



Universidad  
Tecnológica  
de Pereira

CAPÍTULO COMPLETO: COMPLEJIDAD COMPUTACIONAL  
CURSO DE ESTRUCTURA DE DATOS - Código IS304

Programa de Ingeniería de Sistemas y Computación  
Profesor Hugo Humberto Morales Peña  
Semestre Agosto/Diciembre de 2021

## Algoritmo de Búsqueda Binaria (Binary Search)

En esta función se tienen como parámetros  $A$  que es un arreglo de elementos,  $i$  y  $j$  son índices que indican las posiciones del primer y del último elemento en el rango de valores del arreglo sobre el cual se realiza la búsqueda, y  $k$  es el elemento a buscar en el arreglo. Se exige como **precondición** para la Búsqueda Binaria que los elementos del arreglo  $A$  tienen que estar ordenados de forma ascendente.

```
1: function BINARYSEARCH( $A[ ]$ ,  $i$ ,  $j$ ,  $k$ )
2:   if  $i > j$  then
3:     return  $(-1) * i - 1$ 
4:   else
5:      $m \leftarrow \lfloor (i + j) / 2 \rfloor$ 
6:     if  $k == A[m]$  then
7:       return  $m$ 
8:     else
9:       if  $k > A[m]$  then
10:        return BINARYSEARCH( $A[ ]$ ,  $m + 1$ ,  $j$ ,  $k$ )
11:      else
12:        return BINARYSEARCH( $A[ ]$ ,  $i$ ,  $m - 1$ ,  $k$ )
13:      end if
14:    end if
15:  end if
16: end function
```

Se tiene como **postcondición** para la Búsqueda Binaria que el valor devuelto por la función es un entero correspondiente al índice del elemento que coincide con el valor buscado. Si el valor buscado no se encuentra, entonces el valor devuelto es:  $-1 * (\text{punto de inserción}) - 1$ . El valor del *punto de inserción* es el índice del elemento del arreglo donde debería encontrarse el valor buscado. La expresión:  $-1 * (\text{punto de inserción}) - 1$  garantiza que el índice devuelto será mayor o igual que cero solo si el valor buscado es encontrado.

## Ejemplo

Realizar el paso a paso de la Búsqueda Binaria cuando se tiene el arreglo de elementos:

$A[i]$	2	4	7	9	10	11	13	14	17	30
$i$	1	2	3	4	5	6	7	8	9	10

y se manda a buscar el número 12.

### Paso a paso del algoritmo BINARYSEARCH

```

BINARYSEARCH(A, 1, 10, 12)
  i   j   k   m
  1   10  12  5
  BINARYSEARCH(A, 6, 10, 12)
    i   j   k   m
    6   10  12  8
    BINARYSEARCH(A, 6, 7, 12)
      i   j   k   m
      6   7   12  6
      BINARYSEARCH(A, 7, 7, 12)
        i   j   k   m
        7   7   12  7
        BINARYSEARCH(A, 7, 6, 12)
          i   j   k
          7   6   12
          return -8

```

La función `BinarySearch` devolvió un  $-8$ , como es un número negativo entonces el número 12 no se encuentra presente en el arreglo. Recordar que  $-8 = -1 \cdot \text{puntoInsercion} - 1$ , donde  $\text{puntoInsercion} = 7$ , con lo cual se indica que el elemento 12 debería estar ubicado en la posición 7 del arreglo, pero dicho elemento no se encuentra allí.

### Complejidad en tiempo de ejecución del Algoritmo BINARYSEARCH

Sea  $T(n)$  la función que cuenta el total de operaciones que realiza el algoritmo BINARYSEARCH para determinar si el número  $k$  se encuentra presente en el arreglo  $A$  el cual tiene  $n$  elementos.

$$T(1) = O(1)$$

$$T(n) = \underbrace{O(1)}_{\text{Costo de las líneas 2-7}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{Costo de las líneas 8-14}}, \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

$$T(n) = T\left(\frac{n}{2}\right) + O(1), \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

Para poder resolver la relación de recurrencia primero se deben limpiar todas las notaciones computacionales y considerar que  $n$  toma valores que son potencias de 2, por lo tanto tenemos:

$$T(1) = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + 1, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

En el *Ejemplo 4* de la clase anterior (clase 7) se resolvió esta relación de recurrencia, donde se obtuvo que en términos de la variable original la solución de la relación de recurrencia es:

$$T(n) = \log_2(n) + 1, \text{ para } n = 2^m, m \geq 0, m \in \mathbb{N}$$

Ahora toca aplicar sobre la solución de la relación de recurrencia la notación computacional que fue borrada de la relación de recurrencia, con lo cual obtenemos que  $T(n) = O(\log(n))$ , o dicho en palabras, tenemos que

en el peor de los casos la Búsqueda Binaria realiza  $\log(n)$  operaciones sobre el arreglo  $A$  para determinar si el elemento  $k$  se encuentra presente en el. Donde  $n$  es la cantidad de elementos que se encuentran almacenados en el arreglo  $A$ .

## Algoritmo Iterativo de Búsqueda Binaria

El algoritmo clásico que ya se trabajó previamente en el capítulo para la Búsqueda Binaria es una **recursividad de cola**, debido a que la última instrucción que se ejecuta en el algoritmo es un llamado recursivo. Todo algoritmo que tenga una recursividad de cola puede transformarse a una versión iterativa donde la recursividad se simula por medio de un ciclo de repetición. Las versiones recursiva e iterativa del algoritmo siguen teniendo la misma complejidad teórica, pero el factor constante en la versión iterativa de la Búsqueda Binaria es más pequeño al no tener que hacer uso de la memoria que se reserva para la pila de los trabajos pendientes del sistema operativo. Por este motivo en el momento de implementar los algoritmos siempre se va a preferir la versión iterativa versus la versión recursiva.

El siguiente algoritmo es la variante iterativa de la Búsqueda Binaria, donde se siguen teniendo los mismos parámetros de entrada, la misma precondition y la misma postcondición.

```

1: function BINARYSEARCH( $A[ \ ]$ ,  $i$ ,  $j$ ,  $k$ )
2:    $r \leftarrow -1$ 
3:   while  $i \leq j$  do
4:      $m \leftarrow \lfloor (i + j)/2 \rfloor$ 
5:     if  $k == A[m]$  then
6:        $r \leftarrow m$ 
7:       break
8:     else
9:       if  $k > A[m]$  then
10:         $i \leftarrow m + 1$ 
11:      else
12:         $j \leftarrow m - 1$ 
13:      end if
14:    end if
15:  end while
16:  if  $r == -1$  then
17:     $r \leftarrow (-1) * i - 1$ 
18:  end if
19:  return  $r$ 
20: end function

```

## Implementación del Algoritmo de Búsqueda Binaria

Se implementará la versión iterativa del algoritmo de Búsqueda Binaria.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*****
/* El valor devuelto por la funcion es un entero correspondiente al indice del elemento */
/* que coincide con el valor buscado. Si el valor buscado no se encuentra, entonces el */
/* valor devuelto es: - (punto de insercion) - 1. El valor del punto de insercion es el */
/* indice del elemento del arreglo donde deberia encontrarse el valor buscado.      */
/* La expresion: "- (punto de insercion) - 1" garantiza que el indice devuelto sera */
/* mayor o igual que cero solo si el valor buscado es encontrado.                  */
*****/

```

```

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;
    while(i <= j)
    {
        m = (i + j) >> 1; /* m = (i + j) / 2; */
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])
                i = m + 1;
            else
                j = m - 1;
        }
    }
    if(result == -1)
        result = (-1) * i - 1;

    return result;
}

int main()
{
    int A[100], index, n, queries, idQuery, k, position;

    scanf("%d", &n);
    for(index = 1; index <= n; index++)
        scanf("%d", &A[index]);

    scanf("%d", &queries);
    for(idQuery = 1; idQuery <= queries; idQuery++)
    {
        scanf("%d", &k);
        position = BinarySearch(A, 1, n, k);
        if(position >= 0)
            printf("The element %d is in the array, position: %d\n", k, position);
        else
            printf("The element %d is not in the array, insertion point: %d\n",
                k, -1 * position - 1);
    }

    return 0;
}

```

```

F:\HUGO\EstructuraDeDatos\Clases Virtuales\2021_01\Clase08_Febrero24de2021\codigoEjemploLenguajeC\BinarySearch.exe
10
2 4 7 9 10 11 13 14 17 30
5
12
The element 12 is not in the array, insertion point: 7
13
The element 13 is in the array, position: 7
50
The element 50 is not in the array, insertion point: 11
1
The element 1 is not in the array, insertion point: 1
10
The element 10 is in the array, position: 5

```

Figura 1: Salida del programa de la Búsqueda Binaria.

## Programming Challenge: “The Book Thief”<sup>1</sup>

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  $s$  ( $1 \leq s \leq 10^8$ ) representing the result obtained by Anita, find out the number of the forgotten page and the total number of pages in the book.

### Input specification:

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  $s$  is zero, and this case must not be processed.

### Output specification:

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.

### Example input:

```
1
2
3
4
5
6
9000
499977
4999775
0
```

### Example output:

```
2 2
1 2
3 3
2 3
1 3
4 4
45 134
523 1000
5225 10000
```

## Solución del reto “The Book Thief” usando Búsqueda Binaria

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXT 14142

/*****
```

---

<sup>1</sup><https://www.hackerrank.com/contests/utp-open-2018/challenges/the-book-thief-1>

```

/* Total de numeros triangulares en la talla del reto de programacion: */
/* (14142 * 14143) / 2 = 100005153 */
/*****/

/*****/
/* El valor devuelto por la funcion es un entero correspondiente al indice del elemento */
/* que coincide con el valor buscado. Si el valor buscado no se encuentra, entonces el */
/* valor devuelto es: - (punto de insercion) - 1. El valor del punto de insercion es el */
/* indice del elemento del arreglo donde deberia encontrarse el valor buscado. */
/* La expresion: "- (punto de insercion) - 1" garantiza que el indice devuelto sera */
/* mayor o igual que cero solo si el valor buscado es encontrado. */
/*****/

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;
    while(i <= j)
    {
        m = (i + j)>>1; /* m = (i + j)/2; */
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])
                i = m + 1;
            else
                j = m - 1;
        }
    }
    if(result == -1)
        result = (-1) * i - 1;

    return result;
}

int main()
{
    int s, pages, forgottenPage, i, index;
    int triangularNumbers[MAXT + 1];

    triangularNumbers[0] = 0;
    for(i = 1; i <= MAXT; i++)
        triangularNumbers[i] = triangularNumbers[i - 1] + i;

    while(scanf("%d", &s) && (s > 0))
    {
        index = BinarySearch(triangularNumbers, 1, MAXT, s);

        if(index > 0)
            pages = index + 1;
        else
            pages = -1 * (index + 1);

        forgottenPage = triangularNumbers[pages] - s;
        printf("%d %d\n", forgottenPage, pages);
    }

    return 0;
}

```

```
F:\HUGO\EstructuraDeDatos\Clases Virtuales\Semestre Agosto Diciembre 2020\Clase10_Septiembre 09 de 2020\codigo ejemplo...
1
2 2
2
1 2
3
3 3
4
2 3
5
1 3
6
4 4
9000
45 134
499977
523 1000
49999775
5225 10000
0
```

Figura 2: Salida del programa para el reto “The Book Thief”.

## Búsqueda Binaria de un elemento que está múltiples veces en el arreglo

El algoritmo de Búsqueda Binaria trabaja de forma correcta sobre un arreglo de elementos ordenado de forma ascendente que tenga elementos repetidos. Si el elemento  $k$  que se manda a buscar se encuentra múltiples veces en el arreglo, el algoritmo termina retornando alguna de las posiciones del arreglo sobre la cual se encuentra el elemento almacenado. Pero, no se puede garantizar que la posición devuelta por el algoritmo sea la primera o la última ocurrencia del elemento en el arreglo. Por ello es necesario trabajar las variantes del algoritmo de Búsqueda Binaria para determinar ya sea del caso, la posición de la primera o de la última ocurrencia del elemento  $k$ .

El siguiente algoritmo es la variante de la Búsqueda Binaria para determinar la posición de la primera ocurrencia de un elemento  $k$ , donde se siguen teniendo los mismos parámetros de entrada y la misma precondition del algoritmo de Búsqueda Binaria.

```
1: function BINARYSEARCHFIRSTOCCURRENCE( $A[ \ ], i, j, k$ )
2:    $r \leftarrow$  BINARYSEARCH( $A[ \ ], i, j, k$ )
3:   if  $r \geq 0$  then
4:      $r2 \leftarrow$  BINARYSEARCH( $A[ \ ], i, r - 1, k$ )
5:     while  $r2 \geq 0$  do
6:        $r \leftarrow r2$ 
7:        $r2 \leftarrow$  BINARYSEARCH( $A[ \ ], i, r - 1, k$ )
8:     end while
9:   end if
10:  return  $r$ 
11: end function
```

Se tiene como **postcondición** del algoritmo que el valor devuelto es un entero correspondiente al índice de la primera ocurrencia del elemento que coincide con el valor buscado en el arreglo. Si el valor buscado no se encuentra en el arreglo, entonces el valor devuelto es el resultado del llamado al algoritmo de Búsqueda Binaria de la línea 2, el cual es:  $-1 * (\text{punto de inserción}) - 1$ . De nuevo, el valor del *punto de inserción* es el índice en el

arreglo donde debería encontrarse el valor buscado.

El siguiente algoritmo es la variante de la Búsqueda Binaria para determinar la posición de la última ocurrencia de un elemento  $k$ , donde se siguen teniendo los mismos parámetros de entrada y la misma precondition del algoritmo de Búsqueda Binaria.

```
1: function BINARYSEARCHLASTOCCURRENCE( $A[ \ ], i, j, k$ )
2:    $r \leftarrow$  BINARYSEARCH( $A[ \ ], i, j, k$ )
3:   if  $r \geq 0$  then
4:      $r2 \leftarrow$  BINARYSEARCH( $A[ \ ], r + 1, j, k$ )
5:     while  $r2 \geq 0$  do
6:        $r \leftarrow r2$ 
7:        $r2 \leftarrow$  BINARYSEARCH( $A[ \ ], r + 1, j, k$ )
8:     end while
9:   end if
10:  return  $r$ 
11: end function
```

Se tiene como **postcondición** del algoritmo que el valor devuelto es un entero correspondiente al índice de la última ocurrencia del elemento que coincide con el valor buscado en el arreglo. Si el valor buscado no se encuentra en el arreglo, entonces el valor devuelto es el resultado del llamado al algoritmo de Búsqueda Binaria de la línea 2, el cual es:  $-1 * (\text{punto de inserción}) - 1$ . De nuevo, el valor del *punto de inserción* es el índice en el arreglo donde debería encontrarse el valor buscado.

El código fuente en Lenguaje C de las dos variantes de la Búsqueda Binaria es el siguiente:

```
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;
}

```

## Programming Challenge: “Attractive Subsequence”<sup>2</sup>

You receive a sequence  $S$  of non-negative integers numbers. Your task is calculate the total of Attractive Subsequences. An Attractive Subsequences is a subsequence of consecutive elements in  $S$  that the sum of the elements in it is equal to the value  $K$ . For example, consider the sequence  $S = \langle 0, 0, 25, 0, 0, 25 \rangle$  and the value  $K = 25$ , there are 12 Attractive Subsequences, these are represented with ordered pairs  $(ind_1, ind_2)$ ,  $ind_1$  is the position of the first element and  $ind_2$  is the position of the last element in the original sequence  $S$ . In this representation the Attractive Subsequences are: (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5), (4, 6), (5, 6) and (6, 6).

### Input specification:

The first line in the input contains one integer  $T$  ( $1 \leq T \leq 20$ ), the total of test cases in the input. Each test case contains three lines, the first line contains two positive integers  $N$  ( $1 \leq N \leq 10^5$ ) and  $Q$  ( $1 \leq Q \leq 10^3$ ), the number of elements in the sequence  $S$  and the number of queries respectively. The next line contains  $N$  non-negative integers  $S_i$  ( $0 \leq S_i \leq 10^3$ ). The next line contains  $Q$  positive integers numbers  $K_j$  ( $1 \leq K_j \leq 10^7$ ), for the queries of the attractive subsequences.

### Output specification:

For each case you must print  $Q$  space-separated integers numbers in a single line, one per query, with the total of the attractive subsequences.

### Example input:

```

2
6 2
0 0 25 0 0 25
25 50
7 3
1 6 5 2 3 4 7
7 5 11

```

### Example output:

```

12 3
4 2 2

```

## Solución del reto “Attractive Subsequence”

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define MAXN 100000

/*****
/* El valor devuelto por la funcion es un entero correspondiente al indice del elemento */
/* que coincide con el valor buscado. Si el valor buscado no se encuentra, entonces el */
/* valor devuelto es: - (punto de insercion) - 1. El valor del punto de insercion es el */
/* indice del elemento del arreglo donde deberia encontrarse el valor buscado. */
/* La expresion: "- (punto de insercion) - 1" garantiza que el indice devuelto sera */
*****/

```

<sup>2</sup><https://www.hackerrank.com/contests/utp-open-2018/challenges/attractive-subsequence>

```

/* mayor o igual que cero solo si el valor buscado es encontrado. */
/*****

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;
    while(i <= j)
    {
        m = (i + j)>>1; /* m = (i + j)/2; */
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])
                i = m + 1;
            else
                j = m - 1;
        }
    }
    if(result == -1)
        result = (-1) * i - 1;

    return result;
}

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;
}

```

```

int main()
{
    int i, n, s, element, beginPosition, finalPosition;
    int totalCases, idCase, q, idQuery, index1, index2;
    int integralVector[MAXN + 1];
    long long int result;

    scanf("%d", &totalCases);

    for(idCase=1; idCase<=totalCases; idCase++)
    {
        scanf("%d %d", &n, &q);
        integralVector[0] = 0;

        for(i=1; i<=n; i++)
        {
            scanf("%d", &element);
            integralVector[i] = integralVector[i - 1] + element;
        }

        for(idQuery=1; idQuery<=q; idQuery++)
        {
            result = 0;
            scanf("%d", &s);
            for(i=0; i<=n; i++)
            {
                index1 = BinarySearchFirstOccurrence(integralVector,
                                                       i + 1, n, integralVector[i] + s);
                if(index1 > 0)
                {
                    beginPosition = index1;
                    index2 = BinarySearchLastOccurrence(integralVector,
                                                         index1, n, integralVector[i] + s);
                    finalPosition = index2;
                    result += (finalPosition - beginPosition + 1);
                }
            }
            if(idQuery == 1)
                printf("%lld", result);
            else
                printf(" %lld", result);
        }
        printf("\n");
    }

    return 0;
}

```

```

2
6 2
0 0 25 0 0 25
25 50
12 3
7 3
1 6 5 2 3 4 7
7 5 11
4 2 2

```

Figura 3: Salida del programa para el reto “Attractive Subsequence”.

## Algoritmo de Ordenamiento por Mezclas (Merge Sort)

### Función Merge

En esta subrutina se tienen como parámetros  $A[ ]$  que es un arreglo de elementos,  $p$ ,  $q$  y  $r$  son índices que numeran los elementos del arreglo tal que  $p \leq q < r$ .

```
1: function MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
2:    $n_1 \leftarrow q - p + 1$ 
3:    $n_2 \leftarrow r - q$ 
4:   create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
5:   for  $i = 1$  to  $n_1$  do
6:      $L[i] \leftarrow A[p + i - 1]$ 
7:   end for
8:   for  $j = 1$  to  $n_2$  do
9:      $R[j] \leftarrow A[q + j]$ 
10:  end for
11:   $L[n_1 + 1] \leftarrow \infty$ 
12:   $R[n_2 + 1] \leftarrow \infty$ 
13:   $i \leftarrow 1$ 
14:   $j \leftarrow 1$ 
15:  for  $k = p$  to  $r$  do
16:    if  $L[i] \leq R[j]$  then
17:       $A[k] \leftarrow L[i]$ 
18:       $i \leftarrow i + 1$ 
19:    else
20:       $A[k] \leftarrow R[j]$ 
21:       $j \leftarrow j + 1$ 
22:    end if
23:  end for
24: end function
```

### Función Merge Sort

Esta es la rutina que llama a MERGE.  $A[ ]$  es un arreglo de elementos,  $p$  y  $r$  son dos índices que denotan el inicio y el fin del arreglo, tal que  $p \leq r$ .

```
1: function MERGESORT( $A[ ]$ ,  $p$ ,  $r$ )
2:   if  $p < r$  then
3:      $q \leftarrow \lfloor (p + r) / 2 \rfloor$ 
4:     MERGESORT( $A[ ]$ ,  $p$ ,  $q$ )
5:     MERGESORT( $A[ ]$ ,  $q + 1$ ,  $r$ )
6:     MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
7:   end if
8: end function
```

### Ejemplo

Realizar el paso a paso del algoritmo MERGESORT sobre el arreglo de elementos:

$A[i]$	3	1	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8

## Paso a paso del algoritmo MERGESORT

MERGESORT( $A$ , 1, 8)

$p$	$r$	$q$
1	8	4

MERGESORT( $A$ , 1, 4)

$p$	$r$	$q$
1	4	2

MERGESORT( $A$ , 1, 2)

$p$	$r$	$q$
1	2	1

MERGESORT( $A$ , 1, 1)

$p$	$r$
1	1

MERGESORT( $A$ , 2, 2)

$p$	$r$
2	2

MERGE( $A$ , 1, 1, 2)

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
1	1	2	1	1	1	1	1
					2	2	2
					1	1	3
					2	2	

$A[i]$	3	1	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8

$L[i]$	3	$\infty$
$i$	1	2

$R[j]$	1	$\infty$
$j$	1	2

$A[i]$	1	3	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8

MERGESORT( $A$ , 3, 4)

$p$	$r$	$q$
3	4	3

MERGESORT( $A$ , 3, 3)

$p$	$r$
3	3

MERGESORT( $A$ , 4, 4)

$p$	$r$
4	4

MERGE( $A$ , 3, 3, 4)

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
3	3	4	1	1	1	1	3
					2	2	4
					1	1	5
					2	2	

$A[i]$	1	3	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8

$L[i]$	<b>4</b>	$\infty$
$i$	1	2

$R[j]$	<b>2</b>	$\infty$
$j$	1	2

$A[i]$	1	3	<b>2</b>	<b>4</b>	5	1	3	2
$i$	1	2	3	4	5	6	7	8

MERGE( $A$ , 1, 2, 4)

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
1	2	4	2	2	1	1	1
					2	2	2
					3	3	3
					1	1	4
					2	2	5
					3	3	

$A[i]$	<b>1</b>	<b>3</b>	<b>2</b>	<b>4</b>	5	1	3	2
$i$	1	2	3	4	5	6	7	8

$L[i]$	<b>1</b>	<b>3</b>	$\infty$
$i$	1	2	3

$R[j]$	<b>2</b>	<b>4</b>	$\infty$
$j$	1	2	3

$A[i]$	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	5	1	3	2
$i$	1	2	3	4	5	6	7	8

MERGESORT( $A$ , 5, 8)

$p$	$r$	$q$
5	8	6

MERGESORT( $A$ , 5, 6)

$p$	$r$	$q$
5	6	5

MERGESORT( $A$ , 5, 5)

$p$	$r$
5	5

MERGESORT( $A$ , 6, 6)

$p$	$r$
6	6

MERGE( $A$ , 5, 5, 6)

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
5	5	6	1	1	1	1	5
					2	2	6
					1	1	7
					2	2	

$A[i]$	1	2	3	4	<b>5</b>	<b>1</b>	3	2
$i$	1	2	3	4	5	6	7	8

$L[i]$	<b>5</b>	$\infty$
$i$	1	2

$R[j]$	<b>1</b>	$\infty$
$j$	1	2

$A[i]$	1	2	3	4	<b>1</b>	<b>5</b>	3	2
$i$	1	2	3	4	5	6	7	8

MERGESORT( $A$ , 7, 8)

$p$	$r$	$q$
7	8	7

MERGESORT( $A, 7, 7$ )

$p$	$r$
7	7

MERGESORT( $A, 8, 8$ )

$p$	$r$
8	8

MERGE( $A, 7, 7, 8$ )

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
7	7	8	1	1	1	1	7
					2	2	8
					1	1	9
					2	2	

$A[i]$	1	2	3	4	1	5	<b>3</b>	<b>2</b>
$i$	1	2	3	4	5	6	7	8

$L[i]$	<b>3</b>	$\infty$
$i$	1	2

$R[j]$	<b>2</b>	$\infty$
$j$	1	2

$A[i]$	1	2	3	4	1	5	<b>2</b>	<b>3</b>
$i$	1	2	3	4	5	6	7	8

MERGE( $A, 5, 6, 8$ )

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
5	6	8	2	2	1	1	5
					2	2	6
					3	3	7
					1	1	8
					2	2	9
					3	3	

$A[i]$	1	2	3	4	<b>1</b>	<b>5</b>	<b>2</b>	<b>3</b>
$i$	1	2	3	4	5	6	7	8

$L[i]$	<b>1</b>	<b>5</b>	$\infty$
$i$	1	2	3

$R[j]$	<b>2</b>	<b>3</b>	$\infty$
$j$	1	2	3

$A[i]$	1	2	3	4	<b>1</b>	<b>2</b>	<b>3</b>	<b>5</b>
$i$	1	2	3	4	5	6	7	8

MERGE( $A, 1, 4, 8$ )

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
1	4	8	4	4	1	1	1
					2	2	2
					3	3	3
					4	4	4
					5	5	5
					1	1	6
					2	2	7
					3	3	8
					4	4	9
					5	5	

$A[i]$	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>5</b>
$i$	1	2	3	4	5	6	7	8

$L[i]$	1	2	3	4	$\infty$						$R[j]$	1	2	3	5	$\infty$					
$i$	1	2	3	4	5						$j$	1	2	3	4	5					
$A[i]$	1	1	2	2	3	3	4	5													
$i$	1	2	3	4	5	6	7	8													

## Complejidad en espacio de almacenamiento del Algoritmo de Ordenamiento por Mezclas

La figura 4 representa el consumo de memoria realizado por el algoritmo de Ordenamiento por Mezclas al realizar las copias del sub-arreglo de los elementos de la izquierda y del sub-arreglo de los elementos de la derecha entre los diferentes llamados recursivos del algoritmo.

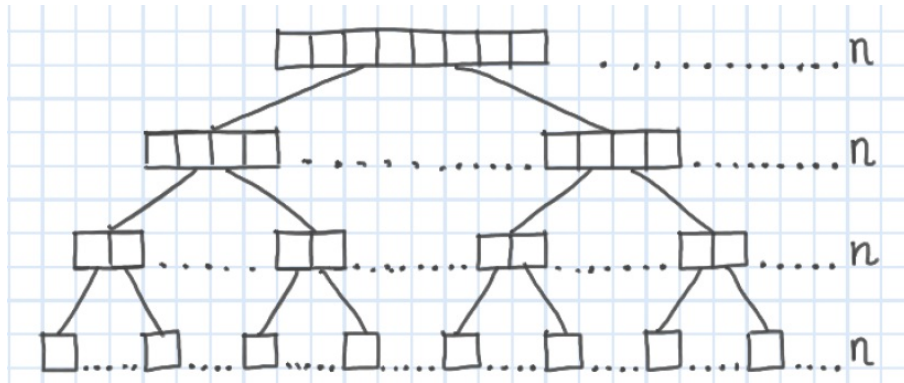


Figura 4: Árbol de consumo de la memoria generado por el algoritmo de Ordenamiento por Mezclas.

La complejidad en consumo de espacio de almacenamiento en memoria es  $n$  (el cual es la cantidad de elementos del arreglo  $A$ ) por la altura del árbol. Por cada nivel del árbol se necesita una copia completa de los  $n$  elementos del arreglo  $A$ .

Para poder determinar la altura del árbol entonces se considerará que  $n$  es una potencia del número 2 y que  $n$  se divide sucesivamente por 2 hasta llegar a 1, por lo tanto:

$$\frac{\left(\frac{n}{2}\right)}{2} = 1, \quad \frac{n}{2^{\text{altura}}} = 1, \quad n = 2^{\text{altura}}, \quad \log_2(n) = \log_2(2^{\text{altura}}) = \text{altura}, \quad \text{altura} = \log_2(n).$$

La complejidad en consumo de espacio de almacenamiento en memoria es:  $n \cdot \log_2(n) = \Theta(n \log n)$ .

## Complejidad en tiempo de ejecución del Algoritmo de Ordenamiento por Mezclas

### Función Merge

```

1: function MERGE( $A[ \ ], p, q, r$ )
2:    $n_1 \leftarrow q - p + 1$ 
3:    $n_2 \leftarrow r - q$ 
4:   create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
5:   for  $i = 1$  to  $n_1$  do
6:      $L[i] \leftarrow A[p + i - 1]$ 
7:   end for
8:   for  $j = 1$  to  $n_2$  do

```



```

9:       $R[j] \leftarrow A[q + j]$ 
10:    end for
11:     $L[n_1 + 1] \leftarrow \infty$ 
12:     $R[n_2 + 1] \leftarrow \infty$ 
13:     $i \leftarrow 1$ 
14:     $j \leftarrow 1$ 
15:    for  $k = p$  to  $r$  do
16:      if  $L[i] \leq R[j]$  then
17:         $A[k] \leftarrow L[i]$ 
18:         $i \leftarrow i + 1$ 
19:      else
20:         $A[k] \leftarrow R[j]$ 
21:         $j \leftarrow j + 1$ 
22:      end if
23:    end for
24: end function

```

Considerar que en el rango de  $p$  a  $r$  en el arreglo  $A$  hay  $n$  elementos. El ciclo de repetición **for** de las líneas 5-7 se ejecuta  $\frac{n}{2}$  veces. El ciclo de repetición **for** de las líneas 8-10 se ejecuta  $\frac{n}{2}$  veces. El ciclo de repetición **for** de las líneas 15-23 se ejecuta  $n$  veces. Por lo tanto, el costo total de la función MERGE es:  $\frac{n}{2} + \frac{n}{2} + n = 2n = \Theta(n)$ .

### Función Merge Sort

```

1: function MERGESORT( $A[ \ ], p, r$ )
2:   if  $p < r$  then
3:      $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
4:     MERGESORT( $A[ \ ], p, q$ )
5:     MERGESORT( $A[ \ ], q + 1, r$ )
6:     MERGE( $A[ \ ], p, q, r$ )
7:   end if
8: end function

```

De nuevo, considerar que en el rango de  $p$  a  $r$  en el arreglo  $A$  hay  $n$  elementos.

Sea  $T(n)$  la función que cuenta el total de operaciones que realiza el algoritmo MERGESORT para ordenar un arreglo  $A$  que tiene  $n$  elementos.

$$T(1) = \Theta(1)$$

$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\text{Costo de la línea 4}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{Costo de la línea 5}} + \underbrace{\Theta(n)}_{\text{Costo de la línea 6}}, \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n), \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

Para poder resolver la relación de recurrencia primero se deben limpiar todas las notaciones computacionales y considerar que  $n$  toma valores que son potencias de 2, por lo tanto tenemos:

$$T(1) = 1$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Primero se debe hacer el cambio de variable en la relación de recurrencia:

$$T(2^0) = 1 = 2^0$$

$$T(2^m) = 2 \cdot T(2^{m-1}) + 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Ahora si se puede aplicar el Método de Iteración con respecto al cambio de variable en la relación de recurrencia:

$$T(2^0) = 2^0$$

$$T(2^1) = 2 \cdot T(2^0) + 2^1 = 2 \cdot [2^0] + 2^1 = 2^1 + 2^1$$

$$T(2^2) = 2 \cdot T(2^1) + 2^2 = 2 \cdot [2^1 + 2^1] + 2^2 = 2^2 + 2^2 + 2^2$$

$$T(2^3) = 2 \cdot T(2^2) + 2^3 = 2 \cdot [2^2 + 2^2 + 2^2] + 2^3 = 2^3 + 2^3 + 2^3 + 2^3$$

⋮

$$T(2^m) = \underbrace{2^m + 2^m + 2^m + \dots + 2^m}_{m+1 \text{ veces el } 2^m}$$

$$T(2^m) = (m+1) \cdot 2^m$$

Por lo tanto la solución de la relación de recurrencia en términos del cambio de variable es:

$$T(2^m) = 2^m \cdot (m+1), \text{ para } m \geq 0, m \in \mathbb{N}.$$

Por último, se debe llevar la solución de la relación de recurrencia a la variable original.

Recordar que  $n = 2^m$ , si a la igualdad le aplico la misma operación en ambos lados ... sigue siendo una igualdad!,  
 $\log_2(n) = \log_2(2^m) = m$

Por lo tanto la solución de la relación de recurrencia en términos de la variable original es:

$$T(n) = n \cdot (\log_2(n) + 1),$$

$$T(n) = n \cdot \log_2(n) + n, \text{ para } n = 2^m, m \geq 0, m \in \mathbb{N}$$

Ahora toca aplicar sobre la solución de la relación de recurrencia la notación computacional que fue borrada para poderla resolver, con lo cual obtenemos que  $T(n) = \Theta(n \log n)$ , o dicho en palabras, tenemos que en el mejor y el peor de los casos el Algoritmo de Ordenamiento por Mezclas realiza del orden de  $n \log n$  operaciones sobre el arreglo  $A$  para ordenarlo. Donde  $n$  es la cantidad de elementos que se encuentran almacenados en el arreglo  $A$ .

## Implementación del Algoritmo de Ordenamiento por Mezclas

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, 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) >> 1;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }
}

int main()
{
    int A[20], i, n;

    while(scanf("%d", &n) != EOF)
    {
        for(i = 1; i <= n; i++)
            scanf("%d", &A[i]);

        for(i = 1; i <= n; i++)
            printf("%d ", A[i]);
        printf("\n");

        MergeSort(A, 1, n);

        for(i = 1; i <= n; i++)
            printf("%d ", A[i]);
        printf("\n");
    }

    return 0;
}

```

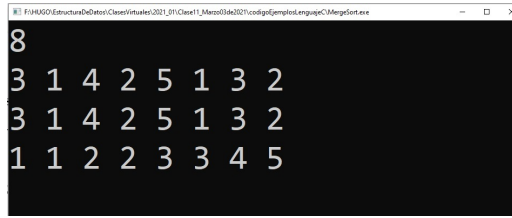


Figura 5: Salida del programa que realiza el Ordenamiento por Mezclas.

## Programming Challenge: “How Many Inversions?”<sup>3</sup>

*Humbertov Moralov* in his student days, he is attended 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  $A[1 \dots n]$  an array of distinct integers of size  $n$ . If  $i < j$  and  $A[i] > A[j]$ , then the pair  $(i, j)$  is called an **inversion** of  $A$ .

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

1. List all inversions in  $\langle 3, 2, 8, 1, 6 \rangle$ .
2. What array of size  $n$ , with all the numbers from the set  $1, 2, 3, \dots, n$  has the largest amount of inversions? How many inversions?
3. Write an algorithm to determine the number of inversions in any permutation of  $n$  elements with  $\theta(n \log n)$  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:

```

1: inv  $\leftarrow$  0
2: function MERGE( $A[ \ ], p, q, r$ )
3:    $n_1 \leftarrow q - p + 1$ 
4:    $n_2 \leftarrow r - q$ 
5:   create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
6:   for  $i = 1$  to  $n_1$  do
7:      $L[i] \leftarrow A[p + i - 1]$ 
8:   end for
9:   for  $j = 1$  to  $n_2$  do
10:     $R[j] \leftarrow A[q + j]$ 
11:  end for
12:   $L[n_1 + 1] \leftarrow \infty$ 
13:   $R[n_2 + 1] \leftarrow \infty$ 
14:   $i \leftarrow 1$ 
15:   $j \leftarrow 1$ 
16:  for  $k = p$  to  $r$  do
17:    if  $L[i] \leq R[j]$  then
18:       $A[k] \leftarrow L[i]$ 

```

<sup>3</sup><https://www.hackerrank.com/contests/utp-open-2018/challenges/how-many-inversions>

```

19:          $i \leftarrow i + 1$ 
20:     else
21:          $A[k] \leftarrow R[j]$ 
22:          $j \leftarrow j + 1$ 
23:          $inv \leftarrow inv + n_1 - i + 1$ 
24:     end if
25: end for
26: end function

```

```

27: function MERGESORT( $A[ ]$ ,  $p$ ,  $r$ )
28:     if  $p < r$  then
29:          $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
30:         MERGESORT( $A[ ]$ ,  $p$ ,  $q$ )
31:         MERGESORT( $A[ ]$ ,  $q + 1$ ,  $r$ )
32:         MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
33:     end if
34: end function

```

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

Please help *Humbertov Moralo* 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 specification:

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

The first line contains a positive integer  $n$  ( $1 \leq n \leq 10^6$ ), which represent the length of  $A$ .

The second line contains  $n$  space-separated positive integers which make up the array  $A$ , these values are in the closed interval  $[1, 10^8]$ .

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

#### Output specification:

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

#### Example input:

```

5
3 2 8 1 6
5
5 4 3 2 1
1
10
0

```

#### Example output:

```

5
10
0

```

## Solución del reto “How Many Inversions?”

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647
#define MAXT 1000000

unsigned long long int inv = 0;

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, 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 += n1 - i + 1;
            /* inv = inv + n1 - i + 1; */
        }
    }
}

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[MAXT + 1], i, n;

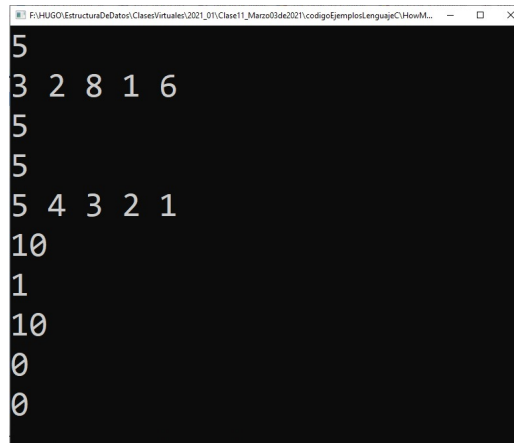
    while(scanf("%d", &n) && (n > 0))
    {
        inv = 0;
        for(i = 1; i <= n; i++)
            scanf("%d", &A[i]);
    }
}
```

```

    MergeSort(A, 1, n);
    printf("%llu\n", inv);
}

return 0;
}

```



```

5
3 2 8 1 6
5
5
5 4 3 2 1
10
1
10
0
0

```

Figura 6: Salida del programa para el reto “How Many Inversions?”.

## Programming Challenge: “How Many Sub Sets?”<sup>4</sup>

We have a set  $A$  of positive integers with cardinality  $N$  ( $|A| = N$ , 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  $S$ .

For example, if you have the following set  $A = \{6, 5, 1, 4, 2, 3\}$  and  $S = 7$ , then there is a total of 9 subsets of size two whose sum of its elements is less than or equal to 7, said subsets are:  $\{6, 1\}$ ,  $\{5, 1\}$ ,  $\{5, 2\}$ ,  $\{1, 4\}$ ,  $\{1, 2\}$ ,  $\{1, 3\}$ ,  $\{4, 2\}$ ,  $\{4, 3\}$  and  $\{2, 3\}$ .

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  $S$ .

### Input specification:

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  $n$  ( $1 \leq n \leq 5 \cdot 10^5$ ) and  $q$  ( $1 \leq q \leq 50$ ), which represent respectively the cardinality of the set  $A$  and the total of queries that will be made on the set  $A$ . The next line contains exactly  $n$  space-separated positive integer numbers  $A_1, A_2, A_3, \dots, A_n$  ( $1 \leq A_i \leq 10^8$ , for  $1 \leq i \leq n$ ), it is obviously guaranteed that the  $n$  elements of the set  $A$  are different. The next line contains exactly  $q$  space-separated positive integer numbers  $S_1, S_2, S_3, \dots, S_q$  ( $1 \leq S_j \leq 2 \cdot 10^8$ , for  $1 \leq j \leq q$ ), for the queries.

### Output specification:

Your program should print  $q$  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  $S$ .

<sup>4</sup><https://www.hackerrank.com/contests/utp-open-2018/challenges/how-many-sub-sets>

Example input:

```
6 3
6 5 1 4 2 3
7 8 12
```

Example output:

```
9
11
15
```

## Solución “ingenua” del reto “How Many Sub Sets?”

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXN 500000

int main()
{
    int n, q, A[MAXN + 1], s, result;
    int idElement, idQuery, i, j;

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

    for(idElement = 1; idElement <= n; idElement++)
        scanf("%d", &A[idElement]);

    for(idQuery = 1; idQuery <= q; idQuery++)
    {
        result = 0;
        scanf("%d", &s);
        for(i = 1; i < n; i++)
        {
            for(j = i + 1; j <= n; j++)
            {
                if((A[i] + A[j]) <= s)
                    result++;
            }
        }
        printf("%d\n", result);
    }
    return 0;
}
```

Esta solución genera resultados correctos para el reto, pero ... la complejidad computacional es muy alta!, el fragmento de código que contiene los ciclos anidados:

```
    for(i = 1; i < n; i++)
    {
        for(j = i + 1; j <= n; j++)
        {
            if((A[i] + A[j]) <= s)
                result++;
        }
    }
```

genera la siguiente cantidad de operaciones:

- cuando  $i = 1$  el ciclo anidado se ejecuta 499,999 veces
- cuando  $i = 2$  el ciclo anidado se ejecuta 499,998 veces
- cuando  $i = 3$  el ciclo anidado se ejecuta 499,997 veces



- :

- cuando  $i = 499,999$  el ciclo anidado se ejecuta una (1) vez

por lo tanto el doble ciclo `for` ejecuta un total de operaciones de:

$$\begin{aligned}
 499,999 + 499,998 + 499,997 + 499,996 + \dots + 1 &= 1 + 2 + 3 + 4 + \dots + 499,999 \\
 &= \frac{499,999 \cdot (500,000)}{2} \\
 &= 124,999,750,000 \\
 &= O(10^{11})
 \end{aligned}$$

Una complejidad en el peor de los casos de  $O(10^{11})$  para cada consulta en este reto de programación es inaceptable!, no es aceptable tener que poner a “correr” dicha solución por no menos de dos hora para que pueda generar todas las salidas para el archivo de entrada de los casos de prueba. Estamos en la obligación de seguir trabajando en una solución que en la complejidad del tiempo de ejecución sea mejor.

## Solución “ingenua mejorada” del reto “How Many Sub Sets?”

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647
#define MAXN 500000

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, 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) >> 1; /* q = (p + r) / 2 */
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }
}

int main()
{
    int n, q, A[MAXN + 1], s, result;
    int idElement, idQuery, i, j;

    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++)
    {
        result = 0;
        scanf("%d", &s);
        for(i = 1; i < n; i++)
        {
            for(j = i + 1; j <= n; j++)
            {
                if((A[i] + A[j]) <= s)
                    result++;
                else
                    break;
            }
        }
        printf("%d\n", result);
    }
    return 0;
}

```

Esta solución genera resultados correctos para el reto, pero ... la complejidad computacional sigue siendo muy alta!, el fragmento de código que contiene los ciclos anidados:

```

        for(i = 1; i < n; i++)
        {
            for(j = i + 1; j <= n; j++)
            {
                if((A[i] + A[j]) <= s)
                    result++;
                else
                    break;
            }
        }
    }
}

```

sigue teniendo una complejidad en el peor de los casos por consulta de  $O(10^{11})$  independientemente de tener actualmente los elementos en el arreglo ordenados de forma ascendente, esto se presenta cuando  $S = 2 \cdot 10^8$  donde se garantiza que cualquier subconjunto de tamaño dos en el conjunto  $A$  tendrá una suma de sus elementos menor que dicha cantidad, y donde el orden del total de subconjuntos de tamaño dos es  $O(10^{11})$ . De nuevo estamos en la obligación de plantear una solución que en la complejidad del tiempo de ejecución sea del orden de  $O(10^7)$  u  $O(10^8)$ .

## Solución eficiente en tiempo de ejecución para el reto “How Many Sub Sets?”

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647
#define MAXN 500000

```

```

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, 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) >> 1;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }
}

/*****
/* El valor devuelto por la funcion es un entero correspondiente al indice del elemento */
/* que coincide con el valor buscado. Si el valor buscado no se encuentra, entonces el */
/* valor devuelto es: - (punto de insercion) - 1. El valor del punto de insercion es el */
/* indice del elemento del arreglo donde deberia encontrarse el valor buscado.      */
/* La expresion: "- (punto de insercion) - 1" garantiza que el indice devuelto sera */
/* mayor o igual que cero solo si el valor buscado es encontrado.                  */
*****/

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;
    while(i <= j)
    {
        m = (i + j) >> 1; /* m = (i + j)/2; */
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])

```

```

        i = m + 1;
    else
        j = m - 1;
    }
}
if(result == -1)
    result = (-1) * i - 1;

return result;
}

int main()
{
    int n, q, A[MAXN + 1], s, index, element;
    int idElement, idQuery, i;
    unsigned 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++)
    {
        result = 0;
        scanf("%d", &s);
        for(i = 1; i < n; i++)
        {
            element = s - A[i];
            if(element > A[i])
            {
                index = BinarySearch(A, i + 1, n, element);
                if(index < 0)
                    index = (-1 * index) - 2;

                result += (index - i);
            }
            else
                break;
        }
        printf("%llu\n", result);
    }
    return 0;
}

```

Esta solución además de generar resultados correctos para el reto también cuenta con una complejidad en tiempo de ejecución aceptable. El mayor costo computacional se presenta en el fragmento de código que contiene el ciclo en el cual se hace el llamado a la búsqueda binaria:

```

    for(i = 1; i < n; i++)
    {
        element = s - A[i];
        if(element > A[i])
        {
            index = BinarySearch(A, i + 1, n, element);
            if(index < 0)
                index = (-1 * index) - 2;

            result += (index - i);
        }
        else
            break;
    }
}

```

La búsqueda binaria tiene una complejidad de  $O(\log n)$  y se ejecuta  $n$  veces en el ciclo de repetición, por lo tanto la complejidad en tiempo de ejecución es  $O(n \log n)$ . La cantidad de operaciones es del orden de  $(5 \cdot 10^5) \log(5 \cdot 10^5)$ ,

donde  $(5 \cdot 10^5) \log(5 \cdot 10^5) \leq (5 \cdot 10^5) \log(10^6) = (5 \cdot 10^5) \cdot 6 = (6 \cdot 5) \cdot 10^5 = 30 \cdot 10^5 = 3 \cdot 10 \cdot 10^5 = 3 \cdot 10^6$ .

Como se realiza un máximo de 50 consultas, entonces la complejidad en tiempo de ejecución de la solución es  $O(q \cdot (n \log n)) = O((5 \cdot 10) \cdot (3 \cdot 10^6)) = O((5 \cdot 3) \cdot (10 \cdot 10^6)) = O(15 \cdot (10^7)) = O(10^8)$ .