where n is length of input array arr.

As to store input array it will take O(n) and auxiliary space used is O(n).

Hence, O(n) + O(n) → O(n).

This space complexity is for function findMaxPossibleArea.

3) optimal\_solution2.java

Description:

This would be more optimal approach in comparison to approach of optimal\_solution2.java.

In optimal\_solution1, we traverse array 3 times to find answer but here we will traverse array once to find our answer.

For every bar ‘x’, we calculate the area with ‘x’ as the smallest bar in the rectangle. If we calculate such area for every bar ‘x’ and find the maximum of all areas, our task is done. How to calculate area with ‘x’ as smallest bar? We need to know index of the first smaller (smaller than ‘x’) bar on left of ‘x’ and index of first smaller bar on right of ‘x’. Let us call these indexes as ‘left index’ and ‘right index’ respectively.

We traverse all bars from left to right, maintain a stack of bars.

Every bar is pushed to stack once. A bar is popped from stack

when a bar of smaller height is seen. When a bar is popped, we calculate the area with the popped bar as smallest bar. How do we get left and right indexes of the popped bar – the current index tells us the ‘right index’ and index of previous item in stack is the ‘left index’. Following is the complete algorithm.

1) Create an empty stack.

2) Start from first bar, and do following for every bar ‘arr[i]’ where ‘i’ varies from l to r (As we are calling this function for each query with given l and r).

……a) If stack is empty or arr[i] is higher than the bar at top of stack, then push ‘i’ to stack.

……b) If this bar is smaller than the top of stack, then keep removing the top of stack while top of the stack is greater. Let the removed bar be arr[tp]. Calculate area of rectangle with arr[tp] as smallest bar. For arr[tp], the ‘left index’ is previous (previous to tp) item in stack and ‘right index’ is ‘i’ (current index).

3) If the stack is not empty, then one by one remove all bars from stack and do step 2.b for every removed bar.

For better understanding, please look solution once.

Time Complexity:

O(n) where n is length of input array arr.

As we are traversing through array to calculate our answer and size of array is r-l+1 in worst case it can be n. So, time complexity will be O(n).

This time complexity is for function findMaxPossibleArea.

Auxiliary Space Used:

O(n) where n is length of input array arr.

To maintain stack of size r-l+1 in worst case it can be n which is length of input array arr. So, auxiliary space used will be O(n).

This auxiliary is for function findMaxPossibleArea.

Space Complexity:

O(n) where n is length of input array arr.

As to store input array it will take O(n) and auxiliary space used is O(n).

Hence, O(n) + O(n) → O(n).

This space complexity is for function findMaxPossibleArea.

## Solutions Provided By Us

/\*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PROBLEM DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Find the largest rectangular area possible in a given histogram, where

\* the largest rectangle can be made of a number of contiguous bars.

\* For simplicity, assume that all bars have same width and the width is 1 unit.

\* You will be given an array arr of height of bars of size n.

\*/

import java.io.\*;

import java.util.\*;

class Result {

// ============================ Start ============================

static long findMaxPossibleArea(long[] arr, int l, int r) {

int n = r - l + 1;

long maxArea = 0;

// rightSmaller[i] = j, implies that j is smallest index greater than i

// such that arr[l+i]>arr[r+j] where 0<=i<(r-l+1) and 0<=j<(r-l+1)

// To fill rightSmaller array

int[] rightSmaller = new int[n];

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

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

if (stack.isEmpty()) {

stack.push(i);

} else {

while (!stack.isEmpty() && arr[stack.peek()] > arr[i]) {

int popped = stack.pop();

rightSmaller[popped - l] = i - l;

}

stack.push(i);

}

}

while (!stack.isEmpty()) {

int popped = stack.pop();

rightSmaller[popped - l] = n;

}

// leftSmaller[i] = j, implies that j is smallest index greater than i

// such that arr[l+i]>arr[r+j] where 0<=i<(r-l+1) and 0<=j<(r-l+1)

// To fill leftSmaller array

int[] leftSmaller = new int[n];

for (int i = r; i >= l; i--) {

if (stack.isEmpty()) {

stack.push(i);

} else {

while (!stack.isEmpty() && arr[stack.peek()] > arr[i]) {

int popped = stack.pop();

leftSmaller[popped - l] = i - l;

}

stack.push(i);

}

}

while (!stack.isEmpty()) {

int popped = stack.pop();

leftSmaller[popped - l] = -1;

}

// Notice that all elements in subarray arr[leftSmaller[i]+1], arr[leftSmaller[i]+2],...,

// arr[rightSmaller[i]-1] are greater than or equal to arr[i]

// So, the max area a rectangle can have such that it must contain ith bar is

// (arr[i]\*((rightSmaller[i]-1) - (leftSmaller[i]+1) + 1)

// to calculate histogram area. here n = r - l + 1

long currentMaxArea;

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

currentMaxArea = arr[i + l] \* (rightSmaller[i] - leftSmaller[i] - 1);

maxArea = Math.max(currentMaxArea, maxArea);

}

return maxArea;

}

// ============================= End ==============================

}

class Solution {

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int heightsCount = Integer.parseInt(bufferedReader.readLine().trim());

long[] heights = new long[heightsCount];

for (int heightsItr = 0; heightsItr < heightsCount; heightsItr++) {

heights[heightsItr] = Long.parseLong(bufferedReader.readLine().trim());

}

int q = Integer.parseInt(bufferedReader.readLine().trim());

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

int l = Integer.parseInt(bufferedReader.readLine().trim());

int r = Integer.parseInt(bufferedReader.readLine().trim());

long res = Result.findMaxPossibleArea(heights, l, r);

bufferedWriter.write(res+"\n");

}

bufferedWriter.close();

}

}

/\*\*

\* Time complexity: O(n)

\* Space complexity: O(n)

\*/

/\*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PROBLEM DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Find the largest rectangular area possible in a given histogram, where

\* the largest rectangle can be made of a number of contiguous bars.

\* For simplicity, assume that all bars have same width and the width is 1 unit.

\* You will be given an array arr of height of bars of size n.

\*/

import java.io.\*;

import java.util.\*;

class Result {

// ============================ Start ============================

static long findMaxPossibleArea(long[] arr, int l, int r) {

// Create an empty stack. The stack holds indexes of arr[] array which can be from l to r.

// The bars stored in stack are always in increasing order of their heights.

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

long max\_area = 0; // Initialize max area

int tp; // To store top of stack

long area\_with\_top; // To store area with top bar as the smallest bar

// Run through all bars of given histogram

int i = l;

while (i <= r)

{

// If this bar is higher than the bar on top stack, push it to stack

if (stack.empty() || arr[stack.peek()] <= arr[i])

stack.push(i++);

// If this bar is lower than top of stack, then calculate area of rectangle

// with stack top as the smallest (or minimum height) bar. 'i' is

// 'right index' for the top and element before top in stack is 'left index'

else

{

tp = stack.peek(); // store the top index

stack.pop(); // pop the top

// Calculate the area with arr[tp] stack as smallest bar

area\_with\_top = arr[tp] \* 1l \* (stack.empty() ? i-l : i - stack.peek() - 1);

// update max area, if needed

if (max\_area < area\_with\_top)

max\_area = area\_with\_top;

}

}

// Now pop the remaining bars from stack and calculate area with every

// popped bar as the smallest bar

while (stack.empty() == false)

{

tp = stack.peek();

stack.pop();

area\_with\_top = arr[tp] \* 1l \* (stack.empty() ? i-l : i - stack.peek() - 1);

if (max\_area < area\_with\_top)

max\_area = area\_with\_top;

}

return max\_area;

}

// ============================= End ==============================

}

class Solution {

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int heightsCount = Integer.parseInt(bufferedReader.readLine().trim());

long[] heights = new long[heightsCount];

for (int heightsItr = 0; heightsItr < heightsCount; heightsItr++) {

heights[heightsItr] = Long.parseLong(bufferedReader.readLine().trim());

}

int q = Integer.parseInt(bufferedReader.readLine().trim());

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

int l = Integer.parseInt(bufferedReader.readLine().trim());

int r = Integer.parseInt(bufferedReader.readLine().trim());

long res = Result.findMaxPossibleArea(heights, l, r);

bufferedWriter.write(res+"\n");

}

bufferedWriter.close();

}

}

/\*\*

\* Time complexity: O(n)

\* Space complexity: O(n)

\*/

Mine:

# Complete the findMaxPossibleArea function below.

def findMaxPossibleArea1(heights, l, r):

s = []

maxArea = 0

i = 0

while (i <= r-l):

if (not s or heights[l + s[-1]] <= heights[l+i]):

s.append(i)

i += 1

else:

tp = s[-1]

s.pop()

width = i - s[-1] -1 if s else i

area = width\*heights[l+tp]

maxArea = max(maxArea, area)

while s:

tp = s[-1]

s.pop()

width = r - l - s[-1] if s else r - l + 1

area = width\*heights[l+tp]

maxArea = max(maxArea, area)

return maxArea

def findMaxPossibleArea(h, l, r):

s = []

max\_area = 0

i = l

while i <= r:

if (not s or h[s[-1]] <= h[i]):

s.append(i)

i += 1

else:

top\_idx = s[-1]

s.pop()

width = (i - s[-1] - 1) if s else (i - l)

area = width \* h[top\_idx]

max\_area = max(max\_area, area)

while s:

top\_idx = s[-1]

s.pop()

width = (r - s[-1]) if s else (r - l + 1)

area = width \* h[top\_idx]

max\_area = max(max\_area, area)

return max\_area

# Pascal's Triangle

Problem Statement:

Pascal’s triangle is a triangular array of the binomial coefficients. Write a function that takes an integer value n as

input and returns 2d Array representing pascal’s triangle.

pascalTriangleArray is a 2D array of size n\*n, where

pascalTriangleArray[i][j] = BinomialCoefficient(i, j); if j<=i,

pascalTriangleArray[i][j] = 0; if j>i

Following are the first 6 rows of Pascal’s Triangle:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

Input/Output Format For The Function:

Input Format:

There is only one argument n, denoting the number of lines of Pascal's triangle to be considered.

Output Format:

Return an 2d integer array result, denoting pascal’s triangle where each value of result 2d array must be modulo with (10^9 + 7).

Size of result[i] for 0 <= i < n should be (i + 1) i.e. 0s for pascalTriangleArray[i][j] = 0; if j>i, should be ignored.

Input/Output Format For The Custom Input:

Input Format:

There should be one line for input, containing a single integer n, denoting the number of lines of Pascal's triangle to be considered.

If n = 6, then input should be:

6

Output Format:

There will be 2d array of integers, where each row of result 2d array will denotes row of pascal’s triangle in same order.

For input n = 6, output will be:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

Constraints:

* 1 <= n <= 1700

Sample Test Cases:

Sample Input 1:

4

Sample Output 1:

1

1 1

1 2 1

1 3 3 1

Explanation 1:

Pascal's Triangle for given n=4:

Using equation,

pascalTriangleArray[i][j] = BinomialCoefficient(i, j); if j<=i,

pascalTriangleArray[i][j] = 0; if j>i

Generated pascal’s triangle will be:

1

1 1

1 2 1

1 3 3 1

Sample Input 2:

6

Sample Output 2:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

Explanation 2:

Pascal's Triangle for given n=6:

Using equation,

pascalTriangleArray[i][j] = BinomialCoefficient(i, j); if j<=i,

pascalTriangleArray[i][j] = 0; if j>i

Generated pascal’s triangle will be:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution.java

Description:

A naive approach would be to calculate each (binomial coefficient % mod) separately. Binomial coefficient nCr = n!/((n-r)! \* r!). So, calculate numerator = (n! % mod) , denominator = (((n-r)! \* r!) % mod). Finally nCr can be found as ((numerator \* moduloInverse(denominator)) % mod).

Time Complexity:

O(n^3) where n is given number.

As there are n rows and each row can have n element in worst  cases. For calculating nCr for each element it will take O(n). Hence for n\*n elements it will take O(n^3).

Auxiliary Space Used:

O(1).

As we are not storing anything extra. (Here we are ignoring space used to store output 2d array result which will be O(n\*n))

Space Complexity:

O(n \* n).

As input is O(1), auxiliary space used is O(1) and output space is O(n \* n).

2) optimal\_solution.java

Description:

An optimal approach would be as follows:

As we know that for pascals triangle pascalsTriangle[i][j] = pascalsTriangle[i-1][j] + pascalsTriangle[i-1][j-1] and pascalsTrianlge[i][0] = 1 and pascalsTriangle[i][i]=1. For 0<=i<n and 0<=j<=i.

We use these facts and iterate each row and find out the pascalsTriangle.

Time Complexity:

O(n^2) where n is given number.

As there are n rows and each row can have n element in worst  cases so, to iterate over n\*n elements it will take O(n^2).

Auxiliary Space Used:

O(1).

As we are not storing anything extra. (Here we are ignoring space used to store output 2d array result which will be O(n\*n))

Space Complexity:

O(n \* n).

As input is O(1), auxiliary space used is O(1) and output space is O(n \* n).

## Solutions Provided By Us

import java.io.\*;

import java.math.\*;

import java.security.\*;

import java.text.\*;

import java.util.\*;

import java.util.concurrent.\*;

import java.util.function.\*;

import java.util.regex.\*;

import java.util.stream.\*;

import static java.util.stream.Collectors.joining;

import static java.util.stream.Collectors.toList;

class Result {

// -------------------- START ----------------------

/\*

\* Complete the 'findPascalTriangle' function below.

\*

\* The function accepts INTEGER as parameter.

\* Return 2D INTEGER ARRAY.

\*/

static List<List<Integer>> findPascalTriangle(int n) {

int mod = 1000000007;

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

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

// Every ith row has number of integers

// equal to row number

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

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

// First and last values in every row are 1

if (i == j || j == 0){

row.add(1);

}

// Other values are sum of values just

// above and left of above

else{

row.add((result.get(i-1).get(j-1) + result.get(i-1).get(j))%mod);

}

}

result.add(row);

}

return result;

}

// -------------------- END ----------------------

}

class Solution{

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int n = Integer.parseInt(bufferedReader.readLine().trim());

List<List<Integer>> result = Result.findPascalTriangle(n);

result.stream()

.map(

r -> r.stream()

.map(Object::toString)

.collect(joining(" "))

)

.map(r -> r + "\n")

.collect(toList())

.forEach(e -> {

try {

bufferedWriter.write(e);

} catch (IOException ex) {

throw new RuntimeException(ex);

}

});

bufferedWriter.close();

bufferedReader.close();

}

}

/\*\*

\* Time complexity: O(n^2)

\* Space complexity: O(n^2)

\*/

Mine:

#

# Complete the findFofPascalTriangle function below.

#

def findFofPascalTriangle(n):

# Sum of the binomial coefficients

# The formula

# for k = 0-n ∑ nCk = 2 \*\* n

# says the elements in the nth row of Pascal's triangle always

# add up to 2 raised to the nth power

#

# The problem asks to report the

# F(array) = Summation over all i ( Summation over all j (

# (i+1)\*array[i][j] + (j+1) ) ) % (10^9 + 7),

# where 0<=i<n and 0<=j<=i

#

# This in other words is

# F(array) = Sum of the row \* row\_idx + sum of total number of elems in a row

# Sum of the row in sum of all binomial coefficient is (2 \*\* n)

# that translates to

# F(array) = n \* (2 \*\* n) + (n \* (n + 1))/2

target = 0

i = 1

#for i in range(1, n+1):

while i < n+1:

target += i \* (2 \*\* (i - 1)) + (i \* (i + 1))//2

i += 1

return target % (10 \*\* 9 + 7)

# Sum Zero

Problem Statement:

Given an array of integers arr of size n, find a non-empty subarray resSubArray such that sum of elements in resSubArray is zero.

Input/Output Format For The Function:

Input Format:

There is only one argument, arr denoting input array.

Output Format:

Return an array of integer res of size 2, where res[0] and res[1] denotes start index and end index(0 based indexing) (both inclusive) respectively for resSubArray in arr.

Note that:

* If there is no such subarray, then return array res of size one and res[0] = -1.
* If there are multiple such subarray, then return indices for any one of them.
* If a matching subarray is a subarray of a larger matching subarray, then return indices for either one.
* If there is a number '0' in the array arr, then it counts as a valid sum zero subarray.

Input/Output Format For The Custom Input:

Input Format:

The first line of the input should contain a single integer n, denoting the size of input array arr.

In the next n lines, ith line should contain an integer denoting arr[i].

If n = 6 and arr = [5, 1, 2, -3, 7, -4], then input should be:

6

5

1

2

-3

7

-4

If n = 5 and arr = [1, 2, 3, 5, -9], then input should be:

5

1

2

3

5

-9

Output Format:

There are two cases here:

1. If a valid sum zero subarray exists in arr, then there will be two lines for output. First line will have an integer res[0] and second line will have an integer res[1], denoting starting index and ending index of required subarray (0 based indexing, both inclusive).

2. Otherwise if there is no valid sum zero subarray, there will be only one line for output, having an integer -1.

For input n = 6 and arr = [5, 1, 2, -3, 7, -4], output will be:

1

3

For input n = 5 and arr = [1, 2, 3, 5, -9], output will be:

-1

Constraints:

* 1 <= n <= 5\*10^5
* -10^9 <= arr[i] <= 10^9, (i = 0,1,...,(n-1))

Sample Test Cases:

Sample Input 1:

6

5

1

2

-3

7

-4

Sample Output 1:

1

3

Explanation 1:

For given input array arr, arr[1]+arr[2]+arr[3] = 1+2+(-3) = 0. So, subarray starting from index 1 upto index 3 (0 based indexing) is sum zero subarray.

(3,5 and 1,5 are the other correct solutions)

Sample Input 2:

5

1

2

3

5

-9

Sample Output 2:

-1

Explanation 2:

For given input array arr, there is no subarray such that sum of integers in that subarray is 0.

So, -1 is printed as output.

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution.java

Description:

A naive approach would be, to iterate over all possible subarrays of input array arr, such that while on subarray [i,j], i.e. subarray starting from ith index and ending at jth index, find sum of elements in it and if it's zero, return {i,j}. If no such subarray found, return {-1}.

Time Complexity:

O(n\*n) where n is length of input arr.

As we are iterating over all possible subarrays of input array arr, time complexity will be O(n\*n).

Auxiliary Space Used:

O(1).

We are not storing anything extra.

Space Complexity:

O(n) where n is length of input arr.

To store input it takes O(n) and as auxiliary space used is O(1).

Hence, O(n) + O(1) → O(n).

2) optimal\_solution.java

Description:

An optimal approach would be as follows:

Notice that if there exists a zero sum subarray [i,j] in a given input array arr, then prefix sum (denote it as prefix where prefix[k] = arr[0] + arr[1] + arr[2] + ... + arr[k]) prefix[j] should be equal to prefix[i-1], as prefix[j] = prefix[i-1] + (arr[i] + arr[i+1] + ... + arr[j]), where the term in bracket is sum of subarray [i,j], which is 0.

Considering this fact, build prefix sum array prefix. If for some i, j, 0 <= i <= j < n, prefix[i-1] = prefix[j], then subarray [i,j] is the zero sum subarray.

Time Complexity:

O(n) where n is length of input arr.

To find out if any two sums of subarrays are equal or not we will store them in HashMap as prefix[k] (i.e. sum) as key and k as value. To maintain hashmap it will take O(n) time complexity in worst case to get and store n sums.

Auxiliary Space Used:

O(n) where n is length of input arr.

We are using hashmap to store sums. It will take O(n) of space.

Space Complexity:

O(n) where n is length of input arr.

To store input it takes O(n) and as auxiliary space used is O(n).

Hence, O(n) + O(n) → O(n).

## Solutions Provided By Us

/\*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PROBLEM DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Given an array of integers arr of size n, find a subarray whose sum is zero.

\*/

import java.io.\*;

import java.util.\*;

class Result {

// -------------------- START ----------------------

static int[] sumZero(int[] arr) {

// To store interval (start, end) for which sum is zero

int[] res = new int[2];

// To store prefix sum i.e. sum of subarray starting at index 0 and ending at index i

// Key of hashmap will be sum and value will be index i for prefix sum

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

// To check whether prefix sum it self is equal to zero

map.put(0l, -1);

// To store current sum

long sum = 0;

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

// To check if we encountered with value which itself is zero

if(arr[i]==0){

res[0]=i;

res[1]=i;

return res;

}

// Adding current value in current sum

sum += arr[i];

// If we found value sum in our hashmap means we have encountered with this sum before

// means arr[0, map.get(sum)] = sum and

// arr[0, map.get(sum)] + arr[map.get(sum)+1, i] = sum

// which implies arr[map.get(sum)+1, i] = 0 and hence interval we are looking for is

// start = map.get(sum)+1 and end = i

if (map.containsKey(sum)) {

res[0] = map.get(sum) + 1;

res[1] = i;

return res;

} else {

map.put(sum, i);

}

}

// If no subarray having sum = 0 found then we will return [-1]

return new int[]{-1};

}

// -------------------- END ----------------------

}

class Solution{

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int n = Integer.parseInt(bufferedReader.readLine());

int[] arr = new int[n];

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

arr[i] = Integer.parseInt(bufferedReader.readLine());

}

int[] resSubArray = Result.sumZero(arr);

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

bufferedWriter.write(String.valueOf(resSubArray[i])+"\n");

}

bufferedWriter.newLine();

bufferedWriter.close();

bufferedReader.close();

}

}

/\*\*

\* Time complexity: O(n)

\* Space complexity: O(n)

\*/

Mine:

# Complete the sumZero function below.

def sumZero(a):

#n = len(a)

sum = 0

sum\_map = {}

res = [None] \* 2

for i in range(len(a)):

if not a[i]:

res[0] = i

res[1] = i

return res

else:

sum += a[i]

if sum in sum\_map:

res[0] = sum\_map[sum] + 1

res[1] = i

return res

elif not sum:

res[0] = 0

res[1] = i

return res

sum\_map[sum] = i

res[0] = -1

return res

def sumZero\_brute(arr):

n = len(arr)

for i in range(n):

total = arr[i]

if not total:

return [i, i]

for j in range(i+1, n):

total += arr[j]

if not total:

return [i, j]

return [-1]

# Minimum Element In A Sorted And Rotated Array

Problem Statement:

You are given a sorted array arr which is rotated by unknown pivot k. You need to find minimum element from given array using fastest possible way which uses only constant space.

Input Format:

Only argument for function, integer array named arr.

Output Format:

Return integer which is minimum element in given array.

Constraints:

1 <= n <= 10^5 where n is number elements in given array.

Every element of array will be unique.

For every element arr[i],

-10^9 <= arr[i] <= 10^9 where 0 <= i <= (n-1)

Sample Test Case:

Sample Input:

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

Sample Output:

1

Explanation:

For given arr = [ 4, 5, 6, 7, 8, 1, 2, 3] which is sorted in ascending order and right rotated by pivot 5 has minimum value as 1 at index 5.

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution.java

Time Complexity:

O(n) where n is number of elements in array.

This approach is very simple, We just need to iterate over the given array and maintain the minimum value found and return that minimum value which will be our answer.

Auxiliary Space Used:

O(1).

As we are not storing anything.

Space Complexity:

O(n).

Input is O(n) because we are storing n elements of array and auxiliary space used is O(1). So, O(n) + O(1) -> O(n).

2) suboptimal\_solution.java

Time Complexity:

Time complexity for the function find\_minimum: O(log n) where n is number of elements in array.

Time complexity for the complete program: O(n) where n is number of elements in array, because size of input is n.

In this approach we used recursive binary search.

If we take some examples and look closely, we would observe some patterns:

If array was previously sorted in ascending order:

* The minimum element is the only element whose previous element is greater than it.
* If we found any subarray ( from low to high ) which is ascending sorted then minimum element will be element at low.
* Else minimum element lies in either left half or right half.
* If middle element is greater than element at low, then the minimum element lies in right half.
* Else minimum element lies in left half.

If array was previously sorted in descending order:

We use these patterns to make solution:

* The minimum element is the only element whose next element is greater than it.
* If we found any subarray ( from low to high ) which is descending sorted then minimum element will be element at high.
* Else minimum element lies in either left half or right half.
* If middle element is less than element at low, then the minimum element lies in right half.
* Else minimum element lies in left half.

As the time complexity of binary search will be

T(n) = T(n/2) + c ( Each iteration reducing array in half ).

The above function can be solved either using recurrence Tree method or Master method. It falls in case II of Master Method and solution of the function is O(log n) hence, complexity of our solution (find\_minimum function) is O(log n).

Auxiliary Space Used:

O(log n) where n is number of elements in array.

Similarly by above logic for time complexity, number of recursive calls will be O(log n) and hence size of function stack used will be O(log n).

Space Complexity:

O(n) where n is number of elements in array.

Input is O(n) because we are storing n elements of array and auxiliary space used is O(1). So, O(n) + O(1) -> O(n).

3) optimal\_solution.java

Time Complexity:

Time complexity for the function find\_minimum: O(log n) where n is number of elements in array.

Time complexity for the complete program: O(n) where n is number of elements in array, because size of input is n.

Here we are using iterative approach of binary search. Explanation will same as mentioned above for suboptimal\_solution.

Auxiliary Space Used:

O(1).

As we are using only constant extra space.

Space Complexity:

O(n) where n is number of elements in array.

Input is O(n) because we are storing n elements of array and auxiliary space used is O(1). So, O(n) + O(1) -> O(n).

## Solutions Provided By Us

import java.io.\*;

import java.math.\*;

import java.security.\*;

import java.text.\*;

import java.util.\*;

import java.util.concurrent.\*;

import java.util.function.\*;

import java.util.regex.\*;

import java.util.stream.\*;

class Result {

/\*

\* Complete the 'find\_minimum' function below.

\*

\* The function accepts INTEGER ARRAY as parameter.

\* Return INTEGER.

\*/

// ============================ Start ============================

public static int find\_minimum(List<Integer> arr) {

int n = arr.size();

if(n==1) {

return arr.get(0);

}

if(n==2) {

return Math.min(arr.get(0), arr.get(1));

}

if(n==3) {

return Math.min(arr.get(0), Math.min(arr.get(1), arr.get(2)));

}

/\*

\* All numbers of array are unique as given in question so don't consider the cases when

\* numbers are equal and get confuse.

\*/

/\*

\* consider example [4, 7, 8, 10, 15].

\* for this given array was [4, 7, 8, 10, 15] which was sorted in ascending order

\* and rotated right by 0

\*/

if(arr.get(0)-arr.get(n-1)<0&&arr.get(0)-arr.get(1)<0&&arr.get(n-2)-arr.get(n-1)<0) {

return arr.get(0);

}

/\*

\* consider example [15, 10, 8, 7, 4].

\* for this given array was [15, 10, 8, 7, 4] which was sorted in ascending order

\* and rotated right by 0

\*/

if(arr.get(0)-arr.get(n-1)>0&&arr.get(0)-arr.get(1)>0&&arr.get(n-2)-arr.get(n-1)>0) {

return arr.get(n-1);

}

if(arr.get(0)-arr.get(n-1) > 0) {

/\*

\* consider example [10, 13, 15, 4, 6, 8].

\* for this given array was [4, 6, 8, 10, 13, 15] which was sorted in ascending order

\* and rotated right by 3

\*/

return find\_minimum\_in\_increasing(arr);

}

else {

/\*

\* consider example [8, 6, 4, 15, 13, 10].

\* for this given array was [15, 13, 10, 8, 6, 4] which was sorted in descending order

\* and rotated right by 3

\*/

return find\_minimum\_in\_decreasing(arr);

}

}

public static int find\_minimum\_in\_increasing(List<Integer> arr) {

int low = 0;

int high = arr.size()-1;

while(low<=high) {

if(arr.get(low)-arr.get(high)<=0) {

return arr.get(low);

}

int mid = (low+high)/2;

if(arr.get(mid)-arr.get(low)>=0) {

//Minimum is in right subarray

low = mid + 1;

}else {

//Minimum is in left subarray

high = mid;

}

}

return -1;

}

public static int find\_minimum\_in\_decreasing(List<Integer> arr) {

int low = 0;

int high = arr.size()-1;

while(low<=high) {

if(arr.get(low)-arr.get(high)>=0) {

return arr.get(high);

}

int mid = (low + high)/2;

if(arr.get(mid)-arr.get(low)<0) {

//Minimum is in right subarray

low = mid;

}else {

//Minimum is in left subarray

high = mid;

}

}

return -1;

}

// ============================= End ==============================

}

class Solution {

public static void main(String args[]) {

/\*

This function is used to increase the size of recursion stack. It makes the size of stack

2^26 ~= 10^8

\*/

new Thread(null, new Runnable() {

public void run() {

try{

solve();

}

catch(Exception e){

e.printStackTrace();

}

}

}, "1", 1 << 26).start();

}

public static void solve() throws IOException {

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

int n = Integer.parseInt(bufferedReader.readLine().trim());

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

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

arr.add(Integer.parseInt(bufferedReader.readLine().trim()));

}

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int result = Result.find\_minimum(arr);

bufferedWriter.write(result+"\n");

bufferedWriter.close();

}

}

# Hamming Weight

Problem Statement:

Hamming Weight of an integer x is defined as the total number of set bits in the binary representation of the integer x. Now, given an array s of n integers. We need to calculate the total hamming weight of the array s i.e. to sum-up the individual hamming weight of each integer in the array S.

Input/Output Format For The Function:

Input Format:

First and only parameter of the function that is to be implemented the array of 64-bit integers s.

Output Format:

The function returns an integer variable storing the value of the total hamming weight of the given input array s.

Input/Output Format For The Custom Input:

Input Format:

First line of the input contains one single integer n, denoting number of elements in array s.

Next n lines of the input, each line contains single integer denoting the ith element in the array s.

If n = 3 and s = [ 1, 2, 3 ], then custom input format will be:

3

1

2

3

Output Format:

Print one single line containing one integer denoting the total hamming weight of the input array s.

For the above provided custom input, output would be:

4

Constraints:

* 1 <= n <= 10^5
* 0 <= s[i] < 2^32 where 0 <= i < n.

Sample Test Case:

n = 3

s = [ 1, 2, 3 ]

Sample Output:

4

Explanation:

For the array s = [1, 2, 3], number of set bits for each element of the array is mentioned below:

Binary representation of 1 is “1” so set bits in 1 is 1.

Binary representation of 2 is “10” so set bits in 2 is 1.

Binary representation of 3 is “11” so set bits in 3 are 2.

So, total set bits in 1 + set bits in 2 + set bits in 3 are 1 + 1 + 2 = 4. Hence Summing hamming weight of given array [1, 2, 3] is 4.

We have provided solutions which contain necessary comments to understand the approach used:

1) brute\_force\_solution:

Description:

As per the constraints, all the elements in the array can be stored in a 32-bit integer and hence, to calculate the number of set bits in an integer x, we can iterate on all these 32 bits of the corresponding integer and keep a count of the number of set bits.

We repeat the same process for all integers in the given input array s and keep the count of total set bits in all integers. To optimize the solution we can break traversal over the bits once we encounter the MSB(Most Significant Bit / Leftmost set bit) in the integer x.

Time Complexity:

O(n\*32) where n is number elements in the given input array.

For every integer in the array, we are iterating over 32 bits in its binary representation. So, for n integers, time complexity becomes O(n\*32).

Auxiliary Space Used:

O(1).

Since we are only traversing on the bits of the integers in array s without storing any data regarding set bits and hence, the Auxiliary Space complexity is O(1).

Space Complexity:

O(n) where n is number of elements in the given array s.

For storing input, it will take O(n) and as auxiliary space used is O(1) hence total complexity will be O(n) + O(1) → O(n).

2) optimal\_solution:

Description:

Our main aim of the solution is to calculate the hamming distance of an integer x in constant time with some precomputation.

As the integer size is 32 bit as per the input constraints. So, we can divide the 32-bit integer x, into two 16-bits integers.

[31th , 30th , ……. , 17th, 16th] [ 15th , 14th, ………. , 1th , 0th]

A B

Let’s call the first part i.e. from [31th bit to 16th bit] as A and second part from [15th bit to 0th] bit as B. Also, note that both these integers A and B are 16-bit integers. Now, the total set bits in the integer x is equal to number of set bits in integer A + number of set bits in integer B and this is the key idea for this solution. Now let Sz be the number of all possible 16-bit integers. So, we precompute the number of set bits for all Sz integers and store it in memory. We can compute the set bits for all Sz integers in linear time. Let’s say dp[i] denotes the number of set bits in integer i. So, we can compute dp[i] using the below state relation :

dp[i] = dp[i >> 1] + (i&1)

In the above relation (i&1) tells if the 0th bit is set in the binary representation of the integer i. To illustrate the above relation, consider the calculation of dp[5].

Now, (5)base10 = (101)base2

So, dp[5] = dp[5 >> 1] + 5&1

Here 5&1 is 1 as the 0th bit is set in binary representation of 5.

As now, we have taken 0th bit of 5 under consideration and hence, now we can right shift the binary representation of 5 by 1 to omit the 0th bit and then calculate the number of set bits in the resulting integer. Also, as right shifting 5 by 1 will result in an integer which is less than 5 and as we are iteratively computing dp states the resultant state dp[5>>1] would have already been computed.

Hence, dp[5] = dp[2] + 1 i.e. last bit in 5 plus the number of set bits in 2.

Once, we have precomputed set bits individually for all 16-bit integers. We can answer calculate the number of set bits in a 32-bit integer by two lookups in the precomputed state values.

So, for an integer x we first divide it into A and B as explained above and then we do a lookup in our dp[A] and dp[B] to get the set bits in the integer x.

We repeat the same process for all integers in the array s and keep count of the total number of set bits and hence, the hamming weight of the array.

Also, instead of dividing the array into 2 parts, we can divide it into 4 integers each of size 8 bits and proceed the same way as we did in the above explanation. This will reduce the space complexity significantly, but will require 4 lookups and hence will double the previous time complexity. Though, the asymptotics remain the same.

Bonus take away – this is called as the Space-Time trade off.

Kindly, refer to the solution for implementation details.

Time Complexity:

O(n + Sz) where n is number of elements in the given array s and Sz be the number of all possible 16-bit integers i.e. 2^16.

Precomputing dp state for all 16 bit integers take a linear time O(2^16) as explained above. To calculate the set bits in an integer x we are performing 2 iterations. Hence, for all n integers the time complexity become O(2\*n). Summing up the overall time complexity becomes O( 2\*n + Sz ) →  O(n + Sz).

Auxiliary Space Used:

O(Sz) where Sz be the number of all possible 16-bit integers i.e. 2^16.

As, we are pre-computing set bits for all 16 bit integers and storing it. Hence the space complexity is O(Sz).

Space Complexity:

O(n + Sz) where n is number of elements in the given array s and Sz be the number of all possible 16-bit integers i.e. 2^16.

For storing input, it will take O(n) and as auxiliary space used is O(Sz) hence total complexity will be O(n) + O(Sz) → O(n + Sz).

## Solutions Provided By Us

#include "bits/stdc++.h"

using namespace std;

// ------------------------------ START ------------------------------

int calculateHammingWeight(vector<long long> &s)

{

// size of mem dp table

int sz = 1 << 16;

// number of elements in s

int N = s.size();

// stores set bits in integers

int memo[sz];

// 0 set bits in integer 0

memo[0] = 0;

// using dp-state relation to populate

// all dp states

for (int i = 1; i < sz; i++)

{

memo[i] = memo[i >> 1] + (i & 1);

}

// total set bits in all N elements of s

int totalSetBits = 0;

// bit mask = (1<<16) - 1 = (1111111111111111) in binary

int bitMask = sz - 1;

// iterate over all elements in array

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

{

// add set bits from (0th to 15th) bits position

totalSetBits += memo[s[i] & bitMask];

// shift s[i] 16 positions to right

s[i] = s[i] >> 16;

// again add set bits from (0th to 15th) bits position

totalSetBits += memo[s[i] & bitMask];

}

return totalSetBits;

}

// ------------------------------ STOP ------------------------------

int main(int argc, char const \*argv[])

{

// freopen(

// "..//test\_cases//handmade\_test\_cases\_input.txt", "r",

// stdin);

// freopen(

// "..//test\_cases//handmade\_test\_cases\_expected\_output.txt", "w",

// stdout);

// freopen(

// "..//test\_cases//generated\_big\_test\_cases\_input.txt", "r",

// stdin);

// freopen(

// "..//test\_cases//generated\_big\_test\_cases\_expected\_output.txt", "w",

// stdout);

// string testcases;

// getline(cin, testcases);

// int t = stoi(testcases);

// while (t--)

// {

int n;

cin >> n;

vector<long long> s(n);

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

{

cin >> s[i];

}

int ans = calculateHammingWeight(s);

cout << ans << endl;

// }

return 0;

}

# Run Length Encoder

Problem Statement:

Compress a string (only has alphabet characters), with basic encoding, where you simply count the number of repeated characters. Then also write a routine to decompress it.

e.g.

Input: "AAAAA"

Output: "5A"

Input: "BAAAB"

Output: "B3AB"

Input: "ABAB"

Output: "ABAB" [We are not concerned about characters repeating in groups]

Assume that a given character will not repeat more than 127 times.

Input format:

There is only one argument named strInput denoting the input string.

Output format:

Return the compressed string

Constraints:

* String consists of alphabetic characters only
* 1 <= length\_of\_input\_string <= 6500

Sample Test Cases:

Sample Input 1:

AAAAA

Sample Output 1:

5A

Explanation 1:

Character “A” is repeated 5 times consecutively.

Sample Input 2:

ABaaaBCC

Sample Output 2:

AB3aB2C

Explanation 2:

Character “a” is repeated 3 times in consecutively, character “C” is repeated 2 times consecutively.

We have provided only optimal solution for this problem. The problem asks to encode the input string in such a way so that it’s length remain same or decrease. To do so, we counted repeated consecutive characters. If the count is more than one, we replaced the repeated portion by the number following the character only.

Time complexity:

O(N)

Auxiliary space:

O(N) because of storing output

Space complexity:

Including input, O(N).

## Solutions Provided By Us

#include <bits/stdc++.h>

using namespace std;

// ============================ Start ============================

string convertToString(int num){

string ret="";

while(num>0){

int rem = num % 10;

ret+=(char)(rem+'0');

num/=10;

}

reverse(ret.begin(), ret.end());

return ret;

}

string RLE(string strInput){

string ans="";

int cnt=1;

char prev = strInput[0];

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

if(strInput[i]==prev){

cnt++;

} else {

if(cnt>1){

ans+=convertToString(cnt);

}

ans+=prev;

cnt=1;

prev = strInput[i];

}

}

if(cnt>1){

ans+=convertToString(cnt);

}

ans+=prev;

return ans;

}

// ============================ End ============================

void solve(string inputFile, string outputFile){

ifstream fin(inputFile);

ofstream fout(outputFile);

cerr<<"Running "<<inputFile<<endl;

int testCase;

fin>>testCase;

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

string s;

fin>>s;

string ans = RLE(s);

fout<<ans<<endl;

}

}

int main(){

solve("..//test\_cases//sample\_test\_cases\_input.txt", "..//test\_cases//sample\_test\_cases\_expected\_output.txt");

solve("..//test\_cases//handmade\_test\_cases\_input.txt", "..//test\_cases//handmade\_test\_cases\_expected\_output.txt");

solve("..//test\_cases//generated\_big\_test\_cases\_input.txt", "..//test\_cases//generated\_big\_test\_cases\_expected\_output.txt");

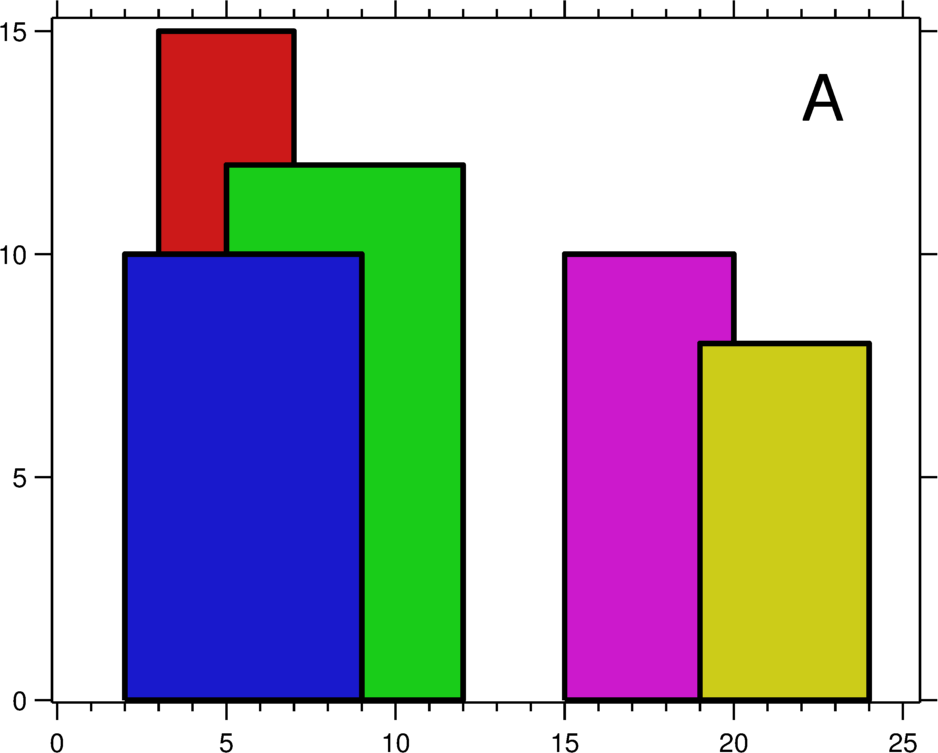
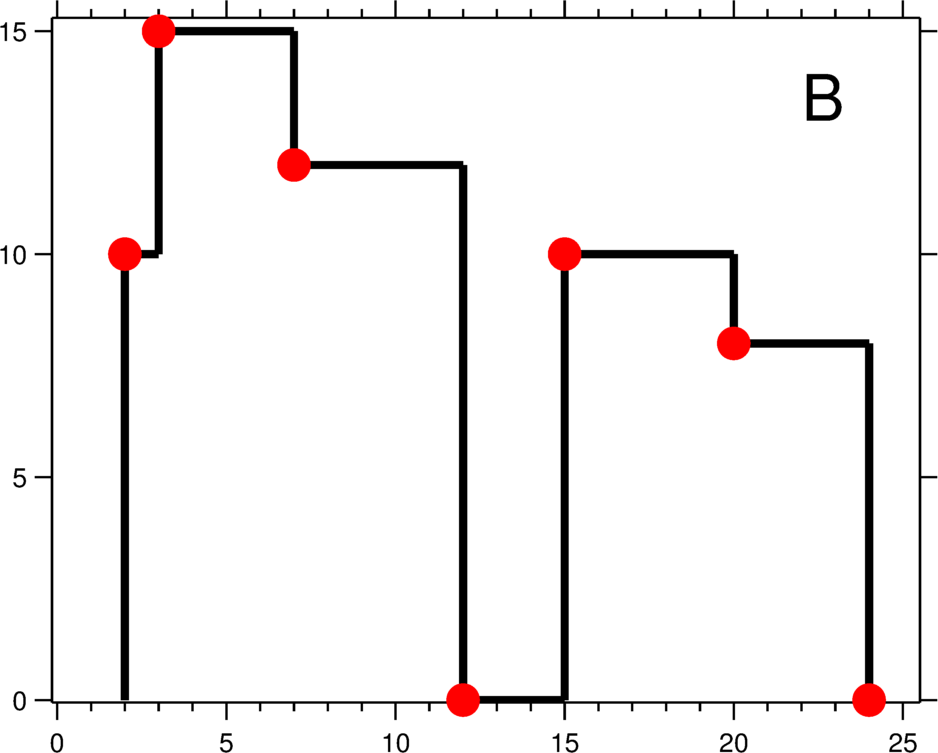
return 0;

}

# Skyline

Problem Statement:

Given n buildings on a 2D plane, find the skyline of these buildings. Each building on the 2D plane has a start coordinate, end coordinate, and height. The skyline is defined as a unique representation of rectangular strips of different heights which are created after the overlap of multiple buildings in the 2D plane. Refer this image for clarity, on the left are multiple buildings and on the right is the skyline for the same. The output skyline is sorted on the x coordinate of the rectangular strip.



Input/Output Format For The Function:

Input Format:

The input of buildings is given in a 2D integer array format. The outer array contains multiple buildings where each building is an array of integers of size 3. The first integer represents the start coordinate of the building, the second integer represents the end coordinate of the building and the third integer represents its height.

Output Format:

Return a 2D integer array. The outer array has different rectangular strips, each element is an array of size two. The first element in the inner array is the x coordinate of the strip and the second element is the y coordinate of the strip (red dots in the image above).

Input/Output Format For The Custom Input:

Input Format:

The first line in the input contains an integer n, the number of buildings. The next n lines contain three integers: x, y, height representing the buildings.

Output Format:

Each line represents the unique rectangular strip in a sorted format. Each line has 2 integers, x0, and y0 coordinates respectively of the strip.

Constraints:

* 1 <= n <= 10^5
* 1 <= x, y, height <= 2 \* 10^9

Sample Test Case:

Sample Input:

5

2 9 10

3 7 15

5 12 12

15 20 10

19 24 8

Sample Output:

2 10

3 15

7 12

12 0

15 10

20 8

24 0

Explanation:

From the image referenced above, we see the blue building at the start and the corresponding red dot in the right image at (2,10). The next change in skyline occurs at an x coordinate of 3 with red building coming up at the height of 15, so in the output, the next line is printed as 3 15. Similarly, all the buildings are traversed to find the output as given in the sample output

1) optimal\_solution.java

Description:

A BuildingIndex class is defined in the solution consisting of three members:

* index: it is the value of x-coordinate of the potential skyline point.
* startEnd: it is a character that denotes if this particular index is a start point or an end point of the building.
* height: it is the height of the building.

A priority queue named priorityQ is created. All the start and end points of all the buildings as objects of BuildingIndex class are added in this queue with the following insertion constraints:

* A lower value of index gets priority.
* If the index values are the same, then 'start' point gets priority.
* If two buildings are starting at the same index, then BuildingIndex with higher height gets priority.
* If two buildings are ending at the same index, then BuildingIndex with smaller height gets priority.

A binary search tree named heightCountQ is created which stores the count of heights ordered by the heights of buildings.

Also, a variable named maxHeight is initialized with 0 which stores the current max height of the skyline.

Now pop elements from the priorityQ, one by one, till it is empty.

* If the popped element is the starting index, then check

if it is greater than maxHeight, if so add the start index and height pair to the ans list.

* If the popped element is the end index, then check if it was the maxHeight till now, if yes, then update the ans list with the current index and height from next tallest building from the heightCountQ. And decrease the count in heightCountQ, if the count is 1, then the node is removed from the tree.

Time Complexity (assuming that input arguments are already given and excluding time used in the declaration of output):

Sorting all the buildings as per the given constraints in the description would take O(2n\*log(2n)), where n is the number of buildings in the given array and 2\*n is the BuildingIndex. Also maintaining the count of heights take O(n\*log(n)), since each insertion and deletion is O(log(n)) in a BST.

So, the total time complexity is O(n\*log(n)).

Time Complexity:

Time to read the input is O(3\*n) since for each line we have 3 integer values. For output, we can have 2\*n skyline points in worst case scenario. Each output line will have 2 integer values, so the output time complexity is 2\*O(2\*n).

Overall time complexity is O(n\*log(n)) + O(n) + O(n) ~= O(n\*log(n))

Auxiliary Space Used:

The priorityQ and heightCount both require O(n) auxiliary space to store n buildings. So, total auxiliary space complexity is O(n).

Space Complexity:

Input and output arrays both require 3\*n and 2\*n amount of space respectively. Total space complexity including auxiliary space comes out to be O(n).

## Solutions Provided By Us

import java.util.\*;

public class optimal\_solution {

public static class BuildingIndex {

int index;

char startEnd;

Integer height;

public BuildingIndex(int index, char startEnd, int height) {

this.index = index;

this.startEnd = startEnd;

this.height = height;

}

@Override

public String toString() {

return index + " " + startEnd + " ";

}

}

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int numOfTestCases = in.nextInt();

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

int numOfBuildings = in.nextInt();

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

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

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

building.add(in.nextInt());

building.add(in.nextInt());

building.add(in.nextInt());

buildings.add(building);

}

List<List<Integer>> skyline = findSkyline(buildings);

for(List<Integer> point : skyline){

for(Integer value : point){

System.out.print(value + " ");

}

System.out.println("");

}

System.out.println("");

}

}

private static List<List<Integer>> findSkyline(List<List<Integer>> buildings) {

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

int numOfBuildings = buildings.size();

TreeSet<BuildingIndex> priorityQ = new TreeSet<>(new Comparator<BuildingIndex>() {

@Override

public int compare(BuildingIndex o1, BuildingIndex o2) {

if (o1.index != o2.index) {

return o1.index - o2.index;

}

if (o2.startEnd != o1.startEnd) {

// start index is higher priority than end index

if (o2.startEnd == 'e') {

return -1;

} else {

return 1;

}

}

// If two buildings start at the same point, then building with higher height should be higher priority.

if (o1.startEnd == 's') {

return o2.height - o1.height;

}

// If two buildings end at the same point, then building with smaller height should be looked at first.

else {

return o1.height - o2.height;

}

}

});

// Creating the priority queue of starting and end indices of all buildings

for (List<Integer> building : buildings) {

BuildingIndex buildingIndex1 = new BuildingIndex(building.get(0), 's', building.get(2));

priorityQ.add(buildingIndex1);

BuildingIndex buildingIndex2 = new BuildingIndex(building.get(1), 'e', building.get(2));

priorityQ.add(buildingIndex2);

}

// A map (heightCountQ) which keeps track of all buildings so far in decreasing order of their heights.

TreeMap<Integer, Integer> heightCountQ = new TreeMap<>();

heightCountQ.put(0, 1);

Integer maxHeight = 0;

while (!priorityQ.isEmpty()) {

BuildingIndex buildingIndex = priorityQ.pollFirst();

// starting index signifies start of a buiding, so check if highest.

if (buildingIndex.startEnd == 's') {

heightCountQ.putIfAbsent(buildingIndex.height, 0);

heightCountQ.put(buildingIndex.height, heightCountQ.get(buildingIndex.height) + 1);

// if tallest buiding detected then update the skyline.

if (buildingIndex.height > maxHeight) {

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

list.add(buildingIndex.index);

list.add(buildingIndex.height);

maxHeight = buildingIndex.height;

ans.add(list);

}

}

// end index signifies end of a buiding, so remove it from the heightCountQ, and update skyline if highest.

else {

if (heightCountQ.get(buildingIndex.height) == 1) {

heightCountQ.remove(buildingIndex.height);

} else {

heightCountQ.put(buildingIndex.height, heightCountQ.get(buildingIndex.height) - 1);

}

// update skyline, if tallest buiding ends.

if (maxHeight.equals(buildingIndex.height)) {

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

list.add(buildingIndex.index);

Map.Entry<Integer, Integer> val = heightCountQ.pollLastEntry();

if (!maxHeight.equals(val.getKey())) {

list.add(val.getKey());

ans.add(list);

}

heightCountQ.put(val.getKey(), val.getValue());

maxHeight = val.getKey();

}

}

}

return ans;

}

}

# Find The Next Palindromic Number

Problem Statement:

Given a number n, you have to find next palindromic number pal. To be precise, you have to find an integer pal, which is smallest palindromic number, greater than n.

Input/Output Format For The Function:

Input Format:

There is only one argument denoting integer n.

Output Format:

Return the required number pal.

Input/Output Format For The Custom Input:

Input Format:

There should be only one line, containing an integer n.

If n = 5, then input should be:

5

Output Format:

There will be one line, containing resultant integer pal.

For input n = 5, output will be:

6

Constraints:

* 0 <= n <= 2147483647

Sample Test Cases:

Sample Test Case 1:

Sample Input 1:

5

Sample Output 1:

6

Explanation 1:

6 is a palindromic number, and bigger than 5. There is no palindromic number less than 6 and bigger than 5.

Sample Test Case 2:

Sample Input 2:

10

Sample Output 2:

11

Notes:

Suggested time in interview: 30 minutes.

The “Suggested Time” is the time expected to complete this question during a real-life interview, not now in homework i.e. For the first attempt of a given homework problem, the focus should be to understand what the problem is asking, what approach you are using, coding it, as well as identifying any gaps that you can discuss during a TA session. Take your time, but limit yourself to 2 one hour sessions for most problems.

If you are getting wrong answer, then first thing you should make sure is to use appropriate data type. (using int will not work, it will overflow for given constratins!)

We have provided two solutions, optimal\_solution1.cpp and optimal\_solution2.cpp.

First have a look at optimal\_solution1.cpp (it might look lengthy, but we have focused on readability of the code) and then look at optimal\_solution2.cpp. Try some examples to understand the solutions clearly.

optimal\_solution2.cpp is different from most of the solutions you will see online. It will take some time to understand initially, but after that in interviews it will be less error prone to write the solution.

In optimal\_solution1.cpp we can use int array instead of strings to simplify some of the steps, but it will take more memory. int is 4 bytes while char is only 1 byte. So 3 times more memory!

For both the solutions:

Time Complexity:

O(n).

Auxiliary Space Used

O(n).

Space Complexity:

O(n).

## Solutions Provided By Us

#include<bits/stdc++.h>

using namespace std;

// -------------------------- START --------------------------

// Function to convert string to long long int.

// Alternatively you can use the built in function "stoll" introduced by C++11.

long long int string\_to\_lli(string &no\_str)

{

long long int no = 0;

int len = no\_str.length();

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

{

no = 10LL \* no + (no\_str[i] - '0');

}

return no;

}

/\*

This function copies digit at position i to position len - 1 - i for i in range 0 to l. If

876555123 and l = 2, r = 6 then this function will return 876555678.

\*/

long long int copy\_left\_to\_right(int len, string &n\_str, int l, int r)

{

while (l >= 0)

{

n\_str[r] = n\_str[l];

l--;

r++;

}

return string\_to\_lli(n\_str);

}

/\*

This function adds 1 to lth position and propagates carry if needed. It also copies lth

position to rth position i.e if 190 then this function will return 202.

\*/

long long int add\_to\_mid\_and\_copy\_left\_to\_right(int len, string &n\_str, int l, int r)

{

n\_str[l]++;

while (l >= 0)

{

if (n\_str[l] >= char (10 + '0'))

{

n\_str[l - 1]++;

n\_str[l] = (n\_str[l] - '0') % 10 + '0';

}

n\_str[r] = n\_str[l];

l--;

r++;

}

return string\_to\_lli(n\_str);

}

// Function to find next smallest palindrome of n\_str.

long long int solve(int len, string &n\_str, int l, int r)

{

int l\_copy = l, r\_copy = r;

/\*

Start from middle and try to find first mismatch.

n\_str = "87655078", l = 3, r = 4.

First compare n\_str[3] and n\_str[4], then n\_str[2] and n\_str[5], then n\_str[1] and n\_str[6]...

We will stop when we will find the mismatch i.e. n\_str[l] != n\_str[r].

\*/

while (l >= 0 && n\_str[l] == n\_str[r])

{

l--;

r++;

}

// Given no is already palindrome i.e. 12321.

if (l < 0)

{

return add\_to\_mid\_and\_copy\_left\_to\_right(len, n\_str, l\_copy, r\_copy);

}

// No is of type 8(7)6(5)4 here 8 > 5 hence next smallest palindrome will be 87678.

if (n\_str[l] > n\_str[r])

{

return copy\_left\_to\_right(len, n\_str, l\_copy, r\_copy);

}

// No is of type 8(7)6(8)9 here 7 < 8 hence next smallest palindrome will be 87778.

else

{

return add\_to\_mid\_and\_copy\_left\_to\_right(len, n\_str, l\_copy, r\_copy);

}

}

// Function to check if all digits are 9 or not.

bool all\_nine(string &n\_str)

{

int len = n\_str.length();

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

{

if (n\_str[i] != '9')

{

return false;

}

}

return true;

}

// Function to convert int to string.

// Alternatively you can use the built in function "to\_string" introduced by C++11.

string int\_to\_string(int no)

{

if (no == 0)

{

return "0";

}

string no\_str = "";

while (no)

{

no\_str = (char) (no % 10 + '0') + no\_str;

no /= 10;

}

return no\_str;

}

// Function to find next smallest palindrome of n.

long long int next\_palindrome(int n)

{

string n\_str = int\_to\_string(n);

int len = n\_str.length();

// Only possible case when no of digits will be increased.

if (all\_nine(n\_str))

{

long long int ret = 1LL;

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

{

ret \*= 10LL;

}

ret++;

return ret;

}

// 1-indexed. When n = 1234321 then l = 4, r = 4 and when n = 12344321 then l = 4, r = 5.

int l, r;

if (len % 2)

{

l = (len + 1) / 2;

r = (len + 1) / 2;

}

else

{

l = len / 2;

r = len / 2 + 1;

}

// It will be easy if l and r are 0 indexed instead of 1 indexed.

return solve(len, n\_str, l - 1, r - 1);

}

// -------------------------- STOP ---------------------------

int main()

{

//freopen("..//test\_cases//sample\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//sample\_test\_cases\_expected\_output.txt", "w", stdout);

freopen("..//test\_cases//handmade\_test\_cases\_input.txt", "r", stdin);

freopen("..//test\_cases//handmade\_test\_cases\_expected\_output.txt", "w", stdout);

//freopen("..//test\_cases//generated\_small\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//generated\_small\_test\_cases\_expected\_output.txt", "w", stdout);

//freopen("..//test\_cases//generated\_big\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//generated\_big\_test\_cases\_expected\_output.txt", "w", stdout);

//freopen("..//test\_cases//ignore.txt", "w", stdout);

int test\_cases;

cin >> test\_cases;

while (test\_cases--)

{

long long int n;

cin >> n;

assert(0 <= n);

assert(n <= INT\_MAX);

long long int ans = next\_palindrome(n);

cout << ans << endl;

}

return 0;

}

#include<bits/stdc++.h>

using namespace std;

// -------------------------- START --------------------------

// Function to convert long long int to string.

// Alternatively you can use the built in function "to\_string" introduced by C++11.

string lli\_to\_string(long long int no)

{

if (no == 0LL)

{

return "0";

}

string no\_str = "";

while (no)

{

no\_str = (char) (no % 10LL + '0') + no\_str;

no /= 10LL;

}

return no\_str;

}

// Function to convert string to long long int.

// Alternatively you can use the built in function "stoll" introduced by C++11.

long long int string\_to\_lli(string no\_str)

{

long long int no = 0LL;

int len = no\_str.length();

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

{

no = 10LL \* no + (no\_str[i] - '0');

}

return no;

}

long long int next\_palindrome(int n)

{

string n\_str = lli\_to\_string(n);

int len = n\_str.length();

/\*

Will be used to divide string into left half and right half.

n = 123456 -> left half= "123" -> right half = "456",

n = 1234567 -> left half = "1234" -> right half = "567".

\*/

int offset = len % 2;

/\*

n = 999 -> all\_nine\_case = 1001,

n = 125 -> all\_nine\_case = 1001.

Suppose n = 2147447411 -> all\_nine\_case = 10000000001, which can not be stored in int, so use

long long int to avoid overflow.

\*/

long long int all\_nine\_case = pow(10, len) + 1LL;

/\*

n = 123456 -> left\_half\_str = "123",

n = 1234567 -> left\_half\_str = "1234".

\*/

string left\_half\_str = n\_str.substr(0, len / 2 + offset);

/\*

n = 123456 -> left\_half\_str = "123" -> left\_half\_plus\_one\_str = "124",

n = 1234567 -> left\_half\_str = "1234" -> left\_half\_plus\_one\_str = "1235".

\*/

string left\_half\_plus\_one\_str = lli\_to\_string(string\_to\_lli(left\_half\_str) + 1LL);

/\*

n = 123456 -> left\_half\_str = "123" -> mirror\_without\_addition\_case = 123321,

n = 1234567 -> left\_half\_str = "1234" -> mirror\_without\_addition\_case = 1234321.

\*/

long long int mirror\_without\_addition\_case = string\_to\_lli(

left\_half\_str +

string(

left\_half\_str.rbegin() +

offset, left\_half\_str.rend()

)

);

/\*

n = 123456 -> left\_half\_str = "123" -> left\_half\_plus\_one\_str = "124" ->

mirror\_with\_addition\_case = 124421,

n = 1234567 -> left\_half\_str = "1234" -> left\_half\_plus\_one\_str = "1235" ->

mirror\_with\_addition\_case = 1235321.

Suppose n = 2147447411 -> mirror\_with\_addition\_case = 2147557412, which can not be stored in

int, so use long long int to avoid overflow.

\*/

long long int mirror\_with\_addition\_case = string\_to\_lli(

left\_half\_plus\_one\_str +

string(

left\_half\_plus\_one\_str.rbegin() +

offset,

left\_half\_plus\_one\_str.rend()

)

);

/\*

In these cases mirror\_without\_addition\_case will be smaller than n, so it can not be the

answer.

n = 123456 -> left\_half\_str = "123" -> mirror\_without\_addition\_case = 123321,

n = 1234567 -> left\_half\_str = "1234" -> mirror\_without\_addition\_case = 1234321.

In these cases mirror\_without\_addition\_case will be greate than n, so it is the answer.

n = 123156 -> left\_half\_str = "123" -> mirror\_without\_addition\_case = 123321,

n = 1234167 -> left\_half\_str = "1234" -> mirror\_without\_addition\_case = 1234321.

\*/

if (mirror\_without\_addition\_case - n > 0LL)

{

return mirror\_without\_addition\_case;

}

// When mirror\_without\_addition\_case - n <= 0.

return min(all\_nine\_case, mirror\_with\_addition\_case);

}

// -------------------------- STOP ---------------------------

int main()

{

//freopen("..//test\_cases//sample\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//sample\_test\_cases\_expected\_output.txt", "w", stdout);

freopen("..//test\_cases//handmade\_test\_cases\_input.txt", "r", stdin);

freopen("..//test\_cases//handmade\_test\_cases\_expected\_output.txt", "w", stdout);

//freopen("..//test\_cases//generated\_small\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//generated\_small\_test\_cases\_expected\_output.txt", "w", stdout);

//freopen("..//test\_cases//generated\_big\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//generated\_big\_test\_cases\_expected\_output.txt", "w", stdout);

//freopen("..//test\_cases//ignore.txt", "w", stdout);

int test\_cases;

cin >> test\_cases;

while (test\_cases--)

{

long long int n;

cin >> n;

assert(0 <= n);

assert(n <= INT\_MAX);

long long int ans = next\_palindrome(n);

cout << ans << endl;

}

return 0;

}

# Alternating Positives and Negatives

Problem Statement:

Given an array named array of size n, that contains both positive and negative numbers. Rearrange the array elements so that positive and negative numbers appear alternatively in the output. The order in which the positive elements appear should be maintained. Similarly, the order in which the negative elements appear should also be maintained.

Number of positive and negative integers may not be equal and extra positives or negatives have to appear in the end of the array.

Input Format:

There is only one argument in input, denoting integer array named array.

Output Format:

Return an integer array with alternate positive and negative numbers with order maintained.

Constraints:

* 1 <= n <= 500000
* -2 \* 10^9 <= array[i] <= 2 \* 10^9
* Consider 0 as a positive integer for this particular question.
* Start the array with the positive integer unless all the integers in the input array are negative.

Sample Test Case:

Sample Input:

array = [2 3 -4 -9 -1 -7 1 -5 -6]

Sample Output:

[2 -4 3 -9 1 -1 -7 -5 -6]

Explanation:

Order of positive integers in the input array is [2 3 1] which is similar to order of positive integers in the output array.

Order of negative integers in the input array is [-4 -9 -1 -7 -5 -6] which is similar to order of negative integers in the output array.

The output array starts with a positive integer and keeps alternating with negative integers until all positive integers are exhausted. Rest of the output array is filled with leftover negative integers.

We have used two approaches to solve this problem.

1) optimal\_solution.cpp

Description:

We keep two pointers, one pointing to all positive numbers in the original array and another pointing to all negative numbers in the original array.

Let us call them positive\_pointer and negative\_pointer respectively.

Then similar to merge sort, we will merge them the numbers pointed by these two pointers in a new array keeping positive and negative numbers alternatively placed to each other in the final array.

Example:

Let the input array be:

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

We initialise both positive\_pointer and negative\_pointer to 0 (index at the start of the array).

We will run a loop to fill each index of the output array.

We move the pointers forward unless we find the next positive element for the positive\_pointer and next negative element for the negative\_pointer.

For our example, the first positive integer is 5 which is at index 0 and the first negative number is -3 which is at index 4.

Now, at first iteration, since the index of iteration in the loop is 0, and we are starting with the positive element first, we will place the next positive element there which is 5 and move the positive\_pointer ahead by 1 place and go to next iteration.

Now we repeat same by finding the next positive and negative elements and continue the process ahead.

Once the value of positive\_pointer or negative\_pointer becomes 10 (size of array), it means that no more positive or negative elements respectively are left. We will keep adding other elements available.

Time Complexity:

O(n)

positive\_pointer and negative\_pointer iterate over each element in the array once. O(n) + O(n) -> O(n)

Auxiliary Space Used:

O(n)

As we are using an array to store all the resultant elements.

Space Complexity:

O(n)

As input is O(n) and auxiliary space used is O(n). O(n) + O(n) -> O(n)

2) other\_solution.cpp

Description:

In this solution, wrong\_index points to the element that shouldn’t be there, either a positive element is at an odd index or a negative element is at an even index. If a wrong\_index is set, it is replaced with eligible current index whenever possible.

Example:

Let the input array be:

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

Initially the value of wrong\_index is -1. Then we run a loop for each element in the array.

For each index, we check if the wrong\_index is set and if it is set, we replace it with the next eligible element and push rest of the elements forward upto the index of current element and if wrong\_index is not set, we just check if the current element is at wrong index.

For index 0, wrong\_index is not set (as wrong\_index equals -1), we check if the element at index 0 is at a wrong position. Since, array[0] which is 5 is positive and expected at even index, we do nothing and simply move forward to analyse the rest of the array.

For index 1, at an odd index we have 0, which is considered positive for this problem, we mark wrong\_index as 1.

Moving forward to index 2, since wrong\_index is set (not equal to -1),  we check if the current element (at index 2) should be at wrong\_index. The current element is positive and we are looking for a negative element for wrong\_index (which is currently 1 and odd).

Now at index 3, wrong\_index is still set and array[3] is -3 which is a negative number. It is the one supposed to be at wrong index. So we right rotate the subarray from wrong\_index (which is at 1) to the current index in the loop which is at 3.

The rotated array becomes:

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

After rotation, the element next to wrong\_index would be at correct position given the fact that it should be of opposite sign already. So, the next candidate for wrong\_index is 2 steps from its current value. That value is at wrong\_index if the current index in the loop is more than 2 steps away as they all were of same sign and couldn’t replace the element at wrong\_index. If the gap is not more than 2, we just move ahead, unset wrong\_index and then check if the element at current\_index is at wrong index and keep repeating this entire procedure for all the indexes in the array.

Time Complexity:

O(n^2)

wrong\_index continuously keep moving forward linearly. For each pair of wrong\_index and right\_index, the rotation takes the time proportional to the elements between the pair of indices. n such pairs and n time to rotate each pair in worst case.

Auxiliary Space Used:

O(1)

As we are using constant space to rotate the array and store the variables and no extra space is used throughout. To solve this problem with constant auxiliary space, a minimum time complexity of O(n^2) is required.

Space Complexity:

O(n)

As input is O(n) and auxiliary space used is O(1). O(n) + O(1) -> O(n)

﻿﻿﻿﻿﻿﻿

Note:

If we are asked to write solution with O(1) auxiliary space, then we need to write solution with time complexity O(n^2). We can not do better than O(n^2) time complexity, if only constant auxiliary space is allowed.

## Solutions Provided By Us

#include<bits/stdc++.h>

using namespace std;

const int MAX\_N = 500000, MIN\_VAL = -2000000000, MAX\_VAL = 2000000000;

// -------------------------- START --------------------------

vector<int> alternating\_positives\_and\_negatives(vector<int> array)

{

int n = array.size();

vector<int> ans;

// positive\_pointer and negative\_pointer would correspond to the next positive and negative

// element's index in the output array.

int positive\_pointer = 0;

int negative\_pointer = 0;

// Each iteration would result in addition of next element in the output array.

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

{

// Find the next negative element in the input array.

while(array[negative\_pointer] >= 0 && negative\_pointer <= n-1)

{

negative\_pointer++;

}

// Find the next positive element in the input array.

while(array[positive\_pointer] < 0 && positive\_pointer <= n-1)

{

positive\_pointer++;

}

// If no more positive elements are left, push the next negative element as the 'i'th element

// in the output array.

if (positive\_pointer == n)

{

ans.push\_back(array[negative\_pointer]);

negative\_pointer++;

}

// If no more negative elements are left, push the next positive element as the 'i'th element

// in the output array.

else if (negative\_pointer == n)

{

ans.push\_back(array[positive\_pointer]);

positive\_pointer++;

}

// Both positive and negative elements are remaining to be added in output array.

else

{

// At even index in the output array, push the next positive element.

if(i % 2 == 0)

{

ans.push\_back(array[positive\_pointer]);

positive\_pointer++;

}

// At odd index in the output array, push the next negative element.

else

{

ans.push\_back(array[negative\_pointer]);

negative\_pointer++;

}

}

}

return ans;

}

// -------------------------- STOP ---------------------------

int main()

{

// freopen(

// "..//test\_cases//sample\_test\_cases\_input.txt",

// "r", stdin

// );

// freopen(

// "..//test\_cases//sample\_test\_cases\_expected\_output.txt",

// "w", stdout

// );

// freopen(

// "..//test\_cases//handmade\_test\_cases\_input.txt",

// "r", stdin

// );

// freopen(

// "..//test\_cases//handmade\_test\_cases\_expected\_output.txt",

// "w", stdout

// );

// freopen(

// "..//test\_cases//generated\_small\_test\_cases\_input.txt",

// "r", stdin

// );

// freopen(

// "..//test\_cases//generated\_small\_test\_cases\_expected\_output.txt",

// "w", stdout

// );

freopen(

"..//test\_cases//generated\_big\_test\_cases\_input.txt",

"r", stdin

);

freopen(

"..//test\_cases//generated\_big\_test\_cases\_expected\_output.txt",

"w", stdout

);

// freopen(

// "..//test\_cases//ignore.txt",

// "w", stdout

// );

int test\_cases;

cin >> test\_cases;

assert(test\_cases >= 0);

while (test\_cases--)

{

int n;

cin >> n;

assert(1 <= n);

assert(n <= MAX\_N);

vector<int> array(n);

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

{

cin >> array[i];

assert(MIN\_VAL <= array[i]);

assert(array[i] <= MAX\_VAL);

}

vector<int> ans = alternating\_positives\_and\_negatives(array);

int len = ans.size();

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

{

cout << ans[i] << endl;

}

cout << endl;

}

return 0;

}

# Is It A Rotation Of A Palindrome?

Problem Statement:

Given a string s of length N containing only lower case letters (a - z), we have to check if it is a rotation of some palindromic string or not.

Input/Output Format For The Function:

Input Format:

There is only one argument denoting string s.

Output Format:

Return one integer res. Return 1 if given string s is a rotation of some palindromic string else return 0.

Input/Output Format For The Custom Input:

Input Format:

There should be one line, containing input string s.

If s = “aab”, then input should be:

aab

Output Format:

There will be one line, containing the returned integer res.

For input s = “aab”, output will be:

1

Constraints:

* 1 <= N <= 4000
* s will only contain lower case letters (a - z).
* String may or may not be sorted.

Sample Test Case:

Sample Input:

aab

Sample Output:

1

Explanation:

Given string aab is a rotation of palindromic string aba. Right rotation on aba will give aab.

Notes:

Expected solution is O(N ^ 2). That will be very easy to write in an interview.

Maximum time allowed in interview: 20 Minutes.

Python: If getting run time error then try to use iterative version to check if string is a palindrome or not.

For any string of length N, number of different string obtained by rotating it will be <= N (in some cases it can be < N, due to duplicates). Initially we might think that there are many ways to apply rotations like (left rotation -> left rotation -> left rotation -> right rotation...). But left rotation and right rotation cancel out each other. Also applying left shift x times is same as applying left shift by x % N times and applying right shift x times is same as applying right shift by x % N times. For better understanding try some examples.

If any string x is rotations of some palindromic string p, then at least one rotation of x will be p. So we can try all N rotations of given string and check if resulting string is palindrome or not.

Finding if a given string is palindrome or not is O(N). So, overall time complexity will be O(N) \* O(N) = O(N ^ 2).

Space complexity of the programme will be O(N) because space complexity includes the input space also. Auxiliary space used by programme will depend on our implementation. It can be O(N) or (1). Solution provided by us is using O(1) auxiliary space.

Just for information:

More efficient solution like O(N logN) possible using suffix array and LCP array.

Link for suffix array:

http://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/

Link for LCP array:

http://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/

Also more efficient solution like O(N) possible using longest palindromic substring.

Link for longest palindromic substring:

https://en.wikipedia.org/wiki/Longest\_palindromic\_substring

## Solutions Provided By Us

// Author: Dipen Dadhaniya

// Optimal Solution

#include<bits/stdc++.h>

using namespace std;

const int MAX\_N = 4000;

// ---------------------------------- START ------------------------------------

/\*

Problem is very easy. Here is the link for one solution.

http://www.geeksforgeeks.org/check-given-string-rotation-palindrome/

Still I will like to present other solution with same time complexity O(N^2) but more efficient.

In the above solution substring and string concatination is used, but we can build our solution

using just two pointers!

First we will see when N is odd. Consider s = cdcbaab. Here observe that if we consider d as

middle element then new string will look like abcdcba. So we only need to check if for atleast one

position after considering it as middle element we can get palindromic string or not.

When N is even we do the same thing but now there will be two middle elements. Consider s =

cddcbaab then we need to consider each two adj chars and check for palindromic string. If we

consider dd as middle elements then new string will look like abcddcba.

We are not using substring and concatination hence it will be more effective than the previous

solition.

\*/

/\*

Consider N = 7. When idx = 5 then after decrement idx = 4, when idx = 0 then after decrement

idx = 6.

\*/

int decrement\_index(int idx, int N)

{

return (idx - 1 + N) % N;

}

/\*

Consider N = 7. When idx = 5 then after increment idx = 6, when idx = 6 then after increment

idx = 0.

\*/

int increment\_index(int idx, int N)

{

return (idx + 1) % N;

}

int check\_if\_rotated(string s)

{

int N = s.length();

/\*

When we start from the middle element/elements checking for palindrome, how many pairs we need

to check. Take example of both odd and even and will get it.

\*/

int steps = (N + 1) / 2;

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

{

int l = i;

int r = i;

if (N % 2 == 0)

{

// N is even then two elements!

r = increment\_index(r, N);

}

bool valid = true;

for (int j = 0; j < steps && valid; j++)

{

if (s[l] != s[r])

{

valid = false;

}

else

{

l = decrement\_index(l, N);

r = increment\_index(r, N);

}

}

if (valid == true)

{

return 1;

}

}

return 0;

}

// ---------------------------------- END ------------------------------------

int main()

{

//freopen("..//test\_cases//sample\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//sample\_test\_cases\_output.txt", "w", stdout);

freopen("..//test\_cases//handmade\_test\_cases\_input.txt", "r", stdin);

freopen("..//test\_cases//handmade\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//small\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//small\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//big\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//big\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//ignore.txt", "w", stdout);

int test\_cases;

cin >> test\_cases;

while (test\_cases--)

{

string s;

cin >> s;

int N = s.length();

assert(1 <= N);

assert(N <= MAX\_N);

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

{

assert('a' <= s[i]);

assert(s[i] <= 'z');

}

int ans = check\_if\_rotated(s);

assert(0 <= ans);

assert(ans <= 1);

cout << ans << endl;

}

return 0;

}

# Detect Prime Numbers

Problem Statement:

Given an integer array a of size N. For each integer in a we have to check if it is a prime number or not.

Input/Output Format For The Function:

Input Format:

There is only one argument denoting integer array a.

Output Format:

Return a string res of size N, where ith character of string contains '1' if a[i] is a prime number else it should contain '0'. (1 is neither a prime nor a composite number, hence according to the previous statement, it should go in else part, i.e. ith character should contain '0', when a[i] = 1.)

Input/Output Format For The Custom Input:

Input Format:

The first line should contain a number N, denoting the number of elements in the array a. In next N lines, ith line should contain a number a[i], denoting ith element in a.

If N = 4 and a = [6, 2, 5, 8], then input should be:

4

6

2

5

8

Output Format:

There will be one line, containing a resultant string res.

For input N = 4 and a = [6, 2, 5, 8], output will be:

0110

Constraints:

* 1 <= a[i] <= 4 \* 10^6
* 1 <= N <= 3 \* 10^5

Sample Test Case:

Sample Input:

[6, 2, 5, 8]

Sample Output:

0110

Explanation:

6 is not a prime number. (6 = 2 \* 3)

2 is a prime number.

5 is a prime number.

8 is not a prime number. (8 = 2 \* 4)

Notes:

Maximum time allowed in interview: 20 Minutes.

For any number a[i], we iterate over 2 to a[i] - 1 and check if any of them divides a[i] or not. If we find atleast one number that divides a[i] then a[i] is not a prime number. So, this is a brute force solution with time complexity O(N \* MAX\_A). Have a look at the "brute\_force\_solution.cpp" provided by us.

Any positive number x, which is non-prime (and not 1), can be written as x = a \* b where a > 1 and b > 1. Here both a and b can not be > root(x), because if a > root(x) and b > root(x) then, a \* b > root(x) \* root(x), hence a \* b > x, which contradicts a \* b = x. When number is a square like 16 then it can be written as root(16) \* root(16). So, non-prime number x can be writen as a \* b having at least one of them <= root(x). Now in previous solution for each a[i], loop from root(a[i]) + 1 to a[i] - 1 is not necessary. So, now our solution will become O(N \* root(MAX\_A)). Have a look at the "sub\_optimal\_solution.cpp" provided by us.

Still we can improve. We can write any composite number as multiplication of prime numbers. Like 60 = 2 \* 2 \* 3 \* 5. So, when we find any prime number, we can visit all the multiple of it and mark them as visited (non-prime). We start from 2 as base case, consider 2 as prime number and mark all its multiples 4, 6, 8, 10, ... as visited (non-prime) because they are multiples of 2. After considering 2, we will consider 3. Now, 3 is not marked as visited, which means, there are no factors of 3 and it is a prime number. Hence for 3 also, we do the same thing, mark all its multiples 6, 9, 12, 15 ... as visited (non-prime), because they are multiple of 3. Now 4 comes, but 4 is marked as visited by 2, so we know that 4 is not a prime number. Here also we can do the same thing mark all its multiples 8, 12, 16 ... as visited, but all those positions are already marked as visited by 2, because all multiples of 4 are also multiples of 2. So, no need to do rework! Now, we keep on following the same steps for next numbers. Try to find prime numbers till 30 and you will get more clear idea.

Still there can be some optimizations. When we find any prime number like 11, we mark 22, 33. 44, 55, 66 ... them as non-prime, but 22, 33, 44, 55 are already visited by smaller prime numbers. Like 22 is visited by 2, 33 is visited by 3, 44 is visited by 4 ... 110 is visited by 10. So, instead of start marking multiples as non-prime from x + x, we can start from x \* x.

This solution will be O(N \* log(log MAX\_A)). Have a look at the "optimal\_solution.cpp" provided by us. This algoritham is called as "Sieve of Eratosthenes".

Can we say that solution with time complexity O(N \* log(log MAX\_A)) is better than solution with time complexity O(N \* root(MAX\_A))? NO! It depends on situation. In terms of time complexity of course it is better, but we should also consider space! Solution with time complexity O(N \* root(A)) requires O(1) extra space, but other needs O(MAX\_A) extra space. So, when space is more imp than time, we should opt the slower one!

When we are given a single integer and we need to find if it is a prime or not, we should use Sieve of Eratosthenes? No!! If there is only one number to check then, we can do it in O(root(A)) by iterating over 2 to root(A). We should use Sieve of Eratosthenes only when, numbers are given in some range where pre-processing can work. Think!

In this problem we are given the range of a[i]'s. But when given a stream of random numbers and nothing is specified about range of a[i] then we can use caching (though caching will be useful only when some numbers are going to repeat, but in real life it is likely to happen). We maintain two hash-tables. One containing prime numbers encountered till now and other containing non-prime numbers encountered till now. For next number, we check the presence in our hash tables and if it is present in one of them then nothing much to do, but if it is not present then we use O(root(number)) method to check if it is prime or non-prime and add into appropriate hash table for future reference.

Here also we can improve, there is no need to store even numbers as non-prime. We can handle the case of even number separately. Doing this will help to save some space.

## Solutions Provided By Us

#include<bits/stdc++.h>

using namespace std;

// ----------------------------- START -------------------------------

const int MAX\_N = 300000, MAX\_A = 4000000;

bool is\_prime[MAX\_A + 1];

void pre\_process(int N, int max\_value)

{

fill\_n(&is\_prime[0], max\_value + 1, true);

// IMP. 1 is not a prime no.

is\_prime[1] = false;

// i <= max\_value is also correct but this is more efficient.

for (int i = 2; i \* i <= max\_value; i++)

{

/\*

If not a prime no, then its multiples would have been already visited by

its prime factors previously. e.g. for i = 4, its multiples would have

been already visited by its prime factor 2.

\*/

if (!is\_prime[i])

{

continue;

}

/\*

In most of the implementations people start from j = i + i, but

this will be just waste of time. Think when i = 5 now we can visit

like 10, 15, 20, 25, 30, 35... but here note that 10 = 2 \* 5 so

when i = 2 we have already marked it, same for 15 = 3 \* 5 so when

i = 3 we have already marked it! So it is same as starting from

i \* i. But directly starting from i \* i will save time!

\*/

for (int j = i \* i; j <= max\_value; j += i)

{

is\_prime[j] = false;

}

}

}

string detect\_primes(vector<int> a)

{

int N = a.size();

pre\_process(N, \*max\_element(a.begin(), a.end()));

string ans(N, '0');

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

{

if (is\_prime[a[i]])

{

ans[i] = '1';

}

}

return ans;

}

// ------------------------------ END --------------------------------

int main()

{

//freopen("..//test\_cases//sample\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//sample\_test\_cases\_output.txt", "w", stdout);

freopen("..//test\_cases//handmade\_test\_cases\_input.txt", "r", stdin);

freopen("..//test\_cases//handmade\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//small\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//small\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//big\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//big\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//ignore.txt", "w", stdout);

int test\_cases;

cin >> test\_cases;

assert(1 <= test\_cases);

while (test\_cases--)

{

int N;

cin >> N;

assert(1 <= N);

assert(N <= MAX\_N);

vector<int> a(N);

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

{

cin >> a[i];

assert(1 <= a[i]);

assert(a[i] <= MAX\_A);

}

string ans = detect\_primes(a);

assert(ans.length() == N);

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

{

assert(ans[i] == '0' || ans[i] == '1');

}

cout << ans << endl;

}

return 0;

}

Sub optimal

#include<bits/stdc++.h>

using namespace std;

// ----------------------------- START -------------------------------

const int MAX\_N = 300000, MAX\_A = 4000000;

int check\_if\_prime(int no)

{

if (no == 1)

{

return 0;

}

for (int i = 2; i \* i <= no; i++)

{

if (no % i == 0)

{

return 0;

}

}

return 1;

}

string detect\_primes(vector<int> a)

{

int N = a.size();

string ans(N, '0');

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

{

if (check\_if\_prime(a[i]))

{

ans[i] = '1';

}

else

{

ans[i] = '0';

}

}

return ans;

}

// ------------------------------ END --------------------------------

int main()

{

//freopen("..//test\_cases//sample\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//sample\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//handmade\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//handmade\_test\_cases\_output.txt", "w", stdout);

freopen("..//test\_cases//small\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//small\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//big\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//big\_test\_cases\_output.txt", "w", stdout);

freopen("..//test\_cases//ignore.txt", "w", stdout);

int test\_cases;

cin >> test\_cases;

assert(1 <= test\_cases);

while (test\_cases--)

{

int N;

cin >> N;

assert(1 <= N);

assert(N <= MAX\_N);

vector<int> a(N);

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

{

cin >> a[i];

assert(1 <= a[i]);

assert(a[i] <= MAX\_A);

}

string ans = detect\_primes(a);

assert(ans.length() == N);

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

{

assert(ans[i] == '0' || ans[i] == '1');

}

cout << ans << endl;

}

return 0;

}

# Divide An Integer By Another Integer

Problem Statement:

Given two integers a and b, you have to find quotient q, when a is divided by b.

Input/Output Format For The Function:

Input Format:

Two integers a and b.

Output Format:

A integer q, denoting quotient of a / b.

Input/Output Format For The Custom Input:

Input Format:

The first line should contain an integer, denoting a.

Second line should contain an integer, denoting b.

If a = 5 and b = 2, then input should be:

5

2

Output Format:

There will be one line, containing a integer q, denoting quotient value of a / b.

For input a = 5 and b = 2, output will be:

2

Constraints:

* -10^18 <= a, b <= 10^18
* b != 0
* You are not allowed to use division (/) operator.
* You are not allowed to use multiplication (\*) operator.
* You are not allowed to use mod (%) operator.

Sample Test Case:

Sample Input:

a = 5, b = 2

Sample Output:

2

Notes:

Here we have mentioned explicitly that b != 0, but in interview you should clarify this with the interviewer and handle the case accordingly.

Maximum time allowed in interview: 20 Minutes.

Solution with time complexity O(a / b) and space complexity O(1) can be achieved using addition or subtraction.

We can divide our problem in four parts:

1) a >= 0 and b > 0

2) a < 0 and b > 0

3) a >= 0 and b < 0

4) a < 0 and b < 0

When a >= 0 and b > 0 then we can do something like this:

sum = 0

q = 0

while (sum + b <= a)

sum += b

q++

return q

Some modification in the above code will also work with other combinations.

But still we can improve time complexity.

Lets take a and b such that a % b = 0 so we can write q = a / b. q \* b = a. To solve the problem thinking in terms of binary representation will help.

q \* b = q

(q31q30...q0) \* b = a (in binary representation)

(2^31 \* q31 + 2^30 \* q30 + ... + 2^0 \* q0) \* b = a

To find value of each bit we can start from left side.

First we try to set q31 = 1, if 2^31 \* b <= a then we set q31 = 1, but if 2^31 \* b > a then we set q31 = 0.

When we set q31 = 0 then we have to solve

(2^30 \* q30 + ... + 2^0 \* q0) \* b = a

and when we set q31 = 1 then we have to solve

(2^30 \* q30 + ... + 2^0 \* q0) \* b = a - 2^31 \* b.

Consider 37 / 3. We keep on left shifting divisor by 1 (multiply by 2) till it does not exceed 37, here it will be 3 -> 6 -> 12 -> 24. Now we can write our division 37 / 3 = (37 / (3 \* 8)) \* 8 + (37 - (3 \* 8)) / 3. Now first part is (37 / 24) \* 8 = 1 \* 8 = 1 \* 2^(number of shifts). Second part is 13 / 3 and this problem is smaller version of the same problem we are trying to solve.

We have provided 3 solutions, and all of them uses the same idea described above, but different implementations.

1) other\_solution1.cpp: Recursive solution.

Time complexity of the function is O(log(a) ^ 2). Recursive function can be called O(log(a)) times, and in each function call we are shifting no\_of\_shifts times that is O(log(a)). (Shift operation is considered as O(1).)

Auxiliary space and space complexity is O(log(a)) due to recursive function call.

2) other\_solution2.cpp: Iterative solution.

Time complexity of the function is O(log(a) ^ 2). Recursive function can be called O(log(a)) times, and in each function call we are shifting no\_of\_shifts times that is O(log(a)). (Shift operation is considered as O(1).)

Auxiliary space and space complexity is O(1).

3) optimal\_solution.cpp: Iterative solution.

After some observations, in other\_solution2.cpp we can notice that, value of variable no\_of\_shifts decreases always. Suppose it decreases like 60 -> 55 -> 50 -> ... -> 0. Then in other\_solution2.cpp, we start no\_of\_shifts from 0 and increment it to 60, again for 55 we will increment no\_of\_shifts from 0 to 55. But as we know it will decrease from 60 to 55, we can directly start from 60 and quickly reach 55. Then from 55 to 50 (instead of 0 to 55)... After this optimization, overall we will do O(loga) shifts.

Time complexity of the function is O(log(a) + log(a)) -> O(log(a)). (Shift operation is considered as O(1).)

Auxiliary space and space complexity is O(1).

If you are using C then direct copy pasting the code provided by us (which is in C++) will not work. In C abs(x) and fabs(x) are different. Use fabs instead of abs.

## Solutions Provided By Us

#include<bits/stdc++.h>

using namespace std;

const long long int MAX\_AB = 1000000000000000000LL;

//-------------------------START------------------------------

long long int divide(long long int a, long long int b)

{

// 0 / non\_zero = 0

if (a == 0)

{

return 0;

}

// neg / pos or pos / neg

if ((a < 0 && b > 0) || (a > 0 && b < 0))

{

return -divide(abs(a), abs(b));

}

// neg / neg

if (a < 0 && b < 0)

{

a = -a;

b = -b;

}

long long int ans = 0;

int no\_of\_shifts = 0;

while ((b << (no\_of\_shifts + 1)) <= a)

{

no\_of\_shifts++;

}

while(a >= b)

{

while ((b << no\_of\_shifts) > a)

{

no\_of\_shifts--;

}

ans += (1LL << no\_of\_shifts);

a -= (b << no\_of\_shifts);

}

return ans;

}

//-------------------------STOP------------------------------

int main()

{

//freopen("..//test\_cases//sample\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//sample\_test\_cases\_output.txt", "w", stdout);

freopen("..//test\_cases//handmade\_test\_cases\_input.txt", "r", stdin);

freopen("..//test\_cases//handmade\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//small\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//small\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//big\_test\_cases\_input.txt", "r", stdin);

//freopen("..//test\_cases//big\_test\_cases\_output.txt", "w", stdout);

//freopen("..//test\_cases//ignore.txt", "w", stdout);

int test\_cases;

cin >> test\_cases;

while (test\_cases--)

{

long long int a, b;

cin >> a >> b;

assert(-MAX\_AB <= a);

assert(a <= MAX\_AB);

assert(-MAX\_AB <= b);

assert(b <= MAX\_AB);

assert(b != 0);

long long int ans = divide(a, b);

assert(ans == a / b);

cout << ans << endl;

}

return 0;

}