

### 3.1. Introducción.

Ordenación y búsqueda son dos temas fundamentales en programación, y por esto son ampliamente estudiados. Esto se debe a dos razones: la primera, es que muchos problemas pueden ser resueltos de manera más eficiente si los datos están ordenados, y muchos otros necesitan ordenar o buscar datos como pasos intermedios; la segunda razón es que los algoritmos de ordenación utilizan una gran variedad de técnicas de programación, por lo cual son útiles para aprenderlas.

Al ser tan estudiados, existe gran diversidad de algoritmos, y la elección de cual es mejor depende en gran medida de cual va a ser su aplicación. Algunos factores que afectan a la elección son: la cantidad de datos, en que orden los obtenemos, posibles limitantes de los datos (principalmente el tamaño), restricciones de memoria, tipo de almacenamiento que utilicemos, si los datos los podemos procesar en paralelo, etc.

Nosotros no debemos preocuparnos mucho por estas limitantes, ya que dentro del ICPC muchos de estos factores ya están determinados. Podemos considerar que nuestros programas correrán dentro de una PC convencional, y que los datos estarán alojados en RAM. Otra limitante, que no es del concurso pero que vamos a agregar, es que los datos los guardaremos dentro de un arreglo.

Una de las definiciones más comunes de ordenación es la siguiente: dada una lista de datos (generalmente números), encontrar la forma de acomodarlos de forma no decreciente, es decir, de menor a mayor. De manera formal lo podemos definir como: dado un conjunto de  $n$  elementos  $a_1, a_2, \dots, a_n$ , encontrar una permutación tal que  $a_1 \leq a_2 \leq \dots \leq a_n$ . Aunque generalmente se ordena de forma no decreciente, también se puede ordenar un arreglo de forma no creciente de forma que la permutación de sus elementos sea  $a_1 \geq a_2 \geq \dots \geq a_n$ .

Decimos que una ordenación es estable cuando datos que tienen el mismo valor de comparación, permanecen en la misma secuencia después de ordenar el arreglo. Esto es particularmente útil cuando queremos ordenar datos con varios campos. Por ejemplo: si deseamos ordenar un arreglo que contiene fechas, ordenando primero por días, luego por meses y después por años, si el algoritmo es estable las fechas terminarían ordenadas correctamente, mientras que si no lo es, los años estarían ordenados pero muy posiblemente los meses y los días no.

La mayoría de los algoritmos de ordenación se basan en la comparación entre datos para después acomodarlos. Los más sencillos e intuitivos son de orden  $O(n^2)$ , y son útiles cuando necesitamos ordenar pocos datos o cuando la ordenación no es una parte crítica en el programa.

El más común y utilizado de estos algoritmos es, probablemente, la ordenación por burbuja (*bubblesort*). El inconveniente de este método, es que es uno de los más lentos. Por esto, si vamos a recurrir a un algoritmo  $O(n^2)$  para ordenar, es mejor utilizar ordenación por inserción (*insertionsort*), que a pesar de ser del mismo orden, realiza menos operaciones y es casi igual de intuitivo.

Existen ordenaciones más veloces. Una de ellas es la Ordenación Rápida de Hoare (*quicksort*), la cuál se basa en técnicas de “divide y vencerás”. Esta ordenación es una de las más rápidas y utilizadas, inclusive C ya tiene una implementación que podemos usar. El problema que tiene es que, a pesar de tener cota  $\Theta(n \lg n)$ , su peor caso es  $O(n^2)$ . Otros algoritmos garantizan ser  $O(n \lg n)$ , como el caso de la ordenación por mezcla (*mergesort*) y por montículos (*heapsort*), y aunque en promedio son más lentos, no tenemos problemas con el peor caso.

Está demostrado que mediante comparaciones, la cota más baja que podemos obtener es  $\Omega(n \lg n)$ . Pero si no utilizamos comparaciones y sabemos algunos datos adicionales, podemos obtener cotas más bajas. Un ejemplo de esto es la ordenación por conteo, que puede ordenar datos enteros en  $O(n)$  si se conoce el rango de dichos números.

Podemos definir búsqueda como el acto de revisar un conjunto de datos hasta encontrar el que nos interesa o averiguar que el dato no existe en conjunto. De manera formal: dado un conjunto de  $n$  elementos  $a_1, a_2, \dots, a_n$  y un elemento  $d$  a buscar, encontrar una  $i$  ( $1 \leq i \leq n$ ) tal que  $a_i = d$ , o determinar que no existe tal  $i$ .

El mejor algoritmo de búsqueda dependerá de la forma en que estén arreglados los datos. Si los datos no se encuentran en orden, necesitamos buscar todo el arreglo hasta encontrar la solución. En caso contrario, podemos utilizar un algoritmo conocido como búsqueda binaria que encuentra los datos en  $O(\lg n)$ .

### 3.2. Ordenación por inserción. (*Insertionsort*)

Esta ordenación es, junto con la de burbuja, una de las más conocidas, pero a diferencia de la primera, es de las más eficientes en  $O(n^2)$ . Su funcionamiento es el siguiente: Si sólo tenemos un dato, entonces el arreglo ya está ordenado. Agregando un segundo dato, podemos ordenar el arreglo acomodando este dato antes o después del primero, según corresponda. Para el tercero, vemos si queda antes del primero, del segundo o en su posición actual. En general, si ya tenemos un arreglo ordenado y queremos agregar un nuevo dato, basta con acomodarlo antes de los datos que son mayores a él.

## Flip Sort

Sorting in computer science is an important part. Almost every problem can be solved efficiently if sorted data are found. There are some excellent sorting algorithms, which have already achieved the lower bound  $n \lg n$ . In this problem we will also discuss about a new sorting approach. In this approach only one operation (Flip) is available and that is you can exchange two adjacent terms. If you think a while, you will see that it is always possible to sort a set of numbers in this way.

### Problem

A set of integers will be given. Now using the above approach we want to sort the numbers in ascending order. You have to find out the minimum number of flips required. Such as to sort "1 2 3" we need no flip operation whether to sort "2 3 1" we need at least 2 flip operations.

### Input

The input will start with a positive integer  $n$  ( $n \leq 1000$ ). In next few lines there will be  $n$  integers. Input will be terminated by EOF.

### Output

For each data set print "Minimum exchange operations :  $m$ " where  $m$  is the minimum flip operations required to perform sorting. Use a separate line for each case.

Valladolid: 10327

**Solución:** Nos están indicando que la ordenación se va a realizar por medio de "flips" (cambios). Sabemos que la ordenación que se basa en cambios es la de por burbuja (como se explico en el ejemplo del capítulo 2), por lo que sería la primera opción que tomaríamos.

Sin embargo, también podemos utilizar inserción. Cuando vayamos a ingresar un nuevo dato al arreglo ya ordenado, sólo necesitamos contar cuantos valores son mayores a él y esta será la cantidad de cambios que son requeridos.

*Nota:* Existe una manera de contar los cambios sin necesidad de ordenar los datos. Para hacerlo, sólo necesitamos recorrer el arreglo y contar cuantos elementos que ya recorrimos son menores al actual. No lo haremos así por fines didácticos.

```

01: Var
02:   i,m,n: longint;
03:   a: array [1..1000] of longint;
04:
05: Procedure InsertionSort;
06:   var i,j,temp: longint;
07:   begin
08:     for i:= 2 to n do
09:       for j:= i downto 2 do
10:         if (a[j-1]> a[j]) then begin
11:           temp:= a[j-1];
12:           a[j-1]:= a[j];
13:           a[j]:= temp;
14:           Inc(m);
15:         end
16:         else break;
17:       end;
18:

```

```

19: Begin
20:     While (not eof(input)) do
21:         begin
22:             readln(input,n);
23:             for i:= 1 to n do
24:                 read(input,a[i]);
25:             readln(input);
26:             m:= 0;
27:             InsertionSort;
28:             writeln(output, 'Minimum exchange operations : ',m);
29:         end;
30: End.

```

En las primeras 3 líneas tenemos la declaración de variables: **i** una variable para ciclos, **n** es la cantidad de datos, **m** es la cantidad mínima de intercambios, y **a** es el arreglo donde se van a guardar los datos.

De la línea 5 a la 7 es donde se ejecuta el ordenamiento por inserción. Primero definimos las funciones **i** y **j** que usaremos en ciclos, y **temp** que es una variables temporal para el intercambio de datos. El ciclo en la línea 8 lo utilizamos para insertar el **i**-ésimo dato a la parte del arreglo que ya se encuentra ordenada. Con el siguiente ciclo (línea 9) recorremos la parte que ya está ordenada y movemos el nuevo dato hasta la posición que le corresponde. En la línea 14 contamos cuantos cambios hicimos. Esta línea no es parte del algoritmo y la podemos quitar al utilizarlo en otros programas.

Después tenemos la parte principal del programa. La línea 20 hace que el programa se quede en un ciclo mientras no se llegue al fin del archivo. De la línea 22 a la 25 leemos el arreglo, empezamos leyendo la cantidad de datos en el arreglo, después leemos los datos y terminamos leyendo el fin de línea (es importante para poder encontrar el fin de archivo). En la línea 26 inicializamos el contador de cambios a cero, después mandamos ordenar los datos e imprimimos la cantidad mínima de cambios.

La implementación anterior utiliza cambios para llevar el nuevo dato hasta su posición. Tiene la ventaja de que el código queda corto, y va de acuerdo con lo que pide el problema. Aún así, se están realizando operaciones innecesarias. Podemos cambiar código para que se ejecute más rápido.

```

01: Procedure InsertionSort;
02:     var i,j,temp: longint;
03:     begin
04:         for i:= 2 to n do
05:             begin
06:                 j:= i-1;
07:                 temp:= a[i];
08:                 while (temp< a[j]) and (j>=1) do
09:                     begin
10:                         a[j+1]:= a[j];
11:                         Dec(j);
12:                         Inc(m);
13:                     end;
14:                 a[j+1]:= temp;
15:             end;

```

```
16:      end;
```

Esta es la implementación más común de la ordenación por inserción. Las variables son las mismas y tienen el mismo uso que en la implementación anterior. En la línea 4 buscamos insertar el  $i$ -ésimo dato en el arreglo ordenado. Ahora lo que hacemos es copiar el dato a insertar en la variable temporal (línea 7) y recorrer todos los valores que sean de menor tamaño un lugar hacia arriba (ciclo entre las líneas 8 y 13). Al final, acomodamos el valor en su nueva casilla (línea 14). Al igual que con la implementación anterior, el incremento de  $m$  (línea 12) no es parte del algoritmo y lo podemos quitar al reutilizar el código.

**Ejemplo:**

Entrada:	6 1 3 6 2 5 4 1 3 6 2 5 4 1 3 6 2 5 4 1 3 6 2 5 4 1 3 6 2 5 4 1 3 6 2 5 4 ← m = 1 1 3 2 6 5 4 ← m = 2
Desarrollo:	1 2 3 6 5 4 1 2 3 6 5 4 ← m = 3 1 2 3 5 6 4 1 2 3 5 6 4 ← m = 4 1 2 3 5 4 6 ← m = 5 1 2 3 4 5 6
Salida:	Minimum exchange operations : 5

**3.3. Ordenación Rápida de Hoare. (*Quicksort*)**

A pesar de que el peor caso de este algoritmo es  $O(n^2)$ , es uno de los más empleados. Esto se debe a que su implementación es compacta, por lo que las constantes del tiempo de ejecución son muy bajas. Además, su peor caso ocurre muy pocas veces en aplicaciones de propósito general.

Este algoritmo utiliza la técnica “Divide y Vencerás” (sección 11.7) para ordenar los datos. Lo que hacemos es escoger un elemento en la lista, al cual llamaremos pivote, y ordenamos los datos en dos subarreglos, dejando los datos que son mayores a este elemento en uno y los menores en el otro. Una vez que hayamos hecho esto, aplicamos el mismo procedimiento en los subarreglos, y repetimos este procedimiento hasta que todos los datos se encuentren ordenados.

El peor caso de esta ordenación ocurre cuando en cada recursión, el pivote queda como único elemento de un subarreglo. Si esto sucede, tomaríamos todos los elementos de la lista como pivotes en algún momento, y cada que hiciéramos esto, recorreríamos todo el arreglo creando los subarreglos, por lo que tenemos la cota de  $O(n^2)$ .

Tomando como pivote el primer dato, el peor caso se presenta únicamente cuando el arreglo ya está ordenado, ya sea en forma no decreciente o no creciente. Para minimizar esto, hay versiones de la ordenación rápida que permutan aleatoriamente la entrada antes de ordenarla, por lo que ninguna entrada en especial ocasiona el peor caso (aunque todavía puede ocurrir).

La mejor elección del pivote es la mediana del vector, pero encontrarla implica un tiempo extra que vuelve al algoritmo ineficiente. Si conocemos la naturaleza de los datos a ordenar, podemos escoger los datos de acuerdo a esto. Como generalmente no la conocemos, podemos tomar cualquier elemento como pivote. Otra implementación común es tomar tres elementos aleatorios y utilizar como pivote la mediana de estos tres.

Otra mejora que se puede aplicar al algoritmo es utilizar otro método para ordenar los subarreglos una vez que sean de tamaño pequeño. Por ejemplo, si los subarreglos son de tamaño menor a 10 podríamos mandarlos a ordenar mediante inserción. El valor de 10 es meramente demostrativo, y en la práctica lo podríamos obtener a prueba y error.

## Problem 22

### Problem

Using [names.txt](#), a 46K text file containing over five thousand first names, begin by sorting it into alphabetical order. Then working out the alphabetical value for each name, multiply this value by its alphabetical position in the list to obtain a name score.

For example, when the list is sorted into alphabetical order, COLIN, which is worth  $3 + 15 + 12 + 9 + 14 = 53$ , is the 938<sup>th</sup> name in the list. So, COLIN would obtain a score of  $938 \times 53 = 49714$ .

What is the total of all the name scores in the file?

*Project Euler: 22*

**Solución:** El algoritmo para obtener el valor de cada nombre es simple, sólo que necesitamos tener los nombres en orden. Aunque lo podemos ordenar con casi cualquier método, escogemos *Quicksort* ya que la entrada es relativamente grande (5000 nombres).

```

01:  Var
02:    ch: char;
03:    i,j,n,r,t: longint;
04:    a: array [1..10000] of string;
05:
06:  Procedure Quicksort(l,r: longint);
07:    var i,j: longint;
08:        x,t: string;
09:    begin
10:      i:= l; j:= r;
11:      x:= a[(l+r) div 2];
12:      repeat
13:        while (a[i]<x) do Inc(i);
14:        while (x<a[j]) do Dec(j);
15:        if (i<=j) then
16:          begin
17:            t:= a[i]; a[i]:= a[j]; a[j]:= t;

```

```

18:         Inc(i); Dec(j);
19:         end;
20:         until (i>j);
21:         if (l<j) then quicksort(l,j);
22:         if (i<r) then quicksort(i,r);
23:         end;
24:
25: Begin
26:     assign(input, 'names.txt'); reset(input);
27:     assign(output, 'out.txt'); rewrite(output);
28:     n:= 0;
29:     while (not eof(input)) do
30:         begin
31:             Inc(n);
32:             a[n]:= '';
33:             read(input, ch);
34:             while (ch<>',' and (not eof(input))) do
35:                 begin
36:                     if (ch<>'') then a[n]:= a[n]+ch;
37:                     read(input, ch);
38:                 end;
39:             end;
40:             quicksort(1,n);
41:             r:= 0;
42:             for i:= 1 to n do
43:                 begin
44:                     t:= 0;
45:                     for j:= 1 to length(a[i]) do
46:                         Inc(t, ord(a[i,j])-64);
47:                     Inc(r, t*i);
48:                 end;
49:             writeln(output, r);
50:             close(output);
51: End.

```

Empezamos declarando en las primeras cuatro líneas las variables a utilizar. La lectura de los nombres la haremos caracter por caracter mediante **ch**, y guardamos los nombres en el arreglo **a**. Para los ciclos empleamos **i** y **j**, **n** para el total de datos, **t** es una variable temporal, y en **r** guardamos el resultado.

Enseguida tenemos la función **quicksort**, que ordena un arreglo entre las casillas **l** y **r**. Las variables **i** y **j** son para recorrer el arreglo, **x** es el pivote y **t** es una variable temporal. En las líneas 10 y 11 inicializamos las variables. Entre la 13 y la 14 recorremos el arreglo buscando elementos que no estén acomodadas con respecto al pivote. Una vez que los encontremos, los cambiamos de lugar y continuamos el proceso. Esto lo repetimos hasta que las variables con las que recorremos el arreglo se crucen. Terminamos revisando la posición de las variables y mandando a ordenar los subarreglos en caso que sea necesario.

Algo importante es que el pivote lo escogimos como la casilla media del arreglo (que no es la mediana del arreglo), por lo que el peor caso no se daría cuando estuvieran ordenados los datos, sino con entradas en las que el menor o el mayor elemento se encontraran siempre a la mitad del subarreglo.

Al final, tenemos entre las líneas 25 y 51 la parte principal de código. Para manejar los archivos de entrada y salida, en línea 26 asignamos el archivo *names.txt* como el de entrada, en la 27 *out.txt* como el de salida, y en la 50 cerramos el archivo de salida. El archivo de entrada no es necesario cerrarlo (aunque podemos hacerlo), pero si no cerramos el archivo de salida, es posible que no se escriban todos los datos.

Entre las líneas 28 y 39 leemos los nombres. Primero inicializamos el total a cero (línea 28) y mientras no se termine el archivo seguimos leyendo. Cada que vamos a leer otro nombre, incrementamos el total y limpiamos cadena donde lo guardaremos (líneas 31 y 32). Cada nombre lo leemos hasta llegar a una coma, excepto el último que es hasta el fin de archivo. En la línea 36 omitimos las comillas, ya que no las necesitamos.

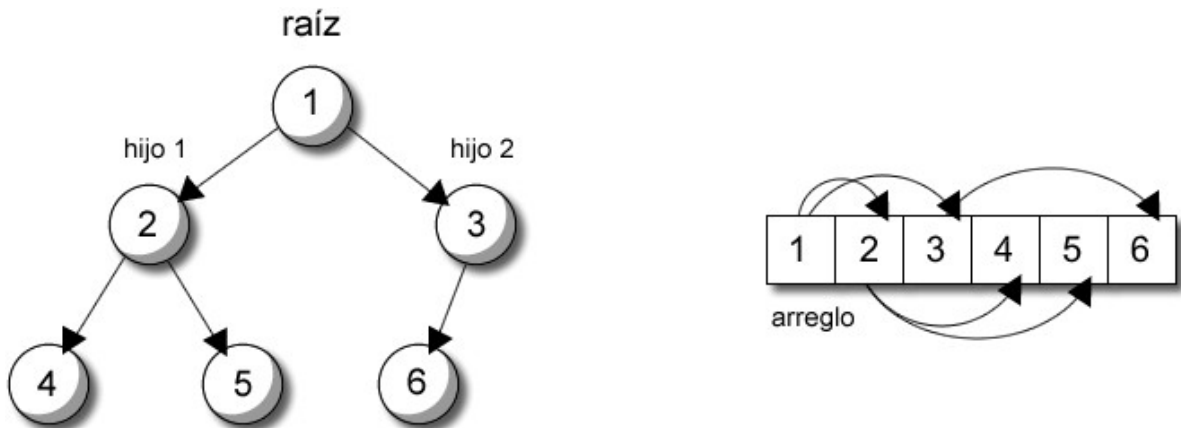
Mandamos ordenar los datos, e inicializamos el resultado. Por cada nombre, sumamos cada una de las letras (líneas 45 y 46) e incrementamos el resultado por el producto de la suma y la posición del nombre. En la línea 46 restamos 64 ya que *ord* nos devuelve el valor ASCII del caracter. Terminamos imprimiendo la respuesta en la línea 49.

**Ejemplo:**

Entrada:	"MARY", "PATRICIA", "LINDA", "BARBARA", "ELIZABETH", "JENNIFER", "MARIA", "SUSAN"
	MARY PATRICIA LINDA BARBARA ELIZABETH JENNIFER MARIA SUSAN
	<b>MARY</b> PATRICIA LINDA <b>BARBARA</b> ELIZABETH JENNIFER MARIA SUSAN
	BARBARA PATRICIA LINDA MARY ELIZABETH JENNIFER MARIA SUSAN
	<b>PATRICIA</b> LINDA MARY <b>ELIZABETH</b> JENNIFER MARIA SUSAN
	ELIZABETH LINDA MARY PATRICIA JENNIFER MARIA SUSAN
	LINDA MARY <b>PATRICIA</b> JENNIFER <b>MARIA</b> SUSAN
	LINDA MARY MARIA JENNIFER <i>PATRICIA</i> SUSAN
	LINDA <b>MARY</b> MARIA <b>JENNIFER</b>
	LINDA JENNIFER MARIA <i>MARY</i>
	<b>LINDA JENNIFER</b> MARIA
	<i>JENNIFER</i> LINDA MARIA
	<i>LINDA</i> MARIA
Desarrollo:	<i>PATRICIA</i> SUSAN
	BARBARA ELIZABETH JENNIFER LINDA MARIA MARY PATRICIA SUSAN
	i t r
	1 43 43
	2 88 219
	3 81 462
	4 40 622
	5 42 832
	6 57 1174
	7 77 1713
	8 74 2305
Salida:	2305

### 3.4. Ordenación por montículos. (*Heapsort*)

La ordenación por montículos toma su nombre de las estructuras conocidas como montículos, que son una clase especial de árboles (sección 9.1). Este tipo de árboles poseen la característica de que son binarios completos, lo que significa que cada vértice tiene a lo más dos hijos, y que todos sus niveles están completos, excepto (posiblemente) el último, el cuál se llena de izquierda a derecha. Otra particularidad que deben cumplir es que todo vértice debe ser igual o mayor a cualquiera de sus hijos. En consecuencia, el elemento mayor siempre los podemos encontrar en la raíz.



La ordenación por montículos utiliza la propiedad de la raíz para ordenar el arreglo. Una vez que el arreglo cumpla las propiedades del montículo, quitamos la raíz y la colocamos al final del arreglo. Con los datos que sobran, creamos otro montículo y repetimos hasta que todos los datos estén ordenados.

#### Sequence Median

Given a sequence of  $n$  nonnegative integers. Let's define the median of such sequence. If  $n$  is odd the median is the element with stands in the middle of the sequence after it is sorted. One may notice that in this case the median has position  $\frac{n+1}{2}$  in sorted sequence if sequence elements are numbered starting with 1. If  $n$  is even then the median is the semi-sum of the two "middle" elements of sorted sequence. I.e. semi-sum of the elements in positions  $\frac{n}{2}$  and  $\frac{n}{2} + 1$  of sorted sequence. But original sequence might be unsorted.

#### Problem

Your task is to write program to find the median of given sequence.

#### Input

The first line of input contains the only integer number  $n$  - the length of the sequence. Sequence itself follows in subsequent lines, one number in a line. The length of the sequence lies in the range from 1 to 250000. Each element of the sequence is a positive integer not greater than  $2^{32} - 1$  inclusive.

**Output**

You should print the value of the median with exactly one digit after decimal point.

Ural: 1306

**Solución:** Encontrar la mediana en un arreglo ordenado no es complicado, pero las limitantes en memoria nos impiden guardar el arreglo completo. Una solución es guardar únicamente uno más de la mitad de los datos y cada que leamos uno nuevo, si es menor al más grande del arreglo, sacamos el mayor y guardamos el que leímos.

```

01: Var
02:   x,y: extended;
03:   i,m,n,t: longint;
04:   a: array [1..125001] of longint;
05:
06: Procedure Max_heapify(i,n: longint);
07:   var l,r,max,temp: longint;
08:   begin
09:     l:= 2*i; r:= 2*i+1;
10:     if (l<= n) and (a[l]> a[i]) then max:= l
11:                                     else max:= i;
12:     if (r<= n) and (a[r]>a[max]) then max:= r;
13:     if (max<>i) then
14:       begin
15:         temp:= a[i]; a[i]:= a[max]; a[max]:= temp;
16:         Max_heapify(max,n);
17:       end;
18:   end;
19:
20: Procedure Build_max_heap(n: longint);
21:   var i: longint;
22:   begin
23:     for i:= n div 2 downto 1 do
24:       max_heapify(i,n);
25:   end;
26:
27: Procedure Heapsort(n: longint);
28:   var i,temp: longint;
29:   begin
30:     build_max_heap(n);
31:     for i:= n downto 2 do
32:       begin
33:         temp:= a[i]; a[i]:= a[1]; a[1]:= temp;
34:         max_heapify(1,i-1);
35:       end;
36:   end;
37:
38: Begin
39:   readln(input,n);
40:   m:= (n div 2) +1;
41:   for i:= 1 to m do
42:     readln(input,a[i]);
43:   build_max_heap(m);
44:   for i:= m+1 to n do
45:     begin
46:       readln(input,t);
47:       if (t<a[1]) then a[1]:= t;
48:       max_heapify(1,m);

```

```

49:         end;
50:     heapsort(m);
51:     x:= a[m];
52:     if odd(n) then y:= a[m]
53:         else y:= a[m-1];
54:     writeln(output, (x+y)/2:0:1);
55: End.

```

De la línea 1 a la 4 definimos las variables a emplear. El entero **i** lo utilizamos para los ciclos, **n** es el total de datos, **m** es la mitad más uno y **t** es un entero temporal. En el arreglo **a** guardamos los datos a utilizar, y las variables **x** e **y** son temporales que utilizamos para calcular la mediana ya que el tamaño de los datos puede exceder la capacidad de un *longint*.

La función **max\_heapify** (líneas 6 a 18) se encarga de hacer que un elemento **i** en un arreglo de tamaño **n** cumpla la condición de los vértices, esto es, que el padre sea mayor que los hijos. En la línea 9 obtenemos cuales son los hijos del vértice. Es fácil notar que el hijo izquierdo de un vértice en la posición **i** es el que se encuentra en la posición  $2i$  y el hijo derecho es el que le sigue. En las líneas 10 a 12 encontramos cual de estas casillas contiene el mayor elemento. Si el mayor elemento no es el padre, entonces intercambiamos el padre con el mayor elemento (línea 15), cumpliéndose así la condición del montículo. Revisamos recursivamente si el elemento que cambiamos cumple con la propiedad del montículo en su nueva posición (línea 16).

Después contamos con la función **build\_max\_heap** (líneas 20 a 25) la cual crea un montículo dentro del arreglo. Lo primero que debemos notar es que podemos crear el montículo utilizando el procedimiento **max\_heapify** del último elemento al primero. Esto es porque al utilizar el **i**-ésimo elemento, sabemos que todos los demás elementos que ramifican de él ya cumplen con las propiedades del montículo, excepto (posiblemente) **i**. Otra cosa que podemos notar es que los datos que están en el último nivel (desde  $n+1$  hasta **n**), no tienen hijos, por lo que podemos ignorarlos.

Los datos los ordenamos a través de la función **heapsort** (líneas 27 a 36). Empezamos creando el montículo en la línea 30. El mayor elemento (la raíz), lo mandamos a su posición ordenada (línea 33). Al intercambiar las casillas, tenemos que hacer que los datos restantes sigan cumpliendo las propiedades del montículo. Como el único elemento que no cumple es el que acabamos de cambiar, utilizamos **max\_heapify** en esta casilla. Así, en cada iteración, los últimos datos ya están ordenados.

La parte principal del código está entre las líneas 38 y 55. Empezamos leyendo la cantidad de datos en **n** (línea 39) y leemos la primera mitad mas uno de los datos (líneas 40 a 42). Teniendo estos datos, construimos el montículo (línea 43). Para los siguientes datos, cada que leemos uno, revisamos si es más chico que la raíz (línea 47), y de ser así, los cambiamos y acomodamos el elemento que acabamos de agregar. Con esto nos aseguramos que en el arreglo contamos con los **m** elementos menores. Una vez que tenemos ordenados los datos (línea 50), la mediana la encontramos en la posición **m** si la cantidad de datos es impar o en el promedio de la **m** y la **m-1** si es par. Utilizamos variables *extended* para evitar que los datos se desborden al realizar la suma (podemos utilizar cualquier otra variable mayor a *longint*, como *int64*).



### 3.5. Ordenación por conteo. (*Countingsort*)

El ordenamiento por conteo tiene la particularidad de que en ningún momento necesitamos realizar comparaciones entre los números, por lo que la cota mínima de  $O(n \lg n)$  no aplica.

Este método tiene la restricción de que sólo puede ser aplicado en números naturales en el rango de 1 a  $k$ . Su cota de tiempo es de  $O(n + k)$ , por lo que para ser mejor que el ordenamiento a base de comparaciones, se debe cumplir que  $k < n \lg n$ . Por esto, este algoritmo es útil principalmente cuando contamos con muchos datos en un rango corto.

Para ordenar un arreglo mediante este método, contamos en un arreglo auxiliar cuantas veces se encuentra cada número en el arreglo original. Una vez que tengamos las veces que aparecen, podemos saber cuantos números son menores sumando las casillas anteriores. Ahora que conocemos la cantidad de datos que son menores, podemos acomodar el dato en la casilla que le corresponde.

#### Questions and Answers

The database of the Pentagon contains top-secret information. We don't know what the information is — you know, it's top-secret, — but we know the format of its representation. It is extremely simple. We don't know why, but all the data is coded by the natural numbers from 1 up to 5000. The size of the main base (we'll denote it be  $n$ ) is rather big — it may contain up to 100 000 those numbers. The database is to process quickly every query. The most often query is: "Which element is  $i$ -th by its value?"— with  $i$  being a natural number in a range from 1 to  $n$ .

#### Problem

Your program is to play a role of a controller of the database. In the other words, it should be able to process quickly queries like this.

#### Input

The standard input of the problem consists of two parts. At first, a database is written, and then there's a sequence of queries. The format of database is very simple: in the first line there's a number  $n$ , in the next  $n$  lines there are numbers of the database one in each line in an arbitrary order. A sequence of queries is written simply as well: in the first line of the sequence a number of queries  $k$  ( $1 \leq k \leq 100$ ) is written, and in the next  $k$  lines there are queries one in each line. The query "Which element is  $i$ -th by its value?" is coded by the number  $i$ . A database is separated from a sequence of queries by the string of three symbols "#".

#### Output

The output should consist of  $k$  lines. In each line there should be an answer to the corresponding query. The answer to the query " $i$ " is an element from the database, which is  $i$ -th by its value (in the order from the least up to the greatest element).

*Ural: 1026*

**Solución:** Teniendo los datos ordenados, obtener el  $i$ -ésimo dato es sencillo. Como la cantidad de datos es grande (100 000) pero el rango no (1 a 5000), el ordenamiento por conteo es buena opción.

```

01: Var
02:   i,k,n,t: longint;
03:   a: array [1..100000] of longint;
04:
05: Procedure countsort(n: longint);
06:   var i,m: longint;
07:       b: array [1..100000] of longint;
08:       count: array [1..5000] of longint;
09:   begin
10:     m:= 0;
11:     fillchar(count, sizeof(count), 0);
12:     for i:= 1 to n do
13:       begin
14:         Inc(count[a[i]]);
15:         if (a[i]>m) then m:= a[i];
16:       end;
17:     for i:= 2 to m do
18:       Inc(count[i], count[i-1]);
19:     b:= a;
20:     for i:= n downto 1 do
21:       begin
22:         a[count[b[i]]]:= b[i];
23:         Dec(count[b[i]]);
24:       end;
25:     end;
26:
27: Begin
28:   readln(input,n);
29:   for i:= 1 to n do
30:     readln(input, a[i]);
31:   countsort(n);
32:   readln(input);
33:   readln(input,k);
34:   for i:= 1 to k do
35:     begin
36:       readln(input,t);
37:       writeln(output,a[t]);
38:     end;
39: End.

```

Las primeras tres líneas son donde declaramos las variables que utilizaremos. Para los ciclos usaremos **i**, **n** para la cantidad de datos en el arreglo y **k** para la cantidad de datos por buscar. En **t** leemos la posición del dato a buscar y **a** es el arreglo a ordenar.

De la línea 5 a la 25 tenemos la función **countsort** que es la que efectúa la ordenación por conteo. La variable **i** nos va a servir en los ciclos, en **m** guardamos el número máximo, **count** y **b** son arreglos auxiliares. Entre las líneas 10 y 16 contamos la aparición de cada número, y guardamos al mayor en **m**. El siguiente par de líneas (17 y 18), incrementa cada casilla de **count** en el valor anterior, por lo que al final tendremos en cada casilla cuantos números son menores o iguales.

Copiamos el arreglo original **a** en **b**. Por cada dato que tenemos en el arreglo, vemos en cuantos datos existen que son menores o iguales, y lo colocamos en esa posición. Como pueden existir varios datos con el mismo valor, cada vez que insertamos un dato, decrementamos la cantidad de datos que son menores o iguales. Recorremos el arreglo en reversa para asegurar que el ordenamiento sea estable.

La parte principal del código lo encontramos entre las líneas 27 y 39. Iniciamos leyendo los datos (líneas 28 a 30) y enseguida los ordenamos. Una vez ordenados, leemos **t** y buscamos el **t**-ésimo dato en el arreglo, esto lo repetimos **k** veces.

**Ejemplo:**

Entrada:	<pre> 10 5 1 4 9 7 9 6 4 9 8 ### 4 3 3 2 5 </pre>
Desarrollo:	<pre>       1 2 3 4 5 6 7 8 9 count 1 0 0 2 1 1 1 1 3 (cuenta de cada uno) count 1 1 1 3 4 5 6 7 10 (acumulado)  b          a          count 8, count[8]= 7   x x x x x x 8 x x x   1 1 1 3 4 5 6 6 10 9, count[9]= 10  x x x x x x 8 x x 9   1 1 1 3 4 5 6 6 9 4, count[4]= 3   x x 4 x x x 8 x x 9   1 1 1 2 4 5 6 6 9 6, count[6]= 5   x x 4 x 6 x 8 x x 9   1 1 1 2 4 5 6 6 9 9, count[9]= 9   x x 4 x 6 x 8 x 9 9   1 1 1 2 4 4 6 6 8 7, count[7]= 6   x x 4 x 6 7 8 x 9 9   1 1 1 2 4 4 5 6 8 9, count[9]= 8   x x 4 x 6 7 8 9 9 9   1 1 1 2 4 4 5 6 7 4, count[4]= 2   x 4 4 x 6 7 8 9 9 9   1 1 1 1 4 4 5 6 7 1, count[1]= 1   1 4 4 x 6 7 8 9 9 9   0 1 1 1 4 4 5 6 7 5, count[5]= 4   1 4 4 5 6 7 8 9 9 9   0 1 1 1 3 4 5 6 7 </pre>
Salida:	<pre> 4 4 4 6 </pre>

### 3.6. Búsqueda lineal (secuencial).

Esta búsqueda es directa y se basa en el uso de “fuerza bruta”. Si el dato que necesitamos buscar se encuentra dentro del arreglo, buscando en todas las casillas lo debemos encontrar. Para hacerlo de una manera organizada, empezamos por el primer dato y continuamos hasta llegar al dato que buscamos o hasta que lleguemos al fin del arreglo.

Un algoritmo tan simple puede parecer inútil, pero si no queremos ordenar los datos (o no hay forma de hacerlo) es la única opción que tenemos. En general, ordenar los datos nos tomaría  $O(n \lg n)$  y buscarlos eficientemente  $O(\lg n)$ , por lo que si necesitamos buscar pocas veces algún valor, no es conveniente.

#### Final Standings

Old contest software uses bubble sort for generating final standings. But now, there are too many teams and that software works too slow.

#### Problem

You are asked to write a program, which generates exactly the same final standings as old software, but fast.

#### Input

The first line of input contains only integer  $1 < n < 150000$  - number of teams. Each of the next  $n$  lines contains two integers  $1 \leq ID \leq 10000000$  and  $0 \leq m \leq 100$ .  $ID$  - unique number of team,  $m$  - number of solved problems.

#### Output

Output should contain  $n$  lines with two integers  $ID$  and  $m$  on each. Lines should be sorted by  $m$  in descending order using bubble sort (or analog).

*Ural: 1100*

**Solución:** Nos afirman que la ordenación por burbuja es demasiado lenta (lo cual podemos comprobar fácilmente), y que debemos encontrar una alternativa. Cualquier otro algoritmo  $O(n^2)$  será igual de ineficiente, por lo que debemos buscar uno con cota menor.

Un problema con el que nos enfrentamos es que la ordenación por burbuja es estable, por lo que el algoritmo que lo reemplaza también debe ser estable. Ordenación por Inserción cumple este requisito, pero tiene la misma complejidad que la ordenación por Burbuja. Podríamos ordenarlos por Mezcla o por Conteo, pero estos métodos requieren demasiada memoria (más de la que disponemos).

Es posible modificar la ordenación Rápida de Hoare para que sea estable, pero existe una solución más simple. Guardamos todos los datos, y en lugar de ordenarlos, buscamos primero los equipos que resolvieron más problemas, luego los que resolvieron uno menos, y así sucesivamente hasta que lleguemos a los que no resolvieron problema alguno.

```

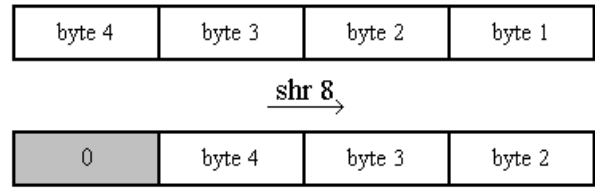
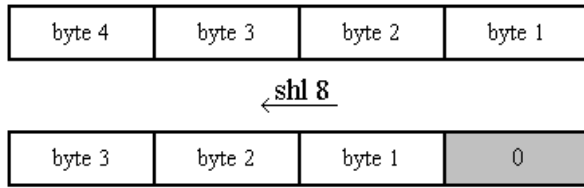
01: Var
02:   i,m,n,t: longint;
03:   d: array [1..150000] of longint;
04:
05: Procedure BusqLineal(x: longint);
06:   var i: longint;
07:   begin
08:     for i:= 1 to n do
09:       if (x= d[i] AND $00FF) then
10:         writeln(output,d[i] shr 8,' ',x);
11:     end;
12:
13: Begin
14:   t:= 0;
15:   readln(input,n);
16:   for i:= 1 to n do
17:     begin
18:       readln(input,d[i],m);
19:       if (m>t) then t:= m;
20:       d[i]:= (d[i] shl 8) OR m;
21:     end;
22:   for i:= t downto 0 do
23:     BusqLineal(i);
24: End.

```

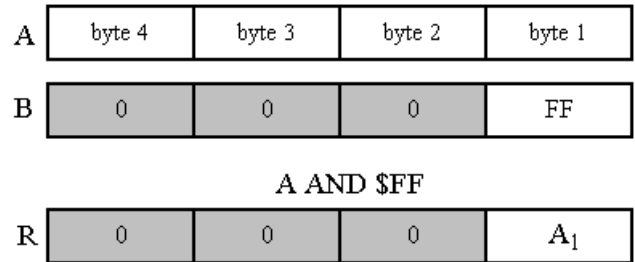
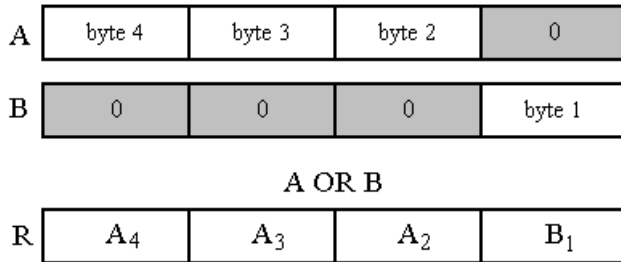
En las primeras tres líneas declaramos las variables globales que vamos a utilizar. En **n** leemos la cantidad de datos a utilizar, en **m** leemos el número de problemas resueltos, **t** sirve para encontrar la cantidad máxima de problemas resueltos, **i** es una variable para ciclos, y en **d** guardamos tanto el número de equipos como la cantidad de problemas que resolvió.

Una de nuestras principales limitantes es la memoria, por lo que vamos a realizar un pequeño “truco” para eficientar espacio. Normalmente utilizaríamos un arreglo de *longint* (4 bytes) para los números de los equipos y uno de *shortint* (1 byte) para los problemas resueltos. Con esto, el espacio que necesitaríamos por cada equipo es de 5 bytes. El número de identificación de máximo que puede tener un equipo es de 10 millones, que necesita menos de 3 bytes. Por esto, podemos guardar los problemas en el primer byte y el número en los siguientes tres, todo dentro del *longint*. Con esto tenemos un ahorro del 20% en memoria.

Ya sabemos como ahorrar memoria, pero también necesitamos saber como implementarlo. Conocemos que un byte es equivalente a ocho bits, por lo que desplazando el número de equipo ocho veces a la izquierda (los desplazamiento se explican más a fondo en la sección 6.5), ahora el número iniciaría a partir del segundo byte. Para regresar el número a como estaba originalmente lo hacemos mediante ocho desplazamientos a la derecha.



Para poder meter la cantidad de problemas en el primer byte, lo hacemos mediante la operación OR entre el número en el que lo queremos guardar (**d[i]**) y el número que queremos guardar (**m**). Si queremos volver a leer el número de problemas, lo hacemos mediante la operación AND entre el número y FF (8 unos en el primer byte).



Entre las líneas 5 y 11 se encuentra el procedimiento de la búsqueda lineal (**busqlineal**). Empezamos declarando en la línea 6 la variable **i** que nos va a servir para recorrer el arreglo. Entre las líneas 8 y 10 buscamos el número, y en caso de encontrarlo lo imprimimos. Normalmente, la búsqueda la realizaríamos hasta el primer dato que fuera igual a **x** y devolveríamos la posición en la que lo encontramos o alguna notificación en caso contrario.

La parte principal del código lo encontramos de la línea 13 a la 24. Primero inicializamos el máximo de resueltos y leemos la cantidad de equipos. Después leemos el número del equipo y la cantidad de problemas que resolvieron (línea 18), y los juntamos en un solo *longint* (línea 20). En dado caso que la cantidad de problemas sea mayor al máximo, actualizamos el máximo (línea 19).

Por último, buscamos linealmente los equipos que resolvieron la mayor cantidad, después los que resolvieron uno menos, y seguimos realizándolo hasta que llegemos a los que no resolvieron problemas.

**Ejemplo:**

Entrada:	8
	1 2
	16 3
	11 2
	20 3
	3 5
	26 4
	7 1
22 4	
Desarrollo:	( 1 shl 8) OR 2 = 258

	(16 shl 8) OR 3 = 4099
	(11 shl 8) OR 2 = 2818
	(20 shl 8) OR 3 = 5123
	( 3 shl 8) OR 5 = 773
	(26 shl 8) OR 4 = 6660
	( 7 shl 8) OR 1 = 1793
	(22 shl 8) OR 4 = 5636
	búsqueda
	5 258, 4099, 2818, 5123, <b>773</b> , 6660, 1793, 5636
	4 258, 4099, 2818, 5123, 773, <b>6660</b> , 1793, <b>5636</b>
	3 258, <b>4099</b> , 2818, <b>5123</b> , 773, 6660, 1793, 5636
	2 <b>258</b> , 4099, <b>2818</b> , 5123, 773, 6660, 1793, 5636
	1 258, 4099, 2818, 5123, 773, 6660, <b>1793</b> , 5636
	0 258, 4099, 2818, 5123, 773, 6660, 1793, 5636
Salida:	3 5
	26 4
	22 4
	16 3
	20 3
	1 2
	11 2
	7 1

### 3.7. Búsqueda binaria.

Si la lista donde vamos a buscar se encuentra previamente ordenada, podemos utilizar este método que es notablemente más rápido que el anterior (corre en  $O(\lg n)$ ). Esta búsqueda utiliza una subcategoría especial de “Divide y Vencerás”, conocida como “Decrementa y Vencerás”.

La diferencia entre estas técnicas es que en la primera, para resolver un problema lo descomponemos en subpartes, las cuales resolvemos para obtener la solución. En la segunda, también comenzamos dividiendo el problema en subproblemas, pero en lugar de resolverlos todos, desechamos los que no son útiles para obtener la respuesta y resolvemos con los demás.

Para realizar una búsqueda binaria, lo que hacemos es lo siguiente: dividimos el arreglo en dos y tomamos la casilla que se encuentra entre estos subarreglos. Si la casilla que escogimos es igual al dato que buscamos, hemos encontrado la solución. Si ocurre que son distintos, revisamos los dos posibles casos: si la casilla es más grande que el dato que estamos buscando, las casillas mayores también lo serán y podemos desecharlas; en caso contrario nos deshacemos de las casillas menores. Después aplicamos el mismo método en las casillas restantes.

Cabe mencionar que la partición en mitades produce un resultado óptimo. Si realizamos particiones en tamaños desiguales o en más de dos partes, la complejidad del algoritmo no cambia pero las constantes se incrementan significativamente.

## History Exam

### Problem

Teacher of history decided to simplify examination process. Every student should write list of years with well-known events. Teacher has its own list. Student get mark according to number of years in his list which are contained in teacher's list.

### Input

First line contains  $n$  - number of years in teacher's list.  $1 \leq n \leq 15000$ . Next  $n$  lines contain teacher's list. Every year in list doesn't exceed  $10^9$ . Teacher's list is sorted in ascending order.

Next line contains  $m$  - number of years in student's list.  $1 \leq m \leq 100000$ . Next  $m$  lines contain student's list. This list isn't sorted. Years in student's list can appear more than once.

### Output

One integer - amount of numbers in student's list which are contained in teacher's list.

Ural: 1196

**Solución:** Tenemos que buscar las fechas de los alumnos en la lista de la maestra. Utilizando búsqueda lineal nos tardaríamos demasiado, por lo que lo vamos a hacer con una búsqueda binaria. Primero leemos la lista de la maestra y después leemos cada dato de los alumnos. Siempre que encontremos un dato en la lista, incrementamos un contador.

```

01: Var
02:   i,n,m,t,x: Longint;
03:   lista: array [1..15000] of Longint;
04:
05: Function BusqBinaria(x: longint): longint;
06:   var sup,inf,mitad: longint;
07:   begin
08:     inf:= 1; sup:= n;
09:     while (sup >= inf) do
10:       begin
11:         mitad:= (sup + inf) div 2;
12:         if (x> lista[mitad]) then inf:= mitad + 1 else
13:         if (x< lista[mitad]) then sup:= mitad -1 else
14:                                     sup:= -1;
15:       end;
16:       if (sup=-1) then BusqBinaria:= mitad
17:       else BusqBinaria:= 0;
18:     end;
19:
20: Begin
21:   readln(input,n);
22:   for i:= 1 to n do
23:     readln(input,lista[i]);
24:   readln(input,m);
25:   t:= 0;
26:   for i:= 1 to m do
27:     begin
28:       readln(input,x);
29:       if (BusqBinaria(x)<>0) then
30:         Inc(t);
31:     end;
32:   writeln(output,t);
33: End.
```

En las primeras tres líneas declaramos las variables a utilizar. Las variables **n** y **m** son la cantidad de fechas en la lista de la maestra y en la de los estudiantes, respectivamente. Utilizamos **i** para los ciclos y **t** para contar el total de datos en la lista de la maestra. Para guardar la lista utilizamos el **arreglo** lista y en **x** leemos el año a buscar.

A continuación tenemos la función que realiza la búsqueda binaria (líneas 5 a 18). La implementación que tenemos es iterativa y se puede hacer recursiva sin muchos problemas. Las variables **sup**, **inf** y **mitad** las usamos para señalar las casillas superior, inferior y media, respectivamente. Inicializamos la parte inferior a la primera casilla y la superior a la última en la línea 8.

En la línea 11 encontramos la casilla que se encuentra en medio, para después revisar si es mayor, menor o igual al dato que buscamos. En caso que sea mayor, ajustamos nuestro límite inferior a uno arriba de la mitad; si es menor, ajustamos el superior a uno abajo; si es igual, entonces hacemos que el límite superior tome el valor de  $-1$ . Esto lo repetimos hasta que el valor superior sea menor al inferior. Es importante que los límites los cambiemos uno mayor o menor a la mitad, ya que esto evita que el código se quede ciclado. Al final, sabemos que **sup** no puede tomar valores negativos por lo que si es igual a  $-1$  entonces encontramos el dato, en caso contrario, devolvemos un valor que nos indique que el dato no está en el arreglo.

La siguiente parte es la parte principal del código (líneas 20 a 33). Comenzamos leyendo la cantidad de datos en la lista, para después leerla (líneas 21 a 23). Después leemos la cantidad de fechas en la lista de los alumnos e inicializamos el contador a cero. Para cada fecha lo que hacemos es leerla, buscarla en la lista y, en caso de encontrarla, incrementar el contador. Al final escribimos la cantidad de datos que concordaban en las dos listas.

### *Ejemplo:*

	10
	98
	486
	699
	1054
	1097
	1492
	1493
Entrada:	1763
	1982
	2006
	4
	1492
	65536
	1492
	100

<p>Desarrollo:</p>	<pre> 98 486 699 1054 1097 1492 1493 1763 1982 2006 x= 1492 inf= 1, sup= 10, lista[mitad]= 1097 &lt; 1492 inf= 6, sup= 10, lista[mitad]= 1763 &gt; 1492 inf= 6, sup= 7, lista[mitad]= 1492 = 1492 inf= 6, sup= -1 BusqBinaria= 6  x= 65536 inf= 1, sup= 10, lista[mitad]= 1097 &lt; 65536 inf= 6, sup= 10, lista[mitad]= 1763 &lt; 65536 inf= 9, sup= 10, lista[mitad]= 1982 &lt; 65536 inf= 10, sup= 10, lista[mitad]= 2006 &lt; 65536 inf= 11, sup= 10 BusqBinaria= 0  x= 100 inf= 1, sup= 10, lista[mitad]= 1097 &gt; 100 inf= 1, sup= 4, lista[mitad]= 486 &gt; 100 inf= 1, sup= 1, lista[mitad]= 98 &lt; 100 inf= 2, sup= 1 BusqBinaria= 0 </pre>
<p>Salida:</p>	<p>2</p>