Ejercicio de seminario - tema 2 divide y vencerás resueltos

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

TEMA 2
Divide y vencerás

EJERCICIOS RESUELTOS

Ejercicio 1.- Diseña un método recursivo que permita comparar dos *arrays* genéricos y analiza su coste:

```
public static <T> boolean comparar(T a[], T b[]) { … }
```

Observación: dos *arrays* se consideran iguales si tienen los mismos elementos dispuestos en el mismo orden.

Solución:

```
public static <T> boolean comparar(T a[], T b[]) {
  if (a.length != b.length) return false;
  return comparar(a, b, 0);
}

private static <T> boolean comparar(T a[], T b[], int izq) {
  if (izq < a.length) {
    if (!a[izq].equals(b[izq])) return false;
    return comparar(a, b, izq + 1);
  } else return true;
}
```

Análisis de coste:

- Talla: $N = a.length - izq$ (*a.length* en la llamada más alta)

- Hay instancias significativas:
    - Mejor caso: el primer elemento de *a* es distinto del primer elemento de *b*
    - Peor caso: los arrays *a* y *b* son iguales (tienen los mismos elementos en el mismo orden)

- Ecuaciones de recurrencia:
  $\text{Tcomparar}^{M}(N) = k_1$
  $\text{Tcomparar}^{P}(N = 0) = k_2$
  $\text{Tcomparar}^{P}(N > 0) = \text{Tcomparar}^{P}(N\text{-}1) + k_3$

- Coste asintótico:
  $\text{Tcomparar}(N) \in \Omega(1)$
  $\text{Tcomparar}(N) \in O(N)$, aplicando el Teorema 1 con a=c=1

Ejercicio 2.- Dado el siguiente método:

```
// v ordenado ascendentemente sin elementos repetidos, x < y
public static boolean buscaPar(Integer[] v, Integer x,
       Integer y, int izq, int der) {
  if (izq >= der) return false;
  int mitad = (izq + der) / 2;
  int comp = v[mitad].compareTo(x);
  if (comp == 0) return v[mitad+1].compareTo(y) == 0;
  if (comp < 0) return buscaPar(v, x, y, mitad+1, der);
  return buscaPar(v, x, y, izq, mitad);
}
```

a) Describir qué problema resuelve *buscaPar*, detallando el significado de cada uno de sus parámetros.
b) Calcular la complejidad temporal del método *buscaPar*.

Solución:

a) El método *buscaPar* realiza una búsqueda sobre el vector *v* para comprobar si el par de Integer *x* e *y* ocupa o no posiciones consecutivas dentro del vector.

- Los parámetros *izq* y *der* marcan el intervalo de búsqueda.
- El método devuelve *true* si *x* e *y* son contiguos en *v*[*izq*..*der*] y *false* en caso contrario: no se encuentra *x* o están pero no son contiguos.

b) Talla: $N = der - izq + 1$

- Caso mejor: *x* se encuentra en la mitad del primer intervalo de búsqueda e *y* está a continuación
  $T_{buscarPar}^{M}(N) = k_1 \rightarrow T_{buscarPar}(N) \in \Omega(1)$

- Caso peor: *x* no se encuentra en el vector
  $T_{buscarPar}^{P}(N \leq 1) = k_2$
  $T_{buscarPar}^{P}(N > 1) = T_{buscarPar}^{P}(N / 2) + k_3 \rightarrow T_{buscarPar}(N) \in O(\log_2 N)$

Ejercicio 3.- Diseña un método recursivo genérico que determine si un *array* dado es capicúa:

```
public static <T> boolean esCapicua(T[] v) { … }
```

Indica qué tipo de método recursivo es y analiza su coste.

Solución:

```
public static <T> boolean esCapicua(T[] v) {
  return esCapicua(v, 0, v.length - 1);
}

private static <T> boolean esCapicua(T[] v, int ini, int fin) {
  if (ini < fin) {
    if (!v[ini].equals(v[fin])) return false;
    return esCapicua(v, ini + 1, fin - 1);
  } else return true;
}
```

El método recursivo es lineal final.

- Talla: $N = \text{fin} - \text{ini} + 1$ (*v.length* en la llamada más alta)
- Hay instancias significativas:
  - Mejor caso: el primer elemento de *v* y el último son distintos
  - Pero caso: *v* es capicúa
- Ecuaciones de recurrencia:
  $T_{esCapicua}^{M}(N) = k_1$
  $T_{esCapicua}^{P}(N \leq 1) = k_2$
  $T_{esCapicua}^{P}(N > 1) = T_{esCapicua}^{P}(N\text{-}2) + k_3$
- Coste asintótico:
  $T_{esCapicua}(N) \in \Omega(1)$
  $T_{esCapicua}(N) \in O(N)$, aplicando el Teorema 1 con a=1 y c=2

Ejercicio 4.- Diseña una función recursiva que devuelva el máximo de un *array* genérico y analiza su coste.

Solución:

```
public static <T extends Comparable<T>> T maximo(T v[]) {
  return maximo(v, 0);
}

private static <T extends Comparable<T>> T maximo(T v[], int inicio) {
  if (inicio == v.length) return null;
  else {
    T max = maximo(v, inicio + 1);
    if (max == null || v[inicio].compareTo(max) > 0)
       max = v[inicio];
    return max;
  }
}
```

- Talla: N = v.length – inicio (*v.length* en la llamada más alta)
- Hay instancias significativas: no hay, se trata de un recorrido
- Ecuaciones de recurrencia:
  Tmaximo($N = 0$) = k_1
  Tmaximo($N > 0$) = Tmaximo(N-1) + k_2
- Coste asintótico:
  Tmaximo (N) $\in \Theta(N)$, aplicando el Teorema 1 con a=1 y c=2

Ejercicio 5.- Analiza el coste de los siguientes métodos:

```
private static int sumar1(int v[], int ini, int fin) {
   int suma = 0;
   if ( ini == fin ) suma = v[ini];
   if ( ini < fin ) {
      suma = v[ini] + v[fin];
      suma += sumar1(v, ini+1, fin-1);
   }
   return suma;
}

private static int sumar2(int v[], int ini, int fin) {
   int suma = 0;
   if ( ini == fin ) suma = v[ini];
   if ( ini < fin ) {
     int mitad = (fin + ini) / 2;
     suma = sumar2(v,ini,mitad) + sumar2(v,mitad+1,fin);
   }
   return suma;
}
```

Solución:

a) Talla del problema: $x = fin - ini + 1$

- No hay instancias significativas pues hay que recorrer todo el vector en cualquier caso
- Ecuaciones de recurrencia:

  $T_{sumar1}(x \leq 1) = k$

  $T_{sumar1}(x > 1) = 1 * T_{sumar1}(x - 2) + k$
- Para la última ecuación aplicamos el teorema 1, con a=1 y c=2:
  Coste asintótico del método:

  $T_{sumar1}(x) \in \Theta(x)$

b) Talla del problema: $x = fin - ini + 1$

- No hay instancias significativas pues hay que recorrer todo el vector en cualquier caso
- Ecuaciones de recurrencia:

  $T_{sumar2}(x \leq 1) = k$

  $T_{sumar2}(x > 1) = 2 * T_{sumar2}(x / 2) + k$
- Para la última ecuación aplicamos el teorema 3, con a=2 y c=2:
  Coste asintótico del método:

  $T_{sumar2}(x) \in \Theta(x)$

Ejercicio 6.- Sea v un vector de componentes *Integer* positivas que se ajustan al perfil de una curva cóncava, es decir, que existe una única posición *k* en el vector tal que:

- Los elementos a la izquierda de *k* están ordenados descendentemente
- Los elementos a la derecha de *k* están ordenados ascendentemente

Ejemplo:

			k						
4	3	2	1	2	3	4	5	6	7

Diseñar el método recursivo que más eficientemente determine dicha posición *k*. Indica las instancias significativas y analiza el coste del método.

Solución:

```
public static int buscarPosK(Integer v[]) {
  return buscarPosK(v, 0, v.length - 1);
}

private static int buscarPosK(Integer v[], int inicio, int fin) {
  if (inicio > fin) return -1;
  else {
      int resAnt = 1, resSig = 1, mitad = (inicio + fin) / 2;
      if (mitad > inicio) resAnt = v[mitad-1].compareTo(v[mitad]);
      if (mitad < fin) resSig = v[mitad+1].compareTo(v[mitad]);
      if (resAnt < 0 && resSig > 0)
         return buscarPosK(v, inicio, mitad - 1);
      else if (resAnt > 0 && resSig < 0)
         return buscarPosK(v, mitad + 1, fin);
      else return mitad;
}
```

Talla del problema:

- N = fin – inicio + 1
- En la llamada más alta: N = v.length

Instancias significativas:

- *Mejor caso*: la posición *k* se encuentra en la posición (inicio + fin) / 2.
- *Peor caso*: el vector *v* está totalmente ordenado. Por lo tanto, *k* coincide con uno de los extremos del vector

Ecuaciones de recurrencia:

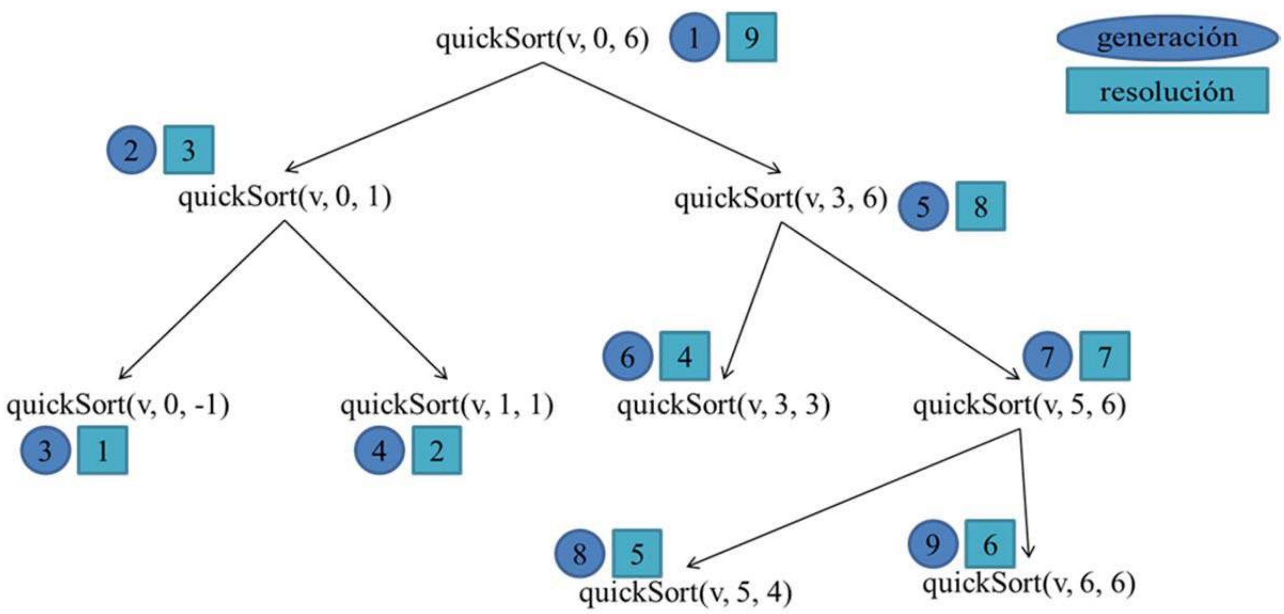
- Mejor caso:
  $T_{buscarPosK}^{M}(N) = k$
- Peor caso:
  $T_{buscarPosK}^{P}(N{=}0) = k$
  $T_{buscarPosK}^{P}(N{>}0) = 1 * T_{buscarPosK}^{P}(N/2) + k$

Coste:

- $T_{buscarPosK}(N) \in \Omega(1)$
- $T_{buscarPosK}(N) \in O(\log_2 N)$, aplicando el teorema 3 con a=1 y c=2.

Ejercicio 7.- Realiza una traza completa en árbol de las llamadas recursivas que genera *quickSort* para el array $v = \{8, 12, 6, 9, 18, 15, 1\}$, indicando el orden en el que se generan y se resuelven.

<u>Solución:</u>



Ejercicio 8.- Realiza una traza de *mergeSort* para el array {3, 41, 52, 26, 38, 57, 9, 49}.

Solución:

```
mergeSort(v, 0, 7)
    mergeSort(v, 0, 3)
        mergeSort(v, 0, 1)
            mergeSort(v, 0, 0)
            mergeSort(v, 1, 1)
            mezclaNatural(v, 0, 1, 1)
        mergeSort(v, 2, 3)
            mergeSort(v, 2, 2)
            mergeSort(v, 3, 3)
            mezclaNatural(v, 2, 3, 3)
        mezclaNatural(v, 0, 2, 3)
    mergeSort(v, 4, 7)
        mergeSort(v, 4, 5)
            mergeSort(v, 4, 4)
            mergeSort(v, 5, 5)
            mezclaNatural(v, 4, 5, 5)
        mergeSort(v, 6, 7)
            mergeSort(v, 6, 6)
            mergeSort(v, 7, 7)
            mezclaNatural(v, 6, 7, 7)
        mezclaNatural(v, 4, 6, 7)
    mezclaNatural(v, 0, 4, 7)
```

0	1	2	3	4	5	6	7
3	41	52	26	38	57	9	49

0	1	2	3	4	5	6	7
3	41	52	26	38	57	9	49

0	1	2	3	4	5	6	7
3	41	26	52	38	57	9	49
3	26	41	52	38	57	9	49

0	1	2	3	4	5	6	7
3	41	52	26	38	57	9	49

0	1	2	3	4	5	6	7
3	41	26	52	38	57	9	49
3	26	41	52	9	38	49	57
3	9	26	38	41	49	52	57