Examen 4 Enero 2016, preguntas y respuestas

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

Primer Parcial de EDA - 18 de Abril de 2016 - Duración: 2h 30' - Puntuación 3

APELLIDOS, NOMBRE	GRUPO

1.- La Agencia Nacional de Evaluación de la Calidad y Acreditación (ANECA) ha decidido evaluar a los profesores que quieran optar a una promoción en base al impacto de su labor investigadora. A cada profesor se le asocia entonces un índice de impacto (`int`). La ANECA solo deja optar a la promoción a aquellos profesores cuyo índice de impacto sea mayor que un cierto umbral `u` (`int`).
Se dispone de un array `h` en el que `h[i]` es el índice de impacto del profesor `i`. Este array está ordenado de manera no decreciente por índice de impacto; específicamente, `h[i]` ≤ `h[i+1]` (0 ≤ `i` < `h.length-1`). Se pide:

(a) Diseña un método estático Divide y Vencerás que, dado el array ordenado `h` y el umbral `u`, devuelva el número de profesores cuyo índice de impacto es mayor que el umbral `u`. **(0.75 puntos)**

```
public static int promocion(int h[], int u) {
      return promocion(h, u, 0, h.length - 1);
}

private static int promocion(int h[], int u, int ini, int fin) {
  if (ini > fin) return 0;
  else {
    int mitad = (ini + fin) /2;
    if (h[mitad] <= u) return promocion(h, u, mitad + 1, fin);
    // h[mitad] > u --> todos los profesores con índice > mitad también tienen índice de impacto mayor que u
    return promocion(h, u, ini, mitad - 1) + (fin - mitad + 1);
  }
}
```

(b) Estudia el coste del método diseñado **(0.25 puntos)**

Talla: `n = fin - ini + 1`

Instancias significativas: `no hay`

Ecuaciones de recurrencia:

$T_{promocion}(n = 0) = k1$

$T_{promocion}(n > 0) = 1 * T_{promocion}(n/2) + k2$

Coste temporal asintótico:

$T_{promocion}(n) \in \Theta(\log_2 n)$

2.- Dos jugadores se enfrentan en una partida de un videojuego desde lugares distintos, usando cada uno de ellos una pantalla física distinta. De todas las posiciones de la pantalla que cada jugador visita durante la partida solo algunas le proporcionan un cierto número de puntos, en función del momento en el que las visita. Para guardar dichas posiciones y los puntos en ellas obtenidas, el videojuego dispone de un `Map<Posicion, Integer>`; en el apartado (a) de este ejercicio se darán los detalles de la clase `Posicion`.

Así mismo, para establecer cuál de los dos jugadores ha ganado la partida, el videojuego almacena en la `ListaConPI<Posicion> optima` la secuencia temporal de posiciones a visitar para obtener la máxima puntuación; de esta forma, gana la partida aquel jugador que haya acumulado un mayor número de puntos. Es importante señalar lo siguiente:

- El ganador no obtiene la máxima puntuación si no ha seguido la secuencia temporal óptima de visita.
- Cuando un jugador no ha visitado una posición de la lista `optima`, sus puntos se decrementan en una cantidad fija `penalty`.
- Dos jugadores pueden empatar si ambos obtienen la misma puntuación tras la partida.

(a) Si `Map<Posicion, Integer>` se implementa mediante una Tabla Hash Enlazada, completa el cuerpo de la siguiente clase con los métodos imprescindibles para que pueda ser la clase de la clave de dicho Map. **(0.3 puntos)**

```
public class Posicion {
    private int x, y;
    public Posicion(int vX, int vY) { x = vX; y = vY; }
    public int getX() { return x; }
    public int getY() { return y; }

    /*COMPLETAR*/

    public boolean equals(Object otra) {
        return otra instanceof Posicion
               && this.x == ((Posicion) otra).x
               && this.y == ((Posicion) otra).y;
    }

    public int hashCode() {
        return (x + "," + y).hashCode();
    }


}
```

(b) Completa el cuerpo del siguiente método que, dados los Map con los pares (posición, puntos) obtenidos por los jugadores 1 y 2 durante una partida, `mJ1` y `mJ2`, un `penalty` y la lista `optima`, devuelva el número del jugador (1 o 2) que ha ganado la partida, o 0 si han empatado. **(0.7 puntos)**

```
public int ganador(Map<Posicion, Integer> mJ1, Map<Posicion, Integer> mJ2,
                   int penalty, ListaConPI<Posicion> optima) {

    int puntosJ1 = 0, puntosJ2 = 0;
    for (optima.inicio(); !optima.esFin(); optima.siguiente()) {
        Posicion p = optima.recuperar();
        Integer aux = mJ1.recuperar(p);
        if (aux != null) { puntosJ1 += aux; }
        else { puntosJ1 -= penalty;}
        aux = mJ2.recuperar(p);
        if (aux != null) { puntosJ2 += aux; }
        else { puntosJ2 -= penalty;}
    }
    if (puntosJ1 == puntosJ2) { return 0; }
    else if (puntosJ1 > puntosJ2) { return 1; }
    else { return 2; }

}
```

3.- En un ABB en el que se admiten elementos repetidos, los datos del subárbol izquierdo de un nodo son menores o iguales que el que contiene dicho nodo, mientras que los de su subárbol derecho son siempre mayores. Se pide:

(a) Diseña en la clase `ABB` un método que devuelva el número de veces que aparece un elemento dado `e` en un ABB. **(0.75 puntos)**

```
public int numApariciones(E e) {
  return numApariciones(e, raiz);
}

private int numApariciones(E e, NodoABB<E> actual) {
  int res = 0;
  if (actual != null) {
    int cmp = actual.dato.compareTo(e);
    if (cmp > 0) { res = numApariciones(e, actual.izq); }
    else if (cmp < 0) { res = numApariciones(e, actual.der); }
    else { res = 1 + numApariciones(e, actual.izq); }
  }
  return res;
}
```

(b) Estudia el coste del método diseñado para un ABB Equilibrado. **(0.25 puntos)**

Talla: n = tamaño del nodo actual

Instancias significativas: No hay

Ecuaciones de recurrencia:

$T_{numApariciones}(n = 0) = k1$

$T_{numApariciones}(n > 0) = 1 * T_{numApariciones}(n/2) + k2$

Coste temporal asintótico:

$T_{numApariciones}(n) \in \Theta(\log_2 n)$

ANEXO

Las clases `ListaConPI`, `Map` y `ABB` del paquete `modelos`.

```
public interface ListaConPI<E> {
  void insertar(E e);
  /** SII !esFin() */ void eliminar();
  void inicio();
  /** SII !esFin() */ void siguiente();
  void fin();
  /** SII !esFin() */ E recuperar();
  boolean esFin();
  boolean esVacia();
  int talla();
}
```

```
public interface Map<C, V> {
       V insertar(C c, V v);
       V eliminar(C c);
       V recuperar(C c);
       boolean esVacio();
       int talla();
       ListaConPI<C> claves();
}
```

```
public class ABB<E extends Comparable<E>> {
       protected NodoABB<E> raiz;
       public ABB() {…}
       public boolean esVacio() {…}
       public int talla() {…}
       public E recuperar(E e) {…}
       public void insertar(E e) {…}
       public void eliminar(E e) {…}
       public E eliminarMin() {…}
       public E recuperarMin() {…}
       public E eliminarMax() {…}
       public E recuperarMax() {…}
       public E sucesor(E e) {…}
       public E predecesor(E e) {…}
       public String toStringInOrden() {…}
       public String toStringPreOrden() {…}
       public String toStringPostOrden() {…}
       public String toStringPorNiveles() {…}
       …
}
```

Teoremas de coste

Teorema 1: $f(x) = a \cdot f(x - c) + b$, con $b \geq 1$

- si $a=1$, $f(x) \in \Theta(x)$;
- si $a>1$, $f(x) \in \Theta(a^{x/c})$;

Teorema 2: $f(x) = a \cdot f(x - c) + b \cdot x + d$, con b y $d \geq 1$

- si $a=1$, $f(x) \in \Theta(x^2)$;
- si $a>1$, $f(x) \in \Theta(a^{x/c})$;

Teorema 3: $f(x) = a \cdot f(x/c) + b$, con $b \geq 1$

- si $a=1$, $f(x) \in \Theta(\log_c x)$;
- si $a>1$, $f(x) \in \Theta(x^{\log_c a})$;

Teorema 4: $f(x) = a \cdot f(x/c) + b \cdot x + d$, con b y $d \geq 1$

- si $a<c$, $f(x) \in \Theta(x)$;
- si $a=c$, $f(x) \in \Theta(x \cdot \log_c x)$;
- si $a>c$, $f(x) \in \Theta(x^{\log_c a})$;

Teoremas maestros

Teorema para recurrencia divisora: la solución a la ecuación $T(n) = a \cdot T(n/b) + \Theta(n^k)$, con $a \geq 1$ y $b>1$ es:

- $T(n) = O(n^{\log b^a})$ si $a>b^k$;
- $T(n) = O(n^k \cdot \log n)$ si $a=b^k$;
- $T(n) = O(n^k)$ si $a<b^k$;

Teorema para recurrencia sustractora: la solución a la ecuación $T(n) = a \cdot T(n-c) + \Theta(n^k)$ es:

- $T(n) = \Theta(n^k)$ si $a<1$;
- $T(n) = \Theta(n^{k+1})$ si $a=1$;
- $T(n) = \Theta(a^{n/c})$ si $a>1$;