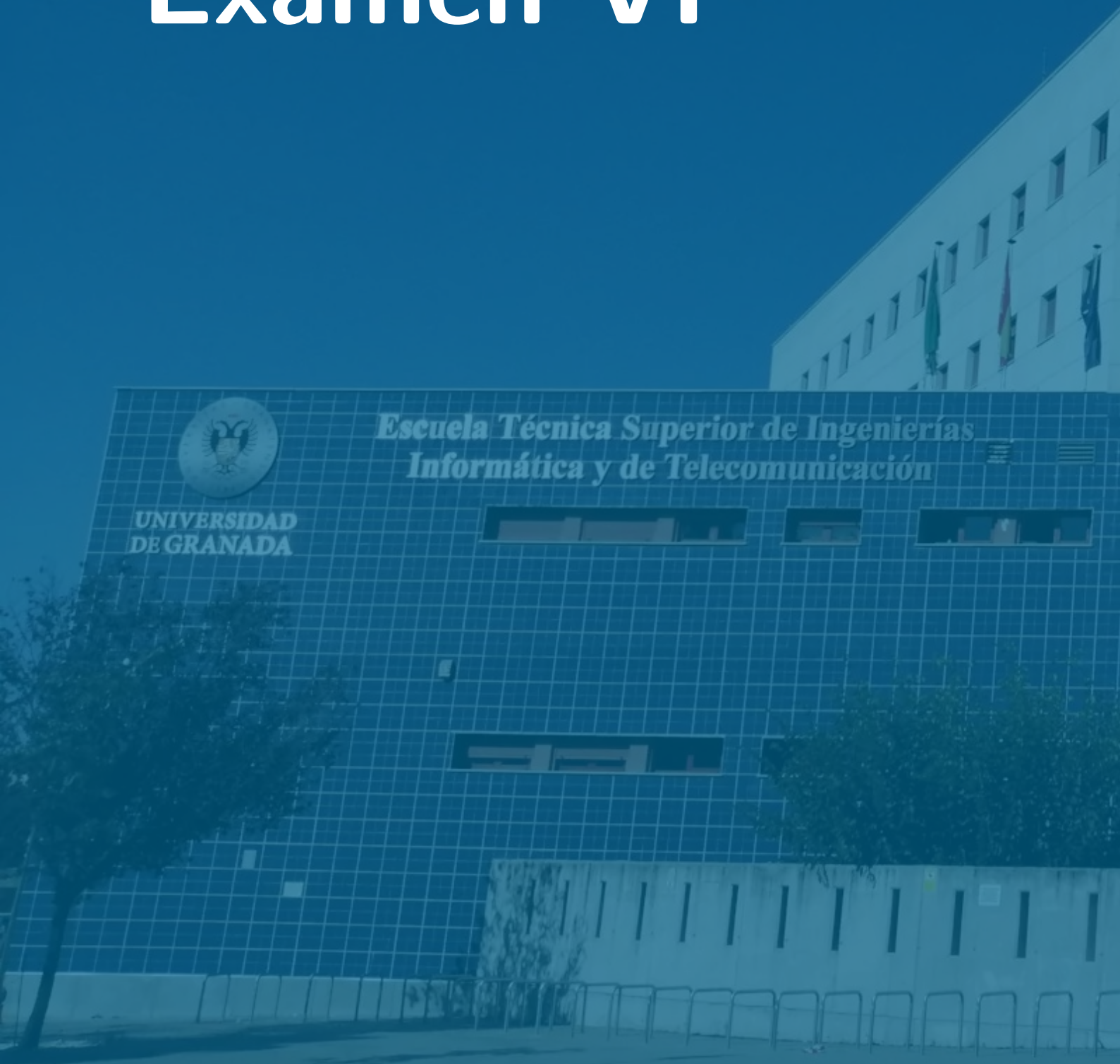
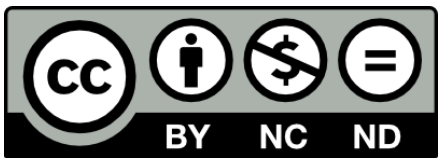
Algorítmica
Examen VI



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Algorítmica
Examen VI

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán

Granada, 2025

Asignatura Algorítmica.

Curso Académico 2024/25.

Grado Doble grado en Ingeniería Informática y Matemáticas.

Grupo Único.

Profesor Francisco Javier Cabrerizo Lorite.

Descripción Examen de la Convocatoria Ordinaria.

Fecha 16 de junio de 2025.

Duración 2 horas y media.

Ejercicio 1 (2 puntos)**.** La ecuación en recurrencias resultante del análisis de eficiencia de un algoritmo depende de una condición dada C, de modo que tenemos $T(n) = 2T(n/4) + n^2$ si C es cierta, y $T(n) = 4T(n/2) + \log_2(n)$ si C es falsa. Determine los órdenes de complejidad O y Ω del algoritmo y justifique si este tiene, o no, orden de complejidad Θ.

Ejercicio 2 (2 puntos)**.** Dado un vector de números enteros de tamaño n, ordenado de menor a mayor, y donde puede haber números duplicados, se quiere contar cuántas veces aparece un determinado número x o indicar que dicho número no aparece ninguna vez. Diseñe un algoritmo basado en la técnica "divide y vencerás" que resuelva el problema de la forma más eficiente posible. ¿Cuál es el orden de complejidad O del algoritmo diseñado?

Ejercicio 3 (2 puntos)**.** Una empresa de transportes hace de forma regular una ruta con su flota de vehículos. A lo largo de esta ruta, se ha identificado un conjunto de n puntos kilométricos específicos $(k_1, k_2, \ldots, k_n)$ en los que sus vehículos deben detenerse. Con el fin de facilitar las tareas de mantenimiento y control, se desea instalar un conjunto de estaciones a lo largo de la ruta. Cada estación puede cubrir los vehículos que se encuentran en un rango de L kilómetros. Se desea identificar dónde ubicar el *menor* conjunto de estaciones, garantizando que todos los puntos de parada prefijados queden cubiertos por al menos una estación. Diseñe un algoritmo voraz que resuelva este problema de forma óptima.

Ejercicio 4 (2 puntos)**.** Una empresa de mantenimiento eléctrico debe inspeccionar una red de n transformadores interconectados. Cada transformador está identificado por un número del 1 al n, y se conoce el conjunto de cables subterráneos que conectan pares de transformadores, representado mediante pares (i, j). Por razones de seguridad, no es posible inspeccionar directamente los cables. En cambio, se debe enviar personal técnico a ciertos transformadores para monitorear los cables conectados a ellos desde ahí. Cada equipo técnico ubicado en un transformador puede inspeccionar todos los cables que están conectados directamente a ese transformador. Por ejemplo, si hay 7 transformadores y 7 cables conectados como sigue: $(1, 2), (1, 3), (3, 4), (3, 5), (4, 6), (5, 6), (6, 7)$, entonces ubicar personal técnico en los transformadores $\{1, 3, 6\}$ permite cubrir todos los cables de la red. Diseñe un algoritmo basado en la técnica de exploración en grafos que permita conocer el número *mínimo* de transformadores donde se debe ubicar personal técnico para que cada cable esté supervisado al menos desde uno de sus extremos.

Ejercicio 5 (2 puntos)**.** Se dispone de K euros para hacer la compra y tenemos una lista de n posibles productos que podemos comprar. Cada producto i tiene un precio, $p(i)$ (que será siempre un número entero), y una utilidad, $u(i)$. De cada producto podemos comprar como máximo 2 unidades. Además, tenemos una oferta según la cual la segunda unidad nos cuesta 1 euro menos. Queremos elegir los productos a comprar, y cuántos de cada tipo, *maximizando* la utilidad de los productos comprados. Diseñe un algoritmo basado en la técnica de programación dinámica para resolver el problema. Aplíquelo, construyendo la tabla correspondiente, para el caso en que hay $n = 3$ productos, con precios $p = (3, 4, 6)$, utilidades $u = (7, 8, 11)$ y el presupuesto es $K = 10$.

Ejercicio 1 (2 puntos)**.** La ecuación en recurrencias resultante del análisis de eficiencia de un algoritmo depende de una condición dada C, de modo que tenemos $T(n) = 2T(n/4) + n^2$ si C es cierta, y $T(n) = 4T(n/2) + \log_2(n)$ si C es falsa. Determine los órdenes de complejidad O y Ω del algoritmo y justifique si este tiene, o no, orden de complejidad Θ.

Primero resolvemos las dos recurrencias por separado. Para la primera:

$$T(n) = 2T(n/4) + n^2$$

Hacemos el cambio de variable $n = 2^h$:

$$T\left(2^h\right) = 2T\left(2^{h-2}\right) + \left(2^h\right)^2$$

Escribiendo $t_h = T(2^h)$:

$$t_h = 2t_{h-2} + 4^h \Longrightarrow t_h - 2t_{h-2} = 4^h$$

Obtenemos:

- $b = 4$, $q(h) = 1$, de grado 0; lo que nos da el término $(x - 4)$.
- $p(x) = x^2 - 2 \Longrightarrow x = \pm\sqrt{2}$, lo que nos da los términos $\left(x - \sqrt{2}\right)\left(x + \sqrt{2}\right)$.

En definitiva, el polinomio característico de la recurrencia es:

$$p(x) = \left(x - \sqrt{2}\right)\left(x + \sqrt{2}\right)(x - 4)$$

Por lo que:

$$t_h = C_1\left(\sqrt{2}\right)^h + c_2\left(-\sqrt{2}\right)^h + c_3 4^h$$

Al deshacer el cambio de variable, tenemos $h = \log_2 n$, por lo que:

$$\begin{aligned} T(n) &= c_1\left(\sqrt{2}\right)^{\log_2 n} + c_2\left(-\sqrt{2}\right)^{\log_2 n} + c_3 4^{\log_2 n} \\ &= C - 1\sqrt{n} + c_2\left(-\sqrt{2}\right)^{\log_2 n} + c_3 n^2 \end{aligned}$$

De donde $T(n) \in O(n^2)$ si C es cierta.

Para la segunda recurrencia, procedemos de forma análoga:

$$T(n) = 4T(n/2) + \log_2 n$$

Hacemos el cambio de variable $n = 2^h$:

$$T\left(2^h\right) = 4T\left(2^{h-1}\right) + \log_2 2^n$$

Escribiendo $t_h = T\left(2^h\right)$:

$$t_h = 4t_{h-1} + h \Longrightarrow t_h - 4t_{h-1} = h$$

Obtenemos:

- $b = 1$, $q(h) = h$, de grado 1; lo que nos da el término $(x-1)^2$.
- $p(x) = (x-4)$

En definitiva, el polinomio característico de la recurrencia es:

$$p(x) = (x-4)(x-1)^2$$

Por lo que:

$$t_h = c_1 4^h + c_2 1^h + c_3 h 1^h = c_1 4^h + c_2 + c_3 h$$

Si deshacemos el cambio, tenemos $h = \log_2 n$, con lo que:

$$\begin{aligned} T(n) &= C_1 4^{\log_2 n} + c_2 + c_3 \log_2 n \\ &= c_1 n^2 + c_2 + c_3 \log_2 n \end{aligned}$$

De donde tenemos que $T(n) \in O(n^2)$ si C es falsa.

En ambos casos tenemos que $T(n) \in O(n^2)$, por lo que tanto el mejor como el peor caso coinciden, es decir, $T(n) \in O(n^2)$ y $T(n) \in \Omega(n^2)$. Como coinciden, tendremos también que $T(n) \in \Theta(n^2)$.

Ejercicio 2 (2 puntos)**.** Dado un vector de números enteros de tamaño n, ordenado de menor a mayor, y donde puede haber números duplicados, se quiere contar cuántas veces aparece un determinado número x o indicar que dicho número no aparece ninguna vez. Diseñe un algoritmo basado en la técnica "divide y vencerás" que resuelva el problema de la forma más eficiente posible. ¿Cuál es el orden de complejidad O del algoritmo diseñado?

La idea consiste en modificar la búsqueda binaria para que localice la primera ocurrencia de x y la última. Así, si la primera ocurrencia de x está en la posición p y la última está en la posición u, el número de ocurrencias de x es $u - p + 1$. Como se hacen dos búsquedas binarias, que tienen una eficiencia $O(\log_2 n)$, el algoritmo resultante es $O(\log_2 n)$.

Mostramos a continuación el pseudocódigo de lo que podría ser una implementación de esta idea:

```
1  cuantos(v[], inicio, fin, buscado){
       p = posicion_izquierda(v, inicio, fin, buscado);
       if(v[p] == buscado){
          u = posicion_derecha(v, inicio, fin, buscado);
5         return u - p + 1;
       }else
          return 0;
   }

10 posicion_izquierda(v[], inicio, fin, buscado){
       resultado = -1;
       while(inicio <= fin){
          mitad = (inicio + fin)/2;
          if(v[mitad] == buscado){
15           resultado = mitad;
```

```
            fin = mitad - 1;
         }else if(buscado < v[mitad])
            fin = mitad - 1;
         else
            inicio = mitad + 1;
      }

      return resultado;
   }

   posicion_derecha(v[], inicio, fin, buscado){
      resultado = -1;
      while(inicio <= fin){
         mitad = (inicio + fin)/2;
         if(v[mitad] == buscado){
            resultado = mitad;
            inicio = mitad + 1;
         }else if(buscado < v[mitad])
            fin = mitad - 1;
         else
            inicio = mitad + 1;
      }

      return resultado;
   }
```

Ejercicio 3 (2 puntos)**.** Una empresa de transportes hace de forma regular una ruta con su flota de vehículos. A lo largo de esta ruta, se ha identificado un conjunto de n puntos kilométricos específicos $(k_1, k_2, \ldots, k_n)$ en los que sus vehículos deben detenerse. Con el fin de facilitar las tareas de mantenimiento y control, se desea instalar un conjunto de estaciones a lo largo de la ruta. Cada estación puede cubrir los vehículos que se encuentran en un rango de L kilómetros. Se desea identificar dónde ubicar el *menor* conjunto de estaciones, garantizando que todos los puntos de parada prefijados queden cubiertos por al menos una estación. Diseñe un algoritmo voraz que resuelva este problema de forma óptima.

Conjunto de candidatos. Los n puntos kilométricos específicos, $k_1, k_2, \ldots, k_n$ en los que los vehículos deben detenerse.

Conjunto de candidatos elegidos. La solución es una tupla $(x_1, x_2, \ldots, x_s)$ son $s \leqslant n$ que en cada x_i (con $i \in \{1, \ldots, s\}$) contiene un punto kilométrico específico en el que se ubicuará una estación.

Función solución. Todos los puntos kilométricos específicos $k_1, k_2, \ldots, k_n$ están cubiertos por al menos una estación de las ubicadas en $x_1, x_2, \ldots, x_s$:

$$\forall k_i (i \in \{1, \ldots, n\})\ \exists j \in \{1, \ldots, s\} \text{ tal que } |k_i - x_j| \leqslant L/2$$

Función de factibilidad. La selección de cualquier punto kilométrico específico para ubicar en él una estación siempre da lugar a una solución factible.

Función objetivo. Número de puntos kilométricos específicos elegidos para ubicar en ellos una estación.

Función selección. Considerando el punto kilométrico específico que cubrirá más a la izquierda de los restantes, se selecciona el punto kilométrico específico más a la derecha de este dentro del rango de cobertura permitida ($L/2$), y se ubica ahí una estación.

La idea consiste en que siempre que se encuentre un punto kilométrico específico sin cubrir, se coloca una estación en el punto kilométrico específico más a la derecha dentro del rango de cobertura permitida ($L/2$).

```
puntos[1..n];   // ordenados de menor a mayor
x[];
i = 1;
num_estaciones = 0;
while(i <= n){
   inicio = puntos[i];
   j = i;
   while(j <= n && puntos[j] <= inicio + L/2)
      j++;
   estacion = puntos[j-1];
   num_estaciones++;
   x[num_estaciones] = estacion;
   while(i <= n && puntos[i] <= estacion + L/2)
      i++;
}
```

Demostración de la optimalidad. Sea $X = \{x_1, \ldots, x_s\}$ la solución dada por el algoritmo voraz. Supongamos que existe otra solución $Y = \{y_1, \ldots, y_r\}$ tal que $r < s$. Sea x_1 la primera estación en la solución voraz. Por definición, cubre un conjunto de puntos kilométricos que vienen desde k_1 hasta $k_j \leqslant x_1 + L/2$. La estación x_1 fue seleccionada porque es el punto kilométrico más a la derecha que cubre el primer punto kilométrico k_1, es decir, $x_1 \leqslant k_1 + L/2$. Cualquier estación colocada más a la izquierda que x_1 cubre menos puntos, ya que el intervalo se desplaza a la izquierda. Por tanto, ninguna estación más a la izquierda que x_1 puede cubrir más puntos kilométricos que la que elige el algoritmo voraz. Ahora, supongamos que la solución óptima Y, usa una estación y_1 en lugar de x_1 para cubrir esos puntos. Como el algoritmo voraz elige la mejor (más a la derecha) estación que cubre desde k_1, $y_1 \leqslant x_1$. Por tanto, no cubre más puntos que x_1. Si repetimos el proceso, por cada atracción y_i de la solución óptima que cubre un bloque que también cubre x_i, podemos reemplazar y_i por x_i sin perder puntos kilométricos cubiertos. Haciendo esta transformación convertimos la solución Y en X, pero habíamos supuesto que $|Y| < |X|$, contradicción.

Ejercicio 4 (2 puntos)**.** Una empresa de mantenimiento eléctrico debe inspeccionar una red de n transformadores interconectados. Cada transformador está identificado por un número del 1 al n, y se conoce el conjunto de cables subterráneos que conectan pares de transformadores, representado mediante pares (i, j). Por razones de seguridad, no es posible inspeccionar directamente los cables. En cambio, se debe enviar personal técnico a ciertos transformadores para monitorear los cables conectados a ellos desde ahí. Cada equipo técnico ubicado en un transformador

puede inspeccionar todos los cables que están conectados directamente a ese transformador. Por ejemplo, si hay 7 transformadores y 7 cables conectados como sigue: $(1,2),(1,3),(3,4),(3,5),(4,6),(5,6),(6,7)$, entonces ubicar personal técnico en los transformadores $\{1,3,6\}$ permite cubrir todos los cables de la red. Diseñe un algoritmo basado en la técnica de exploración en grafos que permita conocer el número *mínimo* de transformadores donde se debe ubicar personal técnico para que cada cable esté supervisado al menos desde uno de sus extremos.

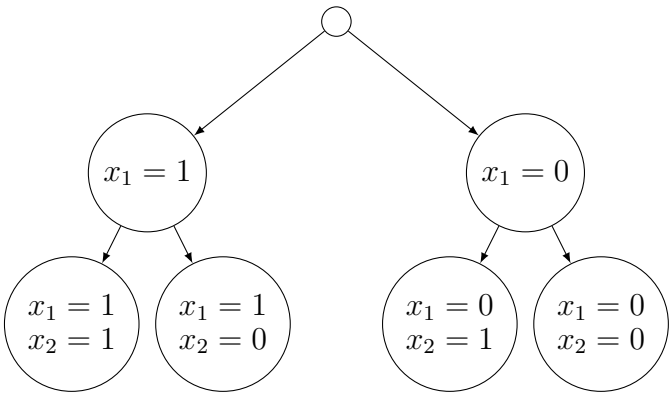
Vamos a diseñar un algoritmo de vuelta atrás (backtracking). La solución se va a representar como una tupla $X = (x_1, x_2, \ldots, x_n)$, donde cada x_i tendrá un valor de 0 para indicar que no se enviará un técnico al transformador $i-$ésimo, o de 1 para indicar que sí se enviará un técnico al transformador $i-$ésimo.

Restricciones explícitas. $x_i \in \{0,1\}$, $\forall i \in \{1,\ldots,n\}$.

Restricciones implícitas. Sea $M[i][j]$ la matriz de adyacencia con valores 1 y 0 para indicar que existe un cable (1) o no (0) entre los transformadores i y j, entonces:

$$\forall i,j \text{ tales que } M[i][j] = 1 \Longrightarrow x_i = 1 \vee x_j = 1$$

Árbol de estados. Árbol binario de altura $n+1$. En cada nivel i, se decide si se manda (1) o no (0) un técnico al transformador $i-$ésimo.



Función objetivo. Número de técnicos enviados a los transformadores, $\sum_{i=1}^{n} x_i$.

Función de acotación. Función que poda el árbol si al añadir a la solución parcial $x_1,\ldots,x_i$ el valor correspondiente de x_{i+1} no se satisfacen las restricciones implícitas. También se poda si el numero de técnicos asignados a los transformadores de la solución parcial es igual o mayor que el de la mejor solución encontrada hasta el momento.

Una implementación en pseudocódigo de esta solución es la siguiente:

```
1  transformadores(x[1..n], M[1..n][1..n], mejor_solucion, k){
      if(k == n+1){   // nodo hoja (solución completa)
         if(num_tecnicos(x,n) < num_tecnicos(mejor_solucion, n))
            mejor_solucion = x;
5     }else{
         x[k] = 1; // asignar trabajador al transformador k-ésimo
         if(num_tecnicos(x,k) < num_tecnicos(mejor_solucion, k))
```

```
            transformadores(x, M, mejor_solucion, k+1);

        x[k] = 0; // no se asigna un trabajador al k-ésimo transformador
        if(factible(x,M,k)) // si se cumplen las restricciones implícitas
            transformadores(x, M, mejor_esolucion, k+1);
    }
}

// devuelve si se satisfacen o no las restricciones implícitas
factible(x[1..n], M[1..n][1..n], k) {
    for(i = 1; i <= k; i++){
        if(M[k][i] == 1)
            if(x[i] == 0)
                return false;
    }
    return true;
}

// devuelve el nº de técnicos asignados a transformadores
num_tecnicos(x[1..n], k){
    tecnicos = 0;
    for(i = 1; i <= k; i++)
        tecnicos = tecnicos + x[i];
    return tecnicos;
}
```

Ejercicio 5 (2 puntos)**.** Se dispone de K euros para hacer la compra y tenemos una lista de n posibles productos que podemos comprar. Cada producto i tiene un precio, $p(i)$ (que será siempre un número entero), y una utilidad, $u(i)$. De cada producto podemos comprar como máximo 2 unidades. Además, tenemos una oferta según la cual la segunda unidad nos cuesta 1 euro menos. Queremos elegir los productos a comprar, y cuántos de cada tipo, *maximizando* la utilidad de los productos comprados. Diseñe un algoritmo basado en la técnica de programación dinámica para resolver el problema. Aplíquelo, construyendo la tabla correspondiente, para el caso en que hay $n = 3$ productos, con precios $p = (3, 4, 6)$, utilidades $u = (7, 8, 11)$ y el presupuesto es $K = 10$.

Definición de la solución. Se plantea la solución como una secuencia de decisiones $x_1, x_2, \ldots, x_n$, donde cada x_i indica cuántas unidades se compran del producto $i-$ésimo.

Sea $U(i, j)$ La máxima utilidad que se puede conseguir con j euros cuando se consideran los productos del 1 al i, la solución viene dada por $U(n, K)$.

Verificación del principio de optimalidad. Si $x_1, x_2, \ldots, x_n$ es óptima para $U(n, K)$, entonces hay que demostrar que $x_1, x_2, \ldots, x_{n-1}$ es óptima para:

- $U(n-1, K)$ si $x_n = 0$.
- $U(n-1, K-p(n))$ si $x_n = 1$.
- $U(n-1, K-2p(n)+1)$ si $x_n = 2$.

Se demuestra por reducción al absurdo. Para el caso $x_n = 0$, si no fuese así, entonces existiría una solución $y_1, \ldots, y_{n-1}$ de forma que:

$$\sum_{i=1}^{n-1} y_i u(i) > \sum_{i=1}^{n-1} x_i u(i), \quad \text{con} \quad \sum_{i=1}^{n-1} y_i p(i) \leqslant K$$

Pero entonces, haciendo $y_n = x_n = 0$, tenemos que:

$$\sum_{i=1}^{n} y_i p(i) \leqslant K$$

Luego $y_1, \ldots, y_n$ es una solución para $U(n, K)$ y además:

$$\sum_{i=1}^{n} y_i u(i) > \sum_{i=1}^{n} x_i u(i)$$

Por lo que $x_1, \ldots, x_n$ no sería óptima, contradicción. Los casos $x_n \in \{1, 2\}$ se demustran de forma análoga.

Definición recursiva de la solución óptima. Definimos:

$$U(i,j) = \begin{cases} 0 & \text{si } i = 0 \vee j = 0 \\ U(i-1,j) & \text{si } p(i) > j \\ \text{máx}\{U(i-1,j), U(i-1,j-p(i)) + u(i)\} & \text{si } 2p(i) - 1 > j \\ \text{máx}\{U(i-1,j), U(i-1,j-p(i)) + u(i), U(i-1,j-2p(i)+1) + 2u(i)\} & \text{en otro caso} \end{cases}$$

Por lo que una implementación en pseudocódigo del algoritmo sería:

```
for i = 0 to n:
   U(i,0) = 0;
for j = 1 to n:
   U(0,j) = 0;
for i = 1 to n:
   for j = 1 to K:
      U(i,j) = U(i-1,j);
      V(i,j) = 0;    // 0 unidades
      if(p(i) <= j)
         if(U(i-1,j-p(i)) + u(i) > U(i,j))
            U(i,j) = U(i-1,j-p(i)) + u(i);
            V(i,j) = 1;    // 1 unidad
      if(2p(i) + 1 <= j)
         if(U(i-1,j-2p(i)+1) + 2*u(i) > U(i,j))
            U(i,j) = U(i-1,j-2*p(i)+1) + 2*u(i);
            V(i,j) = 2;    // 2 unidades

// La tabla V se utiliza para recuperar la solución:
i = n
j = K
while(i >= 1)
   solucion(i) = V(i,j);
   if(V(i,j) == 2)
      j = j - 2*p(i) + 1;
   else
      j = j - V(i,j)*p(i);
   i--;
```

Construimos las tablas U y V para el caso $n = 3$, $K = 10$, $p = (3, 4, 6)$ y $u = (7, 8, 11)$:

U	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	7	7	14	14	14	14	14	14
2	0	0	0	7	8	14	14	16	16	22	23
3	0	0	0	7	8	14	14	16	16	22	23

V	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	2	2	2	2	2	2
2	0	0	0	1	0	0	2	2	1	2
3	0	0	0	0	0	0	0	0	0	0