## Ejercicio 1

Se tiene un maratón de cine en el que se van a proyectar durante todo un día películas, todas diferentes en las N salas disponibles, de cada película se conoce entre otras cosas la duración, la sala de proyección y la hora de comienzo.
Escribir un algoritmo voraz para ayudar a la gente a planificar el maratón sabiendo que lo que quiere es ver el máximo número posible de películas. Analizar la complejidad del algoritmo resultante.
Nota: Como tenemos en horario de inicio y la duración podemos saber el final de cada película

```
Función películas (C[1..N], D[1..N]: array de enteros ): sol[1..N]: bool
VAR
    F[1..N], I[1..N] : array de enteros
    final← F[I[1]]
INICIO
    Desde i ← 1 hasta n hacer
        F[i]← C[i]+D[i]    //Cuando finaliza cada pelicula
        SOL[i] ←falso
    Fin_desde
    i ← ordenarIndices(F) //Ordenamos por las que finalicen antes
    SOL [I[1]]←cierto

    Desde i←2 hasta n hacer
        Si C[I[i]]>= final entonces //Si la siguiente que comienza
            SOL [I[i]] ←cierto
            Final←F [I[i]]
        Fin_si
Fin_desde
Fin_funcion
```

## Ejercicio 2

Deseamos almacenar en una cinta magnética de longitud L un conjunto de n programas {P1, P2, ... , Pn}. Sabemos que cada Pi necesita un espacio ai de la cinta y que (∑ ai) > L. 1<=i <=n
Construir un algoritmo que seleccione aquel subconjunto de programas que hace que el número de programas almacenado en la cinta sea máximo.

```
Función maximizarEspacio (P[1..N]:array de enteros, L:entero ): sol[1…n]:bool
VAR
    Ocupado ← 0
    I← 1
Inicio
    Desde i ←1 hasta n hacer
        Sol[i] ← falso
    Fin_desde
```

```
    Ordenar(P)

    Mientras ocupado + P[i] <= L hacer
        Sol[i]← verdadero
        Ocupado ← ocupado + P[i]
        I++
    Fin_mientras
Fin_funcion
```

## Ejercicio 3

Diseñar un algoritmo para almacenar N programas de longitud li<=Ii <= H en una cinta magnética de forma que el tiempo medio de lectura de los mismos sea mínimo. Se supone que los programas se leen con igual frecuencia y que antes de leer cada programa se rebobina la cinta. Se entiende por tiempo medio de lectura:

$$T = (1/N)\sum_{k=1}^{N}\sum_{i=1}^{k} l_i$$

```
Función minimizarTiempo (P[1..N]:array de enteros): sol[1..n]: array de enteros
Inicio
    Ordenar(P)
    Desde i ←1 hasta n hacer
        Sol[i] ← P[i]
    Fin_desde
Fin_funcion
```

## Ejercicio 4

Un vertex cover (recubrimiento de vértices) de un grafo no dirigido G=(V,E) es un conjunto de vértices
U, U⊆V, tal que cada arista del grafo G incide en, como mínimo, un vértice de U. Diseñar un algoritmo voraz que calcule el vertex cover de tamaño mínimo, si es que eso es posible. Comprobar si la estrategia elegida conduce siempre al óptimo. Repetir el mismo proceso pero para grafos que sean árboles.

1. Iniciar el coste de cada vértice a INFINITO, previo de cada nodo a NULL y visitado de cada vértice a FALSE.
2. Coste del nodo raíz a cero.
3. Obtener el vértice u con menor coste y que no haya sido visitado.
    3.1. Marcar u como visitado.
    3.2. Añadir arista (u,previo[u]) al árbol abarcador mínimo.
    3.2. Para cada vértice v adyacente a u y que aún no haya sido visitado.
        3.2.1. si coste[v] > peso de la arista (u,v)
            3.2.1.1. Actualizar coste[v]=peso de la arista (u,v)
            3.2.1.2. Asignar u como previo de v

4. Si aún no se han visitado todos los vértices, volver al paso 3.
   4.1. Si no, devolver el árbol abarcador mínimo

```
Prim ( L [1..n , 1..n ]) : 'conjunto de arcos

'Inicialización: sólo el nodo 1 se encuentra en B

T =NULL 'T contendrá los arcos del árbol de extensión mínima Distmin[1]=-1

para i=2 hasta n hacer
      más_próximo [ i ]=1
      distmin [ i ]=L [ i , 1]

para i=1 hasta n -1 hacer
      min=infinito

            para j=2 hasta n hacer
                  si 0 <= distmin [ j ] < min entonces
                  min=distmin [ j ]
                  k=j

             T=T union {{mas_próximo [ k ], k }}
             distmin [ k ]= -1 'se añade k a B

            para j=2 hasta n hacer
                   si L [ j , k ] < distmin [ j ] entonces
                  distmin [ j ]=L [ j , k ]
                  más_próximo [ j ]=k

devolver T
```

## Ejercicio 5

Se han de procesar n tareas. Cada tarea i tiene un instante de terminación ti y un valor vi>=0 si se procesa en dicho instante o antes. Se dispone de un sólo procesador que necesita sólo una unidad de tiempo para ejecutar cada tarea. Una solución factible es una secuencia S de tareas que cumplen las restricciones de terminación. Una solución óptima es aquella que maximiza V tal que:
V = ∑ vs; s € S
Diseñar un algoritmo voraz para encontrar la secuencia S

```
función PLAN_TAREAS(l:lista_tareas) : s:: lista_tareas
VAR
      Tfin ← 0
```

```
    Valor ←0
Inicio
    lo← ordenar_crec_por_instante_fin(l);
    mientras (!vacia(lo)) hacer
        t← primera(lo)
        avanzar(lo)
        si t.inicio ⩾ tfin entonces
            s← añadir(s, t)
            tfin←t.fin
            valor← t.valor
        fin_si
     fin_mientras
     devolver s
Fin_funcion
```