TEMARIO-Y-EJERCICIOS-DE-EXAMEN-R...

Anónimo

Diseño de Algoritmos

3° Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

TEMA 1: ALGORITMOS DEVORADORES (DEVORÓ 💋)

Esquema Algoritmo Devorador

$$
\begin{aligned}
&devorador(C) \rightarrow S \\
&S \leftarrow \varnothing \\
&\text{mientras } \neg solución(S) \wedge C \neq \varnothing \\
&\qquad p \leftarrow selección(C) \\
&\qquad C \leftarrow C - \{p\} \\
&\qquad \text{si } factible(S \cup \{p\}) \\
&\qquad\quad S \leftarrow S \cup \{p\}
\end{aligned}
$$

Elementos:
- Conjunto de candidatos (C)
- Conjunto de candidatos seleccionados (S)
- Función solución: Comprueba si un conjunto de candidatos (seleccionados) es una solución.
- Función selección: Indica el candidato más prometedor de los que quedan.
- Función de factibilidad: Verifica si un conjunto de candidatos se puede ampliar (con el nuevo candidato seleccionado) para obtener una solución.
- Función objetivo: Asocia un valor a una solución, y que queremos optimizar.
- Objetivo: minimizar o maximizar

A destacar:
- El candidato más prometedor puede no ser único.
- Una vez seleccionado un candidato (p), este se elimina del conjunto original (C) [**MUY IMPORTANTE**].
- Si dicho candidato resulta factible (evaluado en la función de factibilidad), se añade al conjunto solución (S).
- A cada paso, se toma la mejor opción sin preocuparse del futuro.
- La función solución puede indicarse de forma implícita, incluyendo una condición booleana.
- Existe un objetivo implícito, que puede ser maximizar o minimizar el resultado del algoritmo devorador. Dicho comportamiento vendría definido en el criterio de selección de la función selección.
- Es posible que al terminar el algoritmo voraz, S no sea una solución, o ni siquiera una óptima.

Ejemplo (Algoritmo de la mochila continuo):
Dados un conjunto de objetos (O), cada uno con un valor (v) y un peso (p), y una mochila con una capacidad (c) que limita el peso total que puede transportar, se desea hallar la composición de la mochila que maximiza el valor de la carga.

Estrategia → Seleccionar los objetos en orden decreciente de relación peso/valor.

```
mochila(O, c) → S
C ← O
S ← ∅
mientras c ≠ 0 ∧ C ≠ ∅
        (v, p) ← selecciona-objeto(C)
        C ← C − {(v, p)}
        si p ≤ c
          S ← S ∪ {(v, p)}
          c ← c − p
        si no
          S ← S ∪ {(v · c/p, c)}
          c ← 0
```

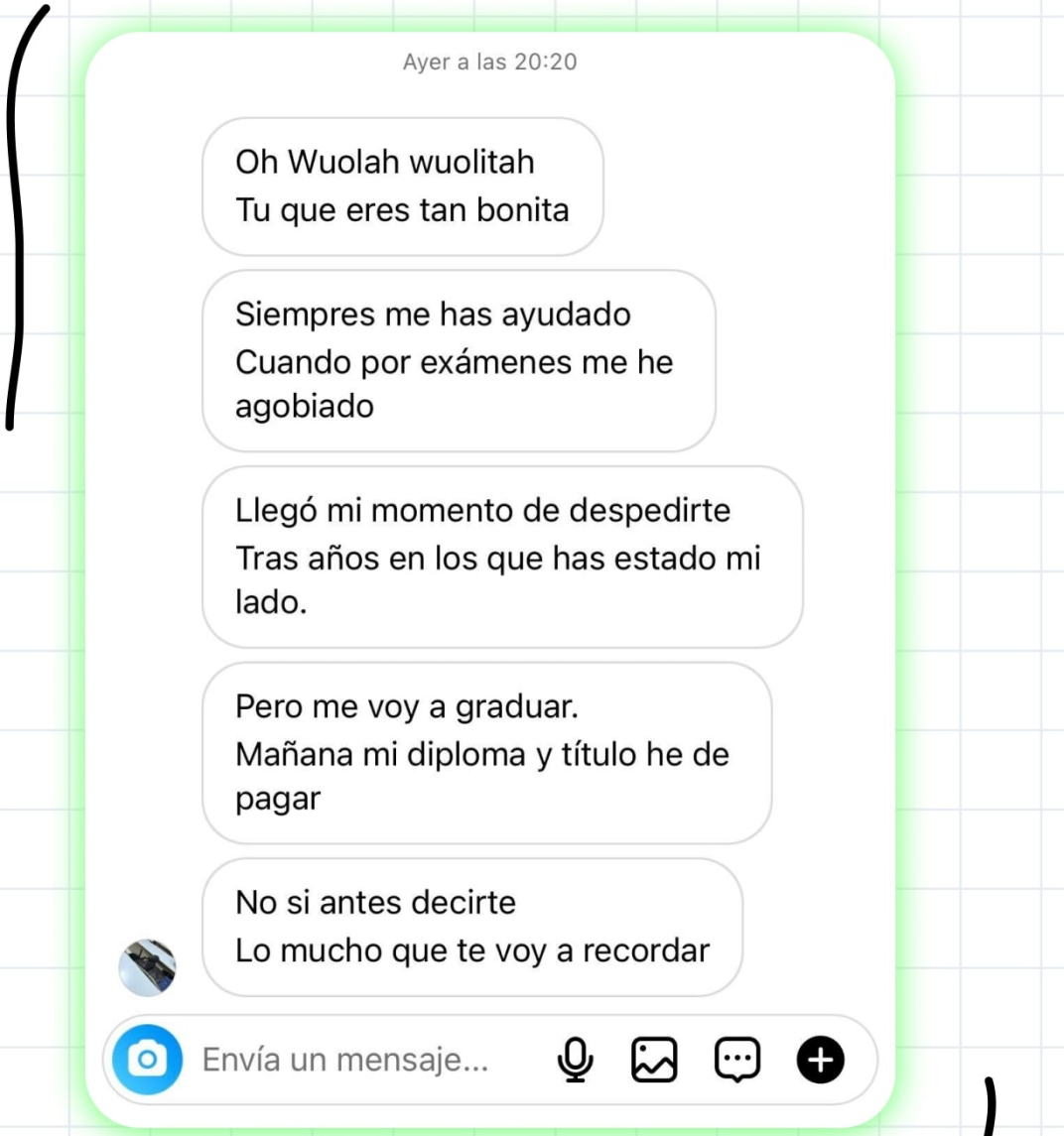
```
selecciona-objeto(C) → (v, p)
r ← −∞
para todo (a, b) ∈ C
      si a/b > r
        r ← a/b
        (v, p) ← (a, b)
```

Elementos
- Conjunto de candidatos: Los objetos (O) [Estos se copiarán al conjunto C para evitar la modificación del conjunto original]
- Conjunto de candidatos seleccionados → Solución al algoritmo, inicialmente vacío
- Función solución: ¿Se ha llenado ya la mochila?
- Función de selección: Elige el objeto con máxima relación peso/valor (*selecciona-objeto*)
- Función de factibilidad: ¿Se puede introducir el objeto sin exceder la capacidad?
- Función objetivo: El valor total de los objetos devueltos.
- Objetivo: Maximizar.

Análisis
Sea n = |O| = |C|
→ En el peor caso, el bucle (mientras) realiza **n** iteraciones (por ejemplo, con una mochila de capacidad igual a la suma de los pesos de todos los objetos).
→ En el peor caso, el conjunto de objetos estará ordenado por relación peso/valor ascendentemente, por lo que siempre deberemos iterar hasta el último elemento para encontrar el valor máximo.
Por tanto, la función de selección efectúa **n** operaciones elementales, en el peor de los casos.
→ Éstos órdenes, al estar anidados, deberemos multiplicarlos, por lo que obtendremos una **complejidad temporal $t(n) \in O(n^2)$**.

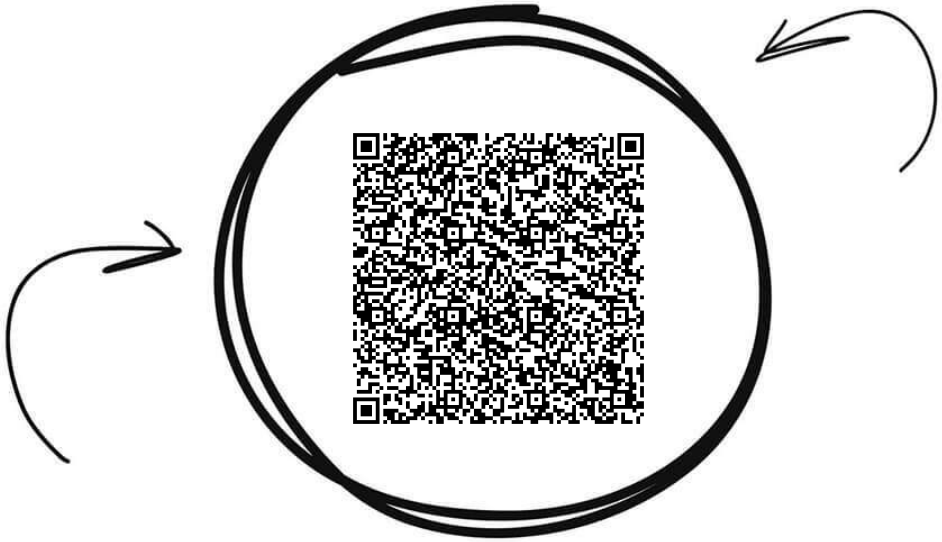
Que no te escriban poemas de amor cuando terminen la carrera 🚩🚩🚩🚩🚩🚩

(a nosotros por suerte nos pasa)



WUOLAH

Diseño de Algoritmos



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

Ejemplo (Algoritmo de la mochila continuo optimizado):
Mismo problema que planteamos anteriormente, con la excepción de:
→ Se pre-ordenan los elementos en función de la relación peso/valor de manera descendente (los más valiosos primero).
→ La función de selección puede pasar a ser extrae-primero, que selecciona el primer elemento del conjunto de candidatos y lo elimina de este (extraer), por lo que no resulta necesario eliminar el objeto del conjunto explícitamente.

$mochila(O, c) \rightarrow S$
$C \leftarrow ordena(O)$
$S \leftarrow \varnothing$
mientras $c \neq 0 \wedge C \neq \varnothing$
    $(v, p) \leftarrow extrae\text{-}primero(C)$
    si $p \leq c$
        $S \leftarrow S \cup \{(v, p)\}$
        $c \leftarrow c - p$
    si no
        $S \leftarrow S \cup \{(v \cdot c/p, c)\}$
        $c \leftarrow 0$

Análisis
Al pre ordenar los objetos, la selección y eliminación (extracción) de objetos se convierte en una operación elemental (de orden constante Θ(1)).
→ Coste de recorrer los candidatos (C): O(n)
→ Coste de ordenar los elementos (*ordena(O)*): O(n log n)

Por tanto, por la regla del máximo:

$$t(n) = O(n) + O(n \log n) = O(n \log n)$$

$$\mathbf{t(n) \in O(n \log n)}$$

Árbol de expansión mínimo

Dado un **grafo** (G), con sus respectivos vértices (V) y aristas (A) **[G = (V,A)]**, conexo y ponderado con valores no negativos en sus aristas, llamamos **árbol de expansión mínimo** a aquel subgrafo que una todos los vértices (**<< expansión >>**) con el mínimo número de aristas [|V| - 1] y cuyo valor de suma (la de todas las aristas ponderadas) sea **<< mínimo >>**.

Un grafo puede tener más de un árbol de expansión mínimo.

Se denomina árbol porque el resultado es un grafo acíclico y conexo [definición de árbol].

Supondremos la existencia de un orden en A inducido por la función de ponderación p:

$$\{i, j\} \leq \{k, l\} \Leftrightarrow p(i, j) \leq p(k, l)$$

Según la **aproximación para el control de ciclos** en la generación de árboles de expansión mínimo, podremos distinguir **dos algoritmos**:

→ **Algoritmo de Kruskal**

→ **Algoritmo de Prim**

1. Algoritmo de Kruskal

El algoritmo de Kruskal construye un árbol de expansión mínimo a partir de un bosque de G (inicialmente, vacío).

```
Kruskal(V, A) → S
C ← A
S ← ∅
n ← |V|
p ← partición-inicial(n)
ordena(C)
mientras |S| ≠ n − 1
        {i, j} ← extrae-primero(C)
        (e1, e2) ← (búsqueda(p, n, i), búsqueda(p, n, j))
        si e1 ≠ e2
           unión(p, n, e1, e2)
           S ← S ∪ {{i, j}}
```

→ Se emplean estructuras de partición (para la comprobación de ciclos)

→ Para mayor eficiencia, se preordena A.

→ A también podría ser ordenado mediante un montículo invertido para A.

Elementos:

→ Conjunto de candidatos (C): Aristas (A)

→ Función solución: ¿Se tiene ya un árbol de expansión? Esto sucederá cuando existan n-1 aristas exactamente, siempre que se compruebe la condición de ciclo para cada arista introducida (factibilidad).

→ Función selección: Elige la arista de valor mínimo, disponible en el conjunto (C).

→ Función de factibilidad: ¿Se puede incluir la arista sin introducir ciclos?

→ Función objetivo: El valor total de las aristas devueltas.

→ Objetivo: Minimizar.

Secuencias de instrucciones

1. Se crea una copia de los objetos (A) en (C) para evitar la modificación del conjunto original.
2. Se genera el conjunto solución (S) de aristas, inicialmente vacío.
3. *n* será igual al número de vértices, número que será necesario comparar para la función solución.
4. Se crea una partición (p) inicial de *n* nodos (un conjunto de conjuntos, al principio todos disjuntos)
5. Se pre-ordenan las aristas, para obtener una mejor eficiencia
6. Se comprueba la función solución (¿existen n-1 aristas en el conjunto solución?)
   - 6.1 Se extrae (selecciona y elimina) del conjunto el primer candidato que, al estar pre ordenado, será la arista con menor peso.
   - 6.2 Se realizan dos búsquedas paralelas, para encontrar los representantes de partición (e1,e2) de ambos vértices seleccionados.
   - 6.3 La función de factibilidad comprobará si ambos vértices se encuentran en particiones distintas (según sus representantes)
     - 6.3.1 Si efectivamente se tratan de dos particiones distintas, ambas particiones se unen mediante sus representantes.
     - 6.3.2 Se amplía el conjunto solución con la arista seleccionada.

Análisis (usando un bosque con control de alturas)

Sea n = |V| y a = |A|:

→ Se ordenan las aristas (merge-sort): O(a log a)

→ Se crea una partición inicial: O(n)

→ Se ejecutan 2a búsquedas como máximo: 2 O(a log n)

→ Se ejecutan n-1 uniones exactamente: O(n)

Por tanto, y aplicando la regla del máximo:
$t(n, a) \in O(a \log a) + O(n) + O(a \log n) + O(n)$
$t(n, a) \in O(a \log a)$

dependiendo de la densidad del grafo, se obtiene que:

$$t(n,a) \in \begin{cases} O(n \log n), & a \in \Theta(n) \\ O(n^2 \log n), & a \in \Theta(n^2) \end{cases}$$

2. Algoritmo de Prim

El algoritmo de Prim construye un árbol de expansión mínimo a partir de un subárbol de G. Inicialmente, este tiene un único vértice (el primero, por ejemplo).
Esta versión actúa sobre un grafo representado por su matriz de pesos (p):

```
Prim(p, n) → S
C ← ∅
desde j ← 2 hasta n
        C ← C ∪ {j}
        (c[j], d[j]) ← (1, p[1, j])
S ← ∅
mientras C ≠ ∅
        k ← selecciona-vértice(C, d)
        C ← C − {k}
        S ← S ∪ {{c[k], k}}
        para todo j ∈ C
              si p[k, j] < d[j]
                 (c[j], d[j]) ← (k, p[k, j])
```

```
selecciona-vértice(C, d) → k
v ← ∞
para todo j ∈ C
      si d[j] < v
         v ← d[j]
         k ← j
```

Elementos:

→ Conjunto de candidatos (C): Vértices (n)
→ Función solución: ¿Se tiene ya un árbol de expansión? Esto sucederá cuando no existan más vértices en el conjunto de candidatos.
→ Función selección: Calcula el vértice de C más cercano al conjunto de vértices seleccionados.
→ Función de factibilidad: ¿Se puede incluir la arista sin introducir ciclos?
→ Función objetivo: El valor total de las aristas devueltas.
→ Objetivo: Minimizar.

Secuencias de instrucciones

1. Se inicializa el conjunto de candidatos (vértices) a vacío.
2. Recorriendo el rango [2,n] … (se empieza por el 2 para excluir al primer vértice, considerado origen)
    2.1 Ampliamos el conjunto de candidatos con un vértice.
    2.3 En la posición j del vector de caminos colocaremos un 1 (primer vértice) y en la matriz de distancias, en la posición j también, la distancia entre el vértice 1 (origen) y el vértice j.
3. Inicializamos el conjunto solución (S) a vacío.
4. Recorremos el conjunto de candidatos mientras este no esté vacío.
    4.1 Seleccionamos el vértice (k) con menor distancia respecto al vértice de origen.
    4.2 Eliminamos el vértice seleccionado del conjunto de candidatos (*característico alg. devorador*)
    4.3 Ampliamos el conjunto solución con una arista que une el vértice predecesor al seleccionado y el propio vértice.
    4.4 Recorremos todos los vértices restantes en el conjunto de candidatos para actualizar su distancia y nodos predecesores.
        4.4.1 Si el coste de ir del vértice seleccionado [en el paso 4.1] (k) al vértice j, es menor que el coste directo de ir desde el nodo origen al nodo j…
            4.4.1.1 Actualizamos el nodo predecesor de j al nodo seleccionado k
            4.4.1.2 Reemplazamos la distancia directa con la distancia entre k y j.

Análisis

Sea n = |V|:
→ El bucle externo se realiza n-1 veces, siendo de orden O(n)
→ Los dos bucles internos varían desde n-1 hasta 1, siendo de orden O(n)
Por tanto, operando y aplicando la regla del máximo:

$$t(n) \in O(n) * O(n) + O(n) * O(n)$$

$$\mathbf{t(n) \in O(n^2)}$$

Caminos mínimos

Dado un grafo orientado y ponderado con valores no negativos, se trata de encontrar el camino mínimo que conduce de un vértice a otro.

Algoritmo de Dijkstra

Permite obtener dicho camino mínimo, calculando desde un vértice origen a todos los demás.
Con la matriz de pesos, sus valores se calculan como sigue:

```
Dijkstra(p, n, i) → d
C ← ∅
desde j ← 1 hasta n
        C ← C ∪ {j}
        d[j] ← p[i,j]
C ← C − {i}
mientras C ≠ ∅
        k ← selecciona-vértice(C, d)
        C ← C − {k}
        para todo j ∈ C
                d[j] ← mín(d[j], d[k] + p[k,j])
```

```
selecciona-vértice(C, d) → k
v ← ∞
para todo j ∈ C
        si d[j] < v
                v ← d[j]
                k ← j
```

Análisis

Sea n = |V|:
→ El bucle externo se realiza n-1 veces, siendo de orden O(n)
→ Los dos bucles internos varían desde n-1 hasta 1, siendo de orden O(n)
Por tanto, operando y aplicando la regla del máximo:

$$t(n) \in O(n) * O(n) + O(n) * O(n)$$

$$\mathbf{t(n) \in O(n^2)}$$

Recuperación del camino mínimo con Dijkstra

```
Dijkstra(p, n, i) → (c, d)
C ← ∅
desde j ← 1 hasta n
        C ← C ∪ {j}
        (c[j], d[j]) ← (i, p[i,j])
C ← C − {i}
mientras C ≠ ∅
        k ← selecciona-vértice(C, d)
        C ← C − {k}
        para todo j ∈ C
                si d[k] + p[k,j] < d[j]
                        (c[j], d[j]) ← (k, d[k] + p[k,j])
```

← **La siguiente modificación (respecto al algoritmo original) permite recuperar los caminos mínimos.**

El predecesor de j en el camino mínimo de i a j es c[j].
→ Esto permite reconstruir el camino hacia atrás, comenzando por el final.

TEMA 2: PROGRAMACIÓN DINÁMICA

Problemas de optimización

Los problemas se resuelven en base a una secuencia de decisiones. Cuando dicha secuencia es óptima, el resultado también será óptimo.
Cuando el resultado de un problema **debe ser** óptimo, estamos ante un problema de optimización.

Principio de optimalidad

Cualquier problema de optimización puede resolverse por fuerza bruta,
evaluando las p^n [p: posibilidades, n: decisiones] secuencias posibles (explosión combinatoria).
El principio de optimalidad permite reducir el número de secuencias posibles, eliminando las que descubre que no son óptimas.
No es una ley que se cumpla en todos los problemas, pero en los que se cumple indica cuándo una decisión no es apropiada.

Problema de la mochila discreta

Dados n objetos, cada uno con un valor vi y un peso pi, y una mochila con una capacidad, c, que limita el peso total que puede transportar, se desea hallar la composición de la mochila que maximiza el valor de la carga.

Para todo problema de optimización, deberemos seguir estos 4 pasos:
→ Estructurar la **tabla de subproblemas resueltos**.
→ **Plantear** una **función** que representen todos los casos posibles a la hora de tomar decisiones.
→ **Diseñar** el **algoritmo** correctamente.
→ **Analizar temporal** y **espacialmente** el algoritmo.

Tabla de subproblemas resueltos

La tabla de subproblemas resueltos para los problemas de optimización, a través del enfoque de programación dinámica, deberá representar los distintos estados del problema al tomar o descartar una opción.

En este problema, tenemos n objetos distintos y c capacidades distintas. Parece intuitivo diseñar una tabla de dimensiones (n x c) para recoger todos los casos posibles.

Planteamiento de la función

En este caso, deberemos plantear una función en base al número de objetos disponibles y la capacidad restante de la mochila, de la siguiente forma:

$$f(n,c) = \begin{cases} 0, & n = 1 \wedge c < p_1 \\ v_1, & n = 1 \wedge c \geq p_1 \\ f(n-1,c), & n > 1 \wedge c < p_n \\ \text{máx}\{f(n-1,c), f(n-1,c-p_n)+v_n\}, & n > 1 \wedge c \geq p_n \end{cases}$$

Si solo hay un objeto i (n = 1), puede ocurrir que:
→ $c < p_i$: no se incluye, el valor máximo de la carga será 0.
→ $c \geq p_i$: se incluye, el valor máximo de la carga será vi.

Si hay varios objetos (n>1) y estamos con el objeto i, puede ocurrir que:
→ $c < p_i$: no se incluye, el valor máximo de la carga será el que se obtenga de considerar los restantes objetos.
→ $c \geq p_i$: el valor máximo de la carga será el máximo de:
→ El que se obtiene al descartar el objeto y considerar los restantes para rellenar la mochila.
→ El que se obtiene al incluir el objeto y considerar los restantes para rellenar la mochila.

Diseño del algoritmo

```
mochila(v, p, n, c) → r { v: Matriz de valores, p: Matriz de pesos, n: Número de objetos, c: Capacidad }
    desde j ← 0 hasta c
        si j < p[1]
            f [1, j] ← 0
        si no
            f [1, j] ← v[1]
    desde i ← 2 hasta n
        desde j ← 0 hasta c
            si j < p[i]
                f [i, j] ← f [i − 1, j]
            si no
                f [i, j] ← máx(f [i − 1, j], f [i − 1, j − p[i]] + v[i])
    r ← f [n, c]
```

Análisis

Temporal

Siendo n el número de objetos distintos y c la capacidad de la mochila:
→ Se realiza un primer bucle de c iteraciones, orden O(c)
→ Se realiza un segundo bucle de n iteraciones, orden O(n)
→ Dentro del segundo bucle, se realiza otro bucle de c iteraciones, orden O(c)
Por tanto, calculando, simplificando y aplicando la regla del máximo:
$t(n) \in O(c) + O(nc) \Rightarrow t(n) \in O(nc)$

Espacial

Referido a la estructura de la tabla de subproblemas resueltos.
En este caso, la tabla diseñada en el primer apartado era de dimensiones n x c, por tanto el orden de la complejidad espacial de este algoritmo también es del orden O(nc).

Caminos mínimos

Problema

Sea G = (V,A) un grafo ponderado positivamente y representado por su matriz de pesos P, se desea obtener, para cada pareja de vértices, el camino mínimo entre ambos.
Se consideran las siguientes restricciones iniciales (*reflejadas en la matriz de pesos adjunta*):
→ Un vértice siempre está de sí mismo a una distancia nula (0).
→ Si dos vértices no están conectados por una arista directa su distancia es infinita.
→ La distancia de una arista que conecta dos vértices será su correspondiente a la matriz de pesos.

¿Cómo resolverlo? Introducción teórica

Floyd

Este algoritmo resuelve el problema de encontrar caminos mínimos.
Recibe inicialmente la matriz de pesos (P) y sobre ella va calculando los valores de dichos caminos mínimos.
Tras cada iteración k, $P^{[k]}$ contiene los valores de los caminos mínimos que únicamente emplean los vértices $\{v_1, \ldots, v_k\}$:
→ $p^{[0]}_{ij} = p_{ij}$
→ $p^{[k]}_{ij} = \text{mín}\{p^{[k-1]}_{ij}, p^{[k-1]}_{ik} + p^{[k-1]}_{kj}\}, k > 0$

La iteración 0, es decir P0 sería la matriz inicial. Piensa que empezamos por el vértice 1.
Pues se comprueba con todas las parejas de vértice, si el vértice 1 mejora el camino.
Luego con el vértice 2, es decir, que después de terminar esa iteración, se ha comprobado el 1 y el 2.
Cuando vayamos por k, se habrán comprobado los vértices de 1 a k, de forma que cualquier camino mínimo que sólo emplee los vértices de 1 a k, ya está incluido (pero no sabremos que ese es mínimo hasta terminar el algoritmo entero.
Cuando termine entero, se tendrán los caminos mínimos entre cualquier par de vértices.
Pij[k] Representa el camino de i a j contemplando que se pase por k (es decir, se ha mirado qué es mejor, si el que ya estaba previamente calculado ($P^{[k-1]}_{ij}$) o un nuevo camino pasando por k como nodo intermedio).

Tabla de subproblemas resueltos

Deberíamos estructurar una tabla que recoja toda combinación de parejas de vértices posibles, por lo que una aproximación adecuada sería una tabla del orden O(|V|), siendo V el conjunto de vértices del grafo.
Esta tabla realmente ya existe, y corresponde a la matriz de pesos. Tal y como se introdujo teóricamente el funcionamiento del algoritmo de Floyd, este operará sobre la misma matriz de pesos (P) que es del mismo orden que la tabla de subproblemas resueltos a diseñar y refleja la misma información, sólo que modificada respecto a la búsqueda de caminos mínimos entre cada par de vértices.

Planteamiento de la función

$$f(k) = \begin{cases} p_{ij}^{[0]} = p_{ij} & k = 0 \\ p_{ij}^{[k]} = \text{mín}\{p_{ij}^{[k-1]}, p_{ik}^{[k-1]} + p_{kj}^{[k-1]}\}, & k > 0 \end{cases}$$

Diseño del algoritmo

```
Floyd(p, n) → p
    desde i ← 1 hasta n
        p[i, i] ← 0
    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                p[i, j] ← mín(p[i, j], p[i, k] + p[k, j])
```

Análisis

Se realizan dos bucles externos, del orden O(n)
Dentro del segundo bucle externo, se anidan dos bucles más, de n iteraciones cada uno.
Por tanto, simplificando y aplicando la regla del mayor:
t(n) ∈ O(n^3)

Matriz de adyacencia
La matriz de adyacencia, refleja en sí misma las conexiones directas que existen por cada par de vértices, señalando 1 si existe un camino directo entre ambos y 0 en caso contrario.

Matriz de accesibilidad
Resulta de la aplicación del algoritmo de Warshall sobre la matriz de adyacencia de un grafo.
Amplía la información de la matriz de adyacencia, señalando positivamente aquellos pares de vértices a los que se pueda llegar mediante algún camino intermedio pasando por el vértice k.

Warshall

Tabla de subproblemas resueltos

El algoritmo de Warshall recibe inicialmente la matriz de adyacencia del grafo y **sobre ella** va construyendo la matriz de accesibilidad.

Planteamiento de la función

$$a_{ij}^{[0]} = \begin{cases} \top, & i = j \\ a_{ij}, & i \neq j \end{cases}$$

$$a_{ij}^{[k]} = a_{ij}^{[k-1]} \vee (a_{ik}^{[k-1]} \wedge a_{kj}^{[k-1]}), \quad k > 0$$

Diseño del algoritmo

```
Warshall(a, n) → a
    desde i ← 1 hasta n
        a[i, i] ← ⊤
    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                a[i, j] ← a[i, j] ∨ (a[i, k] ∧ a[k, j])
```

Análisis

Temporal
Sea n = |V|:
→ Se realizan dos bucles externos, con n iteraciones cada uno (O(n)).
→ Dentro del segundo bucle externo, se anidan dos bucles consecutivamente, también con n iteraciones cada una.

Por tanto, calculando, simplificando y aplicando la regla del máximo:

$$t(n) \in O(n^3)$$

Espacial
Sea n = |V|:
La complejidad espacial vendrá dada por el número de vértices, ya que la TSP recoge todas las posibles combinaciones de pares de vértices.
Por tanto, la TSP es de dimensiones n x n y su orden $O(n^2)$.

TEMA 3: DIVIDE (➗) Y VENCERÁS (💪)

Ecuaciones de Recurrencia: Guía para resolución de problemas.

Paso 1:
Del caso general, despejar los t(n) y agruparlos a la izquierda, dejando el término independiente o el 0 (homogénea), al que llamaremos h(n), a la derecha.

Paso intermedio 1:
Si tenemos un término de t(n) que contenga una división (n/x), deberemos hacer un cambio de variable. De tal forma que, $n = x^k \Rightarrow k$, siendo x el divisor y k nuestra nueva variable.

Sustituiremos entonces, primeramente, todas las n por x^k.

Efectuamos todas las simplificaciones (que no resulten en una dificultad añadida para la resolución del problema).

Hacemos el cambio de variable de x^k a k [$t(n) \Rightarrow t(k)$]]
Solo en la parte de t(n). No realizar estos cambios en h(n)

Paso 2:
Si tu ecuación es homogénea, debes saltar este paso.
Ahora sacaremos una ecuación equivalente a h(n), donde
$X * n^m * S^n$ debe ser igual a h(n), es decir $\Rightarrow X * n^m * S^n = h(n)$

Deberemos entonces obtener unos valores para X, S, y m que hagan que se cumpla dicha igualdad.
Cuidado: Si trabajamos con k, especificaremos k en lugar de n.

Paso 3:
Sacaremos unas ecuaciones adicionales, c(x1) y c(x2)
c(x1) se obtiene de extraer los factores de t(n) y convertirlos en un polinomio.
Por ejemplo: si la ecuación es t(n) - 2t(n-1) - 5t(n-2) la ecuación c(x1) resultaría $x^2 - 2x - 5$

c(x2) se obtiene de la siguiente fórmula: $(x-S)^{m+1}$, cuyos términos fueron obtenidos en el paso 2.

Si la ecuación es homogénea, esta ecuación no podrá ser obtenida. Su impacto se verá en el paso 6.

Paso 4:
Obtenemos B, que resulta de obtener las raíces de las ecuaciones planteadas en el paso anterior.
R → Raíz
E → Ecuación [$c(x_i)$]
De forma que $B = \{R1E1^n, R2E1^n, R1E2^n, R2E2^n, ...\}$

Si una raíz es múltiple, se indicará tantas veces como su número sea, pero multiplicando por n el término anterior.
Por ejemplo: $(x-1)^3 \Rightarrow 1^n, 1^n n, 1^n n^2$
Cuidado: Si trabajamos con k, especificaremos k en lugar de n.

Paso 5:
Planteamos un primer acercamiento a t(n), obtenido al aplicar términos (de nuestra libre elección, aunque por comodidad se usan términos griegos)

$$t(n) = \alpha\ R1E1^n + \beta\ R2E1^n + \gamma\ R1E2^n + \lambda\ R2E2^n + \dots$$

Cuidado: Si trabajamos con k, especificaremos k en lugar de n.

Paso 6:
Consideraremos para este paso que la parte correspondiente a c(x2) es el valor de t(n) y sustituiremos en la ecuación obtenida en el paso 1.

Por ejemplo, si la ecuación obtenida en el paso 1 es t(n) - 2t(n-1) - 5t(n-2), nos quedaría:
γ R1E2n + λ R2E2n - 2(γ R1E2(n-1) + λ R2E2(n-1)) - 5(γ R1E2(n-2) + λ R2E2(n-2)),
y simplificamos.

Si la ecuación es homogénea, no podrás haber considerado nada ya que c(x2) no aplicaba.
En este caso, generalmente, dispondrás de algunas consideraciones (normalmente suele haber más de un caso base en estas situaciones) que te permitan obtener ecuaciones adicionales y plantear un sistema de ecuaciones.

Paso 7:
En este paso, podemos encontrarnos ante dos situaciones al simplificar:
→ Obtenemos una ecuación en función de una única incógnita
→ Saltamos al siguiente paso

→ Obtenemos una ecuación en función de más de una incógnita
→ Hacemos una distinción similar a la que se hacía al operar con números complejos.
→ Igualamos los términos dependientes de h(n) con h(n) y los términos independientes con 0.

Paso intermedio 2:
Si procedimos con el paso intermedio 1, deberemos deshacer el cambio de variable.
Basta con sustituir en las ecuaciones obtenidas en el apartado anterior con el cambio propuesto en el paso intermedio 1.
Donde figure x^k pondremos n.
Donde ponga k, pondremos $\log_x n$ (en t(k) directamente t(n))

Paso 8:
Aplicamos los valores que se establecen en el caso base y obtenemos de las ecuaciones anteriores los valores restantes de los términos griegos que establecimos en el paso 5.

Una vez obtenido todos los términos griegos, habremos terminado y nuestra ecuación de recurrencia resultaría de sustituir estos términos a los términos dependientes de n.

Comprobación de la ecuación de recurrencia:
Basta con sustituir en el t(n) original como en el t(n) obtenido [ecuación de recurrencia] y comprobar que para una misma n obtenemos el mismo resultado.

¿En qué consiste "Divide y vencerás"?
No es más que una técnica particular de **diseño recursivo**.

Esquema general
1. **Descomposición**: Se descompone el ejemplar a resolver en sub ejemplares del mismo problema.
2. **Resolución recursiva**: Se resuelve cada uno de los sub ejemplares recursivamente.
a. **Etapa de combinación** (Su aplicación dependerá del tipo de problema): Se obtiene la solución total a partir de cada una de las soluciones parciales de cada sub ejemplar.

Deberemos distinguir tres **variantes** de este esquema, en función de una serie de determinantes:
Si la **solución final es de menor tamaño que la entrada inicial**:
→ **Esquema de simplificación / reducción**: En este esquema no se produce la etapa de combinado. (Por ejemplo, buscar en un vector de 100 posiciones, el índice del elemento x).
Si la **solución final es del mismo tamaño que la entrada inicial**:
→ Se produce etapa de combinación
Si los **sub ejemplares** generados en cada llamada recursiva son (aprox.) de **igual tamaño**:
→ **Esquema de equilibrado**
De lo contrario:
→ **Esquema de partición**

Búsqueda binaria
Es un problema de **simplificación binaria**, que busca la posición de la primera aparición de un elemento x en un vector ordenado $[v_i, \ldots, v_j]$.
→ Si x está en [vi , . . . , vj], p será su posición.
→ Si no, p contendrá la primera posición donde x podría insertarse sin alterar el orden.

```
búsqueda-binaria(x, v, i, j) → p
    n ← j − i + 1
    si n = 1
        si x ≤ v[i]
            p ← i
        si no
            p ← i + 1
    si no
        k ← i − 1 + n div 2
        si x ≤ v[k]
            p ← búsqueda-binaria(x, v, i, k)
        si no
            p ← búsqueda-binaria(x, v, k + 1, j)
```

$$t(n) = \begin{cases} 1, & n = 1 \\ t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1, & n > 1 \wedge x \leq v_k \\ t\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1, & n > 1 \wedge x > v_k \end{cases}$$

La **operación patrón** de este algoritmo es la comparación de valor entre el elemento a buscar y el elemento en la posición k del array.

Potencia rápida

Se desea calcular la n-ésima potencia de un elemento a ∈ A.
El siguiente algoritmo funciona por **simplificación binaria** del exponente:

```
potencia(a, n) → r
    si n = 0
        r ← 1
    si no
        si n = 1
            r ← a
        si no
            si n mod 2 = 0
                r ← potencia(a · a, n div 2)
            si no
                r ← a · potencia(a, n − 1)
```

$$t(n) = \begin{cases} 0, & n \leq 1 \\ t\left(\frac{n}{2}\right) + 1, & n > 1 \wedge n \text{ es par} \\ t(n-1) + 1, & n > 1 \wedge n \text{ es impar} \end{cases}$$

$$t\left(\frac{n-1}{2}\right) + 2$$

Donde la **operación patrón** es la multiplicación de bases o base por potencia recursiva.

Mínimo y máximo

Este algoritmo funciona por **equilibrado binario**.
Calcula recursivamente el mínimo y el máximo de dos subvectores de tamaños similares y los combina para formar el resultado.

```
mín-máx(v, i, j) → (a, b)
    n ← j − i + 1
    si n = 1
        (a, b) ← (v[i], v[i])
    si no
        k ← i − 1 + n div 2
        (c, d) ← mín-máx(v, i, k)
        (e, f ) ← mín-máx(v, k + 1, j)
        (a, b) ← (mín(c, e), máx(d, f))
```

$$t(n) = \begin{cases} 0, & n = 1 \\ t\left(\lfloor \frac{n}{2} \rfloor\right) + t\left(\lceil \frac{n}{2} \rceil\right) + 2, & n > 1 \end{cases}$$

Donde la **operación patrón** es la comparación (dentro de mín y max).

Ordenación fusión

Este algoritmo funciona por **equilibrado binario**.
Descompone el vector en dos subvectores de tamaños similares, los ordena recursivamente y combina los resultados en un vector ordenado.

```
ordenación-fusión(v, i, j) → v
    n ← j − i + 1
    si n ≤ n0
        ordenación-inserción(v, i, j)
    si no
        k ← i − 1 + n div 2
        ordenación-fusión(v, i, k)
        ordenación-fusión(v, k + 1, j)
        fusión(v, i, k, j)
```

```
fusión(v, i, k, j) → v
    n ← j − i + 1
    (p, q) ← (i, k + 1)
    desde l ← 1 hasta n
        si p ≤ k ∧ (q > j ∨ v[p] ≤ v[q])
            (w[l], p) ← (v[p], p + 1)
        si no
            (w[l], q) ← (v[q], q + 1)
    desde l ← 1 hasta n
        v[i − 1 + l] ← w[l]
```

La fusión emplea **n comparaciones** entre elementos del vector en el peor caso y un espacio auxiliar de n elementos

Ordenación rápida

Este algoritmo funciona por **partición**.
Descompone el vector de forma que al ordenar recursivamente las partes el resultado quede ya ordenado.

```
ordenación-rápida(v, i, j) → v
    n ← j − i + 1
    si n ≤ n0
        ordenación-inserción(v, i, j)
    si no
        p ← pivote(v, i, j)
        ordenación-rápida(v, i, p − 1)
        ordenación-rápida(v, p + 1, j)

pivote(v, i, j) → (v, p)
    (p, x) ← (i, v[i])
    desde k ← i + 1 hasta j
        si v[k] ≤ x
            p ← p + 1
            v[p] ↔ v[k]
            v[i] ← v[p]
            v[p] ← x
```

La descomposición toma un elemento como pivote y reorganiza el subvector para que a la izquierda del pivote queden los menores o iguales, y a su derecha los mayores.
Se devuelve la posición final del pivote.
Esta versión pivota sobre el primer elemento, realizando n − 1 comparaciones entre elementos del vector, con n = j − i + 1.

TEMA 4: EXPLORACIÓN (🧙) EN GRAFOS

Ordenación topológica

Dado un grafo G = (V, A) dirigido y acíclico:
Un orden topológico es un **orden lineal (<)** definido sobre el **conjunto de vértices (V)** en el que se cumple la propiedad de los vértices **i < j** [si (i, j) ∈ A (conjunto de aristas)].

Para toda arista, el vértice origen precede al destino en el orden topológico.

El **algoritmo de ordenación topológica**, que representa una exploración exhaustiva de un grafo, **recibe G** y **devuelve** una **secuencia** con los elementos de V en orden **topológico**, a partir de una secuencia vacía y una **búsqueda en profundidad**.

El algoritmo recorrerá el vector (l) de *n* listas de adyacencias (cada vértice posee su propia lista de adyacencia). La secuencia se representa mediante un vector de n elementos (número de vértices) que se rellena en orden inverso.

```
ordenación-topológica(l, n) → v
        desde i ← 1 hasta n
                m[i] ← ⊥
        k ← n
        desde i ← 1 hasta n
                si ¬m[i]
                        profundidad(l, i, m, v, k)
          profundidad(l, i, m, v, k) → (m, v, k)
        m[i] ← ⊤
        para todo j ∈ l[i]
                si ¬m[j]
                        profundidad(l, j, m, v, k)
        v[k] ← i
        k ← k − 1
```

Resolución de un laberinto

→ Un laberinto puede representarse mediante una matriz bidimensional [m x n], l.
→ Originalmente, cada celda está libre (' ') o bloqueada ('X').
→ El objetivo es comprobar si desde una posición inicial (i, j) se puede llegar a una salida.
→ Dicha posición inicial debe ser válida y corresponder a una celda libre.
→ A este efecto, se considera que una salida es cualquier celda libre del borde.
→ A la matriz se puede asociar un grafo implícito (l) que podemos recorrer mediante una búsqueda en profundidad desde el vértice correspondiente a la posición inicial.
→ No es necesario construir dicho grafo, pues podemos trabajar directamente sobre la matriz mediante un marcado de sus celdas.
→ Durante la búsqueda de una salida se marcan las celdas visitadas ('V').
→ Si a partir de una celda visitada es posible encontrar una salida, se marca dicha celda como perteneciente a la solución ('S').

```
solución-laberinto(l, m, n, i, j) → (l,s) - i, j son los índices recursivos a pasar en cada llamada recursiva
    si l[i, j] = ' '
        si borde(m, n, i, j)
            s ← ⊤
        sí no
            l[i, j] ← 'V'
            s ←  solución-laberinto(l, m, n, i − 1, j) ∨
                 solución-laberinto(l, m, n, i + 1, j) ∨
                 solución-laberinto(l, m, n, i, j − 1) ∨
                 solución-laberinto(l, m, n, i, j + 1)
        si s
            l[i, j] ← 'S'
        sí no
            s ← ⊥

borde(m, n, i, j) → B
    si i ∈ {1, m} ∨ j ∈ {1, n}
        B ← T
    sí no
        B ← ⊥
```

Laberinto 3D

Mismo problema que el anterior, pero con una matriz tridimensional (se añade una coordenada z)

```
solución-laberinto(l, m, n, k, i, j, z) → (l,s)
    si l[i, j, z] = ' '
        si borde(m, n, k, i, j, z)
            s ← ⊤
        sí no
            l[i, j, z] ←'V'
            s ←  solución-laberinto(l, m, n, i − 1, j, z) ∨
                 solución-laberinto(l, m, n, i + 1, j, z) ∨
                 solución-laberinto(l, m, n, i, j − 1, z) ∨
                 solución-laberinto(l, m, n, i, j + 1, z) ∨
                 solución-laberinto(l, m, n, i, j, z - 1) ∨
                 solución-laberinto(l, m, n, i, j, z + 1)
        si s
            l[i, j, z] ←'S'
        sí no
            s ← ⊥
```

```
borde(m, n, k, i, j, z) → B
    si i ∈ {1, m} ∨ j ∈ {1, n} ∨ z ∈ {1, k}
        B ← T
    sí no
        B ← ⊥
```

Esquema general de la búsqueda con retroceso

Hay que tomar una decisión en cada nivel k del árbol de búsqueda.

El algoritmo comienza su ejecución con k = 1.

```
siguiente-nivel(s, k) → s
    prepara-nivel(k)
    mientras ¬último-hijo-nivel(k)
        s[k] ← siguiente-hijo-nivel(k)
        si solución(s, k)
            procesa(s, k)
        sí no
            si completable(s, k)
                siguiente-nivel(s, k + 1)
            - si no retroceso (no avanza al siguiente nivel)
```

Elementos

→ s: vector con las decisiones tomadas en los niveles 1, . . . , k.
→ solución(s, k): ¿es s una solución completa?
→ completable(s, k): ¿puede ser s parte de una solución completa?
→ procesa(s, k): procesa la solución encontrada

Coloreado de grafos: búsqueda con retroceso

```
coloreado-grafo(l, n, m, s, k) → s
desde c ← 1 hasta m
    s[k] ← c
    si coloreable(l, s, k)
      si k = n
        imprimir(s, k)
      si no
        coloreado-grafo(l, n, m, s, k + 1)
```

```
coloreable(l, s, k) → r
r ← ⊤
j ← 1
mientras j < k ∧ r
        si k ∈ l[j] ∨ j ∈ l[k]
          r ← s[j] ≠ s[k]
        j ← j + 1
```

Los vértices se seleccionan en orden decreciente de su grado.

Algoritmo de la mochila con pesos reales

→ En M se va guardando la mejor solución hasta el momento encontrada.
→ mdr es la función llamada que se encarga de llamar por 1ª a la función recursiva.

```
mdr(v, p, n, c) → S
desde i ← 1 hasta n
    S[i] ← ⊥
M ← S
S ← mdr-aux(v, p, n, c, S, 1, M)
```

```
mdr-aux(v, p, n, c, S, k, M) → M
si p[k] + peso(S) ≤ c
  S[k] ← ⊤
  si k = n
    si valor(S) > valor(M)
      M ← S
  si no
    M ← mdr-aux(v, p, n, c, S, k + 1, M)
S[k] ← ⊥
si k = n
  si valor(S) > valor(M)
    M ← S
si no
  M ← mdr-aux(v, p, n, c, S, k + 1, M)
```

TEMA 1: ALGORITMOS DEVORADORES

Un fontanero dispone de un conjunto de trozos de tuberías T de diferentes longitudes y diámetros.
Diseñe un algoritmo eficiente, basado en una estrategia devoradora, que minimice el número de uniones necesarias para construir una única tubería de longitud exacta l, teniendo en cuenta que: el fontanero puede realizar tantos cortes en las tuberías como estime oportuno; no pueden unirse tuberías de distinto diámetro; las tuberías de longitud par tienen un diámetro igual a 10 mm; las tuberías de longitud impar tienen un diámetro igual a 15 mm. Especifique, en caso de emplear alguna de esas funciones, si la ordenación se realiza de forma creciente o decrecientemente. Un algoritmo con un orden de complejidad temporal superior al necesario se considerará incorrecto. Analice la complejidad del algoritmo en el peor de los casos. Describa los elementos que lo identifican como perteneciente al esquema general de los algoritmos voraces.

Elementos:
→ Conjunto de candidatos (C): Tuberías (T)
→ Conjunto de candidatos (S): Tuberías seleccionadas para unión. Inicialmente vacío.
→ Función solución: ¿Se ha conseguido unir la longitud exacta?
→ Función selección: Se extrae la primera tubería del conjunto de candidatos, habiendo pre ordenado descendentemente dicho conjunto según la longitud.
→ Función factibilidad: ¿Puede unirse la tubería seleccionada sin exceder la longitud?
→ Función objetivo: El número de uniones total con las tuberías incluidas.
→ Objetivo: Minimizar.

Algoritmo

```
tuberias(T, l) → S
    C ← ordena-decreciente(T) - Según longitud en orden decreciente
    S ← Ø
    d = selecciona-primero(C).diametro
    mientras C ≠ Ø ∧ l > 0
        tuberia ← extrae-primero(C)
        si d = tuberia.diametro
            si l ≥ tuberia.longitud
                l ← l - tuberia.longitud
            si no
                tuberia.longitud ← l
                l ← 0
            inserta(tuberia, S)
```

Análisis
Siendo n = |T|:
→ El coste de la ordenación (por fusión, por ejemplo) es de O(n log n).
→ El bucle externo se ejecutará n veces
    → El coste de extracción es de orden constante (O(1))

Por tanto, calculando y por la regla del máximo:

$$t(n) \in O(n \log n) + O(n)$$

$$\mathbf{t(n) \in O(n \log n)}$$

Se dispone de un buque portacontenedores que hay que cargar con tantos contenedores disponibles en la terminal de carga como sea posible, todos de idénticas dimensiones, pero cada uno con su propio peso.
El buque podría cargar todos los contenedores de estar vacíos, pero al no ser el caso, su peso total podría exceder la capacidad de carga del buque, lo que no debe suceder.
Diseñe un algoritmo devorador para intentar resolver el problema. Mejore su eficiencia con pre-ordenación, y luego, con montículo.
a. Identificar los elementos del esquema
b. Diseñar el algoritmo
c. Analizar la eficiencia
d. Explicar cómo mejorarlo con las técnicas y analizar las mejoras

a) Elementos del esquema
Conjunto de Candidatos: Contenedores
Conjunto de Candidatos seleccionados: Contenedores cargados en el buque
Función selección: Menor peso
Función solución: ¿Se ha agotado la capacidad o no quedan más contenedores?
Función factibilidad: ¿Excede la capacidad de carga del buque?
Función objetivo: Número de contenedores
Objetivo: Maximizar

b) Diseño del Algoritmo

```
buque(T, c) → S
        S ← Ø
        C ← T
        mientras c ≠ 0 ∧ C ≠ ø
                contenedor ← extrae-contenedor(C)
                si contenedor ≤ c
                        c ← c - contenedor
                        inserta(contenedor, S)

extrae-contenedor(C) → contenedor
        contenedor ← Infinito
        para todo c e C
                si c ≤ contenedor
                        contenedor ← c
        elimina(contenedor, C)
```

c) Análisis
Siendo n = |T|, en el peor de los casos:
- El bucle externo se ejecutará n veces: O(n)
    - La función de selección se ejecutará n veces (buscará entre los n candidatos): O(n)

Por tanto, calculando y simplificando la complejidad, obtenemos que t(n) ∈ O(n^2)

d) Mejoras

Pre-ordenación:

```
buque(C, c) → S
    C ← ordena(C)
    S ← Ø
    mientras c ≠ 0 ∧ C ≠ ø
        contenedor ← extrae(C)
        si contenedor ≤ c
            inserta(contenedor, S)
            c ← c - contenedor
```

Siendo n = |T|, en el peor de los casos:
- Se pre-ordena el conjunto, usando un buen algoritmo de ordenación: O(n log n)
- El bucle externo se ejecutará n veces: O(n)
    - La función de selección extrae el primer candidato (el mejor): O(1)

Por tanto, calculando y simplificando la complejidad, obtenemos que t(n) ∈ O(n log n)

Montículo:

```
buque(C, c) → S
    C ← montículo(C)
    S ← Ø
    mientras c ≠ 0 ∧ C ≠ ø
        contenedor ← extrae-cima(C)
        si contenedor ≤ c
            inserta(contenedor, S)
            c ← c - contenedor
```

Siendo n = |T|, en el peor de los casos:
- Se pre-ordena el conjunto, volcandólo en una estructura de montículo: O(n log n)
- El bucle externo se ejecutará n veces: O(n)
    - La función de selección extrae el primer candidato (el mejor): O(1)

Por tanto, calculando y simplificando la complejidad, obtenemos que t(n) ∈ O(n log n)

Debido a la inflación, un ahorrador desea invertir todos sus ahorros en depósitos a plazo fijo anuales. Para ello analiza la oferta de este tipo de depósitos, anotando el capital máximo que cada banco permite invertir a cada cliente y el interés anual (TAE) que ofrece. Considere las estrategias, y aplíquelas a este ejemplo, 44.000 € a invertir en tres depósitos con, respectivamente 1%, 2%, 4% de TAE, y capitales máximos de 30.000 €, 36.000 € y 10.000 €. ¿Garantiza un algoritmo devorador que se maximice el beneficio de su inversión con algunas de estas estrategias?
Diseña un algoritmo devorador óptimo.
Rubrica:
a) Se plantean bien los ejemplos.
Se ha de plantear el beneficio máximo obtenido invirtiendo según las 3 estrategias posibles:
1. Invertir primero en el banco con mayor TAE.
2. Invertir primero en el banco con mayor capital máximo.
3. Invertir primero en el banco que tenga la mayor relación capital máximo/TAE.

b) Se explica (si existe) la opción óptima.
La solución óptima, aplicando varios ejemplos y realizando varias trazas, siempre se consigue invirtiendo primero en el banco con mayor TAE.

c) Se identifican los elementos del esquema devorador.
Elementos:
→ Conjunto de candidatos (C): Bancos (B)
→ Conjunto de candidatos seleccionados (S): Bancos invertidos
→ Función solución: ¿Se ha invertido todo el dinero?
→ Función selección: Se elige el banco con mayor TAE.
→ Función factibilidad: ¿Puede invertirse todo el dinero restante?
→ Función objetivo: Beneficio obtenido.
→ Objetivo: Maximizar.

d) Se diseña correctamente el Algoritmo.
```
inversion(B, e) → S
    C ← ordena-decreciente(B) - según TAE
    S ← Ø
        mientras e ≠ 0 ∧ C ≠ Ø
            b ← extrae-primero(C)
            si b.max_capital ≥ e
                e ← 0
            si no
                e ← e-b.max_capital
            insertar(b, S)
```

Diseñe un algoritmo que calcule un bosque de expansión de coste mínimo de un grafo ponderado. El algoritmo recibirá la matriz de pesos del grafo y devolverá un conjunto de árboles, cada uno de los cuales estará representado mediante un conjunto de aristas.
Criterios:
a) Se identifican correctamente todos los elementos del esquema general de los algoritmos voraces – 1
b) Además, se diseña un algoritmo que resuelve directamente el problema – 1.5

a) <u>Elementos</u>:
→ Conjunto de candidatos (C): Aristas
→ Conjunto de candidatos seleccionados (S): Aristas seleccionadas
→ Función solución: ¿Se ha obtenido el bosque de expansión mínima?
→ Función selección: Se elige la arista de menor peso.
→ Función factibilidad: ¿Están ya unidos?
→ Función objetivo: Beneficio obtenido.
→ Objetivo: Maximizar.

b) Algoritmo modificado

```
ConvertirMatriz(p, n) → A
        A ← ∅
        para i ← 1 hasta n
                para j ← i+1 hasta n
                        si p[i, j] ≠ ∞  - Existe una arista
                                inserta({i, j}, A)

KruskalBosqueConMatriz(p, n) → S
        C ← ConvertirMatrizAPares(p, n)
        S ← ∅
        p ← partición-inicial(n)
        ordena(C)
        mientras C ≠ ∅
                {i, j} ← extrae-primero(C)
                (e1, e2) ← (búsqueda(p, n, i), búsqueda(p, n, j))
                si e1 ≠ e2
                        unión(p, n, e1, e2)
                        inserta({i, j}, S)
```

Solución de Palomo:
Supongamos que pasamos un grafo que no sea conexo a la versión que hemos presentado del algoritmo de Prim, que fue diseñado para grafos conexos. El problema está en qué ocurre cuando el algoritmo de Prim no puede seleccionar más vértices (porque el resto está en otras componentes). La función que selecciona el vértice más prometedor es:

```
selecciona-vértice(C, d) → k
v ← ∞
para todo j ∈ C
        si d[j] < v
        v ← d[j]
        k ← j
```

Pero todos los vértices j de C tienen d[j] infinita. En tal caso, ¿qué k se devuelve? Puesto que no se asigna nada a k, su valor no está bien definido, por lo que el algoritmo sería erróneo.

Debemos devolver un k que nos permita identificar esta situación y terminar el bucle devorador del algoritmo en tal caso. Por ejemplo, basta dar a k un valor inicial que no pueda confundirse con el de ningún vértice (por ejemplo, 0, si los vértices se numeran de 1 a n).

```
selecciona-vértice(C, d) → k
v ← ∞
k ← 0
para todo j ∈ C
        si d[j] < v
        v ← d[j]
        k ← j
```

Ahora, cuando ya no queden vértices en dicha componente se devolverá 0.

A partir de aquí hay dos posibilidades:

Reiniciar la ejecución internamente para que empiece con otra componente.

Terminar la ejecución, con lo que tendremos un algoritmo que devolverá la componente conexa a la que pertenezca el vértice inicial. Podemos entonces modificar Prim para que comience en un vértice arbitrario (que recibiría como parámetro), en lugar del vértice 1. A partir de esta versión, que devuelve correctamente el árbol de expansión de coste mínimo asociado a un vértice arbitrario, es muy sencillo crear un algoritmo principal que defina un vector de marcas, vaya ejecutando Prim sobre el siguiente vértice no marcado, y coleccione los árboles generados para obtener el bosque de expansión de coste mínimo. Lo único extra que tendría que hacer Prim es marcar los vértices visitados.

Pero existe otra solución aún más simple:

```
selecciona-vértice(C, d) → k
v ← ∞
para todo j ∈ C
        si d[j] ≤ v
        v ← d[j]
        k ← j
```

Ahora, cuando se agote una componente conexa, se devolverá un k con d[k] = ∞, es decir, un k de otra componente, y el algoritmo continuará de manera natural.

Un fontanero dispone de un conjunto T de trozos de tuberías de diferentes longitudes. Diseñe un algoritmo eficiente, basado en una estrategia devoradora, que minimice el número de uniones necesarias para construir una única tubería de longitud exacta l. El fontanero no puede realizar ningún corte en las tuberías. Analice la complejidad del algoritmo en el peor de los casos. Describa los elementos que lo identifican como perteneciente al esquema general de los algoritmos voraces.
- Se identifican correctamente todos los elementos del esquema general de los algoritmos voraces. 0,50
- Además, se diseña un algoritmo eficiente que resuelve correctamente el problema. 1,50
- Además, se analiza correctamente la complejidad del algoritmo. 0,50

a) Elementos:
→ Conjunto de candidatos (C): Tuberías (T)
→ Conjunto de candidatos (S): Tuberías seleccionadas para unión. Inicialmente vacío.
→ Función solución: ¿Se ha conseguido unir la longitud exacta?
→ Función selección: Se extrae la primera tubería del conjunto de candidatos, habiendo-pre ordenado descendentemente dicho conjunto según la longitud.
→ Función factibilidad: ¿Puede unirse la tubería seleccionada sin exceder la longitud?
→ Función objetivo: El número de uniones total con las tuberías incluidas.
→ Objetivo: Minimizar.

b) Algoritmo

```
tuberias(T, l) → S
    C ← ordena-decreciente(T)
    S ← ∅
    mientras l ≠ 0 ∧ C ≠ ∅
        t ← extrae-primero(C)
            si l ≥ t
                l ← l - t
                inserta(t, S)
```

c) Análisis:
Siendo n = |T|:
→ El coste de la ordenación (por fusión, por ejemplo) es de O(n log n).
→ El bucle externo se ejecutará n veces
    → El coste de extracción es de orden constante (O(1))

Por tanto, calculando y por la regla del máximo:

$$t(n) \in O(n \log n) + O(n)$$

$$\mathbf{t(n) \in O(n \log n)}$$

Realice un análisis general del tiempo que emplea el algoritmo de Kruskal en el peor caso. Deberá expresar el tiempo en función del número de vértices y de aristas del grafo, así como de los tiempos correspondientes a las operaciones de la estructura de partición. Posteriormente:
a) Calcule el orden asintótico que se obtiene cuando la estructura de partición se implementa mediante un bosque sin control de alturas.
b) Calcule el impacto de la densidad del grafo en dicho orden.
c) Compare estos resultados con los que se obtienen para el algoritmo de Prim.

Algoritmo:

```
Kruskal(V, A) → S
    C ← ordena(A)
    S ← ∅
    n ← |V|
    p ← particion-inicial(n)
    mientras S ≠ n-1
        a ← extrae-primero(C)
        (e1,e2) ← (busqueda(p, n, a.i), busqueda(p ,n, a.j))
        si e1 ≠ e2
            union(e1, e2, p, n)
            inserta({a.i, a.j}, S)
```

SIN control de alturas vs. CON control de alturas

En las particiones, se emplean árboles como estructuras subyacentes a dicha estructura de datos.
En un grafo representado mediante una estructura de partición con control de alturas implica que el árbol subyacente crece de forma equilibrada. En cambio, en una estructura de partición sin control de alturas, este árbol puede crecer de forma arbitraria, pudiéndose deformar en el peor de los casos en una sola rama (vector.
Esto, a la hora de realizar una búsqueda, tiene un gran impacto:
→ Con control de alturas, se garantiza una búsqueda de O(log n)
→ Sin control de alturas, en el peor de los casos la búsqueda será de orden lineal O(n)

Siendo n = |V| y a = |A|
El algoritmo de Kruskal realiza las siguientes operaciones:
- Se crea una partición → O(n)
- Se ordenan las aristas → O(a log a)
- El bucle se ejecuta en el peor caso 'a' veces → O(a)
    - Se ejecutan 2 búsquedas → 2O(n)
    - Se efectúan n-1 uniones → t(n - 1) → O(n)

t(n,a) = O(n) + O(a log a) + (O(a) * (2O(n) + O(n)))
t(n,a) = O(n) + O(a log a) + 3O(an)

Debemos entonces obtener una relación entre el número de vértices y el número de aristas:
- Para grafos dispersos (árboles) tendremos como mínimo n-1 aristas.
- Para grafos completos (cliques) existirán n(n-1)/2 aristas.
Por tanto:

$$n-1 \leq a \leq n(n-1)/2$$

Obtenemos entonces la complejidad temporal en función de la densidad del grafo:
1. Árbol (a = n-1) → t(n) = O(n) + O(n-1 log n-1) + 3O(n-1 * n)
Aplicando la eliminación de constantes y la regla del máximo obtenemos que → t(n) e $O(n^2)$

2. Clique (a = n(n-1)/2) → t(n) = O(n) + O(n(n-1)/2 log n(n-1)/2) + 3O((n(n-1)/2) * n)
Aplicando la eliminación de constantes y la regla del máximo obtenemos que → t(n) e $O(n^3)$

Una empresa de telecomunicaciones dispone del presupuesto necesario para desplegar X antenas en una ciudad.
La empresa ha dividido la ciudad en NxN celdas y ha estimado el número medio de habitantes en cada una de esas celdas.
Cada antena colocada en la celda c = <i, j> da cobertura a los habitantes que hay a una distancia d.
Suponga que dispone de una función dist(a, b) de complejidad temporal de orden constante, que devuelve la distancia entre las celdas a y b.

a) Diseñe un algoritmo que, siguiendo un enfoque voraz, determine la mejor ubicación de las antenas para conseguir dar servicio al mayor número posible de clientes. Implemente una función de factibilidad explícita.

```
cobertura-antena: T × i × j × d → r
        r ← T[i, j]
        para todo <l, k> ∈ T
                distancia ← dist (<i, j>, <l, k>)
                si distancia ≤ d
                        r ← r + T[l, k]

selecciona - celda: C × d → <i, j>
        v ← −∞
        para todo <l, k> ∈ C
                cob ← cobertura-antena(C, l, k, d)
                si cob > v
                        v ← cob
                        <i, j> ← <l, k>

factible: T × i × j × d → b
        b ← cobertura-antena(T, i, j, d) > 0

antenas: T × n × d × x → S
        C ← T
        cantAntenas ← x
        mientras T = Ø ∧ cantAntenas > 0
                <i, j> ← selecciona-celda(T, d)
                si factible(T, i, j, d)
                        cantAntenas ← cantAntenas − 1
                        S ← S ∪ {<i, j>}
                T ← T − {<i, j>}
```

b) Describa los elementos que lo identifican como perteneciente al esquema de los algoritmos voraces.
Los elementos que lo identifican como perteneciente al esquema de los algoritmos voraces son:
• Un conjunto de candidatos seleccionados, las celdas.
• Una función de selección, que escoge la celda en la que una antena da mayor cobertura.
• Una función de factibilidad, que indica si una antena tiene cobertura o no.
• Una función solución: ¿hemos gastado todas las antenas?
• Una función objetivo: la cantidad de clientes cubiertos.
• Un objetivo: maximizar.

c) Analice su complejidad temporal.

En el peor caso, que es en el que agotáramos las celdas, tendríamos:

• La función de cobertura tiene un orden que depende de O (|T|).

• La función de selección tiene un orden que depende de $O(|T|^2)$

• La función de factibilidad tiene un orden que depende de O (|T|).

Ya que tiene dentro la función de factibilidad más la función de selección que depende del bucle exterior.

Esto es:

$$t(n) \in O(n^3)$$

TEMA 2: PROGRAMACIÓN DINÁMICA

Sea G = (V, A) un grafo conexo y ponderado compuesto por V vértices y A aristas. Cada arista tiene asignada una distancia d, y un color c. Desarrolle un algoritmo que obtenga la distancia más corta entre cada par de nodos del grafo sin emplear aristas de un determinado color c, que será pasado como argumento al algoritmo. Especifique claramente la estructura de la tabla de subproblemas resueltos, incluyendo los tipos de datos que almacenará, y analice la complejidad espacial y temporal del algoritmo en el peor de los casos.

Tabla de subproblemas resueltos:
Floyd trabaja con la matriz de pesos asociados al grafo G.
Deberíamos diseñar una tabla de subproblemas resueltos del orden $O(|V|^2)$, para contener a todos los vértices integrantes del grafo.
En vez de diseñar explícitamente una tabla adicional para este cometido, trabajaremos sobre la propia matriz de pesos asociada, que resulta del mismo orden y contiene los mismos elementos.

Algoritmo

```
FloydColor(p, n, c) → p
    desde i ← 1 hasta n
        p[i,i].p ← 0

    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                si p[i][j].c = c
                    p[i][j].p ← ∞
                p[i, j].p ← mín(p[i, j].p, p[i, k].p + p[k, j].p)
```

Análisis
Sea n = |V|:
La complejidad temporal es de $\Theta(n^3)$ operaciones elementales.
La complejidad espacial resulta del orden $O(n^2)$ [explicado en el apartado 1].

Suponga que dispone de información acerca de qué usuarios siguen a otros en una red social. Los bulos se propagan a través de cadenas de seguidores. Diseña un algoritmo que determine cuantos usuarios bastan para propagar un bulo entre los usuarios conectados por la cadena de seguidores más larga. Analice su eficiencia espacio temporal en el peor de los casos.
Rubrica:
a) Se explica la tabla de subproblemas resueltos.
La TSP deberá contemplar toda combinación de usuarios posible, por tanto, deberá ser de dimensiones n*n [$O(n^2)$].
En esta TSP, inicialmente figurará 1 en aquellos índices donde exista una relación de seguimiento, e infinito de lo contrario.
Dicha TSP será construida a partir de la matriz de adyacencia proporcionada por parámetro al algoritmo.

b) Se diseña un algoritmo que soluciona el problema.
A: Matriz de adyacencia donde figura las relaciones directas entre seguidores.

```
Bulo(A, n) → c
    desde i ← 1 hasta n
        desde j ← 1 hasta n
            si A[i][j]
                p[i][j] = 1
            si no
                p[i][j] = Infinito

    desde i ← 1 hasta n
        p[i][i] ← 0

    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                p[i][j] ← mín(p[i][j],  p[i,k] + p[k,j])

    c ← - Infinito
    desde i ← 1 hasta n
        desde j ← 1 hasta n
            c ← máx(c, p[i][j])
```

c) Además, se analiza su eficiencia temporal.
Primer bucle externo: $O(n^2)$
Segundo bucle externo: $O(n)$
Tercer bucle externo: $O(n^3)$
Cuarto bucle externo: $O(n^2)$
Por tanto, operando y aplicando la regla del máximo, obtenemos que $t(n) \in O(n^3)$

d) Además, se analiza su eficiencia espacial.
Como introducimos en el primer apartado, la TSP posee dimensiones de n*n, siendo su eficiencia espacial del orden $O(n^2)$

Un responsable de movilidad necesita conocer si es posible viajar en vehículo entre todas las intersecciones de las calles que hay en una determinada ciudad. En esa ciudad todas las calles tienen un único sentido de circulación, y entre cada par de intersecciones solo discurre una única calle.
Desarrolla un algoritmo que obtenga la respuesta que busca el responsable (si o no). Específica claramente, la estructura de la tabla del subproblema resuelto, incluyendo los tipos de datos que almacenará, y analiza la complejidad espacial y temporal del algoritmo en el peor de los casos
<u>Criterios</u>:
• Se diseña correctamente la tabla de subproblemas resueltos – 0.5
• Además, se diseña correctamente el algoritmo – 1
• Además, se analiza correctamente la complejidad temporal del algoritmo – 0.5
• Además, se analiza correctamente la complejidad espacial del algoritmo – 0.5

a) Tabla de subproblemas resueltos.
La TSP deberá contener toda combinación de intersecciones posible, por tanto, deberá ser de dimensiones n*n [$O(n^2)$].
En esta TSP, inicialmente figurará true (T) en aquellos índices donde exista una intersección que conecte dos calles (i,j), y falso (F) de lo contrario.
Dicha TSP será proporcionada por parámetro al algoritmo.

b) Algoritmo.

```
intersecciones(A, n) → b
    desde i ← 1 hasta n
        A[i][i] ← T
    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                A[i][j] ← (A[i][j] ∨ A[i][k] ∧ A[k][j])

    b ← T
    i ← 1
    mientras b = T ∧ i ≠ n
        j ← 1
        mientras b = T ∧ j ≠ n
            b ← A[i][j]
            j ← j + 1
        i ← i + 1
```

c) Complejidad temporal.
En el peor de los casos obtenemos que:
Primer bucle externo: O(n)
Segundo bucle externo: $O(n^3)$
Tercer bucle externo: $O(n^2)$
Por tanto, operando y aplicando la regla del máximo, obtenemos que $t(n) \in O(n^3)$.

d) Complejidad espacial.
Como introducimos en el primer apartado, la TSP posee dimensiones de n*n, siendo su complejidad espacial del orden $O(n^2)$.

Una empresa de traducción dispone de D diccionarios. Cada diccionario <i, j> ∈ D permite hacer la traducción entre dos idiomas diferentes i y j. Diseñe un algoritmo basado en programación dinámica que determine si es posible llevar a cabo la traducción de un idioma a otro cualquiera en base a los diccionarios de los que dispone la empresa. El algoritmo debe devolver también el número mínimo de diccionarios que hacen falta para hacer la traducción. Describa la estructura de la tabla de subproblemas resueltos. Indique el estado final de esta suponiendo que la empresa dispone de los siguientes diccionarios: SPA-FRA, SPA-ENG, SPA-GER, ENG-FRA y ENG-GER.
Analice la complejidad espacial y temporal del algoritmo en el peor de los casos.

a) Tabla de subproblemas resueltos.
En este problema, existen dos TSP idénticas, sobre las que se operará simultáneamente. Estas TSP resultan ser las matrices de adyacencia y de pesos sobre las que trabajará nuestro algoritmo basado en la implementación de Floyd.
Las TSP deberán contener toda combinación de idiomas posible, por tanto, deberán ser de dimensiones n*n [$O(n^2)$].
En estas TSP, inicialmente figurarán true (T) [adyacencia] o 1 [pesos] en aquellos índices donde exista un diccionario que traduzca entre ambos idiomas (i,j), e infinito [pesos] o falso (F) de lo contrario.
Dichas TSP serán construidas a partir del conjunto de diccionarios D (cada diccionario siendo un par de vértices) proporcionado por parámetro al algoritmo.

b) Algoritmo.

```
traducción(D) → a, p
    n ← -1
    para todo d e D
        p[d.i][d.j] ← 1
        a[d.i][d.j] ← T
        n ← máx(n, máx(i, j)) - Número de idiomas que figuran en cualquier problema genérico

    desde i ← 1 hasta n
        p[i][i] ← 0
        a[i][i] ← T

    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                p[i][j] ← mín(p[i][j], p[i][k] + p[k][j])
                a[i][j] ← (p[i][j] v (p[i][k] ^ p[k][j]))
```

c) Complejidad temporal.
En el peor de los casos obtenemos que:
Primer bucle externo: O(|D|)
Segundo bucle externo: $O(n^3)$
Tercer bucle externo: $O(n^2)$
Por tanto, operando y aplicando la regla del máximo, obtenemos que $t(n) \in O(n^3)$.

d) Complejidad espacial.
Como introducimos en el primer apartado, la TSP posee dimensiones de n*n, siendo su complejidad espacial del orden $O(n^2)$.

Se dispone de un tablero, T, de dimensión mxn que tiene asociado un valor numérico t_{ij} con cada casilla (i,j). Deseamos mover un robot situado en la casilla (1,1) al extremo opuesto (m,n) de manera que la suma de los valores de las casillas que atraviesa sea mínima. En cada paso, el robot únicamente puede moverse a la casilla de la derecha o a la de abajo.

a) Resuelva el problema restringido consistente en calcular los valores de los caminos óptimos empleando programación dinámica. Detalle el proceso de obtención del algoritmo y no emplee más espacio que el del propio tablero.

```
costesMinimosRobot : T × m × n → T
    // Calculamos la primera columna moviéndonos siempre abajo.
    desde i ← 2 hasta m
        T[i, 1] ← T[i, 1] + T[i − 1, 1]
    // Calculamos la primera fila moviéndonos siempre a la derecha.
    desde j ← 2 hasta n
        T[1, j] ← T[1, j] + T[1, j − 1]

    // Calculamos el coste mínimo a (i, j) entre ir desde arriba o desde abajo.
    desde i ← 2 hasta m
        desde j ← 2 hasta n
            T[i][j] ← T[i, j] + mín (T[i − 1, j], T[i, j − 1])
```

b) Diseñe un algoritmo que a partir del resultado del anterior construya una matriz binaria con 1 en las posiciones que forman parte de algún camino óptimo y 0 en las demás.

```
caminoMinimo: T × m × n → A
    desde i ← 1 hasta m
        desde j ← 1 hasta n
            A[i, j] ← 0
            A[m, n] ← 1

    // Miramos del revés los valores mínimos, sin la columna y fila primeras.
    desde i ← m hasta 2
        desde j ← n hasta 2
            si A[i, j] = 1
                si T[i − 1, j] < T[i, j − 1]
                    A[i − 1, j] ← 1
                si_no
                    A[i, j − 1] ← 1

    // Miramos la primera fila .
    desde j ← n hasta 2
        si A[1, j] = 1
            A[1, j − 1] ← 1

    // Miramos la primera columna .
    desde i ← m hasta 2
        si A[i, 1] = 1
            A[i − 1, 1] ← 1
```

Para modelar una red de comunicaciones utilizamos un grafo G = <V, A> orientado y ponderado, donde V es el conjunto de los vértices, A ⊆ V ×V el conjunto de las aristas, p: V×V → [0, 1] la función de ponderación y n = |V|.
Los vértices representan equipos de comunicaciones y las aristas las líneas de comunicaciones que unen los equipos. La función de ponderación es la fiabilidad de la línea que une dos equipos, es decir, la probabilidad de que la transmisión de una unidad de información desde un equipo a otro tenga éxito. Estas probabilidades se suponen independientes.
Diseñe un algoritmo que calcule las fiabilidades máximas que se pueden conseguir en las transmisiones entre parejas de equipos cualesquiera. No emplee más espacio del estrictamente necesario, teniendo en cuenta que puede modificar la propia matriz de p, que recibirá como parámetro.

```
redComunicaciones(p,n) → p
    desde i ← 1 hasta n
        p[i][i] ← 1 - Se supone fiabilidad máxima al comunicar un equipo consigo mismo.

    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                p[i][j] ← máx(p[i][j], p[i][k] * p[k][j])
```

Una empresa dispone de una serie de centros logísticos, una flota de vehículos permite transportar paquetes entre ellos, para lo que la empresa ha establecido un mapa con las rutas que los conectan.
Al realizar cualquier ruta entre dos centros logísticos, el vehículo asignado debe parar en todos los centros intermedios situados en ella, de forma que pueda procederse a la carga y descarga de los paquetes necesarios.
Diseñe un algoritmo que determine cuántas paradas ha de realizar como MÍNIMO un vehículo en ruta entre los dos centros más alejados de la red antes de llegar a su destino.

a. Diseñar tabla de subproblemas resueltos.
La TSP deberá contener toda combinación de centros logísticos posible, por tanto, deberá ser de dimensiones n*n [$O(n^2)$].
En esta TSP, inicialmente figurará 1 en aquellos índices donde exista una ruta que conecte dos centros logísticos(i,j), e infinito de lo contrario.
Dicha TSP será construida a partir de la matriz de adyacencia (que indica si existe una ruta o no entre dos centros logísticos) proporcionada por parámetro al algoritmo.

b. Diseñar algoritmo.

```
centrosLogisticos(A, n) → p
    desde i ← 1 hasta n
        desde j ← 1 hasta n
            si A[i,j]
                M[i,j] ← 1
            si no
                M[i,j] ← Infinito

    desde i ← 1 hasta n
        M[i,j] ← 0

    desde k ← 1 hasta n
        desde i ← 1 hasta n
            desde j ← 1 hasta n
                M[i,j]  ← mín(M[i,j], M[i,k] + M[k,j])
    p ← - Infinito
    desde i ← 1 hasta n
        desde j ← 1 hasta n
            p ← máx(p, M[i][j])
```

c. Analizar eficiencia temporal y espacial del algoritmo.

Eficiencia temporal.
Primer bucle externo: $O(n^2)$
Segundo bucle externo: $O(n)$
Tercer bucle externo: $O(n^3)$
Cuarto bucle externo: $O(n^2)$
Por tanto, operando y aplicando la regla del máximo, obtenemos que $t(n) \in O(n^3)$

Eficiencia espacial.
Como introducimos en el primer apartado, la TSP posee dimensiones de n*n, siendo su eficiencia espacial del orden $O(n^2)$

TEMA 3: DIVIDE Y VENCERÁS

Ejercicio 3. Sea v = [v1, ···, vn] ∈ $\mathbb{Z}$+ tal que v1 < ··· < vh. Diseñe un algoritmo de coste logarítmico que permita encontrar un índice p ∈ [1, N] tal que vp = p, si existe un índice tal, o bien devuelva 0 en caso de que no exista

a. Identificar elementos del esquema.

Etapas:

1. Descomposición: En cada iteración el ejemplar se divide en dos subejemplares de igual tamaño.
2. Resolución recursiva: Tras cada descomposición, se procesa el subejemplar apto para su resolución.
3. Combinación: No aplicable a este algoritmo. El subejemplar no apto no se procesa (se descarta).

Variantes:

Simplificación: El subejemplar no válido se descarta.
Equilibrado: Los tamaños de los dos (binario) subejemplares son de igual tamaño.

Este algoritmo funciona por simplificación binaria.

b. Diseñar el algoritmo correctamente con su eficiencia.

```
busquedaBinariaIndice(v, i, j) → p
    n ← j - i + 1
    si n = 1
        si v[i] = i
            p ← i
        si no
            p ← 0
    si no
        k ← i - 1 + n div 2
        si v[k] = k
            p ← k
        si no
            si v[k] > k
                p ← busquedaBinariaIndice(v, i, k-1)
            si no
                p ← busquedaBinariaIndice(v, k + 1, j)
```

c. Plantear ecuación de recurrencia y resolverla.

Para ello, deberemos establecer la operación patrón, que es aquella que más veces se repite y que es la esencia del algoritmo. En este caso, resulta la comparación del elemento central.

$$t(n) = \begin{cases} 1, & \text{si } n = 1 \\ 1, & \text{si } n > 1 \wedge v[k] = k \\ t(n/2) + 1, & \text{si } n > 1 \wedge v[k] \neq k \end{cases}$$

Dadas la matriz identidad binaria I, y una matriz binaria A, cuadradas de dimensión n, se define su cláusula reflexiva y transitiva A* = I + A + A · A + A · A · A + ... donde + y · son operaciones matriciales
binarias: la suma y productos matriciales, pero son ∨ y ∧ para los elementos de las matrices. Diseñe un algoritmo, siguiendo el enfoque divide y vencerás, para calcular A*. Analice su eficiencia temporal y compárala con la del algoritmo de Warshall.
Rubrica:
a) Se identifican los elementos del esquema de Divide y vencerás.
Etapas:
1. Descomposición: En cada iteración, el ejemplar se reduce en un subejemplar de una unidad menor.
2. Resolución recursiva: Tras la descomposición, se resuelve recursivamente el subejemplar resultante.
3. Combinación: Tras calcular recursivamente cada subejemplar, se suman los resultados parciales.

Variantes:
Equilibrado: Se observa que los subejemplares generados son de aproximadamente el mismo tamaño.

b) Se diseña un algoritmo de forma correcta.

```
PotenciaRapidaMatriz(A, M, I, k) → M
    Si k = 0
        M ← M + I
    Si no
        Si k = 1
            M ← M + PotMatriz(A, M, I, k - 1)
        Si no
            M ← M + A * PotMatriz(A, M, I, k - 1)
```

c) Se plantea correctamente su ecuación de recurrencia.
Para ello, deberemos establecer la operación patrón, que es aquella que más veces se repite y que es la esencia del algoritmo. En este caso, resulta la suma.

$$t(n) = \begin{cases} 1, & \text{si } n = 0 \\ t(n-1) + 1, & \text{si } n > 1 \end{cases}$$

d) Además, se resuelve de forma correcta.

El siguiente algoritmo permite encontrar la posición de la primera aparición de un elemento de un vector ordenado. Analiza el algoritmo y determina el número total de asignaciones (←) que se realizan en el peor de los casos.

```
búsqueda_binaria: x x V x i x j → p
    n ← j-i+1
    si n = 1
        si x ≤ v[i]
            p ← i
        si no
            p ← i+1
    si no
        k ← i-1 + ndiv2
            si x=v[k]
                p ← k
            si no si x < v[k]
                p ← busqueda_binaria(x,V,i,k)
            si no
                p ← busqueda_binaria(x,V,k+1,j)
```

• Se identifican los elementos del esquema "divide y vencerás" y el subesquema

Etapas:

1. Descomposición: En cada iteración el ejemplar se divide en dos subejemplares de igual tamaño.
2. Resolución recursiva: Tras cada descomposición, se procesa el subejemplar apto para su resolución.
3. Combinación: No aplicable a este algoritmo. El subejemplar no apto no se procesa (se descarta).

Variantes:

Simplificación: El subejemplar no válido se descarta.
Equilibrado: Los tamaños de los dos (binario) subejemplares son de igual tamaño.

Este algoritmo funciona por simplificación binaria.

• Se plantea correctamente la ecuación de recurrencia

$$t(n) = \begin{cases} 2, & \text{si } n = 1 \\ 3, & \text{si } n > 1 \wedge v[k] = x \\ t(n/2) + 3, & \text{si } n > 1 \wedge v[k] \neq x \end{cases}$$

Los siguientes algoritmos determinan si un vector se encuentra ordenado:

```
ordenado1: v × i × j → r
    r ← T
    n ← j − i + 1
    si n > 1
        k ← i − 1 + n div 2
        r ← ordenado1(v, i, k) ∧ ordenado1(v, k + 1, j) ∧ v[k] ≤ v[k + 1]

ordenado2 :v × i × j → r
    r ← T
    w ← v
    ordenación−rápida(w)
    desde k ← i hasta j
        r ← r ∧ v[k] = w[k]
```

a) Determine su pertenencia al esquema de divide y vencerás indicando los elementos que los determinan como perteneciente a dicho esquema. En ese caso ¿Con qué subesquema de este se identifican?

- Algoritmo 1 -

Etapas:
Descomposición: En cada iteración, se descompone a la mitad el ejemplar.
Resolución recursiva: En cada iteración, se ordena recursivamente cada uno de los dos ejemplares.
Combinación: Se fusionan los resultados parciales (subvectores ordenados) al finalizar el algoritmo.
Variantes:
Equilibrado: Binario. Se descomponen en dos vectores de igual tamaño recursivamente.

- Algoritmo 2 -

No pertenece al esquema divide y vencerás.

b) Analice la eficiencia temporal de ambos algoritmos. Para los pertenecientes al esquema de divide y vencerás, especifique claramente que operación elemental tomara como patrón y suponga que n es potencia de dos.

- Algoritmo 1 -

Al ser un algoritmo recursivo, su eficiencia temporal deberá ser determinada a través de una ecuación de recurrencia. Debemos plantear una función t(n) que refleje los distintos casos. Para ello, deberemos establecer la operación patrón, que es aquella que más veces se repite y que es la esencia del algoritmo. En este caso, resulta la comparación con el elemento.

$$t(n) = \begin{cases} 0, & \text{si } n \leq 1 \\ 2t(n/2) + 1, & \text{si } n > 1 \end{cases}$$

Por el teorema maestro, podemos obtener la eficiencia temporal en el peor caso.
Siendo a = 2, b = 2 y k = 0, obtenemos que $t(n) \in O(\log n)$

- Algoritmo 2 -

Al no resultar un algoritmo recursivo, podemos analizarlo de manera normal.
En este caso, por la regla del máximo, obtenemos que el orden de complejidad temporal es del orden de O(n log n), que resulta de aplicar el algoritmo de ordenación-rápida al vector w.

El siguiente algoritmo suma y obtiene la diferencia de los elementos de un vector:

```
suma - resta: v × i × j → s × r
    n ← j − i + 1
    si n = 1
        k ← v[i]
    si_no
        k ← i − 1 + n div 2
        d ← i + j − k
        s ← suma − resta (v, i, k) + suma − resta (v, d, j)
        r ← suma − resta (v, i, k) − suma − resta (v, d, j)
```

a) Identifique los elementos que lo determinan como perteneciente al esquema, divide y vencerás ¿Con qué subesquema de este se identifica?

Etapas:

1. Descomposición: En cada iteración el ejemplar se divide en dos subejemplares de igual tamaño.
2. Resolución recursiva: Tras cada descomposición, se procesa el subejemplar apto para su resolución.
3. Combinación: No aplicable a este algoritmo. El subejemplar no apto no se procesa (se descarta).

Variantes:

Simplificación: El subejemplar no válido se descarta.
Equilibrado: Los tamaños de los dos (binario) subejemplares son de igual tamaño.

Este algoritmo funciona por simplificación binaria.

b) Obtenga el número de operaciones de suma (+) que se realizan en el algoritmo en función del tamaño del vector, que puede suponer siempre potencia de dos.

$$t(n) = \begin{cases} 1, & \text{si } n = 1 \\ 2t(n/2) + 4, & \text{si } n \neq 1 \end{cases}$$

El siguiente algoritmo permite encontrar la posición de la primera aparición de un elemento en un vector ordenado. Suponiendo que siempre existe el elemento buscado (x) en el vector, analice el algoritmo y determine el número total de operaciones de suma (+) que se realizan en total en el peor de los casos.

$búsqueda\text{-}binaria : x \times v \times i \times j \to p$
$n \leftarrow j - i + 1$
si $n = 1$
  si $x \leq v[i]$
    $p \leftarrow i$
  si no
    $p \leftarrow i + 1$
si no
  $k \leftarrow i - 1 + n \operatorname{div} 2$
  si $x = v[k]$
    $p \leftarrow k$
  si-no si $x < v[k]$
      $p \leftarrow búsqueda\text{-}binaria(x, v, i, k)$
  si no
      $p \leftarrow búsqueda\text{-}binaria(x, v, k + 1, j)$

Se identifican correctamente los elementos del esquema "divide y vencerás" y el subesquema.
Etapas:
1. Descomposición: En cada iteración el ejemplar se divide en dos subejemplares de igual tamaño.
2. Resolución recursiva: Tras cada descomposición, se procesa el subejemplar apto para su resolución.
3. Combinación: No aplicable a este algoritmo. El subejemplar no apto no se procesa (se descarta).

Variantes:
Simplificación: El subejemplar no válido se descarta.
Equilibrado: Los tamaños de los dos (binario) subejemplares son de igual tamaño.

Este algoritmo funciona por simplificación binaria.

Se plantea correctamente la ecuación de recurrencia

$$t(n) = \begin{cases} 1, & \text{si } n = 1 \land x \leq v[i] \\ 2, & \text{si } n = 1 \land x > v[i] \\ 2, & \text{si } n \neq 1 \land x = v[i] \\ t(n/2) + 2, & \text{si } n \neq 1 \land x < v[i] \\ t(n/2) + 3, & \text{si } n \neq 1 \land x \geq v[i] \end{cases}$$

TEMA 4: EXPLORACIÓN EN GRAFOS

Diseñe un algoritmo que calcule las componentes conexas de un grafo orientado acíclico simultáneamente a su ordenación topológica.
Analice su eficiencia temporal.

a) Se diseña un algoritmo que resuelve el problema.

```
ordenación-topológica-con-componentes(l, n) → (v, componentes)
    desde i ← 1 hasta n
        m[i] ← ⊥
    k ← n
    componentes ← lista vacía
    desde i ← 1 hasta n
        si ¬m[i]
            actualComponente ← ø
            profundidad(l, i, m, v, k)
            inserta(actualComponente, componentes)

profundidad(l, i, m, v, k) → (m, v, k)
    m[i] ← ⊤
    inserta(i, actualComponente)
    para todo j ∈ l[i]
        si ¬m[j]
            profundidad(l, j, m, v, k)
    v[k] ← i
    k ← k − 1
```

b) Además, se analiza su eficiencia temporal.

En la función de ordenación topológica modificada, existen dos bucles externos, del orden de O(n).
En el segundo bucle externo, figura una llamada recursiva, por lo que deberemos analizar su complejidad también.
En la función profundidad modificada, existe otro bucle variable en función del número de vértices adyacentes a un vértice dado, cuya disparidad dependerá del nivel de conexión del grafo. Se plantea entonces una complejidad temporal en función del número de aristas en los casos extremos (árbol y cliqué).

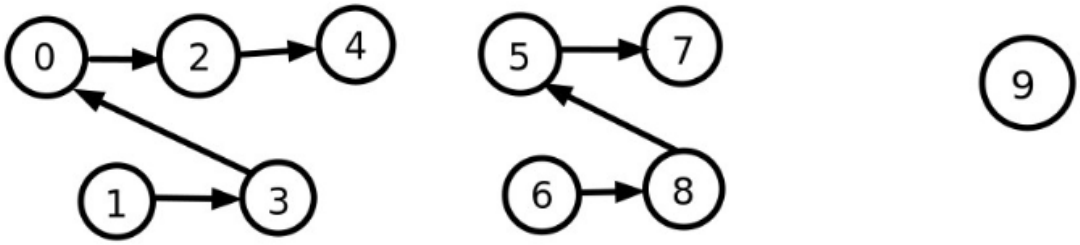
El número de aristas de cada vértice, *a*, está relacionado con el número de vértices:

$$0 \leq a \leq n - 1$$

Siendo 0 el caso de que el vértice no tenga ningún vértice adyacente y n-1 el caso de que tenga todos los demás vértices adyacentes.

Obtenemos entonces que:
- En el mejor caso → $t(n) = O(n)$
- En el peor caso → $t(n) = O(n^2)$

5 – Realice las modificaciones que considere oportunas en el algoritmo de ordenación topológica para que funcione correctamente con grafos no conexos. Justifique su respuesta. Aplíquelo al grafo de la figura e indique el resultado que obtendría. Relacione el número de soluciones válidas que pueden obtenerse para este problema con el número de componentes conexos de un grafo dado.
Criterios:
• Se diseña y justifica correctamente el algoritmo – 0,75
• Además, se obtiene una solución correcta al aplicarlo – 0,5
• Además, se relaciona correctamente el número de soluciones con el número de componentes conexas del grafo – 0,25
- Encuentre una cota inferior del número de órdenes topológicos posibles para un grafo orientado y acíclico en función del número de componentes conexas que se obtiene al ignorar la orientación de sus aristas (es decir, de su clausura simétrica). [Lo mismo que el apt. anterior]



Modificación del algoritmo:
El algoritmo de ordenación topológica, visto en clases, ya funciona correctamente con grafos no conexos. El bucle principal, que itera sobre cada nodo del grafo, garantiza que se visite y se procese cada componente conexa. Por lo tanto, no es necesario realizar ninguna modificación en el algoritmo para que maneje grafos no conexos.

Justificación:
Cuando un grafo es no conexo, está compuesto por múltiples componentes conexas. El algoritmo, al utilizar un bucle principal que itera sobre cada nodo, asegura que se llame a la función de búsqueda en profundidad para cada componente conexa que aún no haya sido visitada. Esto significa que se obtendrá una ordenación topológica para cada componente, y al final del proceso, se habrá incluido cada nodo del grafo en la ordenación.

Aplicación al grafo de la figura:
Aplicando el algoritmo de ordenación topológica al grafo proporcionado, que tiene las componentes conexas en el orden: 1-3-0-2-4, 6-8-5-7, 9, se obtiene el siguiente ordenamiento topológico:

4-2-0-3-1, 7-5-8-6, 9

Relación con componentes conexos:
El número de soluciones válidas de la ordenación topológica está directamente relacionado con el número de componentes conexos en el grafo. En este caso, dado que el grafo tiene 3 componentes conexas, hay (3!) (3 factorial) soluciones válidas de ordenación topológica, que es igual a 6. Esto se debe a que podemos permutar el orden de los componentes conexas de 3 maneras diferentes:

Componente 1, Componente 2, Componente 3
Componente 1, Componente 3, Componente 2
Componente 2, Componente 1, Componente 3
Componente 2, Componente 3, Componente 1
Componente 3, Componente 1, Componente 2
Componente 3, Componente 2, Componente 1
Esto muestra la relación entre el número de soluciones válidas y el número de componentes conexos en un grafo.

Diseñe un algoritmo que resuelva un laberinto tridimensional mediante búsqueda con retroceso. El laberinto está representado por un cubo en el que, originalmente, cada celda aparece libre (' ') o bloqueada ('X'). Se trata de encontrar, si es posible, una salida desde una posición inicial dada. A este efecto, se considera que una salida es cualquier celda libre del borde, es decir, de una de las caras del cubo. El algoritmo recibirá el laberinto y la posición inicial, que ha de ser válida y corresponder a una celda libre, y devolverá un booleano que indique si existe o no solución. Además, debe marcar las celdas visitadas ('V') y las que forman parte del camino de salida ('S'), si este existe.
a. Diseñe un algoritmo que resuelva el problema.

```
solución-laberinto(l, m, n, k, i, j, z) → (l,s)
    si l[i, j, z] = ' '
        si borde(m, n, k, i, j, z)
            s ← ⊤
        sí no
            l[i, j, z] ←'V'
            s ←  solución-laberinto(l, m, n, i − 1, j, z) ∨
                 solución-laberinto(l, m, n, i + 1, j, z) ∨
                 solución-laberinto(l, m, n, i, j − 1, z) ∨
                 solución-laberinto(l, m, n, i, j + 1, z) ∨
                 solución-laberinto(l, m, n, i, j, z - 1) ∨
                 solución-laberinto(l, m, n, i, j, z + 1)
        si s
            l[i, j, z] ←'S'
        sí no
            s ← ⊥
```

```
borde(m, n, k, i, j, z) → B
    si i ∈ {1, m} ∨ j ∈ {1, n} ∨ z ∈ {1, k}
        B ← T
    sí no
        B ← ⊥
```

b. Explicar la complejidad temporal.
En el peor de los casos, el algoritmo visita cada celda una vez, por lo que tiene una complejidad de O(m×n×k).

Un ordenador con *n* programas instalados está experimentando fallos debido a una incompatibilidad entre un programa o combinación de programas y el sistema operativo.
a) Diseñe un algoritmo basado en técnicas de exploración en grafos que determine que programa o combinación de programas producen el fallo. El algoritmo debe encontrar la solución más concreta. La representación del grafo en el algoritmo debe ser implícito. Suponga la existencia de una función "fallo" que recibe como argumento los programas que han sido iniciados y que devuelve el valor verdadero si el conjunto de esos programas producen un fallo en el ordenador.

```
programas(p,i,n,k,m,f) → k
        si i = n
                si fallo(p,n) ^ k < m
                        f ← p
                si no
                        k ← m
        si no
                p[i] ← T
                m ← programas(p, i+1, n, k+1, m, f)
                p[i] ← F
                m ← programas(p, i+1, n, k, m, f)
```

b) Determine la complejidad espacial y temporal del algoritmo.
Complejidad temporal:
Se trata de un algoritmo recursivo que explora todas las combinaciones posibles de programas. En cada llamada recursiva, el algoritmo toma una de dos decisiones: activa el programa o lo desactiva.
Dado que en cada llamada recursiva se toman dos decisiones, y hay n paradas en total, el árbol de recursión tendría una profundidad de n. En cada nivel del árbol, el número de nodos se duplica, lo que nos lleva a un árbol binario completo de altura n.
Por lo tanto, el número total de nodos en el árbol de recursión resultaría en una serie geométrica que suma a $2^{n+1}-1$.

En consecuencia, el orden de complejidad del algoritmo es $O(2^n)$.

Complejidad espacial:
El algoritmo opera sobre un vector booleano de n elementos, uno para cada programa en el S.O.
Por tanto, ya que este vector es pasado por parámetro en cada llamada recursiva y no se crea ninguno adicional, la complejidad espacial del algoritmo resulta del orden O(n).

Un camión ha de cubrir una ruta en la que existen m estaciones de servicio, ...,em. El camionero desea parar en el mínimo número posible de estaciones de servicio, teniendo en cuenta que llena el depósito cada vez que para, lo que le permite recorrer n kilómetros. El camionero parte con el depósito lleno y conoce las distancias entre las distintas estaciones, su lugar de origen y su destino, como se muestra en la figura.

a. Diseñé un algoritmo que empleé la técnica de exploración de grafos.
i: paradas consideradas
k: paradas realizadas
paradas: paradas mínimas
gasolina: combustible restante
p: vector paradas (booleano)
m: número de estaciones de servicio
d: vector distancias (km)

```
camionero(i, k, paradas, gasolina, p, m, d) → paradas
   si i = m
        si gasolina ≥ 0 ∧ paradas > k
                paradas ← k
   si no
        p[i] ← cierto
        k ← camionero(i+1, k+1, paradas, gasolina, p, m, d)
        p[i] ← falso
        k ← camionero(i+1, k, paradas, gasolina - d[i+1], p, m, d)
```

b. Explicar la complejidad temporal.
Se trata de un algoritmo recursivo que explora todas las combinaciones posibles de programas. En cada llamada recursiva, el algoritmo toma una de dos decisiones: activa el programa o lo desactiva.
Dado que en cada llamada recursiva se toman dos decisiones, y hay m paradas en total, el árbol de recursión tendría una profundidad de m. En cada nivel del árbol, el número de nodos se duplica, lo que nos lleva a un árbol binario completo de altura m.
Por lo tanto, el número total de nodos en el árbol de recursión resultaría en una serie geométrica que suma a $2^{m+1}-1$.

En consecuencia, el orden de complejidad del algoritmo es $O(2^m)$.

c. Explicar la complejidad espacial.
El algoritmo opera sobre un vector booleano de m elementos, uno para cada programa en el S.O.
Por tanto, ya que este vector es pasado por parámetro en cada llamada recursiva y no se crea ninguno adicional, la complejidad espacial del algoritmo resulta del orden O(m).