DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Informe y análisis de resultados.

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Benitti, Raul	592/08	raulbenitti@gmail.com
Mengarda,Lucas		l.j.mengarda@gmail.com
Scarpino, Gino		gino.scarpino@gmail.com
Vallejo, Nicolás		nico_pr08@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

Introducción

En el presente trabajo práctico se intenta resolver ciertos problemas brindados por la cátedra, por medio de algoritmos que resuelvan dichos problemas. Se decidió implementar dichas soluciones usando el lenguaje *Java* y *C++*. Junto con las implementaciones, se adjunta un informe que especifica los detalles sobre el desarrollo de las soluciones, así como pruebas, cálculos de complejidad temporal, mediciones, etc.
Cabe aclarar, que dada la naturaleza de los problemas planteados, se decidió calcular y expresar la complejidad de los algoritmos involucrados, usando el **modelo de costos uniforme**.

1. Problema 1

1.1. Introducción

1.2. Desarrollo

1.2.1. Análisis de complejidad

1.3. Conclusiones

2. Problema 2

2.1. Introducción

2.2. Desarrollo

2.2.1. Análisis de complejidad

2.3. Conclusiones

3. Problema 3

3.1. Introducción

Un aserradero posee una máquina para cortar listones de madera. El costo de cada corte con la máquina es proporcional a largo del listón a cortar en cada momento. Como los cortes se aplican uno a la vez, el orden en que se realicen implica distintas longitudes para los listones resultantes en cada paso y, por lo tanto, distintos costos finales.

Dados un listón inicial de longitud **l** y una serie de **m** cortes a realizar a lo largo de éste, se busca minimizar el costo final del trabajo mediante el reordenamiento de la secuencia en que los cortes deben realizarse. Para clarificar la idea, supongamos que tenemos un listón de longitud 10 y se necesitan cortes en las posiciones 4, 5, 7 y 8. Si no se altera el orden y se procede a cortar, se generará un costo de 10 para el primer corte en 4, un coste de 6 para el corte en 5, un coste de 5 para el corte en 7, y finalmente un costo de 3 al cortar en 8, con un costo total de 24. En cambio, una secuencia posible que minimiza el costo final corresponde a la secuencia 4, 7, 5 y 8: el valor final será 22, ya que el primer corte en 4 cuesta 10, el segundo en 7 cuesta 6, y tanto el corte en 5 como en 8 cuestan 3.

(a) **(b)**

Figura 1: Ejmplo de un listón de largo 10 con cortes requeridos en las marcas 4,5,7 y 8. **(a)** Si se aplica la secuencia de cortes 4,5,7 y 8, el resultado final es un costo de 24. **(b)** En cambio, si la secuencia de cortes es 4, 7, 5 y 8, se obtiene el costo óptimo de 22.

3.2. Desarrollo

En principio intentamos resolver el problema con una idea sencilla y que nos garantiza revisar todas las posibles secuencias de cortes de manera ordena. Pero ejecutar backtracking para secuencias de muchos corte se vuelve impracticable ya que deberíamos revisar todas las posibles permutaciones para secuencias de largo **m**, lo que corresponde al orden $O(m!)$. En la Figura 2, podemos ver el árbol que debería recorrer un algortimo de backtracking para un caso particular. Si observamos dicho árbol con un poco de detalle, podemos apreciar que varias secuencias de corte se repiten y que éstas siempre dan siempre el mismo resultado. Esto se puede apreciar claramente en las hojas del árbol.
Es por ésto que empezamos a pensar que el problema podría ser resuelto con **Programación Dinámica**.

Figura 2: Backtracking para cortes 4,5,7,8; Longitud 10

Comenzaremos por definir una secuencia S de números enteros positivos (incluido el 0) estrictamente creciente. Dada una determinada instancia del problema, en S se

incluirán las marcas de comienzo y fin del listón junto con las m marcas donde los cortes deben realizarse. Utilizaremos la notación $S_{i..j}$, con $j \geq i+1$, para representar un listón que comienza en S_i y termina en S_j. La longitud de dicho listón se obtendrá, entonces, como $S_j - S_i$; mientras tanto, los puntos de corte que le corresponden serán aquellos S_k donde $i < k < j$. Observemos que, asi definido, un listón $S_{i..j}$ no requerirá cortes si $j = i + 1$; éste será el caso que llamaremos trivial.

Consideremos entonces una instancia no trivial del problema: si tomamos un listón $S_{i..j}$ (con $j > i + 1$), existe al menos un valor S_k (con $i < k < j$) donde debemos cortar el listón a fin de obtener el costo óptimo. Como resultado de este corte obtenemos dos nuevos listones, $S_{i..k}$ y $S_{k..j}$, que contiene, respectivamente, las marcas previas y posteriores a S_k (en caso de que éstas existan). El costo mínino total será, entonces, el costo de realizar el corte en S_k (es decir, $S_j - S_i$), más el costo de cortar las restantes marcas en $S_{i..k}$ y $S_{k..j}$: tenemos dos subproblemas del mismo tipo que el problema original.

Supongamos que tenemos el costo mínimo optimo del problema original, y que la solución a alguno de los subproblemas, llamemosla c, no es óptima. Debe existir, sin embargo, una subsolución óptima para ese subproblema, subsolución que agrega un valor c'. Al ser c' el valor óptimo y c un valor no óptimo, podemos asegurar que $c' < c$. Pero entonces podríamos utilizar la subsolucion que suma c' en lugar de c, resultando en un costo final menor que el costo óptimo, y esto contradice nuestra suposición de que teniamos una solución óptima para el problema original. Deducimos que el problema cumple con el principio de optimalidad, pues una solución óptima se forma necesariamente a partir de subsoluciones óptimas a subproblemas del mismo género.

Pasemos ahora a encontrar una función recursiva que devuelva el costo mínimo. Dado un listón de longitud l y una secuencia s que contine los m cortes ordenados de manera creciente, armamos la secuencia $S_{1..m+2} = [0] + s + [l]$. Luego definimos $C(i, j)$ con $1 \leq i < j \leq m + 2$, como el costo correspondiente a cortar cada sub-listón de largo $S_j - S_i$ (considerando S_i como el inicio y S_j como el fin del listón). Obtendremos el costo mínimo para el problema orignal al calcular $C(1, m + 2)$.

Para el caso base consideraremos un listón que no requiere cortes. Esto equivale a $S_{i..j}$ cuando $j = i + 1$,

$$C(i, j) = 0 \text{ si } j = i + 1$$

.

Para el caso recursivo consideramos que se realiza un corte en la posicion $S_k (i < k < j)$ y que al costo de este se le suman los costos de cortar los listones resultantes del primer corte ($S_{i..k}$ y $S_{k..j}$),

$$C(i, j) = S_j - S_i + C(i, k) + C(k, j)$$

Por supuesto, en el caso general puede no existir un único k que pueda ser seleccionado. Por ello, hay que tener en cuenta todos los posibles puntos de corte (los S_k con $i < k < j$, es decir, $j - i - 1$ opciones), y tomar aquel que conduzca a una solución óptima. La formula recursiva que obtememos es:

$$C(i,j) = \begin{cases} 0 & \text{si } j = i+1 \\ S_j - S_i + \min\limits_{i<k<j}\{C(i,k) + C(k,j)\} & \text{si } j \geq i+1 \end{cases}$$

Dado que el problema cumple con el principio de optimalidad y una solución recursiva conlleva la repeticion de cálculos, utilizaremos alguna técnica de programación dinámica para implementar un algoritmo más eficiente. Primero implementaremos una version top-down del algortimo recursivo utilizando memoización, y luego una versión bottom-up no recursiva.

3.2.1. Algortimo Top-Down recursivo con memoización

El concepto clave reside en almacenar en alguna estructura de datos los valores que la función recursiva devuelve para distintos parámetros de entrada. Esto se realiza una única vez para cada gurpo de parametros distintos de tal forma que, si la función es llamada con parámetros para los cuales el cálculo ya fue realizaco y almacenado, se podrá obtener el resultado directamente de la estructura de datos. Para abordar este problema en particular, utilizaremos una matriz de tamaño nxn, donde $n = m + 2$ es la longitud de la secuencia S (m es el número de cortes). La matriz, a la que llamaremos *memo*, contendrá en la posición (i, j) el costo devuelto por $C(i, j)$, es decir, el costo óptimo para realizar los cortes dentro del listón $S_{i..j}$. Será necesario que la matriz *memo* se encuentre inicializada en todas sus posiciones con algún valor que permita decidir si es necesario efectuar o no el cálculo (en el algoritmo usaremos INDEFINIDO). Como veremos luego, si bien nuestro algoritmo utiliza sólo la mitad de la capacidad de la tabla, el resto de las posiciones pueden utilizarse para mantener información que nos permita luego reconstruir la secuencia óptima de cortes que se nos pide en principio.

Algoritmo 3.1

```
COSTO-MINIMO-TOP-DOWN(Liston, i, j)
1  if memo[i][j] == INDEFINIDO
2       if j == i + 1
3            memo[i][j] = 0
4       else
5            costoMinimo = mín{COSTO-MINIMO-TOP-DOWN(S, i, k)
                                +COSTO-MINIMO-TOP-DOWN(S, k, j) : i < k < j}
6            memo[i][j] = S[j] − S[i] + costoMinimo
7  return memo[i][j]
```

$$\begin{vmatrix} * & 0 & 5 & 10 & 15 & 22 \\ * & * & 0 & 3 & 7 & 12 \\ * & * & * & 0 & 3 & 8 \\ * & * & * & * & 0 & 3 \\ * & * & * & * & * & 0 \\ * & * & * & * & * & * \end{vmatrix}$$

Figura 3: Ejemplo de la matriz memo generada para un listón de longitud 10 y cortes en los puntos 4, 5, 7 y 8. Para este caso, $S = [0, 4, 5, 7, 8, 10]$.

3.2.2. Algortimo Bottom-Up iterativo.

El algoritmo previo, al ser del estilo top-down, se basa en resolver una instancia del problema asumiendo que se conocen las soluciones para las subinstancias necesarias (éstas son calculadas en las llamadas recursivas). Lo que haremos ahora será implementar una versión bottom-up: empezamos por las soluciones de las instancias bases para ir armando gradualmente las soluciones de las instancias más grandes hasta llegar a la que nos interesa. Para aplicar esta técnica es menester entender como realizar el llenado de la tabla. En el algortimo anterior podemos observar que para calcular el elemento $memo[i, j]$ necesitamos los valores definidos en la fila i y en la columna j, en ambos casos desde la primera super-diagonal hasta el elemento $memo[i, j]$ (sin contarlo).

$$\begin{vmatrix} * & 0 & 5 & 10 & ? & \\ * & * & & & 7 & \\ * & * & * & & 3 & \\ * & * & * & * & 0 & \\ * & * & * & * & * & \\ * & * & * & * & * & * \end{vmatrix}$$

Figura 4: En este ejemplo se quiere calcular el valor correspondiente a la posicion $memo[1, 5]$. Siguiendo el algortimo top-down, puede verse que solo se necesitan los valores que se muestran en la tabla.

Es interesante notar que, si comenzamos a contar las super-diagonales a partir de 0, en cada super-diagonal d se encuentran los costos de los listones $S_{i..j}$ que tiene d cortes por aplicar. Refiriendonos nuevamente al algortimo top-down y teniendo en cuenta las diagonales, podemos ver que al calcular un elemento en la diagonal d necesitamos tomar los pares de elementos que se encuentran en las diagonales e y f (anteriores a d) tales que $d = e + f - 1$. Por ejemplo, dada una instancia que genere una matriz de $5x5$, para calcular un elemento en la diagonal 3 se requerira examinar los elementos de los pares de diagonales 0 y 2, 1 y 1, 2 y 0.

	0	1	2	3	4	
∗	0	5	10	15	22	4
∗	∗	0	3	7	12	3
∗	∗	∗	0	3	8	2
∗	∗	∗	∗	0	3	1
∗	∗	∗	∗	∗	0	0
∗	∗	∗	∗	∗	∗	

Figura 5: En el ejemplo con $S_{1..6} = 0, 4, 5, 7, 8, 10$, la super-diagonal 0 corresponde a los casos en los que no se necesitan cortes (es decir, $j = i + 1$), como $S_{1.,2}$ o $S_{4.,5}$, mientras que la super-diagonal 4 correponde al problema original (realizar los cortes en 4, 5, 7 y 8)

Llegamos a que la base del algortimo bottom-up estará en llenar la matriz por diagonales, empezando desde el centro hacia afuera y desde arriba hacia abajo (aunque este último orden es irrelevante pués, como vimos, para llenar una diagonal no se necesitan elementos de la diagonal misma).
En este momento vamos a agregar lo necesario para generar la salida requerida; es decir, devolver la secuencia de cortes y no el costo final. Si mantenemos un registro sobre que punto de corte debe seleccionarse a fin de minimizar el costo para cada listón $S_{i..j}$, bastará con recorrer este registro para armar una de las posibles secuencias. Es decir, necesitamos registrar que k que se utiliza para separar la secuencia $S_{i..j}$ en $S_{i..k}$ y $S_{k..j}$ para cada par (i, j). Por lo tanto, utilizaremos una matriz de tamaño nxn (con $n = m+2$) . Al igual que para los costos, solo se utilizara un triángulo de la matriz, sin contar la diagonal principal (pues un punto no representa a un liston) ni la diagonal con elementos (i, j) tales que $j = i + 1$ (dado que estos representan a listones que no necesitan ser cortados). En particular, en nuestro algortimo utilizaremos la misma matriz para almacenar tanto los costos como los k a seleccionar, los primeros en el triángulo superior y los segundos en el inferior. Nos permitimos hacer un abuso en la notacion de la función mínimo en la línea 9, notando como resultado no solo el valor mínimo de los considerados, sino también el indice k que lo genera. Ademas, ahora devolvemos la matriz *memo* completa, pues nos será de utilidad en el último paso de la resolución del ejercicio.

$$\left|\begin{array}{cccccc} * & 0 & 5 & 10 & 15 & 22 \\ * & * & 0 & 3 & 7 & 12 \\ 1 & * & * & 0 & 3 & 8 \\ 1 & 2 & * & * & 0 & 3 \\ 1 & 2 & 3 & * & * & 0 \\ 1 & 3 & 4 & 4 & * & * \end{array}\right|$$

Figura 6: Matriz *memo* generada por el algoritmo bottom-up. En el triángulo superior se encuentran los valores de los costos óptimos para las secuencias $S_{i..j}$. En el triángulo inferior, los valores k minimizaban los costos para las secuencias $S_{j..i}$ (los índices se encuentran espejados).

Algoritmo 3.2 Version bottom-up de costo minimo

```
costoMimimo_BottomUp(Liston)
 1  n = Liston.longitud
 2  for i = 1 to n − 1
 3       j = i + 1
 4       memo[i][j] = 0
 5
 6  for diag = 2 to n − 1
 7       for i = 1 to n − diag
 8            j = i + diag
 9            (k, costoMinimo) = mín{memo[i][k] + memo[k][j] : i < k < j}
10            memo[i][j] = Liston[j] − Liston[i] + costoMinimo
11            memo[j][i] = k
12  return memo
```

3.2.3. Algortimo para calcular la secuencia óptima

Una vez que tenemos la matriz *memo* generada por el algoritmo bottom-up, podemos utilizar el siguiente algoritmo recursivo para reconstruir la secuencia de óptima, la cual es el resultado de invocarlo con los parámetros $i = 1$ y $j = n = Liston.longitud$

Algoritmo 3.3 Generador de secuencia optima a partir de matriz memo
g

```
SECUENCIA-OPTIMA(i, j, Liston, Memo)
1  if j = i + 1
2       return []
3  else
4       k = Memo[j][i]
5       return Liston[k] + SECUENCIA-OPTIMA(i, k, Liston, Memo)
                         +SECUENCIA-OPTIMA(k, j, Liston, Memo)
```

Como se ve, el algoritmo implementado es simplemente BFS. A continuación presentamos una versión iterativa.

Algoritmo 3.4 Version iterativa del generador de secuencia optima a partir de matriz memo

```
SECUENCIA-OPTIMA(Liston, Memo)
 1  cola.encolar(1)
 2  cola.encolar(Liston.longitud)
 3  while cola no es vacia
 4       i = cola.desencolar()
 5       j = cola.desencolar()
 6       if i ≠ j + 1
 7            k = Memo[i][j]
 8            secuenciaOptima.agregar(Liston[k])
 9            cola.encolar(i)
10            cola.encolar(k)
11            cola.encolar(k)
12            cola.encolar(j)
13  return secuenciaOptima
```

3.2.4. Análisis de complejidad

Si observamos un ejemplo del árbol de llamadas que se genera con una implementacion directa de esta formula, notaremos que muchos casos son utilizados mas de una vez: se repiten cálculos. Si llamamos $T(m)$ al tiempo necesario para decidir la mejor forma de realizar m cortes sobre un listón, tenemos que:

$$T(m) = \begin{cases} O(1) & \text{si m} = 0 \\ \sum_{i=0}^{m-1} T(m-i) + T(i) + O(m) = 2 * \sum_{i=0}^{m-1} T(i) + O(m) & \text{si } m \geq 1 \end{cases}$$

donde el termino O(m) provée el costo de decidir cuál debe ser el lugar dónde cortar. Usando el método de sustitución, puede probarse que T(m) = $\Omega(2^m)$; es decir, el algortimo recursivo es exponencial en la cantidad de cortes a realizar.

Realizemos el cálculo de complejidad temporal sobre el algoritmos Bottom-Up. Fácilmente concluiremos que una cota para su complejidad temporal es del orden $O(m^3)$. Ésto se desprende del siguiente anális:
El bucle de la linea 2, que rellena la primera super-diagonal de la matriz con 0, se ejecuta m veces (desde 1 hasta $S.longitud - 1$). El bucle principal , que comienza en la linea 6, se ejecuta $m - 1$ veces (desde 2 hasta $S.longitud - 1$); el siguiente bucle anidado se ejecuta, como máximo, $m - 1$ veces ($d = 2$, i desde 1 a $S.longitud - 2$) . Por último, para calcular el minimo se requieren como maximo $m + 1$ comparaciones ($i = 1$, $d = S.longitud - 1$, $j = S.longitud$, k desde 2 a $S.longitud$). Teniendo en cuenta esto y que las operaciones de asignación, acceso a aleatorio a la matriz, la función min, sumas y restas son operaciones con complejidad constantes y por lo tanto las podemos acotar por con constante c, resulta $c*m*(m-1)*(m+1)$, es decir, $O(m^3)$.

Por ultimo, el Algoritmo 3.4 tiene complejidad $O(m)$: dado un grafo G(V, E), BFS tiene complejidad $O(|E| + |V|)$. En nuestro caso particular, el grafo es un árbol binario donde cada hoja corresponde a un listón que no requiere cortes. Tenemos, por lo tanto, $m + 1$ hojas y m nodos internos. Entonces, resulta $|V| = 2m + 1$, $|V| = 2m$ y la complejidad del algortimo es $O(m)$.

3.3. Tests

Realizamos algunas observaciones sobre los posibles casos a probar. El orden relativo de los elementos no genera familias de casos, porque las secuencias de cortes son siempre es crecientes. La longitud y la cantidad de cortes, por si solos, no son factores de interés. Sin embargo, podría resultar interesante considerar la cantidad de cortes en relación a la longitud del listón y la distribución de la concentración de los cortes a lo largo del listón.

3.4. Conclusiones

En el desarrollo de este ejercicio resultaron especialmente útiles los conceptos teóricos sobre Programación dinamica, pues llegamos a una implementación satisfactoria partiendo de un análisis de este tipo. Es notable la reducción de la complejidad algoritmica que logramos se logro, pasando de $O(2^n)$ a $O(n^3)$. Los casos de prueba no encontramos familias de casos que produjeran un mejor o peor rendimiento.

4. Problema 1

4.1. Introducción

4.2. Desarrollo

4.2.1. Análisis de complejidad

4.3. Conclusiones

Referencias