Diseño de algoritmos

Divide y Vencerás

Jesús Bermúdez de Andrés

Universidad del País Vasco/Euskal Herriko Unibertsitatea **(UPV/EHU)**

Curso 2008-09

1. Divide y Vencerás
   - Búsqueda dicotómica
   - Ordenación por Mezcla
   - Ordenación rápida
   - Selección del k-ésimo
   - Segmento de suma máxima

Técnica Divide y Vencerás

Consiste en:

- descomponer, el ejemplar de problema a resolver, en un cierto número de (sub)ejemplares más pequeños del mismo problema,
- resolver independientemente cada subejemplar — mediante el mismo procedimiento — y
- combinar los resultados obtenidos para construir una solución del ejemplar original.

func DIVIDE_Y_VENCERÁS (P) **return** Solución
  if P es pequeño o simple **then**
    return SOLUCIÓN_SIMPLE (P)
  else
    DESCOMPONER P en k subproblemas $P_1, \ldots, P_k$
    for $i \leftarrow 1$ **to** k **loop**
      $S[i] \leftarrow$ DIVIDE_Y_VENCERÁS (P_i)
    end loop
  return COMBINAR ($S[1], \ldots, S[k]$)

Análisis típico

Para que esta técnica resulte rentable, la descomposición en subejemplares y la combinación de soluciones de subejemplares debe ser eficiente.
El análisis típico de un algoritmo divide y vencerás requiere la resolución de una recurrencia del estilo siguiente:

$$T(n) = \begin{cases} O(1) & \text{si } n \leq n_0 \\ k \cdot T(n/c) + DC(n) & \text{si } n > n_0 \end{cases}$$

donde:

- n_0 es una constante que determina el tamaño pequeño de un problema
- k es el número de subproblemas y n/c es el tamaño de cada subproblema. Si hay subproblemas con diferentes tamaños, esta parte será la suma de las expresiones correspondientes a cada uno de los subproblemas
- $DC(n)$ es la función de coste de la descomposición más la composición

Búsqueda dicotómica

El algoritmo de **búsqueda dicotómica** decide si un determinado valor X se encuentra entre los elementos de una tabla ordenada $T(I..F)$

```
func Búsqueda_Dicotómica (T: Tabla; I, F: Integer; X: Clave)
return Boolean
   if I > F then return False
   else
      Mitad ← (I + F)/2
      if T(Mitad) = X then
         return True
      elsif T(Mitad) < X then
         return Búsqueda_Dicotómica (T, Mitad+1, F, X)
      else     {T(Mitad) > X}
         return Búsqueda_Dicotómica (T, I, Mitad-1, X)
```

Análisis de Búsqueda_Dicotómica(T, 1, n, X)

Número de operaciones elementales:

$$f(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ f(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

que resolviendo, resulta ser $f(n) = O(\log n)$.

Espacio extra de $O(\log n)$, que es lo máximo que puede crecer la pila de recursión.

Ordenación por Mezcla (1/2)

El algoritmo de **Ordenación por Mezcla** descompone la tabla de entrada en dos mitades, ordena cada mitad recursivamente, y mezcla ordenadamente cada mitad previamente ordenada.

proc ORDEN_POR_MEZCLA (T: Tabla; *Inicio*, *Fin*: Natural)
  if *Inicio* $<$ *Fin* **then**
    ORDEN_POR_MEZCLA(T, *Inicio*, $(Inicio + Fin)/2$)
    ORDEN_POR_MEZCLA(T, $((Inicio + Fin)/2) + 1$, *Fin*)
    MEZCLA (T, *Inicio*, $(Inicio + Fin)/2$, *Fin*)

Ordenación por Mezcla (2/2)

pre: $T(I..Centro)$ y $T(Centro+1..F)$ están ordenados y $I \leq Centro \leq F$
post: $T(I..F)$ es la mezcla ordenada de $T(I..Centro)$ y $T(Centro+1..F)$

```
proc MEZCLA (T: Tabla; I, Centro, F: Natural)
   Izq ← I
   Der ← Centro + 1
   A ← I      {primer subíndice de La_Mezcla}
   while Izq ≤ Centro ∧ Der ≤ F loop
      if T(Izq) ≤ T(Der) then
         La_Mezcla(A) ← T(Izq)
         Izq ← Izq+1
      else
         La_Mezcla(A) ← T(Der)
         Der ← Der+1
      end if
      A ← A+1
   end loop
   if Izq > Centro then
      La_Mezcla(A..F) ← T(Der..F)
   else
      La_Mezcla(A..F) ← T(Izq..Centro)
   end if
   T(I..F) ← La_Mezcla(I..F)
```

Análisis de Orden_Por_Mezcla $(T, 1, n)$

Número de operaciones elementales:

$$f(n) = 2f(\frac{n}{2}) + \Theta(n)$$

que resolviendo, resulta $f(n) = \Theta(n \log n)$.

NO ordena "in situ", necesita espacio $\Theta(n)$ para la mezcla.

Ordenación rápida

El algoritmo de Ordenación_Rápida descompone el problema original de una manera tan interesante que hace que la fase de composición de soluciones sea trivial.
El algoritmo Partición(T, i, j) permuta los elementos de la tabla $T(i..j)$ de manera que, con el índice p que devuelve, se satisface lo siguiente: $\forall x.(i \leq x \leq p \rightarrow T(x) \leq T(p)) \wedge (p < x \leq j \rightarrow T(x) > T(p))$.

```
proc Ordenación_Rápida(T: Tabla, i, j: Natural)
    if i < j then
        p ← Partición(T, i, j)
        Ordenación_Rápida(T, i, p − 1)
        Ordenación_Rápida(T, p + 1, j)
```

Ordenación rápida

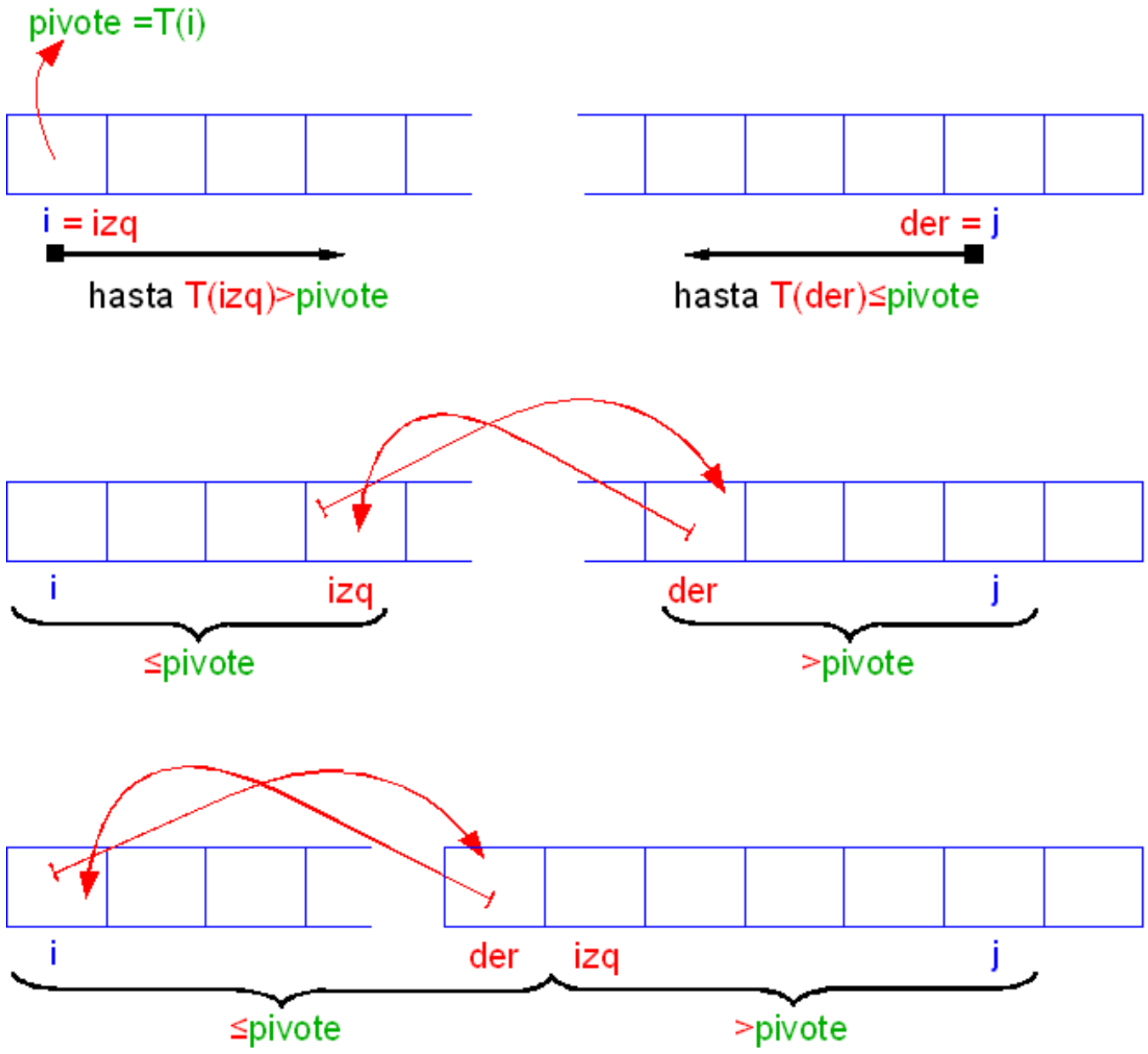
Para una codificación simétrica de Partición, que permita desentendernos del tratamiento de índices de la tabla fuera del rango, decidimos instalar un valor centinela al final de la tabla inicial.

(Hay otras versiones para Partición que pueden consultarse en las lecturas recomendadas).

Entonces, la llamada inicial para ordenar una tabla $T(1..n)$ sería del siguiente modo:

$T(n+1) \leftarrow T(1) + 1$ {valor centinela}
Ordenación_Rápida(T, 1, n)

Partición para Ordenación rápida (1/2)



Partición para Ordenación rápida (2/2)

Permuta los elementos de la tabla $T(i..j)$ de manera que, con el índice p que devuelve, se satisface lo siguiente:
$\forall x.(i \leq x \leq p \rightarrow T(x) \leq T(p)) \wedge (p < x \leq j \rightarrow T(x) > T(p))$.

```
func PARTICIÓN(T:Tabla, i, j: Natural) return Natural
    pivote ← T(i)
    izq ← i
    der ← j
    while izq < der loop
        while T(izq) ≤ pivote loop izq ← izq +1 end loop
        while T(der) > pivote loop der ← der −1 end loop
        if izq < der then
            INTERCAMBIAR(T(izq), T(der))
            izq ← izq +1
            der ← der −1
    end loop
    INTERCAMBIAR(T(i), T(der))
    return der
```

Análisis de Ordenación_Rápida, en el caso medio (1/4)

- Sea k el número de elementos a ordenar.
- Denominaremos $M(k)$ al número promedio de comparaciones entre componentes del vector que realiza Ordenación_Rápida sobre un vector de k elementos.
- La probabilidad de que el pivote p quede en la posición número i es $\frac{1}{n}$ en un vector de tamaño n.
- Entonces el valor de $M(n)$ será $n+2$ comparaciones, que realiza Partición$(T, 1, n)$, más la suma del número de comparaciones que se producen en promedio para cada caso en que la posición del pivote sea $i = 1, \ldots n$.

Expresado en forma de recurrencia obtenemos lo siguiente:

$$\begin{aligned} M(n) &= n+2+\sum_{i=1}^{n}\frac{1}{n}(M(i-1)+M(n-i)) \quad si \quad n \geq 2 \\ M(1) &= M(0)=0 \end{aligned}$$

Análisis de Ordenación_Rápida, en el caso medio (2/4)

y observando los sumandos del caso general es fácil ver que

$$M(n) \quad = \quad n+2+\frac{1}{n}\sum_{i=2}^{n-1} 2M(i) \quad si \quad n \geq 2$$

En vez de intentar resolver de manera directa esta recurrencia, hacemos la siguiente conjetura:

- $M(n) \leq cn \ln n$ para alguna constante $c > 0$, que demostraremos por inducción.

Hipótesis de inducción: Suponemos que $M(i) \leq ci \ln i$ (para todo i: $1 \leq i \leq n-1$).
Demostraremos que $M(n) \leq cn \ln n$, ajustando la constante c.
Aplicando la hipótesis de inducción a la última ecuación, tenemos que:

$$M(n) \quad \leq \quad n+2+\frac{2}{n}\sum_{i=2}^{n-1} ci \ln i$$

Análisis de Ordenación_Rápida, en el caso medio (3/4)

En vez de buscar el valor exacto del sumatorio, lo acotamos superiormente por el valor de una integral definida

$$\sum_{i=2}^{n-1} ci \ln i \quad \leq \quad c \int_2^n x \ln x \; dx = \frac{n^2 \ln n}{2} - \frac{n^2}{4} - 2 \ln 2 + 1$$

Por lo tanto:

$$\begin{aligned} M(n) &\leq n + 2 + \frac{2c}{n}\left(\frac{n^2 \ln n}{2} - \frac{n^2}{4} - 2 \ln 2 + 1\right) \\ &= cn \ln n + 2 + n\left(1 - \frac{c}{2}\right) + \frac{2c}{n}(1 - 2 \ln 2) \end{aligned}$$

Obsérvese que $(1 - 2 \ln 2) < 0$ porque $\ln 2 \simeq 0{,}7$; y si elegimos el valor de la constante $c \geq 6$ entonces $2 + n(1 - \frac{c}{2}) \leq 0$ para todo $n \geq 1$.

Análisis de Ordenación_Rápida, en el caso medio (4/4)

Por lo que podemos concluir que

$$M(n) \quad \leq \quad cn \ln n \quad \textit{para} \quad c \geq 6 \quad \textit{y} \quad n \geq 1$$

Además, se satisface la conjetura para el **caso básico** $n = 1$ ya que $M(1) = 0 = c1 \ln 1$.

Por lo tanto, Ordenación_Rápida$(T, 1, n)$ es $O(n \lg n)$ en el caso medio.

Selección del k-ésimo (1/3)

Para calcular el primero de n elementos, según una relación de orden ($\leq$), se necesitan $n-1$ comparaciones. Para calcular el segundo de esos n elementos, podemos realizar $n-2$ comparaciones más (aunque hay varias formas de calcular el segundo con sólo $3\lceil \frac{n}{2} \rceil - 2$ comparaciones).

¿Cómo podríamos calcular el k-ésimo de manera eficiente? ($1 \leq k \leq n$)

Un problema interesante es calcular *la mediana* de un conjunto de n elementos, que es el $\lceil \frac{n}{2} \rceil$-ésimo elemento.
Obsérvese que, realizado de la manera trivial, tenemos un algoritmo de $\Theta(kn)$ comparaciones; y si $k = \Theta(n)$ tenemos un coste de $\Theta(n^2)$.
Si ordenamos los elementos, y devolvemos el que aparezca en la posición k, resolvemos en tiempo $O(n \lg n)$.

Selección del k-ésimo (2/3)

El algoritmo $\textsc{Partición}(T, i, j)$ puede ayudar. Recuérdese que permuta los elementos de la tabla $T(i..j)$ de manera que, con el índice p que devuelve, se satisface lo siguiente:
$\forall x.(i \leq x \leq p \rightarrow T(x) \leq T(p)) \wedge (p < x \leq j \rightarrow T(x) > T(p))$.

Si buscamos el k-ésimo de la tabla $T(i..j)$, entonces debe ser $1 \leq k \leq j - i + 1$ y el índice que le correspondería en una permutación ordenada de $T(i..j)$ sería el $s = i + k - 1$.

Podemos aplicar $p \leftarrow \textsc{Partición}(T, i, j)$ y estudiar la relación entre p y s.

Selección del k-ésimo (3/3)

Permuta los valores de $T(i \ldots j)$ y devuelve el que ocupa la posición de índice $s = i + k - 1$ de manera que $(\forall x \in [i..s-1].\, T(x) \leq T(s))$ y $(\forall x \in [s+1..j].\, T(s) < T(x))$.

func SELECCIONAR(T, i, j, k) **return** *Nat*
    $s \leftarrow i + k - 1$ {s es el k-ésimo índice relativo a $[i..j]$}
    if $i = j$ **then**
        return $T(i)$
    else
        $p \leftarrow$ PARTICIÓN(T, i, j)
        case of
            $s = p$: **return** $T(p)$
            $s < p$: **return** SELECCIONAR(T, i, $p-1$, k)
            $s > p$: **return** SELECCIONAR(T, $p+1$, j, $s-p$)

ANÁLISIS: Aunque, en el peor caso, este algoritmo es $O(n^2)$; si tenemos suerte y $p \approx \frac{n}{2}$ entonces la función de coste de SELECCIONAR(T, 1, n, k) tiene la forma $f(n) = f(n/2) + \Theta(n)$ que resulta $f(n) = \Theta(n)$. Puede hacerse un análisis más cuidadoso que demuestre que es $\Theta(n)$ en el caso medio.

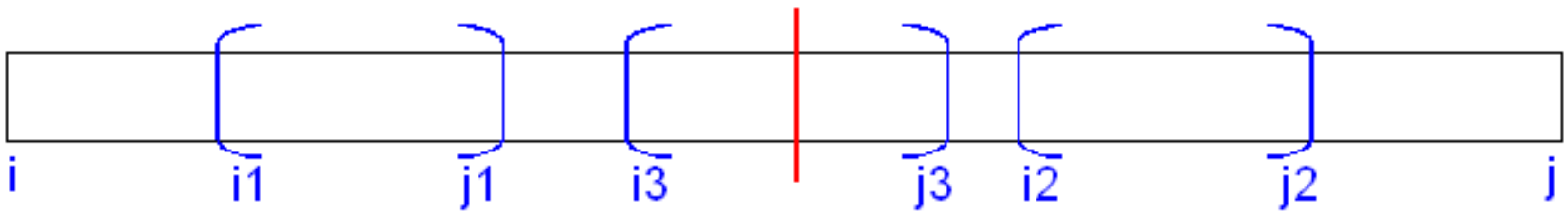
Segmento de suma máxima

Determinar dos índices i y j $(1 \leq i \leq j \leq n)$ de una tabla de enteros $V[1 \ldots n]$ tales que $\sum_{k=i}^{j} V(k)$ es el mayor valor que se puede conseguir sumando números consecutivos de la tabla $V[1 \ldots n]$.

El algoritmo de fuerza bruta, que estudia todos los segmentos posibles para quedarse con el que más suma, es $\Omega(n^2)$.

Segmento de suma máxima: Primera versión (1/4)

- Resolver el problema para cada mitad de la tabla no es suficiente. El segmento de suma máxima podría tener elementos de las dos mitades.
- El segmento de suma máxima sólo puede estar en la primera mitad, en la segunda mitad o tener elementos de la primera y segunda mitad: El mejor de estos tres casos es el que buscamos.



Segmento de suma máxima: Primera versión (2/4)

func SEG_SUMA_MAX(V, i, j)) **return** $Nat \times Nat \times Nat$
{Calcula los índices extremos del segmento y la suma
del segmento de suma máxima de la tabla $V(i..j)$
    if $i = j$ **then**
        return $(i,\ i,\ V(i))$
    else
        $(i1, j1, sum1) \leftarrow$ SEG_SUMA_MAX(V, i, $\lfloor i + j/2 \rfloor$)
        $(i2, j2, sum2) \leftarrow$ SEG_SUMA_MAX(V, $\lfloor i + j/2 \rfloor + 1$, j)
        $(i3, j3, sum3) \leftarrow$ SUMA_MAX_CENTRO(V, i, j, $\lfloor i + j/2 \rfloor$)
        if $sum2 \leq sum1 \wedge sum3 \leq sum1$ **then return** $(i1,\ j1,\ sum1)$
        if $sum1 \leq sum2 \wedge sum3 \leq sum2$ **then return** $(i2,\ j2,\ sum2)$
        if $sum1 \leq sum3 \wedge sum2 \leq sum3$ **then return** $(i3,\ j3,\ sum3)$

Segmento de suma máxima: Primera versión (3/4)

```
func SUMA_MAX_CENTRO(V, izq, der, centro) return Nat × Nat × Nat
{Calcula el segmento de suma máxima que tiene algún elemento de V(izq..centro)
y algún elemento de V(centro + 1..der)}
    partelzq ← V(centro)
    aux_izq ← centro
    parcial ← V(centro)
    for k ← centro − 1 downto izq loop
        parcial ← parcial + V(k)
        if partelzq < parcial then
            partelzq ← parcial
            aux_izq ← k
    end for
    parteDer ← V(centro + 1)
    aux_der ← centro + 1
    parcial ← V(centro + 1)
    for k ← centro + 2 to der loop
        parcial ← parcial + V(k)
        if parteDer < parcial then
            parteDer ← parcial
            aux_der ← k
    end for
    return (aux_izq, aux_der, partelzq + parteDer)
```

Segmento de suma máxima: Primera versión (4/4)

Análisis:

Es fácil comprobar que $\textsc{Suma\_Max\_Centro}(V, 1, n, \lfloor n+1/2 \rfloor)$ es $\Theta(n)$. Por lo tanto, la función de coste de $\textsc{Seg\_Suma\_Max}(V, 1, n)$ es de la forma

$$f(n) = 2f(\frac{n}{2}) + \Theta(n)$$

que es una función conocida de $\Theta(n \lg n)$

Segmento de suma máxima: Segunda versión (1/5)

Podemos mejorar la eficiencia de la versión anterior si guardamos más información de cada llamada recursiva.

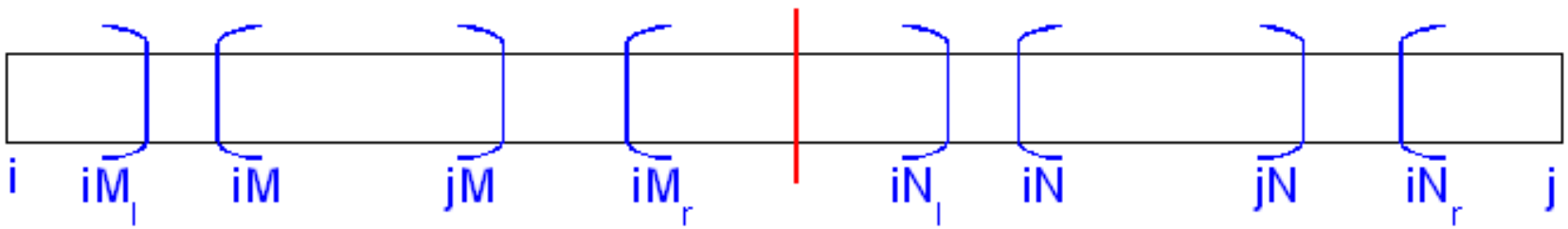
De cada llamada recursiva sobre una tabla $V(i..j)$ vamos a recopilar la siguiente información:

- máxima suma del segmento que comienza en i y el índice donde termina ese segmento.
- máxima suma del segmento que termina en j y el índice donde empieza ese segmento.
- máxima suma de un segmento de $V(i..j)$, el índice donde empieza y el índice donde termina ese segmento.
- suma de todos los valores de la tabla $V(i..j)$.

Segmento de suma máxima: Segunda versión (2/5)

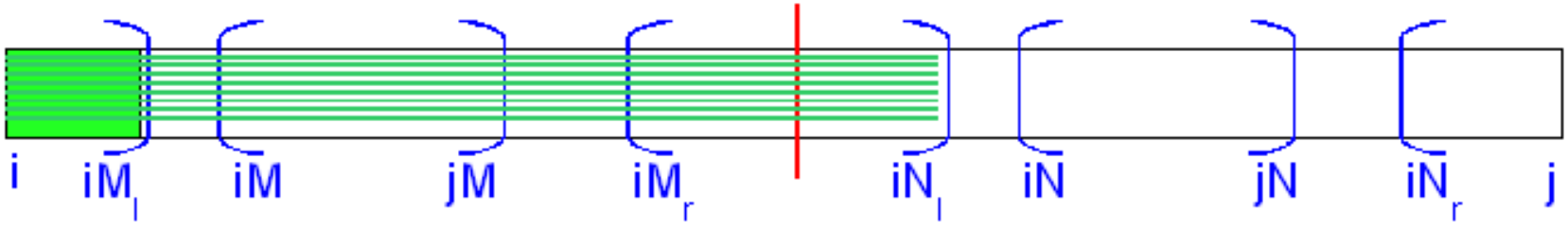
Definimos una estructura RESULTADO para recopilar esa información:
RESULTADO=$\langle$ $(Nat \times Nat)$, $(Nat \times Nat)$, $(Nat \times Nat \times Nat)$, Nat $\rangle$

Dados dos resultados de segmentos consecutivos (resultantes de llamadas recursivas para la mitad izquierda y mitad derecha de una tabla $V(i..j)$):
$\langle$ (M_l, iM_l), (M_r, iM_r), (M_{abs}, iM, jM), M_{sum} $\rangle$ y
$\langle$ (N_l, iN_l), (N_r, iN_r), (N_{abs}, iN, jN), N_{sum} $\rangle$

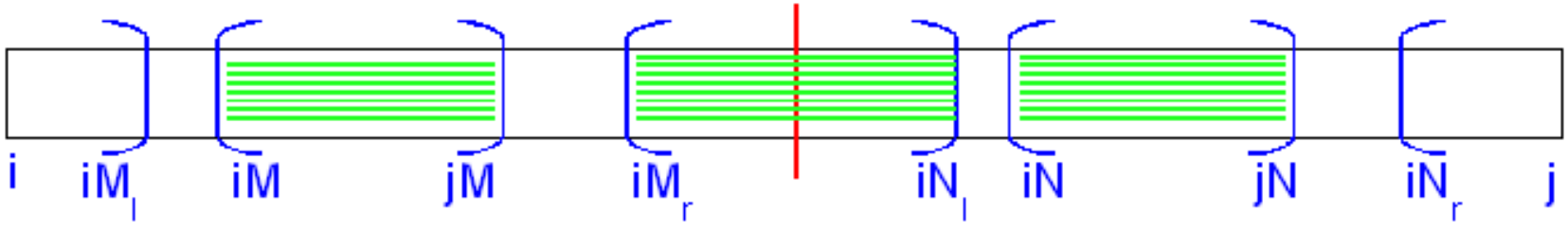


Segmento de suma máxima: Segunda versión (3/5)

Para componer el resultado de la tabla $V(i..j)$ hay que observar que la máxima suma del segmento que comienza en i puede conseguirse sumando toda la mitad izquierda y algo más. Por ejemplo, para:
$[8, -10, 8, -10, 8]$ $[8, -10, 8, -10, 8]$ el segmento sería
$[8, -10, 8, -10, 8, 8]$.
Análogamente para la máxima suma del segmento que termina en j.



La máxima suma absoluta del segmento $V(i..j)$ será el máximo de estos tres valores: $\{M_{abs},\ M_r + N_l,\ N_{abs}\}$.



Segmento de suma máxima: Segunda versión (4/5)

func $\mathrm{SSM}(V,\ i,\ j)$ **return** $\textsc{Resultado}$

  if $i = j$ **then**

    return $\langle (V(i), i), (V(i), i), (V(i), i, i), V(i)\rangle$

  else

    $\langle (M_l, iM_l), (M_r, iM_r), (M_{abs}, iM, jM), M_{sum}\rangle \leftarrow \mathrm{SSM}(V,\ i,\ \lfloor i + j/2 \rfloor)$

    $\langle (N_l, iN_l), (N_r, iN_r), (N_{abs}, iN, jN), N_{sum}\rangle \leftarrow \mathrm{SSM}(V,\ \lfloor i + j/2 \rfloor + 1,\ j)$

    $(O_l, iO_l) \leftarrow \textsc{El\_Mayor}((M_l, iM_l),\ (M_{sum} + N_l, iN_l))$

    $(O_r, iO_r) \leftarrow \textsc{El\_Mayor}((N_r, iN_r),\ (M_r + N_{sum}, iM_r))$

    $(O_{abs}, iO, jO) \leftarrow \textsc{El\_Mejor}((M_{abs}, iM, jM),\ (M_r + N_l, iM_r, iN_l),$
    $(N_{abs}, iN, jN))$

    $O_{sum} \leftarrow M_{sum} + N_{sum}$

    return $\langle (O_l, iO_l), (O_r, iO_r), (O_{abs}, iO, jO), O_{sum}\rangle$

Segmento de suma máxima: Segunda versión (5/5)

func $\textsc{El\_Mayor}((A, iA), (B, iB))$ **return** $Nat \times Nat$
    if $A > B$ **then return** (A, iA)
    else return (B, iB)

Análogamente para $\textsc{El\_Mejor}((A, iA, jA), (B, iB, jB), (C, iC, jC))$

Análisis de $\mathrm{SSM}(V, 1, n)$:
La función de coste es del estilo

$$f(n) = 2f(\frac{n}{2}) + \Theta(1)$$

que resulta ser $f(n) = \Theta(n)$