Programación II

Tema 6. Divide y Vencerás

6.1 Características Generales

Técnica "Divide y Vencerás" (*Divide and Conquer*):

- Se divide el problema en subproblemas
  - y, recursivamente, cada subproblema se divide de nuevo
- Cuando el caso es lo suficientemente sencillo se resuelve utilizando un algoritmo directo (no recursivo)
  - el algoritmo directo debe ser eficiente para problemas sencillos
  - no importa que no lo sea para problemas grandes
- Cada subproblema se resuelve de forma independiente
- Finalmente se combinan las soluciones de todos los subproblemas para formar la solución del problema original

Pseudocódigo genérico de un algoritmo DyV

```
método divideYVencerás(x) retorna y
  si x es suficientemente sencillo entonces
    // caso directo
    retorna algoritmoDirecto(x)
  fsi
  // caso recursivo
  descompone x en subproblemas x_1, x_2, .., x_s
  desde i := 1 hasta s hacer
    // llamadas recursivas
    y_i := divideYVencerás(x_i)
  fhacer
  // combina las soluciones
  y := combinación de las soluciones parciales (y_i)
  retorna y
fmétodo
```


Cuando utilizar algoritmos DyV

Para que resulte interesante aplicar DyV debe verificarse que:

- **la formulación recursiva nunca resuelva el mismo subproblema más de una vez**
- **la descomposición en subproblemas y la combinación de las soluciones sean operaciones eficientes**
- **los subproblemas sean aproximadamente del mismo tamaño**

6.2 Eficiencia de los algoritmos DyV

- **Se obtiene aplicando el "*Master Theorem*" que permite resolver recurrencias del tipo:**

  $t(n) = s \cdot t(n/b) + g(n)$ (donde $g(n)$ es $O(n^k)$)

- **Donde se supone que:**
  - **el algoritmo divide el problema en s subproblemas**
    - **cada uno de un tamaño aproximado n/b**
  - **`g(n)` es el tiempo necesario para realizar la descomposición y combinación de resultados**

- **Cuando `g(n)` es `O(n^k)` puede demostrarse que `t(n)` es:**
  - $\Theta(n^k)$ si $s<b^k$
  - $\Theta(n^k \log n)$ si $s=b^k$
  - $\Theta(n^{\log_b s})$ si $s>b^k$

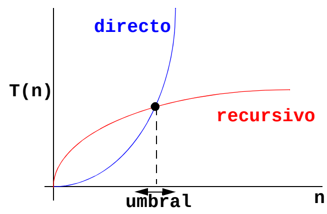
Selección del umbral de utilización del algoritmo directo

Utilizaremos el algoritmo directo cuando el tamaño del problema sea menor que el umbral elegido

Tiene gran influencia en el tiempo de ejecución del algoritmo

- **aunque no en su ritmo de crecimiento**

El valor apropiado estará cerca del tamaño para el que el tiempo empleado por el algoritmo recursivo se iguala con el utilizado por el algoritmo directo



Puede obtenerse teóricamente o realizando medidas de tiempos

6.3 Búsqueda binaria

Búsqueda de un elemento en una tabla ordenada

Solución secuencial:

```
método búsquedaSecuencial(entero[1..n] t, entero x)
                                       retorna entero
  desde i:=1 hasta n hacer
    si t[i] = x entonces
      retorna i  // encontrado
    fsi
  fhacer
  retorna -1  // no encontrado
fmétodo
```

Eficiencia: O(n)

Solución DyV: algoritmo "búsqueda binaria"

Se trata de una de las aplicaciones más sencillas de DyV

- **realmente no se va dividiendo el problema sino que se va reduciendo su tamaño a cada paso**
  - **algoritmos DyV denominados de reducción o simplificación**

Ejemplo: búsqueda de x=9

	1	2	3	4	5	6	7	8	9	10	11	x = t[k]	x > t[k]
	-5	-2	0	3	8	8	9	12	15	26	31		
paso 1	i					k					j	no	sí
paso 2							i		k		j	no	no
paso 3							i,k	j				sí	-

- **En cada paso (hasta que x=t[k] o i>j)**
  - si `x > t[k]` ⇒ `i = k+1`
  - si `x < t[k]` ⇒ `j = k-1`
  - `k = (i + j) / 2`

```
método búsquedaBinaria(entero[1..n] t, entero i,
                       entero j, entero x) retorna entero
  // calcula el centro
  k := (i + j)/2
  // caso directo
  si i > j entonces
    retorna -1  // no encontrado
  fsi
  si t[k] = x entonces
    retorna k  // encontrado
  fsi
  // caso recursivo
  si x > t[k] entonces
      retorna búsquedaBinaria(t, k+1, j, x)
  sino
      retorna búsquedaBinaria(t, i, k-1, x)
  fsi
fmétodo
```


Eficiencia de "búsqueda binaria"

Como vimos, el tiempo requerido por un algoritmo DyV es de la forma:

`t(n) = s·t(n/b) + g(n)`

Para este algoritmo:

- cada llamada genera una llamada recursiva (s=1)
- el tamaño del subproblema es la mitad del problema original (b=2)
- sin considerar la recurrencia el resto de operaciones son O(1) luego $g(n)$ es $O(1)=O(n^0)$ (k=0)

Estamos en el caso:

- $s=b^k$ ($1=2^0$)
- luego $t(n)$ es $\Theta(n^k \log n) = \Theta(n^0 \log n) = \Theta(\log n)$

6.4 Problema de la selección

Búsqueda del k-ésimo menor elemento de una tabla

- es decir: si la tabla estuviera ordenada crecientemente, el elemento devuelto sería el que ocuparía el k-ésimo lugar

Solución obvia: ordenar la tabla y acceder al k-ésimo elemento

- coste $O(n \log n)$ (coste de la ordenación)

Es posible encontrar un algoritmo más eficiente utilizando DyV:

- se elige un valor como "pivote"
- se reorganiza la tabla en dos partes, una con los elementos mayores que el pivote y otra con los elementos menores
  - la clave estará en la selección correcta del pivote
- se realiza de nuevo el proceso sobre la parte de la tabla que contiene el elemento buscado

Implementación del algoritmo de selección

Método público `select`

- llama a `selectRec` con los parámetros iniciales

```
/**
 * Retorna el elemento del array t que ocuparía la
 * posición k-ésima en el caso de que el array
 * estuviera ordenado
 * @param t array
 * @param k posición del elemento buscado (la primera
 * posición es la 1, no la 0)
 * @return valor del elemento buscado
 */
public static int select(int[] t, int k) {
  return selectRec(t,0,t.length-1,k);
}
```


Método privado `selectRec`

- es el realmente implementa el algoritmo recursivo DyV

```
/**
 * Retorna el elemento de la parte del array t
 * comprendida entre los índices ini y fin que ocuparía
 * la posición k-ésima (en esa parte) en el caso de que
 * esa parte estuviera ordenada
 * @param t array
 * @param ini índice inicial de la parte de t utilizada
 * @param fin índice final de la parte de t utilizada
 * @param k posición del elemento buscado (la primera
 * posición es la 1, no la 0)
 * @return valor del elemento buscado
 */
private static int selectRec(int[] t,
      int ini, int fin, int k) {
  ... código en la transparencia siguiente ...
}
```



```
private static int selectRec(int[] t,
      int ini, int fin, int k) {

  // caso directo
  if (ini == fin) {
      return t[ini];  // elemento en la pos. k-ésima
  }

  // reorganiza los elementos y retorna la posición
  // del último elemento menor que el pivote
  int p = reorganiza(t, ini, fin);
  int k1 = p - ini + 1; // offset del primer elemento
                        // mayor que el pivote

  // divide
  if (k <= k1) {
    return selectRec(t,ini,p,k);
  } else {
    return selectRec(t,p+1,fin,k-k1);
  }
}
```

```java
/**
 * Reorganiza la parte de t comprendida entre los
 * índices ini y fin en dos partes, una con los
 * elementos mayores que el pivote y otra con los
 * menores. Se toma como pivote t[ini]
 * @param t array
 * @param ini índice inicial de la parte de t utilizada
 * @param fin índice final de la parte de t utilizada
 * @return índice del último ele. menor que el pivote
 */
private static int reorganiza(int[] t,
      int ini, int fin){
  ... código en la transparencia siguiente ...
}
```



```java
private static int reorganiza(int[] t,
      int ini, int fin) {
    int x=t[ini]; // usa el primer ele. como pivote
    int i=ini-1; int j=fin+1;
    while (true) {
      do {  // busca ele. menor o igual que el pivote
        j--;
      }while (t[j]>x);
      do { // busca ele. mayor o igual que el pivote
        i++;
      } while (t[i]<x);
      if (i < j) {
        int z=t[i]; t[i]=t[j]; t[j]=z; // intercambio
      } else {
        return(j);
      }
    }
  }
```

Es más eficiente usar la "pseudo-mediana" (ver pg. 21)

Ejemplo de ejecución

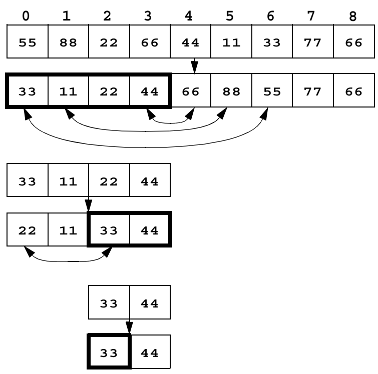
	0	1	2	3	4	5	6	7	8
t	55	88	22	66	44	11	33	77	66

pos 3

```
select(t,0,8,3)
reorganiza(t,0,8)→3
k1←4, ini←0, fin←3

select(t,0,3,3)
reorganiza(t,0,3)→1
k1←2, ini←2, fin←3

select(t,2,3,1)
reorganiza(t,2,3)→2
k1←1, ini←2, fin←2
```



select(t,2,2,1)→ $\boxed{33}$

Eficiencia

Caso promedio: `O(n)`

Peor caso: array ordenado de menor a mayor y buscamos el elemento que ocupa la última posición

	0	1	2	3	4	5	6	7
t	10	33	35	35	44	70	80	85

En este caso se realizan n llamadas recursivas, en cada una:

- sólo se descarta un elemento
- `reorganiza` recorre todos los elementos no descartados (n en la primera iteración, `n-1` en la segunda y así sucesivamente)

$$\sum_{i=0}^{n-1} (n-i) = n \cdot \frac{n+1}{2} = \frac{n^2}{2} - \frac{n}{2}$$

La eficiencia del algoritmo es $O(n^2)$ → (¡¡mayor que `O(n log n)`!!)

Selección de un buen pivote

La eficiencia mejora si somos capaces de modificar `reorganiza` para que elija un buen pivote

- que divida aproximadamente a la mitad el vector
- y cuya búsqueda se realice en un tiempo aceptable (p.e. `O(n)`)
- el tiempo de ejecución de `reorganiza` seguirá siendo `O(n)` (`O(n)` para encontrar el pivote, más `O(n)` para reordenar)

En ese caso el tiempo de ejecución del algoritmo de selección será:

`t(n) = t(n/2) + O(n)   →   (s=1, b=2 y k=1)`

Estamos en el caso:

- $s<b^k$ $(1<2^1)$
- luego $t(n)$ es $\Theta(n^k) = \Theta(n)$

Cálculo de la pseudo-mediana

El pivote ideal sería la *mediana* de los elementos de la tabla:

- elemento que utilizado pivote divide a la tabla en dos mitades
- su cálculo es demasiado complejo

En su lugar se utiliza la *pseudo-mediana*:

- valor cercano a la mediana
- que puede calcularse en $\Theta(n)$ (no lo vamos a demostrar)

Cálculo de la pseudo-mediana:

- se divide la tabla en grupos de `r` elementos (puede demostrarse que un valor apropiado de `r` es 5)
- para cada grupo se calcula su mediana exacta
- se obtiene la mediana de las medianas utilizando el algoritmo de selección recursivamente

Algoritmo de búsqueda de la pseudo-mediana

```
método pseudomediana(entero[0..n-1] t, entero ini,
                     entero fin) retorna entero
  numGrupos := (fin-ini+1)/r
        // un valor apropiado de r es 5
        // "/" es la división entera. Los restantes
        // "(fin-ini+1)%r" elemen. no se consideran
  // calcula medianas
  desde i:=0 hasta numGrupos-1 hacer
    medianaGrupo[i]:=
      calculaMediana(t[r*i+ini] .. t[r*(i+1)+ini-1])
  fhacer
  // calcula la mediana de las medianas
  retorna select(medianaGrupo, 0, numGrupos-1,
                 numGrupos/2)
fmétodo
```

- hay que modificar `reorganiza` para que utilice este método

Bondad de la pseudo-mediana

La mediana de cada grupo es mayor o igual que 3 de los 5 elementos de su grupo

La pseudo-mediana es mayor o igual que las medianas de la mitad de los grupos

- luego es mayor o igual que `3*numGrupos/2` elementos

Si suponemos que n es múltiplo de 5 (`numGrupos=n/5`)

- la pseudo-mediana es mayor o igual que `3n/10` elementos
- y menor que `7n/10` elementos

grupos	7 -2 9 -5 4	4 -1 -4 5 3	-7 1 0 -3 -8	6 -5 7 5 4	-5 1 9 -9 0	-3 -1 -4 -1 7
medianas	4	3	-3	5	1	-1

Esta proximidad a la mediana exacta, junto con el hecho de que `pseudomediana` sea $\Theta(n)$, permite asegurar que el algoritmo `select` sea $\Theta(n)$ (no lo vamos a demostrar)

6.5 Subsecuencia de suma máxima

El problema consiste en buscar la subsecuencia de suma máxima dentro de un vector

Por ejemplo:

1	-2	11	-4	13	-5	2	2

subsecuencia de suma máxima
(suman 20)

Buen ejemplo de algoritmo DyV ya que es relativamente sencillo y contiene todos los elementos de este tipo de algoritmos:

- caso directo
- descomposición en subproblemas y caso recursivo
- recombinación de las soluciones parciales

Pero existe una solución más eficiente que la DyV

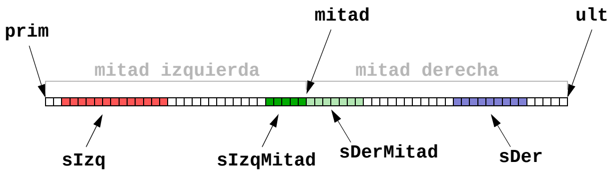
- con ritmo de crecimiento lineal (ver pág. 35)

Algoritmo DyV

Se divide el vector en dos mitades

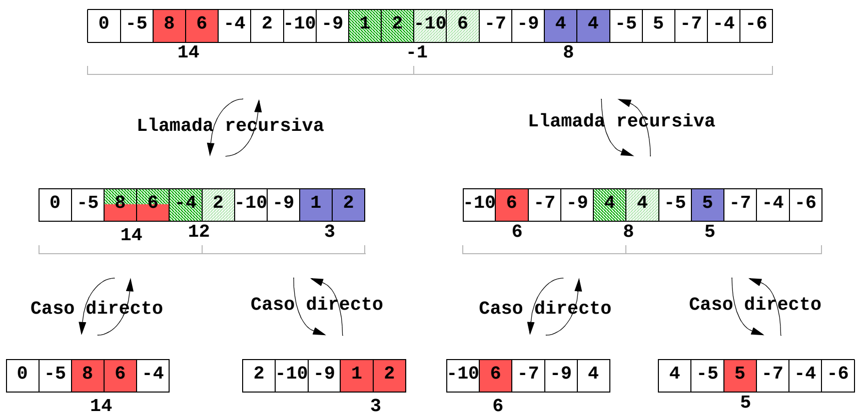
La subsecuencia de suma máxima será:

- **la encontrada en la mitad izquierda (`sIzq`)**
- **o la encontrada en la mitad derecha (`sDer`)**
- **o la que incluye el elemento en la mitad del vector (`sIzqMitad+sDerMitad`)**



`subsecuencia de suma máxima = MAX(sIzq, sDer, sIzqMitad+sDerMitad)`

`subsecuencia de suma máxima = MAX(14, 8, -1) = 14`



Implementación del algoritmo DyV

```java
/**
 * Clase auxiliar: subsecuencia del array total
 * representada mediante sus índices izquierdo y
 * derecho y la suma de todos sus elementos
 */
public static class SubSecuencia {
  int suma;
  int izq;
  int der;

  /**
   * Constructor. Crea una subsecuencia vacía
   */
  public SubSecuencia() {
    izq=0;
    der=-1;
    suma=Integer.MIN_VALUE;
  }
```

```
   /**
    * Retorna la subsecuencia de mayor suma
    * @param s1 primera subsecuencia a comparar
    * @param s2 segunda subsecuencia a comparar
    * @return la subsecuencia de mayor suma
    */
   public static SubSecuencia max(SubSecuencia s1,
         SubSecuencia s2){

     if (s1.suma>=s2.suma)
       return s1;
     else
       return s2;
   }

}
```


El algoritmo DyV se estructura en dos métodos estáticos:

- busquedaDyV: método público que llama al método recursivo busquedaDyVRec con los valores iniciales apropiados:
  - rango de búsqueda igual a toda la longitud del array (`prim=0`, `ult=a.length-1`)
- busquedaDyVRec: método privado que es el que realmente implementa el algoritmo mediante llamadas recursivas

```
/**
 * Busca la subsecuencia de suma máxima en el array a
 * utilizando un algoritmo DyV
 * @param a array en el que buscar la subsecuencia
 * @return subsecuencia de suma máxima
 */
public static SubSecuencia busquedaDyV(int[] a){
  return busquedaDyVRec(a, 0, a.length-1);
}
```



```
/**
 * Busca la subsecuencia de suma máxima en el trozo
 * del array a entre los índices prim y ult
 * @param a array en el que buscar la subsecuencia
 * @param prim primer índice del trozo de búsqueda
 * @param ult último índice del trozo de búsqueda
 * @return subsecuencia de suma máxima en el trozo
 * del array indicado
 */
private static SubSecuencia busquedaDyVRec(int[] a,
    int prim, int ult) {

  // aplicamos el algoritmo directo?
  if (ult-prim+1 <= umbralCasoDirecto)
    return busquedaCuadrática(a,prim,ult);

  // caso recursivo
  int mitad=(ult+prim) / 2;
  SubSecuencia sIzq = busquedaDyVRec(a, prim, mitad);
  SubSecuencia sDer = busquedaDyVRec(a, mitad+1, ult);
  SubSecuencia mejorSol = SubSecuencia.max(sIzq,sDer);
```

Implementación del algoritmo DyV (cont.)

```
    // recombinación de las soluciones
    // busca la mejor secuencia en la parte izquierda
    int sumaIzqMitad=Integer.MIN_VALUE;
    int izq=-1;
    int suma = 0;
    for(int i=mitad; i>=prim; i--) {
      suma += a[i];
      if (suma > sumaIzqMitad) { // nueva mejor suma
        izq=i;  sumaIzqMitad=suma;
      }
    }
    // busca la mejor secuencia en la parte derecha
    int sumaDerMitad=Integer.MIN_VALUE;
    int der=-1;
    suma = 0;
    for(int j=mitad+1; j<=ult; j++) {
      suma += a[j];
      if (suma > sumaDerMitad) { // nueva mejor suma
        der=j;  sumaDerMitad=suma;
      }
    }
```


Implementación del algoritmo DyV (cont.)

```
    // la secuencia central es la mejor?
    if (sumaDerMitad+sumaIzqMitad > mejorSol.suma) {
      mejorSol.suma = sumaDerMitad + sumaIzqMitad;
      mejorSol.izq = izq;
      mejorSol.der = der;
    }

    // retorna la mejor secuencia encontrada
    return mejorSol;
  }
```


Solución cuadrática para el caso directo

```
  private static SubSecuencia busquedaCuadrática(
        int[] a, int prim, int ult){
    // mejor solución (empieza con una secuencia vacía)
    SubSecuencia sol = new SubSecuencia();
    int suma=0;
    // calcula todas las sumas para cada elemento
    for(int izq=prim; izq<=ult; izq++) {
      // calcula las sumas de a[izq] con los siguientes
      suma=0;
      for(int der=izq; der<=ult; der++) {
        suma += a[der]; // suma de a[izq]+...+a[der]
        if (suma>sol.suma) {
          // mejor suma hasta el momento
          sol.izq=izq; sol.der=der; sol.suma=suma;
        }
      }
    }
    return sol; // retorna la subsecuencia de mayor suma
  }
```

Eficiencia del algoritmo y selección del umbral

Para calcular la *eficiencia* aplicamos el "*Master Theorem*". Para este algoritmo la recurrencia es:

$$t(n) = 2\cdot t(n/2) + O(n) \quad \rightarrow \quad (s=2,\ b=2 \text{ y } k=1)$$

- **Estamos en el caso:**
  - **$s=b^k$ $(2=2^1)$**
  - **luego $t(n)$ es $\Theta(n^k \log n) = \Theta(n \log n)$**

Para elegir el *umbral de utilización del algoritmo directo* realizamos medidas para un problema de tamaño n=10000000

umbral	1	10	14	18	20	30	40	60	100	500
t (ms.)	962	757	756	759	774	775	826	826	993	2131

- **en vista de los resultados elegimos 14 como valor de umbral**

Solución lineal (no DyV)

La solución de problema de la búsqueda de las subsecuencia de suma máxima por DyV

- **constituye un buen ejemplo de este tipo de algoritmos**

Pero NO deberemos utilizar dicha solución basada en DyV ya que

- ***existe otra solución no basada en DyV más eficiente***
  - **con ritmo de crecimiento lineal ($O(n)$)**


```java
public static Solución busquedaLineal(int[] a){
  Solución mejorSol = new Solución();
  int suma=0; int inicioSección=0;
  // calcula todas las sumas para cada elemento
  for(int i=0; i<a.length; i++) {
    suma += a[i]; // suma = a[inicioSección]+...+a[i]
    if (suma>mejorSol.suma) {
      mejorSol.izq=inicioSección; mejorSol.der=i;
      mejorSol.suma=suma;
    }
    if (suma<=0) {
      // si la suma hasta i no es positiva, esa parte
      // del vector no entra en la suma máxima
      inicioSección=i+1;
      suma=0;
    }
  }
  return mejorSol; // retorna la mejor solución
}
```

6.6 Otros algoritmos DyV

Veremos otros algoritmos que utilizan la técnica DyV en el siguiente tema dedicado a los algoritmos ordenación:

- **Ordenación por fusión (*mergesort*)**
- **Ordenación rápida (*quicksort*)**

6.7 Bibliografía

[1] **Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.**

[2] **Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.**

[3] **Sartaj Sahni, *Data Structures, Algoriths, and Applications in Java*. Mc Graw Hill, 2000**