1. En el esquema general de los *algoritmos devoradores*, no aparece explícitamente la función objetivo. ¿En qué parte del esquema es tenida en cuenta? ¿Cómo?

```
devorador(C) → S
    S ← ∅
    mientras ¬solucion & C ≠ ∅
        p ← seleccion_C(C)
        C ← C - {p}
        si factible(p)
            S ← S ∪ {p}
```

2. Un pintor dispone de varias latas de pintura de diferentes capacidades, expresadas en litros. Cada litro cubre 1 m^2 de superficie. El pintor debe pintar una serie de habitaciones, cada una de ellas de diferente superficie. Diseñe un algoritmo que, siguiendo una estrategia devoradora, minimice el número de latas que sobran al finalizar el trabajo. Analice la complejidad del algoritmo. Describa los elementos que lo identifican como perteneciente al esquema general de los algoritmos devoradores.

candidatos: botes pintura
F.selección: latas + pequeñas o de - capacidad
F.objetivo: nº latas que sobran
F.factibilidad: —
F.solución: ¿se ha pintado todo?
Objetivo minimizar

```
devorador(C, sup) → S        (sup → superficie)
    S ← ∅                    p = superficie de la lata
    L ← C
    pared ← 0
    ordenar(L)               ] Θ(n log n)
    mientras pared ≠ sup & L ≠ ∅
        p ← extrae(L)
        S ← S ∪ {p}
        pared ← pared + p
```

$Θ(n)$

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

3. La versión entera (o 0/1) del *problema de la mochila* consiste en llenar una mochila con *objetos indivisibles*, maximizando su valor total. Se conocen los pesos, $p_1, \ldots, p_n$, y valores, $v_1, \ldots, v_n$, de los objetos disponibles. El peso total de los objetos seleccionados no ha de exceder la capacidad de la mochila, c.

Demuestre, mediante contraejemplos apropiados, que ninguna de las siguientes *estrategias devoradoras* permiten, por sí mismas, resolver exactamente el problema:

a) Primero los objetos más valiosos.

b) Primero los objetos menos pesados.

c) Primero los objetos de mayor relación valor-peso.

Diseñe un algoritmo aproximado para resolver el problema basado en cada una de ellas. Intente primero obtener un algoritmo que realice $O(n^2)$ operaciones elementales y estudie luego cómo podría mejorarse.

↳ poniendo un ordena y extrae.
↳ o usando un montículo.

```
devorador(C, captotal) → S
    S ← ∅
    capactual ← 0
    mientras capactual < captotal ∧ C ≠ ∅   ] Θ(n)
        (v,p) ← seleccionar(C)              ] Θ(n)
        C ← C - {(v,p)}
        si capactual + p ≤ captotal
            S ← S ∪ {(v,p)}
            capactual ← capactual + p
```

$Θ(n^2)$

```
a) seleccionar(C) → (v,p)
    x ← -∞
    para todo (a,b) ∈ C
        si a > x
            x ← a
            (v,p) ← (a,b)

b) seleccionar(C) → (v,p)
    x ← +∞
    para todo (a,b) ∈ C
        si b < x
            x ← b
            (v,p) ← (a,b)

c) seleccionar(C) → (v,p)
    x ← -∞
    para todo a/b > x
        x ← a/b
        (v,p) ← (a,b)
```

4. Tenemos que almacenar n ficheros de tamaños $t_1, \ldots, t_n$, respectivamente, en un disco de capacidad c. Desgraciadamente, no caben todos. Diga cuál sería la *estrategia devoradora* trivial que se seguiría en los siguientes supuestos:

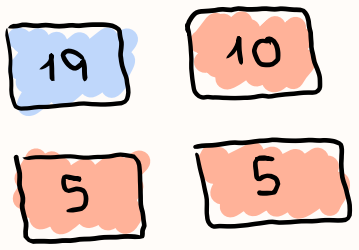
a) Se desea maximizar el número de ficheros. cojemos los más pequeños.

b) Se desea aprovechar al máximo la capacidad del disco. cojemos los más pesados.

¿Es la estrategia devoradora del segundo supuesto óptima? Si lo es, demuéstrelo. Si no lo es, ponga un contraejemplo.

a)

p.e. 20 unidades



b)

"a)" es la más óptima porque se llena entero

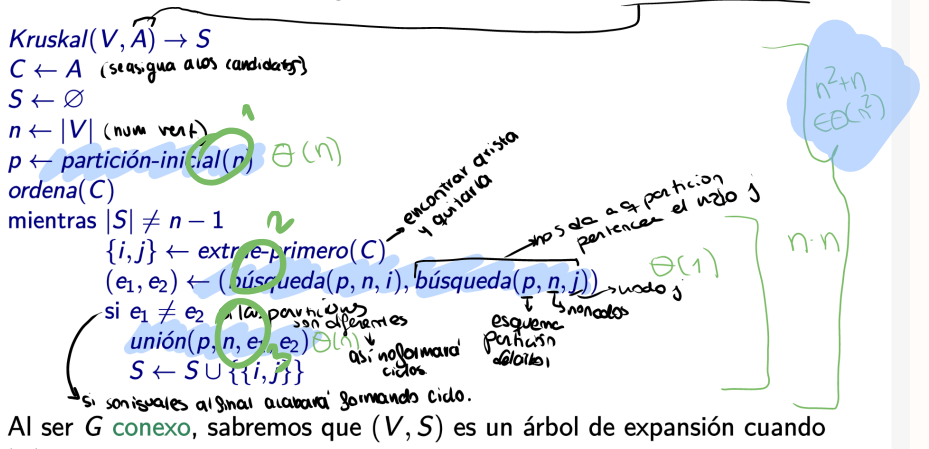
5. ¿Qué orden temporal tiene, en el peor caso, el algoritmo de *Kruskal* si las estructuras de partición se representan mediante *vectores de pertenencia*? Compare el resultado con el del algoritmo de *Prim*.

Hay que tener en cuenta las 3 operaciones.

Algoritmo de Kruskal

bosque con control de altura es igual

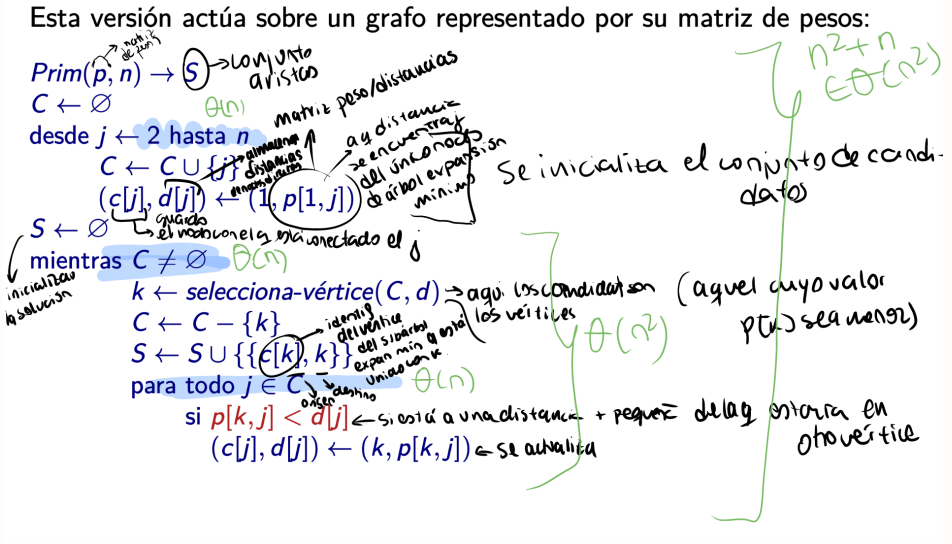
Esta versión actúa sobre un grafo representado por su lista de aristas:



Al ser G conexo, sabremos que (V, S) es un árbol de expansión cuando $|S| = n - 1$, condición que habrá de cumplirse en algún momento.

	Creación	Búsqueda	Unión
vector pertenencia	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
bosque	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
bosque con control altura	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$

Esta versión actúa sobre un grafo representado por su matriz de pesos:

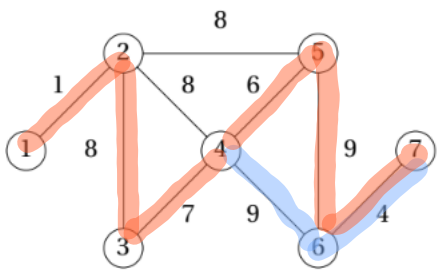


Prim también es $\Theta(n^2)$

(Si es denso, es decir, tiene muchas aristas Kruskal es peor pq almacena todas las aristas y Prim se adapta al caso).

6. Calcule, mediante los algoritmos de *Kruskal* y *Prim*, un árbol de expansión mínimo del siguiente grafo mostrando paso a paso la evolución de los algoritmos.

¿Cuántos árboles de expansión mínimos tiene este grafo? ¿Pueden todos encontrarse con ambos algoritmos? ¿Dónde se reflejan las distintas posibilidades en cada algoritmo?



1 A priori no se sabe y todos son distintos
2. Sí.
3. A partir del vértice 2 porque tenemos diferentes opciones a elegir

7. ¿Qué ocurre exactamente al aplicar los algoritmos de *Prim* y *Kruskal* a un grafo no conexo? Si es necesario, adáptelos para calcular un bosque de expansión mínimo del grafo.

No devolvería nada, peta.

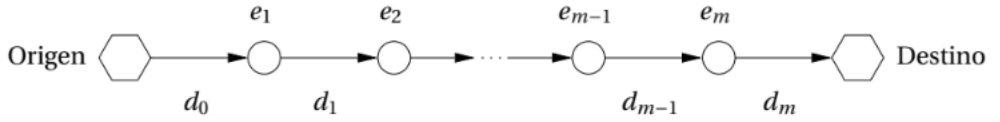
8. Describa cómo se pueden emplear montículos en el algoritmo de *Kruskal* en lugar de preordenación y qué ventaja se obtendría.

Al usar montículos este también reordena porque, cada vez que se extrae un elemento se reordena. Por lo tanto, el orden en ambos casos es el mismo. Pero, con montículos, puede no hacernos falta recorrer todos los candidatos como en la preordenación, si no que solo recorre x aristas (las suficientes).

9. Un camión ha de cubrir una ruta en la que existen m estaciones de servicio $e_1, \ldots, e_m$. 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) Diseñe un *algoritmo devorador* para resolver este problema.

b) Realice un análisis de su eficiencia temporal.



↳ paradas que tenemos

```
ruta(D, m, n) → S
S ← ∅
x ← n (gasolina)
mientras i ← 1 hasta m
    x ← x - D[i]
    si x < D[i+1]
        S ← S ∪ {i}
        x ← n
```

Θ(m)

? → candidatos: estaciones
F. selección: parar en la estación más lejana posible
F. objetivo: estaciones de servicio
F. factibilidad: —
F. solución: ¿Ha llegado al destino?
Objetivo: minimizar

10. Deseamos atravesar una autopista de m carriles en la que la policía ha establecido n *puntos de control*. En cada uno de ellos ha colocado *puestos de control* ocupando uno o más de los carriles, pero no todos.

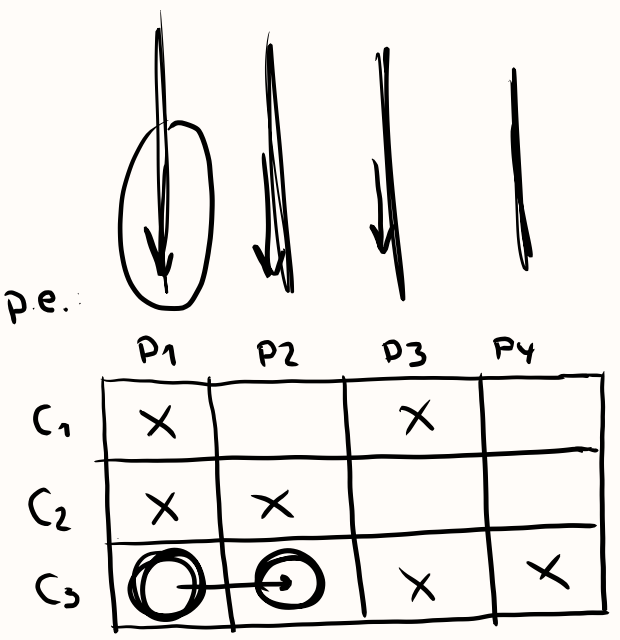
Conocemos, para cada punto de control, en qué carriles están situados los puestos de control. Para atravesar la autopista sin ser interceptados debemos cruzar cada punto de control por un carril en el que no esté la policía. El problema es que debemos realizar el mínimo número posible de cambios de carril (un único cambio de carril puede implicar cruzar varios carriles) a fin de no levantar sospechas.

Diseñe un algoritmo que calcule el camino óptimo, es decir, que diga por qué carril hay que pasar en cada punto de control para minimizar el número de cambios de carril.

El algoritmo recibirá una matriz binaria de m filas y n columnas en la que cada elemento informará sobre la existencia de un puesto de control en un carril y un punto de control dados. Como resultado devolverá un vector de n elementos indicando el carril elegido en cada punto de control.

candidatos: carriles
F. selección:
F. objetivo: cambio de carriles
F. factibilidad
F. solución: ¿ha conseguido el camino óptimo?
Objetivo: minimizar

p.e.:



camino(M) → v : M[8][8] → V[n]

```
aux←0
i←1
cont←0
mientras i≤C //controlar las columnas   (al empezar en 1, ¿≤ o <?)
    si (i==1) //estamos en la columna 1
        j←1
        b←false
        mientras (j<8 ∧ !b)
            si M[j][i]==0
                aux←j
                b←true
        j++
    sino si M[aux][i+1] //caso donde la siguiente posición de la que estábamos no hay hueco (misma fila, distinta columna).
        j←1
        b←false
        mientras (j<8 ∧ !b)
            si M[j][i]==0
                aux←j
                b←true
            j++
    v[i]←aux //si la siguiente posición (siguiente columna) de la misma fila es otro hueco se guarda en esa columna, otra vez, la misma fila.
    i++
```

M[i][j] = 1 → hueco
0 → hueco

11. Un restaurante dispone de mesas para dos, cuatro y seis comensales. Su aforo es limitado y no admite reservas. Cada mesa solo puede ser ocupada por un grupo durante todo el turno. Cuando un grupo llega al restaurante, el camarero debe asignarle mesa inmediatamente. Diseñe un algoritmo devorador que se encargue de asignar las mesas de un determinado turno. Describa los elementos que lo identifican como perteneciente al esquema general de los algoritmos devoradores. ¿Es la estrategia devoradora diseñada óptima?

candidatos: grupos de comensales
F.selección:
F.objetivo: mesas ocupadas
F.factibilidad
F.solución: ¿aforo lleno?
Objetivo maximizar

devorador(C, aforo : gente que llega) → S

```
S ← ∅
capactual←0
mientras (capactual≤aforo ∧ C ≠ ∅) ∨
    g ← extraer(C)
    si g=2 ∨ g=4 ∨ g=6
```

12. A pesar de que el restaurante del ejercicio anterior es de alta cocina, lo cierto es que no va nada bien. Sus responsables deciden pasar a un sistema de reserva obligatoria con fianza, de modo que se conozca con antelación el número de grupos y su tamaño en cada turno. Diseñe un nuevo algoritmo devorador. ¿Es la estrategia devoradora diseñada óptima?

↳ conjunto de tuplas de id y el tamaño de la mesa utilizada

devorador(C, aforo) → S

↳ (id, tamgrupo)

capactual ← 0

S ← ∅

L ← C

mientras capactual ≤ aforo ∧ L ≠ ∅

(i, t) ← extrae(L)

si t <

13. Se encuentra inmerso en el diseño de un videojuego RTS en el que el jugador puede desplazarse directamente de una ciudad a otra consumiendo recursos proporcionales a la distancia entre ambas. Suponga que conoce la posición en el mapa de todas las ciudades, pero que los caminos disponibles entre ciudades adyacentes solo se descubren cuando se entra en una ciudad, debido a la «niebla de la guerra». Diseñe un algoritmo que, siguiendo una estrategia devoradora, intente obtener el camino entre dos ciudades que consuma menos recursos. Describa los elementos que lo identifican como perteneciente al esquema general de los algoritmos devoradores. Analice la complejidad del algoritmo.

14. Un fontanero dispone de varios segmentos de tubo de distintas longitudes. Diseñe un algoritmo que, siguiendo una estrategia devoradora, minimice el número de uniones necesarias para construir una única tubería de longitud l. El fontanero puede realizar los cortes en los tubos que considere oportunos. Analice la complejidad del algoritmo.

candidatos: tubo
F.selección: tubería + grande q quepa
F objetivo. mínimo uniones
F. factibilidad ¿supera l?
F. solución: ¿suman l?
Objetivo minimizar

```
devorador(c, l) → S
    long ← 0
    ordena-decreciente(c)
    mientras long ≠ l ∧ c ≠ ∅
        p ← extrae(c)
        si long+p ≤ l
            S ← S ∪ {p}
            long ← long+p
    sino
        S ← S ∪ {l - long}
        long ← l
```

15. Considere un videojuego clásico de desplazamiento vertical en el que el jugador debe evitar que los asteroides que caen desde la parte superior de la pantalla colisionen con la nave espacial que controla, y que se encuentra en la parte inferior de la pantalla. Los asteroides se desplazan verticalmente, hacia la nave, y esta solo puede desplazarse horizontalmente para evitarlos. Por otro lado, la pantalla se encuentra dividida en celdas de igual tamaño. En cada momento, tanto la nave como cada uno de los asteroides ocupan una celda concreta. La pantalla se actualiza a intervalos regulares, tras los que la nave puede desplazarse a una de las dos columnas adyacentes para evitar a los asteroides, si es que no han colisionado ya con ella. Diseñe un piloto automático para la nave.