CC40A: Diseño y Análisis de Algoritmos
Pauta Control 1 - Semestre Otoño 2000

Profesor: Ricardo Baeza Yates
Prof. Auxiliar: Felipe Saint-Jean

P1 (1.5 ptos.)

El algoritmo tradicional para encontrar el mínimo de un arreglo supone que el primer elemento es el resultado final y luego recorre los elementos restantes cambiando el mínimo cada vez que se encuentra un valor menor. Plantee una recurrencia para el número promedio de veces que al mínimo le asignamos un valor si cualquier elemento de los n tiene la misma probabilidad de ser el mínimo. Resuelva la recurrencia e indique el orden de la solución.

Por inducción, la probabilidad de que el último elemento sea el mínimo es $1/n$. Luego, $T(n) = 1/n + T(n-1)$. El primer elemento es siempre asignado, por lo tanto $T(1) = 1$. Resolviendo por sustitución sucesiva se obtiene

$$T(n) = \sum_{i=1}^{n} \frac{1}{i} \leq \int_{x=1}^{n+1} \frac{dx}{x} = \ln(n+1) = O(\log n)$$

P2 (1.5 ptos.)

Suponga que tiene dos strings a y b, de largos n y m, respectivamente. Decimos que b es una subsecuencia de a, si todos los elementos de b aparecen en el mismo orden en a.

(0.5) *Diseñe un algoritmo de tiempo $O(n)$ para determinar si b es una subsecuencia de a.*

Como lo importante es el orden en b y a, Basta recorrer a buscando los elementos de b, es decir:

```
if( m > n ) return( NOES)   // b no puede ser mas largo que a
i=1;
for( j=1; j <= n; j++ )     // recorremos a
   if( b(i) == a(j) ) i++;  // avanzamos en b
if( i == m+1 ) return( ES ) // encontramos todo b
else           return( NOES )
```

Claramente este algoritmo toma tiempo $O(n+m)$ y como $m \leq n$ si entra en el ciclo, el tiempo es $O(n)$.

(1.0) *Se define b^i como cada elemento de b repetido i veces. Por ejemplo, $(xy)^3 = xxxyyy$. Diseñe un algoritmo eficiente para encontrar el valor máximo de i tal que b^i es subsecuencia de a.*

Dada la definición de b^i, notar que b^j es subsecuencia de b^i para $j < i$. Entonces, si b^i es subsecuencia entonces lo es cualquier b^j con $j < i$. Del mismo modo, si b^i no es subsecuencia, tampoco puede ser subsecuencia b^k, para $k > i$. Por lo tanto, como el i tampoco puede ser mayor a $\lfloor n/m \rfloor$, podemos usar búsqueda binaria para encontrar el i máximo. Si es subsecuencia, usamos el segmento superior y sino el inferior. Luego la complejidad es $O(n \log(n/m))$. Una solución de $O(n^2/m)$ tiene la mitad del puntaje.

P3 (1.5 ptos.)

Suponga que tiene un arreglo A de n números enteros. Diseñe un algoritmo usando inducción que encuentre la subsecuencia más larga de A tal que sus valores son ascendentes, es decir, si la subsecuencia está en las posiciones $i_1 < ... < i_k$, entonces $A[i_1] \leq .... \leq A[i_k]$. Trate de implementarlo en forma eficiente, usando el menor espacio posible e indique la complejidad temporal y espacial de su solución.

Hay muchas formas de resolver este problema. La más sencilla es usar algo similar al problema de la subsecuencia común más larga. Sea $S_{i,j}$ el largo de la subsecuencia ascendente más larga entre $A[i]$ y $A[j]$ inclusives ($1 \leq i \leq j \leq n$). Luego, $S_{i,i}$ es 1 y la subsecuencia asociada es $A[i]$. Para encontrar $S_{i,j}$ suponemos que ya tenemos $S_{i,k}$ para $k < j$ y $S_{k,j}$ para $k < i$. Luego, la solución compuesta será válida si el elemento mayor del lado izquierdo es menor o igual al menor del lado derecho. Es decir,

$$S_{i,j} = \max_{i \leq k < j \text{ y } \mathrm{A[u(i,k)]} \leq \mathrm{A[p(k+1,j)]}} (S_{i,k} + S_{k+1,j})$$

donde $u(i,k)$ es el último elemento de la subsecuencia de $S_{i,k}$ y $p(k+1,j)$ es el primer elemento de la subsecuencia de $S_{k+1,j}$. Esto se puede implementar eficientemente usando programación dinámica, calculando $S_{i,i+j}$ desde $j = 0$ hasta $j = n - i$, vía diagonales. El espacio que necesitamos es $O(n^2)$ (la mitad de una matriz que contiene el largo más los valores iniciales y finales) y el tiempo del algoritmo será $O(n^3)$ ya que hay dos ciclos y calcular el máximo interno tarda $O(n)$. La solución luego se puede recuperar en tiempo $O(n^2)$.

Esto se puede hacer en mejor tiempo y espacio. En esos casos se dará 0.5 ptos extra,

P4 (1.5 ptos.)

Roquepato ha decidido invertir M pesos en acciones de la compañia Monoposoft en NASDAQ.

(0.5) *Calcule el monto final para un algoritmo óptimo que maximiza la ganancia y que sabe de antemano los valores de la acción $v_1, ..., v_d$ en d días.*

Por supuesto lo que conviene es comprar lo más barato posible y vender cuando valga más. Esto puede pasar varias veces porque hay mínimos y máximos locales. Una forma sencilla de expresar la ganancia es que por cada día que el precio sube, se gana $M(v_i - v_{i-1})/v_i$. Luego, considerando sólo las diferencias positivas, el monto final del algoritmo óptimo el día d será $O_d = M + M \prod_{1<i\leq d \text{ y } \mathrm{v_i > v_{i-1}}} (v_i - v_{i-1})/v_i$.

(1.0) *Diseñe un algoritmo en línea para que Roquepato compre/venda acciones y calcule la competitividad del mismo al final de d días.*

Hay muchos algoritmos posibles. Hay estrategias de personas que compran cuando el precio sube, otras cuando el precio baja. Sea E_i el dinero en efectivo y A_i el dinero en acciones el día i. Inicialmente $E_0 = M$ y $A_0 = 0$. El monto final será $E_d + A_d$. Como el problema es maximizar el monto, debemos definir la competitividad al revés para que sea mayor a 1. Es decir, $c = O_d/(E_d + A_d)$. Notar que c es no acotado si perdemos todo el dinero.

Un algoritmo concreto podría estar basado en el precio promedio ponderado p_i al cual compre las acciones que tengo hasta el día i, usando inicialmente $p_0 = v_1$. Si el día i el precio v_i es menor o igual a p_{i-1}, decido comprar la mitad de mi efectivo en acciones ya que están baratas (si $E_{i-1} > 0$). En ese caso $A_i = A_{i-1} + E_{i-1}/2$, $E_i = E_{i-1}/2$ y $p_i = A_{i-1}\ p_{i-1} + v_i\ E_{i-1}/2$ es el nuevo precio ponderado. Por otro lado, si $v_i > p_{i-1}$, vendo la mitad de mis acciones si $A > 0$ (la otra mitad la guardo esperando que suban aún más). En este caso, $A_i = A_{i-1}/2$, $E_i = E_{i-1} + A_{i-1}/2 \times v_i/p_{i-1}$ y p no cambia. Si no puedo ni comprar ni vender en los casos anteriores, entonces no hago nada y $E_i = E_{i-1}$, $A_i = A_{i-1}$ y $p_i = p_{i-1}$.