FEBRERO 2018 – SEMANA 1 (MODELO A)

Grado en Ingeniería Informática y Grado en Ingeniería en Tecnologías de la Información

Normas de valoración del examen:

- La nota del examen representa el 80% de la valoración final de la asignatura (el 20% restante corresponde a las prácticas).
- Cada cuestión contestada correctamente vale 1 punto.
- Cada cuestión contestada incorrectamente baja la nota en 0.3 puntos.
- Debe obtenerse un mínimo de 3 puntos en las cuestiones para que el problema sea valorado (con 3 cuestiones correctas y alguna incorrecta el examen está suspenso).
- La nota total del examen debe ser al menos de 4.5 para aprobar.
- **Las cuestiones se responden en una hoja de lectura óptica**.

Examen tipo A:

Cuestiones:

1. Un dentista pretende dar servicio a n pacientes y conoce el tiempo requerido por cada uno de ellos, siendo t_i , $i= 1,2,\ldots,n$, el tiempo requerido por el paciente i. El objetivo es minimizar el tiempo total que todos los clientes están en el sistema, y como el número de pacientes es fijo, minimizar la espera total equivale a minimizar la espera media. ¿Cuál de los siguientes esquemas es más eficiente de los que puedan resolver el problema correctamente?

   (a) Esquema voraz.
   (b) Esquema de programación dinámica.
   (c) Esquema de vuelta atrás.
   (d) Esquema de ramificación y poda.

   El esquema más apropiado es el Esquema o Algoritmo Voraz, con el que se resuelve problemas de minimización de tiempo en el sistema.

Respuesta A)

2. Respecto a la estructura de datos montículo de mínimos, cuál de las siguientes afirmaciones es **falsa**:

   (a) Con un montículo de mínimos disponemos de una estructura de datos en la que encontrar el mínimo es una operación de coste constante.
   (b) El montículo sirve de apoyo a la creación de un algoritmo de ordenación eficiente conocido como Heapsort.
   (c) El montículo es un árbol binario que puede estar o no balanceado.
   (d) Cuando se extrae el primer elemento de un montículo, restaurar la propiedad de montículo tiene coste O(log n).

   El concepto del montículo (que es siempre un árbol binario que está balanceado), desde el punto de vista de estructura de datos, fue desarrollado ad hoc como apoyo a la creación de un algoritmo de ordenación eficiente, conocido como Heapsort.

   Con el montículo disponemos de una estructura de datos en la que encontrar el mínimo (o el máximo) es una operación de coste constante.

   Una vez extraído el primer elemento, restaurar la propiedad de montículo tiene un coste O(log n). Sin pérdida de generalidad, asumimos que vamos a usar montículos de máximos.

Respuesta C)

3. Se dispone de un vector, V, que almacena números enteros en orden estrictamente creciente, y se desea averiguar si existe algún elemento que cumpla V[i]=i. ¿Cuál sería la estrategia más adecuada para resolver el problema?

(a) Algoritmo voraz.
(b) Divide y vencerás.
(c) Vuelta atrás
(d) Ramificación y poda

El problema del que se habla en el enunciado se trata de la búsqueda binaria en un vector ordenado, donde dado un vector v[1…n] de elementos ordenados, se puede verificar la pertenencia de un valor *x* a dicho vector mediante un algoritmo que emplea la estrategia Divide y Vencerás.

Respuesta B)

4. Sobre al algoritmo de Dijkstra, cuál de las siguientes afirmaciones es **falsa**:

(a) El algoritmo de Dijkstra sigue el esquema voraz.
(b) El algoritmo de Dijkstra determina la longitud del camino de coste, peso o distancia mínima que va desde el nodo origen a cada uno de los demás nodos del grafo.
(c) El coste del algoritmo de Dijkstra es O(n log n).
(d) En el algoritmo de Dijkstra un camino desde el nodo origen hasta otro nodo es *especial* si se conoce el camino de coste mínimo desde el nodo origen a todos los nodos intermedios.

El algoritmo de Dijkstra sigue el esquema Voraz. Con él se puede calcular el camino de coste o peso mínimo entre el origen y un único nodo del grafo. Para ello habría que detener el algoritmo una vez que el camino hasta dicho nodo ya se ha calculado.

El algoritmo Dijkstra utiliza la noción de *camino especial*. Un camino desde el nodo origen hasta otro nodo es especial si todos los nodos intermedios del camino pertenecen a S, es decir, se conoce el camino mínimo desde el origen a cada uno de ellos.

Hará falta un array *especial[]* que en cada paso del algoritmo contendrá la longitud del camino especial más corto (si el nodo está en S), o el camino más corto conocido (si el nodo está en C) que va desde el origen hasta cada nodo del grafo.

Cuando se va a añadir un nodo a S, el camino especial más corto hasta ese nodo es también el más corto de todos los caminos posibles hasta él. Cuando finaliza el algoritmo todos los nodos están en S y, por lo tanto, todos los caminos desde el origen son caminos especiales.

Las longitudes o distancias mínimas que se esperan calcular con el algoritmo estarán almacenadas en el array *especial[]*.

El coste del algoritmo Dijkstra será:

- O(n) → Tareas de inicialización
- $O(n^2)$ → Tiempo de ejecución
- O(log n) → Instrucción que elimina la raíz del montículo
- O(log n) → Coste cuando se actualiza el montículo con w ubicándolo según su valor de especial[w] cuando se encuentra un nodo w tal que, a través de v, se encuentra un camino menos costoso.
- O((n+a) log n) → La raíz del montículo se elimina n-2 veces, y se realizan operaciones de flotar un máximo de a veces.
- O(a log n) → Implementación con montículo en los casos:
  - Grafo Conexo: $n-1 <= a <= n^2$

- Grafo Disperso: El número de aristas es pequeño y cercano a n
- $O(n^2 \log n)$ → Se da cuando el grafo es denso, siendo preferible la implementación con montículo.

Respuesta C)

5. Indica cuál de las siguientes afirmaciones es cierta con respecto a la resolución de colisiones:

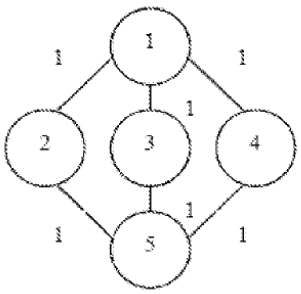
(a) El método de hashing abierto es siempre más eficiente que el hashing cerrado para la resolución de colisiones.
(b) En el método de hashing cerrado con recorrido lineal existe una probabilidad muy baja de colisiones independientemente de los patrones de las claves.
(c) En el método de hashing cerrado con recorrido mediante doble hashing, la función h y h' que se aplican deben ser iguales cuando el número de elementos de la tabla, m, cumple determinadas condiciones.
(d) El método de hashing cerrado con recorrido cuadrático permite una mayor dispersión de las colisiones por la tabla que el hashing cerrado con recorrido lineal.

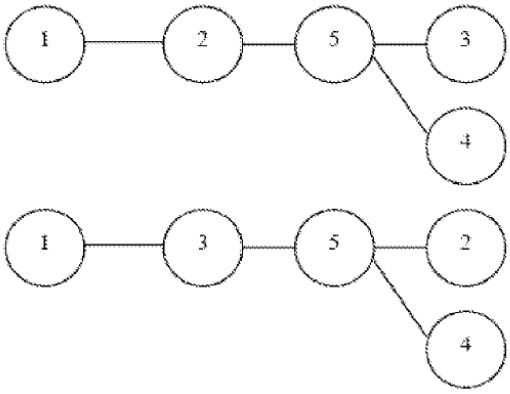
a) El Hashing Cerrado es más eficiente para la resolución de colisiones, ya que busca soluciones en la misma tabla.
b) El Hashing Cerrado permite resolver la colisión mediante la búsqueda en ubicaciones alternativas en la misma tabla, hasta que encontramos un sitio libre en la misma.
   Se debe determinar que hay un sitio libre en la tabla con la presencia de un valor que lo determine, y, si es así, se ubica el valor en la posición indicada por la función Hash.
   En este tipo de direccionamiento, a medida que la tabla se llena, las probabilidades de colisión aumentan significativamente.
   Los métodos más conocidos para resolver las colisiones son:
   - ***Recorrido Lineal:*** a la dirección obtenida por la función Hash h(k) se le añade un incremento lineal que proporciona otra dirección en la tabla.
   - ***Recorrido Cuadrático:*** En el recorrido lineal, la probabilidad de nuevas colisiones es bastante alta para determinados patrones de clave. Hay otro método basado en una expresión cuadrática g(k) que permite mayor dispersión de las colisiones por la tabla, al mismo tiempo que proporciona un recorrido completo por la misma. La exploración cuadrática es completa y con ella se recorre todos los bloques de la tabla.
   - Recorrido mediante Doble Hashing: Se utiliza una segunda función Hash h'(x)' como función auxiliar. Para que este método funcione, la función h'(x) debe cumplir determinadas condiciones:
     - $h'(x) \neq 0$, ya que de lo contrario se producen colisiones
     - La función h' debe ser diferente de la función h
     - Los valores de h'(x) deben ser primos relativos de M (que no comparten factores primos) para que los índices de la tabla se ocupen en su totalidad. Si M es primo, cualquier función puede ser usada como h'.
c) Explicación dada en el apartado anterior. Es falso, ya que h y h' deben ser distintas.
d) Explicación dada en el apartado B. Es cierto.

Respuesta D)

6. Selecciona la afirmación más ajustada de las siguientes. Las siguientes tres figuras corresponden a:

   (a) Las tres corresponden a tres grafos no dirigidos sin ninguna posible relación entre ellos.
   (b) La de la izquierda es un grafo y a los grafos no dirigidos conexos con aristas de igual coste no se les puede asociar más de un árbol de recubrimiento mínimo.
   (c) La de la izquierda es un grafo y las de la derecha son dos posibles árboles de recubrimiento mínimo asociados a él.
   (d) Ninguna de las anteriores es correcta.





Respuesta C)

Problema (4 puntos). Tenemos n objetos de volúmenes v_1... v_n, y un número ilimitado de recipientes iguales con capacidad R (con $v_i \leq R$, para todo i). Los objetos se deben meter en los recipientes sin partirlos, y sin superar su capacidad máxima. Se busca el mínimo número de recipientes necesarios para colocar todos los objetos.

La resolución de este problema debe incluir, por este orden:

1. Elección razonada del esquema más apropiado de entre los siguientes: Voraz, Divide y Vencerás, Vuelta atrás o Ramificación y Poda.
   Escriba la estructura general de dicho esquema e indique como se aplica al problema (0,5 puntos).

   El esquema más apropiado es el de Ramificación y Poda. El esquema general se encuentra en la página 187 del Libro de Texto Base de la asignatura, el cual es:

```
fun RamificacionYPoda (nodoRaiz, mejorSolución: TNodo, cota: real)
    monticulo = CrearMonticuloVacio()
    cota = EstimacionPes(nodoRaiz)
    Insertar(nodoRaiz, monticulo)
    mientras ¬ MonticuloVacio?(monticulo) ∧
              EstimacionOpt(Primero(monticulo)) ≤ cota hacer
        nodo ← ObtenerCima(monticulo)
        para cada hijo extensión válida de nodo hacer
            si solución(hijo) entonces
                si coste(hijo) ≤ cota entonces
                    cota ← coste(hijo)
                    mejorSolucion ← hijo
                fsi
            sino
                si EstimacionOpt(hijo) ≤ cota entonces
                    Insertar(hijo, monticulo)
                    si EstimacionPes(hijo) < cota entonces
                        cota ← EstimacionPes(hijo)
                    fsi
                fsi
            fsi
        fpara
    fmientras
ffun
```

Recordemos que en la solución de Vuelta Atrás se supone que todo objeto cabe en un envase vacío y, por tanto, se necesitan un máximo de n envases.

Las soluciones se representan en tuplas de la forma (x1, . . . , xn), donde xi es el envase donde hemos colocado el objeto i. Como los envases son indistinguibles, el primer objeto siempre se coloca en el primer envase (x1 = 1) y para cada objeto de los restantes se puede usar uno de los envases ya ocupados, si cabe en alguno, o coger uno vacío.

2. Descripción de las estructuras de datos necesarias (0,5 puntos solo si el punto 1 es correcto).

Podemos representar el reparto de objetos entre recipientes mediante un vector en el que cada posición indique a que recipiente se ha asignado cada objeto: *objetos = vector[1..n] de entero*

La solución es la cantidad entera S de recipientes empleados.

Utilizamos un montículo de mínimos en el que cada componente almacene una solución parcial (nodo) con su cota correspondiente:

```
nodo = tupla
        asignaciones: vector[1..N] de enteros
        etapa: cardinal          // objeto por el que se va procesando
        num-recip: cardinal      // num de recipientes
        capacidad[1..N] de real // capacidades disponibles de cada envase utilizado
```

3. Algoritmo completo a partir del refinamiento del esquema general (2.5 puntos sólo si el punto 1 es correcto). Si se trata del esquema voraz debe hacerse la demostración de optimalidad.

Para el desarrollo se utilizan las siguientes estimaciones:

- Cota Inferior: número de envases ya utilizados en la solución parcial.
- Cota Superior: Se hace una estimación metiendo cada objeto de los pendientes en el primer recipiente en el que quepa. Si no cabe en ninguno, se añade un nuevo envase a la solución parcial.

En cuanto a las estimaciones a utilizar para el esquema optimista-pesimista, tenemos:

- optimista: una cota inferior adecuada es el número de envases ya utilizados en la solución parcial.
- pesimista: Una cota superior muy sencilla es considerar un envase extra por cada objeto que nos queda por empaquetar, pero resulta demasiado grosera. Podemos, en cambio, ir considerando cada objeto restante, en el orden en que se haya dado, e intentar meterlo en el primer envase utilizado y, en el caso en que no quepa, intentarlo con el segundo envase, y así hasta agotar todas las posibilidades, en cuyo caso, se añadirá un nuevo envase a la solución parcial.

En cada nodo, además de la información usada (solución parcial, etapa y prioridad), guardamos el número de envases utilizados y las capacidades disponibles de cada envase utilizado.

```
tipos
        nodo = reg
                        sol[1..n] de 1..n
                        k : 1..n
                        envases : 1..n    { prioridad }
```

capacidad[1..n] de real

freg

ftipos

Además, como en la solución mediante la técnica de vuelta atrás (véase el ejercicio 14.22), la búsqueda podrá acabarse si encontramos una solución con el valor

$$\textit{óptimo} = \left\lceil \frac{\sum_{i=1}^{n} v_i}{E} \right\rceil .$$

```
{ ∀i : 1 ≤ i ≤ n : V[i] ≤ E }
fun empaquetar-rp(E : real+, V[1..n] de real+) dev ⟨sol-mejor[1..n] de 1..n, envases-mejor : 1..n⟩
var X, Y : nodo, C : colapr[nodo]
    total := 0
    para i = 1 hasta n hacer total := total + V[i] fpara
    óptimo := ⌈total/E⌉ ; encontrada := falso
    { generamos la raíz: el primer objeto en el primer envase }
    Y.k := 1 ; Y.sol[1] := 1 ; Y.envases := 1
    Y.capacidad[1] := E − V[1] ; Y.capacidad[2..n] := [E]
    C := cp-vacía() ; añadir(C, Y)
    envases-mejor := cálculo-pesimista(E, V, Y)
    mientras ¬encontrada ∧ ¬es-cp-vacía?(C) ∧ mínimo(C).envases ≤ envases-mejor hacer
        Y := mínimo(C) ; eliminar-mín(C)
        { generamos los hijos de Y }
        X.k := Y.k + 1 ; X.sol := Y.sol
        X.envases := Y.envases ; X.capacidad := Y.capacidad
        { probamos con cada envase ya utilizado }
        i := 1
        mientras i ≤ Y.envases ∧ ¬encontrada hacer
            si X.capacidad[i] ≥ V[X.k] entonces
                X.sol[X.k] := i ; X.capacidad[i] := X.capacidad[i] − V[X.k]
                si X.k = n entonces
                    sol-mejor := X.sol ; envases-mejor := X.envases
                    encontrada := (envases-mejor = óptimo)     { terminar }
                si no
                    añadir(C, X)
                    pes := cálculo-pesimista(E, V, X)
                    envases-mejor := mín(envases-mejor, pes)
                fsi
                X.capacidad[i] := Y.capacidad[i]
            fsi
            i := i + 1
        fmientras
        si ¬encontrada entonces      { probamos con un envase nuevo }
            nuevo := Y.envases + 1
            X.sol[X.k] := nuevo ; X.envases := nuevo
            X.capacidad[nuevo] := E − V[X.k]
            si X.envases ≤ envases-mejor entonces
                si X.k = n entonces
                    sol-mejor := X.sol ; envases-mejor := nuevo
                    encontrada := (envases-mejor = óptimo)     { terminar }
                si no
                    añadir(C, X)
                    pes := cálculo-pesimista(E, V, X)
                    envases-mejor := mín(envases-mejor, pes)
                fsi
            fsi
        fsi
    fmientras
ffun
```

Para calcular las estimaciones usamos la siguiente función:

```
fun cálculo-pesimista(E : real+, V[1..n] de real+, X : nodo) dev pes : 1..n
var capacidad-aux[1..n] de real
    pes := X.envases ; capacidad-aux := X.capacidad
    para i = X.k + 1 hasta n hacer
        j := 1
        mientras V[i] > capacidad-aux[j] hacer j := j + 1 fmientras
        capacidad-aux[j] := capacidad-aux[j] − V[i]
        pes := max(pes, j)
    fpara
ffun
```

4. Estudio del coste del algoritmo desarrollado (0,5 puntos solo si el punto 1 es correcto).

   El número de recipientes está limitado a n, es decir, al número de objetos. Una estimación del coste es el tamaño del árbol, que en el peor caso crece como O(n!), ya que cada nodo del nivel k puede expandirse con los n-k objetos que quedan por asignar a recipientes.