



ACM International Collegiate Programming Contest
2005

South American Regional

November 11-12, 2005

Contest Session

(This problem set contains 9 problems; pages are numbered from 1 to 17)

Una Solución (*) del Problema F “Where Are My Genes”

Viktor Khlebnikov
(vkhlebn@pucp.edu.pe)

Enero de 2006

(*) *Esta solución fue desarrollada sin conocimiento de la solución del autor del problema.*

Comencemos a leer el enunciado:

Problem F Where Are My Genes

Source file name: `genes.c`, `genes.cpp`, `genes.java` or `genes.pas`

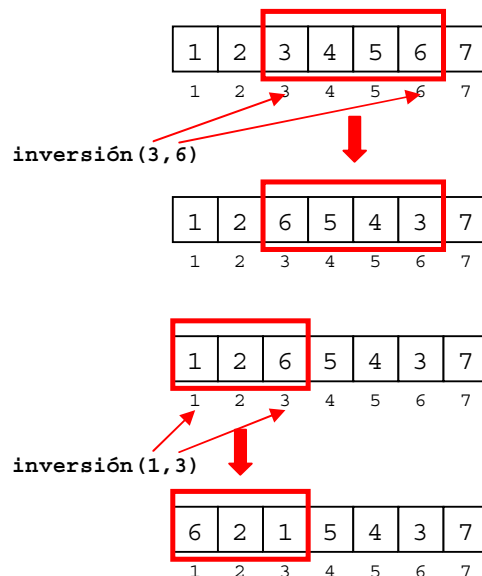
One way that scientists try to measure how one species evolved into another is to find out how the ancestor's genome changed into the other's. Closely related species have several genes in common and it turns out that a good way to compare them is comparing how the common genes changed place.

Tenemos: el nombre del archivo de programa: `genes.c`
el archivo de entrada: `genes.in`
el archivo de salida: `genes.out`

¿Se trata de una comparación de cadenas?
Seguimos con la lectura...

One of the most common mutations that change the order of a genomes' genes is the reversal. If we model a genome as a sequence of N genes with each gene being an integer from 1 to N , then a reversal is a mutation that changes the genome by reverting the order of a block of consecutive genes. A reversal can be described by two indexes (i, j) , $1 \leq i \leq j \leq N$, indicating that it reverts the order of the genes within indexes from i to j . So, when it is applied to a genome $[g_1, \dots, g_{i-1}, g_i, g_{i+1}, \dots, g_{j-1}, g_j, g_{j+1}, \dots, g_N]$, we obtain the genome $[g_1, \dots, g_{i-1}, g_j, g_{j-1}, \dots, g_{i+1}, g_i, g_{j+1}, \dots, g_N]$. As an example, the reversal $(3, 6)$ applied to the genome $[1, 2, 3, 4, 5, 6, 7]$ gives $[1, 2, 6, 5, 4, 3, 7]$. If after that the reversal $(1, 3)$ is applied, we obtain the genome $[6, 2, 1, 5, 4, 3, 7]$.

Se aplican unas *inversiones de segmentos* en un vector de datos:



A scientist studying the evolution of a species wants to try a series of reversals on its genome. Then he wants to query the final position of several genes. Would you take the challenge and help him?

Importante: No interesa el estado final de todo el vector (si el vector será nuestra estructura de datos) sino las posiciones finales de algunos de sus elementos después de aplicar varias inversiones.

Input

The input contains several test cases. The first line of a test case contains one integer N indicating the number of genes in the genome ($1 \leq N \leq 50000$). You may assume that the initial order of the genes is the sequence of integers from 1 to N in increasing order. The second line of a test case contains one integer R ($0 \leq R \leq 1000$) indicating the number of reversals to be applied to the genome. Then R lines follow, each containing two integers i, j ($1 \leq i \leq j \leq N$), separated by a single space, indicating the two indexes that define the corresponding reversal. After the description of the reversals there is a line containing one integer Q ($0 \leq Q \leq 100$), indicating the number of queries for genes, followed by Q lines, where each line contains an integer representing a gene whose final position you must determine.

The end of input is indicated by $N = 0$.

The input must be read from standard input.

El ejemplo de entrada y su interpretación:

```

9          Caso 1:  $N = 9$  (cantidad de elementos) ( $1 \leq N \leq 50000$ )
1           $R = 1$  (cantidad de inversiones) ( $0 \leq R \leq 1000$ ) – puede no haber inversiones
3 6        Inversión 1 = (3, 6)
4           $Q = 4$  (cantidad de consultas) ( $0 \leq Q \leq 100$ ) – puede no haber consultas
1          ¿Dónde está el elemento que estaba inicialmente en la posición 1?
3          en la posición 3?
5          en la posición 5?
1          en la posición 1? (¿otra vez?)
5          Caso 2:  $N = 5$  (cantidad de elementos)
2           $R = 2$  (cantidad de inversiones)
1 2        Inversión 1 = (1, 2)
1 5        Inversión 2 = (1, 5)
2           $Q = 2$  (cantidad de consultas)
5          ¿Dónde está el elemento que estaba inicialmente en la posición 5?
2          en la posición 2?
0          Fin de entrada:  $N = 0$ 

```

Output

For each test case in the input your program must produce $Q + 1$ lines of output. The first line must contain the string "Genome " followed by the number of the test case. The following Q lines must contain one integer each representing the answers of the queries.

The output must be written to standard output.

```

Genome 1
1
6
4
1
Genome 2
1
5

```

Primero pensemos un poco ...

Estamos en el momento de tomar una decisión. La más común y trivial es la de crear un vector con los valores iniciales y aplicar a él todas las inversiones de segmentos dados. Es decir, modelar en la computadora el mismo procedimiento de inversiones descritas y, al obtener el vector resultante, buscar en él el valor requerido. En este caso NO se requiere elaborar ningún algoritmo especial, simplemente se necesita conocer cómo se codifica este modelo en un lenguaje de programación particular. Podríamos decir que nosotros simulamos con nuestro programa todo lo que pasa en el vector y, ya después, pensamos cómo se obtiene el resultado pedido. En este caso el programa será la *codificación* de las mismas acciones que fueron descritos en el enunciado. Lo llamaremos a este enfoque “primero simulamos, después pensamos”.

Llevaremos nuestra situación al caso extremo y exagerado. Si nuestro cliente dice que tiene un vector cuyos 50,000 valores corresponden al número de su posición (índice), y cada valor se duplica, entonces a nadie se le va ocurrir crear este vector para guardar todos los valores. Simplemente, el resultado puede ser calculado por la fórmula elemental – el elemento x en la posición x cambiará su valor a $2x$. Y esto es todo...

Si al vector [1, 2, 3, 4, 5, 6, 7] se aplica primero la inversión (3,6) y después la inversión (1,3), entonces responder a la pregunta sobre la nueva posición del elemento 7 es fácil – este elemento se quedó en su posición inicial 7 porque no fue afectado por las inversiones aplicadas.

Mientras que el elemento 6 será afectado por la primera inversión cambiando su posición a 3 que, a su vez, será afectada por la 2da inversión cambiándose a la posición 1.

Entonces la idea es (*aquí nosotros estamos creando el algoritmo de solución, en vez de codificar el enunciado*) calcular las nuevas posiciones de los elementos consultados (y solamente consultados) según las inversiones que se aplican a partir de la posición inicial conocida.

Podremos pasar a la codificación de nuestra idea (algoritmo). En este artículo el algoritmo se codificará en el lenguaje C, y el programa será desarrollado en Debian GNU/Linux 3.1.

Nota: siendo el profesor de un curso de programación, él que escribe propuso a sus alumnos resolver este problema en un examen. De los 25 estudiantes solamente uno optó por la solución no trivial. Con su permiso, su solución en Pascal se presenta al final de este artículo.

El inicio de codificación

```
$ cat genes.c
#include <stdlib.h>

int
main(void)
{
    exit(0);
}
$ gcc genes.c -o genes
$ ./genes
$
```

Es un buen inicio ya que vemos que nuestro ambiente de desarrollo funciona sin problemas.

El enunciado dice que habrá varios casos de prueba. Cada caso comienza con la cantidad de elementos, y el valor 0 significa el fin de datos de entrada. Construiremos el bucle más externo:

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

int
main(void)
{
    int n, test=1;

    while(1) {
        scanf("%d", &n);
        if (!n) break;
        printf("Genome %d\n", test++);
    }
    exit(0);
}
```

La negación del cero se considerará como true y provocará la salida del bucle “eterno” gracias a break.

El siguiente dato de entrada es la cantidad de inversiones que se aplicarán. Estas inversiones las almacenaremos en un vector para usarlas al momento de consultas. Puede haber hasta 1000 inversiones, y cada inversión se indica con dos enteros:

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

int
main(void)
{
    int n, test=1, r, i, j, rev[2000];

    while(1) {
        scanf("%d", &n);
        if (!n) break;
        printf("Genome %d\n", test++);
        scanf("%d", &r);
        i = 0;
        j = r;
        while (j--) {
            scanf("%d %d", &rev[i<<1], &rev[(i<<1)+1]);
            i++;
        }
    }
    exit(0);
}
```

En el vector se almacena cada pareja de los límites de la inversión; en la posición par (0, 2, 4, ...) se guarda el inicio del segmento de inversión, y en la posición impar (1, 3, 5, ...) se guarda el fin del segmento. En lugar de usar las expresiones $2*i$ y $2*i+1$, se da la preferencia al cálculo equivalente pero más rápido con el operador de corrimiento a la izquierda $<<$.

Ahora se puede proceder con las consultas:

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

int
main(void)
{
    int n, test=1, r, i, j, q, qry, rev[2000];

    while(1) {
        scanf("%d", &n);
        if (!n) break;
        printf("Genome %d\n", test++);
        scanf("%d", &r);
        i = 0;
        j = r;
        while (j--) {
            scanf("%d %d", &rev[i<<1], &rev[(i<<1)+1]);
            i++;
        }
        scanf("%d", &q);
        while (q--) {
            scanf("%d", &qry);
            /* aquí debe estar el cálculo de qry*/
            printf("%d\n", qry);
        }
    }
    exit(0);
}
```

Preparamos el archivo de entrada `genes.in` para probar nuestro programa.
Ejecutemos el programa proporcionando como entrada el archivo de datos creado:

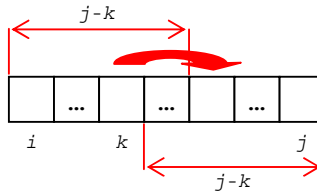
```
$ cat genes.in
9
1
3 6
4
1
3
5
1
5
2
1 2
1 5
2
5
2
0
$ ./genes <genes.in
Genome 1
1073838832
1073838832
1073838832
1073838832
Genome 2
```

1073838832
1073838832
\$

Los valores de las posiciones consultadas aún no han sido calculados. Hay que hacerlo ahora.

La implementación del algoritmo y la versión final del programa

Si se aplica la inversión (i, j) y la posición k está fuera del segmento de inversión ($k < i$ o $j < k$), entonces la posición no se modifica. Para el caso $i \leq k \leq j$, la posición se cambiará a $i + (j - k)$ (k estaba en la distancia $j - k$ de j , ahora estará en la misma distancia de i).



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

int
main(void)
{
    int n, test=1, r, i, j, q, qry, rev[2000];

    while(1) {
        scanf("%d", &n);
        if (!n) break;
        printf("Genome %d\n", test++);
        scanf("%d", &r);
        i = 0;
        j = r;
        while (j-- > 0) {
            scanf("%d %d", &rev[i], &rev[i+1]);
            i++;
        }
        scanf("%d", &q);
        while (q-- > 0) {
            scanf("%d", &qry);
            j = 0; /* ya tiene este valor después del while anterior */
            while (j < r) {
                if ( (rev[j] <= qry) && (qry <= rev[j+1]) )
                    qry = rev[j] + (rev[j+1] - qry);
                j++;
            }
            printf("%d\n", qry);
        }
    }
    exit(0);
}
```

Probemos la versión final del programa:

```

$ ./genes <genes.in
Genome 1
1
6
4
1
Genome 2
1
5
$

```

Los resultados coinciden con los del ejemplo dado en el enunciado. Hemos terminado el desarrollo del programa.

Los datos de entrada, que fueron usados por los jueces en el concurso, están disponibles. Probaremos nuestro programa con estos datos. Son dos archivos, `genes.in` y `genes.sol`:

```

$ ls -l genes.{in,sol}
-rw-r--r-- 1 vkhlebn vkhlebn 175035 2006-01-13 11:36 genes.in
-rw-r--r-- 1 root     root      6116 2006-01-13 11:36 genes.sol
$ wc genes.in
19791 38187 175035 genes.in

```

El archivo `genes.in` contiene 19791 líneas, 38187 palabras y 175035 bytes.

Aquí va la prueba final y la comparación de nuestro resultado con el resultado debido:

```

$ ./genes <genes.in >genes.out
$ cmp genes.out genes.sol
$

```

El programa de comparación `cmp` no encuentra ninguna diferencia entre los archivos. Esto significa que nuestro programa será aceptado por los jueces.

La versión trivial

Para la comparación de las ideas, implementaremos la versión trivial que simulará la inversión de los elementos del vector, la del enfoque “primero simulamos, después pensamos”.

Observación: en la versión anterior no nos interesó que en el lenguaje C los elementos del vector se numeran a partir de 0, mientras que los genes se numeran de 1 a 50000. Porque nosotros nunca hemos almacenado los genes. Ahora reservaremos en la memoria el vector para 50001 elementos para quitar la diferencia en la base de numeración.

```

$ cat genes-triv.c

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

void reversal(int, int);

int gen[50001];

int
main(void)
{
    int n, test=1, r, i, j, q, qry;

```

```

while (1) {
    scanf("%d", &n);
    if (n==0) break;
    for (i = 1; i <= n; i++) gen[i] = i;
    printf("Genome %d\n", test++);
    scanf("%d", &r);
    while (r--) {
        scanf("%d %d", &i, &j);
        reversal(i,j);
    }
    scanf("%d", &q);
    while (q--) {
        scanf("%d", &qry);
        i = 1;
        while (qry != gen[i]) i++;
        printf("%d\n", i);
    }
}

void
reversal(int i, int j)
{
    int temp;

    for ( ; i < j; i++, j--) {
        temp = gen[i];
        gen[i] = gen[j];
        gen[j] = temp;
    }
    return;
}

```

La prueba muestra que la solución proporcionada por el programa es correcta:

```

$ gcc genes-triv.c -o genes-triv
$ ./genes-triv <genes.in >genes.out
$ cmp genes.out genes.sol
$

```

Comparación de ambos programas

Podremos medir el tiempo de ejecución de cada versión del programa:

```

$ time ./genes <genes.in >genes.out

real    0m0.028s
user    0m0.020s
sys     0m0.000s

$ time ./genes-triv <genes.in >genes.out

real    0m2.507s
user    0m2.500s
sys     0m0.000s

```

Mientras que la primera versión del programa resuelve el problema en 28 ms, la segunda necesita más de 2 s (es decir, es 70 veces más lenta). Si los jueces establecen el límite de tiempo de ejecución en los 2 segundos, la segunda versión no será aceptada.

Anexo

Aquí se presenta un bosquejo de la solución del problema en Pascal desarrollado por un estudiante durante un examen (con su permiso otorgado):

```
{ Autor: José G. Rosales Rojas
  a20037208@pucp.edu.pe
  13.12.2005
  Desarrollado durante el Examen Final de FP }
```

```
program P4;

var ...

begin
  Fin := FALSE; Gen := 1; Inicializa(A);
  while NOT Fin do
  begin
    Readln(N);
    if N = 0 then Fin := True
    else
    begin
      Readln(R);
      L := 0;
      for M := 1 to R do
      begin
        Read(I); Readln(J);
        Inc(L);
        A[L] := I;
        A[L+1] := J;
      end;
      Readln(QQ);
      if QQ <> 0 then Writeln('Genome ', Gen);
      for M := 1 to QQ do
      begin
        Readln(Q);
        Pos := Busca_Pos_Final(A, Q);
        Writeln(Pos);
      end;
      Inc(Gen);
      Inicializa(A);
    end;
  end;
end;

function Busca_Pos_Final(var A: vector, Pos: Longword)

var ...

begin
  M := 1;
  while A[M] <> 0 do
  begin
    I := A[M];
    J := A[M+1];
    if Pos in [I..J] then
    begin
      New_Pos := I + (J - Pos);
    end;
    M := M + 2;
  end;
  Busca_Pos_Final := New_Pos;
end;
```
