Pontificia Universidad Católica de Chile
Facultad de Ingenieria
Departamento de Ciencias de la Computación
IIC2283 - Diseño y Análisis de Algoritmos

Ayudantía 5

14 de septiembre, 2018

Ayudante: Andrés Ríos

Problema 1

Dada una secuencia $X = \langle x_1, x_2, ..., x_m \rangle$, otra secuencia $Z = \langle z_1, z_2, ..., z_k \rangle$ es una **subsecuencia** de X si existe una secuencia estrictamente creciente de indices $\langle i_1, i_2, ..., i_k \rangle$ tal que $\forall j = 1, 2, ..., k \;\; x_{i_j} = z_j$.

Dadas dos secuencias $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$, entregue un algoritmo que compute el largo de la subsecuencia más larga común a X e Y. Analice la complejidad del algirtmo.

Solución:

Sea X_i el prefijo de X de largo i.

Dado $X = \langle x_1, x_2, ..., x_m \rangle$, $Y = \langle y_1, y_2, ..., y_n \rangle$ y $Z = \langle z_1, z_2, ..., z_k \rangle$, donde Z es una subsecuencia de tamaño máximo (LCS de ahora en adelante) de X e Y, tenemos que

1. Si $x_m = y_n$, entonces $z_k = x_m = y_n$ y Z_{k-1} es una LCS de X_{m-1} y Y_{n-1}
2. Si $x_m \neq y_n$, entonces $z_k \neq x_m$ implica que Z es una LCS de X_{m-1} e Y
3. Si $x_m \neq y_n$, entonces $z_k \neq y_n$ implica que Z es una LCS de X e Y_{n-1}

Sea $LCS(i, j)$ el largo de la secuencia más larga entre X_i e Y_j. Por la propiedad anterior podemos definir la recurrencia

$$LCS(i,j) = \begin{cases} 0 & i = 0 \vee j = 0 \\ LCS(i-1, j-1) & i, j > 0 \wedge x_i = y_j \\ max(LCS(i, j-1), LCS(i-1, j)) & i, j > 0, x_i \neq y_j \end{cases}$$

Con esto, el algoritmo es:

```
function LCS(X, Y, m, n)
    c[1...m, 1...n] = [0...0, 0...0
    for i := 1 upto m do
        c[i, 0] = 0
    for j := 1 upto n do
        c[0, j] = 0
    for i := 1 upto n do
        for j := 1 upto m do
            if x_i == y_j then
                c[i, j] := c[i − 1, j − 1] + 1
            else if c[i − 1, j] ≥ c[i, j − 1] then
                c[i, j] = c[i − 1, j]
            else
                c[i, j] = c[i, j − 1]
    return c[m, n]
```

El algoritmo corre en tiempo $\Theta(|X| \cdot |Y|)$.

Problema 2

El problema de la mochila continuo se define de la siguente manera: Se entrega una lista de objetos con capacidades y valores, junto con una capacidad máxima. El objetivo es colocar los objetos en una mochila de manera de maximizar el valor de los objetos adentro de la mochila pero sin exceder la capacidad máxima. Se pueden colocar fracciones de los objetos.

Formalmente, dado un arreglo $W[1...n]$ con pesos, un arreglo $V[1...n]$ con valores y una capacidad máxima B, se busca

$$\begin{aligned} \max_{I[1...n]} \quad & \sum_{i=1}^{n} V[i] * I[i] \\ s.t. \quad & \sum_{i=0}^{n} W[i] * I[i] \leq B \\ & 0 \leq I[i] \leq 1 \quad \forall i \end{aligned}$$

Entregue un algoritmo que solucione el problema de la mochila continuo. Analice la complejidad del algoritmo.

Idea de Solución:

Se define el arreglo $R[1...n]$ con los números de 1 a n. Luego R se ordena de mayor a menor según el valor de $\frac{V[i]}{W[i]}$ (En el arreglo R, x está a la izquierda de y si y solo si $\frac{V[x]}{W[x]} \geq \frac{V[y]}{W[y]}$). Luego basta elegir los elementos en el orden definido por R. El algoritmo es:

function CKSP(W, V, B)
  $R[1...n] = [1, 2, ..., n]$
  Ordenar R según el valor de $-\frac{V[i]}{W[i]}$
  $T = 0$
  $i = 1$
  while $B > 0$ and $i \leq n$ **do**
    if $B > W[R[i]]$ **then**
      $B = B - W[R[i]]$
      $T = T + V[R[i]]$
      $i = i + 1$
    else
      $T+ = \frac{B}{W[R[i]]} \cdot V[R[i]]$
      $B = 0$
  return T

Para demostrar la optimalidad, basta darse cuenta que dada una solución óptima, uno siempre puede quitar peso de items con peor razón $\frac{V[i]}{W[i]}$ y agregárselo a items con mejor o igual razón $\frac{V[i]}{W[i]}$ hasta llegar a la solución entregada por nuestro algoritmo codicioso. Además, este proceso solo puede mantener la solución igual o mejorarla. Por lo tanto, la solución entregada por el algoritmo codicioso es óptima.

Respecto a la complejidad, el tiempo de ejecución está dominado por ordenar R. Por lo tanto, el algoritmo corre en $O(n \log(n))$.

Problema 3

El problema de la mochila se define de la misma manera que el problema anterior, pero ahora no se pueden colocar fracciones de los objetos: o se colocan enteros en la mochila o no se colocan en absoluto.

Formalmente, dado un arreglo $W[1...n]$ con pesos, un arreglo $V[1...n]$ con valores y una capacidad máxima B, se busca

$$\begin{aligned} \underset{I[1...n]}{max} & \sum_{i=1}^{n} V[i] * I[i] \\ s.t. & \sum_{i=0}^{n} W[i] * I[i] \leq B \\ & I[i] \in \{0,1\} \quad \forall i \end{aligned}$$

Entregue un algoritmo que solucione el problema de la mochila. Analice la complejidad del algoritmo.

Idea de Solución:

Sea $KSP(W, V, B, i)$ una función que indica la máxima utilidad que se puede obtener con objetos de valor V y peso W, capacidad B y considerando hasta el ítem i-ésimo. Podemos definir KSP por la siguiente recurrencia:

$$KSP(W,V,B,i) = \begin{cases} 0 & i = 0 \\ max(KSP(W,V,B-W[i],i-1)+V[i], KSP(W,V,B,i-1)) & W[i] \leq B \\ KSP(W,V,B,i-1) & W[i] > B \end{cases}$$

Si no tenemos suficiente capacidad, la única opcion es no meter el ítem a la mochila. Por el otro lado, si nuestra capacidad es mayor al peso del item entonces el óptimo es el máximo entre meter el objeto a la mochila o no hacerlo, sumado a la solución del subproblema resultante en cada caso.

La recurrencia anterior se traduce en el siguiente algoritmo de programación dinámica:

```
function KSP(W, V, B)
    n = W.length
    M[0...n, 0...B] = [0, ..., 0]
    for i = 1 upto n do
        for b = 0 upto B do
            if W[i] ≤ b then
                M[i, b] = max(M[i − 1, b − W[i]] + V[i], M[i − 1, b])
            else
                M[i, b] = M[i − 1, b]
    return M[n, B]
```

El algoritmo corre en tiempo $O(n \cdot B)$, con n siendo el número de objetos.

Problema 4

Se tienen dos arreglos A y B de números positivos, de largos n y m respectivamente. Se puede realizar la siguiente operación un número arbitrario de veces: se toma un subarreglo continuo de un arreglo y se reemplaza por la suma de los elementos en el subarreglo. Por ejemplo, del arreglo $[1, 10, 100, 1000, 10000]$ se puede obtener el arreglo $[1, 1110, 10000]$, $[11, 100, 1000, 10000]$, $[1, 10, 11100]$, etc.

Dos arreglos A y B son iguales si tienen el mismo largo y para todo i se cumple que $A[i] = B[i]$.

Dado dos arreglos A y B, se quiere realizar la operación descrita anteriormente hasta que ambos arreglos sean iguales. Más aún, se busca que al terminar el largo de A y B sea máximo.

Entregue un algoritmo que dado dos arreglos A y B retorne el largo máximo de los arreglos tras realizar la transformación descrita anteriormente, o `NO` si no es posible hacer esto. Analice la complejidad del algoritmo.

Idea de Solución:

Notar que si la suma de los arreglos A y B es distinta, entonces es imposible hacer que sean iguales colapsando subarreglos. Por el otro lado, si las sumas son iguales entonces siempre es posible hacer que A y B sean iguales (en el peor caso, se podría colapsar ambos arreglos a un solo elemento).

Consideremos el siguiente algoritmo greedy:

```
function MAXLENGTH(A, B)
    n = A.length
    m = B.length
    if sum(A) ≠ sum(B) then
        return NO
    length = 0
    i = 0
    j = 0
    while i < n do
        s1 = A[i]
        s2 = B[i]
        i = i + 1
        j = j + 1
        while s1 ≠ s2 do
            if s1 < s2 then
                s1 = s1 + A[i]
                i = i + 1
            else
                s2 = s2 + B[i]
                j = j + 1
        length = length + 1
    return length
```

Lo que se hace es sucesivamente tomar los prefijos de A y B más pequeños que sumen lo mismo. Luego esos prefijos se colapsan a un solo número, por lo que se le suma 1 al largo de la solución.

Falta demostrar que el algoritmo es correcto. Primero, notemos que si un cierto subarreglo fue colapsado, entonces podemos asumir que el elemento resultante no es colapsado nuevamente (sería equivalente a colapsar un subarreglo más grande inicialmente). Con esto, los distintos índices donde se colapsa el arreglo generan *particiones*.

Sean $i_1, i_2, ..., i_l$ las particiones de A por nuestro algoritmo `MaxLength`, y sean $j_1, j_2, ..., j_k$ las particiones óptimas, con $k > l$. Sea y el mínimo número tal que $i_y \neq j_y$. Notar que $i_y < j_y$ por el funcionamiento del algoritmo. ¡Tenemos que $j_1, j_2, ..., j_{y-1}, i_y, j_y, ..., j_k$ define una partición mejor que la óptima! Se concluye que la solución óptima no puede ser mejor que la solución encontrada por el algoritmo `MaxLength`.

Sobre la complejidad del algoritmo, se tiene que i y j aumentan monotónicamente desde 0 hasta n y m, respectivamente. Además, se hace una cantidad constante de operaciones por cada aumento de i y j. Por lo tanto, el algoritmo es $O(n + m)$.