Recopilación ejercicios típicos solucionados examenes finales Algoritmos (2012-2019):

1. *(1.5 puntos)* A partir de la siguiente estructura de datos para la implementación de *conjuntos disjuntos*:

```
tipo
  Elemento = entero;
  Conj = entero;
  ConjDisj = vector [1..N] de entero
```

y del siguiente pseudocódigo para la unión de dos conjuntos:

```
procedimiento Unir (C, raíz1, raíz2)
            { supone que raíz1 y raíz2 son raíces }
  si raíz1 < raíz2 entonces C[raíz2] := raíz1
  sino C[raíz1] := raíz2
fin procedimiento
```

a) Escriba el correspondiente pseudocódigo de `Buscar (C, x) : Conj`, que devuelve el nombre del conjunto (es decir, su representante) de un elemento dado.

b) Razone cuál sería la complejidad computacional de una secuencia de m búsquedas y $n-1$ uniones.

Hay 3 funciones "Unir" y cada una tiene asociado un "Buscar". En este caso nos dan la función Unir2, por ello habrá que poner el pseudocódigo Buscar correspondiente, el cual es el siguiente:

a)

```
función Buscar2 (C, x) : Conj
  r := x;
  mientras C[r] <> r hacer
    r := C[r]
  fin mientras;
  devolver r
fin función
```

b)

O(m* n), ya que por cada una de esas "m" busquedas, hay "n" uniones.

(2,5 puntos) Diccionario de datos:

a) Diseñe, escribiendo su pseudocódigo, los algoritmos `inicializarTabla`, `insertar`, `buscar` y `eliminar` usando *exploración cuadrática*, de modo que las tres últimas rutinas se ejecuten en un tiempo promedio constante. Use la siguiente declaración de tipos:

```
tipo
  ClaseDeEntrada = (legítima, vacía, borrada)
  Índice = 0..N-1
  Posición = Índice
  Entrada = registro
    Elemento : TipoElemento
    Información : ClaseDeEntrada
  fin registro
  TablaDispersión = vector [Índice] de Entrada
```

Si utilizase algún procedimiento auxiliar (distinto de la función *hash*), refleje también su pseudocódigo.

b) Con la siguiente función *hash*:

$$hash(\text{“}a\text{''}, 11) = 8$$
$$hash(\text{“}b\text{''}, 11) = 7$$
$$hash(\text{“}c\text{''}, 11) = 7$$
$$hash(\text{“}d\text{''}, 11) = 7$$
$$hash(\text{“}e\text{''}, 11) = 8$$
$$hash(\text{“}f\text{''}, 11) = 8$$

muestre el resultado de insertar las claves: “a”, “b”, “c”, “d”, “e” y “f” (en ese orden) en la siguiente tabla:

0	1	2	3	4	5	6	7	8	9	10

cuando se emplea:

1) exploración lineal
2) exploración cuadrática
3) exploración doble usando $5 - (x \bmod 5)$ como segunda función (siendo x el resultado de la primera función).

Indique asímismo el número total de colisiones que se produce durante las inserciones en cada una de las tres expoloraciones.

a)

```
procedimiento InicializarTabla (D)
  para i := 0 hasta N-1 hacer
    D[i].Información := vacía
  fin para
fin procedimiento
```

```
procedimiento Insertar (Elem, D)
  pos = Buscar(Elem, D);
  si D[pos].Información <> legítima
  entonces         {Bueno para insertar}
    D[pos].Elemento := Elem;
    D[pos].Información := legítima
fin procedimiento
```

```
función Buscar (Elem, D): Posición
  i := 0;
  x = Dispersión(Elem);
  PosActual = x;
  mientras D[PosActual].Información <> vacía y
             D[PosActual].Elemento <> Elem hacer
    i := i + 1;
    PosActual := (x + FunResoluciónColisión(x, i)) mod N
  fin mientras;
  devolver PosActual
fin función
  /*La búsqueda finaliza al caer en una celda vacía
    o al encontrar el elemento (legítimo o borrado)*/
```

```
procedimiento Eliminar (Elem, D)
  pos = Buscar(Elem, D);
  si D[pos].Información = legítima
  entonces
    D[pos].Información := eliminada
fin procedimiento
```

nción FunResolucionColision(x,i) : Posición
Para exploración lineal*/
devolver i
n función

función FunResolucionColision(x,i) : Posición
/*Para exploración cuadrática*/
aux =i ^2;
devolver aux
fin función

función FunResolucionColision(x,i,y) :
/*Para exploración doble*/
aux =y - (x mod y);
devolver aux
fin función

b)

1.

0	1	2	3	4	5	6	7	8	9	10
e	f						b	a	c	d

12 cols

2.

0	1	2	3	4	5	6	7	8	9	10
c	f				d		b	a	e	

8 cols

- La función de resolución de colisiones es cuadrática, por lo general: $f(i) = i^2$.

3.

0	1	2	3	4	5	6	7	8	9	10
	e	d	f				b	a		c

8 cols

5. *(1.5 puntos)* Escriba el pseudocódigo de un algoritmo voraz que genere una *ordenación topológica* de los nodos de un grafo dirigido acíclico. Identifique en él los elementos característicos de los algoritmos voraces. Complete su respuesta analizando el algoritmo e identificando su peor caso.

```
función Ordenación topológica 1 (G:grafo): orden[1..n]
    Grado Entrada [1..n] := Calcular Grado Entrada (G);
    para i := 1 hasta n hacer Número Topológico [i] := 0;
    contador := 1;
    mientras contador <= n hacer
        v := Buscar nodo de grado 0 sin número topológico asignado;
        si v no encontrado entonces
            devolver error "el grafo tiene un ciclo"
        sino
            Número Topológico [v] := contador;
            incrementar contador;
            para cada w adyacente a v hacer
                Grado Entrada [w] := Grado Entrada [w] - 1
        fin si
    fin mientras;
    devolver Número Topológico
fin función
```

- **Análisis**: $O(n+m)$ con listas de adyacencia
  Peor caso: grafo denso $[m \rightarrow n(n-1)]$ y visita todas las aristas
  Mejor caso: grafo disperso $[m \rightarrow 0, m \rightarrow n]$

Características de los algoritmos voraces

- Resuelven **problemas de optimización**:
  En cada fase, toman una decisión (selección de un *candidato*), satisfaciendo un óptimo local según la información disponible, esperando así, en conjunto, satisfacer un óptimo global.
- Manejan un **conjunto de candidatos** C:
  En cada fase, retiran el candidato seleccionado de C, y si es aceptado se incluye en S, el conjunto donde se construye la solución $\equiv$ candidatos aceptados
- **4 funciones** (no todas aparecen explícitamente en el algoritmo):
  1. ¿S es **Solución**?
  2. ¿S es **Factible**? (¿nos lleva hacia una solución?)
  3. **Selección**: determina el mejor candidato
  4. **Objetivo**: valora S (está relacionada con *Selección*)

  $\rightarrow$ Encontrar S: *Solución* que optimiza *Objetivo* (max/min)

Árbol expandido mínimo (3)

- **Comparación**:

		Prim $\Theta(n^2)$	Kruskal $O(mlogn)$
Grafo denso:	$m \rightarrow n(n-1)/2$	$\boxed{\Theta(n^2)}$	$O(n^2 logn)$
Grafo disperso:	$m \rightarrow n$	$\Theta(n^2)$	$\boxed{O(nlogn)}$

Tabla: Complejidad temporal de los algoritmos de Prim y Kruskal

4. *(1'5 puntos)* Presente en una tabla los elementos característicos de los algoritmos voraces que pueda identificar en los algoritmos de Kruskal, Prim y Dijkstra.

Algoritmo de Kruskal (1)

- Inicialmente: T vacío
- Invariante: (N,T) define un conjunto de *componentes conexas* (i. e. subgrafos, árboles)
- Final: sólo una componente conexa: el a. e. m.
- *Selección*: lema $\rightarrow$ **arista más corta...**
- *Factible*?: **...que una componentes conexas distintas**
- Estructuras de datos:
  - "grafo": aristas ordenadas por peso
  - árboles: Conjuntos Disjuntos (buscar(x), fusionar(A,B))

Algoritmo de Prim (1)

- Kruskal: bosque que crece hasta convertirse en el a. e. m.
  Prim: **un único árbol** que va creciendo hasta alcanzar todos los nodos.
- Inicialización: $B = \{\text{nodo arbitrario}\} = \{1\}$, T vacío
- *Selección*: arista más corta que parte de B:
  $(u,v), u \in B \wedge v \in N-B$
  $\Rightarrow$ se añade (u,v) a T y v a B
- Invariante:
  T define en todo momento un a.e.m. del subgrafo (B,A)
- Final: $B = N$ (*Solución?*)

Algoritmo de Dijkstra (4)

- **Teorema**: Dijkstra encuentra los caminos mínimos desde el origen hacia los demás nodos del grafo.
  Demostración por inducción.
- **Análisis**: $|N| = n, |A| = m, L[1..n, 1..n]$
  Inicialización $= \Theta(n)$
  ¿Selección de v?
  $\rightarrow$ *"implementación rápida"*: recorrido sobre C
  $\equiv$ examinar $n-1, n-2, ..., 2$ valores en D, $\sum = \Theta(n^2)$
  Para anidado: $n-2, n-3, ..., 1$ iteraciones, $\sum = \Theta(n^2)$
  $\boxed{T(n) = \Theta(n^2)}$
- **Mejora**: si el grafo es disperso ($m << n^2$), utilizar listas de adyacencia
  $\rightarrow$ ahorro en para anidado: recorrer lista y no fila o columna de L

2. *(2 puntos)* Compare la *ordenación por fusión* con la *ordenación rápida* desde el punto de vista del diseño del algoritmo (técnica de diseño, fases de la técnica utilizada, estrategias para la mejora del tiempo de ejecución...) así como de su complejidad (relaciones de recurrencia según cada caso, que se justificarán y se resolverán, teorema necesario para resolverlas...).

Ordenación por Fusión (2)

- *mergesort*
- O bien, ordenación *por intercalación*.
- Utiliza un algoritmo de **Fusión** de un vector cuyas *mitades* están ordenadas para obtener un vector ordenado.
- El procedimiento Fusión es lineal (n comparaciones).
- Ordenación: algoritmo $\boxed{\text{Divide y Vencerás}}$
  - Divide el problema en 2 *mitades*, que
  - se resuelven recursivamente;
  - *Fusiona* las mitades ordenadas en un vector ordenado.
- **Mejora**: Ordenación por Inserción para *vectores pequeños*:
  $n <$ *umbral*, que se determina empíricamente.

Ordenación por Fusión (4)

- **Análisis** de la versión puramente recursiva (UMBRAL $= 0$):
  $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$ (Fusión)
  $$n = 2^k \Rightarrow \begin{cases} T(1) & = O(1) & = 1 \\ T(n) & = 2T(n/2) + O(n) & = 2T(n/2) + n, n > 1 \end{cases}$$
  Teorema Divide y Vencerás: $l = 2, b = 2, c = 1, k = 1, n_0 = 1$
  caso $l = b^k \Rightarrow \boxed{T(n) = \Theta(nlogn)}$
- Podría mejorarse la complejidad espacial ($= 2n$: vector auxiliar)
  $\rightarrow$ El algoritmo adecuado es *quicksort*
- **Observación**:
  Importancia de *balancear* los subcasos en Divide y Vencerás:
  Si llamadas recursivas con vectores de tamaño $n-1$ y 1
  $\Rightarrow T(n) = T(n-1) + T(1) + n = O(n^2)$

Ordenación Rápida (*quicksort*) [Hoare]

- Paradigma de *Divide y Vencerás*.
  Con respecto a Fusión:
  - más trabajo para construir las subinstancias (pivote...),
  - pero trabajo nulo para combinar las soluciones.
- **Selección del pivote** en $T[i..j]$:
  - Objetivo: obtener una partición lo más balanceada posible
    $\Rightarrow$ ¿**Mediana**? Inviable
  - Usar el **primer valor** del vector ($T[i]$):
    Ok si la entrada es aleatoria.
    Pero elección muy desafortunada con entradas ordenadas o parcialmente ordenadas (caso bastante frecuente)
    $\rightarrow O(n^2)$ para no hacer nada...
  - Usar un valor elegido al azar (**pivote aleatorio**):
    Más seguro, evita el peor caso detectado antes, pero depende del generador de números aleatorios (eficiencia vs. coste).
  - Usar la **mediana de 3 valores**: $T[i], T[j], T[(i+j)div2]$
- **Peor caso**: p *siempre* es el menor o el mayor elemento
  $\Rightarrow T(n) = T(n-1) + cn, n > 1$
  $\Rightarrow \boxed{T(n) = O(n^2)}$
- **Mejor caso**: p *siempre* coincide con la mediana
  $\Rightarrow T(n) = 2T(n/2) + cn, n > 1$
  $\Rightarrow \boxed{T(n) = O(nlogn)}$

- **Análisis** (Cont.):
  - **Caso medio**:
    Sea z: tamaño de la parte izquierda;
    Cada valor posible para z $(0, 1, 2, ..., n-1)$ es *equiprobable*: $p = 1/n$
    $\leftrightarrow T(z) = T(n-z-1) = 1/n \sum_{x=0}^{n-1} T(x)$
    $\Rightarrow T(n) = 2/n[\sum_{x=0}^{n-1} T(x)] + cn, n > 1$

- **Teorema de resolución de recurrencias Divide y Vencerás**

$$T(n) = \ell T(n/b) + cn^k, n > n_0 \qquad (1)$$

Con $\ell \geq 1, b \geq 2, k \geq 0, n_0 \geq 1 \in \mathbb{N}$ y $c > 0 \in \mathbb{R}$, cuando n/n_0 es potencia exacta de b $(n \in \{bn_0, b^2 n_0, b^3 n_0 \dots\})$.

- **Teorema Divide y Vencerás**:
  Si una recurrencia es de la forma (1), se aplica

$$T(n) = \begin{cases} \theta(n^k) & si \quad \ell < b^k \\ \theta(n^k logn) & si \quad \ell = b^k \\ \theta(n^{log_b \ell}) & si \quad \ell > b^k \end{cases} \qquad (2)$$

6. *(2'5 puntos)* Considerando un sistema monetario $M = \{v_1, v_2, \ldots, v_m\}$, se dispone de una función Monedas que utiliza la técnica de Programación Dinámica para encontrar el número mínimo de monedas para pagar la cantidad n, calculando para ello una tabla T con todos los resultados intermedios (solución óptima para pagar la cantidad j con las monedas $v_1..v_i$). El resultado encontrado puede corresponder a una o varias configuraciones (conjuntos de monedas que suman n).

Se plantea el diseño de una función Composición que, a partir de la tabla T ya construida por la función Monedas, devuelva una configuración posible de la solución, especificando el conjunto de monedas que la componen:

a) Proponga un ejemplo del problema, presentando su tabla T y sus posibles soluciones.

b) Proponga un tipo de datos adecuado para la salida de la función Composición, que ilustrará con el ejemplo anterior.

c) Proponga un pseudocódigo de la función Composición.

d) Determine su complejidad.

a)

construir la tabla con la que podría determinarse en programación dinámica la manera óptima de pagar una cantidad de 17 unidades de valor con un mínimo de monedas, sabiendo que el sistema monetario considerado está constituido por monedas de 1, 3, 8 y 12 unidades de valor. Indicar la solución al problema dibujando una traza en la tabla anterior para justificar cómo se obtiene.

Denominación de la moneda

Importe a pagar

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
3	0	1	2	1	2	3	2	3	4	3	4	5	4	5	6	5	6	7
8	0	1	2	1	2	3	2	3	1	2	3	2	3	4	3	4	2	3
12	0	1	2	1	2	3	2	3	1	2	3	2	1	2	3	2	2	3

1 8 8

b) y c)

funcion Composicion (M [1 .. m, 0 .. n], V [1 .. m]) : C[1 .. n]
  i = m
  j = n
  contador = 0
  mientras i <> 1 **y** j <> 0 **hacer**
    si M[i,j] <> M[i-1, j] **entonces**
    C[contador] = V[i]
    incrementar contador
    j = j - V[i]
    si no
     i = i -1
    fin si
  fin mientras
  mientras j <> 0 **hacer**
    C[contador] = V[i]
    incrementar contador
    j = j - v[i]
  fin mientras
 devolver C
fin funcion

d)

- **Análisis**: $\boxed{T(n) = \Theta(mn)}$
- **Problema**: ¿Conjunto de monedas?
  $\Rightarrow$ Algoritmo voraz sobre c: camino $c[m,n] \to c[0,0]$
  m "pasos" hacia arriba $\equiv$ "no utilizar más v_i"
  $+c[m,n]$ "saltos" hacia la izquierda $\equiv$ "utilizar una v_i más"
  $\Rightarrow$ $\boxed{\Theta(m + c[m,n])}$ adicional a la construcción de la tabla

b) **función** Composición (M[1..n,0..m], V[1..n]) : C[1..n]
 para i:=1 **hasta** n **hacer** C[i]:=0;
  v:=M[n,m];
  i:=n; j:=m;
 mientras i>1 y v>0 **hacer**
   si M[i,j] <> M[i-1,j] **entonces**
    C[i]:=C[i]+1;
    j:=j-V[i];
    v:=M[i,j]
   si no
    i:=i-1;
   fin si;
 fin mientras;
 si v>0 **entonces** C[1]:=C[1]+1;   % caso particular: i=1
 devolver C
fin función

2. *(2,0 puntos)* Escriba el pseudocódigo del algoritmo de *ordenación rápida con pivote aleatorio* y justifique el análisis de su complejidad.

```
procedimiento OrdenarAux (V[izq..der])
  si izq+UMBRAL <= der entonces
    x := {nº aleatorio en el rango [izq..der]};
    pivote := V[x];
    intercambiar (V[izq], V[x]);
    i := izq + 1;
    j := der;
    mientras i <= j hacer
      mientras i <= der y V[i] < pivote hacer
        i := i + 1
      fin mientras;
      mientras V[j] > pivote hacer
        j := j - 1
      fin mientras;
      si i <= j entonces
        intercambiar (V[i], V[j]);
        i := i + 1;
        j := j - 1
      fin si
    fin mientras;
    intercambiar (V[izq], V[j]);
    OrdenarAux (V[izq..j-1]);
    OrdenarAux (V[j+1..der])
    fin si
fin procedimiento

procedimiento Ordenación Rápida (V[1..n])
  OrdenarAux(V[1..n]);
  si (UMBRAL > 1) entonces
    Ordenación por Inserción (V[1..n])
```

- **Análisis**: pivote aleatorio y sin umbral
  $$\Rightarrow \begin{cases} T(0) = T(1) = 1 \\ T(n) = T(z) + T(n-z-1) + cn, n > 1 \end{cases}$$
  - **Peor caso**: p *siempre* es el menor o el mayor elemento
    $\Rightarrow T(n) = T(n-1) + cn, n > 1$
    $\Rightarrow \boxed{T(n) = O(n^2)}$
  - **Mejor caso**: p *siempre* coincide con la mediana
    $\Rightarrow T(n) = 2T(n/2) + cn, n > 1$
    $\Rightarrow \boxed{T(n) = O(nlogn)}$

3. *(2,5 puntos)* Dado el algoritmo siguiente:

```
función Kruskal ( G =(N,A) ) :  tipo_salida {1}
    Organizar los candidatos a partir de A; {2}
    n := |N|; T := conjunto vacio; peso := 0;
    inicializar n conjuntos, cada uno con un nodo de N;
    repetir
        Seleccionar y extraer un candidato a; {3}
        ConjuntoU := Buscar (a.nodo1); ConjuntoV := Buscar (a.nodo2);
        si ConjuntoU <> ConjuntoV entonces
            Fusionar (ConjuntoU, ConjuntoV);
            T := T U {a};
            peso := peso + a.peso;
        fin si
    hasta |T| = n-1;
    devolver <T, peso>
fin función
```

a) Precise el tipo de datos de la salida de la función ({1}).

b) Reescriba las instrucciones {2} y {3} introduciendo el uso de un montículo, decisión que justificará en su respuesta.

c) Enuncie de forma precisa el problema que resuelve este algoritmo.

d) Concrete los elementos característicos de la técnica voraz que correspondan a este ejemplo.

e) Análisis detallado de la complejidad.

a) Tipo Árbol

b) {2}: M=CrearMonticulo(); {3}:EliminarMax(M).
Justificación: No cambia la complejidad del peor caso pero se obtienen mejores tiempos

c) Encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo.

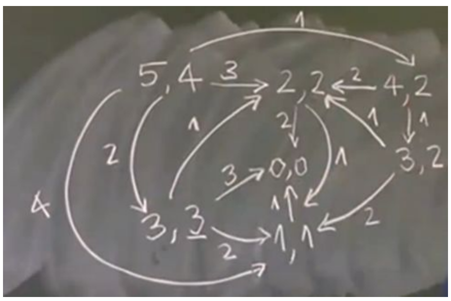
d) **Algoritmo de Kruskal (1)**

- Inicialmente: T vacío
- Invariante: (N, T) define un conjunto de *componentes conexas* (i. e. subgrafos, árboles)
- Final: sólo una componente conexa: el a. e. m.
- *Selección*: lema $\rightarrow$ **arista más corta...**
- *Factible*?: **...que una componentes conexas distintas**
- Estructuras de datos:
  - "grafo": aristas ordenadas por peso
  - árboles: Conjuntos Disjuntos (buscar(x), fusionar(A, B))

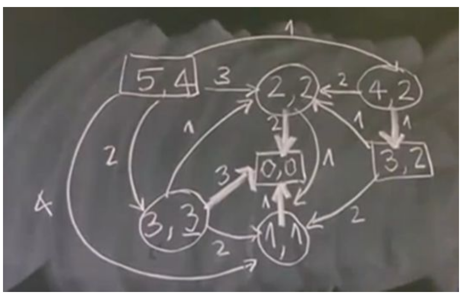
e)
- **Análisis**: $|N| = n \wedge |A| = m$
  ordenar A: $O(m\log m) \equiv O(m\log n)$: $n-1 \leq m \leq n(n-1)/2$
  + inicializar n conjuntos disjuntos: $\Theta(n)$
  + $2m$ buscar (peor caso)
    y $n-1$ fusionar (siempre): $O(2m\alpha(2m, n)) = O(m\log n)$
  + resto: $O(m)$ (peor caso)
  $\Rightarrow \boxed{T(n) = O(m\log n)}$

7. *(1 punto)* Represente mediante un grafo decorado todas las situaciones de juego que podrían alcanzarse a partir de un montón de 5 palillos para la variante del *juego de Nim* vista en clase.

5 palillos, 4 jugadas posibles

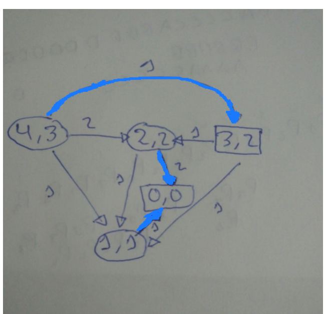


Grafo sin decorar



Grafo decorado

Con 4 palillos:



6. Cálculo de un coeficiente binomial C(n,k) utilizando la técnica de *Programación Dinámica* de forma que las necesidades de memoria sean mínimas:

   a) Proponga un ejemplo concreto, partiendo del triángulo de Pascal, para explicar el algoritmo.

   b) Determine su complejidad temporal y espacial.

```
función C(n, k): valor
    si k = 0 ó k = n entonces devolver 1
    sino devolver C(n-1, k-1) + C(n-1, k)
    fin si
fin función
```

b) $T(n) = \Theta(nk)$ y la complejidad espacial también.

a)

	0	1	2	…	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
…						
$n-1$					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n						$\binom{n}{k}$

1. *(2,5 puntos)* A partir de la siguiente estructura de datos para la implementación de un *árbol binario de búsqueda* :

```
tipo
    PArbol = ^Nodo
    Nodo =
      registro
        Elem : TipoElemento
        Izq, Der : PArbol
      fin registro
    ABB = PArbol
```

   a) Diseñe, escribiendo su pseudocódigo, un recorrido del árbol en *orden de nivel* (es decir, todos los nodos de profundidad p se procesan antes que cualquier nodo con profundidad $p+1$) de modo que esta rutina se ejecute en **tiempo lineal**. El procesamiento de cada nodo consiste en su visualización.

   Refleje en el diseño todos los procedimientos y **estructuras de datos** necesarias.

   b) Determine, justificando su respuesta sobre el pseudocódigo, la O de cada operación.

- Orden de nivel: Todos los nodos con profundidad p se procesan antes que cualquier nodo con profundidad $p+1$.
  - Se usa una cola en vez de la pila implícita en la recursión. O(n)

```
procedimiento OrdenDeNivel (A)
  CrearCola(C);
  si A <> nil entonces InsertarEnCola(A, C);
  mientras no ColaVacía(C) hacer
    p:= QuitarPrimero(C);
    escribir(p^.Elemento); {Operación principal}
    si p^.Izq <> nil entonces InsertarEnCola(p^.Izq, C);
    si p^.Der <> nil entonces InsertarEnCola(p^.Der, C);
  fin mientras
fin procedimiento
```

Funciones aux de TAD Cola:

```
tipo Cola = registro
  Cabeza_de_cola, Final_de_cola: 1..Tamaño_máximo_de_cola
  Tamaño_de_cola : 0..Tamaño_máximo_de_cola
  Vector_de_cola : vector [1..Tamaño_máximo_de_cola]
                       de Tipo_de_elemento
fin registro
```

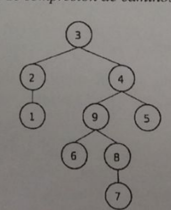
```
procedimiento Crear_Cola ( C ) O(1)
  C.Tamaño_de_cola := 0;
  C.Cabeza_de_cola := 1;
  C.Final_de_cola := Tamaño_máximo_de_cola
fin procedimiento
```

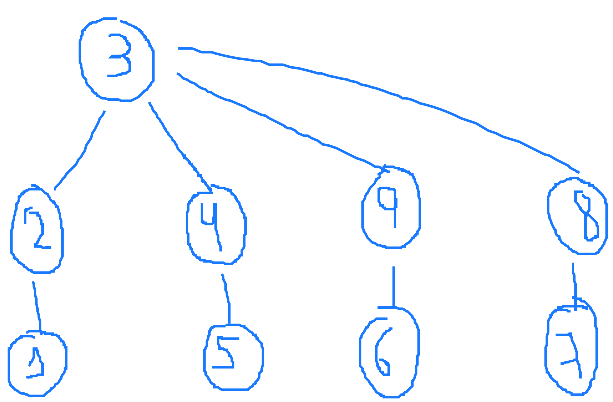
```
función Cola_Vacía ( C ) : test O(1)
  devolver C.Tamaño_de_cola = 0
fin función
```

```
procedimiento incrementar ( x ) (* privado *) O(1)
  si x = Tamaño_máximo_de_cola entonces x := 1
  sino x := x + 1
fin procedimiento
```

```
procedimiento Insertar_en_Cola ( x, C ) O(1)
  si C.Tamaño_de_Cola = Tamaño_máximo_de_cola entonces
    error Cola llena
  sino
    C.Tamaño_de_cola := C.Tamaño_de_cola + 1;
    incrementar(C.Final_de_cola);
    C.Vector_de_cola[C.Final_de_cola] := x;
fin procedimiento
```

2. *(0.5 puntos)* ¿Cómo quedaría el siguiente árbol, que representa un conjunto, tras buscar el elemento 8 usando la técnica de *compresión de caminos*?





4. *(1,5 puntos)* Dado el grafo dirigido pesado $G = (N, A)$, con $N = \{a, b, c, ...h\}$ y A determinado por la lista de aristas siguiente:

(a,d)	(a,f)	(d,h)	(d,f)	(h,e)	(h,g)	(e,g)	(f,g)	(f,b)	(f,c)	(b,c)
1	2	1	6	2	7	2	3	4	5	2

a) Dibuje el grafo G.

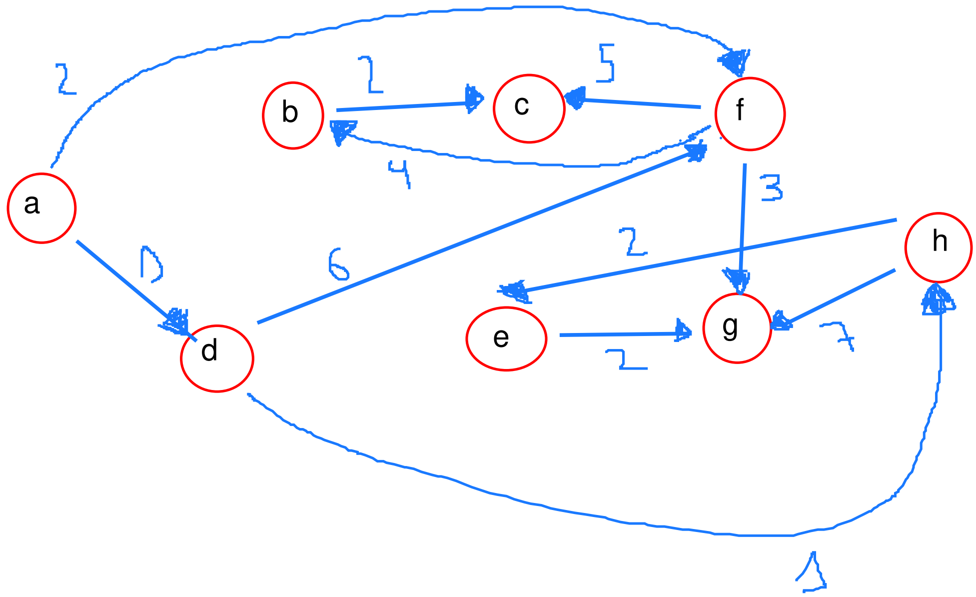
b) Indique el resultado de un recorrido en profundidad sobre G, partiendo del nodo *a*.

c) Presente una ordenación topológica de los nodos de G.

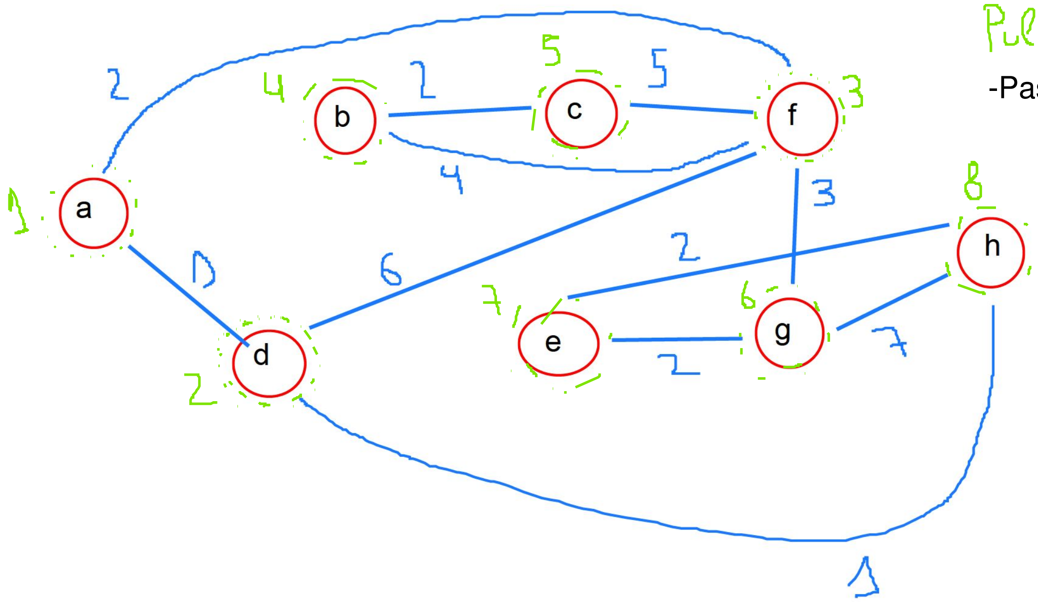
d) Dibuje un árbol expandido mínimo del grafo no dirigido subyacente, indicando igualmente su peso total.

e) Dibuje el árbol con los caminos mínimos entre el nodo *a* y los demás, asociando a cada nodo la distancia mínima calculada.

a)



b)



Pila

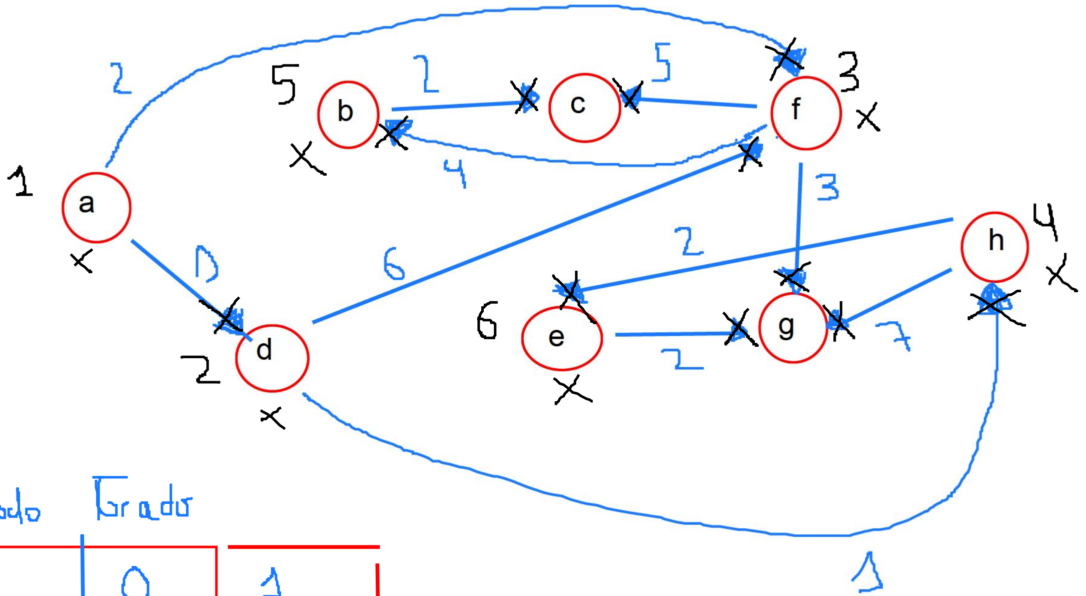
a	d	f	b	c

-Paso 5, desapilamos hasta f

a	d	f	g	e	h

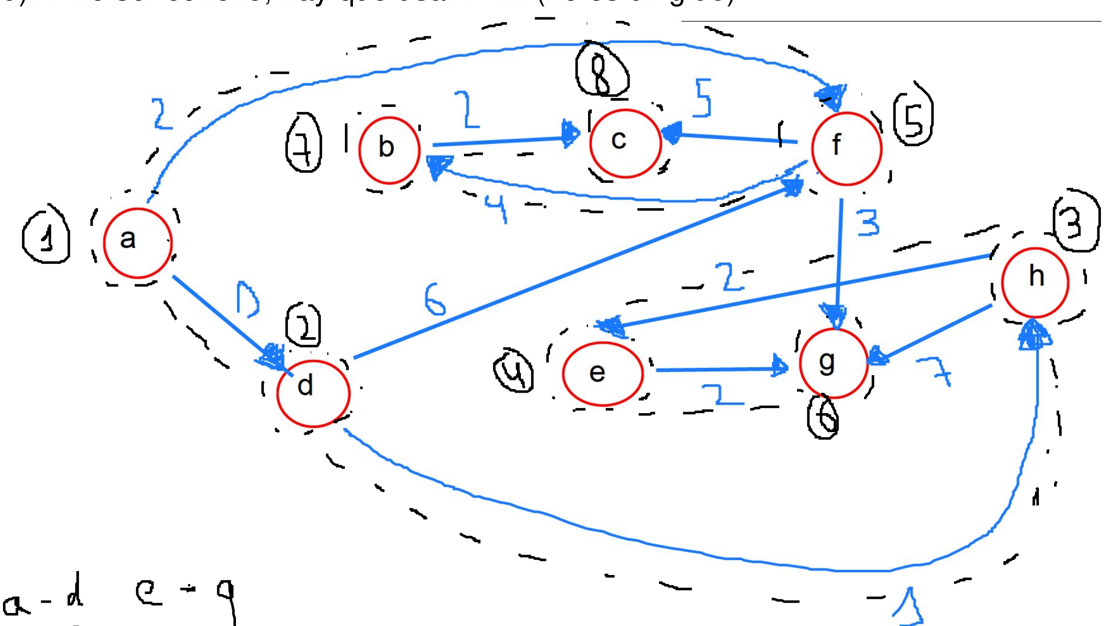
Solución: a,d,f,b,c,g,e,h

c)



Nodo	Grado	
a	0	1
b	1	5
c	2	7
d	1	2
e	1	6
f	2	3
g	3	8
h	1	4

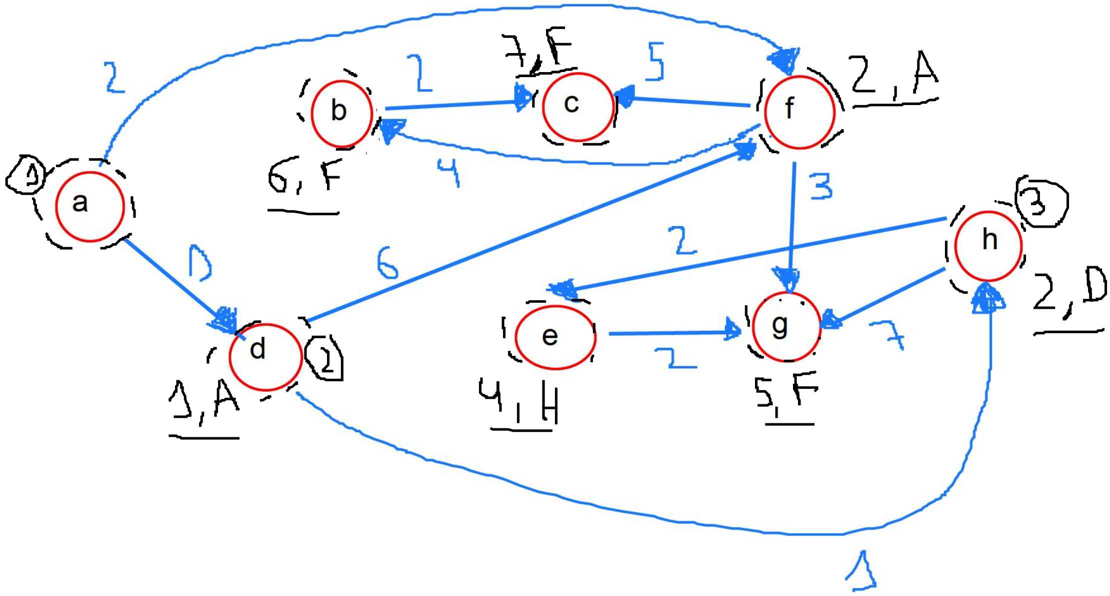
d) Al no ser conexo, hay que usar Prim (no es dirigido)



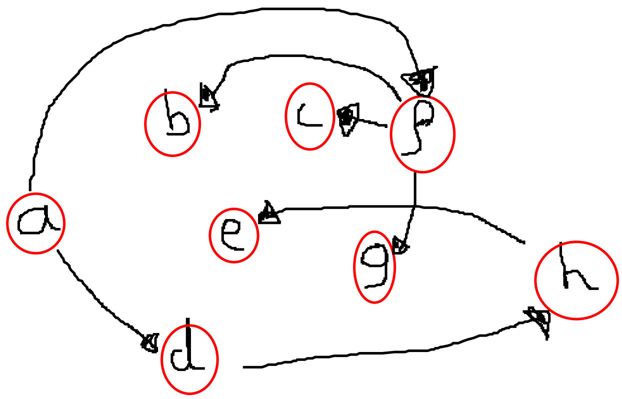
a - d   e - g
a - f   f - b
d - h   b - c

Total= 1+2+1+2+2+4+2=14 peso

e)



Solución:



A-D	1
A-F	2
A-D-H	2
A-D-H-E	4
A-F-B	6
A-F-C	7
A-F-G	5

8. *(0.5 puntos)* Supongamos que descubrimos una máquina de Turing determinista que resuelve el problema de la suma de subconjuntos en un número de pasos acotado por una función polinómica p(n), siendo n el tamaño del problema. ¿Qué podremos deducir entonces sobre la relación entre las clases de complejidad P y NP?

El problema de $\mathcal{P}$ y $\mathcal{NP}$ (2)

- Se cree que $\mathcal{P} \neq \mathcal{NP}$; pero hasta ahora no se ha logrado demostrarlo.
- Si efectivamente $\mathcal{P} \neq \mathcal{NP}$, nunca encontraremos soluciones polinómicas al problema de las sumas de subconjuntos (y otros problemas prácticos que están en $\mathcal{NP}$), porque sabremos que no existen (veremos por qué).
- Si $\mathcal{P} = \mathcal{NP}$, existirían soluciones polinómicas al problema de las sumas de subconjuntos y todos los demás problemas de $\mathcal{NP}$.

3. *(2 puntos)* Mostrar el estado del siguiente vector tras cada una de las iteraciones de su ordenación mediante los algoritmos siguientes:

   a) ordenación de Shell con incrementos de Hibbard

   b) ordenación por montículos

10	4	9	7	8	5	11	3	0	6	1	12	2

4. *(1.5 puntos)* Ejemplos de aplicación del teorema de resolución de recurrencias *Divide y Vencerás*. Enuncie el teorema. A continuación, complete una tabla con las columnas siguientes:

- *caso* para el que se aplica el teorema (mejor, medio, peor, o todos)
- *caracterización* del caso (¿cuándo se produce el caso especificado en la columna anterior?)
- *ecuación* de recurrencia (de la forma $T(n) = lT(n/b) + cn^k, n > n_0$)
- *resultado* de la aplicación del teorema

En cada línea de la tabla se considerará un algoritmo diferente:

a) *bb*: búsqueda binaria
b) *mergesort*: ordenación por fusión
c) *quicksort*: ordenación rápida

Nombre	Caso	Caracterizacion	Ecuacion	Resultado
BB	Peor caso	Cuando el elemento que buscamos no esta en el arbol o cuando ese elemento esta escondido en el ultimo buscado.	T(n) = T(n/2)+1	Z(log n)
MS	Todos	Cualquier caso	T(n) = 2*T(n/2) + O(n), n>1	Z(n*log n)
QS	Mejor	Cuando el pivote coincide sistematicamente con la mediana	T(n) = 2*T(n/2) + O(n), n>1	Z(n*log n)

3. *(1 punto)* Análisis de la complejidad de la búsqueda binaria: mejor caso y peor caso, aplicando el teorema de resolución de recurrencias divide y vencerás.

Peor caso:

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

Resolver recurrencia (Cf. seminario) $\rightarrow T(n) = \Theta(logn)$

El peor caso es que el elemento que buscas este situado en la posición anterior o posterior del elemento del medio de tu vector (de ahí sale T(n/2)) más la busqueda de ese elemento (1).

L=1,b=2,k=0;

1=2^0 --> l=b^k

Entonces O(log n)

Mejor caso : (no estoy seguro de su demostración ya que no viene en traspas)

El mejor caso es O(1) porque va al elemento del medio

T(n)=1 si n>0

L=0,k=0,b=x;

0<x^0 -->> l<b^k ; 0 es menor que 1 (todo lo elevado a 0 es 1)

0<1;

Entonces aplicamos O(n^k)--> O(n^0) --> O(1)

Pregunta 1
Ainda non respondido
Puntuado fóra de 1,00
Marcar a pregunta

Utilizando la técnica de programación dinámica, escriba el pseudocódigo de un algoritmo que permita calcular **Fibonacci(n)** en tiempo estrictamente lineal y espacio constante.

Fonte Tamaño Párrafo

```
función fib2(n): valor
    i := 1; j := 0;
    para k := 1 hasta n hacer
        j := i+j; i := j-i
    fin para;
    devolver j
fin función
```

- $\boxed{T(n) = \Theta(n)}$ y espacio en $\Theta(1)$

Pregunta 2
Ainda non respondido
Puntuado fóra de 1,50
Marcar a pregunta

Con la siguiente función de dispersión:

hash("a") = 8
hash("b") = 9
hash("c") = 8
hash("d") = 9
hash("e") = 8
hash("f") = 9

mostrar el resultado de insertar las claves: "a", "b", "c", "d", "e" y "f" en la siguiente tabla:

0	1	2	3	4	5	6	7	8	9	10

cuando se emplea:

a) exploración lineal

b) exploración cuadrática

c) exploración doble usando 7-(x mod 7) como segunda función (siendo x el resultado de la primera función de dispersión).

Indique también el número total de colisiones que se produce durante las inserciones en cada una de las tres exploraciones.

0	1	2	3	4	5	6	7	8	9	10	
d	e	f						a	b	c	12 cols
	c	f				e		a	b	d	8 cols
		d	c	e			f	a	b		12 cols

Determine de forma empírica la complejidad_computacional de un algoritmo a partir de la siguiente tabla de tiempos:

Determine empirically the computational_complexity of an algorithm from the following table of execution times:

n	t(n)
1000	0.42
2000	0.95
4000	2.24
8000	5.09
16000	11.41
32000	25.63
64000	57.48

A medida que se aumenta el tamaño (n) del vector, su tiempo t(n) también aumenta con la misma relacion. Si el tamaño se dobla, el tiempo también. Es decir, su complejidad computacional es lineal (*2), osea O(n).