Soluciones Ejercicios Tema 5

Germán Moltó
gmolto@dsic.upv.es
Estructuras de Datos y Algoritmos
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Mostrar Números Ascendentemente

```
public static void numAscendentes(int i, int n){
 if (i <= n) {
     System.out.println("Numero: " + i);
     numAscendentes(i+1,n);
   }
}
```

- Si intercambiamos las dos lineas de código se obtendrán los números en orden descendente.

```
public static void numDescendentes(int i, int n){
 if (i <= n) {
     numDescendentes(i+1,n);
     System.out.println("Numero: " + i);
  }
}
```


toString Recursivo de LEG

```
public String toString(){
    return toString(primero);
}

private String toString(NodoLEG<E> aux){
    if (aux == null) return "";
    else return aux.dato.toString() + " " + toString(aux.siguiente);
}
```

- Se ha definido un método público que lanza al método privado recursivo que trabaja a partir de un Nodo de la LEG.
- Este método formaría parte de la implementación de LEG.

Suma Recursiva de un Vector (1/3)

- Estilo de programación respetando el Esquema General Recursivo

```
private static int sumarArray(int v[] , int  inicio, int fin) {
    int resMetodo, resLlamada ;
    if ( inicio > fin )  resMetodo = 0;
    else {              resLlamada = sumarArray(v, inicio + 1, fin);
                        resMetodo  = v[inicio] + resLlamada;
    }
   return resMetodo;
}
```

- Estilo de programación compacto (preferido!):

```
private static int sumarArray(int v[] , int  inicio, int fin) {
    if ( inicio > fin )  return  0;
    else    return v[inicio] + sumarArray(v, inicio + 1, fin);
}
```

Suma Recursiva de un Vector (2/3)

- Podemos prescindir del parámetro fin ya que su valor nunca cambia a lo largo de la ejecución (fin = v.length-1)

```
private static int sumarArray(int v[] , int  inicio) {
    if ( inicio == v.length ) return 0;
    else  return v[inicio] + sumarArray(v, inicio + 1);
}
```

- El método vía es un método de ayuda que permite utilizar el método general aplicado a todo el vector:

```
public static int sumarArray(int v[]) {
        return  sumarArray(v, 0);
}
```


Suma Recursiva de un Vector (3/3)

- Validación del Diseño Recursivo:
- Prueba de Terminación

  El valor del parámetro inicio se incrementa en una unidad en cada llamada, desde 0 hasta v.length, donde se alcanza el caso base y, por lo tanto, el algoritmo termina.

Inversión Recursiva de un Vector

```
private static <T> void invierte(T[] v, int inicio, int fin){
    if (inicio < fin){
        swap(v, inicio, fin);
        invierte(v, inicio + 1, fin - 1);
    }
}

private static <T> void swap(T[] v, int pos1, int pos2){
  T tmp;
  tmp = v[pos1];
  v[pos1] = v[pos2];
  v[pos2] = tmp;
}
```


Diferentes Diseños de Sumar Array (1/2)

```
public static int sumarArrayV1(int v[], int ini){
   if (ini == v.length) return 0;
   else return v[ini] + sumarArrayV1(v,ini+1); }
```

- Talla del problema:
  - v.length – ini
- Instancias significativas:
  - No hay, es un problema de recorrido.
- Ecuaciones de Recurrencia:
  - $T_{sumarArrayV1}(talla = 0) = k$
  - $T_{sumarArrayV1}(talla > 0) = 1 * T_{sumarArrayV1}(talla - 1) + k'$
- Acotamos empleando el teorema 1 con a = 1 y c = 1
  - $T_{sumarArrayV1}(talla) \in \Theta(talla)$

Diferentes Diseños de Sumar Array (2/2)

```
public static int sumarArrayV2(int v[], int ini){
    if (ini == v.length -1) return v[ini];
    else return v[ini] + sumarArrayV2(v,ini+1); }
```

- Talla del problema:
  - v.length – ini
- Instancias significativas:
  - No hay, es un problema de recorrido.
- Ecuaciones de Recurrencia:
  - $T_{sumarArrayV2}(talla = 1) = k$
  - $T_{sumarArrayV2}(talla > 1) = 1 * T_{sumarArrayV2}(talla - 1) + k'$
- Acotamos empleando el teorema 1 con a = 1 y c = 1
  - $T_{sumarArrayV2}(talla) \in \Theta(talla)$

Máximo Elemento de un Vector (1/2)

- Estrategia con recorrido descendente y sin especificar el máximo como argumento:

```
public static <T extends Comparable<T>> T maximoDes(T[] v, int i){
  if (i == 0) return v[i];
  else{
    T maximoActual = maximoDes(v, i - 1);
     if ( maximoActual.compareTo(v[i]) > 0 ) return maximoActual;
     else return v[i];
    }
}
```

- Llamada de más alto nivel: **maximoDes**(v, v.length – 1);

Máximo Elemento de un Vector (2/2)

- Coste del Algoritmo Recursivo:
- Talla del Problema (expresada en función de los parámetros):
  - Cantidad de datos a procesar: i+1
    - En la llamada más alta equivale al número de elementos del vector.
- Instancias Significativas:
  - No hay puesto que siempre se deberán procesar todas las componentes del vector.
- Ecuaciones de Recurrencia:
  - $T_{maximoDes}(talla = 1) = k'$
  - $T_{maximoDes}(talla > 1) = 1 * T_{maximoDes}(talla - 1) + k$
- Acotamos usando Teorema 1 con a = 1, c = 1
  - $T_{maximoDes}(talla) \in \Theta(talla)$

De Lista a Vector (Externo) (1/2)

- Método via o lanzadera

```
public static <T> void toArray(LEG<T> l, T v[]){
   toArray(l, 0, v);
}
```

- Método recursivo

```
private static <T> void toArray(LEG<T> l, int ini, T v[]){
      if (ini < l.talla()){
         v[ini] = l.recuperar(ini);
         toArray(l, ini+1, v);
       }
   }
```

De Lista a Vector (Externo) (2/2)

- Análisis del coste:
- Talla del Problema en función de los argumentos del método:
  - Cantidad de datos a procesar: l.talla() – i
- Instancias Significativas:
  - No hay caso mejor ni peor. Es un problema de recorrido.
- Ecuaciones de Recurrencia:
  - $T_{toArray}(talla = 0) = k$
  - $T_{toArray}(talla > 0) = 1 * T_{toArray}(talla - 1) + talla*k'$
- La sobrecarga es dependiente de la talla del problema puesto que el método recuperar de LEG tiene un coste lineal con la talla de la lista.
- Acotamos empleando el teorema 2 con a = 1 y c = 1
  - $T_{toArray}(talla) \in \Theta(talla^2)$

De Lista a Vector (Interno) (1/2)

```
public void toArray(E[] v){
    toArray(primero, v, 0);
  }

private void toArray(NodoLEG<E> aux, E[] v, int i){
  if (aux != null) {
    v[i] = aux.dato;
    toArray(aux.siguiente, v, i+1);
  }
}
```


De Lista a Vector (Interno) (2/2)

- Análisis del coste:
- Talla del Problema:
  - Cantidad de datos a procesar: nelem – i
    - En la llamada más alta corresponde con el número de elementos de la lista enlazada.
- Instancias Significativas:
  - No hay caso mejor ni caso peor puesto que es un problema de recorrido.
- Ecuaciones de Recurrencia:
  - $T_{toArray}(talla = 0) = k$
  - $T_{toArray}(talla > 0) = 1 * T_{toArray}(talla - 1) + k'$
- Acotamos empleando el teorema 1 con a = 1 y c = 1
  - $T_{toArray}(talla) \in \Theta(talla)$

Mismo Valor que Posicion

- Estrategia seguida: Análoga a la búsqueda binaria, ya que sabemos que el vector está ordenado.

```
private static int mismoValorPos(Integer[] v, int inicio, int fin){
 if ( inicio > fin ) return -1;
 else {
    int mitad = (inicio + fin) / 2 ;
    int resC = v[mitad].compareTo(new Integer(mitad));
    if (resC == 0) return mitad;
    else if (resC > 0) return mismoValorPos(v, inicio, mitad - 1);
         else return mismoValorPos(v, mitad + 1, fin);
  }
}
```

Mismo Valor que Posicion (II)

- Cálculo del Coste Temporal:
  - Ecuaciones de Recurrencia:
    - $T_{mismoValorPos}(talla = 0) = k'$
    - $T_{mismoValorPos}(talla > 0) = 1 * T_{mismoValorPos}(talla / 2) + k$
  - Complejidad Temporal: Aplicando el Teorema 3 con a = 1, c = 2
    - $T^{P}_{mismoValorPos}(talla = v.length) \in \mathbf{\Theta(log_2\ v.length)}$

Es Capicúa

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

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


Sumar Vector 2 Versiones (I)

1. Talla del problema

   $talla_{sumarV1} = v.length – inicio$

   $talla_{sumarV2} = fin – inicio + 1$

2. Instancias Significativas

   No las hay en ninguno de los dos algoritmos. Se trata de un problema de recorrido en el que hay que procesar todas las componentes del array:

Sumar Vector 2 Versiones (II)

3. Ecuaciones de Recurrencia y Cotas

- Algoritmo sumarV1:
  - Ecuaciones de Recurrencia:
    - $T_{sumarV1}(talla = 0) = k'$
    - $T_{sumarV1}(talla > 0) = 1 * T_{sumarV1}(talla - 1) + k$
  - Complejidad Temporal (Teorema 1 con a = 1, c = 1)
    - $T_{sumarV1}(talla) \in \Theta(talla)$
- Algoritmo sumarV2:
    - $T_{sumarV2}(talla = 1) = k'$
    - $T_{sumarV2}(talla > 1) = 2 * T_{sumarV2}(talla / 2) + k$
  - Complejidad Temporal (Teorema 3 con a = 2, c = 2)
    - $T_{sumarV2}(talla) \in \Theta(talla^{\log_2(2)}) \rightarrow \Theta(talla)$

Inserción Directa Recursivo

```
private static <T extends Comparable<T>> void
  insercionDirectaR(T a[], int inicio, int fin) {
  if ( inicio <= fin ) {
     T elemAInsertar = a[inicio];
     int   posIns = inicio ;
     for(; posIns>0 && elemAInsertar.compareTo(a[posIns-
 1])<0; posIns--)
          a[posIns] = a[posIns - 1];
     a[posIns] = elemAInsertar;
     insercionDirectaR(a, inicio+1, fin);
  }
}
```


Inserción Directa Recursivo (II)

1. Talla del Problema.
   talla = fin – inicio + 1 (nº de elementos del vector)
2. Instancias Significativas.
   - Caso Mejor: Vector ya ordenado ascendentemente:
     $T^{M}_{insercionDirectaR}(talla) = talla*k$
   - Caso Peor: Vector ordenado descendentemente:

- $T^{P}_{insercionDirectaR}(talla = 0) = k$
- $T^{P}_{insercionDirectaR}(talla > 0) = 1 * T^{P}_{insercionDirectaR}(talla - 1) + talla * k'$

1. Complejidad Asintótica para cada Instancia Significativa (Teorema 2):
   - $T^{P}_{insercionDirectaR}(talla) \in \Theta(talla^2)$
   - $T^{M}_{insercionDirectaR}(talla) \in \Theta(talla)$

   $T_{insercionDirectaR}(talla = v.length) \in O(v.length^2)$

   $T_{insercionDirectaR}(talla = v.length) \in \Omega(v.length)$

buscaPar (I)

1. Describir problema resuelto por buscaPar
   - 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 el valor 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.

buscaPar (II)

1. Describir la estrategia de diseño seguida en buscaPar:
   - Estrategia de búsqueda binaria: Se divide la entrada original v en dos partes disjuntas y de igual talla, la mitad de la original.
   - Una vez dividido el problema, el par de Integer x e y puede aparecer:
     1. bien en el centro, si x es igual a v[mitad] e y es igual a v[mitad+1];
     2. bien en la segunda mitad de v, si v[mitad] es menor que x;
     3. bien en la primera mitad de v, si v[mitad] es mayor que x.
   - El caso base se define para el intervalo de búsqueda vacío o con un solo elemento, por lo que la respuesta del método es entonces false.

buscaPar (III)

- Complejidad temporal del método buscaPar:
  - Talla del problema:
    - Número de elementos en el espacio de búsqueda (talla=der-izq+1)
- Instancias significativas:
  - Caso Mejor: x se encuentra en la mitad del primer intervalo de búsqueda:
    - $T^{M}_{buscaPar}(talla) = k1$ $T^{M}_{buscaPar}(talla) \in \Theta(1)$
  - Caso Peor: x no se encuentra en el vector:
    - $T^{P}_{buscaPar}(talla <= 1) = k$
    - $T^{P}_{buscaPar}(talla > 1) = 1*T^{P}_{buscaPar}(talla / 2) + k2$
    - $T^{P}_{buscaPar}(talla > 1) \in \Theta(\log_2(talla))$
  - $T_{buscaPar}(talla) \in \Omega(1)$ $T_{buscaPar}(talla) \in O(\log_2(talla))$

Coste del Método Comparar (I)

1. Tipo de recursión:
   - Recursión Múltiple, ya que se realiza más de una llamada recursiva en el caso general.
2. Talla, en función de los parámetros:
   1. Talla = fin – inicio + 1
3. Instancias significativas para una talla dada:
   1. Sí que hay.
      1. Caso Mejor: Las componentes centrales de los vectores a y b no son iguales.
      2. Peor de los casos: Los arrays a y b son iguales.

Coste del Método Comparar (II)

4. Ecuaciones de Recurrencia
   - Para el caso mejor son triviales: $T^{M}_{comparar}(talla) = k_1$
   - Para el caso peor:
     - $T^{P}_{comparar}(talla = 0) = k_2$
     - $T^{P}_{comparar}(talla > 0) = 2*T^{P}_{comparar}(talla/2) + k_3$

5. Complejidad temporal del método:
   1. Utilizando el Teorema 3 con a = c = 2
      - $T^{P}_{comparar}(talla) \in \Theta(talla)$
      - $T^{M}_{comparar}(talla) \in \Theta(1)$
      - $T_{comparar}(talla) \in \Omega(1)$, $T_{comparar}(talla) \in O(talla)$

Complejidad Temporal de Selección Rápida (I)

- Talla del Problema
  - Talla = der – izq + 1
- Instancias significativas:
  - Sí que las hay
    - Mejor Caso: No se produce llamada recursiva en el caso general, pues tras efectuar la primera partición del subarray v[izq .. der] ya se cumple que k-1 == indiceP
    - Peor Caso: siempre se produce una llamada recursiva en el caso general pues la ***particion*** que se efectúa del sub*array v[izq … der]* es, siempre, lo más desequilibrada posible:
      - Siempre, bien *indiceP == der* o bien *indiceP == izq* y, siempre además, *k-1* es distinto de *indiceP* (bien *indiceP > k-1* si *indiceP == der* o bien *indiceP < k-1* si *indiceP == izq*). En este caso entonces la talla del sub*array* de *v* solo decrece en una unidad en cada llamada recursiva

Complejidad Temporal de Selección Rápida (II)

- Ecuaciones de Recurrencia
  - (Obviadas las ecuaciones para el caso base talla <= 1)
  - $T^{M}_{seleccionRapida}(\text{talla} > 1) = T_{particion}(\text{talla}) + k = k' * \text{talla} + k$
  - $T^{P}_{seleccionRapida}(\text{talla} > 1) = 1 * T^{P}_{seleccionRapida}(\text{talla}-1) + T_{particion}(\text{talla}) + k$
  - Reformulamos la ecuación para el caso peor:
    - $T^{P}_{seleccionRapida}(\text{talla}>1) = 1 * T^{P}_{seleccionRapida}(\text{talla}-1) + k' * \text{talla}$

- Cotas de Complejidad Asintótica
  - $T_{seleccionRapida}(\text{talla}) \in \Omega(\text{talla})$
  - $T_{seleccionRapida}(\text{talla}) \in O(\text{talla}^2)$