**FUNCION BINARYSEACRH**

int binarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return binarySearch (A, m+1, j, k);

else

return binarySearch (A, i, m-1, k);

}

}

}

**FUNCION BINARYSEARCH QUE RETORNA LA ULTIMA POSICION DONDE ENCUENTRA EL ELEMENTO**

nt binarySearchLastOccurrence(int A[], int i, int j, int k) {

int result, result2;

result = binarySearch(A, i, j, k);

if (result >= 0) {

result2 = binarySearch(A, result + 1, j, k);

while (result2 >= 0) {

result = result2;

result2 = binarySearch(A, result + 1, j, k);

}

}

return result;

}

**FUNCION BINARYSEARCH QUE RETORNA LA PRIMERA POSICION DONDE ENCUENTRA EL ELEMENTO**

t binarySearchFirstOccurrence(int A[], int i, int j, int k) {

int result, result2;

result = binarySearch(A, i, j, k);

if (result >= 0) {

result2 = binarySearch(A, i, result - 1, k);

while (result2 >= 0) {

result = result2;

result2 = binarySearch(A, i, result - 1, k);

}

}

return result;

}

**FUNCION MERGESORT**

void myMerge (int A[], int p, int q, int r) {

int n1 = q-p+1, n2 = r-q, i, j, k;

int L[n1+2], R[n2+2];

for (i=1; i<=n1; i++)

L[i] = A[p+i-1];

for (j=1; j<=n2; j++)

R[j] = A[q+j];

L [n1+1] = myInfinite;

R [n2+1] = myInfinite;

i=1;

j=1;

for (k=p; k<=r; k++) {

if (L[i] <= R[j]) {

A[k] = L [i];

i++;

}

else {

A[k] = R[j];

j++;

}

}

}

void MergeSort (int A[], int p, int r) {

int q;

if (p<r) {

q = (p+r)/2;

MergeSort (A, p, q);

MergeSort (A, q+1, r);

myMerge (A, p, q, r);

}

}

**FUNCION BINARYSEARCH PARA CADENAS DE TEXTO (A PARTIR DE UNA ESTRUCTURA, SE CAMBIA “STREET” POR CHAR)**

int BinarySearch(street A[],int i,int j,char k[]){  
    int m;  
    if(i > j){  
        return -1 \* i - 1;  
    }  
    else{  
        m = (i + j) / 2;  
        if(strcmp(A[m].name , k) == 0){  
            return m;  
        }  
        else  
        {  
            if(strcmp(A[m].name, k) < 0)  
                return BinarySearch(A,m + 1, j, k);  
             else  
                return BinarySearch(A,i,m - 1,k);  
        }  
    }  
}

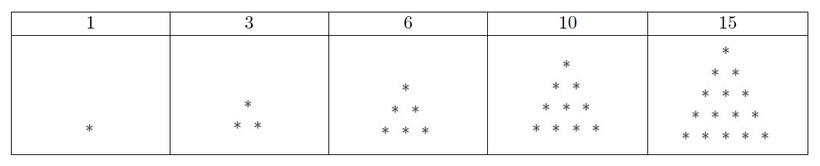
**FUNCION MERGESORT PARA CADENAS DE TEXTO (A PARTIR DE UNA ESTRUCTURA, SE CAMBIA “STREET” POR CHAR)**

void myMerge(street A[], int p, int q, int r)  
{  
    int i, j, k, n1 = q - p + 1, n2 = r - q;  
    street L[n1 + 2], R[n2 + 2];  
     
    for(i = 1; i <= n1; i++)  
        L[i] = A[p + i - 1];  
         
    for(j = 1; j <= n2; j++)  
        R[j] = A[q + j];  
     
    strcpy (L[n1 + 1].name, myInfinite);  
    strcpy (R[n2 + 1].name, myInfinite);  
    i = 1;  
    j = 1;  
     
    for(k = p; k <= r; k++)  
    {  
        if(strcmp(L[i].name, R[j].name) <= 0)  
        {  
            A[k] = L[i];  
            i++;  
        }  
        else  
        {  
            A[k] = R[j];  
            j++;  
        }  
    }  
  
}  
  
void MergeSort(street A[], int p, int r)  
{  
    int q;  
    if(p < r)  
    {  
        q = (p + r) / 2;  
        MergeSort(A,p,q);  
        MergeSort(A, q + 1, r);  
        myMerge(A,p,q,r);  
    }  
}

**Triangular Numbers 2**

Triangular Numbers are positive integer numbers such that they represent an amount of "dots" with which you can form a compact equilateral triangle of dots.

The first five triangular numbers are:



For this problem, you must create a program that determines if a given number  is triangular or not.

**Input Format**

Input may contain several test cases. Each test case is given in a line of its own, and contains an integer  (). Input ends with a test case in which  is zero, and it must not be processed.

**Constraints** 1 <= n <= 16\* 1018

**Output Format**

For each test case given in the input, your program must print YES or NO, indicating whether  is a triangular number or not. There must be a single line of output for each test case.

**Sample Input 0**

1

15

16

101

15999999994386249876

0

**Sample Output 0**

YES

YES

NO

NO

YES

**SOLUCION:**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

int main() {

unsigned long long int n, triangular, k, kplus1;

while (scanf ("%llu", &n) && (n>0)) {

k = (-1 + (unsigned long long int)sqrt(1+8\*(double)n)) / 2;

kplus1 = k+1;

if (k%2 == 0)

k /= 2;

else

kplus1 /= 2;

triangular = k\*kplus1;

if (triangular == n)

printf ("YES\n");

else

printf ("NO\n");

}

return 0;

}

**The Book Thief I**

On February 18, 2014, Red Matemática proposed the following mathematical challenge on their twitter account (@redmatematicant): "While Anita read: *The book thief*by Markus Zusak, She added all the page numbers starting from 1. When she finished the book, she got a sum equal to 9.000 but she realized that one page number was forgotten in the process. What is such number? and, how many pages does the book have?"

Using this interesting puzzle as our starting point, the problem you are asked to solve now is: Given a positive integer   representing the result obtained by Anita, find out the number of the forgotten page and the total number of pages in the book.

**Input Format**

The input may contain several test cases. Each test case is presented on a single line, and contains one positive integer s. The input ends with a test case in which  is zero, and this case must not be processed.

**Constraints** 1<= s <= 10^8

**Output Format**

For each test case, your program must print two positive integers, separated by a space, denoting the number of the forgotten page and the total number pages in the book. Each valid test case must generate just one output line.

**Sample Input 0**

1

2

3

4

5

6

9000

499977

49999775

0

**Sample Output 0**

2 2

1 2

3 3

2 3

1 3

4 4

45 134

523 1000

5225 10000

**SOLUCION**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

int main () {

int s, forgottenPage, totalPages, page, triangular;

while (scanf("%d", &s) && (s>0)) {

page = (-1 + sqrt(1+8 \*(double)s)) / 2;

triangular = (page \* (page+1)) /2;

if (triangular == s) {

totalPages = page+1;

forgottenPage = page+1;

}

else {

totalPages = page+1;

forgottenPage = triangular + (page + 1) -s;

}

printf ("%d %d\n", forgottenPage, totalPages);

}

return 0;

}

## How Many Inversions?

Humberto Morales in his student days, he is attending system engineering at "University of missing hill". He was evaluated in its first course of Analysis of Algorithms (at the first half of 1997) with the following topics and questions:

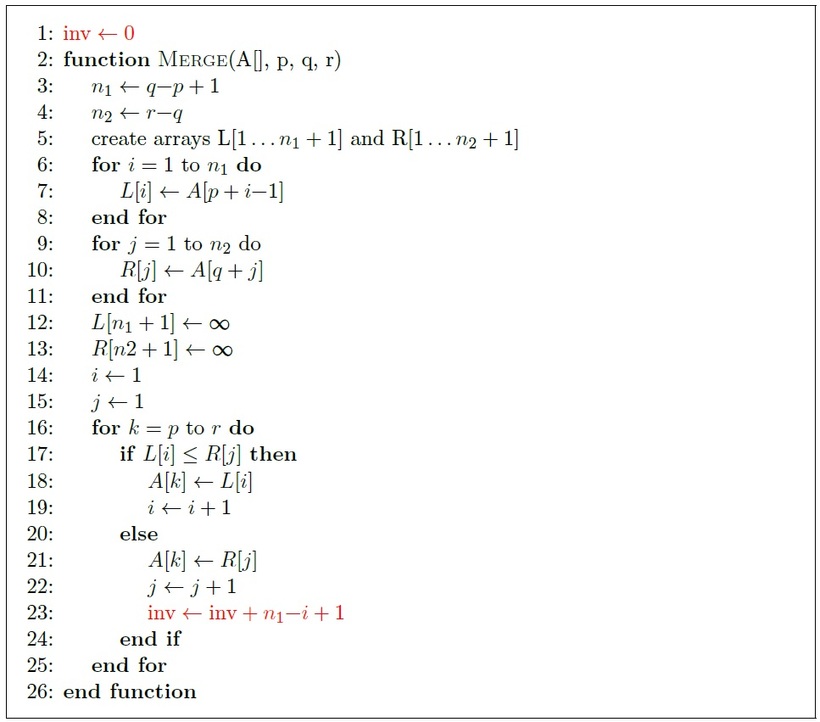
**Inversions :**

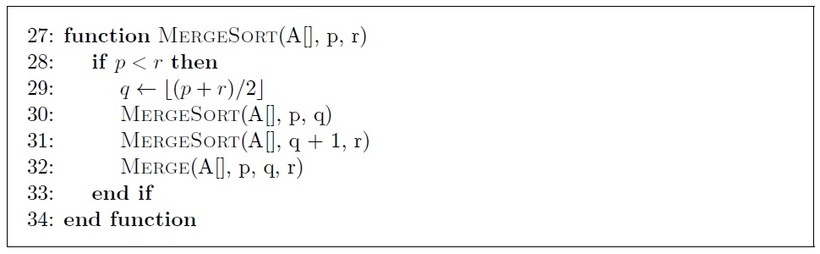
Let A [1,…, n] an array of distinct integers of size . If  and , then the pair  is called an *inversion* of .

Given the above definition about an inversion, *Humbertov Moralov* must answer the following questions:

* List all inversions in .
* What array of size , with all the numbers from the set  has the largest amount of inversions?. How many inversions?.
* Write an algorithm to determine the number of inversions in any permutation of  elements with  in the worst case run time.

*Humbertov Moralov* answered questions 1. and 2. without any problem, but he was not able to solve the question 3. at time. Days later he thought the following solution:





Will this code solve the problem?. Just adding the lines 1 and 23 will be enough to solve the problem?

Please help *Humbertov Moralov* to validate this solution!. For this, you must implement this solution in any of the programming languages accepted by the ICPC (International Collegiate Programming Contest) and verify if the expected results are generated.

**Input Format**

The input contains several test cases, each one has the following structure:

The first line contains a positive integer  , which represent the length of .

The second line contains  space-separated positive integers which make up the array , these values are in the closed interval .

The input ends with a test case in which  is zero, and this case must not be processed.

**Constraints**

**Output Format**

For each test case, your program must print a non-negative integer representing the total number of inversions in the array . Each valid test case must generate just one output line.

**Sample Input 0**

5

3 2 8 1 6

5

5 4 3 2 1

1

10

0

**Sample Output 0**

5

10

0

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define myInfinite 2147483647

#define MAXT 1000000

long long int inv = 0;

void myMerge (int A[], int p, int q, int r) {

int n1 = q-p+1, n2 = r-q, i, j, k;

int L[n1+2], R[n2+2];

for (i=1; i<=n1; i++)

L[i] = A[p+i-1];

for (j=1; j<=n2; j++)

R[j] = A[q+j];

L [n1+1] = myInfinite;

R [n2+1] = myInfinite;

i=1;

j=1;

for (k=p; k<=r; k++) {

if (L[i] <= R[j]) {

A[k] = L [i];

i++;

}

else {

A[k] = R[j];

j++;

inv = inv+n1-i+1;

}

}

}

void MergeSort (int A[], int p, int r) {

int q;

if (p<r) {

q = (p+r)/2;

MergeSort (A, p, q);

MergeSort (A, q+1, r);

myMerge (A, p, q, r);

}

}

int main() {

int n, i, A [MAXT+1];

while (scanf("%d", &n) && (n>0)) {

inv = 0;

for (i=1; i<=n; i++)

scanf ("%d ", &A[i]);

MergeSort (A, 1, n);

printf ("%lld\n", inv);

}

return 0;

}

## How Many Sub Sets?

We have a set  of positive integers with cardinality  , remember that the cardinality of a set is the total of different elements it has. You want to find out how many subsets of size two have a sum of their elements less than or equal to .

For example, if you have the following set  and , then there is a total of  subsets of size two whose sum of its elements is less than or equal to , said subsets are: , , , , , , ,  and .

Your mission, if you decide to accept it, is to count the total of subsets of size two that have a sum of their elements less than or equal to .

**Input Format**

The input of the problem consists of a single test case. The test case contains three lines, the first line contains two positive integer numbers   and  , which represent respectively the cardinality of the set  and the total of queries that will be made on the set . The next line contains exactly  space-separated positive integer numbers , , , ,  , for , it is obviously guaranteed that the  elements of the set  are different. The next line contains exactly  space-separated positive integer numbers , , , ,  , for , for the queries.

**Constraints**

**Output Format**

Your program should print  lines, each of them containing a single value that represents the total result of subsets of size two that the sum of their elements is less or equal to .

**Sample Input 0**

6 3

6 5 1 4 2 3

7 8 12

**Sample Output 0**

9

11

15

**MI SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define myInfinite 2147483647

#define MAXT 500000

void myMerge (int A[], int p, int q, int r) {

int n1 = q-p+1, n2 = r-q, i, j, k;

int L[n1+2], R[n2+2];

for (i=1; i<=n1; i++)

L[i] = A[p+i-1];

for (j=1; j<=n2; j++)

R[j] = A[q+j];

L [n1+1] = myInfinite;

R [n2+1] = myInfinite;

i=1;

j=1;

for (k=p; k<=r; k++) {

if (L[i] <= R[j]) {

A[k] = L [i];

i++;

}

else {

A[k] = R[j];

j++;

}

}

}

void MergeSort (int A[], int p, int r) {

int q;

if (p<r) {

q = (p+r)/2;

MergeSort (A, p, q);

MergeSort (A, q+1, r);

myMerge (A, p, q, r);

}

}

long long int TotalSubSets (int A[], int s, int n) {

int i=1, j=n;

long long int x = 0;

while (i<j) {

if (A[i]+A[j]<=s) {

x += j-i;

i++;

}

else

j--;

}

return x;

}

int main() {

int n, q, i, A[MAXT], S[50];

long long int x = 0;

scanf ("%d %d", &n, &q);

for (i=1; i<=n; i++)

scanf ("%d ", &A[i]);

for (i=1; i<=q; i++)

scanf ("%d ", &S[i]);

MergeSort(A, 1, n);

for (i=1; i<=q; i++) {

x = TotalSubSets (A, S[i], n);

printf ("%lld\n", x);

}

return 0;

}

**SOLUCION DEL PROFE**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define myInfinite 2147483647

#define MAXN 500000

int BinarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return BinarySearch (A, m+1, j, k);

else

return BinarySearch (A, i, m-1, k);

}

}

}

void myMerge (int A[], int p, int q, int r)

{

int i, j, k, n1 = q - p + 1, n2 = r - q;

int L [n1 + 2], R[n2 + 2];

for (i = 1; i <= n1; i++)

L[i] = A[p + i - 1];

for (j = 1; j <= n2; j++)

R[j] = A[q+j];

L[n1+1] = myInfinite;

R[n2+1] = myInfinite;

i=1;

j=1;

for(k=p; k<=r; k++)

{

if (L[i] <= R[j])

{

A[k] = L [i];

i++;

}

else

{

A[k] = R [j];

j++;

}

}

}

void MergeSort (int A[], int p, int r)

{

int q;

if (p < r)

{

q = (p + r) / 2;

MergeSort(A, p, q);

MergeSort(A, q + 1, r);

myMerge(A, p, q, r);

}

}

int main()

{

int A[MAXN + 1], n, q, s, element, position;

int idQuery, idElement, i;

long long int result;

scanf("%d %d", &n, &q);

for (idElement = 1; idElement <= n; idElement++)

scanf("%d", &A[idElement]);

MergeSort(A, 1, n);

for(idQuery = 1; idQuery <= q; idQuery++)

{

scanf("%d", &s);

result = 0;

for(i = 1; i < n; i++)

{

element = s - A[i];

if (element > A[i]) {

position = BinarySearch (A, i+1, n, element);

if (position < 0)

position = -1 \* (position+2);

result += position-i;

}

else

break;

}

printf("%lld\n", result);

}

return 0;

}

## Toby y las Galletas 1

Toby tiene mucha hambre, así que fue a la nevera por algo de comer, sacó una caja de galletas de muchos tipos. La caja puede ser vista como un arreglo de  elementos, Toby sabe que si come muchas galletas su mamá se dará cuenta. Así que decidió comer solo las galletas que son de un tipo especial, aquellas que su peso son menores o iguales a  gramos. Toby tiene muchas consultas, para cada consulta se tiene que responder cuántas galletas se comerá él.

**Input Format**

La entrada contiene múltiples casos de prueba. Cada caso de prueba contiene cuatro líneas, donde, la primera línea contiene un número entero positivo  , el cual representa la cantidad de unidades (es decir, la cantidad de galletas) que contiene la caja de galletas. La segunda línea contiene  números enteros positivos separados por un espacio en blanco, los cuales representan los pesos de las  galletas de la caja, cada uno de los pesos de las galletas pertenecen al intervalo cerrado . La tercera línea contiene un número entero positivo  , el cual representa el total de consultas que tiene que contestar Toby. La cuarta y última línea del caso de prueba contiene  números enteros positivos separados por un espacio en blanco, los cuales representan los pesos consultados, cada uno de los pesos pertenecen al intervalo cerrado . Se deben leer casos de prueba hasta alcanzar el fin de archivo (EOF).

**Constraints**

**Output Format**

Para cada caso de prueba de la entrada, se debe imprimir una línea conteniendo  números enteros positivos separados por un espacio en blanco, los cuales representan el total de galletas que se puede comer Toby en cada una de las consultas evaluadas.

**Sample Input 0**

10

4 2 4 5 3 1 5 3 3 4

5

4 2 5 3 1

10

21 4 32 1 16 9 30 12 6 10

4

20 10 15 26

**Sample Output 0**

8 2 10 5 1

7 5 6 8

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define MAXT 100000

#define myInfinite 2147483647

int binarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return binarySearch (A, m+1, j, k);

else

return binarySearch (A, i, m-1, k);

}

}

}

int binarySearchLastOccurrence ( int A[] , int i, int j, int k) {

int result , result2 ;

result = binarySearch (A, i, j, k);

if( result >= 0) {

result2 = binarySearch (A, result + 1 , j, k) ;

while ( result2 >= 0) {

result = result2 ;

result2 = binarySearch (A, result + 1 , j, k) ;

}

}

return result ;

}

void myMerge (int A[], int p, int q, int r)

{

int i, j, k, n1 = q - p + 1, n2 = r - q;

int L [n1 + 2], R[n2 + 2];

for (i = 1; i <= n1; i++)

L[i] = A[p + i - 1];

for (j = 1; j <= n2; j++)

R[j] = A[q+j];

L[n1+1] = myInfinite;

R[n2+1] = myInfinite;

i=1;

j=1;

for(k=p; k<=r; k++)

{

if (L[i] <= R[j])

{

A[k] = L [i];

i++;

}

else

{

A[k] = R [j];

j++;

}

}

}

void MergeSort (int A[], int p, int r)

{

int q;

if (p < r)

{

q = (p + r) / 2;

MergeSort(A, p, q);

MergeSort(A, q + 1, r);

myMerge(A, p, q, r);

}

}

int main() {

int n, i, A [MAXT], q, consults [MAXT], result;

while (scanf ("%d", &n) != EOF) {

for (i=1; i<=n; i++)

scanf ("%d ", &A[i]);

MergeSort(A, 1, n);

scanf ("%d", &q);

for (i=1; i<=q; i++) {

result = 0;

scanf ("%d ", &consults[i]);

result = binarySearchLastOccurrence(A, 1, n, consults [i]);

if (result < 0)

result = -1 \* (result+2);

printf ("%d ", result);

}

printf ("\n");

}

return 0;

}

## Johann's Function

Typically in a first-year programming course in an engineering or computer science academic program, students are taught to build functions that make use of the doubly nested loop structure such as the following:

unsigned long long JohannsFunction(int n)

{

int i, j;

unsigned long long result = 0;

for(i = 1; i <= n; i++)

{

for(j = 1; j <= i; j++)

{

result += j;

}

}

return result;

}

The running-time function of the previous algorithm belongs to , with a value of  the total number of operations performed by the algorithm, in order to give the result would require  steps. As a competitor of the different phases of the ICPC, you know that a solution that performs that amount of steps, will obtain a verdict of **Time Limit Exceeded** in competition, for this reason you are asked to propose a solution that has a running time as close to  as possible, regardless of the size of , in which you must make use of all your experience as a competitor of ICPC!

**Input Format**

The input begins with a positive integer  , which represents the total number of test cases. Then  lines are presented, each containing a positive integer   for which the result of the **Johann's Function** must be calculated.

**Constraints**

**Output Format**

The output must contain  lines, each containing a long positive integer as a result of the **Johann's Function**.

**Sample Input 0**

7

1

10

100

1000

10000

100000

1000000

**Sample Output 0**

1

220

171700

167167000

166716670000

166671666700000

166667166667000000

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define MAXT 1000000

unsigned long long int SergiosFunction (int n) {

unsigned long long int result = 0;

result = (unsigned long long int)n\*(n+1)\*(n+2)/6;

return result;

}

int main() {

int t, i, cases[MAXT];

unsigned long long int result = 0;

scanf ("%d", &t);

for (i=1; i<=t; i++)

scanf ("%d", &cases[i]);

for (i=1; i<=t; i++) {

result = SergiosFunction (cases[i]);

printf ("%llu\n", result);

}

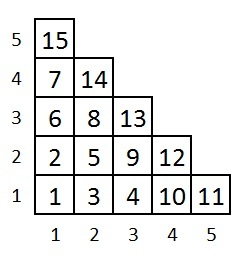
return 0;

}

## Dora the Explorer I

One day, there was this little ant called "Dora the Explorer", the ant arrived to a triangular board of d diagonals. the ant wanted to explore every boxes of the board, so she began to walking on this board starting on the diagonal containing only one box.

Dora the explorer began on the box . In the first place she took one step forward to the box , then she descended by the diagonal to reach the box , after that, the ant took one step to the right to the box , then she ascended by the respective diagonal passing trough the boxes  and . Each time the ant added a new diagonal to her trajectory either ascending or descending by the diagonal depending of the first box she reached in the respective diagonal.



For example, the ant in her first 15 steps made the following trajectory in the triangular board, where the number of each box or cell indicates the order in which it was visited:

The 10th step was in the box , while the 15th step was in the box .

The objective consists of determining "Dora's the ant" position in triangular board given an step, assuming that the triangular board is big enough to admit any step.

**Input Format**

Each line of the input contains an integer , which indicates the number of the step, where . The input ends with a line containing .

**Constraints**

**Output Format**

For each test case, print a line with two integer numbers  indicating the values of the column and the row, respectively. Between  and  there's only a white space.

**Sample Input 0**

1

6

10

11

15

16

17

0

**Sample Output 0**

1 1

1 3

4 1

5 1

1 5

1 6

2 5

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#include <stdbool.h>

bool esTriangular (unsigned long long int n) {

unsigned long long int k, triangular;

k = (-1 + (unsigned long long int)sqrt(1 + 8 \* (double)n)) / 2;

triangular = k\*(k+1)/2;

if (n==triangular)

return true;

else

return false;

}

int main() {

unsigned long long int n, x=1, y=1, k, pastTriangular;

while (scanf ("%llu", &n) && (n>0)) {

k = (-1 + (unsigned long long int)sqrt(1 + 8 \* (double)n)) / 2;

if (n==1)

x=y=1;

else if (n==2) {

x=1;

y=2;

}

else {

if ((esTriangular(n)) && (k%2==0)) {

x=k;

y=1;

}

else if ((esTriangular (n)) && (k%2==1)) {

x=1;

y=k;

}

else if (k%2==0) {

pastTriangular = k\*(k+1)/2;

x= k+1 -(n-pastTriangular-1);

y=1+n-pastTriangular-1;

}

else {

pastTriangular = k\*(k+1)/2;

y= k+1 -(n-pastTriangular-1);

x=1+n-pastTriangular-1;

}

}

printf ("%llu %llu\n", x, y);

}

return 0;

}

## Triangles 12

You will be given  points on a circle. You must write a program to determine how many distinct equilateral triangles can be constructed using the given points as vertices.

The figure below illustrates an example: (a) shows a set of points, determined by the lengths of the circular arcs that have adjacent points as extremes; and (b) shows the two triangles which can be built with these points.



**Input Format**

The input contains several test cases. The first line of a test case contains an integer , the number of points given. The second line contains  integers , representing the lengths of the circular arcs between two consecutive points in the circle: for ,  represents the length of the arc between between points  and ;  represents the length of the arc between points  and .

**Constraints**

, for

**Output Format**

For each test case your program must output a single line, containing a single integer, the number of distinct equilateral triangles that can be constructed using the given points as vertices.

**Sample Input 0**

8

4 2 4 2 2 6 2 2

8

2 2 2 2 2 4 4 6

8

6 4 4 2 2 2 2 2

8

1 1 1 1 1 1 1 1

9

1 1 1 1 1 1 1 1 1

6

3 4 2 1 5 3

**Sample Output 0**

2

1

1

0

3

1

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

int BinarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return BinarySearch (A, m+1, j, k);

else

return BinarySearch (A, i, m-1, k);

}

}

}

int procesarCasosDePrueba() {

int n, i, triangles = 0, numberLookingFor, lengthSide, numberPosition;

if (scanf("%d", &n) != 1) {

return -1;

}

int points[n], addPoints[n];

addPoints[0] = 0;

for (i = 1; i <= n; i++) {

if (scanf("%d", &points[i]) != 1) {

return -1;

}

addPoints[i] = addPoints[i - 1] + points[i];

}

if (addPoints[n] % 3 != 0) {

triangles = 0;

} else {

lengthSide = addPoints[n] / 3;

i = 1;

while (points[i] != 0) {

numberLookingFor = addPoints[i] + lengthSide;

points[i] = 0;

numberPosition = BinarySearch(addPoints, i, n, numberLookingFor);

if (numberPosition < 0) {

i++;

} else {

numberLookingFor = addPoints[numberPosition] + lengthSide;

points[numberPosition] = 0;

numberPosition = BinarySearch(addPoints, numberPosition, n, numberLookingFor);

if (numberPosition < 0) {

i++;

} else {

points[numberPosition] = 0;

triangles++;

i++;

}

}

}

}

return triangles;

}

int main() {

int triangles;

while ((triangles = procesarCasosDePrueba()) != -1) {

printf("%d\n", triangles);

}

return 0;

}

## Investigating Frog Behaviour on Lily Pad Patterns

Recently, the biologist Ina discovered a new frog species on the lily pads of a pond. She observed the frogs for a while and found them to be very conscious about their personal space because they avoided sharing a lily pad with other frogs. Also, they seemed quite lazy as they did not move often, and if they did, they always jumped to the nearest empty lily pad.

To confirm her hypotheses about the frogs' movement pattern, Ina set up a large number of lily pads in a pool in her laboratory, arranged in a straight line. Since the frogs were attracted to light, she was able to simplify the test setup further by placing a bright light at one end of that line. This way, the frogs would always jump in one direction (towards the light).

Of course, Ina could now place some frogs on the lily pads and sit there all day watching the frogs jump around. But as the frogs move so rarely, it would take ages to gather a sufficient amount of data.

She therefore attached to each frog a tiny device that could log all jumps of that frog. This way, she could put the frogs on the lily pads, leave them alone for a few hours and come back later to collect the data. Unfortunately, the devices had to be so tiny that there was no space for a position tracking system; instead, the devices could only record the times of the jumps.

But if the movement pattern of the frogs is as restricted as Ina thinks, surely the individual movements of the frog can be reconstructed only from the initial positions and the recorded jump time stamps?

**Input Format**

The input consists of:

* One line with an integer  , the number of frogs.
* One line with  integers  , the number of the lily pad on which the th frog initially sits. The lily pads are numbered consecutively, starting at . It is guaranteed that the initial positions are strictly increasing, i.e. .
* One line with an integer  , the number of jumps recorded.
* lines, each containing an integer  , indicating that the th frog jumped. The jumps are given in chronological order and you may assume that a jumping frog lands before the next jump begins. The frogs always jump to the nearest empty lily pad with a larger number, and you may assume that such a lily pad always exists.

**Constraints**

**Output Format**

For each jump, output the number of the lily pad the frog lands on.

**Sample Input 0**

5

1 2 3 5 7

3

1

2

4

**Sample Output 0**

4

6

8

**Sample Input 1**

5

1 2 3 5 7

4

1

1

1

1

**Sample Output 1**

4

6

8

9

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define MAXT 1200001

#define MAXFROG 200001

int binarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return binarySearch (A, m+1, j, k);

else

return binarySearch (A, i, m-1, k);

}

}

}

int main() {

int n, i, x[MAXFROG], q, lilyPadsPosition=1, lookForFrog, lilyPads [MAXT], frogJumping, destiny, temp;

scanf ("%d", &n);

for (i=1; i<=n; i++)

scanf ("%d", &x[i]);

for (i=1; i<=MAXT; i++) {

lookForFrog = binarySearch (x, 1, n, i);

if (lookForFrog < 0) {

lilyPads [lilyPadsPosition] = i;

lilyPadsPosition++;

}

}

scanf ("%d", &q);

for (i=1; i<=q; i++) {

frogJumping = 0;

scanf ("%d", &frogJumping);

destiny = binarySearch (lilyPads, 1, lilyPadsPosition - 1, x[frogJumping]);

destiny = -(destiny+1);

printf ("%d\n", lilyPads [destiny]);

temp = x[frogJumping];

x[frogJumping] = lilyPads [destiny];

lilyPads [destiny] = temp;

}

return 0;

}

## Dynamic Collection

Se cuenta con una colección de elementos que son números enteros positivos, sobre la cual se permiten elementos repetidos. Sobre la colección se permiten dos tipos de operaciones. La operación  para actualizar la colección y la operación  para consultar la colección. El formato de las dos operaciones es el siguiente:



Para la operación ,  es un número entero sobre el cual se realiza una de las siguientes tres acciones:

* Si el elemento  se encuentra en la colección entonces no se hace nada
* Si el elemento  es más grande que cualquiera de los elementos en la colección entonces se adiciona a la colección
* Si el elemento  no encuentra en la colección entonces se reemplaza por  la primera ocurrencia del elemento más pequeño que es más grande que

Para la operación ,  y  son dos números enteros  sobre los cuales se debe consultar en la colección cuantos elementos están en dicho rango.

Por ejemplo, sea la colección que inicialmente tiene diez elementos:

después de la operación "" el resultado generado es .

Después de la operación "" la colección queda como:

Ahora al volver a lanzar la consulta "" el resultado generado es .

Después de la operación "" la colección queda con once elementos:

**Input Format**

La entrada consiste de un único caso de prueba. La primera línea contiene dos números enteros positivos    los cuales representan respectivamente la cantidad de elementos de la colección y el total de operaciones a realizarse sobre la colección. La segunda línea contiene los  elementos de la colección separados por un único espacio en blanco, cada elemento de la colección es un número entero positivo en el rango . Las siguientes  líneas del caso de prueba son las operaciones, las cuales pueden ser del tipo  o del tipo , con el siguiente formato:  o , con  y .

**Constraints**

**Output Format**

Para cada operación del tipo  (operación de consulta) en la entrada del caso de prueba se debe imprimir una sola línea que contenga un número entero positivo  el cual representa el total de elementos en la colección que están en el rango .

**Sample Input 0**

10 11

7 1 7 1 3 9 7 9 10 4

2 2 8

1 8

2 2 8

2 1 20

1 20

2 1 20

2 7 12

1 5

2 7 12

1 12

2 7 12

**Sample Output 0**

5

6

10

11

6

5

6

**SOLUCION**

#include <math.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <assert.h>

#include <limits.h>

#include <stdbool.h>

#define MAXT 2000000

#define myInfinite 2147483647

int binarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return binarySearch (A, m+1, j, k);

else

return binarySearch (A, i, m-1, k);

}

}

}

int binarySearchFirstOccurrence(int A[], int i, int j, int k) {

int result, result2;

result = binarySearch(A, i, j, k);

if (result >= 0) {

result2 = binarySearch(A, i, result - 1, k);

while (result2 >= 0) {

result = result2;

result2 = binarySearch(A, i, result - 1, k);

}

}

return result;

}

int binarySearchLastOccurrence(int A[], int i, int j, int k) {

int result, result2;

result = binarySearch(A, i, j, k);

if (result >= 0) {

result2 = binarySearch(A, result + 1, j, k);

while (result2 >= 0) {

result = result2;

result2 = binarySearch(A, result + 1, j, k);

}

}

return result;

}

void myMerge (int A[], int p, int q, int r) {

int n1 = q-p+1, n2 = r-q, i, j, k;

int L[n1+2], R[n2+2];

for (i=1; i<=n1; i++)

L[i] = A[p+i-1];

for (j=1; j<=n2; j++)

R[j] = A[q+j];

L [n1+1] = myInfinite;

R [n2+1] = myInfinite;

i=1;

j=1;

for (k=p; k<=r; k++) {

if (L[i] <= R[j]) {

A[k] = L [i];

i++;

}

else {

A[k] = R[j];

j++;

}

}

}

void MergeSort (int A[], int p, int r) {

int q;

if (p<r) {

q = (p+r)/2;

MergeSort (A, p, q);

MergeSort (A, q+1, r);

myMerge (A, p, q, r);

}

}

int main() {

int n, q, i, c[MAXT], tipo, k, a, b, buscar, nuevaPosicion, posicionSuperior, posicionInferior, r;

scanf ("%d %d", &n, &q);

for (i=1; i<=n; i++)

scanf ("%d", &c[i]);

MergeSort (c, 1, n);

for (i=1; i<=q; i++) {

scanf ("%d", &tipo);

if (tipo == 1) {

scanf (" %d", &k);

buscar = binarySearch (c, 1, n, k);

if (buscar < 0) {

nuevaPosicion = -(buscar+1);

c [nuevaPosicion] = k;

if (c[n] < k)

n++;

}

}

else if (tipo == 2) {

scanf (" %d %d", &a, &b);

posicionSuperior = binarySearchLastOccurrence (c, 1, n, b);

posicionInferior = binarySearchFirstOccurrence (c, 1, n, a);

if (posicionInferior < 0)

posicionInferior = -(posicionInferior+1);

if (posicionSuperior < 0)

posicionSuperior = -(posicionSuperior+2);

r = posicionSuperior - posicionInferior + 1;

printf ("%d\n", r);

}

}

return 0;

}

## Streets Ahead

International Connecting Passage Causeway is a long, rutted two-way country road crossed by streets at different points.

There are many drivers, and each will drive along the country road starting at some intersection and ending at some other intersection. For each driver, how many intersections will they drive through?

**Input Format**

The first line contains two integers,   and  , where  is the number of cross streets and  is the number of drivers.

Each of the next  lines contains a string of at most ten lowercase letters representing the name of one of the streets that crosses the country road. All street names are unique. Driving along the country road in some direction, one sees these streets in exactly the order provided.

Each of the next  lines contains two strings of at most ten lowercase letters representing the start and end intersection for each driver. Queries will be between distinct streets.

**Constraints**

**Output Format**

Output  lines, the -th line containing the number of intersections that the -th driver drives through.

**Sample Input 0**

3 3

first

second

third

first second

third first

second third

**Sample Output 0**

0

1

0

**MI SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define MAXT 100001

#define myInfiniteChar "zzzzzzzzzz"

#define myInfiniteInt 2147483647

#define MAXNAME 11

struct Street {

char name [MAXNAME];

int number;

};

struct Street streets [MAXT];

int binarySearch (struct Street streets [], int i, int j, char k []) {

int m, comparar;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

comparar = strcmp (streets[m].name, k);

if (comparar == 0)

return m;

else {

if (comparar < 0)

return binarySearch (streets, m+1, j, k);

else

return binarySearch (streets, i, m-1, k);

}

}

}

void myMerge (struct Street streets [], int p, int q, int r) {

int n1 = q-p+1, n2 = r-q, i, j, k, comparar;

struct Street L[n1+2], R[n2+2];

for (i=1; i<=n1; i++)

L[i] = streets[p+i-1];

for (j=1; j<=n2; j++)

R[j] = streets[q+j];

strcpy(L[n1 + 1].name, myInfiniteChar);

L [n1+1].number = myInfiniteInt;

strcpy(R[n2 + 1].name, myInfiniteChar);

R [n2+1].number = myInfiniteInt;

i=1;

j=1;

for (k=p; k<=r; k++) {

comparar = strcmp (L[i].name, R[j].name);

if (comparar <= 0) {

streets[k] = L [i];

i++;

}

else {

streets[k] = R [j];

j++;

}

}

}

void MergeSort (struct Street streets[], int p, int r) {

int q;

if (p<r) {

q = (p+r)/2;

MergeSort (streets, p, q);

MergeSort (streets, q+1, r);

myMerge (streets, p, q, r);

}

}

int main() {

int n, q, i, startingPosition, lastPosition, result;

char startingStreet [MAXNAME], lastStreet [MAXNAME];

scanf ("%d %d", &n, &q);

for (i=1; i<=n; i++) {

scanf ("%s", streets[i].name);

streets[i].number = i;

}

MergeSort (streets, 1, n);

for (i=1; i<=q; i++) {

scanf ("%s %s", startingStreet, lastStreet);

startingPosition = binarySearch (streets, 1, n, startingStreet);

lastPosition = binarySearch (streets, 1, n, lastStreet);

result = abs(streets[startingPosition].number - streets[lastPosition].number) -1;

printf ("%d\n", result);

}

return 0;

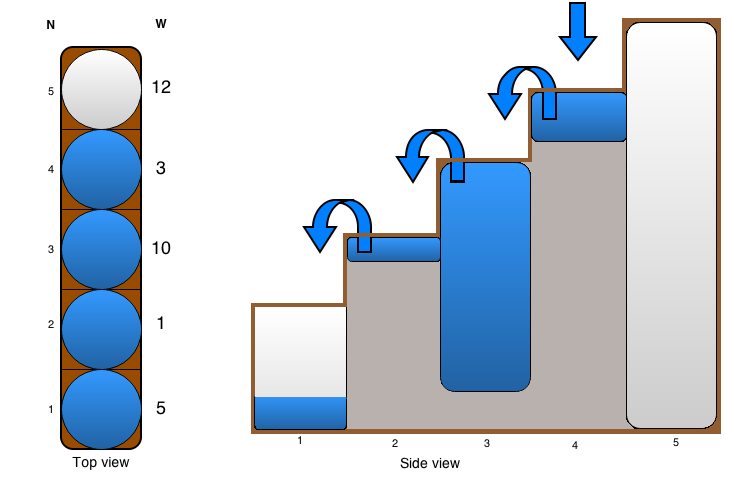
}

**SOLUCION DEL PROFE**

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#define myInfinite "zzzzzzzzzz"  
#define MAXN 100000  
  
typedef struct  
{  
    char name[11];  
    int idStree;  
} street;  
  
void myMerge(street A[], int p, int q, int r)  
{  
    int i, j, k, n1 = q - p + 1, n2 = r - q;  
    street L[n1 + 2], R[n2 + 2];  
     
    for(i = 1; i <= n1; i++)  
        L[i] = A[p + i - 1];  
         
    for(j = 1; j <= n2; j++)  
        R[j] = A[q + j];  
     
    strcpy (L[n1 + 1].name, myInfinite);  
    strcpy (R[n2 + 1].name, myInfinite);  
    i = 1;  
    j = 1;  
     
    for(k = p; k <= r; k++)  
    {  
        if(strcmp(L[i].name, R[j].name) <= 0)  
        {  
            A[k] = L[i];  
            i++;  
        }  
        else  
        {  
            A[k] = R[j];  
            j++;  
        }  
    }  
  
}  
  
void MergeSort(street A[], int p, int r)  
{  
    int q;  
    if(p < r)  
    {  
        q = (p + r) / 2;  
        MergeSort(A,p,q);  
        MergeSort(A, q + 1, r);  
        myMerge(A,p,q,r);  
    }  
}  
  
  
int BinarySearch(street A[],int i,int j,char k[]){  
    int m;  
    if(i > j){  
        return -1 \* i - 1;  
    }  
    else{  
        m = (i + j) / 2;  
        if(strcmp(A[m].name , k) == 0){  
            return m;  
        }  
        else  
        {  
            if(strcmp(A[m].name, k) < 0)  
                return BinarySearch(A,m + 1, j, k);  
             else  
                return BinarySearch(A,i,m - 1,k);  
        }  
    }  
}  
  
int main()  
{  
    int n,q,index,idQuery,idStreet01,idStreet02, result;  
    street A[MAXN + 1];  
    char cadena[11], street01[11], street02[11];  
     
    scanf("%d %d", &n, &q);  
    for(index = 1; index <= n; index++)  
    {  
        scanf("%s", cadena);  
        strcpy(A[index].name, cadena);  
        A[index].idStree = index;  
    }  
     
    MergeSort(A, 1, n);  
     
    for(idQuery = 1; idQuery <= q; idQuery++)  
    {  
        scanf("%s %s", street01, street02);  
        idStreet01 = BinarySearch(A, 1, n, street01);  
        idStreet02 = BinarySearch(A, 1, n, street02);  
         
        result = A[idStreet01].idStree - A[idStreet02].idStree;  
        if(result < 0)  
            result \*= -1;  
        result--;  
         
        printf("%d\n", result);  
    }  
    return 0;  
}

## Tobby and Tanks II

Tobby is a curious dog and he is always thinking about new and revolutionary ideas. Now he imagined  tanks placed in a row (staggered - See the figure). Every tank has a capacity of  liters of water. The cute and curious dog is asked himself, if you have  liters of water, which is the -th tank (where  is the maximum possible), such that if we drop the water there, the water will reach the first tank. As you know Tobby is not a complicated dog, so he does not want completely fill the first tank, he will be happy if the first tank have at least 1 liter of water.



You are given  tanks with capacity , and  queries, each query contains one integer , the amount of water.

**Input Format**

For every test case, the first line contains two integers   and  , the number of tanks and the number of queries respectively, the next line contains  integers  . The next  lines contains a single integer  , the amount of water. (Read until EOF).

**Constraints**

**Output Format**

For every query you must print a single integer, the maximum

**Sample Input 0**

5 2

5 1 10 3 12

16 1

**Sample Output 0**

4 1

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define MAXT 100001

#define MAXQ 10001

int binarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return binarySearch (A, m+1, j, k);

else

return binarySearch (A, i, m-1, k);

}

}

}

int main() {

int n, q, i, w[MAXT], acumulados [MAXT], acumulado, k[MAXQ], result;

while (scanf ("%d %d", &n, &q) != EOF) {

for (i=1; i<=n; i++)

scanf ("%d", &w[i]);

acumulados [1] = 1;

acumulado = 1;

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

acumulado += w[i];

acumulados [i] = acumulado;

}

for (i=1; i<=q; i++)

scanf ("%d", &k[i]);

for (i=1; i<=q; i++) {

result = binarySearch (acumulados, 1, n, k[i]);

if (result < 0)

result = -(result+2);

if (i==1)

printf ("%d", result);

else

printf (" %d", result);

}

printf ("\n");

}

return 0;

}

## Numeric Center I

A **numeric center** is a number that separates in a consecutive and positive integer number list (starting at one) in two groups of consecutive and positive integer numbers, in which their sum is the same. The first numeric center is number 6, which takes the list  and produces two lists of consecutive and positive integer numbers in which their sum (in this case 15) is the same. Those lists are:  and . The second numeric center is 35, that takes the list  and produces the following two lists:  and  , the sum of each list is equal to 595.

The task consists in writing a program that calculates the total of numeric centers between  and .

**Input Format**

The input consists of several test cases. There is only one line for each test case. This line contains a positive integer number  (). The last test case is a value of  equal to zero, this test case should not be processed.

**Constraints**

**Output Format**

For each test case you have to print in one line, the number of numeric centers between 1 and .

**Sample Input 0**

1

7

8

48

49

50

0

**Sample Output 0**

0

0

1

1

2

2

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define MAXT 1000001

int binarySearch (long long int A [], int i, int j, long long int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return binarySearch (A, m+1, j, k);

else

return binarySearch (A, i, m-1, k);

}

}

}

int main() {

long long int numericCenter [MAXT], numberToLookFor, testCenter;

int n, i, numberFlag = 0, j, lastNumberCenter = 2, flag [MAXT];

numericCenter [1] = 1;

flag [1] = 0;

for (i=2; i<MAXT; i++)

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

for (i=2; i<MAXT; i++) {

numberToLookFor = numericCenter [i] + numericCenter [i-1];

testCenter = binarySearch (numericCenter, 1, MAXT - 1, numberToLookFor);

if (testCenter > 0) {

for (j=lastNumberCenter; j<testCenter; j++)

flag [j] = numberFlag;

lastNumberCenter = testCenter;

numberFlag ++;

}

}

for (j=lastNumberCenter; j<MAXT; j++)

flag [j] = numberFlag;

while (scanf ("%d", &n) && (n>0))

printf ("%d\n", flag [n]);

return 0;

}

**Puntos en un Intervalo Cerrado**

Dado un intervalo y un arreglo, buscar en el intervalo. Se da n – tamaño del arreglo, q- numero de consultas y cada consulta en pares A y B – intervalos. Varios casos hasta que n y q son 0 0.

**SOLUCION**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#define MAXT 100001

#define myInfinite 2147483647

int binarySearch (int A [], int i, int j, int k) {

int m;

if (i>j)

return -1\*i -1;

else {

m = (i+j) / 2;

if (A[m] == k)

return m;

else {

if (k >A[m])

return binarySearch (A, m+1, j, k);

else

return binarySearch (A, i, m-1, k);

}

}

}

int binarySearchFirstOccurrence(int A[], int i, int j, int k) {

int result, result2;

result = binarySearch(A, i, j, k);

if (result >= 0) {

result2 = binarySearch(A, i, result - 1, k);

while (result2 >= 0) {

result = result2;

result2 = binarySearch(A, i, result - 1, k);

}

}

return result;

}

int binarySearchLastOccurrence(int A[], int i, int j, int k) {

int result, result2;

result = binarySearch(A, i, j, k);

if (result >= 0) {

result2 = binarySearch(A, result + 1, j, k);

while (result2 >= 0) {

result = result2;

result2 = binarySearch(A, result + 1, j, k);

}

}

return result;

}

void myMerge (int A[], int p, int q, int r) {

int n1 = q-p+1, n2 = r-q, i, j, k;

int L[n1+2], R[n2+2];

for (i=1; i<=n1; i++)

L[i] = A[p+i-1];

for (j=1; j<=n2; j++)

R[j] = A[q+j];

L [n1+1] = myInfinite;

R [n2+1] = myInfinite;

i=1;

j=1;

for (k=p; k<=r; k++) {

if (L[i] <= R[j]) {

A[k] = L [i];

i++;

}

else {

A[k] = R[j];

j++;

}

}

}

void MergeSort (int A[], int p, int r) {

int q;

if (p<r) {

q = (p+r)/2;

MergeSort (A, p, q);

MergeSort (A, q+1, r);

myMerge (A, p, q, r);

}

}

int main () {

int n, q, i, A, B, array [MAXT], inferiorRange, superiorRange, result;

while (scanf ("%d %d", &n, &q) && (n > 0)) {

for (i=1; i<=n; i++)

scanf ("%d", &array[i]);

MergeSort (array, 1, n);

for (i=1; i<=q; i++) {

scanf ("%d %d", &A, &B);

inferiorRange = binarySearchFirstOccurrence (array, 1, n, A);

superiorRange = binarySearchLastOccurrence (array, 1, n, B);

if (inferiorRange < 0)

inferiorRange = -(inferiorRange+1);

if (superiorRange < 0)

superiorRange = -(superiorRange+2);

result = superiorRange - inferiorRange + 1;

printf ("%d\n", result);

}

}

return 0;

}

## Fermat's Polygonal Number Theorem

En 1638, Fermat propuso que cada número entero positivo es la suma de como máximo tres números triangulares, cuatro números cuadrados, cinco números pentagonales, y  números -poligonales. Esto es lo que se conoce como el *Teorema de los números poligonales de Fermat*.

Apoyados en el *Teorema de los números poligonales de Fermat*, en este ejercicio se debe contar la cantidad de formas diferentes de obtener un número entero positivo  con la suma de uno, dos o tres números triangulares, donde el número triangular que se suma podría llegar a ser el mismo.

Por ejemplo el número  se obtiene de dos formas diferentes, sumando tres veces el número triangular  () y sumando los números triangulares  y  ().

**Input Format**

La entrada del problema comienza con un número entero positivo  , el cual representa el total de casos de prueba. Cada caso de prueba es presentado en una sola línea que contiene un número entero positivo  .

**Constraints**

**Output Format**

Para cada caso de prueba, su programa debe imprimir un número entero positivo que denota el total de formas diferentes como se puede obtener el número entero positivo  por la suma de uno, dos o tres números triangulares. Cada caso de prueba debe generar una línea en la salida.

**Sample Input 0**

15

1

2

3

4

5

6

7

8

9

10

21

52

66

112

177

**Sample Output 0**

1

1

2

1

1

2

2

1

2

2

4

5

6

7

8

**SOLUCION**

#include <stdio.h>

#include <stdlib.h>

#define MAXT 99681

#define MAXN 7861109

#define myInfinite 2147483647

int binarySearch (int\* A, int i, int j, int k){

int q;

while (i <= j){

q = (i + j) >> 1;

if (A[q] == k)

return q;

else if (k > A[q])

i = q + 1;

else

j = q - 1;

}

return -i - 1;

}

int binarySearchFirstOcurrence (int\* A, int k){

int result = binarySearch (A, 0, MAXN, k);

if (result >= 0){

while (binarySearch (A, 0, result - 1, k) >= 0)

result --;

}

return result ;

}

int binarySearchLastOcurrence (int\* A, int k) {

int result = binarySearch (A, 0, MAXN, k);

if (result >= 0) {

while (binarySearch (A, result + 1 , MAXN, k) >= 0)

result ++;

}

return result;

}

void myMerge (int\* A, int p, int q, int r){

int n1 = q - p + 1, n2 = r - q;

int i, j, k;

int\* L = (int\*) malloc((n1 + 2) \* sizeof(int));

int\* R = (int\*) malloc((n2 + 2) \* sizeof(int));

for (i = 1; i <= n1; i ++)

L[i] = A[p + i - 1];

for (j = 1; j <= n2; j ++)

R[j] = A[q + j];

L[n1 + 1] = myInfinite;

R[n2 + 1] = myInfinite;

i = j = 1;

for (k = p; k <= r; k ++){

if(L[i] <= R[j]){

A[k] = L[i];

i ++;

} else {

A[k] = R[j];

j ++;

}

}

free(L);

free(R);

}

void mergeSort(int\* A, int p, int r) {

int q;

if (p < r) {

q = (p + r) >> 1;

mergeSort(A, p, q);

mergeSort(A, q + 1, r);

myMerge(A, p, q, r);

}

}

int main() {

int A = (int) malloc(MAXN \* sizeof(int)), num, cantidadNums, indice = 0;

for (int i = 1, iIndex = 2; i <= MAXT; i += iIndex, iIndex ++){

for (int j = 0, jIndex = 1; j <= i; j += jIndex, jIndex++){

for (int k = 0, kIndex = 1; k <= j; k += kIndex, kIndex++){

if (i + j + k <= 100000){

A[indice] = i + j + k;

indice ++;

}

}

}

}

mergeSort(A, 0, MAXN);

for (scanf ("%d", &cantidadNums); cantidadNums > 0; cantidadNums --){

scanf ("%d", &num);

printf ("%d\n", binarySearchLastOcurrence(A, num) - binarySearchFirstOcurrence(A, num) + 1);

}

free(A);

return 0;

}