Examen de Programación 3
15 de julio de 2022

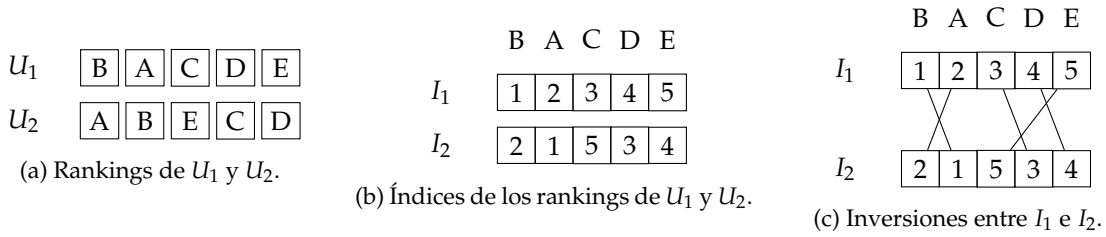
En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

Ejercicio 1 (32 puntos)

La empresa de entretenimientos *Fingflix* desea incorporar una nueva funcionalidad con el objetivo de aumentar las reproducciones de los contenidos que ofrece, al tiempo de fidelizar a sus usuarios. La misma se llamará *Sugeridos* y se trata de una lista de contenidos que se le sugieren al usuario, basándose en su experiencia de uso previa y la similitud que presenta con la experiencia de otros usuarios. La estrategia se basa en comparar los *rankings* otorgados por usuarios distintos a contenidos iguales para encontrar similitudes.

Se requiere implementar un algoritmo que calcule la similitud del ranking de un usuario $U_i, i \neq 1$ con el del usuario U_1. A tales efectos, se considerará que la entrada es un arreglo de naturales I_i que contiene los índices del *ranking* de un usuario U_i y la salida es la similitud entre el ranking de ese usuario con el de U_1.

La similitud se computa contando la cantidad de inversiones en el arreglo I_i tal como se ve en la cantidad de cruces en la Figura 1c. Por ejemplo, la similitud entre U_1 y U_2 es 3.



(a) Rankings de U_1 y U_2.

(b) Índices de los rankings de U_1 y U_2.

(c) Inversiones entre I_1 e I_2.

Figura 1: Ejemplo de instancia del problema.

(a) Dé un algoritmo de tipo *divide y vencerás* que calcule la cantidad de inversiones en un arreglo de naturales. **Sugerencia:** siga una estrategia similar a la del algoritmo *MergeSort*.

(b) Demuestre que su algoritmo admite una implementación con tiempo de ejecución $O(n \log n)$.

Solución:

(a) Implementación

```
1   // retorna la cantidad de inversiones en un arreglo.
2   int inversiones (int *Ii, int n) {
3       return mergeSort_Inv(Ii,0,n-1);
4   }
5
6   int mergeSort_Inv (int *A, int inicio, int fin) {
7       int inv = 0;
8       if (fin-inicio > 0){
9           int medio =  (inicio + fin) / 2;
10          inv = mergeSort_Inv(A,inicio,medio) ;
11          inv += mergeSort_Inv(A,medio+1,fin);
```

```
12                  inv += merge_Inv(A,inicio,medio,fin);
13              }
14          }
15
16          int merge_Inv (int *A, int ini, int medio, int fin) {
17              int arrAux[fin-ini+1];
18              int ini1 = ini; int ini2 = medio+1;
19              int fin1 = medio; int fin2 = fin;
20              int posAux = inv = 0;
21
22              // Mientras haya elementos para intercalar.
23              while (ini1<=fin1) && (ini2<=fin2){
24                  if (A[ini1] < A[ini2]){
25                      arrAux[posAux] = A[ini1];
26                      ini1++;
27                  }
28                  else { // inversión
29                      arrAux[posAux] = A[ini2];
30                      ini2++;
31                      inv += fin1-ini1+1;
32                  }
33                  posAux++;
34              }
35
36              // Si quedan elementos en la primera mitad.
37              while (ini1<=fin1){
38                  arrAux[posAux] = A[ini1];
39                  ini1++;
40              posAux++;
41              }
42
43              // Si quedan elementos en la segunda mitad.
44              while (ini2<=fin1){
45                  arrAux[posAux] = A[ini2];
46                  ini2++;
47              posAux++;
48              }
49
50              // Copiar los elementos ordenados al vector A.
51              for (posAux=0; posAux<(fin-ini+1); posAux++)
52                  A[posAux] = arrAux[posAux];
53
54          }
```

(b) Tiempo de ejecución

- *merge_Inv*
  Cada mitad del arreglo se recorre una única vez -realizando operaciones de costo constante sobre cada dato, y luego se recorre el arreglo completo para actualizarlo con los datos ordenados.
  $T_{merge\_Inv}(n) \leq cn = O(n)$
- *mergeSort_Inv*
  En el peor caso, el tiempo de ejecución de *mergeSort_Inv* respeta la recurrencia:

$$\begin{cases} T_{mergeSort\_Inv}(1) = c \\ T_{mergeSort\_Inv}(n) \leq 2T_{mergeSort\_Inv}(\frac{n}{2}) + T_{merge\_Inv}(n) + c \end{cases}$$

  Expandiendo la recursión se llega a:
  $T_{mergeSort\_Inv}(n) \leq 2T_{mergeSort\_Inv}(\frac{n}{2}) + cn + c =$
  $2(2T_{mergeSort\_Inv}(\frac{n}{4}) + \frac{n}{2}c + c) + nc + c = 4T_{mergeSort\_Inv}(\frac{n}{4}) + 2nc + 3c =$
  $4(2T_{mergeSort\_Inv}(\frac{n}{8}) + \frac{n}{4}c + c) + 2nc + 3c = 8T_{mergeSort\_Inv}(\frac{n}{8}) + 3nc + 7c =$
  ...
  $T_{mergeSort\_Inv}(n) \leq 2^k T_{mergeSort\_Inv}(\frac{n}{2^k}) + knc + (2^k - 1)c$

El paso base se alcanza cuando $\frac{n}{2^k} = 1 \implies n = 2^k \implies k = log_2(n)$.
Sustituyendo obtenemos: $T_{mergeSort\_Inv}(n) \leq nT_{mergeSort\_Inv}(1) + log_2(n)nc + (n-1)c = O(nlog_2(n))$.

- inversiones
  $T_{inversiones}(n) = T_{mergeSort\_Inv}(n) = O(nlog_2(n))$

Otra versión posible:

- *merge_Inv*
  Cada mitad del arreglo se recorre una única vez -realizando operaciones de costo constante sobre cada dato, y luego se recorre el arreglo completo para actualizarlo con los datos ordenados.
  $T_{merge\_Inv}(n) \leq cn = O(n)$
- *mergeSort_Inv*
  En el peor caso, el tiempo de ejecución de *mergeSort_Inv* respeta la recurrencia:

$$\begin{cases} T_{mergeSort\_Inv}(1) = c \\ T_{mergeSort\_Inv}(n) \leq 2T_{mergeSort\_Inv}(\frac{n}{2}) + T_{merge\_Inv}(n) + c \end{cases}$$

Este problema es análogo al tratado en la sección 5.3 del libro de referencia (ver los algoritmos `Merge-and-Count` y `Sort-and-Count`). Por lo tanto valen los argumentos que conducen a la relación de recurrencia (5.1) para cuya solución se muestran varias técnicas, concluyéndose que el orden O del tiempo de ejecución es $O(nlog_2(n))$.

Ejercicio 2 (40 puntos)

Un *palíndromo* es una cadena de caracteres que se lee igual de izquierda a derecha que de derecha a izquierda; por ejemplo, *anilina*.

Dada una cadena de caracteres, $x = x_1, x_2, \ldots, x_n$, una *subsecuencia* de x es una cadena de la forma $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, con $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. En palabras, una subsecuencia de x está formada por la concatenación de caracteres de x, no necesariamente contiguos pero preservando su orden relativo. Por ejemplo, la cadena *ABCDDABE* contiene como subsecuencias a *ABA* y también a *BDB* (entre otras), que son palíndromos de largo 3.

Dada una cadena de caracteres no vacía, x, queremos obtener el largo de las subsecuencias más largas de x que son palíndromos.

Se pide:

(a) Defina una relación de recurrencia que permita calcular el largo máximo entre todas las subsecuencias de x que son palíndromos. Justifique su respuesta explicando la procedencia de cada término.
Sugerencia: Considere una función de la forma $OPT(i, j)$, que resuelve el problema para $x_i, \ldots, x_j$.

(b) Dé un algoritmo eficiente **iterativo**, usando la técnica de Programación Dinámica, para calcular el largo máximo entre todas las subsecuencias de x que son palíndromos.

Solución:

(a) Para $1 \leq i \leq j \leq n$ definimos $OPT(i, j)$ como el máximo largo entre todas las subsecuencias de $x_i, \ldots, x_j$ que son palíndromos.

Las cadenas de largo 1 son en sí mismas palíndromos, por lo cual la mayor subsecuencia que es un palíndromo tiene largo 1, como establece (1). Por otra parte, la subsecuencia más larga de una cadena vacía tiene largo cero, de donde se desprende (**??**). Estas dos ecuaciones definen el paso base de la recurrencia.

Consideremos ahora una cadena $S = x_i, \ldots, x_j$ con al menos dos caracteres, y sea $P = x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ una subsecuencia de largo máximo entre todas las subsecuencias de S que son palíndromos.

Si $x_i = x_j$, afirmamos que existe una subsecuencia de S, P', que es un palíndromo del mismo largo que P e incluye tanto a x_i como a x_j. En efecto, en P tenemos o bien $i_1 = i$, o bien $i_k = j$, porque en caso contrario $x_i P x_j$ sería una subsecuencia de S que además es un palíndromo de largo mayor que P. Si $i_1 = i$, obtenemos P' reemplazando i_k por j, y si $i_k = j$, obtenemos P' reemplazando i_1 por i.

La subcadena $x_{i_2}, \ldots, x_{i_{k-1}}$ de P' es una subsecuencia de $x_{i+1}, \ldots, x_{j-1}$, que es de largo máximo entre todas sus subsecuencias que son palíndromos, de donde surge el segundo término de (2), al cual se suman dos unidades correspondientes a los caracteres x_i y x_j que forman los extremos de P'.

Si en cambio $x_i \neq x_j$, tenemos o bien $i_1 > i$, o bien $i_k < j$, porque en caso contrario P no sería un palíndromo. Si $i_1 > i$, entonces P es una subsecuencia de $x_{i+1}, \ldots, x_j$ cuyo largo es como máximo $OPT(i+1, j)$. Si $i_k < j$, entonces P es una subsecuencia de $x_i, \ldots, x_{j-1}$ cuyo largo es como máximo $OPT(i, j-1)$. Como P es de largo máximo, su largo debe coincidir con el máximo de estos dos valores, como expresa la ecuación (3).

$$OPT(i,i) = 1, \quad 1 \leq i \leq n, \tag{1}$$

$$OPT(i,j) = 0, \quad 1 \leq j < i \leq n, \tag{2}$$

$$OPT(i,j) = 2 + OPT(i+1, j-1), \quad 1 \leq i < j \leq n, x_i = x_j, \tag{3}$$

$$OPT(i,j) = \text{máx}\left\{OPT(i, j-1), OPT(i+1, j)\right\}, \quad 1 \leq i < j \leq n, x_i \neq x_j. \tag{4}$$

```
1  Algorithm MaximoPalindromo
       /* Incialización según paso base definido en (1) y (??)                    */
2      for i = 1 to n do
3          OPT [i, i] = 1
           /* Incialización según (??). Alternativamente es suficiente definir
              OPT[i, i − 1] = 0 para i > 1.                                        */
4          for j = 1 to i − 1 do
5              OPT [i, j] = 0
6          end
7      end
8      for i = n − 1 downto 1 do
9          for j = i + 1 to n do
               /* Se aplica (2) o (3) según corresponda                           */
10             if x_i = x_j then
11                 OPT [i,j] = 2 + OPT [i + 1, j − 1]
12             else
13                 OPT [i,j] = máx{OPT[i + 1, j], OPT[i, j − 1]}
14             end
15         end
16     end
17     return OPT [1, n]
18 end
```

Figura 2: Algoritmo para encontrar el largo maximo de una subsecuencia palindromica.

(b)

Ejercicio 3 (28 puntos)

Un cocinero cuenta con n ingredientes y desea realizar un plato utilizando la mayor cantidad posible de ellos. Sin embargo, no todos los ingredientes van bien juntos, por lo que cuenta con una matriz M simétrica de tamaño $n \times n$, que indica si dos ingredientes i, j son *incompatibles* en un mismo plato. Se tiene que $M[i,j] = 1$ si $i \neq j$ y ambos ingredientes son incompatibles y $M[i,j] = 0$ en caso contrario.

Dado un plato, el nivel de disconformidad del cocinero se define como la cantidad total de pares de ingredientes incompatibles utilizados en dicho plato. Formalmente, el nivel de disconformidad en un plato P que utiliza un conjunto de ingredientes, $P \subseteq \{1, \ldots, n\}$, es igual a $\sum_{i,j \in P, i<j} M[i,j]$.

El problema de decisión *COCINERO*(M, k, d) consiste en determinar si dada la matriz M, es posible preparar un plato con al menos k ingredientes y un nivel de disconformidad de a lo sumo d.

Ejemplo: dada la siguiente matriz M para 4 ingredientes,

0	1	0	0
1	0	1	0
0	1	0	1
0	0	1	0

Se tiene que *COCINERO*$(M, 3, 1)$ es una instancia SÍ, puesto que eligiendo los ingredientes $\{1, 2, 4\}$ logramos un plato con 3 ingredientes y un nivel de disconformidad igual a 1. Por otro lado, *COCINERO*$(M, 3, 0)$ es una instancia NO, ya que cualquier plato con al menos 3 ingredientes siempre tendrá alguno incompatible con otro.

(a) Demuestre que *COCINERO* pertenece a $\mathcal{NP}$. Repita cualquier argumento que utilice de los estudiados en el curso.

(b) Demuestre que *COCINERO* es $\mathcal{NP}$-Completo.

Solución:

(a) La instancia s es una tupla (M, k, d). Como certificado se considera una secuencia t que representa un conjunto de ingredientes. Se debe notar que se cumple $|t| = O(|s|)$.

El certificador $B(s, t)$ calcula la expresión $\sum_{i,j \in t, i<j} M[i,j]$ y da el resultado `Sí` si y solo si esa expresión es menor o igual a d y la cantidad de elementos de t es mayor o igual a k.

- Si s es una instancia `Sí` de *COCINERO* $(M.k, d)$, por definición existe un subconjunto P de k o más ingredientes con nivel de disconformidad menor o igual a d. Entonces si se elige a P como certificado t el resultado de $B(s, t)$ será `Sí`.
- Si el resultado de $B(s, t)$ es `Sí` entonces la secuencia t representa un conjunto de ingredientes con el que se puede puede preparar el plato, por lo que s es una instancia `Sí` de *COCINERO* $(M.k, d)$.

El tiempo de ejecución está determinado por la comparación de cada par de elementos de t, por lo que es $O(|t|^2)$, y por lo tanto es polinomial en el tamaño de la entrada.

(b) En la parte anterior quedó demostrado que el problema está en $\mathcal{NP}$. Por lo tanto alcanza con probar que algún problema $\mathcal{NP}$-Completo se puede reducir a *COCINERO* en tiempo polinomial.

El problema elegido es *INDEPENDENT SET* que se reduce al subconjunto $(M, k, 0)$ de instancias de *COCINERO* de manera inmediata interpretando cada vértice como un ingrediente y cada arista como incompatibilidad entre los ingredientes representados por los vértices que definen la arista.

Otra versión

(a) Demostraremos que *COCINERO* es un problema NP proponiendo un algoritmo certificador $B(s, t)$ eficiente. La entrada s representa una instancia (M, k, d) cualquiera, mientras que la entrada t representa un certificado que consiste en un conjunto de ingredientes. El certificador comprueba que en el conjunto t haya al menos k ingredientes distintos y luego verifica si el nivel de disconformidad, $\sum_{i,j \in t, i<j} M[i,j]$, es a lo sumo d. De cumplirse las condiciones anteriores, $B(s, t)$ responderá afirmativamente.

El tiempo de ejecución de $B(s,t)$ es polinomial en el tamaño de la entrada. Representando al certificado t con una lista encadenada, determinar si la lista tiene al menos k elementos puede implementarse en $\mathcal{O}(1)$, mientras que tiempo de determinar el nivel de disconformidad puede implementarse en $\mathcal{O}(|t|^2)$.

Para demostrar la correctitud de $B(s,t)$, notar que si la instancia s es solución, entonces existe un plato que satisface al cocinero. Tomando como certificado t al conjunto de ingredientes de dicho plato, hará que la invocación a $B(s,t)$ devuelva una respuesta afirmativa. Por otro lado, cuando la invocación a $B(s,t)$ devuelve una respuesta afirmativa, el conjunto de ingredientes t forman entonces un plato que satisface al cocinero, por lo que s es una instancia solución.

(b) Demostraremos que *INDSET* $\leq_p$ *COCINERO*. Puesto que *COCINERO* es $\mathcal{NP}$ y que *INDSET* es $\mathcal{NP}$-Completo, esto implica que *COCINERO* también será $\mathcal{NP}$-Completo.

La reducción propuesta a continuación transforma una instancia (G,k) de *INDSET* en una instancia $(M,k,0)$ de *COCINERO*, interpretando los vértices como ingredientes y las aristas como incompatibilidad entre ingredientes. Notar que la instancia de *COCINERO* implica un plato cuyos ingredientes son todos compatibles entre sí.

La matriz M puede contruirse inicializando todas sus celdas en 0 y luego asignando con 1 aquellas que corresponden a aristas del grafo. El tiempo de ejecución de contruir la matriz es entonces $\mathcal{O}(n^2)$, siendo n la cantidad de vértices del grafo.

Por otra parte, la equivalencia de la reducción es directa. La instancia de *INDSET* es solución si y sólo si existe un subconjunto con al menos k vértices sin aristas entre ellos, lo cual es equivalente a que exista un subconjunto con al menos k ingredientes todos compatibles entre sí.