1.Medium

Alice is playing an arcade game and wants to climb to the top of the leaderboard and wants to track her ranking. The game uses Dense

Ranking, so its leaderboard works like this:

The player with the highest score is ranked 1 number on the leaderboard.

Players who have equal scores receive the same ranking number, and the next player(s) receive the immediately following ranking number.

Complete the climbingLeaderboard function in the editor below. It should return an integer array where each element res[j] represents

Alice's rank after the jth game.

climbingLeaderboard has the following parameter(s):

scores: an array of integers that represent leaderboard scores

alice: an array of integers that represent Alice's scores

CODE:-

#!/bin/python3

import math

import os

import random

import re

import sys

from collections import OrderedDict

def binarySearch (arr, l, r, x):

if arr[l] < x:

return l

if arr[r] > x:

return r+1

# Check base case

if r >= l:

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

# If element is present at the middle itself

if arr[mid] == x:

return mid

# If element is smaller than mid, then it can only

# be present in left subarray

elif arr[mid] < x:

return binarySearch(arr, l, mid-1, x)

# Else the element can only be present in right subarray

else:

return binarySearch(arr, mid+1, r, x)

else:

# Element is not present in the array

return

# Complete the climbingLeaderboard function below.

def climbingLeaderboard(scores, alice):

#score1=[]

#[score1.append(x) for x in scores if x not in score1]

score1 = list(OrderedDict.fromkeys(scores))

res=list()

'''

for i in alice:

flag=0

for j in range(len(score1)):

if(i>=score1[j]):

res.append(j+1)

flag=1

break

if(flag==0):

res.append(j+2)

'''

for i in alice:

if(i<score1[-1]):

res.append(len(score1)+1)

continue

elif(i>score1[0]):

res.append(1)

continue

else:

mid=binarySearch(score1,0,len(score1)-1,i)

res.append(mid+1)

return res

if \_\_name\_\_ == '\_\_main\_\_':

fptr = open(os.environ['OUTPUT\_PATH'], 'w')

scores\_count = int(input())

scores = list(map(int, input().rstrip().split()))

alice\_count = int(input())

alice = list(map(int, input().rstrip().split()))

result = climbingLeaderboard(scores, alice)

fptr.write('\n'.join(map(str, result)))

fptr.write('\n')

fptr.close()

2. (Medium)

Calculate and print the factorial of a given large integer.

Function Description:

Complete the extraLongFactorials function in the editor below. It should print the result and return.

extraLongFactorials has the following parameter(s):

n: an integer

CODE:-

#include <bits/stdc++.h>

#include <vector>

using namespace std;

// Complete the extraLongFactorials function below.

void extraLongFactorials(int n) {

vector <int> answer ;

answer.push\_back(1) ;

vector <int> :: iterator itr ;

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

for(itr = answer.begin() ; itr != answer.end() ; itr++)

\*itr \*= i ;

for(int j = 0 ; j < answer.size() ; j++)

if(answer[j] >= 10){

if(j == answer.size() - 1)

answer.push\_back(0) ;

answer[j+1] = answer[j+1] + (answer[j]/10) ;

answer[j] = answer[j]%10 ;

}

}

for(int i = answer.size() - 1 ; i > -1 ; i--)

cout << answer[i] ;

}

int main()

{

int n;

cin >> n;

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

extraLongFactorials(n);

return 0;

}

3. (Hard)

Given the following:

• An array of beneficial gene strings, genes=[g0,g1,…,gn-1]. Note that these gene sequences are not guaranteed to be distinct.

• An array of gene health values, health=[h0,h1,…,hn-1]. where each hi is the health value for gene gi.

• A set of s DNA strands where the definition of each strand has three components,start ,end , and d, where string d is a DNA

for which genes gstart….gend are healthy.

Find and print the respective total healths of the unhealthiest (minimum total health) and healthiest (maximum total health)

strands of DNA as two space-separated values on a single line.

Input Format:-

The first line contains an integer,n , denoting the total number of genes.

The second line contains n space-separated strings describing the respective values of g0,g1,…,gn-1 (i.e., the elements of genes).

The third line contains n space-separated integers describing the respective values of h0,h1,…,hn-1(i.e., the elements of health).

The fourth line contains an integer, , denoting the number of strands of DNA to process.

Each of the s subsequent lines describes a DNA strand in the form start end d, denoting that the healthy genes for DNA strand d

are gstart,…,gend and their respective correlated health values are hstart,…,hend.

Output Format:-

Print two space-separated integers describing the respective total health of the unhealthiest and the healthiest strands of DNA.

CODE:-

#!/bin/python3

import math

import os

import random

import re

import sys

import re

from collections import Counter

def count\_overlapping(text, search\_for):

return len(re.findall(search\_for, text))

'''def solve(s,sub\_st):

total=Counter([s[i : i+len(sub\_st)] for i in range(len(s))])

return total[sub\_st]

'''

def occurrences(string, sub):

count = start = 0

while True:

start = string.find(sub, start) + 1

if start > 0:

count += 1

else:

return count

if \_\_name\_\_ == '\_\_main\_\_':

n = int(input())

genes = input().rstrip().split()

health = list(map(int, input().rstrip().split()))

s = int(input())

res=list()

for s\_itr in range(s):

firstLastd = input().split()

first = int(firstLastd[0])

last = int(firstLastd[1])

d = firstLastd[2]

big\_string = d

s1=0

for i in range(first,last+1):

substring = genes[i]

count = 0

#count=count\_overlapping(big\_string,substring)

#count=solve(big\_string,substring)

count=occurrences(big\_string,substring)

#for char in range(len(big\_string)):

# count += big\_string[char: char + len(substring)] == substring

s1+=count\*health[i]

res.append(s1)

print(min(res),max(res))