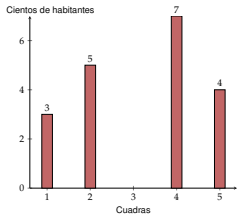
Ejercicios de Programación 3
2018

Parcial-2015

En la administración municipal se consideran las solicitudes para construir edificios en una nueva avenida de largo n cuadras, numeradas de 1 a n. Cada edificio estaría en el centro de una cuadra diferente. El edificio de la cuadra i-ésima alojaría h_i habitantes y si no hay una solicitud para esa cuadra se define $h_i = 0$. Tal vez no se puedan aceptar todas las solicitudes porque por razones de infraestructura (saneamiento, transporte, etc.) la distancia entre dos edificios, medida en cuadras, debe ser por lo menos d, con $1 \leq d \leq n$. Se pretende aceptar solicitudes de tal manera que se maximice la cantidad de habitantes en la avenida.

Ejemplo:
Sea n = 5, $h_1 = 300$, $h_2 = 500$, $h_3 = 0$, $h_4 = 700$, $h_5 = 400$.
Si $d = 3$ la solución óptima es $t = \langle 1, 4 \rangle$ que permite un total de $T = 1000$ habitantes.
Si $d = 1$ una solución óptima es $\langle 1, 2, 4, 5 \rangle$.



1. Se propone el siguiente algoritmo, en el que los edificios se consideran en orden decreciente de la cantidad de habitantes: aceptar el primero y a cada uno de los otros aceptarlo si cumple la restricción de la distancia con los edificios ya aceptados. En el ejemplo anterior, el orden en que se considerarían es $4, 2, 5, 1$.
   a) ¿De qué técnica vista en el curso es un ejemplo este algoritmo? Explique brevemente.
   b) Demuestre que siempre se puede obtener una solución óptima o exhiba un contraejemplo, lo más sencillo posible, en el que no ocurre.
2. De manera independiente a la respuesta del punto anterior, se debe resolver el problema mediante Programación Dinámica asumiendo que se cumple el Principio de Optimalidad. Se define $T(i)$ como el total de habitantes de una solución que optimice el subproblema definido para las cuadras $1, \ldots, i$. Se puede calcular $T(i)$ mediante una recurrencia.
   a) Exprese la recurrencia.
   b) Implemente la recurrencia para el problema $T(n)$ corrigiendo el siguiente algoritmo:

```
1  int total (int * h, int n) {
2      int T[n+1]; // El resultado del subproblema T(i) se mantiene en T[i]
3      T[0] = 0;
4      T[1] = h[1];
5      for (int i = 1; i <= n; i++)
6          if (h[i] > T[i-1])
7              T[i] = h[i] + T[i-d];
8          else
9              T[i] = T[i-1];
10     return T[n];
11 }
```

Solamente puede agregar, suprimir o modificar líneas entre las líneas 3 y 9 incluidas. No se puede usar recursión ni estructuras de datos auxiliares.

Solución:

1. *a*) Es un ejemplo de técnica Greedy. En cada paso se elige el edificio con más habitantes que sea compatible con el estado del problema.
   b) En general este algoritmo no encuentra una solución óptima. Contraejemplo: sea $d = 2$, $n = 3$, $h_1 = 2, h_2 = 3, h_3 = 2$. La solución del algoritmo es $\langle 2 \rangle$, que genera un total 3. Pero con $\langle 1, 3 \rangle$ se puede llegar a 4.

2. *a*) La recurrencia que se pide se puede expresar con

$$T(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max\{h_i + T(\max\{0, i-d\}), T(i-1)\} & \text{si } i \geq 1. \end{cases} \quad (1)$$

b) La implementación iterativa de esta recurrencia, asumiendo que está definida la función $\max$ que devuelve el máximo de sus parámetros, es:

```
int Total (int * h, int n)
{
  int T[n+1];
  T[0] = 0;
  for (int i = 1; i <= n; i++)
    T[i] = max (h[i] + T[max (0, i - d)], T[i-1]);
  return T[n];
}
```

De manera alternativa en la recurrencia se pueden separar los casos en los que $i \leq d$:

$$T(i) = \begin{cases} 0 & \text{si } i = 0 \\ \max\{h_i, T(i-1)\} & \text{si } 1 \leq i \leq d \\ \max\{h_i + T(i-d), T(i-1)\} & \text{si } i > d. \end{cases}$$

Y otra variante puede ser tener como caso base $T(1) = h_1$ en lugar de $T(0) = 0$.

La recurrencia (1) y sus variantes se obtienen tras la aplicación del Principio de Optimalidad.

Sea P_i el subproblema correspondiente a hallar la solución óptima para los edificios en las cuadras $\{1, 2, \ldots, i\}$. La solución, que suponemos óptima, tiene la forma $s = (s_1, s_2, \ldots, s_f)$, con $1 \leq s_1 < s_2 < \ldots < s_f \leq i$, en donde cada s_i representa la cuadra en que está un edificio incluido en la solución. De manera trivial, la solución de P_1 es (1). Para $i > 1$ la última decisión tomada al resolver P_i es si se inclye o no el edificio de la cuadra i-ésima, o sea si $s_f = i$ o $s_f < i$. Llamemos $H_i(t)$ al total de habitantes de una solución t para el problema P_i, esto es $H_i(t) = \sum_{j=1}^{i} h_j$.

- Si se decide no incluir i, esto es si $s_f < i$, la solución de P_i sólo contiene edificios en el conjunto $\{1, \ldots, i-1\}$ por lo que es también solución de P_{i-1} y se cumple $H_{i-1}(s) = H_i(s)$. Y esta solución tiene que ser óptima para P_{i-1} porque cualquier solución r para P_{i-1} no puede superar la cantidad de habitantes de s. Si no fuera así, si $H_{i-1}(r) > H_{i-1}(s)$, se tendría $H_i(r) = H_{i-1}(r) > H_{i-1}(s) = H_i(s) = T(i)$, donde la última igualdad se deriva de la hipótesis de optimalidad de s para el problema P_i. Y esto es una contradicción porque r es también solución para P_i y permitiría alojar una cantidad de habitantes mayor que el óptimo.
- Si la decisión es incluir i entonces en la solución no puede estar ninguno de los edificios anteriores cuya distancia al edificio i-ésimo sea menor que d, lo cual implica que o bien $i \leq d$, lo que implica que en s sólo está i ($f = 1$), o bien $s_{f-1} \leq i - d$.
  - En este último caso la subsolución $(s_1, \ldots, s_{f-1})$ es solución para el problema P_{i-d} y se cumple $H_{i-d}(s) + h_i = H_i(s)$. Y no puede haber una solución para P_{i-d}, $r = (r_1, \ldots r_f)$, mejor que $(s_1, \ldots, s_{f-1})$. Supongamos que r fuera mejor, o sea que $H_{i-d}(r) > H_{i-d}(s)$. Al inluir i en la solución r se obtiene $r' = (r_1, \ldots, r_f, i)$, cuya cantidad de habitantes sería $H_{i-d}(r) + h_i > H_{i-d}(s) + h_i = H_i(s) = T(i)$. De esta forma se llega a la contradicción de que r', que es solución para P_i, permite alojar más habitantes que la solución óptima.
  - En el primer caso el subproblema P_{i-d} no está definido, por lo que de manera trivial no puede haber una mejor solución para P_{i-d} que la subsolución obtenida de excluir i de s.

Por lo tanto se cumple el Principio de Optimalidad.

Como caso base, de manera alternativa a P_1, con $T(1) = h_1$, se puede definir $T(0) = 0$.

Además, para cada cuadra i se define u_i, la última cuadra que cumple la restricción de distancia:

$$u_i = \begin{cases} 0 & \text{si } i \leq d \\ i - d & \text{si } i > d \end{cases}$$

lo que es lo mismo que $u_i = \max\{0, i - d\}$.

Con esto se justifica la recurrencia (1).

Julio-2016

Martín, que es fanático de los programas de televisión, ganó un sorteo en el cual puede llevarse todo lo que quiera de un supermercado hasta un máximo $W = 10.000$ gramos. En el supermercado se sabe que hay n productos distintos, teniendo cada uno un precio d_i y un peso w_i (en gramos) conocidos (ambos números naturales), $i \in \{1, \ldots, n\}$. Para simplificar y sin pérdida de generalidad se asume que hay disponible un solo ítem de cada producto. Martín es astuto y desea revender lo que se lleve del supermercado, entonces quiere llevar la mayor cantidad de dinero en productos posible.

(a) ¿Qué técnica de diseño de algoritmos vista en el curso nos permite resolver de forma óptima y eficiente el problema de determinar cuánto es la ganancia máxima que puede obtener? Especificar matemáticamente el problema y la solución para resolverlo.

(b) Implementar en C* el algoritmo de la parte a. Asuma que se tienen disponibles las funciones *max* y *min*, que devuelven el máximo y mínimo de dos números enteros, respectivamente.

(c) Si se pudiera llevar partes fraccionadas de productos (por ejemplo, cantidad 0,243 de una botella con agua), indicar, justificando brevemente, una manera más eficiente de resolver el problema vista en el curso.

Solución:

(a) La técnica que nos permite resolver el problema es Programación Dinámica.

> Esta realidad se corresponde con el Problema de la Mochila.

Una solución al problema puede representarse mediante las variables $x_i \in \{0, 1\}$, indicando cada una si el ítem i se elige llevar o no. Se desea maximizar el precio de los ítems que se llevan:

$$\max_{x_1,\ldots,x_n} \sum_{i=1}^{n} d_i x_i$$

Sujeto a no sobrepasar el peso máximo:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

Se define $g(k, w)$ como la ganancia de la solución óptima al problema restringido solamente a los ítems 1 a k, teniendo un peso máximo w. Basándose en el Principio de Optimalidad se puede plantear de la siguiente manera:

$$g(k,w) = \begin{cases} g(k-1,w) & \text{si } w < w_k \\ \max\{g(k-1,w), g(k-1,w-w_k) + d_k\} & \text{en otro caso.} \end{cases}$$

Como casos base se tienen:

$$g(0,w) = 0, \forall w \in \{0, \ldots, W\}$$

$$g(k,0) = 0, \forall k \in \{0, \ldots, n\}$$

(b) La implementación del algoritmo es la siguiente:

```
int FKnap(int n, int *w, int *d, int W) {
  int g[n + 1][W + 1];
  for (int j = 0; j <= W; j++)
    g[0][j] = 0;
  for (int k = 1; k <= n; k++)
    for (int j = 0; j <= W; j++)
      if (j < w[k])
```

```
        g[k][j] = g[k-1][j];
      else
        g[k][j] = max(g[k-1][j], g[k-1][j-w[k]] + d[k]);
  return g[n][W];
}
```

(c) En este caso se puede resolver mediante Greedy, ya que se puede eligir en cada paso la mayor cantidad del ítem que da mayor ganancia en relación a su peso, y así obtener la mayor ganancia posible.

Parcial-2016

Sea $A = (A_1, \cdots, A_n)$ una secuencia de n enteros **diferentes**, mayores que 0. Una subsecuencia de A es una secuencia $(A_{h_1}, A_{h_2}, \cdots, A_{h_m})$ de elementos de A, con $1 \leq h_1 < h_2 < \cdots < h_m \leq n$. Una subsecuencia $(A_{h_1}, A_{h_2}, \cdots, A_{h_m})$ es creciente si $A_{h_1} < A_{h_2} < \cdots < A_{h_m}$. Se quiere calcular para cada i, $1 \leq i \leq n$, la longitud, denotada L_i, de la subsecuencia creciente más larga de A que termina en A_i.

Observación: Para cada i puede haber más de una subsecuencia creciente más larga de longitud L_i.

Ejemplo: Dado $A = (5, 2, 4, 7, 1, 6, 3)$, las longitudes de las subsecuencias crecientes más largas son: $1, 1, 2, 3, 1, 3, 2$. La subsecuencia más larga que termina en A_7 puede ser tanto $(2, 3)$ como $(1, 3)$. Las subsecuencias $(5, 6)$ y $(1, 6)$ son subsecuencias crecientes que terminan en A_6, pero $L_6 = 3$ porque también existe la subsecuencia creciente $(2, 4, 6)$, que se obtiene al incluir A_6 al final de la subsecuencia creciente más larga que termina en A_3 (lo cual es válido ya que $A_3 < A_6$).

(a) Llamamos P_i al problema de calcular L_i.
   I. ¿Qué condición se debe cumplir en A para que $L_i = 1$?
   II. Si $L_i \neq 1$, ¿cuáles son los subproblemas que se deben haber resuelto para resolver P_i?

(b) Escriba la recurrencia para calcular las longitudes L_i, para $1 \leq i \leq n$.
Nota: Para la notación se asume que el máximo de un conjunto vacío es 0.

(c) Escriba el seudocódigo que obtiene las longitudes $(L_1, \cdots, L_n)$ implementando la recurrencia de la parte b. La ejecución debe hacerse en tiempo $\Theta(n^2)$ y ocupar $\Theta(n)$ espacio.

Solución:

(a) I. Para que $L_i = 1$ tiene que cumplirse que ninguno de los elementos de A anteriores a la posición i sea menor a A_i: $\nexists\, j \in \{1, \ldots, i-1\}$ tal que $A_j < A_i$.

II. Si $L_i \neq 1$ es necesario resolver los subproblemas P_j para todo j menor que i que cumpla $A_j < A_i$.

(b) Con la siguiente recurrencia se obtienen las longitudes de las subsecuencias crecientes más largas:

$$L_i = 1 + \max_{\substack{1 \leq j < i \\ A_j < A_i}} \{L_j\}, \qquad 1 \leq i \leq n.$$

Se debe notar que si entre los elementos anteriores a i ninguno es menor que A_i (lo que ocurre trivialmente para A_1) se calcula el máximo de un conjunto vacío, que es 0.

(c)
```
Maximas-longitudes-subsecuencias-crecientes(A, n)
  // A = (A_1, ..., A_n)
  FOR i = 1 TO n
      L_i ← 1
      FOR j = 1 TO i - 1
          IF A_j < A_i
              L_i ← max(L_i, 1 + L_j)
  RETURN (L_1, ..., L_n)
```

Debido a los dos ciclos anidados el tiempo de ejecución es $\Theta(n^2)$. Sólo se mantienen las secuencias A y L, por lo que el espacio usado es $\Theta(n)$.

Los siguientes algoritmos, que intentan calcular un L_i genérico en tiempo $\Theta(i)$ no resuelven correctamente el problema. En el ejemplo mostrado en la letra, ambos obtendrían $L_6 = 2$, en lugar de $L_6 = 3$. El algoritmo de la izquierda consideraría que la secuencia más larga que termina en A_6 es $(5,6)$ y el de la derecha consideraría que es $(1,6)$, cuando en realidad es $(2,4,6)$.

```
|$\text{anterior} \gets 0$    // ↩
    menor que todo $A_j$|
|$L_i \gets 1$ // se debe incluir↩
     $A_i$|
FOR j = 1 TO i - 1
    IF |$\text{anterior} < A_j < ↩
        A_i$|
        |$L_i \gets L_i + 1$|
        |$\text{anterior} \gets ↩
            A_j$|
```

```
|$\text{siguiente} \gets A_i$|
|$L_i \gets 1$|
FOR j = i - 1 DOWNTO 1
    IF |$A_j < \text{siguiente}$|
       |$L_i \gets L_i + 1$|
       |$\text{siguiente} \gets ↩
           A_j$|
```

Una alternativa correcta a la solución propuesta utiliza recursión. Sin embargo, a pesar de la correctitud, el tiempo de ejecución es de orden exponencial. La siguiente es una de las posibles variantes:

```
FOR i = 1 TO n
    |$L_i$| ← version_recursiva(A, n, i)

version_recursiva(A, n, i)
    L ← 1
    // en el caso base (i = 1) se devuelve 1
    FOR j = i - 1 DOWNTO 1
        IF |$A_j < A_i$|
            L ← |$\max$|(L, 1 + version_recursiva(A, n, j))
    RETURN L
```

El peor caso se da cuando la secuencia está ordenada de manera creciente. Se puede probar que `version_recursiva(A, n, j)` se invocará 2^{n-j} veces, $1 \leq j \leq n$.

La versión eficiente se resolvió con Programación Dinámica. Para poder aplicar esta técnica se debe cumplir el Principio de Optimalidad. Para cada problema P_i consideremos el problema asociado Q_i que consiste en encontrar la subsecuencia creciente más larga que termina en A_i (o sea, la secuencia cuya longitud es la solución de P_i). Sea $S_i = (A_{h_1}, \cdots, A_{h_k}, A_i)$ la solución óptima de Q_i, y llamemos Pre$_i$ al prefijo $(A_{h_1}, \cdots, A_{h_k})$ de S_i. Consideremos el caso en que Pre$_i$ no es una secuencia vacía (esto es, el caso en que $L_i \neq 0$). Como Pre$_i$ es una subsecuencia creciente podría ser la solución óptima de Q_{h_k}. Supongamos, para llegar a un absurdo, que no lo es. Esto significa que hay otra subsecuencia creciente $(A_{h'_1}, \cdots, A_{h_k})$ cuya longitud L' es mayor que la longitud de Pre$_i$, que es $L_i - 1$. Como $A_{h_k} < A_i$, la subsecuencia $(A_{h'_1}, \cdots, A_{h_k}, A_i)$ es creciente, termina en A_i y tiene longitud $L' + 1$ mayor que L_i, por lo que S_i no puede ser la solución óptima de Q_i. Se llega a un absurdo motivado por suponer que Pre$_i$ no es solución óptima de Q_{h_k}. Por lo tanto se cumple el Principio de Optimalidad.

Febrero-2017

(a) Sea `FibDC` el algoritmo que calcula los números de Fibonacci mediante la técnica *Dividir y Conquistar*. Llamamos $S(n)$ a la cantidad de sumas que se realizan al invocar `FibDC(n)`. Plantee la recurrencia que calcula $S(n)$.

(b) Escriba el algoritmo `FibPD` que calcula los números de Fibonacci mediante *Programación Dinámica*. El algoritmo debe usar $\Theta(1)$ espacio.

(c) Construya una tabla con la cantidad de sumas que realizan `FibDC` y `FibPD` para los valores de n desde 0 hasta 10. Explique en una oración a qué se debe la diferencia del crecimiento de la versión `FibDC` con respecto al de la versión `FibPD`.

Solución:

(a)
```
FibDC(n)
    if (n = 0) OR (n = 1)
        Fn = 1
    else
        Fn = FibDC(n-1) + FibDC(n-2)
    return Fn
```

$$S(n) = \begin{cases} 0 & \text{si } n \in \{0,1\} \\ 1 + S(n-1) + S(n-2) & \text{en otro caso.} \end{cases}$$

(b)
```
FibPD(n)
    F0 = 0
    F1 = 1
    Fn = 1
    for i = 2 to n
        F0 = F1
        F1 = Fn
        Fn = F0 + F1
    return Fn
```

(c)

n	0	1	2	3	4	5	6	7	8	9	10
FibDC	0	0	1	2	4	7	12	20	33	54	88
FibPD	0	0	1	2	3	4	5	6	7	8	9

El crecimiento de la versión `FibDC` es mayor debido a que la superposición de subproblemas hace que cada subproblema se resuelva varias veces.

La cantidad de sumas de la versión `FibPD(n)` es $n-1$ para $n > 0$. La de la versión `FibDC(n)` es $S(n)$ que se puede demostrar que es igual a $Fibonacci(n) - 1$.

Julio-2017

Se considera una grilla como la mostrada en la Figura 1, en la que cada línea horizontal o vertical de la grilla representa una calle por la cual se puede transitar. Aparecen destacados dos edificios E_1 y E_2 en los extremos de la grilla.

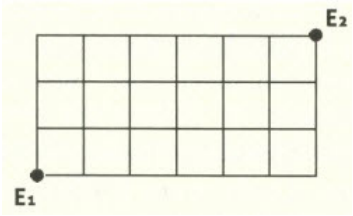


Figura 1: Grilla de calles y edificios destacados

Se asignan coordenadas a los puntos de la grilla que son intersección de dos calles, de modo que el edificio E_1 se considera ubicado en el origen con coordenadas $(0,0)$ y el edificio E_2 está situado en las coordenadas (m,n). Esto corresponde a una grilla con $m+1$ calles verticales y $n+1$ calles horizontales. Para el ejemplo de la Figura 1 se tiene $m=6$ y $n=3$.

Se definen los caminos que llevan desde un punto genérico de la grilla de coordenadas (i,j) hasta el edificio E_2, como los recorridos sobre la grilla desde (i,j) hasta (m,n) con la restricción de que al pasar por una intersección de calles sólo se puede continuar hacia la derecha ó hacia arriba, sin salirse de la grilla (lo que hace que dichos caminos sean de largo mínimo). A modo de ejemplo, en la Figura 2 se muestran tres de los caminos posibles entre E_1 y E_2.

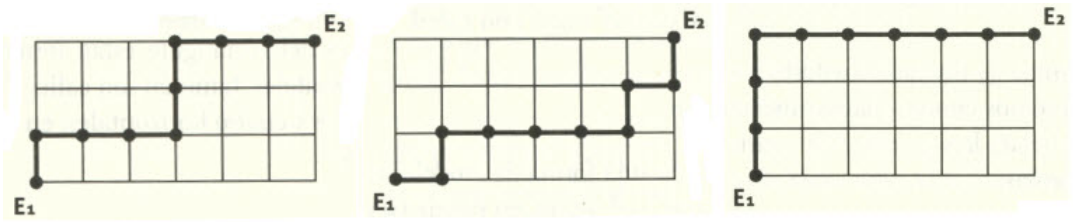


Figura 2: Ejemplos de caminos entre E_1 y E_2

Se define $NC(i,j)$ como el número total de caminos que hay entre el punto de la grilla de coordenadas (i,j) y el edificio E_2. Se toma la convención de que $NC(m,n)=1$.

(a) Especificar completamente la relación de recurrencia que verifica $NC(i,j)$, para $0 \leq i \leq m, 0 \leq j \leq n$.

(b) Implementar en C^* utilizando la técnica de *Programación Dinámica* la función

```
int NumeroDeCaminos(int m,int n)
```

que calcula y devuelve la cantidad de caminos que llevan desde el edificio E_1 al edificio E_2, para una grilla cuyos valores de m y n se pasan como parámetro.

(c) Para el caso particular $m=6$ y $n=3$ correspondiente a la Figura 1, calcular en base al algoritmo implementado la cantidad de caminos entre E_1 y E_2, justificando el resultado obtenido.

Solución:

(a) Para todos los puntos de la grilla situados en los bordes de la misma con coordenada horizontal m ó coordenada vertical n, hay un único camino hacia el edificio E_2.
Para los restantes puntos de la grilla, la cantidad de caminos es la suma de la cantidad de caminos del punto vecino situado inmediatamente a su derecha en la grilla y de la cantidad de caminos del punto vecino situado inmediatamente arriba en la grilla.

La recurrencia puede plantearse entonces como:

$$NC(i,j) = \begin{cases} 1 & \text{si } i = m \text{ ó } j = n \\ NC(i+1,j) + NC(i,j+1) & \text{si } 0 \leq i < m, 0 \leq j < n \end{cases}$$

(b) La idea es usar una tabla NC para almacenar el número de caminos desde cada punto de la grilla hasta E_2. Primero se ponen en 1 los elementos de la tabla correspondientes a los bordes de la grilla con coordenada horizontal m y coordenada vertical n, y luego la tabla se completa recorriendo la grilla de derecha a izquierda y de arriba hacia abajo (podría ser también en el orden inverso), aplicando la recurrencia hallada. El resultado pedido es el último valor de la tabla calculado y corresponde a $NC[0][0]$.

```
int NumeroDeCaminos (int m, int n) {
   int NC[m+1][n+1];
   for(int j =0; j<=n; j++)   // borde derecho de la grilla
        NC[m][j] = 1;
   for(int i =0; i<=m; i++)   // borde superior de la grilla
        NC[i][n] = 1;
   for(i = m-1; i>=0; i--)      // de derecha a izquierda
    for(j = n-1; j>=0; j--)  // de arriba hacia abajo
             NC[i][j] = NC[i+1][j] + NC[i][j+1];
   return NC[0][0];
}
```

(c) En la Figura 3 se muestran para cada punto de la grilla los valores de NC obtenidos mediante el algoritmo implementado. La cantidad de caminos desde E_1 a E_2 es $\mathbf{NC[0][0] = 84}$.

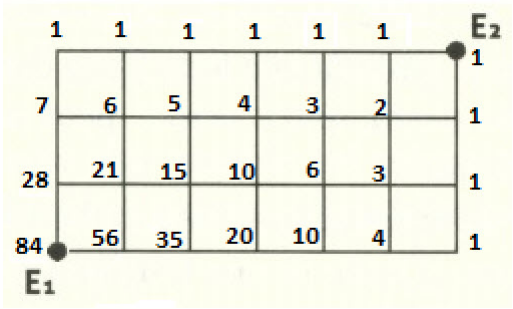


Figura 3: Grilla solución con cantidad de caminos hasta E_2

Este valor puede corroborarse observando que $NC[0][0] = \binom{m+n}{m} = \binom{m+n}{n} = \binom{9}{3} = 84$.

Parcial-2017

Dada una secuencia $A = a_1, \ldots, a_n$ con n caracteres.

Considere las siguientes definiciones:

Subsecuencia: A' es una subsecuencia de A, si es una secuencia formada por caracteres pertenecientes a A, y además se cumple que el orden en que aparecen en A se respeta en A'. Por ejemplo: Si A=f**ot**og**ra**fo, entonces **otra** es una subsecuencia de A, y **rato** no lo es.

Substring: A' es un substring de A, si es una subsecuencia de A que además cumple que los caracteres en A' ocurren de forma consecutiva en A. Por ejemplo, si A=foto**grafo**, **grafo** es un substring de A, y otra no lo es.

Considere el siguiente pseudocódigo,

```
int f (char* A, char* B) {
    n = length(A)
    m = length(B)
    int Mt [n+1][m+1];
    int Mp [n+1][m+1];

    for(int i=0; i<=m; i++){
        Mt[0][i] = 0;
        Mp[0][i] = 0;
    }

    for(int i=0; i<=n; i++){
        Mt[i][0] = 0;
        Mp[i][0] = 0;
    }

    for(int i=1; i<=n; i++){
        for(int j=1; j<=m; j++){
            if (A[i] == B[j]){
                Mp[i][j] = 1+ Mp[i-1][j-1];
            }
            else{
                Mp[i][j] = 0;
            }
            Mt[i][j] = max(Mp[i][j], Mt[i-1][j],Mt[i][j-1]);
        }
    }

    return Mt[n][m];
}
```

Listado 1: Algoritmo 1

(a) Realice el proceso de ejecución para la entrada $A = la$ $B = ala$, escriba en su respuesta solamente el estado final de las matrices Mt y Mp.

(b) Indique qué se almacena en cada estructura auxiliar Mt y Mp.

(c) ¿Qué problema resuelve?

(d) Plantee la recurrencia a partir de la cual se construyó el pseudocódigo.

(e) I. Escriba el pseudocódigo una solución recursiva del problema. Considere la siguiente firma para el pseudocódigo: `int f (int n, int m, int** Mp)`

II. Indique qué invocaciones serían necesarias para resolver el problema Mt(2,3). Sugerencia: utilice el árbol de recursión.

III. ¿Qué inconveniente le encuentra a esta solución?

(f) Determine la complejidad en espacio de memoria del Algoritmo 1. ¿Hay alguna forma de reducirla? Justfique.

Solución:

(a) $A = la\ B = ala$

Mp :

i \ j		a	l	a
	0	0	0	0
l	0	0	1	0
a	0	1	0	2

Mt :

i \ j		a	l	a
	0	0	0	0
l	0	0	1	1
a	0	1	1	2

(b) Los sufijos de una cadena $a_1, \ldots, a_i$ son los substrings $a_h, \ldots, a_i$, con $1 \leq h \leq i$.
$Mt[i][j]$ representa el largo del substring común más largo entre $a_1, \ldots, a_i$ y $b_1, \ldots, b_j$.
$Mp[i][j]$ representa el largo del sufijo común más largo entre $a_1, \ldots, a_i$ y $b_1, \ldots, b_j$.

(c) Resuelve el problema de encontrar el largo del substring común más largo entre A y B.

(d) Se considera un máximo total, representado por Mt, que es donde se almacena la solución al problema y un máximo parcial, representado por Mp, que almacena el substring de largo máximo utilizando los caracteres en la posición actual.

De esta forma, el problema se formaliza como sigue:

$$Mt(i,j) = \begin{cases} 0, & \text{si } i = 0 \text{ o } j = 0 \\ max\{Mp(i,j), Mt(i-1,j), Mt(i,j-1)\}, & \text{en caso contrario} \end{cases}$$

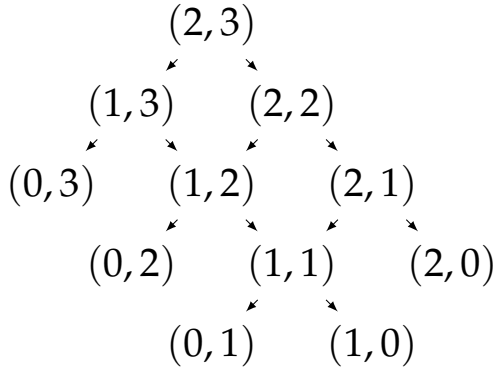
$$Mp(i,j) = \begin{cases} 0, & \text{si } i = 0 \text{ o } j = 0 \\ \mathbb{1}_{a_i,b_j} \times (Mp(i-1,j-1)+1), & \text{en caso contrario} \end{cases}$$

$$\mathbb{1}_{a_i,b_j} = \begin{cases} 0, & \text{si } a_i \neq b_j \\ 1, & \text{en caso contrario} \end{cases}$$

(e) I.

```
int f (int n, int m, int** Mp) {
    if (n == 0 || m ==0){
        return 0    ;
    }
    else{
        return max{f(n-1,m, Mp), f(n,m-1,Mp), Mp[n][m]};
    }
}
```

II.



III. El problema que tiene lo anterior es que se invoca muchas veces al mismo subproblema, calculando más de una vez lo mismo. Esta superposición de problemas es lo que evita la técnica Programación Dinámica.

(f) La complejidad en espacio del algoritmo es $O(n \times m)$ porque se utilizan dos matrices de tamaño $(n + 1) \times (m + 1)$.

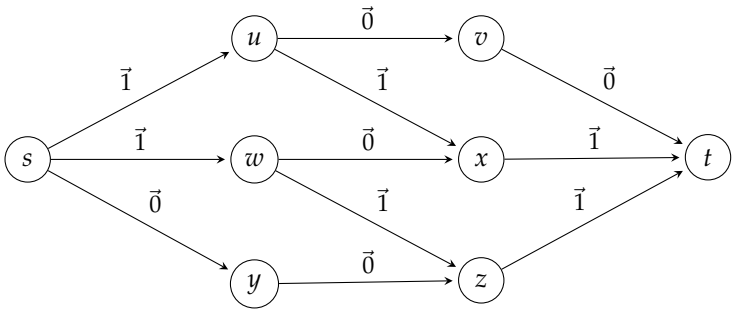
Como para resolver el problema $Mt(i, j)$ solo hay que mirar los subproblemas $Mt(i - 1, j)$, $Mt(i, j - 1)$ alcanza con mantener, además de la fila actual, la fila anterior con un arreglo de tamaño $m + 1$ para Mt. Como además $Mt(i, j)$ utiliza $Mp(i, j)$ y para mantener esta información solo se necesita el valor $Mp(i - 1, j - 1)$ alcanzaría con tener un arreglo de tamaño $m + 1$. Por lo tanto la complejidad en espacio del algoritmo sería $O(m)$.

De manera análoga se puede resolver el problema manteniendo dos columnas para cada matriz. Con esta versión la complejidad en espacio del algoritmo sería $O(n)$.

Parcial-2017

Sea $G(V, E)$ una red de flujo s-t con capacidades enteras en sus aristas.

(a) Sea la instancia de la red con capacidad constante de valor uno y flujo según aristas



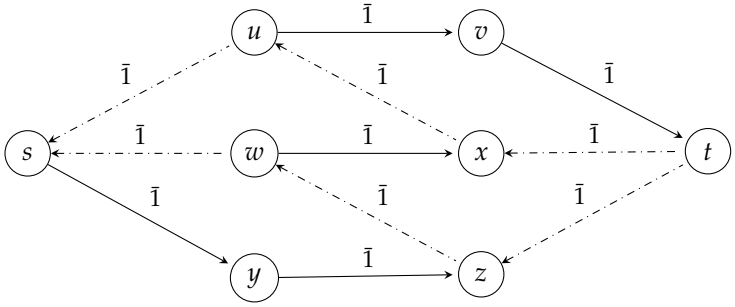
determinar su flujo máximo aplicando el algoritmo de Ford-Fulkerson a partir del flujo dado. [Mostrar los pasos del algoritmo mediante secuencias de grafos residuales y grafos con flujo.]

(b) Suponer que la red, con capacidad constante de valor uno, se construye a partir de un grafo bipartito al que se le agregan un nodo fuente, un nodo destino, aristas entre el nodo fuente y los nodos de una partición y aristas entre los nodos de la otra partición y el nodo destino. Formalmente, $G(V, E)$ se construye a partir de un grafo bipartito $G'(U \cup W, E' \subseteq U \times W)$ tal que $m = |U|$, $n = |W|$, y donde $V = U \cup W \cup \{s, t\}$ y $E = \{(u, w) | \{u, w\} \in E'\} \cup \{(s, u) | u \in U\} \cup \{(w, t) | w \in W\}$.

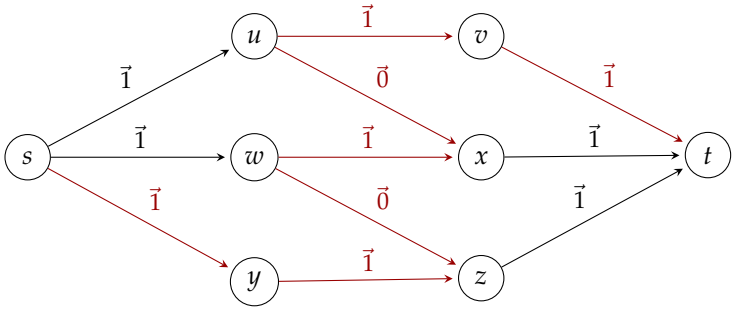
Determinar el largo máximo (cota superior) de los caminos de incremento (augmenting path) obtenidos al aplicar el algoritmo de Ford-Fulkerson en todo $E' \subseteq U \times W$ posible. [Establecer la cota en función de m y n.]

Solución:

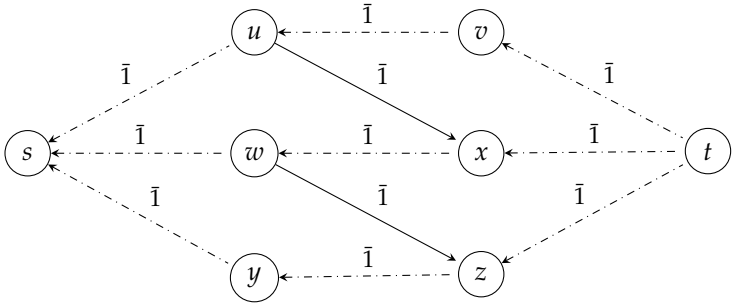
(a) Sea el grafo residual del flujo dado (con las aristas de retorno destacadas en punteado)



En el se determina el camino de incremento (s, y, z, w, x, u, v, t) con flujo de valor uno. Luego de aplicarlo en la red se tiene el nuevo flujo (con cambios respecto al anterior destacados en rojo)



Para este nuevo flujo se tiene el grafo residual



en el cual no hay camino de incremento. Por lo tanto el flujo es máximo y de valor tres.

(b) El camino de incremento de s a t más largo se obtendría al recorrer la máxima cantidad de nodos del grafo bipartito. El camino estaría conformado por el nodo s, al menos un nodo de U, al menos un nodo de W y el nodo t. Para que el camino incluya más de dos nodos del grafo bipartito tendría que haber aristas de retroceso en el grafo residual, tales que luego de acceder a un nodo de W se pueda "retornar" a un nodo de U. En el apartado anterior se tiene una instancia de la red con un flujo que determina un camino, con aristas de retorno, que recorre todos los nodos.

Si $m = n$ podría haber un camino que recorriera todos los nodos de G. Al recorrer todos los nodos, cantidad $2m + 2$, tiene largo $2m + 1$ (aristas).

Si $m \neq n$ el camino no puede recorrer todos los nodos de G', dado que no puede repetir nodos, por lo que a lo sumo puede recorrer $2\,\text{mín}\{m, n\}$ nodos. Entonces el camino más largo contiene $2\,\text{mín}\{m, n\} + 2$ nodos y su largo es $2\,\text{mín}\{m, n\} + 1$ (aristas).

Parcial-2017

Para cada una de las siguientes preguntas, indique si la respuesta es “Sí”. “No”, o “No se sabe. Si se supiera resolvería el problema de decidir si $\mathcal{P} = \mathcal{NP}$”. Dé una explicación breve (una o dos oraciones) de su respuesta. Sugerencia: utilice su conocimiento de a que clase pertenece cada uno de los problemas mencionados en las preguntas.

Definamos la siguiente versión del problema de la *Programación de Intervalos* (Interval Scheduling) como en formato de problema de decisión: dada una colección de intervalos en una linea de tiempo, y una cota k, ¿la colección contiene un subconjunto de intervalos no superpuestos de tamaño al menos k?

(a) ¿Es verdad que *Programación de Intervalos* $\leq_P$ *Cubrimiento de Vértices* (Vertex Cover)?

(b) ¿Es verdad que *Conjunto Independiente* (Independent Set) $\leq_P$ *Programación de Intervalos*?

Solución:

(a) Si. Una solución puede ser: Interval Schedulling puede ser resuelto en tiempo polinómico, y por lo tanto tambien puede ser resuelto en tiempo polinómico con acceso a una caja negra que resuelva el problema Vertex Cover (llamándola 0 veces). Otra solución posible es: Interval Scheduling es NP y cualquier problema NP puede ser reducido al Vertex Cover.

(b) Esto es equivalente a decir que P = NP. Si P = NP, entonces Independent Set puede ser resuelto en tiempo polinómico y ser reducido a Interval Scheduling. Lo contrario también sería verdad ya que Independent Set en NP-Completo.

Diciembre-2017

En una empresa existe una hilera de n baldosas para hacer transitable un camino entre dos de sus edificios. Debido a restricciones presupuestales se desea vender algunas de esas baldosas que según su estado tienen valores de venta diferentes: el valor de la i-ésima baldosa es $b[i]$. Para que el camino continúe siendo transitable no se pueden remover dos baldosas seguidas del camino.

(a) Desarrolle la relación de recurrencia que obtiene el máximo valor de venta que se puede lograr.

(b) Implemente la función

```
int * removidas (int *b, int n, int & c);
```

que devuelve un arreglo con las baldosas que deben removerse para obtener la máxima ganancia. El arreglo devuelto debe estar en orden decreciente. En el parámetro c se debe devolver la cantidad de baldosas que se remueven. Los valores de las n baldosas se pasan en las posiciones 1 a n del arreglo b.

Solución:

(a)

$$C(i) = \begin{cases} 0 & \text{si } i = 0, \\ b_1 & \text{si } i = 1, \\ \max\{C_{i-1}, b_i + C_{i-2}\} & \text{si } i > 1, \end{cases}$$

(b)

```
int * removidas (int *b, int n, int &c) {
  int C[n+1];   // costo
  bool Tb[n+1]; // traceback
  C[0] = 0, C[1] = b[1];
  Tb[1] = true;
  for (int i = 2; i <= n; i++)
    if (b[i] + C[i-2] > C[i-1]) {
      C[i] = b[i] + C[i-2];
      Tb[i] = true;
    } else {
      C[i] = C[i-1];
      Tb[i] = false;
    }
  int *R = new int [n / 2]; // es el máximo posible
  c = 0;
  int i = n;
  while (i > 0) {
    if (Tb[i] == true) {
      c++;
      R[c] = i;
      i -= 2;
    } else {
      i -= 1;
    }
  }
  return R;
}
```

Esta es una versión simplificada del problema de intervalos ponderados. En este caso cada baldosa representa un intervalo y se solapa con la inmediata anterior y la inmediata posterior y sólo con ellas.

Febrero-2018

Sea una red de flujo dada por un grafo dirigido $G = (V, E)$ con nodo fuente $s \in V$, nodo destino $t \in V$ y capacidad $c_e \geq 0$ en cada arista $e \in E$.

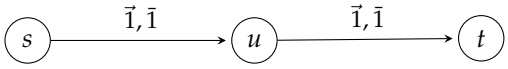
Indique si las siguientes afirmaciones son verdaderas o falsas. Justifique.

(a) Dado un flujo máximo, existe un único corte de capacidad mínima.

(b) Un corte de capacidad mínima define un único flujo máximo.

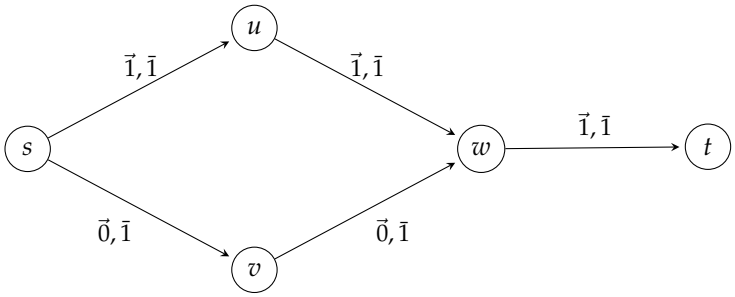
Solución:

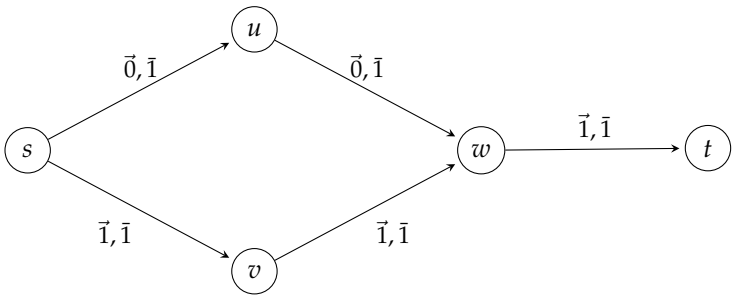
(a) Falsa. Contraejemplo. Dada la red:



Los cortes $(\{s\}, \{u, t\})$ y $(\{s, u\}, \{t\})$ son ambos de capacidad mínima.

(b) Falsa. Contraejemplo.





En ambas redes el corte de capacidad mínima es $(\{s, u, v, w\}, \{t\})$ pero los flujos son distintos.

Febrero-2018

Sea A una secuencia de elementos para los que está definido un orden, $A = (a_1, a_2, \ldots, a_n)$. Las subsecuencias crecientes de A son las secuencias $(a_{h_1}, a_{h_2}, \ldots, a_{h_p})$ que cumplen $1 \leq h_1 < h_2 < \cdots < h_p \leq n$ y $a_{h_i} < a_{h_{i+1}}$. O sea, son secuencias crecientes de elementos de A que aparecen en el mismo orden en que aparecen en A. Se quiere encontrar la subsecuencia creciente más larga de A (`LIS`, por su nombre en inglés *Longest Increasing Subsequence*). Para cada prefijo $A_i = (a_1, a_2, \ldots, a_i)$, $1 \leq i \leq n$, de A se define G_i como el largo de la subsecuencia creciente más larga de A_i y L_i como el largo de la subsecuencia creciente más larga de A_i que termina en a_i.

Ejemplo: Dado $A = (5, 2, 4, 7, 1, 6, 3)$, los valores de G_i son $1, 1, 2, 3, 3, 3, 3$ y los de L_i son $1, 1, 2, 3, 1, 3, 2$. La subsecuencia más larga que termina en a_7 puede ser tanto $(2, 3)$ como $(1, 3)$. Las subsecuencias $(5, 6)$ y $(1, 6)$ son subsecuencias crecientes que terminan en a_6, pero $L_6 = 3$ porque también existe la subsecuencia creciente $(2, 4, 6)$, que se obtiene al incluir a_6 al final de la subsecuencia creciente más larga que termina en a_3 (lo cual es válido ya que $a_3 < a_6$).

Se puede ver que $\{G_i\}$ cumple la relación de recurrencia $G_0 = 0$, $G_i = \max\{G_{i-1}, L_i\}$ para $1 \leq i \leq n$.

(a) Complete las relaciones de recurrencia, incluyendo L_i.

Nota: Para la notación se asume que el máximo de un conjunto vacío es 0.

(b) Complete el código que devuelve el largo de la subsecuencia creciente más larga de A y en `resultado` los elementos que forman esa subsecuencia. El valor de `resultado[i]` debe ser `true` si y sólo si a_i es parte de la subsecuencia buscada.

```
int lis(int n, int * A, bool * &resultado) {
  int L[n + 1];
  int G[n + 1];
  // completar 1
  G[0] = 0;
  for (int i = 1; i <= n; i++) {
    L[i] = 1;
    // completar 2
    for (int j = 1; j < i; j++) {
      if (A[j] < A[i]) {
        if (L[j] >= L[i]) {
          // completar 3

        }
      }
    }
    G[i] = max(G[i - 1], L[i]);
  }
  int largo = 0;
  resultado = new bool[n + 1];
  for (int i = 0; i <= n; i++)
    resultado[i] = false;
  int cursor = n;
  // completar 4









  return largo;
}
```

(c) Dada una secuencia de bandejas rectangulares, $B = (b_i, \ldots, b_n)$ un mozo se propone construir la pila de mayor cantidad de bandejas posibles. Sólo puede apoyar una bandeja sobre otra si sus dos dimensiones, ancho y largo, son menores que las de la bandeja en la que se apoya. Se puede suponer que los anchos de todas las bandejas son diferentes y que los largos de todas las bandejas son diferentes, o sea, $b_i.w \neq b_j.w$ y $b_i.\ell \neq b_j.\ell$ cuando $i \neq j$, donde $b_h.w$ y $b_h.\ell$ son respectivamente el ancho y el largo de la bandeja b_h. Resuelva el problema de calcular el tamaño de la pila solución. Para ello, transforme este problema en el de la subsecuencia creciente más larga.

Solución:

(a)

$$L_i = 1 + \max_{\substack{1 \le j < i \\ a_j < a_i}} \{L_j\}, \qquad 1 \le i \le n.$$

$$G_0 = 0,$$
$$G_i = \max\{G_{i-1}, L_i\} \qquad 1 \le i \le n.$$

(b)

```
int lis(int n, int * A, bool * &resultado) {
  int L[n + 1];
  int G[n + 1];
  int T[n + 1]; // traceback
  G[0] = 0;
  for (int i = 1; i <= n; i++) {
    L[i] = 1;
    T[i] = 0;
    for (int j = 1; j < i; j++) {
      if (A[j] < A[i]) {
        if (L[j] >= L[i]) {
          L[i] = 1 + L[j];
          T[i] = j;
        }
      }
    }
    G[i] = max(G[i - 1], L[i]);
  }
  int largo = 0;
  resultado = new bool[n + 1];
  for (int i = 0; i <= n; i++)
    resultado[i] = false;
  int cursor = n;
  while (cursor > 0) {
    if (G[cursor] != L[cursor])
      cursor--;
    else {
      largo++;
      resultado[cursor] = true;
      cursor = T[cursor];
    }
  }
  return largo;
}
```

(c) Cualquier pila factible recorrida desde el tope hasta el fondo es una secuencia de bandejas en las que tanto los anchos como los largos están ordenados de manera creciente.

Se ordena B según una de las dimensiones, por ejemplo el ancho. Se obtiene una permutación de B, $B' = (b'_1, \ldots, b'_n)$ con $b'_i.w < b'_{i+1}.w$, $1 \le i < n$. Para que una pila sea factible es condición necesaria que sea una subsecuencia de B'.

Para que la pila también esté ordenada según el largo se considera la secuencia $B'.\ell = (b'_1.\ell, \ldots, b'_n.\ell)$. Las pilas factibles se corresponden con las subsecuencias crecientes de esta secuencia. Por lo tanto el largo de la pila solución es el largo de la subsecuencia creciente más larga de $B'.\ell$.

Julio-2018

Suponga que X, Y, Z son problemas que cumplen $X \in \mathcal{P}$, $Y \in \mathcal{NP}$ y Z es $\mathcal{NP}$-*completo*. Para cada una de las siguientes sentencias responda si es siempre verdadera o siempre falsa o no se puede afirmar ninguna de las dos anteriores. Si se da el tercer caso (no se pueda afirmar nada) ¿qué se debe cumplir para que sea verdadera? Justifique cada respuesta.

1. $Y \leq_P Z$
2. $X \leq_P Z$
3. $Z \leq_P Y$
4. $Z \leq_P X$

Solución:

1. $Y \leq_P Z$

   Siempre verdadera. Esto se concluye de la definición de la clase de problemas $\mathcal{NP}$-*completo*.

2. $X \leq_P Z$

   Siempre verdadera. Esto es consecuencia del punto anterior y de que $\mathcal{P} \subseteq \mathcal{NP}$.

3. $Z \leq_P Y$

   No se puede en general afirmar nada. Si fuera cierto se podria deducir que Y es $\mathcal{NP}$-*completo* porque pertenece a $\mathcal{NP}$ y es al menos tan difícil como cualquier problema *NP-completo*.

4. $Z \leq_P X$

   No se puede en general afirmar nada. Si fuera cierto para algún par de problemas X, Z entonces se cumpliría que $\mathcal{P} = \mathcal{NP}$ porque cualquier problema *NP-completo* (y como consecuencia todo problema $\mathcal{NP}$) podría resolverse en tiempo polinómico.

Julio-2018

Se considera una ciudad cuyo mapa de calles tiene forma de matriz de N x N y donde cada esquina se puede identificar como una celda (i, j) de la misma. Un recolector debe determinar el recorrido a realizar con su vehículo desde la esquina $(1, 1)$ hasta la esquina (N, N) de forma de recolectar la mayor cantidad de objetos posibles en las esquinas de la ciudad, sujeto a la restricción de que todas las calles están flechadas en sentido creciente, tanto en la dirección horizontal como en la vertical.

Para ello se cuenta con una matriz E, donde cada $E(i, j)$ ($1 \leq i, j \leq N$) es un entero que indica la cantidad de objetos que se pueden recolectar en la esquina (i, j) (si no hay objetos $E(i, j) = 0$).

(a) Escriba la relación de recurrencia con la que se obtiene C, la cantidad máxima de objetos que se pueden recolectar.

(b) Complete el algoritmo que está a la derecha que calcula la cantidad de objetos que se pueden recolectar y el recorrido.
Parámetros:

- E: Parámetro de entrada. Arreglo bidimensioneal con la cantidad de objetos disponibles en cada esquina. Note que no se tienen en cuenta `E[0][j]` ni `E[i][0]`, para $0 \leq i, j \leq N$.
- rec: Parámetro de salida. Arreglo de tamaño $2N$. Cada `rec[i]` ($1 \leq i \leq 2N - 1$) indica la i-esima esquina por la que se debe pasar. Note que no se tiene en cuenta `rec[0]`.

Return: `recoleccion` retorna la cantidad total de objetos recolectados.

```
/* cantidad de calles horizontales
    y verticales de la ciudad       */
#define N 100

struct Pos {
   int fila;
   int col;
};

int recoleccion(int E[N+1][N+1], Pos rec[2*N]) {
  int C[N+1][N+1];
  for (int i = 1; i <= N; i++ )
    C[i][0] = C[0][i] = 0;
  for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
      // completar 1
    }
  }
  int i = 2*N - 1;
  Pos esq;
  esq.fila = esq.col = N;
  while ((esq.fila > 0) && (esq.col > 0)) {
    // completar 2
  }
  // completar 3
}
```

(c) Para esta parte se modifica el significado de los valores de E. Debido a tareas de mantenimiento, algunas esquinas no son transitables por lo que el recorrido no puede pasar por ellas. Esto se indica con $E(i, j) = -1$. En las esquinas transitables el valor de E conserva el significado de la parte (a). Escriba la relación de recurrencia para calcular C teniendo en cuenta esta nueva restricción. C debe ser 0 si no se puede llegar a la esquina (N, N). Puede definir una relación de recurrencia auxiliar.

Solución:

(a)

$$C(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0, \\ \max(C(i-1,j), C(i,j-1)) + E(i,j) & \text{en otro caso.} \end{cases}$$

(b)

```
int recoleccion(int N, int ** E, int & tope, Pos * & res) {
  int C[N+1][N+1];
  for (int i = 1; i <= N; i++ )
    C[i][0] = C[0][i] = 0;
  for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
      C[i][j] = max(C[i-1][j], C[i][j-1]) + E[i][j];
    }
  }
  tope = 0;
```

```
  Pos esq;
  esq.fila = esq.col = N;
  while (esq.fila >=0 && esq.col >= 0) {
    if (E[esq.fila][esq.col] > 0) {
      res[tope] = esq;
      tope++;
    }
    if (C[esq.fila-1][esq.col] >= C[esq.fila][esq.col-1])
      esq.fila--;
    else
      esq.col--;
  }
  return C[N][N];
}
```

(c) Con $A(i,j)$ se indica si la esquina (i,j) es accesible o no. La esquina (i,j) no es accesible si no es transitable o si ninguna de las esquinas $(i-1,j)$ y $(i,j-1)$ son accesibles. La relación de recurrencia correspondiente es:

$$A(i,j) = \begin{cases} \texttt{false} & \text{si } i = 0 \text{ o } j = 0, \\ \quad \text{o, si } 0 < i \leq M \text{ y } 0 < j \leq N, & \\ \texttt{false} & \text{si } E(i,j) = -1, \text{o} \\ & A(i-1,j) = \texttt{false} \text{ y } A(i,j-1) = \texttt{false}, \\ \texttt{true} & \text{en otro caso.} \end{cases}$$

La cantidad C se calcula teniendo en cuenta si la esquina es accesible.

$$C(i,j) = \begin{cases} 0 & \text{si } A(i,j) = \texttt{false}, \\ \text{máx}(C(i-1,j), C(i,j-1)) + E(i,j) & \text{en otro caso.} \end{cases}$$