

ALGORITMOS Y ESTRUCTURAS DE DATOS III

Trabajo Práctico N°1

Carla Livorno	424/08	carlalivorno@hotmail.com
Daniel Grosso	694/08	dgrosso@gmail.com
Diego Raffo	423/08	enanodr@hotmail.com
Mariano De Sousa Bispo	389/08	marian_sabianaa@hotmail.com

Abril 2010

Índice

Introducción	2
1. Problema 1	2
1.1. Explicación	2
1.2. Detalles de la implementación	2
1.3. Análisis de complejidad	3
1.4. Pruebas y Resultados	5
2. Problema 3	6
2.1. Explicación	6
2.2. Detalles de la implementación	6
2.3. Análisis de complejidad	9
2.4. Pruebas y Resultados	9
3. Mediciones	10
4. Compilación y ejecución de los programas	10

Introducción

Este trabajo tiene como objetivo la aplicación de diferentes técnicas algorítmicas para la resolución de tres problemas particulares, el cálculo de complejidad teórica en el peor caso de cada algoritmo implementado, y la posterior verificación empírica.

El lenguaje utilizado para implementar los algoritmos de todos los problemas fue C/C++

1. Problema 1

enunciado

1.1. Explicación

La resolución del problema consiste en considerar cada uno de los elementos de la secuencia dada como posible máximo de una subsecuencia (al que nos referiremos como “pico”), tal que sea unimodal. Cada subsecuencia unimodal tiene longitud máxima, es decir, contiene la subsecuencia creciente hasta el pico y la subsecuencia decreciente desde el pico, ambas con la mayor cantidad de elementos posibles. Al no necesitar explicitar los elementos de la secuencia, la única información que aporta a la solución del problema es la longitud de cada subsecuencia. De esta manera, para determinar la mínima cantidad de elementos que se deben eliminar para transformar la secuencia dada en unimodal, basta con conocer la diferencia entre la longitud de la secuencia original y el máximo de las longitudes de cada subsecuencia unimodal.

1.2. Detalles de la implementación

Para determinar la máxima longitud de la subsecuencia creciente y decreciente hasta cada elemento, utilizamos la técnica de programación dinámica.

Sea la secuencia dada $S = [s_1, s_2, \dots, s_n]$ y $C = [c_1, c_2, \dots, c_n]$ con $c_i = \max_{0 < j < i} \{c_j / s_j < s_i\} + 1$ ($\forall i \in \mathbb{N}, 0 < i \leq n$), es decir, para cada i tenemos en c_i la máxima longitud de la subsecuencia creciente que incluye a s_i . Análogamente, se define D como la secuencia que contiene las máximas longitudes de las subsecuencias decrecientes de S .

El algoritmo que calcula la solución implementa tanto C como D . Para obtener la máxima longitud de la subsecuencia creciente hasta el índice i , itera por todos los índices $j < i$ en C , buscando el máximo valor entre los c_j tales que s_i sea mayor que s_j . Esto asegura que en la posición c_i está la máxima longitud de la subsecuencia creciente que incluye a s_i ya que, si existiese otra subsecuencia de mayor longitud a la que se pueda agregar s_i , se podría agregar el s_i a esa secuencia y así obtener una con más cantidad de elementos, siendo el valor de c_i la longitud de dicha secuencia más 1. Para calcular D , invertimos S y aplicamos el mismo procedimiento que para C , quedando en d_i la máxima longitud de la subsecuencia decreciente que incluye a s_{n-i} .

Luego de calcular C y D el algoritmo determina la mínima cantidad de elementos a ser eliminados de la secuencia tales que el resto de los elementos forman una secuencia unimodal, de la siguiente forma:

```

secuencia_unimodal(secuencia,  $C$ ,  $D$ )
     $max\_long \leftarrow 0$ 
    for  $i = 1$  to  $n$ 
         $long\_secuencia\_unimodal \leftarrow C[i] + D[n - i] - 1$ 
        if  $long\_secuencia\_unimodal > max\_long$ 
             $max\_long \leftarrow long\_secuencia\_unimodal$ 
    return  $longitud(secuencia) - max\_long$ 

```

1.3. Análisis de complejidad

Como hemos visto anteriormente, nuestro algoritmo utiliza la técnica de programación dinámica. Esta consta de reutilizar información previa para llegar al resultado final. En el análisis de la complejidad veremos como la aplicación de la técnica previamente mencionada, ha permitido conseguir un algoritmo polinomial.

El algoritmo *long_max_creciente* utiliza la técnica de programación dinámica, calculando para cada elemento i de la secuencia original la máxima longitud de la subsecuencia creciente y decreciente que lo incluye, iterando por todos los índices $j < i$ ($\forall 0 \leq i < n$), sabiendo que para cada j ya esta

calculada la longitud de la subsecuencia creciente más larga que lo incluye.

Sea n la longitud de la secuencia dada, $max_long_creciente$ la secuencia que guarda en cada posición la máxima longitud de la subsecuencia creciente.

```

long_max_creciente(secuencia, max_long_creciente)
    max_long_creciente[0] ← 0 O(1)
    for i = 1 to n
        max_long ← 0 O(1)
        for j = i - 1 to 0 O(i)
            if secuencia[j] < secuencia[i] and max_long_creciente[j] > max_long O(1)
                max_long ← max_long_creciente[j] O(1)
            max_long_creciente[i] ← max_long + 1 O(1)

```

Podemos ver dentro de cada ciclo (*for*) que todas las asignaciones tienen costo constante, así como también la guardar del (*if*). De esta manera podemos concluir que por cada iteración del (*for*) anidado tenemos en el peor caso el costo de la guarda del *if*, más el costo de la asignación adentro del mismo, con costo constante.

El ciclo anidado iterará para cada i , $i - 1$ veces, siendo los valores posibles de i desde 1 hasta n .

La complejidad de dicho algoritmo viene dada por: $\sum_{i=0}^{n-1} i = \frac{n*(n-1)}{2}$

Esta sumatoria, se coincide entonces con el costo de la función *long_max_creciente*. La cantidad de operaciones que realiza este algoritmo es $O(n^2)$.

El algoritmo encargado de devolver el resultado del problema es *secuencia_unimodal*, el cual utiliza el algoritmo descrito anteriormente y realizar algunas otras operaciones que se detallarán a continuación para concluir con el análisis de complejidad.

```

secuencia_unimodal(secuencia, n)
    C[n]
    D[n]
    R ← reverso(secuencia)  $O(n)$ 
    long_max_creciente(secuencia, C)  $O(n^2)$ 
    long_max_creciente(R, D)  $O(n^2)$ 
    max_long ← 0  $O(1)$ 
    for i = 1 to n  $O(n)$ 
        long_secuencia_unimodal ← C[i] + D[n − i] − 1  $O(1)$ 
        if long_secuencia_unimodal > max_long  $O(1)$ 
            max_long ← long_secuencia_unimodal  $O(1)$ 
    return longitud(secuencia)-max_long  $O(1)$ 

```

La función *reverso* que toma una secuencia y devuelve otra con los elementos en orden inverso tiene costo lineal, en función de la cantidad de elementos, ya que itera una vez por la secuencia original (*desde* $= 0$ *hasta* $n - 1$), guardando cada valor en la posición $n - 1 - i$ de la secuencia resultante. Cabe destacar que el costo de la asignación es constante.

El ciclo perteneciente a *secuencia_unimodal* (*for*, *linea*7), itera n veces, siendo en el peor caso el costo de cada una, constante (SE PODIRA PONER TODOS LOS O DEL MUNDO).

Por lo tanto, el costo del algoritmo *secuencia_unimodal* es: $O(2*n + 2*n^2)$ que por definición es $O(\max(2*n, 2*n^2)) = O(2*n^2) = O(n^2)$.

1.4. Pruebas y Resultados

2. Problema 3

enunciado

2.1. Explicación

Como Bernardo puede pasar más de una vez por cada habitación el algoritmo restringe el acceso a cada una de estas, podrá acceder a cada habitación tantas veces como pasillos llegan a la habitación. Bernardo recorre las habitaciones que tengan conexión con la habitación donde se encuentra, siempre y cuando pueda entrar, ya sea porque tiene la llave o la habitación no tiene puerta y además, todavía tiene acceso a la habitación a entrar. Su procedimiento continúa hasta que:

- Llega a la habitación n , por lo tanto encontró la salida.
- Se le terminan los accesos a las habitaciones vecinas y por lo tanto no puede escapar.

2.2. Detalles de la implementación

Modelamos este problema con un grafo donde los vértices corresponden a las habitaciones (puede ser una habitación con una llave dentro, con una puerta o sin puerta ni llave) y las aristas a los pasillos.

Generamos la matriz de adyacencia del grafo (de $n \times n$ donde n es la cantidad de habitaciones donde cada posición (i, j) de la matriz contiene un *uno* si la habitación i está conectada con la habitación j . y *cero* en caso contrario). Nos referiremos a la matriz como *conexiones*.

Para toda habitación que tenga puerta existe una llave. Poseer esta llave implica tener la posibilidad de acceder a la habitación. Podemos abstraernos del problema de Bernardo y considerar a las llaves como valores booleanos en un arreglo que nos dice para cada vértice, si este es accesible o no. Tenemos entonces, un arreglo de tipo bool (*tengo_llave*) de tamaño n donde cada vértice, representado por el índice de dicho arreglo, nos dice si es accesible o no.

Por otro lado, tenemos un arreglo *puertas* de tamaño n (donde n es la cantidad de vértices del grafo), donde cada posición, si corresponde a una habitación con llave, tiene el vértice al cual habilita el acceso. En caso contrario, el arreglo contiene el valor 0. El valor es cero, porque como se verá más adelante en el código, modificará información del primer vértice, que a los fines prácticos, no modifica el resultado final del algoritmo. ESTO CAPAS Q ESTA BIEN X FAVOR CHEQUEENLO.

Para la resolución del problema recorrimos el grafo de forma ordenada por niveles. Para esto hicimos una modificación al algoritmo *Breadth First Search*. La modificación consiste en visitar por niveles hasta que algún vértice no sea accesible (mientras sea nodo accesible se continúa recorriendo su subnivel), al encontrarlo, se lo marca como vértice en *espera* hasta que se encuentre el acceso en algún otro vértice al que está en espera. Si existe el acceso, se accede al vértice en espera y se continúa avanzando. Cabe destacar que este algoritmo puede acceder a cada nodo sólo una vez.

El objetivo del *bfs* es llegar desde el primer vértice al último.

Existe un arreglo de bools llamado *enEspera* el cual esta seteado en falso si la habitación i tiene puerta, en verdadero en caso contrario. Entonces, para cada vértice, de existir una posible restriccion de acceso, su posición permanece en falso.

El siguiente pseudocódigo refleja el comportamiento previamente descrito:


```

prision(conexiones, tengo_llave, puertas, n)
    llegue  $\leftarrow$  false
    cola q
    visitados[0..n]  $\leftarrow$  false
    enEspera[0..n]  $\leftarrow$  false
    encolar(q, 0)
    visitados[0]  $\leftarrow$  true
    while !esVacía(q) and !llegue
        actual  $\leftarrow$  primero(q)
        desencolar(q)
        for i  $\leftarrow$  0 to n and !llegue
            if conexiones[actual][i] and tengo_llave[i] and !visitados[i]
                tengo_llave[puertas[i]]  $\leftarrow$  true
                encolar(q, i)
                visitados[i]  $\leftarrow$  true
                llegue  $\leftarrow$  (i == n - 1)
                if enEspera[puertas[i]]
                    encolar(q, puertas[i])
                    visitados[puertas[i]]  $\leftarrow$  true
                    llegue  $\leftarrow$  (n - 1 == puertas[i])
            else
                if todavía_no_tengo_llave(i)
                    enEspera[i]  $\leftarrow$  true
    return llegue

```

En pocas palabras, la idea del algoritmo es buscar con *bfs* todos los caminos posibles, intentando llegar a el vértice *n*, si se encuentra con un vértice que no es accesible, espera hasta que desde otra rama se habilite ese vértice para poder continuar por ese hilo, mientras al mismo tiempo continúa por todo camino que se encuentre habilitado.

El resultado final viene dado de la variable *llegue*. Es inicializada en falso y se setea en verdadero si sólo si en algún momento *i* == *n* - 1. Si esto ocurre, podemos concluir que pudimos llegar al vértice objetivo (desde la habitación 1 de salida, llego a la *n*, por lo tanto Bernardo se escapó de la prisión). Si pasa por todos los nodos habilitados y los que pudo habilitar y

no puede llegar al vértice n , saldrá del ciclo (**while**) si poder cambiar el valor de *llegue* y devolverá falso.

2.3. Análisis de complejidad

Como mencionamos previamente, el algoritmo **no** podrá pasar más de una vez por cada vértice. Mientras quede algun vértice por ver y pueda accederse, seguirá intentando llegar al nodo n . Analizando la complejidad, vemos que si pudiera acceder a todos los vértices del grafo, tendría un costo de n iteraciones.

Dentro del ciclo principal tenemos las primeras dos operaciones con costo constante (una asignación y quitar de la pila el primer elemento), y un ciclo anidado (*for*).

EXPLICAR BIEN EN DETALLE EL COSTO DEL IF DENTRO DEL FOR-WOWOWOWO

El **while** se ejecuta a lo sumo n veces

2.4. Pruebas y Resultados

3. Mediciones

- Para contar la cantidad aproximada de operaciones definimos una variable inicializada en *cero* la cual incrementamos luego de cada operación.
- Para medir tiempo tenemos una función que ejecuta el algoritmo por un mínimo de tiempo pasado como parametro, esto lo hacemos para obtener una mejor precisión. Una vez que se cumple el tiempo tenemos en una variable la cantidad de veces que se ejecutó el algoritmo, luego dividimos el tiempo medido por *c*.

4. Compilación y ejecución de los programas

Para compilar los programas se puede usar el comando **make** (Requiere el compilador **g++**). Se pueden correr los programas de cada ejercicio ejecutando **./secuencia_unimodal**, **./ciudad** y **./prision** respectivamente.

Los programas leen la entrada de stdin y escriben la respuesta en stdout. Para leer la entrada de un archivo **Tp1EjX.in** y escribir la respuesta en un archivo **Tp1EjX.out** se puede usar:

```
./(ejecutable) <Tp1EjX.in >Tp1EjX.out
```

Para medir los tiempos de ejecución: **./(ejecutable) time**. Devuelve para cada instancia el tamaño seguido del tiempo transcurrido (en segundo) para procesar esa instancia.

Para contar la cantidad de operaciones: **./(ejecutable) count**. Devuelve para cada instancia el tamaño seguido de la cantidad de operaciones de cada instancia.