Práctica 8

Ejercicio 1.a)

Queremos demostrar que nuestra solución greedy encuentra la solución óptima a nuestro problema, es decir, maximiza el número de ficheros que pueden guardarse en un disquete de capacidad d.

Sea $F = \{f_1, f_2, \ldots, f_k\}$ el conjunto de archivos seleccionados por nuestro algoritmo greedy, en el orden en que fueron elegidos. Sea $O = \{f'_1, f'_2, \ldots, f'_m\}$ una solución óptima donde los tamaños de los ficheros están ordenados de menor a mayor.

Sea $Size(S)$ el tamaño de memoria ocupada por los ficheros en un conjunto S. Según el orden elegido tanto en F como en O se tiene que:

$$\begin{gathered} Size(\{f_1\}) \leq Size(\{f_2\}) \leq \ldots, Size(\{f_k\}) \\ \wedge \\ Size(F) \leq d, \text{ donde } d \text{ es la capacidad del disquete} \\ \wedge \\ Size(\{f'_1\}) \leq Size(\{f'_2\}) \leq \ldots, Size(\{f'_m\}) \\ \wedge \\ Size(O) \leq d \end{gathered}$$

Debemos mostrar por inducción.

Como caso base tomamos m=1. Como tomamos en nuestro algoritmo greedy el fichero de menor tamaño, entonces debemos tener el caso que $Size(\{f_1\}) \leq Size(\{f'_1\}) \leq d$.

Asumimos que vale para $m = t$ y probaremos que vale para $m = t + 1$. Por hipótesis inductiva sabemos entonces que:

$$Size(\{f_1, f_2, \ldots, f_t\}) \leq Size(\{f'_1, f'_2, \ldots, f'_t\}) \leq d$$

Por esto, si un fichero puede ser agregado a la solución óptima, también debería ser válido agregarlo a nuestra solución greedy.

$$Size(\{f_1, f_2, \ldots, f_t, f'_{t+1}\}) \leq Size(\{f'_1, f'_2, \ldots, f'_t, f'_{t+1}\}) \leq d$$

Pero como en nuestra solución greedy siempre agregamos el fichero de menor tamaño de los que restan, tenemos que:

$$Size(\{f_1, f_2, \ldots, f_t, f_{t+1}\}) \leq Size(\{f'_1, f'_2, \ldots, f'_t, f'_{t+1}\}) \leq d$$

Tenemos entonces para todo $r \leq k$, $Size(\{f_1, f_2, \ldots, f_r\}) \leq Size(\{f'_1, f'_2, \ldots, f'_r\}) \leq d$.

Si F no fuese óptima, tendríamos $m > k$, por lo cual existiría un fichero f_{k+1} en O que no pertenecería a F. Debería ocurrir que:

$$Size(\{f_1, f_2, \ldots, f_k\}) + Size(\{f'_{k+1}) \leq d$$

pero tenemos que:

$$Size(\{f_1, f_2, \ldots, f_k\}) \leq Size(\{f'_1, f'_2, \ldots, f'_k\})$$

entonces sería válido:

$$Size(\{f_1, f_2, \ldots, f_k\}) + Size(\{f'_{k+1}\}) \leq d$$

por lo cual dicho fichero tendría que haber sido elegido por nuestra solución greedy. Por lo tanto, llegamos a una contradicción!

La solución F encontrada por nuestra algoritmo greedy es entonces solución óptima de nuestro problema. O sea, la cantidad de ficheros a guardar en el disquete que nos propone nuestra solución greedy es la máxima posible.

Ejercicio 1.b)

Este algoritmo greedy no encuentra siempre una solución óptima al problema. Lo mostraremos con un ejemplo:

Sean la capacidad de disquete $d = 7$ y 3 ficheros f_1, f_2 y f_3 disponibles para ser guardados en el mismo, donde:

$$Size(f_1) = 6$$
$$Size(f_2) = 2$$
$$Size(f_3) = 5$$

Esta solución propone ordenar los ficheros teniendo en cuenta el orden decreciente de sus tamaños. O sea, el orden para este ejemplo sería:

$$Size(f_1) \geq Size(f_3) \geq Size(f_2)$$

Una vez ordenados, la solución propone ir guardando los ficheros, comenzando con el de mayor tamaño y siguiendo el orden decreciente de los tamaños hasta que no quede espacio suficiente para seguir agregando.

Para este ejemplo particular la solución que propondría el algoritmo greedy sería entonces guardar únicamente el fichero f_6 y la memoria que ocuparía esta solución sería de 6. Puede verse fácilmente que si guardamos los ficheros f_2 y f_3 la memoria ocupada en el disquete

sería 7.

Entonces, podemos concluir que este algoritmo greedy no garantiza un resultado óptimo.

Ejercicio 10
Implementación en C de algoritmo LCS usando programación dinámica:

```
int LCS(char * seq1, int n, char * seq2, int m, int ** res )
{   // res es la matriz que se usa para guardar los resultados
    // ya calculados. Esta matriz tiene como tamanio n x m,
    // siendo n y m las longitudes de las cadenas seq1 y seq2
    // respectivamente

    int result;
    if ((n==0)||(m==0)) return 0;
    else if (res[n][m]!=-1) return res[n][m];     <-- (*)
    else if (seq1[n]==seq2[m]) result= 1 + LCS(seq1,n-1,seq2,m-1,res);
    else result = max(LCS(seq1,n-1,seq2,m,res), LCS(seq1, n, seq2, m-1,res));
    res[n][m] = result; <-- (**)
    return result;

}
```

Complejidad temporal

Nuestros cálculos son guardados en la matriz res en la línea señalada con (**). De esta forma, nos aseguramos de hacer nuevas llamadas recursivas si no hemos ya computado el resultado en la línea de código marcada con (*).

Esta versión (que posee memoria) tiene un tiempo de ejecución de $O(mn)$. Una forma simple de ver esto es como sigue. Primero, notemos que alcanzamos la línea de código (**) como máximo mn veces (como mucho una vez por cada par de valores). Esto quiere decir que hacemos como máximo $2mn$ llamadas recursivas en total (al menos dos llamadas recursivas cada vez que alcanzamos dicha línea).

Complejidad espacial

Esta solución necesita utilizar una memoria para guardar los resultados ya calculados. Para eso utiliza una matriz de tamaño mn, por lo tanto, la complejidad espacial es de $O(mn)$.