TÉCNICA DE DISEÑO

DIVIDE Y CONQUISTA

Divide y conquista

Técnica de diseño de algoritmos resuelve un problema recursivamente, aplicando tres pasos en cada nivel de la recursión:

- **Divide** el problema en un número de *subproblemas* que son instancias menores del mismo problema.
- **Conquista** los subproblemas resolviéndolos recursivamente. Si el tamaño del subproblema es suficientemente pequeño, lo resuelve de manera directa.
- **Combina** las soluciones de los subproblemas para obtener la solución al problema original.

Divide y Conquista: Problemas

Características de los problemas

- El problema debe admitir una formulación recursiva.
- Los subproblemas deben ser del mismo tipo que el problema original, pero con datos de tamaño estrictamente menor.
- El tamaño de los datos que manipulen los subproblemas ha de ser lo mas parecido posible

Divide y Conquista: Esquema algorítmico

```
DyC (P) {

    if ( SIMPLE (P) ) // P pequeño -> su solución es directa
      return Solucion_Directa (P);

    else {  // P es grande

      DIVIDE P en k Subproblemas $P_1$, $P_2$,... $P_k$,  k>1;

      return ( COMBINA ( DyC($P_1$) , DyC($P_2$), ..., DyC($P_k$) );
    }
}
```

Divide y Conquista: Análisis de Eficiencia

```
DyC (P) {
  if ( SIMPLE (P) )
       return Solucion_Directa (P);
  else{
       DIVIDE P en k Subproblemas $P_1$, $P_2$,... $P_k$,  k>1;
       return ( COMBINA ( DyC($P_1$) , DyC($P_2$), ..., DyC($P_k$) );
       }
}
```

$$T(n) = \begin{cases} g(n) & \text{n pequeño} \\ T(n_1) + T(n_2) + \ldots + T(n_k) + \mathbf{f(n)} & \text{n suficientemente grande} \end{cases}$$

T(n) es el tiempo de ejecución de DyC con n entradas,

g(n) es el tiempo para resolver directamente las entradas pequeñas

f(n) es el tiempo de dividir P en subproblemas y combinar las soluciones.

Divide y Conquista: Análisis de Eficiencia

- Nunca resuelve un problema más de una vez.
- **Divide** y **Combina** deben ser eficientes.
- El tamaño de los subproblemas debe ser lo mas parecido posible.
- Si el subproblema es suficientemente pequeño

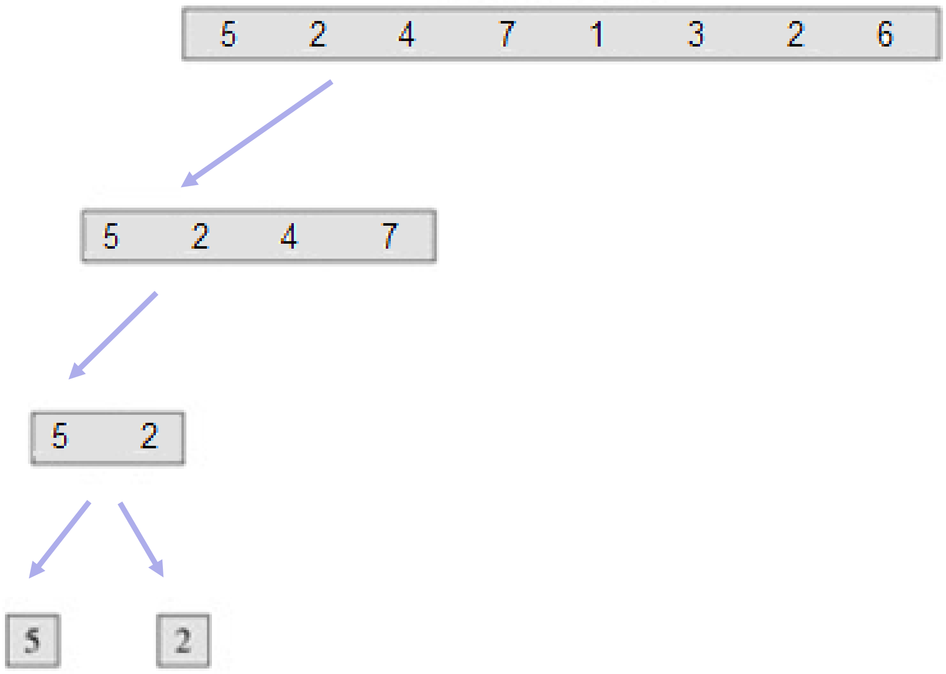
  → evitar generar nuevas llamadas recursivas

Divide y Conquista: Ejemplo

Método de ordenamiento Mergesort

Sea A = { 5, 2, 4, 7, 1, 3, 2, 6 } el arreglo a ordenar

```
MERGE-SORT (A, i, d) {
    if   i < d
         { m ← ⌊(i + d)/2⌋
         MERGE-SORT(A, i, m)
         MERGE-SORT(A, m + 1, d)
         MERGE(A, i, m, d)
         }
}
```

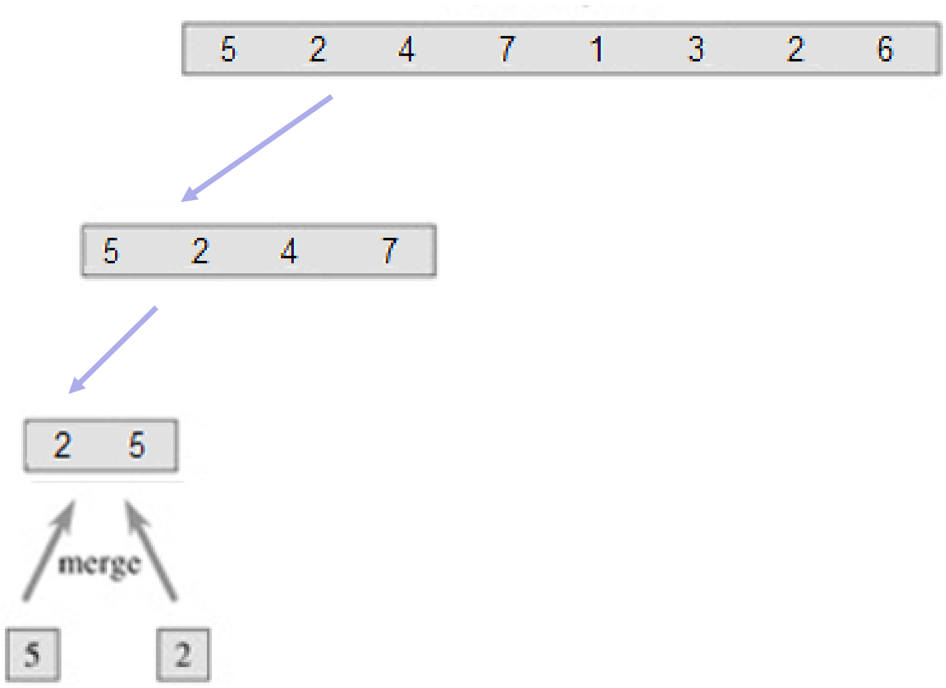


Divide y Conquista: Ejemplo

Método de ordenamiento Mergesort

Sea A = { 5, 2, 4, 7, 1, 3, 2, 6 } el arreglo a ordenar

```
MERGE-SORT (A, i, d) {
     if   $i < d$
          { $m \leftarrow \lfloor (i + d)/2 \rfloor$
            MERGE-SORT(A, i, m)
            MERGE-SORT(A, m + 1, d)
            MERGE(A, i, m, d)
            }
}
```

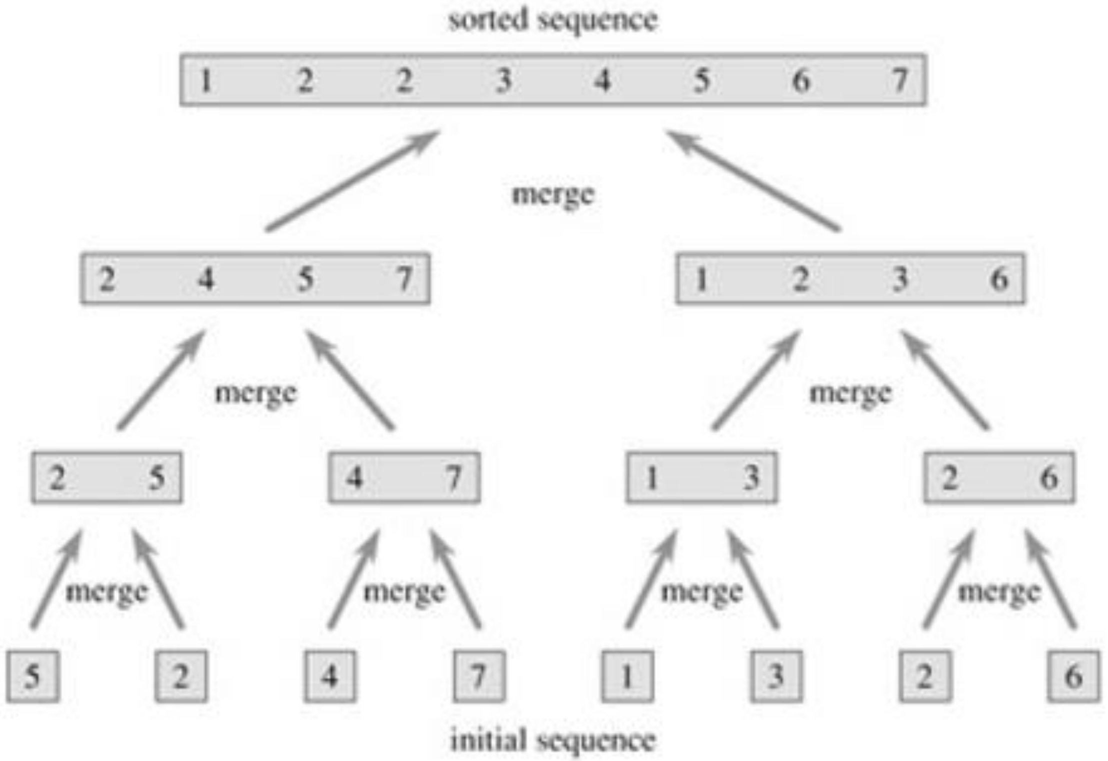


Divide y Conquista: Ejemplo

Método de ordenamiento Mergesort

Sea A = { 5, 2, 4, 7, 1, 3, 2, 6 } el arreglo a ordenar

```
MERGE-SORT (A, i, d) {
    if   i < d
            { m ← ⌊(i + d)/2⌋
            MERGE-SORT(A, i, m)
            MERGE-SORT(A, m + 1, d)
            MERGE(A, i, m, d)
            }
}
```



Divide y Conquista: Ejemplo

Método de ordenamiento Mergesort

Ventaja: el tiempo requerido por mergesort es proporcional a N log N.

$$T(n) = \begin{cases} c_0 & \text{n pequeño} \\ 2\,T(n/2) + cn_1 + c_2 & \text{n suficientemente grande} \end{cases}$$

Desventaja: requiere espacio adicional proporcional a N (para el arreglo auxiliar de la función merge).

QUICKSORT

QUICKSORT

Quicksort es un método que aplica la técnica divide y conquista para ordenar los elementos almacenados en un arreglo:

- **Divide**: Particiona un arreglo en dos subarreglos (posiblemente vacíos) A[i…p-1] y A[p+1.. d] tal que:
  - A[p] queda ordenado (en su posición final)
  - los elementos en A[i…p-1] son menores o igual que A[p] y
  - los elementos en A[p+1..d] son mayores que A[p]

QUICKSORT

Quicksort es un método que aplica la técnica divide y conquista para ordenar los elementos almacenados en un arreglo:

- **Divide**: Particiona un arreglo en dos subarreglos (posiblemente vacíos) A[i…p-1] y A[p+1.. d] tal que:
  - A[p] queda ordenado (en su posición final)
  - los elementos en A[i…p-1] son menores o igual que A[p] y
  - los elementos en A[p+1..d] son mayores que A[p]
- **Conquista**: Ordena los subarreglos A[i…p-1] y A[p+1.. d] llamando recursivamente al quicksort

QUICKSORT

Quicksort es un método que aplica la técnica divide y conquista para ordenar los elementos almacenados en un arreglo:

- **Divide**: Particiona un arreglo en dos subarreglos (posiblemente vacíos) A[i…p-1] y A[p+1.. d] tal que:
  - A[p] queda ordenado (en su posición final)
  - los elementos en A[i…p-1] son menores o igual que A[p] y
  - los elementos en A[p+1..d] son mayores que A[p]
- **Conquista**: Ordena los subarreglos A[i…p-1] y A[p+1.. d] llamando recursivamente al quicksort
- **Combina**: No hay necesidad de combinar las soluciones (por la forma que divide, ordena los subarreglos, luego todo el arreglo queda ordenado)

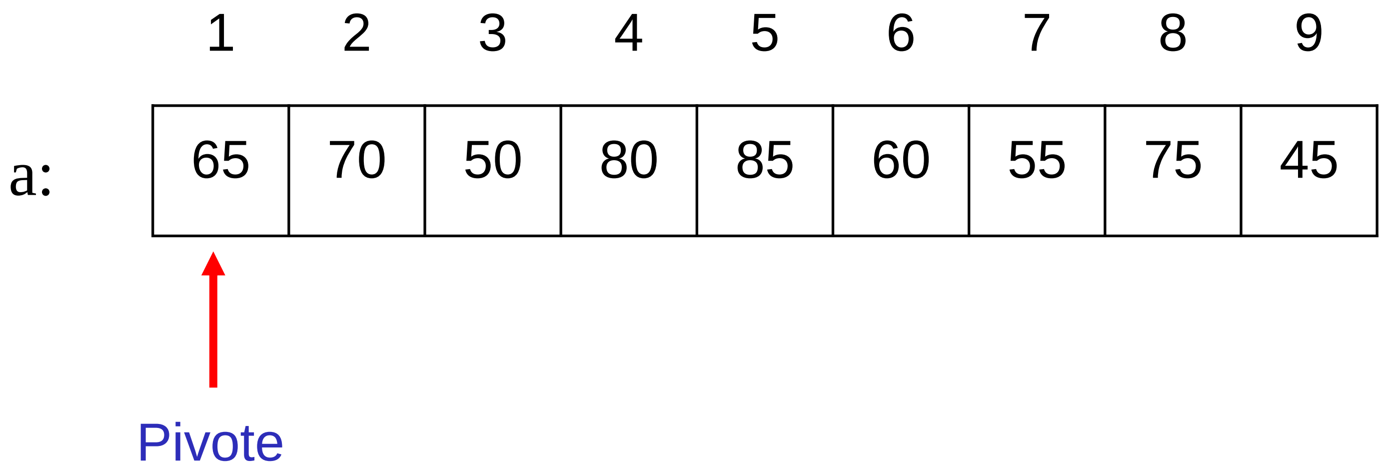
Ordena el arreglo sobre si mismo => no requiere almacenamiento adicional

QUICKSORT

```
void QUICKSORT (int A[], int i, int j) {

// ordena los elementos a[i], a[i+1], ... , a[j-1], a[j]
// en orden ascendente

   if  (i < j)  {
        // Si hay más de un elemento divide el problema de
        // ordenar a en dos subproblemas
      int p =  PARTICION ( A, i, j );
       // p es la posición del pivote

      QUICKSORT (A, i, p-1);//resuelve los subproblemas

      QUICKSORT (A, p+1, j);

     //No hay necesidad de combinar las soluciones.
       }
}
```

QUICKSORT : Partición

Primer paso: selecciona un pivote (a[1])



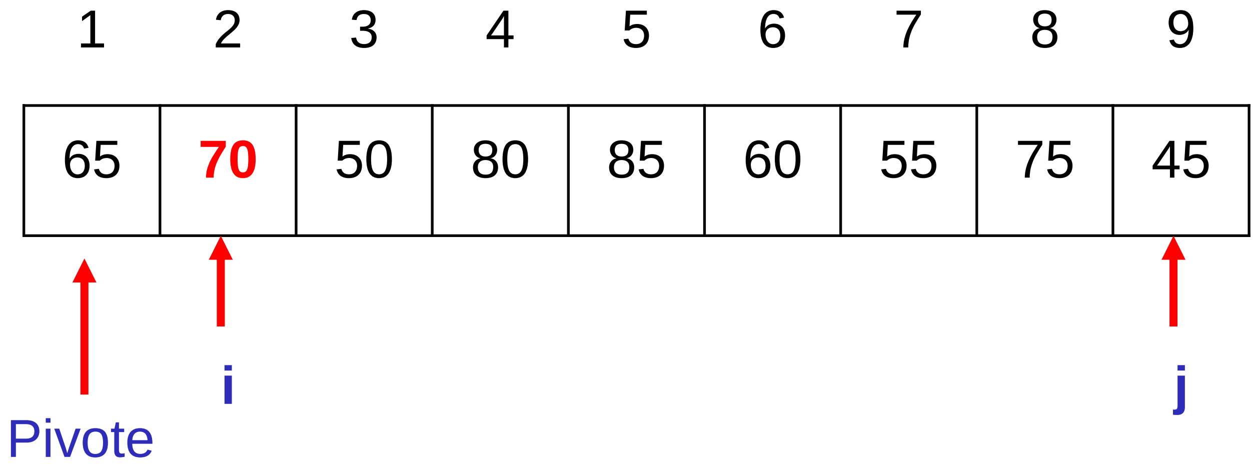
	1	2	3	4	5	6	7	8	9
a:	65	70	50	80	85	60	55	75	45

Pivote

Segundo paso: reordena los otros elementos de modo tal que:

- el pivote queda ordenado
- los elementos menores al pivote quedan a su izquierda
- los elementos mayores al pivote quedan a su derecha

QUICKSORT : Partición

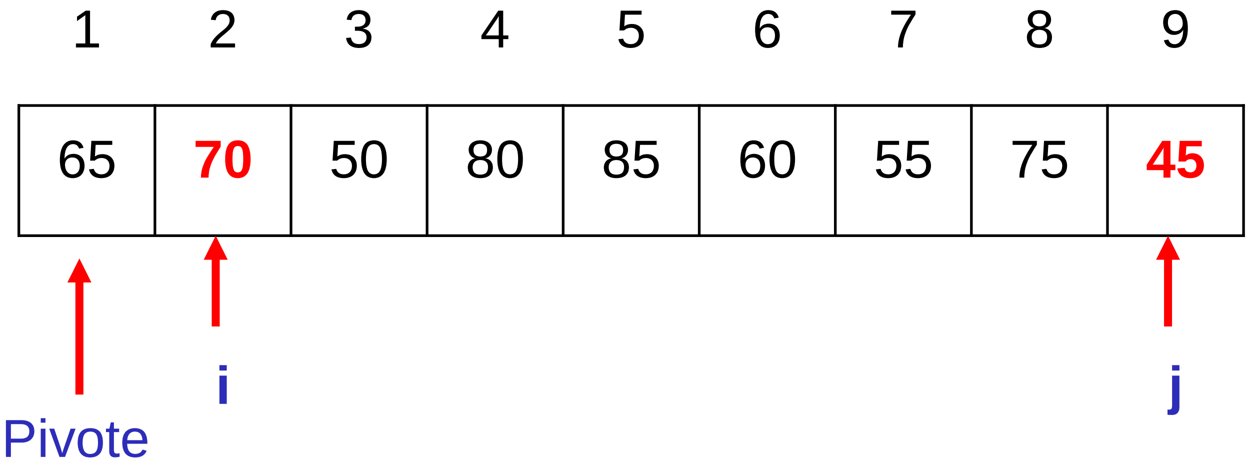


1	2	3	4	5	6	7	8	9
65	**70**	50	80	85	60	55	75	45

Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**70**	50	80	85	60	55	75	**45**

Pivote

i

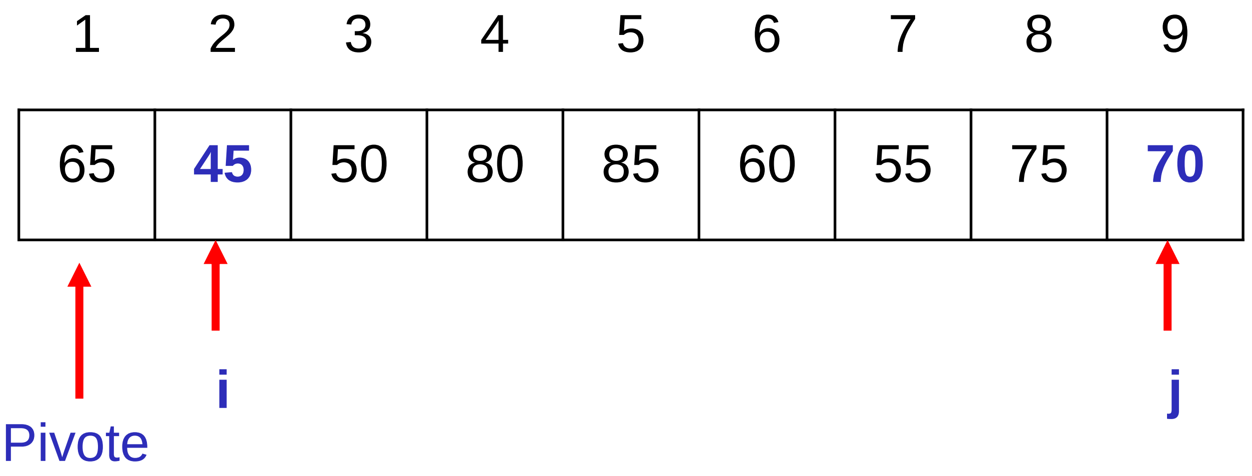
j

Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	50	80	85	60	55	75	**70**

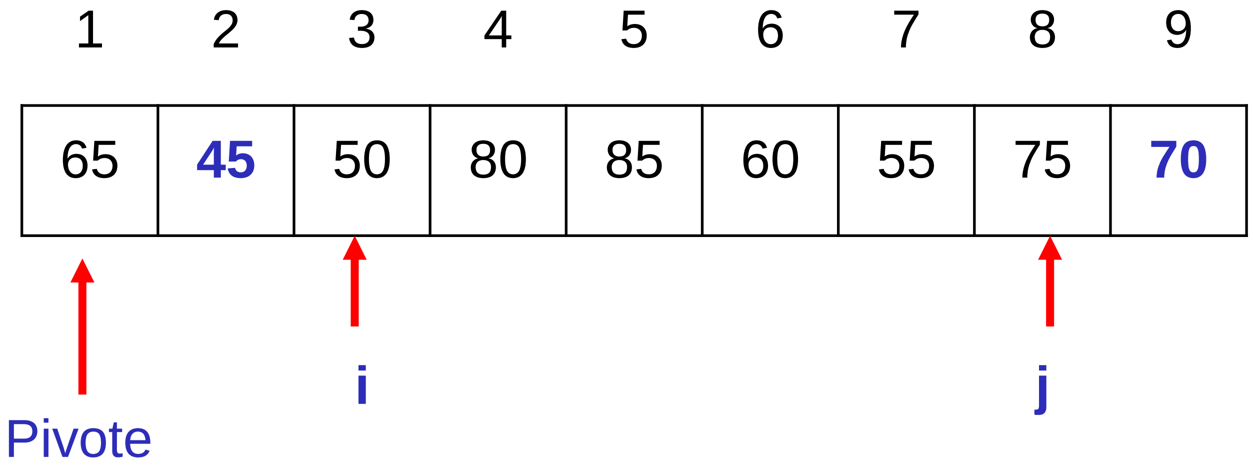
Pivote

i

j

```
Mientras i < j
    Mientras a[i] ≤ pivote -> avanza i
    Mientras a[j] > pivote -> retrocede j
    intercambia a[i] con a[j] y avanza i y retrocede j
```

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	50	80	85	60	55	75	**70**

Pivote

i

j

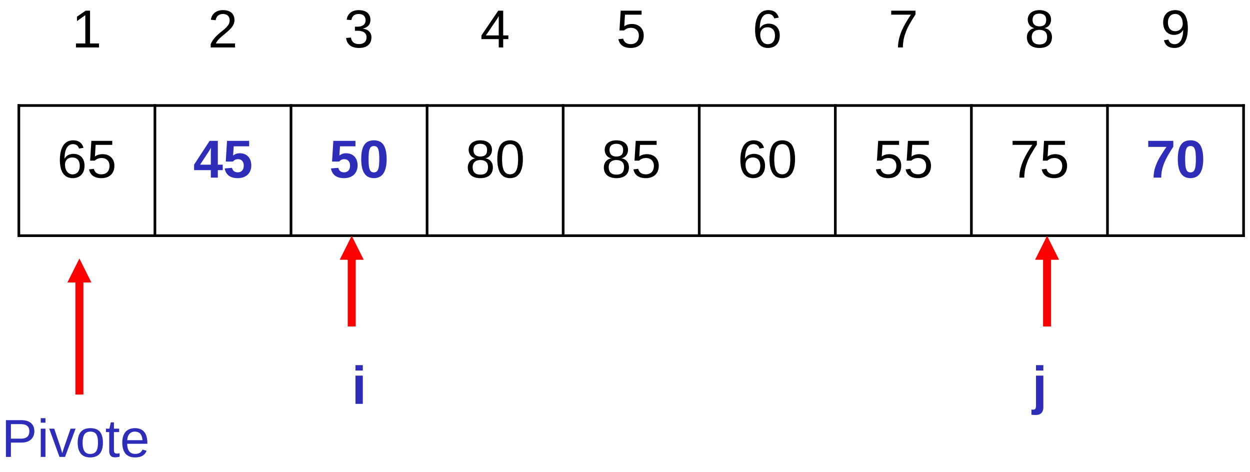
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

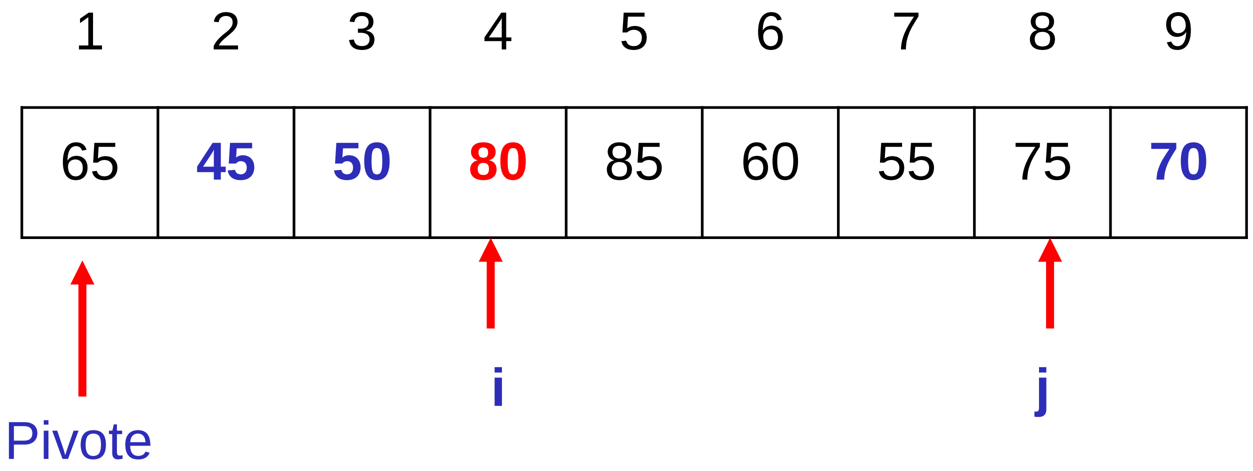


1	2	3	4	5	6	7	8	9
65	**45**	**50**	80	85	60	55	75	**70**

Mientras i < j

- Mientras a[i] ≤ pivote -> avanza i
- Mientras a[j] > pivote -> retrocede j
- intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**80**	85	60	55	75	**70**

Pivote

i

j

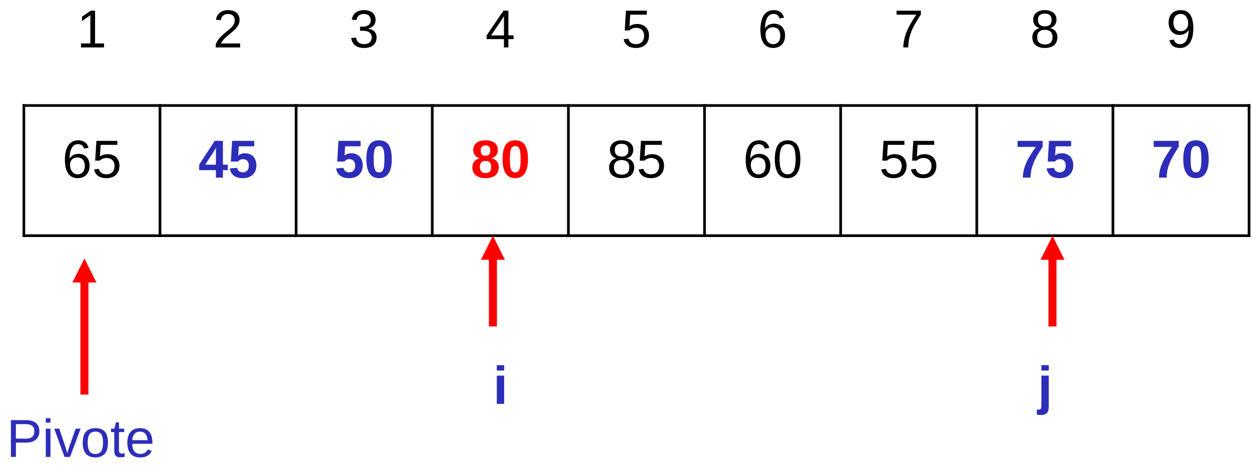
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**80**	85	60	55	**75**	**70**

i

j

Pivote

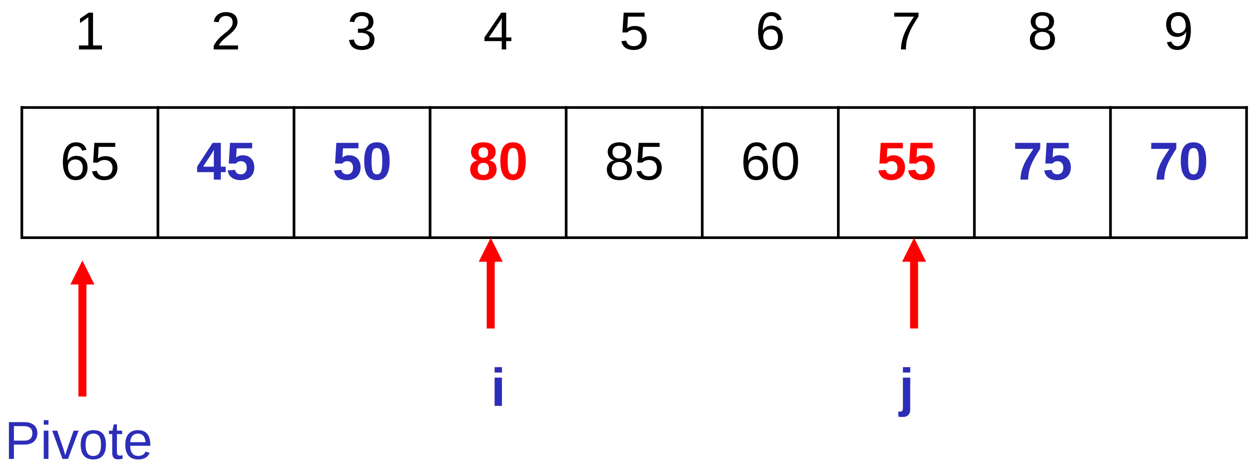
Mientras i < j

    Mientras a[i] ≤ pivote -> avanza i

    Mientras a[j] > pivote -> retrocede j

    intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**80**	85	60	**55**	**75**	**70**

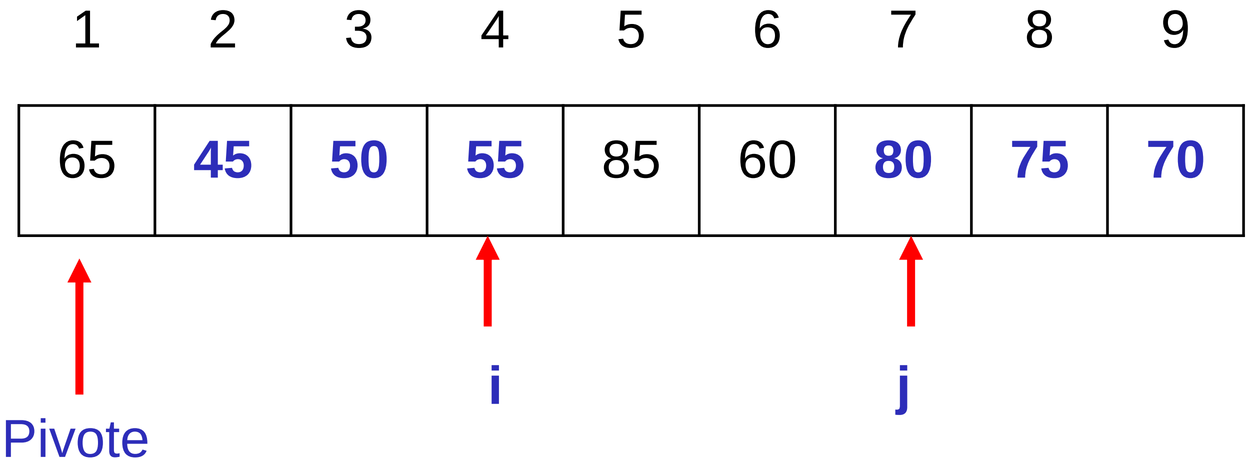
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	85	60	**80**	**75**	**70**

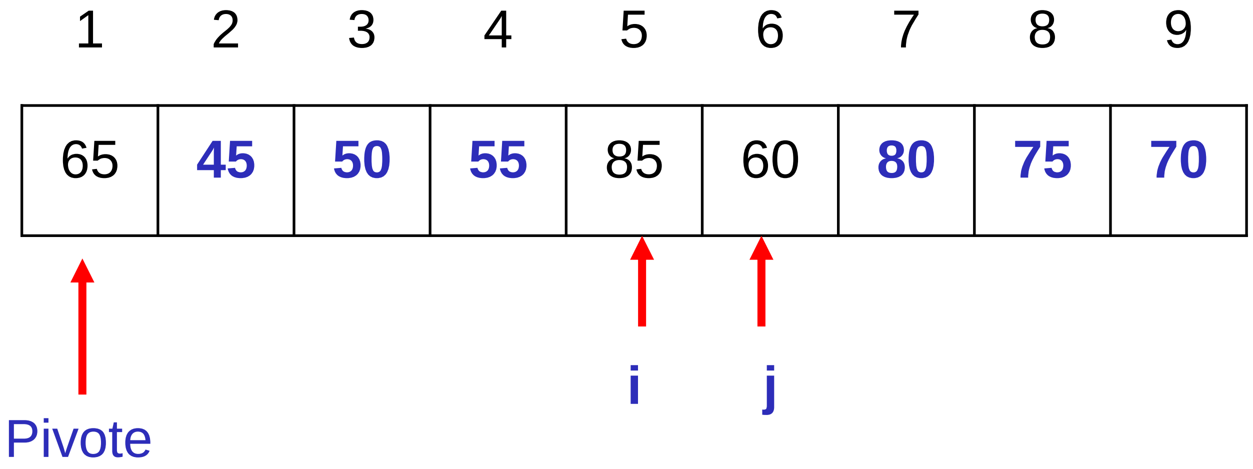
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	85	60	**80**	**75**	**70**

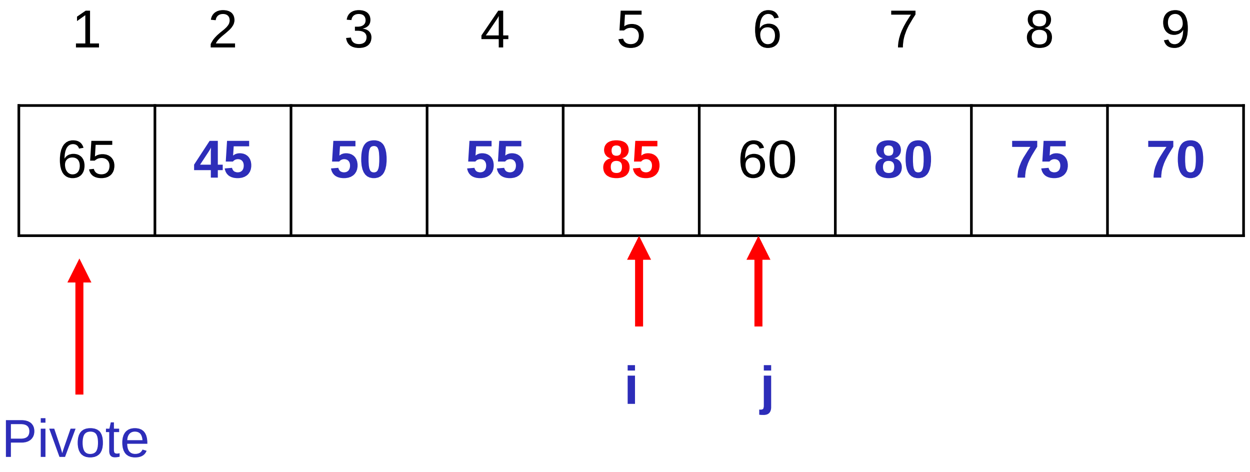
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**85**	60	**80**	**75**	**70**

i **j**

Pivote

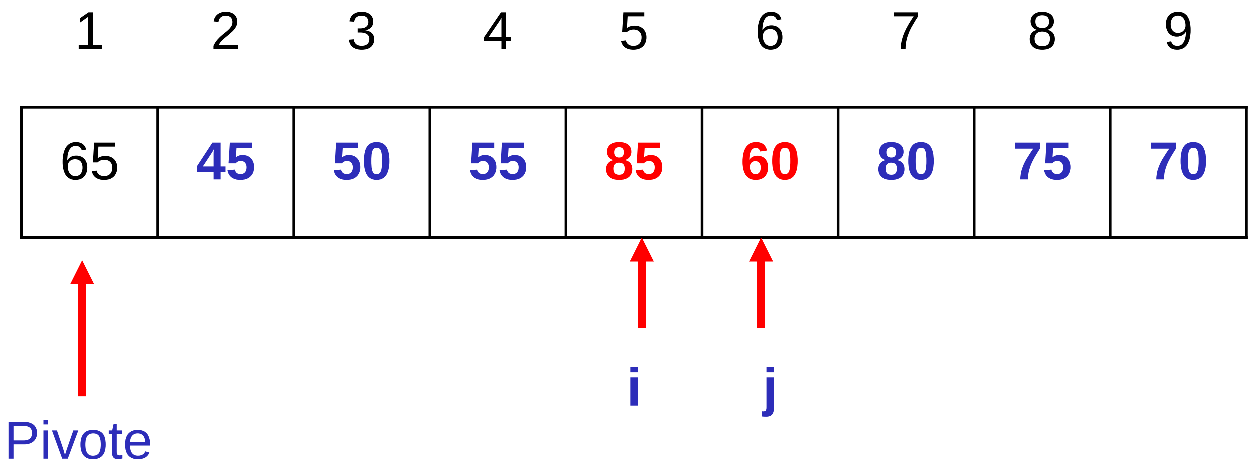
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**85**	**60**	**80**	**75**	**70**

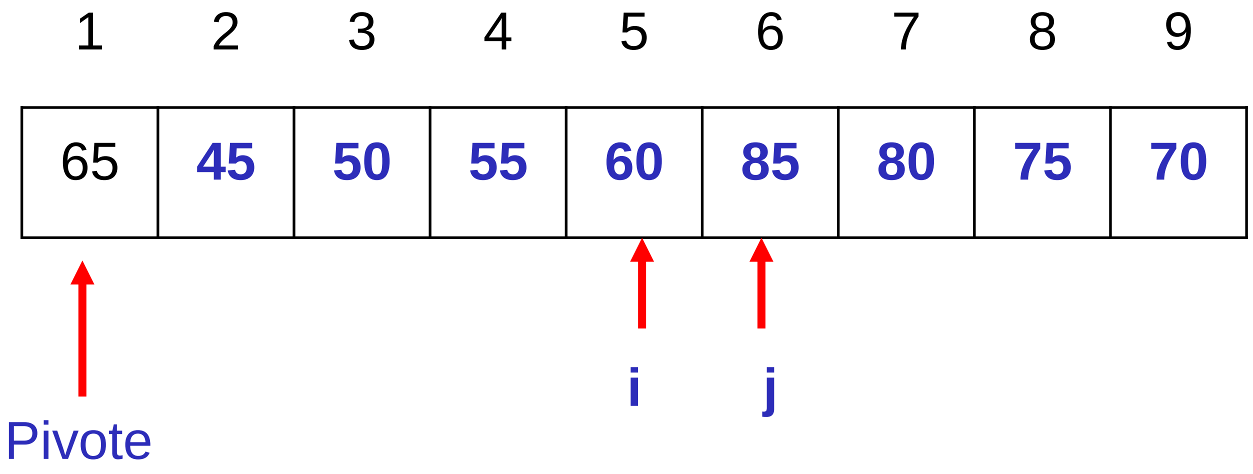
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

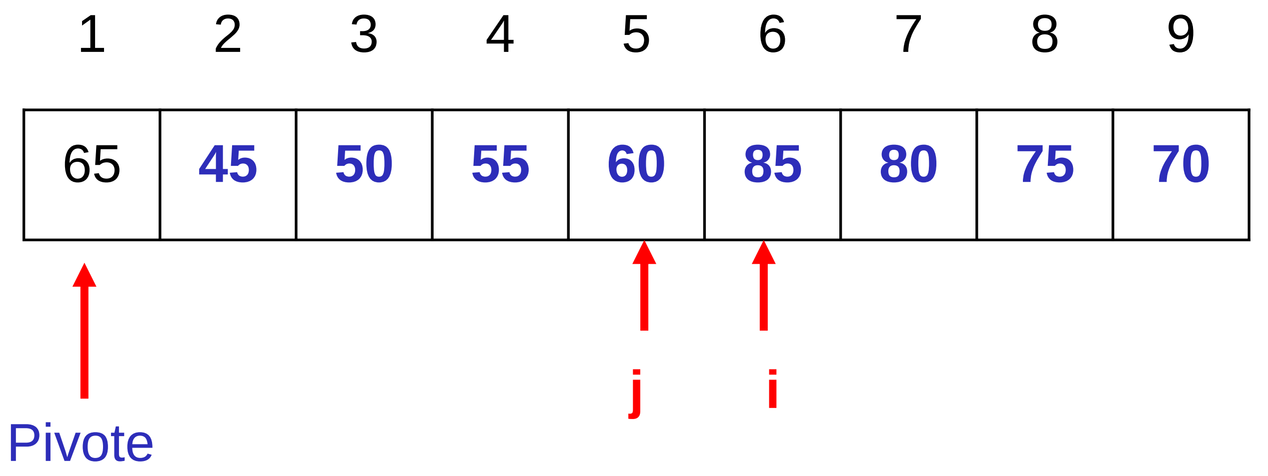
QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**

```
Mientras i < j
    Mientras a[i] ≤ pivote -> avanza i
    Mientras a[j] > pivote -> retrocede j
    intercambia a[i] con a[j] y avanza i y retrocede j
```

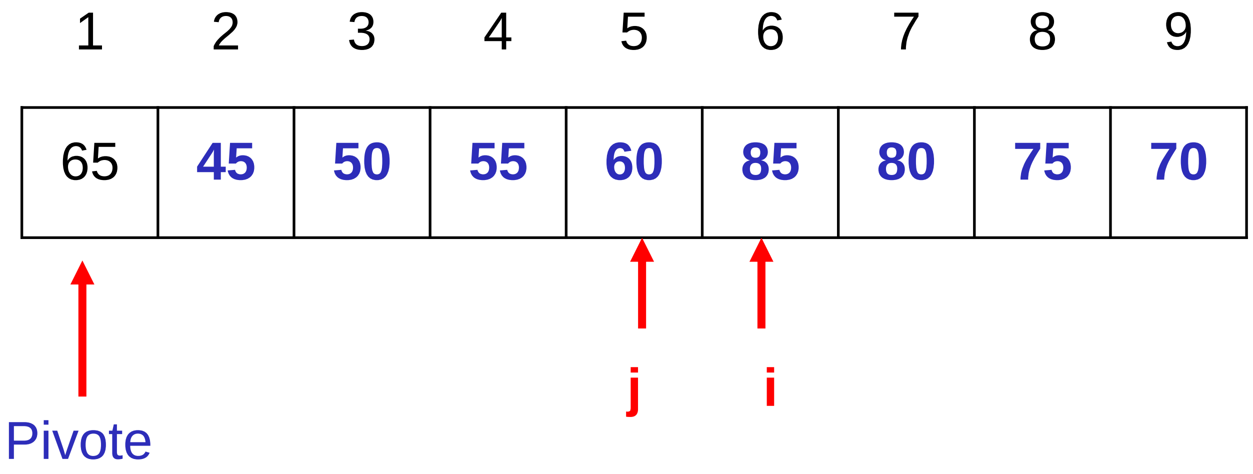
QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**

```
Mientras i < j
    Mientras a[i] ≤ pivote -> avanza i
    Mientras a[j] > pivote -> retrocede j
    intercambia a[i] con a[j] y avanza i y retrocede j
```

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**

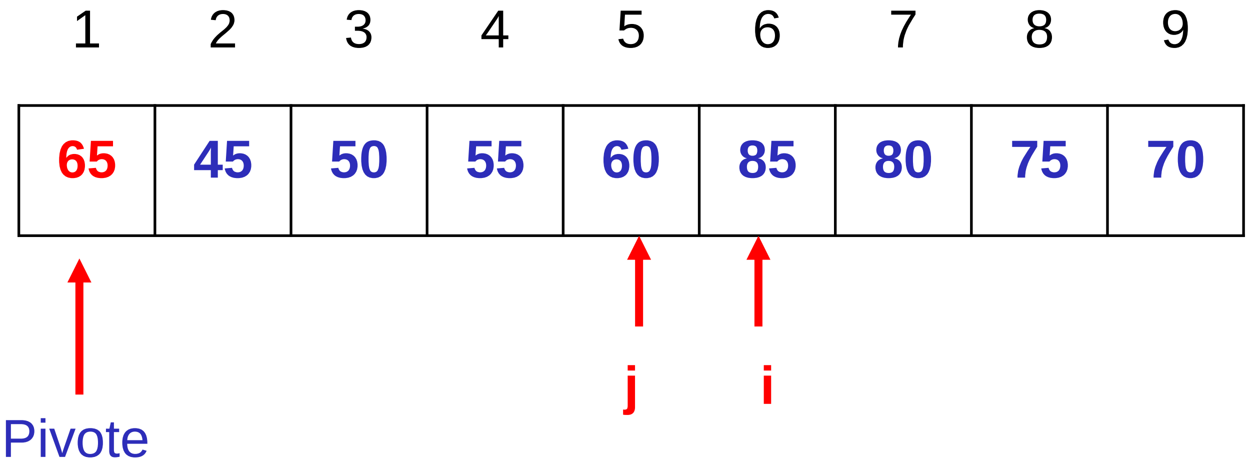
Mientras i < j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición



1	2	3	4	5	6	7	8	9
65	45	50	55	60	85	80	75	70

i > j =>

Intercambia el pivote con a[j]

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**

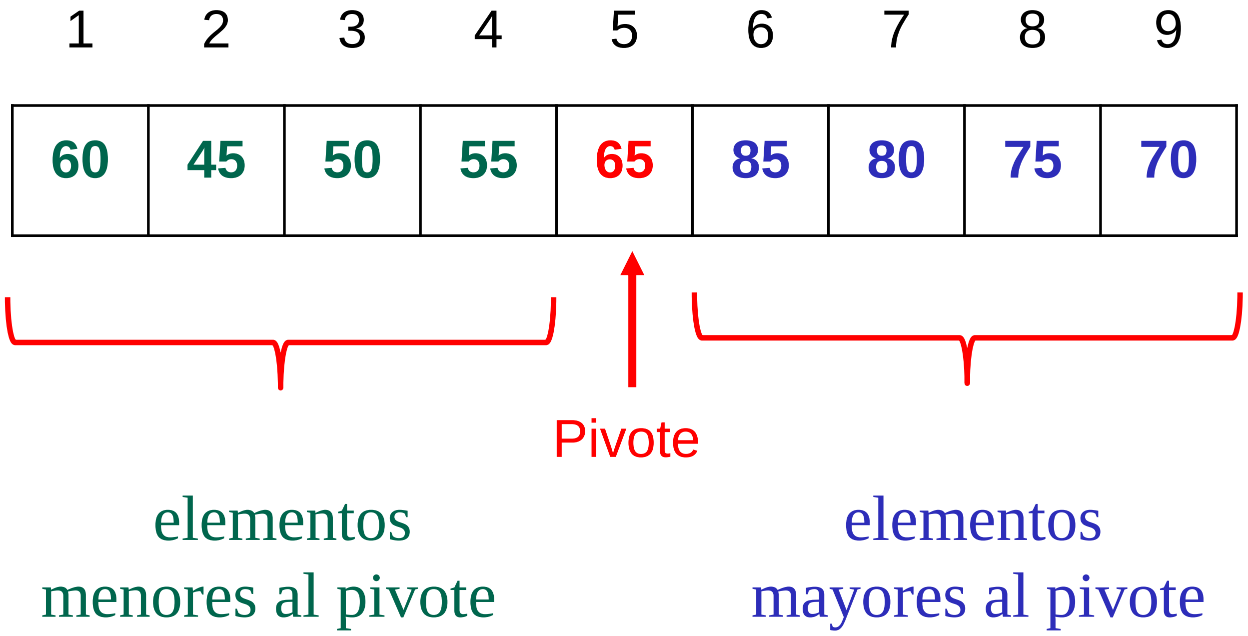
↑

Pivote

i > j =>

Intercambia el pivote con a[j]

QUICKSORT : Partición



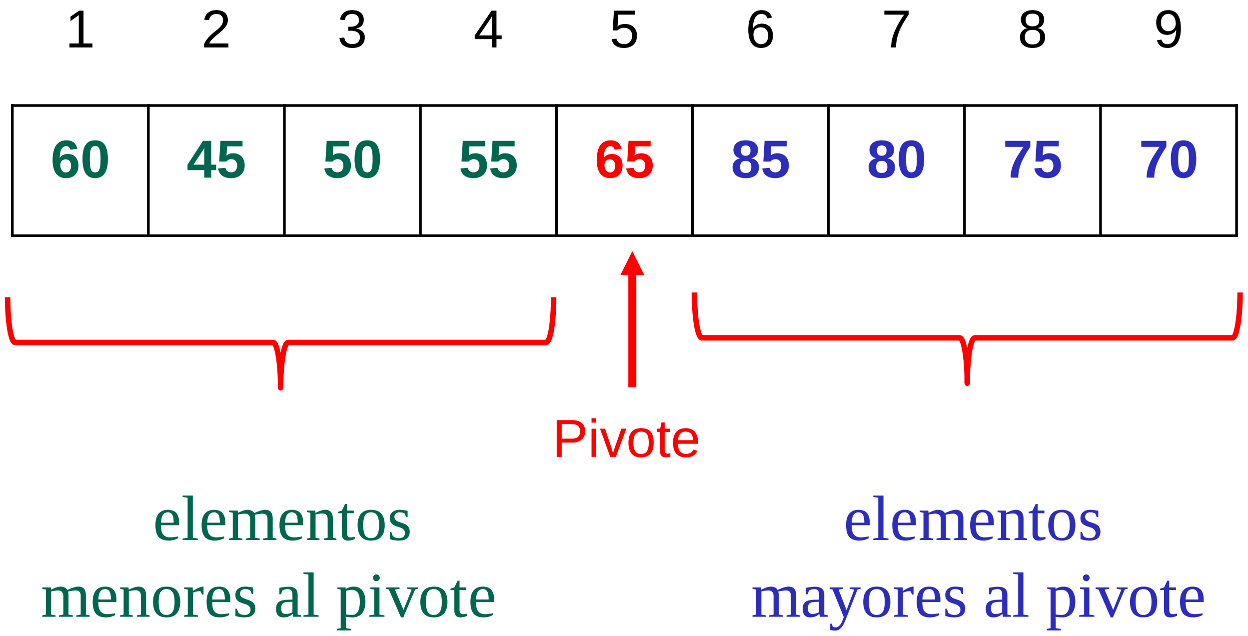
1	2	3	4	5	6	7	8	9
60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**

Pivote

elementos
menores al pivote

elementos
mayores al pivote

QUICKSORT



1	2	3	4	5	6	7	8	9
60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**

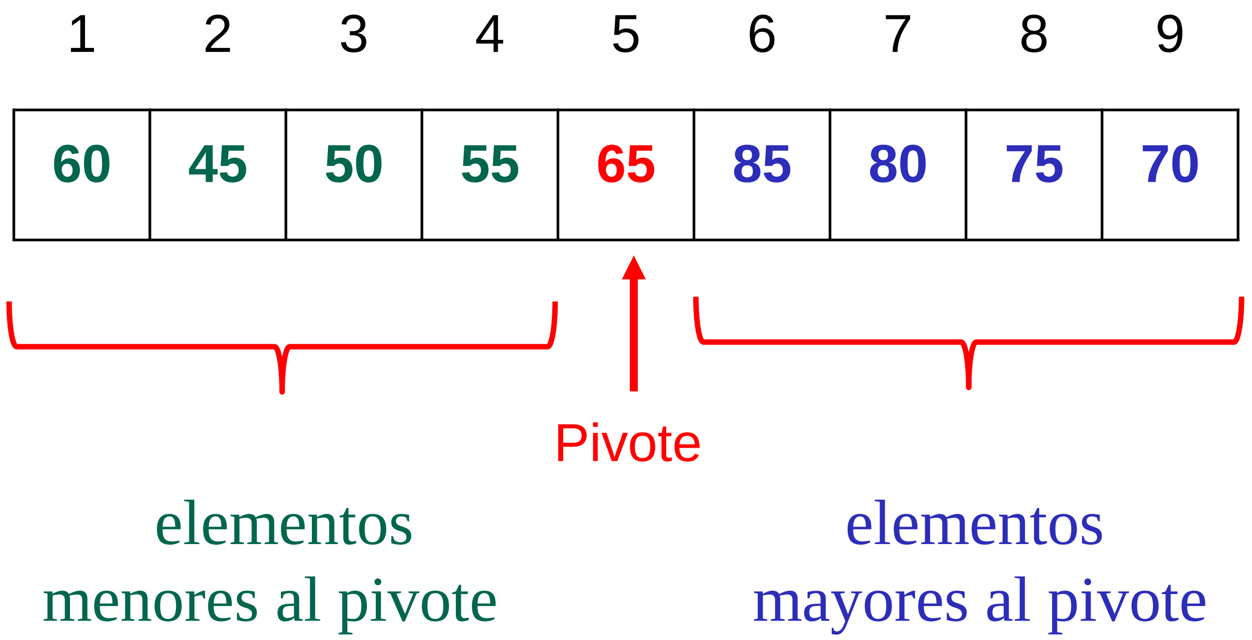
Pivote

elementos
menores al pivote

elementos
mayores al pivote

Una vez realizada la partición, cada subarreglo es ordenado llamando recursivamente a quicksort

QUICKSORT



1	2	3	4	5	6	7	8	9
60	45	50	55	65	85	80	75	70

Pivote

elementos
menores al pivote

elementos
mayores al pivote

Una vez realizada la partición, cada subarreglo es ordenado llamando recursivamente a quicksort

Una vez ordenados los subarreglos, todo el arreglo está ordenado → No hace falta combinar.

QUICKSORT

```
void QUICKSORT (int A[], int i, int j) {

// ordena los elementos a[i], a[i+1], ... , a[j-1], a[j]
// en orden ascendente

   if  (i < j)  {
        // Si hay más de un elemento divide el problema de
        // ordenar a en dos subproblemas
      int p =  PARTICION ( A, i, j );
       // p es la posición del pivote

      QUICKSORT (A, i, p-1);//resuelve los subproblemas

      QUICKSORT (A, p+1, j);

     //No hay necesidad de combinar las soluciones.
        }
}
```

QUICKSORT: Complejidad temporal

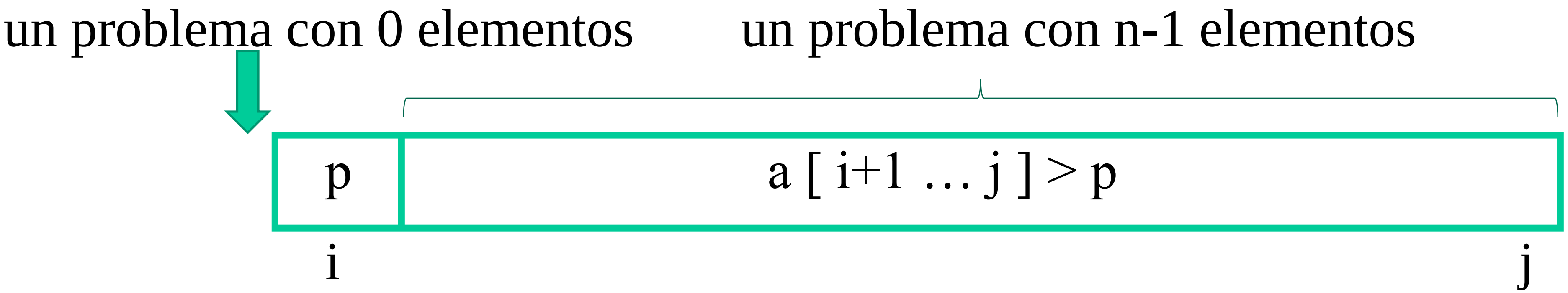
El tiempo de ejecución depende de la partición:

partición balanceada ⇨ el algoritmo corre asintóticamente tan rápido como el ordenamiento **mergesort**.

partición desbalanceada ⇨ el algoritmo corre asintóticamente tan lento como el ordenamiento por **inserción**.

QUICKSORT: Complejidad temporal

El **peor caso** ocurre cuando la partición produce:



un problema con 0 elementos

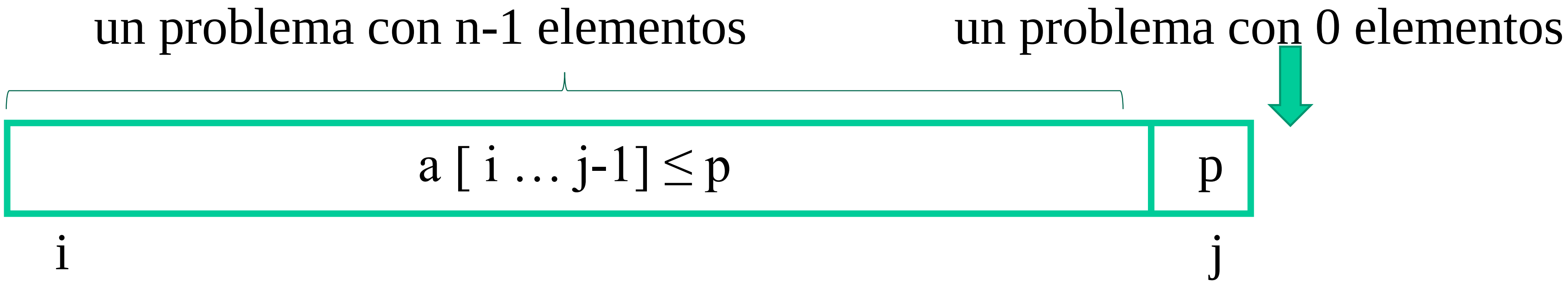
un problema con n-1 elementos

p

a [i+1 … j] > p

i

j



un problema con n-1 elementos

un problema con 0 elementos

a [i … j-1] ≤ p

p

i

j

QUICKSORT: Complejidad temporal

El **peor caso**: la partición desbalanceada ocurre en cada llamada recursiva =>

$$T(n) \begin{cases} c_0 & n \leq 1 \\ T(n-1) + c\, n_1 + c_2 & n > 1 \end{cases}$$

El tiempo de particionar $\in O(n)$

El tiempo de llamar recursivamente sobre un arreglo de tamaño n-1 es T(n-1)

El tiempo de llamar recursivamente sobre un arreglo de tamaño 0 $\in O(1)$

QUICKSORT: Complejidad temporal

El **peor caso**: la partición desbalanceada ocurre en cada llamada recursiva =>

$$T(n) \begin{cases} c_0 & n \leq 1 \\ T(n-1) + c\, n_1 + c_2 & n > 1 \end{cases}$$

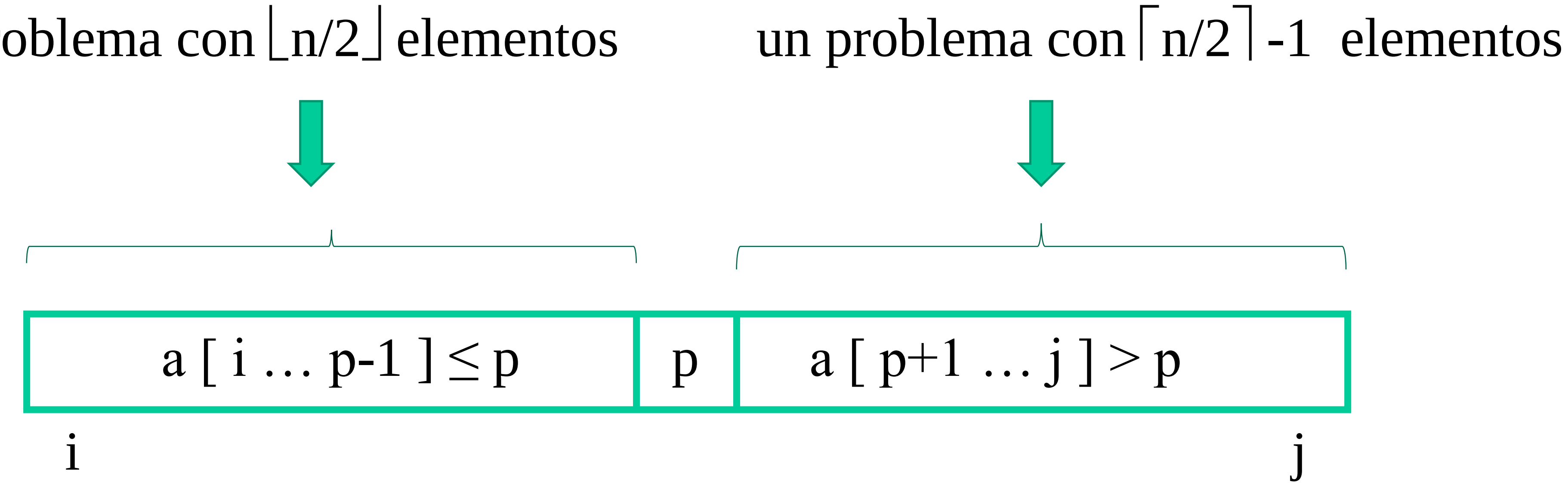
$$T(n) \in O(n^2)$$

El tiempo en el peor de los casos:

- No es mejor que el ordenamiento por inserción
- El peor tiempo ocurre cuando el arreglo ya está ordenado

QUICKSORT: Complejidad temporal

El **mejor caso** ocurre cuando la partición produce:



un problema con $\lfloor n/2 \rfloor$ elementos

un problema con $\lceil n/2 \rceil$-1 elementos

a [i … p-1] $\leq$ p

p

a [p+1 … j] $>$ p

i

j

QUICKSORT: Complejidad temporal

El **mejor caso**: la partición balanceada ocurre en cada llamada recursiva =>

$$T(n) \begin{cases} c_0 & n \leq 1 \\ 2\,T(n/2) + c\,n_1 + c_2 & n > 1 \end{cases}$$

$$T(n) \in O(\,n \log n)$$

El tiempo de llamar recursivamente sobre un arreglo de tamaño a lo sumo n/2

El tiempo de particionar $\in O(n)$

QUICKSORT: Complejidad temporal

El **caso promedio** es mucho más cercano al mejor caso que al peor caso.

El tiempo esperado es O (n log n)

QUICKSORT: Complejidad temporal

El **caso promedio** es mucho más cercano al mejor caso que la peor caso.

El tiempo esperado es O (n log n)

Consideraciones :

- Si el peor caso se da cuando el arreglo está ordenado:

  en lugar de seleccionar el primer elemento como pivote →

  ○ elegir la mediana de tres valores del arreglo (Ej: primero, medio y último)

QUICKSORT: Complejidad temporal

El **caso promedio** es mucho más cercano al mejor caso que la peor caso.

El tiempo esperado es O (n log n)

Consideraciones :

- Si el peor caso se da cuando el arreglo está ordenado:

  en lugar de seleccionar el primer elemento como pivote →

  - elegir la mediana de tres valores del arreglo (Ej: primero, medio y último)
  - seleccionar un pivote al azar (Quicksort randomizado)

QUICKSORT: Complejidad temporal

El **caso promedio** es mucho más cercano al mejor caso que la peor caso.

El tiempo esperado es O (n log n)

Consideraciones:

- Si el peor caso se da cuando el arreglo está ordenado:

  en lugar de seleccionar el primer elemento como pivote →

  - elegir la mediana de tres valores del arreglo (Ej: primero, medio y último)
  - seleccionar un pivote al azar (Quicksort randomizado)

- **Mejora**: cuando los subproblemas son pequeños (n=16) entonces usar un algoritmo de ordenamiento iterativo simple como el de inserción

QUICKSORT

Pensar:

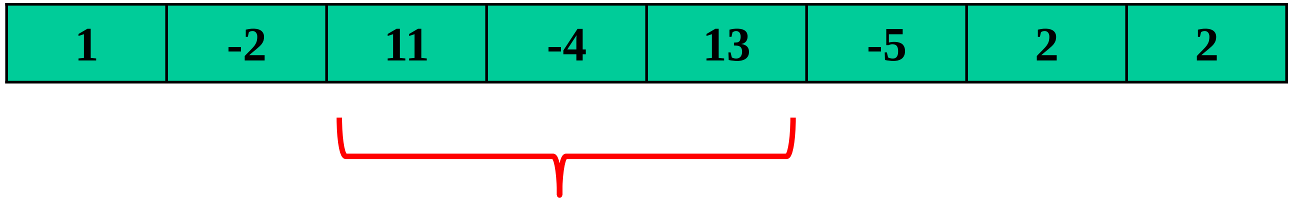
- ✓ Otra forma de particionar el arreglo
- ✓ ¿Cómo particionar el arreglo cuando existen muchos elementos repetidos?

Problema de la subsecuencia de suma máxima

Subsecuencia de suma máxima

Dada una secuencia de n enteros cualesquiera a1,a2,...,an, necesitamos encontrar la subsecuencia que maximice la suma parcial de elementos consecutivos.

Por ejemplo:



1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

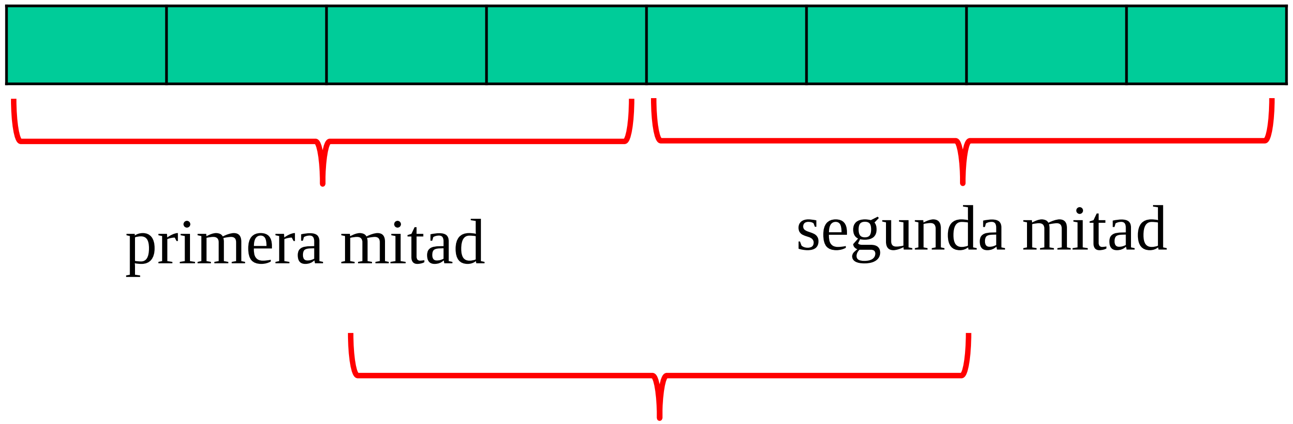
Subsecuencia de suma máxima → suma = 20

El problema es interesante sólo si el arreglo contiene números negativos.
Si los números fuesen positivos la solución sería el arreglo entero.

Subsecuencia de suma máxima

La técnica Divide y Conquista sugiere dividir el arreglo en 2 subarreglos de igual tamaño.

La subsecuencia de suma máxima (SSM) caerá en alguno de los siguientes lugares:



primera mitad

segunda mitad

cruza el punto medio

Subsecuencia de suma máxima

Por lo tanto, podemos:

- Encontrar la SSM en la primer mitad y la SSM en la segunda mitad recursivamente

  (ya que estos 2 subproblemas son instancias menores del problema de hallar la SSM)

- Encontrar la SSM que cruza el punto medio y nos quedamos con la subsecuencia de mayor suma de las tres.

Subsecuencia de suma máxima

Algoritmo para resolver el problema de la SSM por divide y conquista:

```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto == bajo)                    // caso base: solo un elemento
        return  < bajo, alto, A[bajo] >;

    else  {                              // cálculo de subproblemas
        medio = (alto + bajo) / 2;

        // Subproblema: Parte izquierda
        <bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);

        // Subproblema: Parte derecha
        <bajoDer, altoDer,sumaDer> = Encontrar_SSM (A, medio+1, alto);
```

Subsecuencia de suma máxima

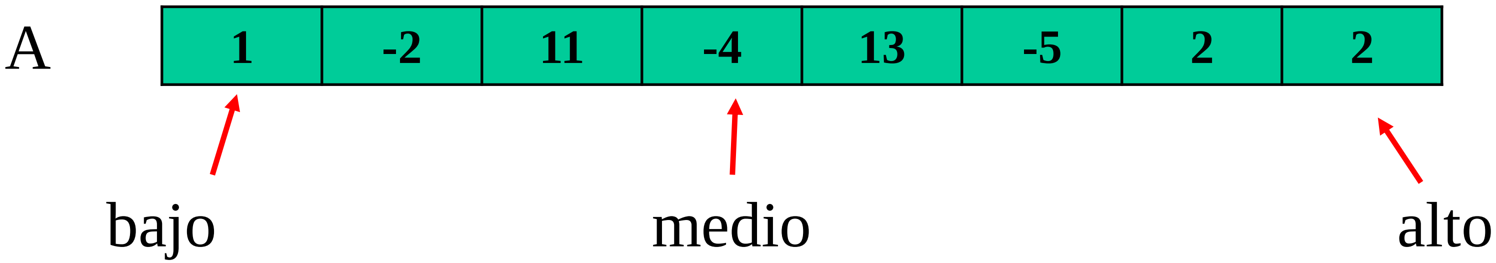
```
Encontrar_SSM ( A, bajo, alto ) {
    …
    …
    // Combinar soluciones
    <bajoMedio, altoMedio, sumaMedio> =
                            SolucionMedio( A, bajo, medio, alto);

    if ((sumaIzq > sumaDer ) and ( sumaIzq > sumaMedio))
          return < bajoIzq, altoIzq, sumaIzq>;

    else if ((sumaDer >= sumaIzq) and (sumaDer > = sumaMedio))
         return < bajoDer, altoDer, sumaDer>;

    else return < bajoMedio, altoMedio, sumaMedio>
```

Subsecuencia de suma máxima

Veamos primero cómo encontrar la SSM que cruza el punto medio…

Este problema **no** es una instancia menor del problema original, ya que tiene la restricción adicional que el subarreglo debe cruzar el punto medio.

SoluciónMedio (A, bajo, medio, alto)



A

1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

bajo

medio

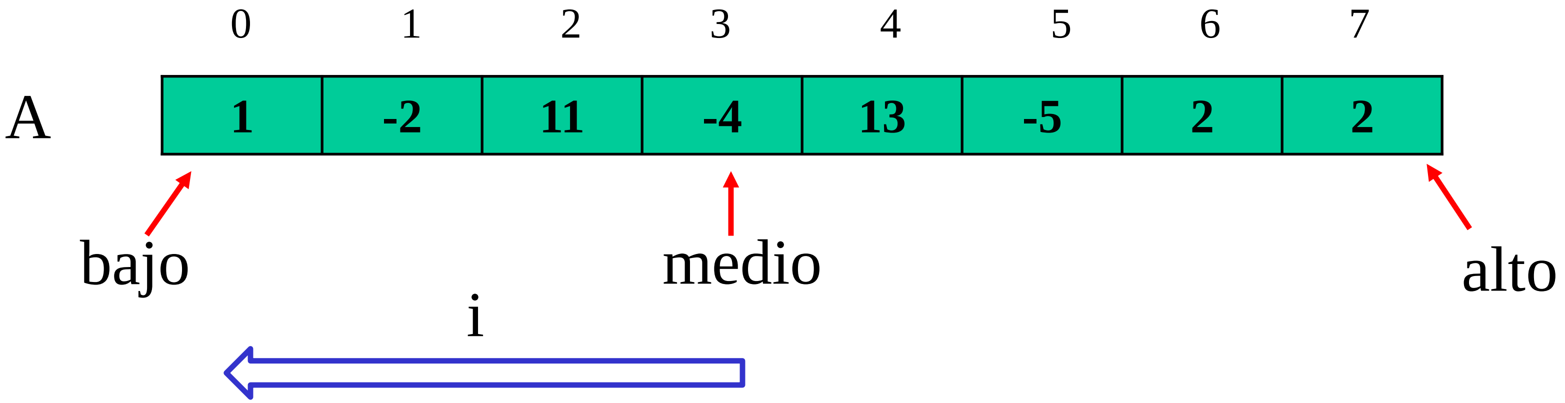
alto

Subsecuencia de suma máxima

```
SoluciónMedio ( A, bajo, medio, alto) {
        sumaIzq = -∞
        suma = 0
        for i = medio hasta bajo
                suma += A[ i ]
                if (suma > sumaIzq )
                        sumaIzq = suma
                        indiceIzq = i
        sumaDer = -∞
        suma = 0
        for j = medio + 1 hasta alto
                suma += A[ j ]
                if (suma > sumaDer )
                        sumaDer = suma
                        indiceDer = j
        return < indiceIzq, IndiceDer, sumaIzq + sumaDer >
}
```

O (n)

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	1	-2	11	-4	13	-5	2	2

bajo

medio

alto

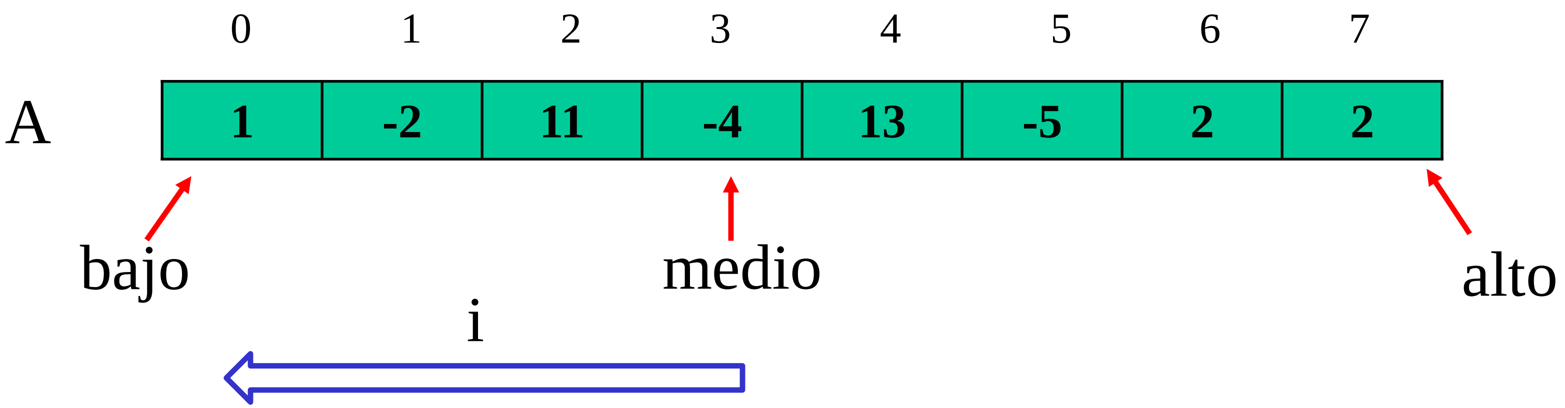
i

Suma = 0

SumaIzq = - ∞

```
sumaIzq = -∞
suma = 0
for i = medio hasta bajo
    suma += A[ i ]
    if (suma > sumaIzq )
        sumaIzq = suma
        indiceIzq = i
```

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

bajo

medio

alto

i

Suma = 0

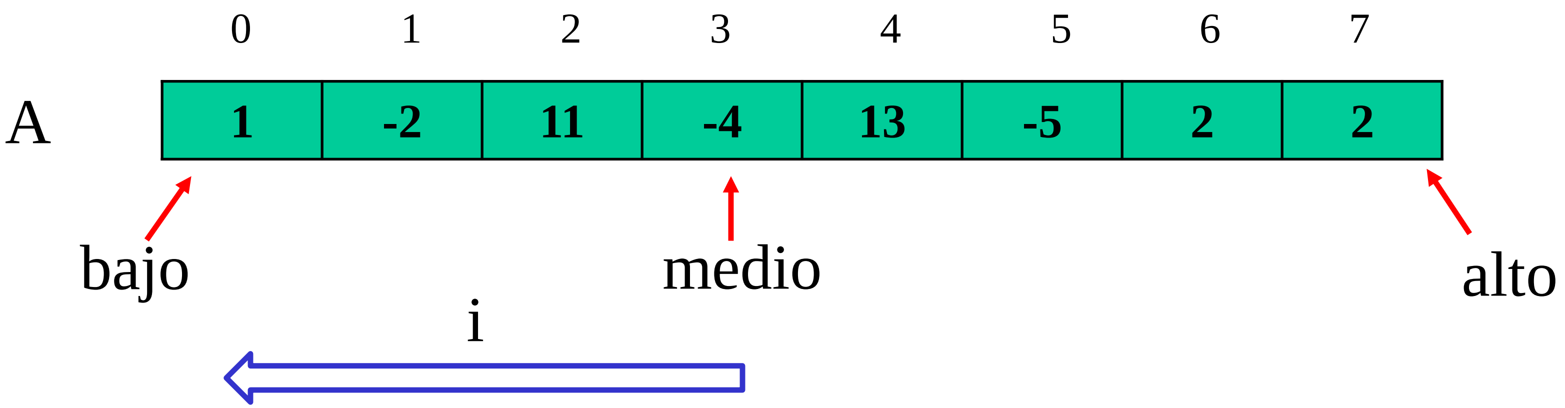
Suma + A[medio] = -4

Suma > sumaIzq

SumaIzq = - ∞

```
sumaIzq = -∞
suma = 0
for i = medio hasta bajo
    suma += A[ i ]
    if (suma > sumaIzq )
        sumaIzq = suma
        indiceIzq = i
```

Subsecuencia de suma máxima



Suma = 0

Suma + A[medio] = -4

Suma > sumaIzq

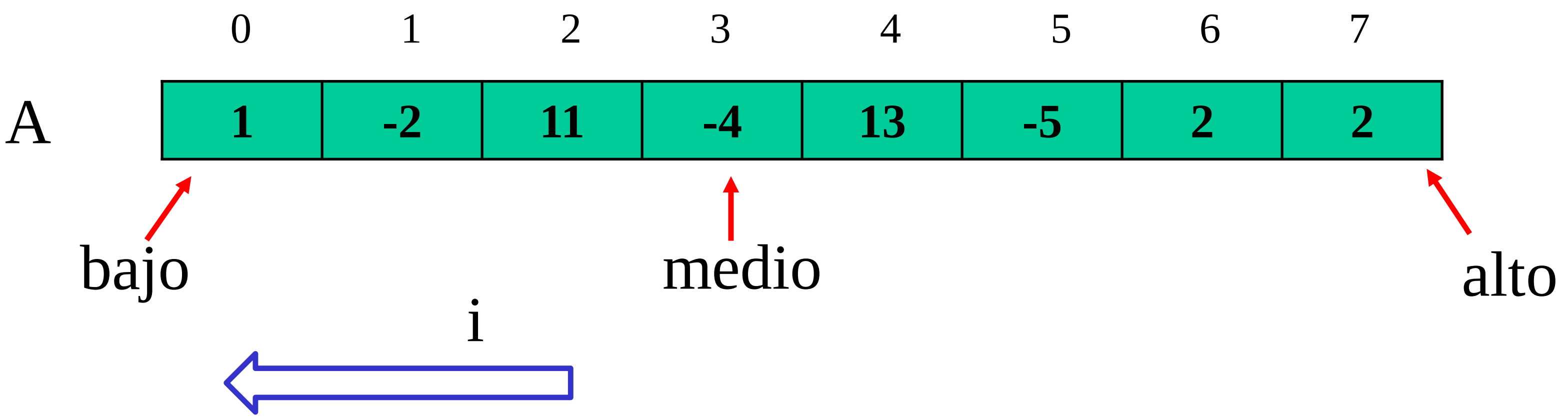
⇩

SumaIzq = - 4

IndiceIzq = medio

```
sumaIzq = -∞
suma = 0
for i = medio hasta bajo
    suma += A[ i ]
    if (suma > sumaIzq )
        sumaIzq = suma
        indiceIzq = i
```

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

bajo

medio

alto

i

Suma = 0

Suma + A[medio] = -4

Suma + A[2] = -4 +11 = 7

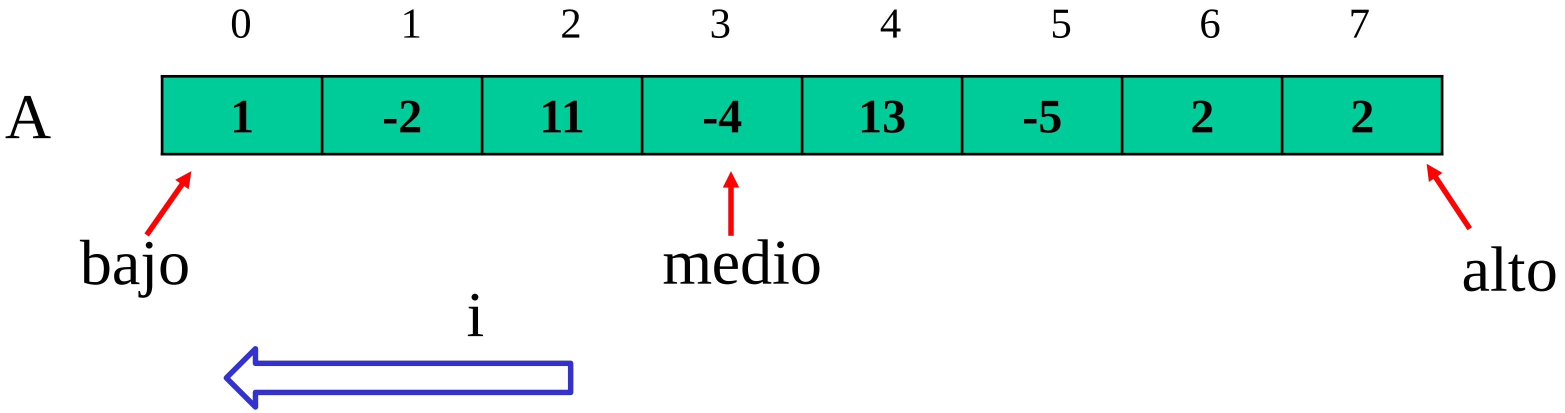
Suma > sumaIzq

SumaIzq = - 4

IndiceIzq = medio

```
sumaIzq = -∞
suma = 0
for i = medio hasta bajo
    suma += A[ i ]
    if (suma > sumaIzq )
        sumaIzq = suma
        indiceIzq = i
```

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

Suma = 0

Suma + A[medio] = -4

Suma + A[2] = -4 +11 = 7

Suma > sumaIzq

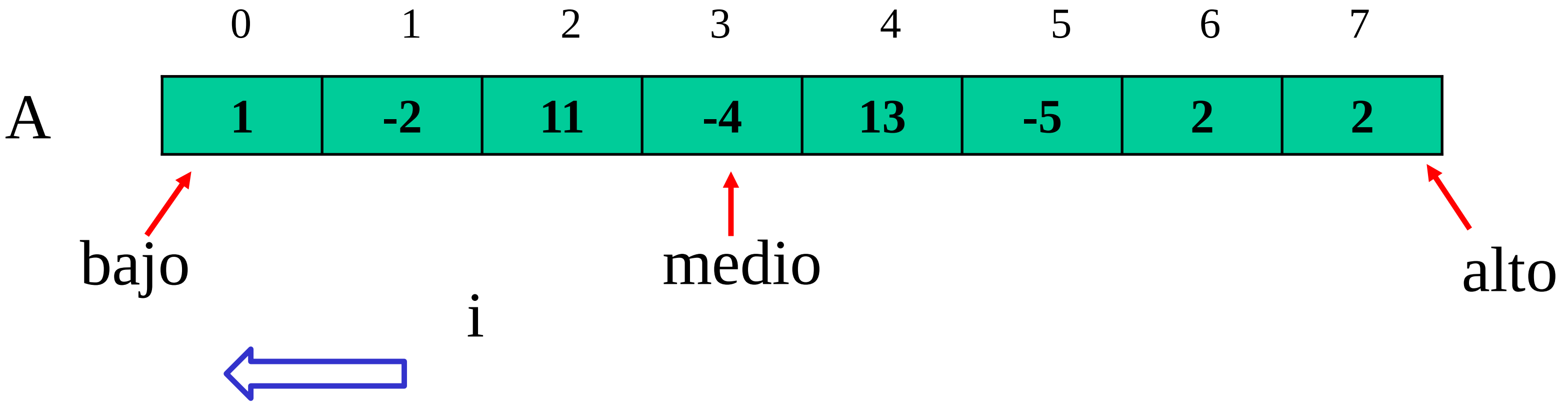
$\Downarrow$

SumaIzq = 7

IndiceIzq = 2

```
sumaIzq = -∞
suma = 0
for i = medio hasta bajo
    suma += A[ i ]
    if (suma > sumaIzq )
        sumaIzq = suma
        indiceIzq = i
```

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

Suma = 0

Suma + A[medio] = -4

Suma + A[2] = -4 +11 = 7

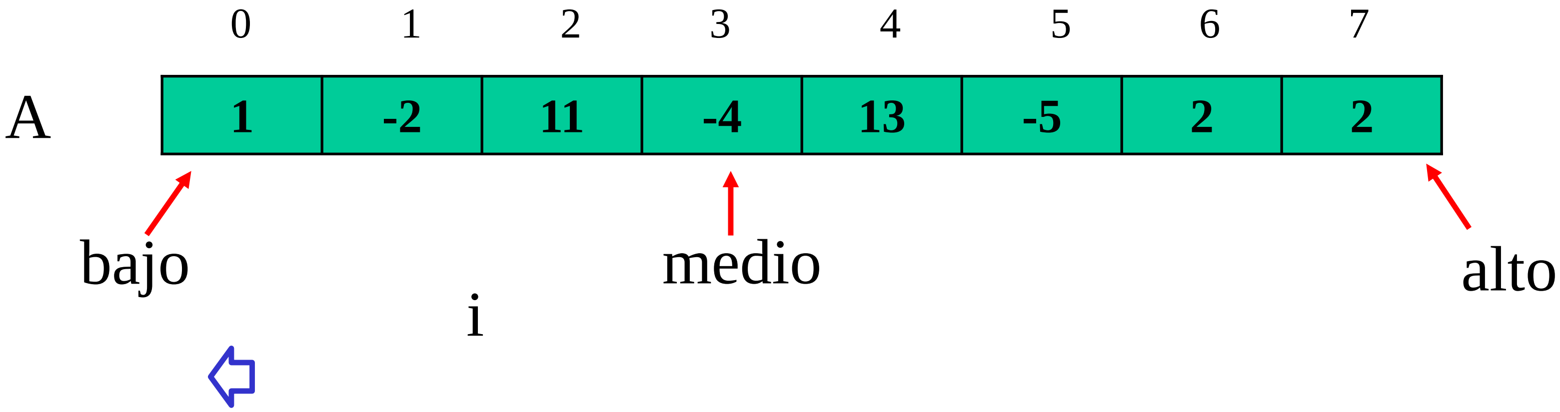
Suma + A[1] = -4 +11 - 2 = 5

SumaIzq = 7

IndiceIzq = 2

```
sumaIzq = -∞
suma = 0
for i = medio hasta bajo
    suma += A[ i ]
    if (suma > sumaIzq )
        sumaIzq = suma
        indiceIzq = i
```

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

Suma = 0

Suma + A[medio] = -4

Suma + A[2] = -4 +11 = 7

Suma + A[1] = -4 +11 - 2 = 5

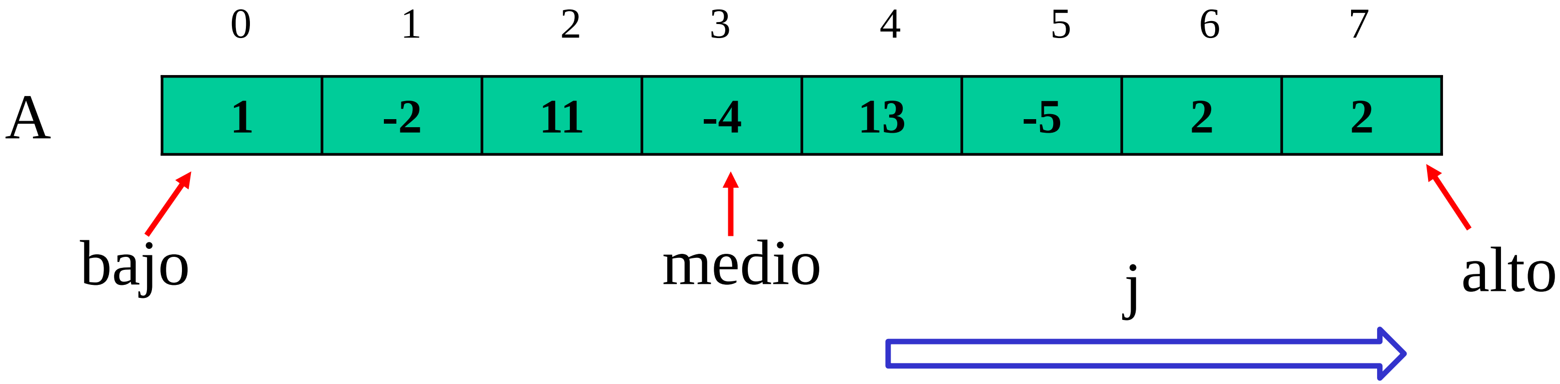
Suma + A[0] = -4 +11 - 2 + 1= 6

SumaIzq = 7

IndiceIzq = 2

```
sumaIzq = -∞
suma = 0
for i = medio hasta bajo
     suma += A[ i ]
     if (suma > sumaIzq )
          sumaIzq = suma
          indiceIzq = i
```

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

bajo

medio

j

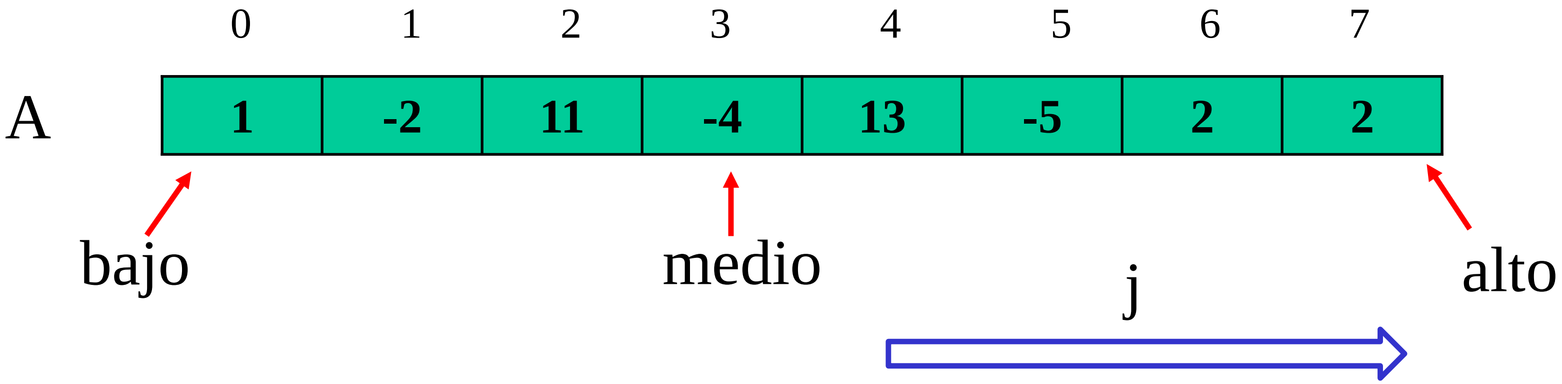
alto

Suma = 0

```
sumaDer = -∞
suma = 0
for j = medio + 1 hasta alto
    suma += A[ j ]
    if (suma > sumaDer )
        sumaDer = suma
        indiceDer = j
```

SumaDer = - ∞

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

bajo

medio

j

alto

Suma = 0

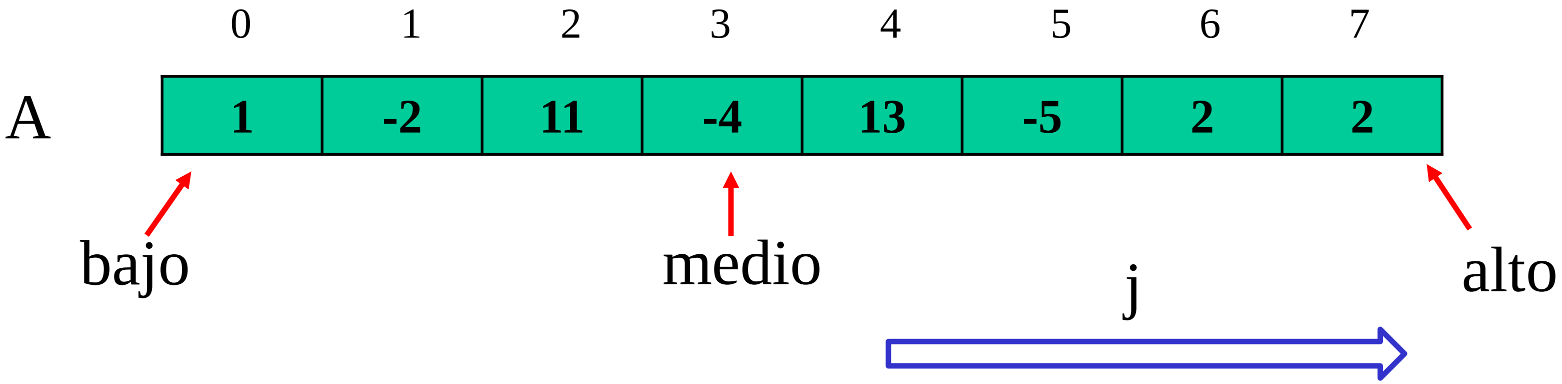
Suma + A[medio+1] = 13

```
sumaDer = -∞
suma = 0
for j = medio + 1 hasta alto
    suma += A[ j ]
    if (suma > sumaDer )
        sumaDer = suma
        indiceDer = j
```

Suma > sumaDer

SumaDer = - ∞

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

bajo

medio

j

alto

Suma = 0

Suma + A[medio+1] = 13

```
sumaDer = -∞
suma = 0
for j = medio + 1 hasta alto
     suma += A[ j ]
     if (suma > sumaDer )
          sumaDer = suma
          indiceDer = j
```

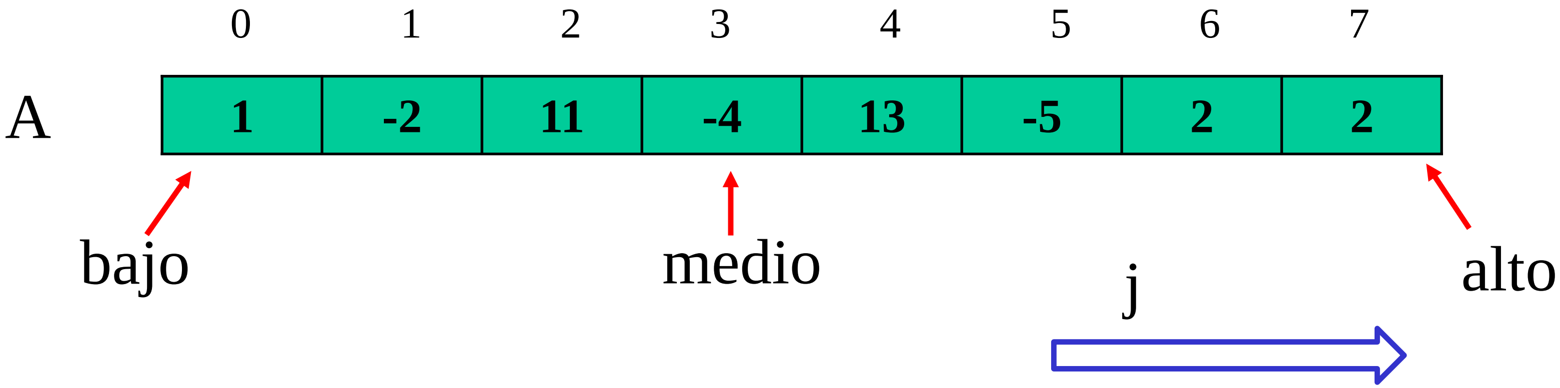
Suma > sumaDer

⇩

SumaDer = 13

IndiceDer = 4

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

Suma = 0

Suma + A[medio+1] = 13

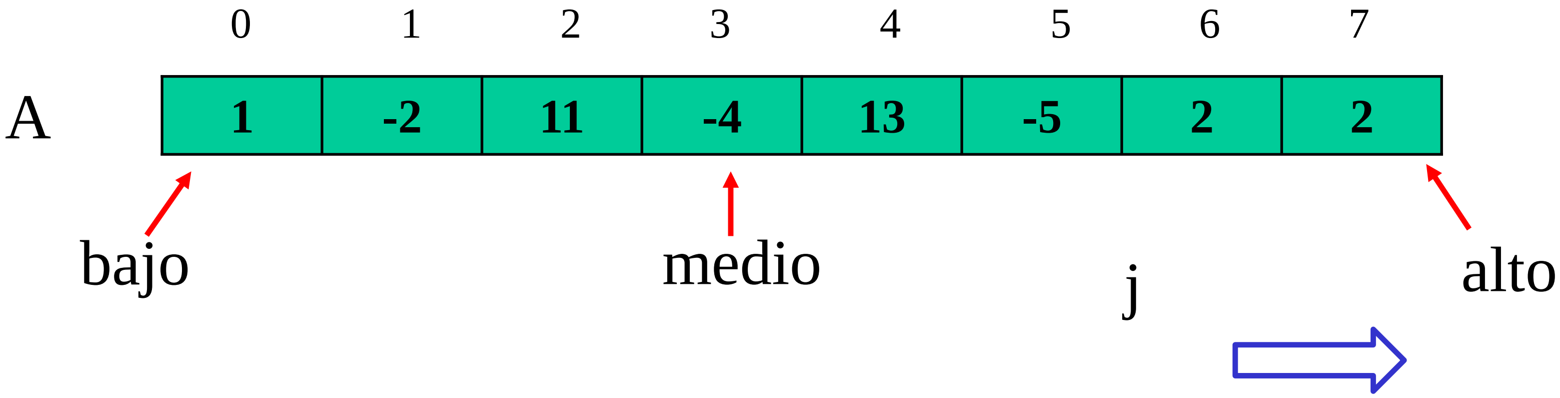
Suma + A[5] = 13 – 5 = 8

```
sumaDer = -∞
suma = 0
for j = medio + 1 hasta alto
    suma += A[ j ]
    if (suma > sumaDer )
        sumaDer = suma
        indiceDer = j
```

SumaDer = 13

IndiceDer = 4

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

bajo

medio

j

alto

Suma = 0

Suma + A[medio+1] = 13

Suma + A[5] = 13 – 5 = 8

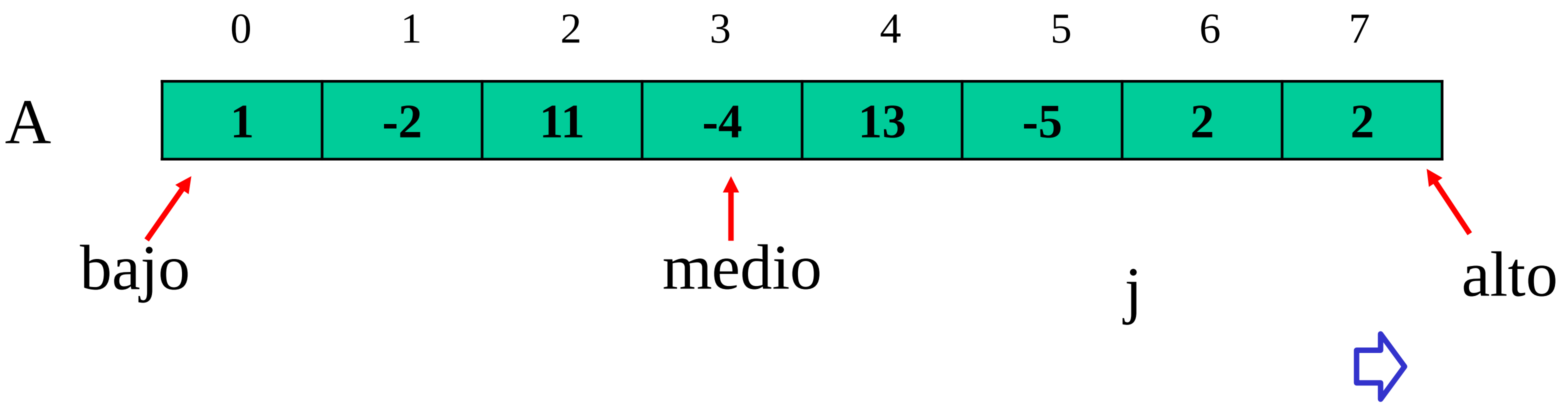
Suma + A[6] = 13 – 5 + 2= 10

```
sumaDer = -∞
suma = 0
for j = medio + 1 hasta alto
    suma += A[ j ]
    if (suma > sumaDer )
        sumaDer = suma
        indiceDer = j
```

SumaDer = 13

IndiceDer = 4

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

Suma = 0

Suma + A[medio+1] = 13

Suma + A[5] = 13 – 5 = 8

Suma + A[6] = 13 – 5 + 2= 10

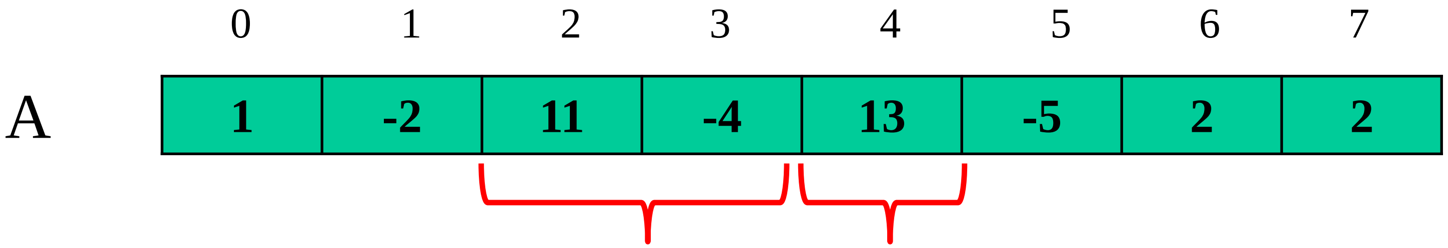
Suma + A[7] = 13 – 5 + 2 + 2 = 12

```
sumaDer = -∞
suma = 0
for j = medio + 1 hasta alto
    suma += A[ j ]
    if (suma > sumaDer )
        sumaDer = suma
        indiceDer = j
```

SumaDer = 13

IndiceDer = 4

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

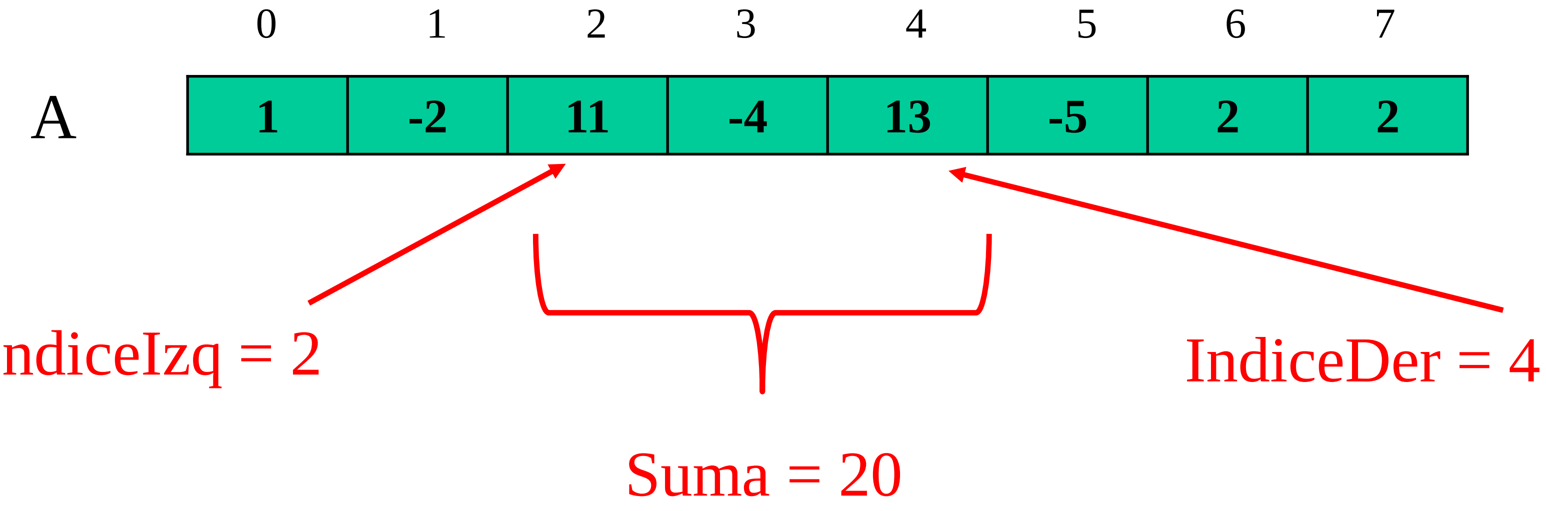
SumaIzq = 7

IndiceIzq = 2

SumaDer = 13

IndiceDer = 4

Subsecuencia de suma máxima



	0	1	2	3	4	5	6	7
A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**

IndiceIzq = 2

IndiceDer = 4

Suma = 20

SolucionMedio

return < IndiceIzq, IndiceDer, SumaIzq + SumaDer >

return < 2, 4, 7 + 13 >

Ir a final

Subsecuencia de suma máxima

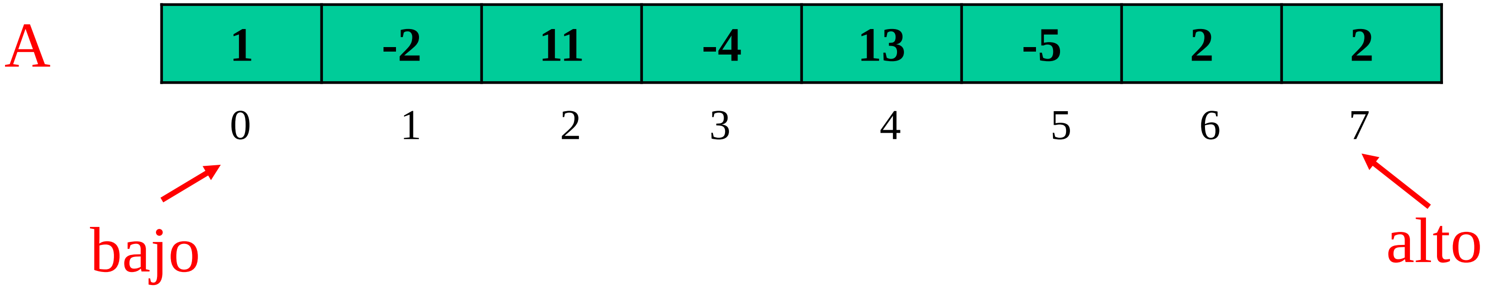
Teniendo el procedimiento para obtener la SSM que cruza el punto medio, podemos resolver el problema de la SSM por divide y conquista:

```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto == bajo)                               // caso base
        return  < bajo, alto, A[bajo] >;
    else                                            // cálculo de subproblemas
        { medio = (alto + bajo) / 2;

          // Subproblema: Parte izquierda
         <bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);

         // Subproblema: Parte derecha
         <bajoDer, altoDer,sumaDer> = Encontrar_SSM (A, medio+1, alto);
```

Subsecuencia de suma máxima

```
Encontrar_SSM ( A, bajo, alto ) {
   …
   …
    // Combinar soluciones
    <bajoMedio, altoMedio, sumaMedio> =
                         SolucionMedio( A, bajo, medio, alto);

    if ((sumaIzq > sumaDer ) and ( sumaIzq > sumaMedio))
         return < bajoIzq, altoIzq, sumaIzq>;

    else if ((sumaDer >= sumaIzq) and (sumaDer > = sumaMedio))
        return < bajoDer, altoDer, sumaDer>;

    else return < bajoMedio, altoMedio, sumaMedio>
```

Subsecuencia de suma máxima



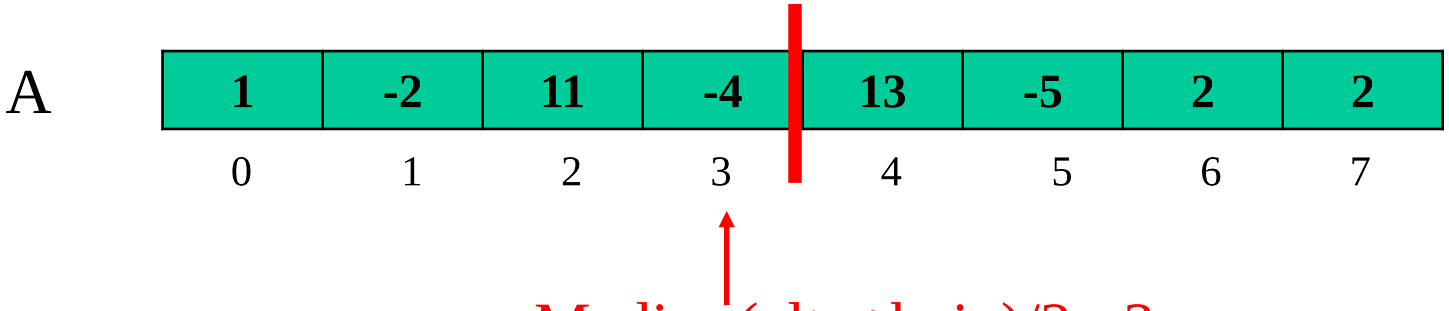
A	1	-2	11	-4	13	-5	2	2
	0	1	2	3	4	5	6	7

bajo

alto

```
Encontrar_SSM ( A, bajo, alto ) {
     if (alto = = bajo)                          // caso base
          return  < bajo, alto, A[bajo] >;
     else                                        // cálculo de subproblemas
          { medio = (alto + bajo) / 2;

 // Subproblema: Parte izquierda
<bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);

// Subproblema: Parte derecha
<bajoDer, altoDer,sumaDer> = Encontrar_SSM (A, medio+1, alto);
```

Subsecuencia de suma máxima

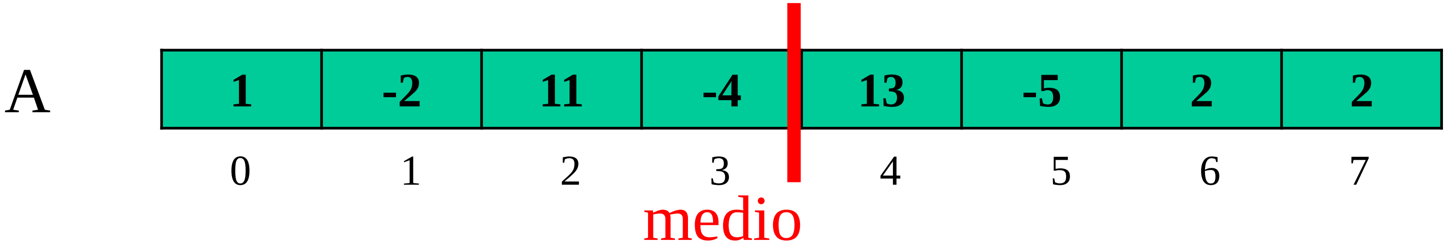


A	1	-2	11	-4	13	-5	2	2
	0	1	2	3	4	5	6	7

Medio=(alto+bajo)/2= 3

```
Encontrar_SSM ( A, bajo, alto ) {
     if (alto = = bajo)                          // caso base
          return  < bajo, alto, A[bajo] >;
     else                                        // cálculo de subproblemas
          { medio = (alto + bajo) / 2;

    // Subproblema: Parte izquierda
   <bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);
…
```

Subsecuencia de suma máxima



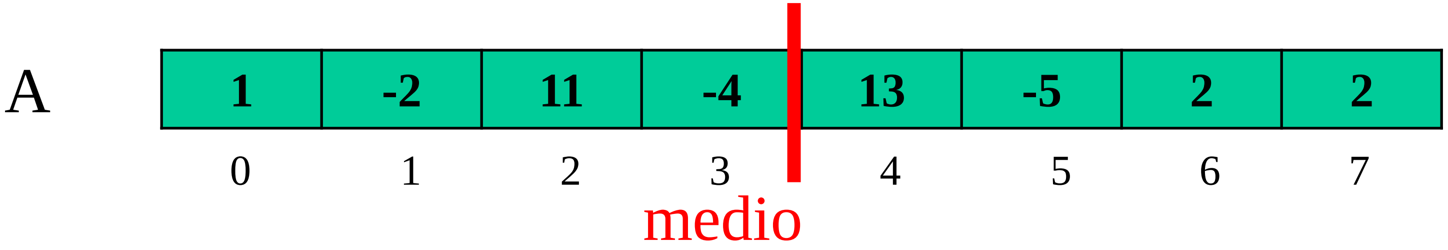
A	1	-2	11	-4	13	-5	2	2
	0	1	2	3	4	5	6	7

medio

1	-2	11	-4

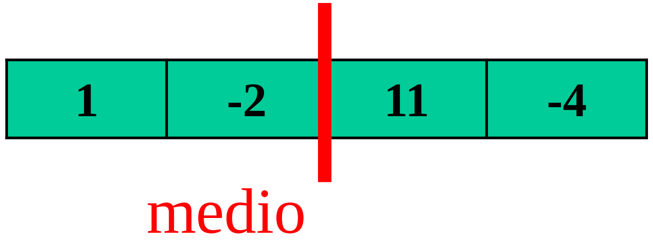
```
Encontrar_SSM ( A, bajo, alto ) {
     if (alto = = bajo)                        // caso base
          return  < bajo, alto, A[bajo] >;
     else                                      // cálculo de subproblemas
          { medio = (alto + bajo) / 2;

 // Subproblema: Parte izquierda
<bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);
…
```

Subsecuencia de suma máxima

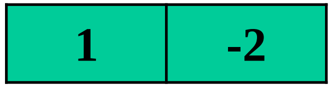


A

1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
0	1	2	3	4	5	6	7



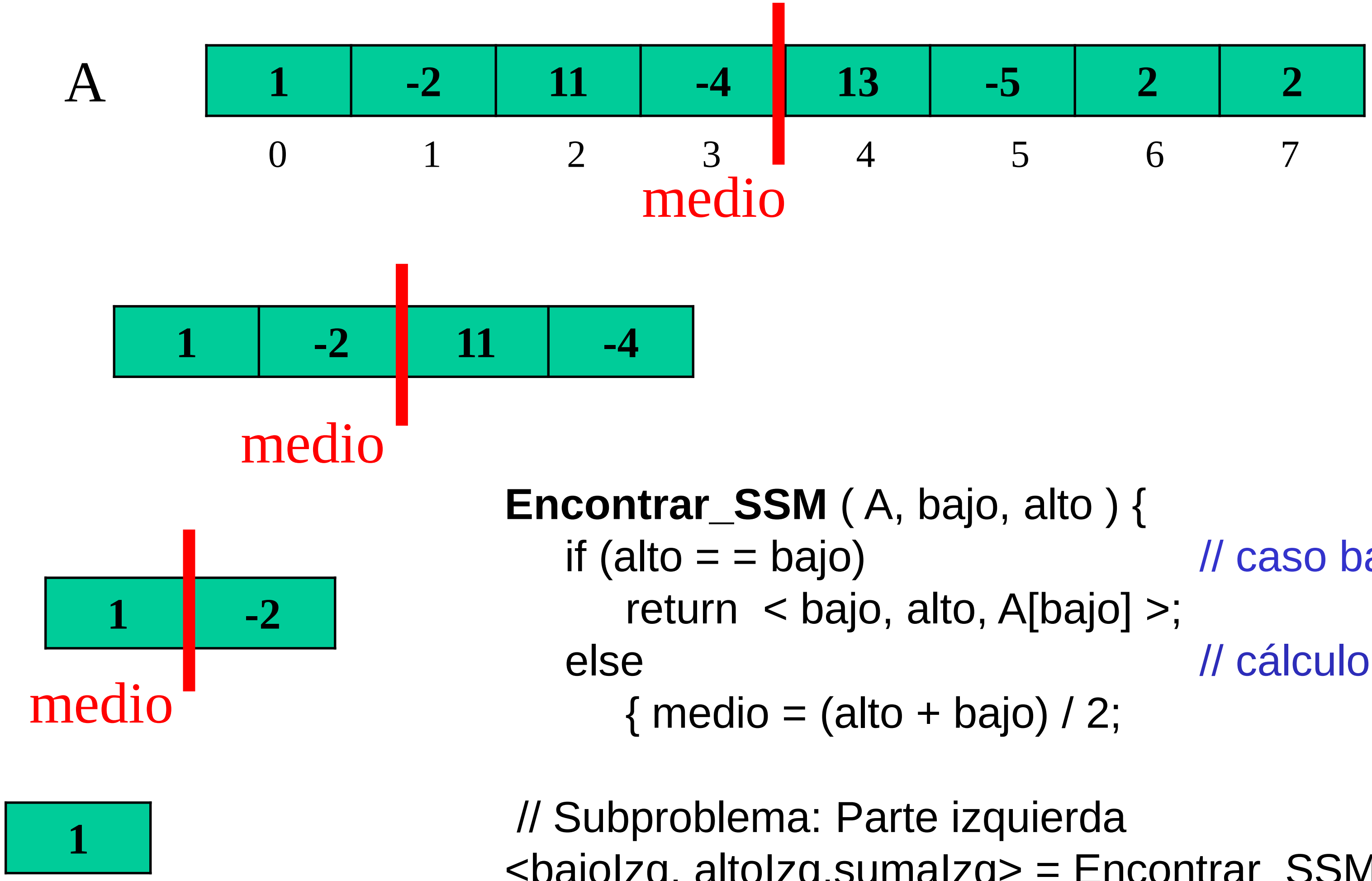
1	**-2**	**11**	**-4**



1	**-2**

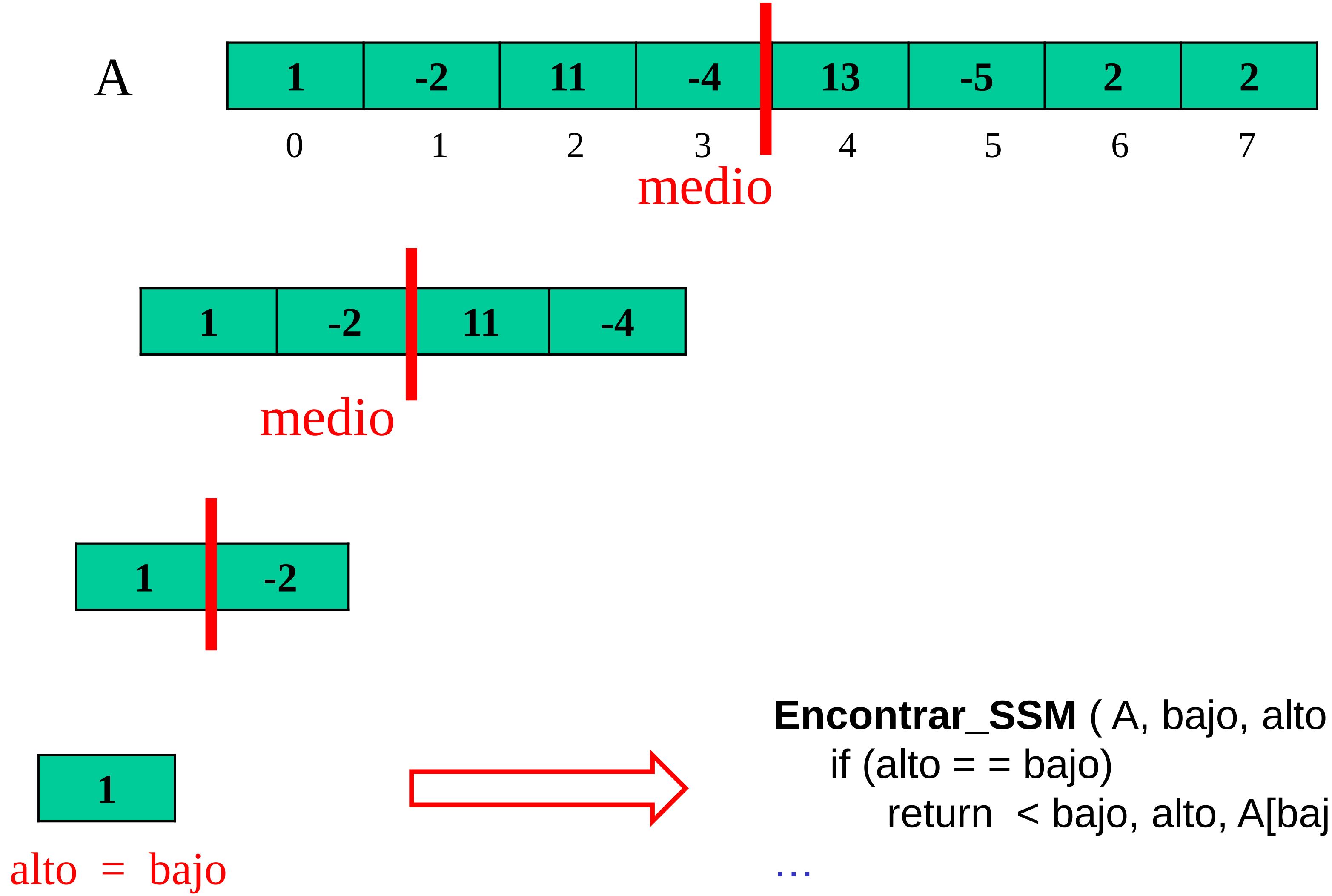
```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)                        // caso base
        return  < bajo, alto, A[bajo] >;
    else                                      // cálculo de subproblemas
        { medio = (alto + bajo) / 2;

 // Subproblema: Parte izquierda
<bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);
…
```

Subsecuencia de suma máxima



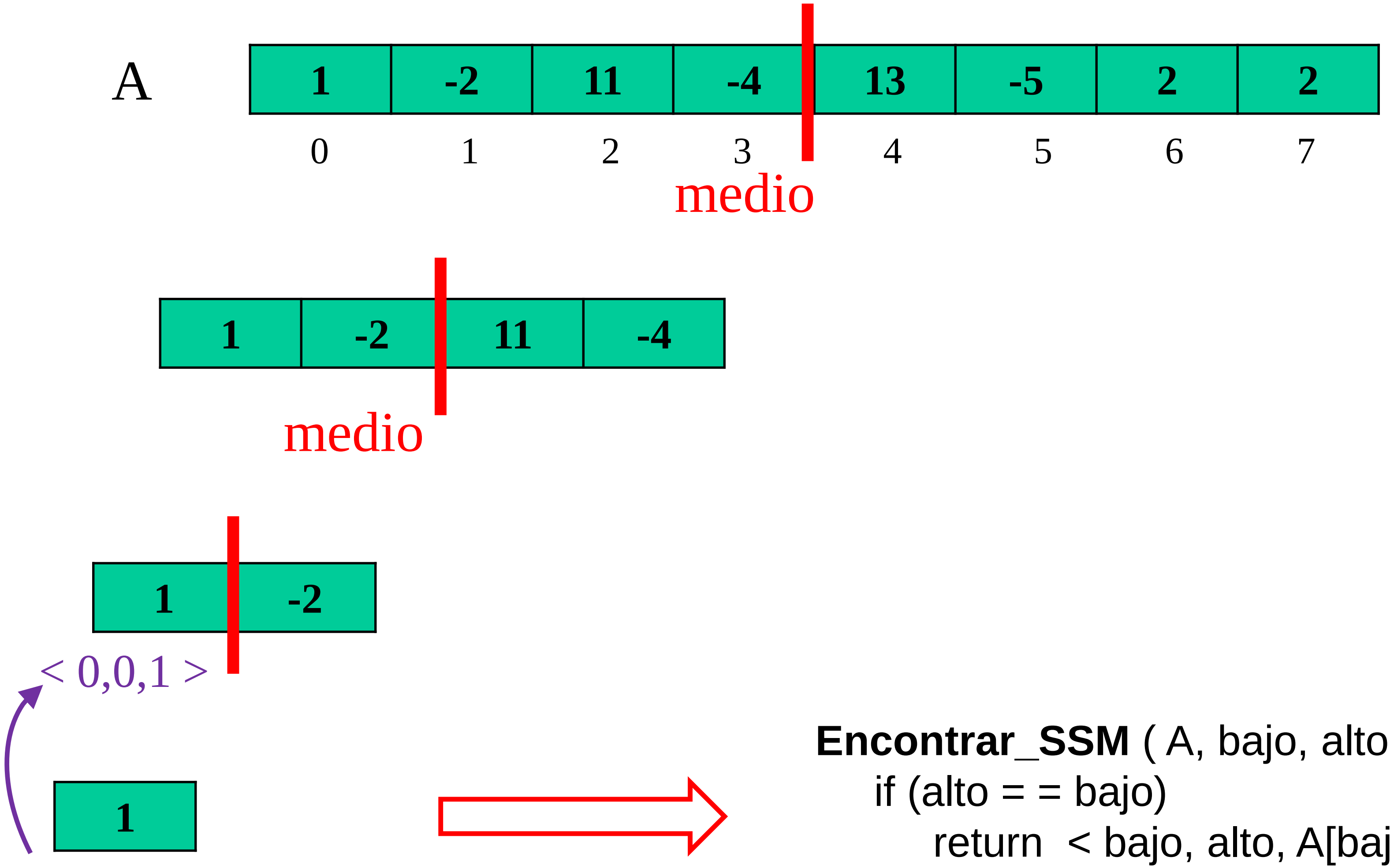
```
Encontrar_SSM ( A, bajo, alto ) {
     if (alto = = bajo)                         // caso base
          return  < bajo, alto, A[bajo] >;
     else                                       // cálculo de subproblemas
          { medio = (alto + bajo) / 2;

 // Subproblema: Parte izquierda
<bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);
…
```

Subsecuencia de suma máxima



```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)                    // caso base
        return  < bajo, alto, A[bajo] >;
…
```

Subsecuencia de suma máxima



A

1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
0	1	2	3	4	5	6	7

medio

1	**-2**	**11**	**-4**

medio

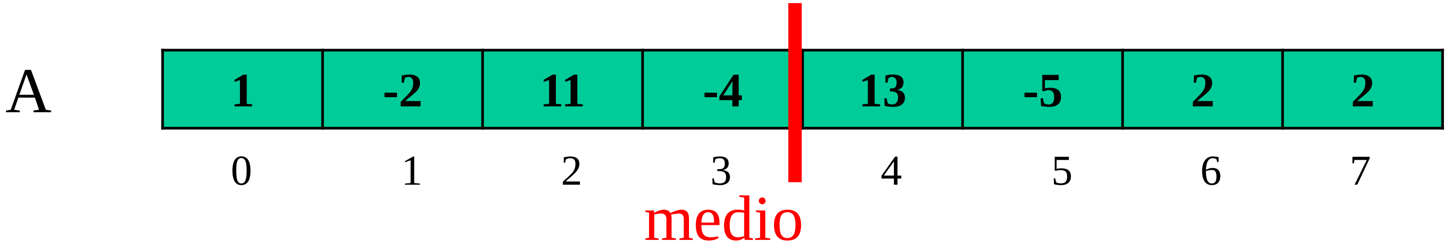
1	**-2**

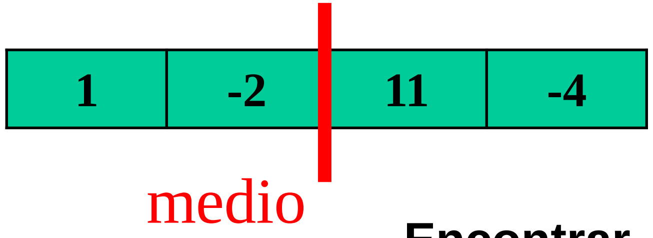
< 0,0,1 >

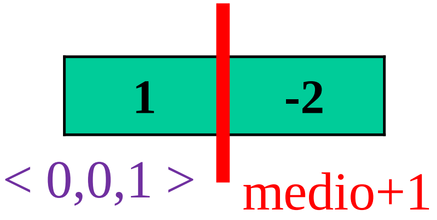
1

```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)                    // caso base
        return  < bajo, alto, A[bajo] >;
…
```

Subsecuencia de suma máxima

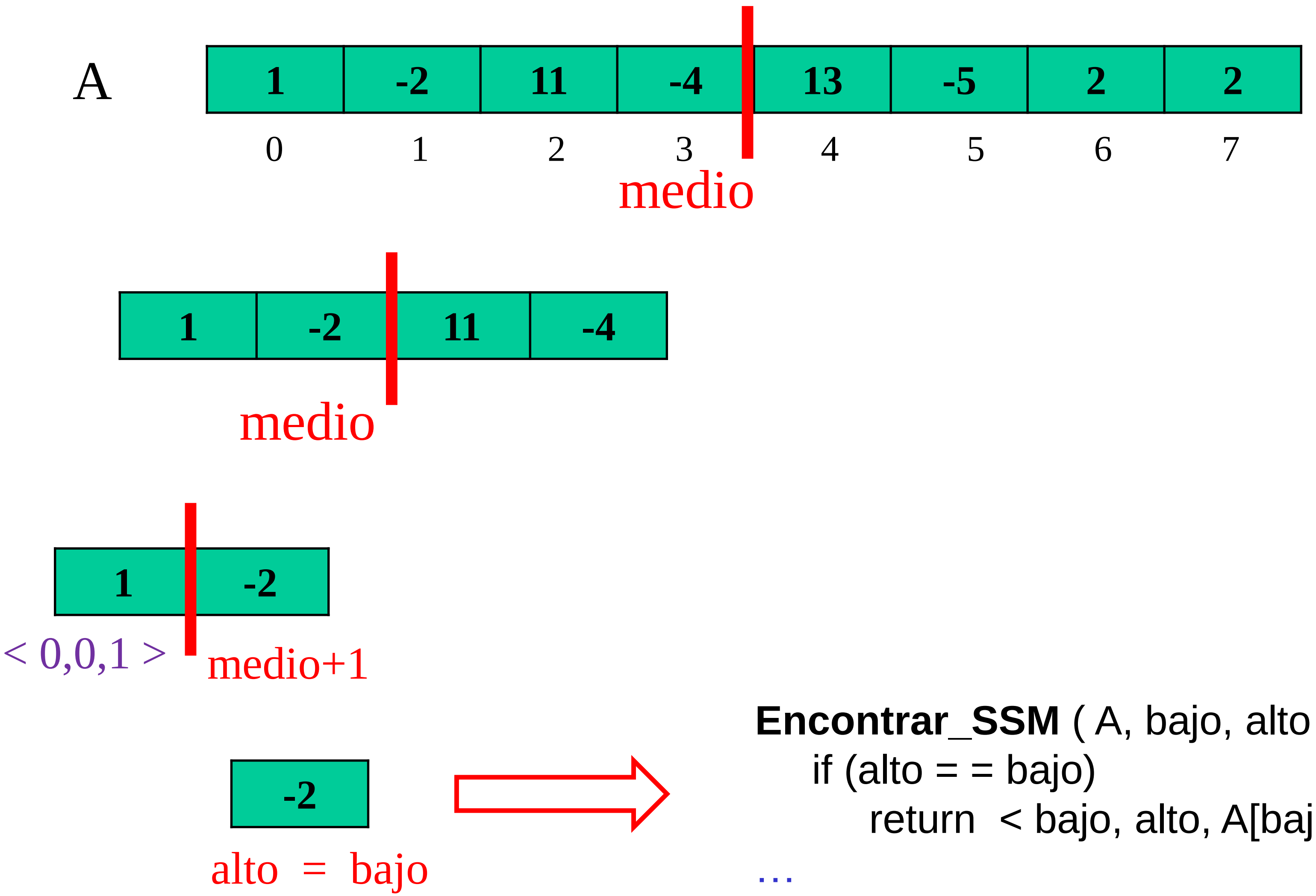






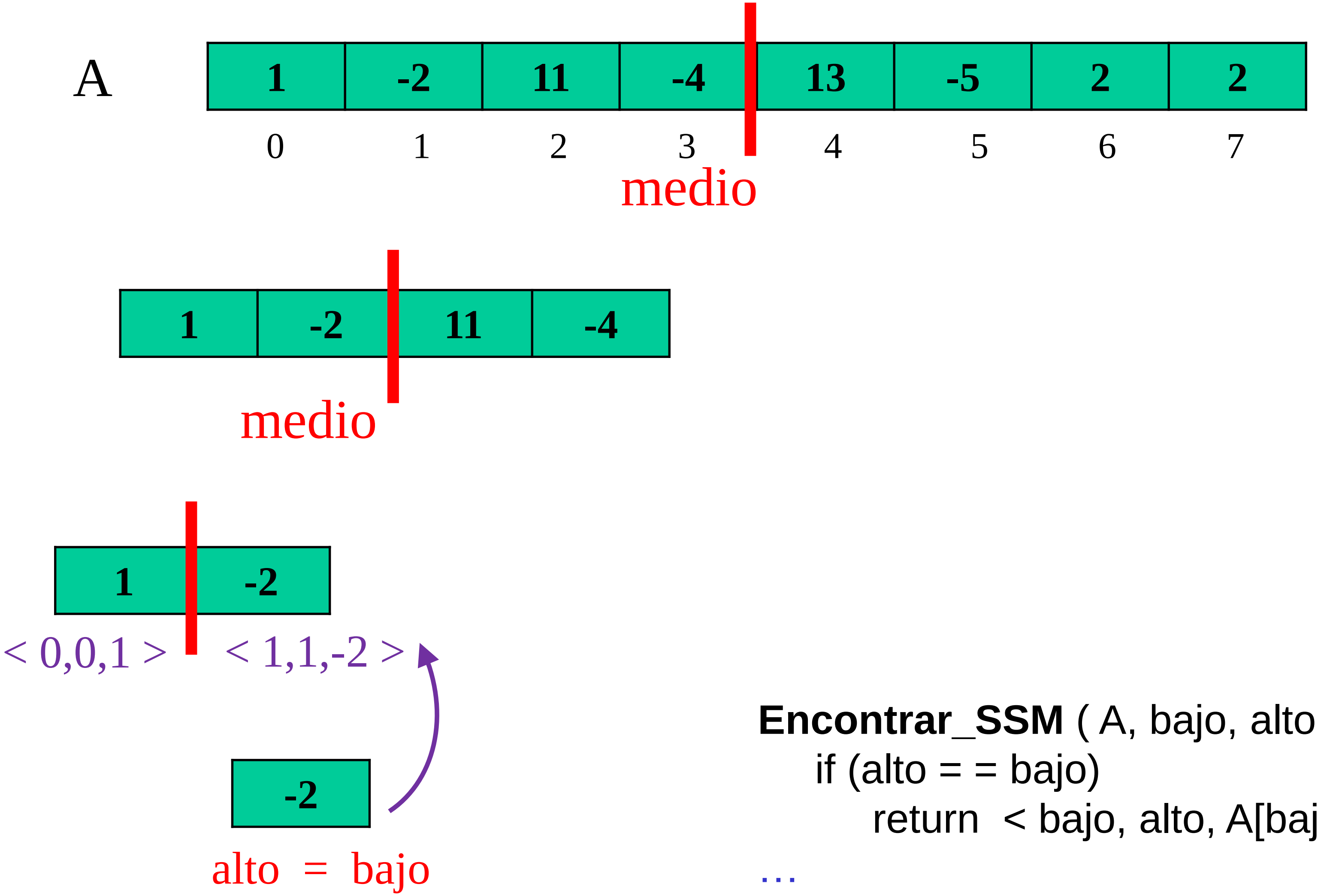
```
Encontrar_SSM ( A, bajo, alto ) {
    if   …
    else                                          // cálculo de subproblemas
        { medio = (alto + bajo) / 2;
 // Subproblema: Parte izquierda
<bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);

// Subproblema: Parte derecha
<bajoDer, altoDer,sumaDer> = Encontrar_SSM (A, medio+1, alto);
```

Subsecuencia de suma máxima



```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)                    // caso base
        return  < bajo, alto, A[bajo] >;
…
```

Subsecuencia de suma máxima



A

1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

medio

1	-2	11	-4

medio

1	-2

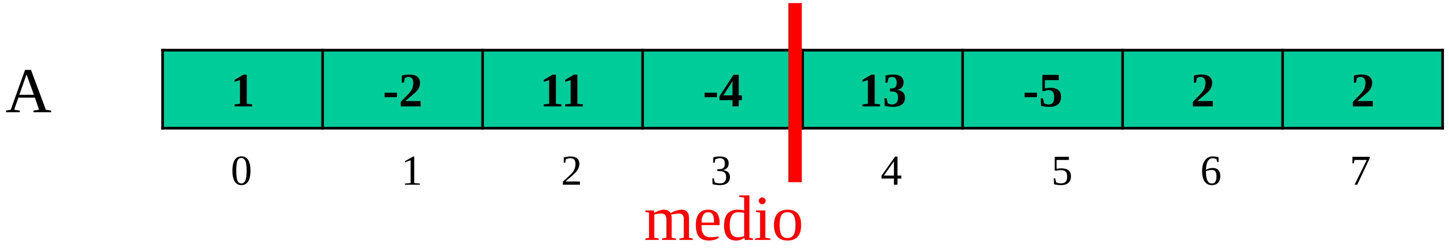
< 0,0,1 > < 1,1,-2 >

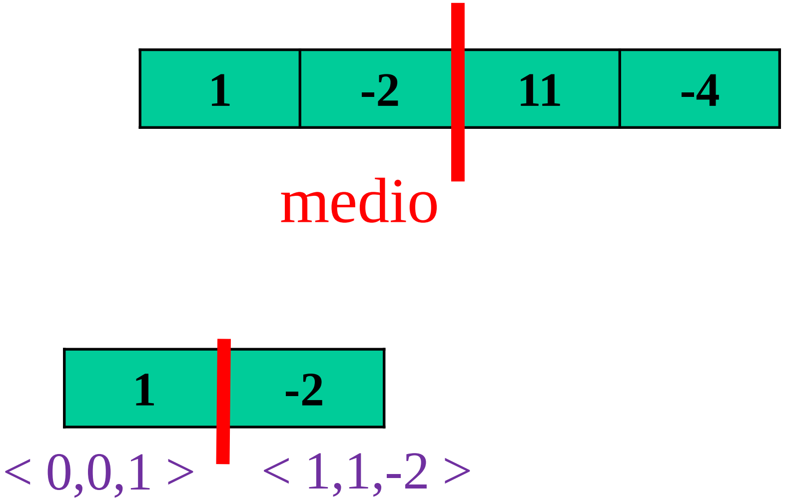
-2

alto = bajo

```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)                    // caso base
        return  < bajo, alto, A[bajo] >;
…
```

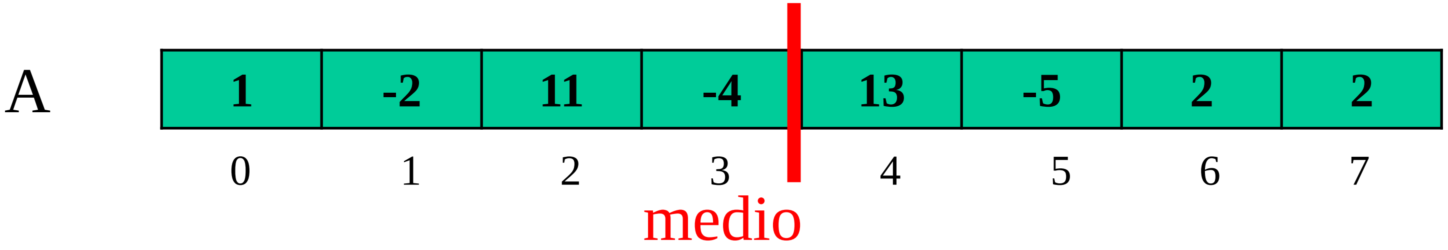
Subsecuencia de suma máxima





```
Encontrar_SSM ( A, bajo, alto ) {
    …
// Subproblema: Parte izquierda
…
// Subproblema: Parte derecha
…
// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
            SolucionMedio( A, bajo, medio, alto);
….
```

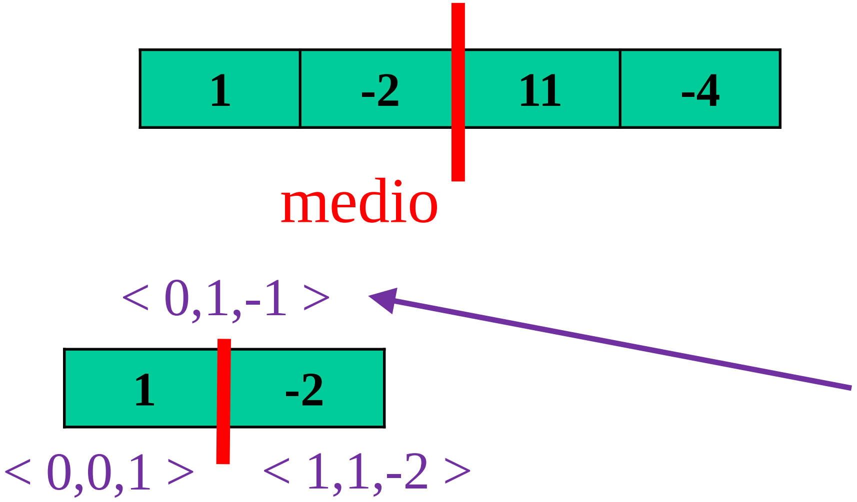
Subsecuencia de suma máxima



A

1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

medio



1	-2	11	-4

medio

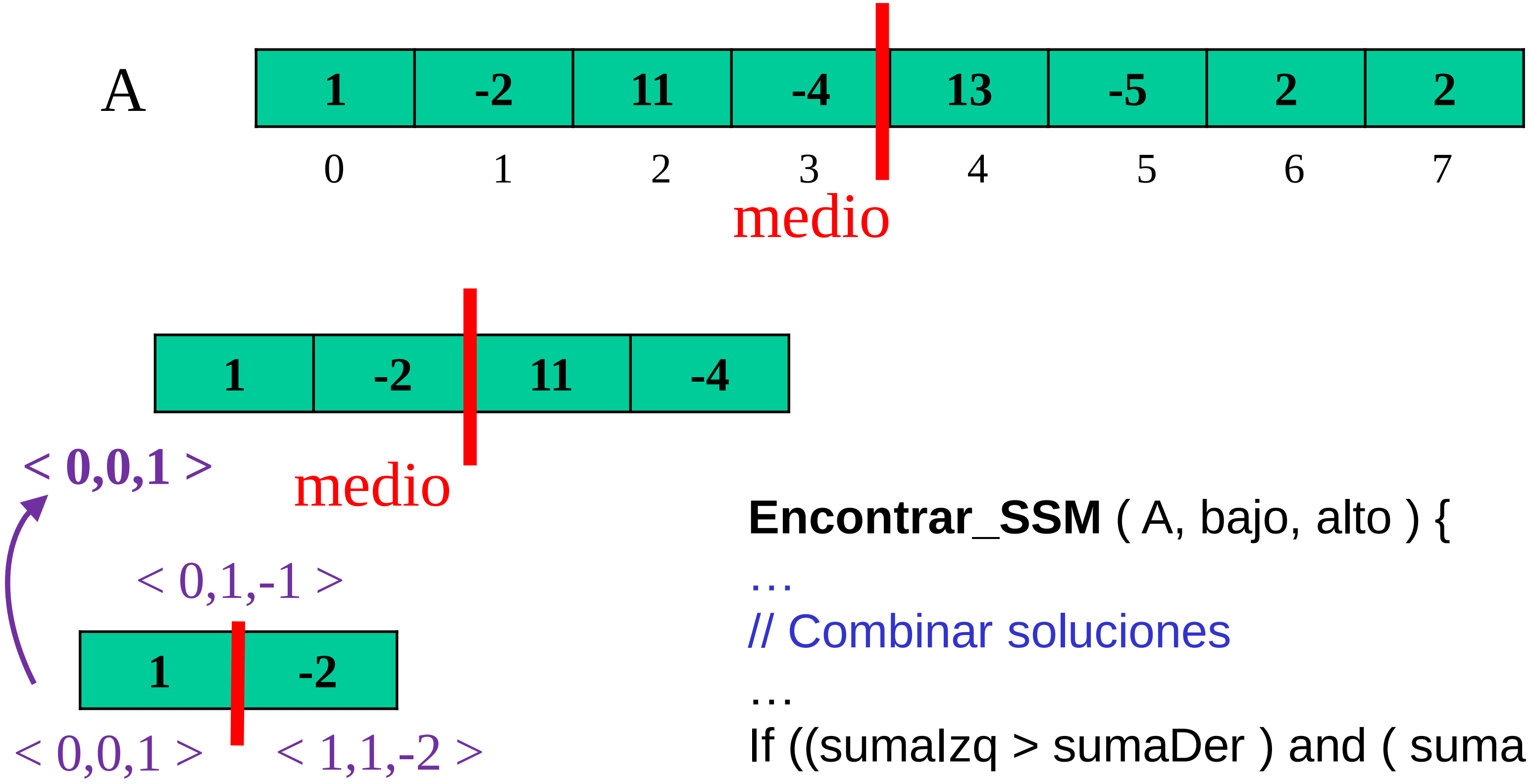
< 0,1,-1 >

1	-2

< 0,0,1 >   < 1,1,-2 >

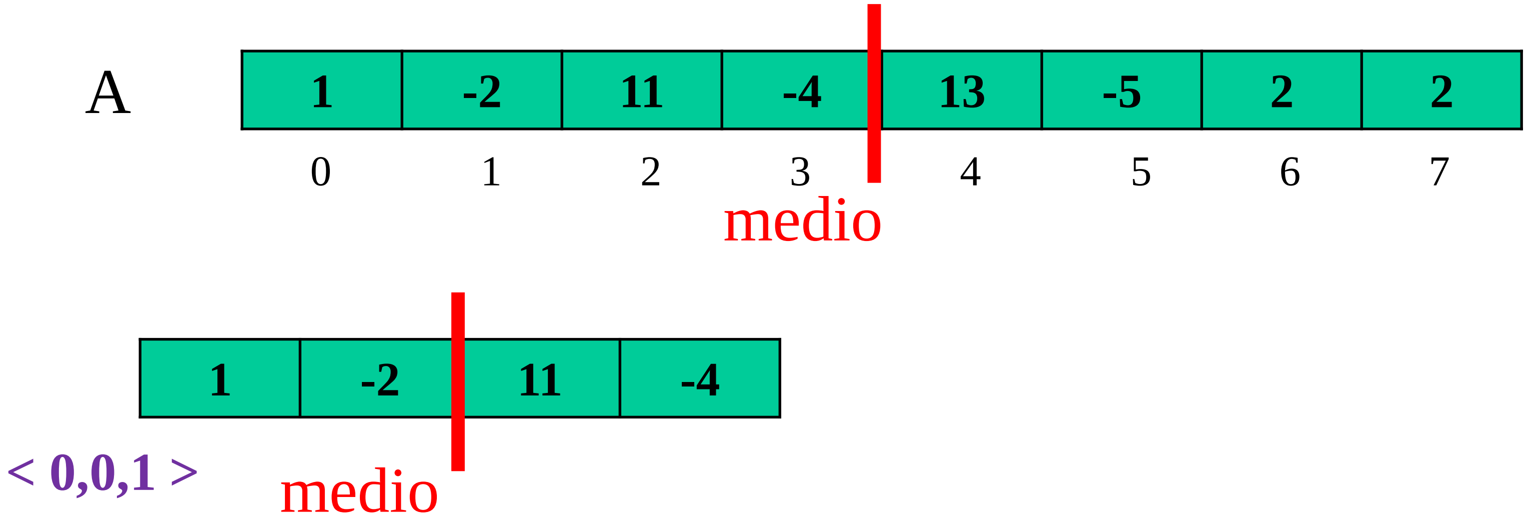
```
Encontrar_SSM ( A, bajo, alto ) {
…
// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
            SolucionMedio( A, bajo, medio, alto);

….
```

Subsecuencia de suma máxima



```
Encontrar_SSM ( A, bajo, alto ) {
…
// Combinar soluciones
…
If ((sumaIzq > sumaDer ) and ( sumaIzq > sumaMedio))
      return < bajoIzq, altoIzq, sumaIzq>;
else If ((sumaDer >= sumaIzq) and (sumaDer > = sumaMedio))
     return < bajoDer, altoDer, sumaDer>;
else return < bajoMedio, altoMedio, sumaMedio>
```

Subsecuencia de suma máxima



A

1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
0	1	2	3	4	5	6	7

medio

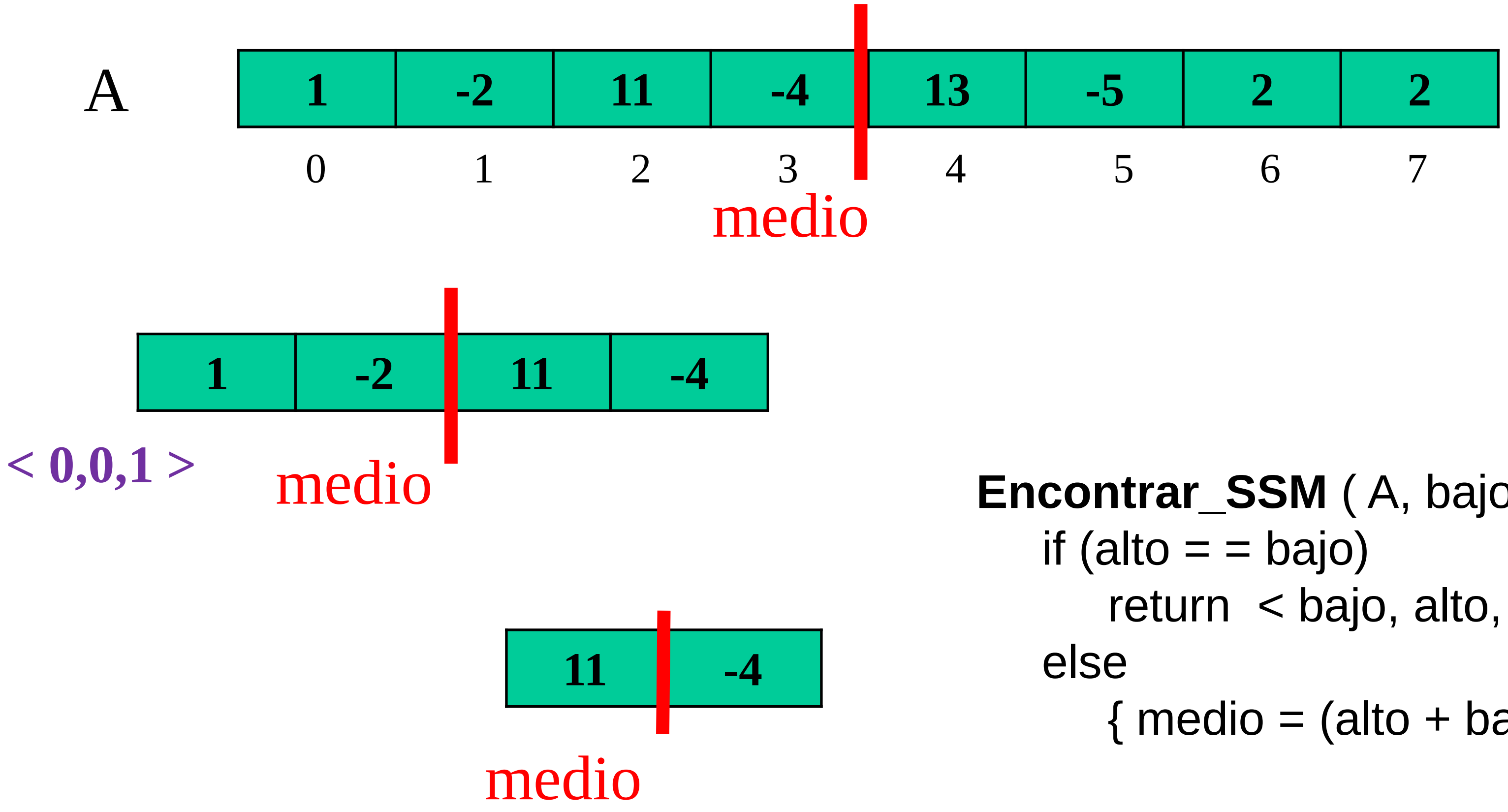
1	**-2**	**11**	**-4**

< 0,0,1 >

medio

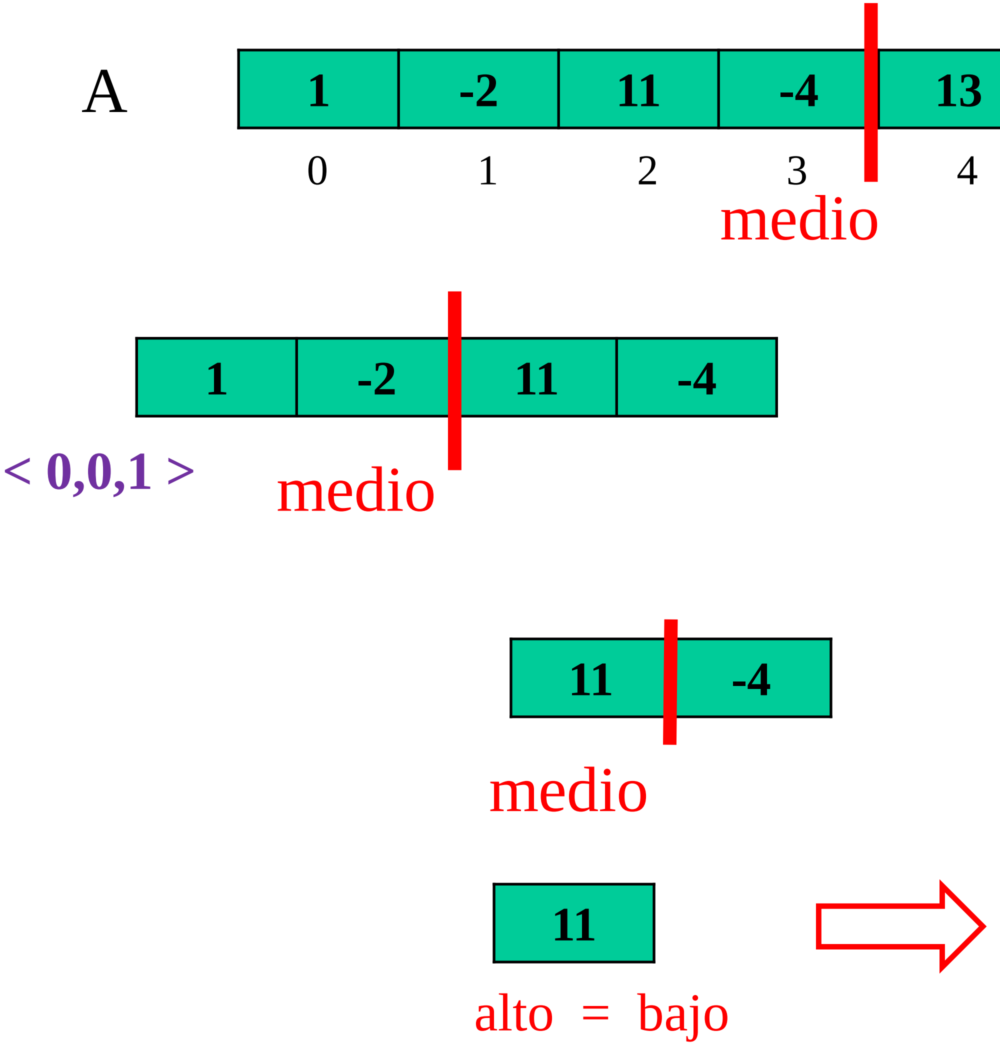
```
Encontrar_SSM ( A, bajo, alto ) {
     if   …
     else                                                // cálculo de subproblemas
          { medio = (alto + bajo) / 2;
 // Subproblema: Parte izquierda
<bajoIzq, altoIzq,sumaIzq> = Encontrar_SSM (A, bajo, medio);

// Subproblema: Parte derecha
<bajoDer, altoDer,sumaDer> = Encontrar_SSM (A, medio+1, alto);
```

Subsecuencia de suma máxima



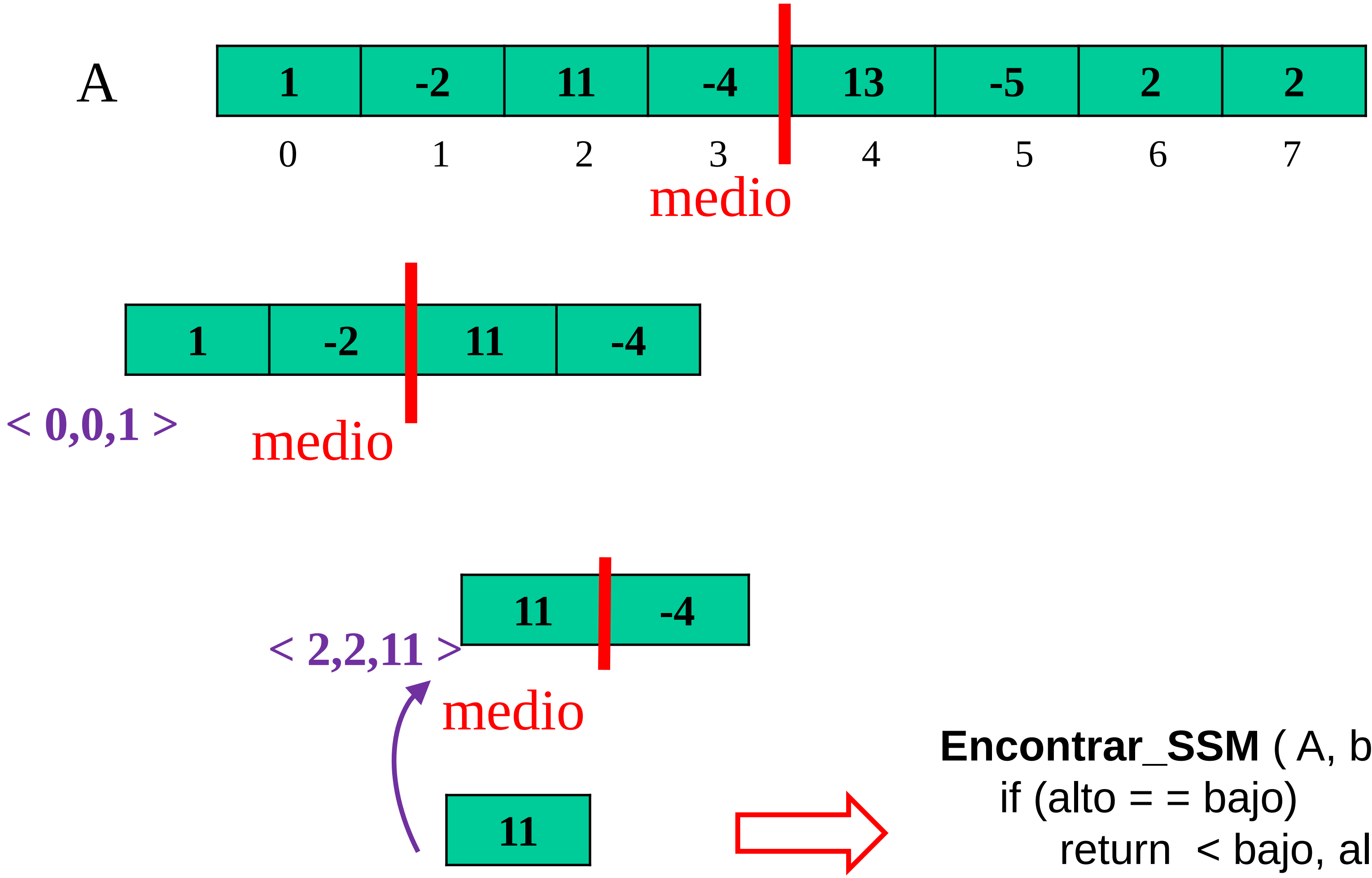
```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)
        return  < bajo, alto, A[bajo] >;
    else
        { medio = (alto + bajo) / 2;

 // Subproblema: Parte izquierda
<bajoIzq, altoIzq,sumaIzq> =
              Encontrar_SSM (A, bajo, medio);
…
```

Subsecuencia de suma máxima



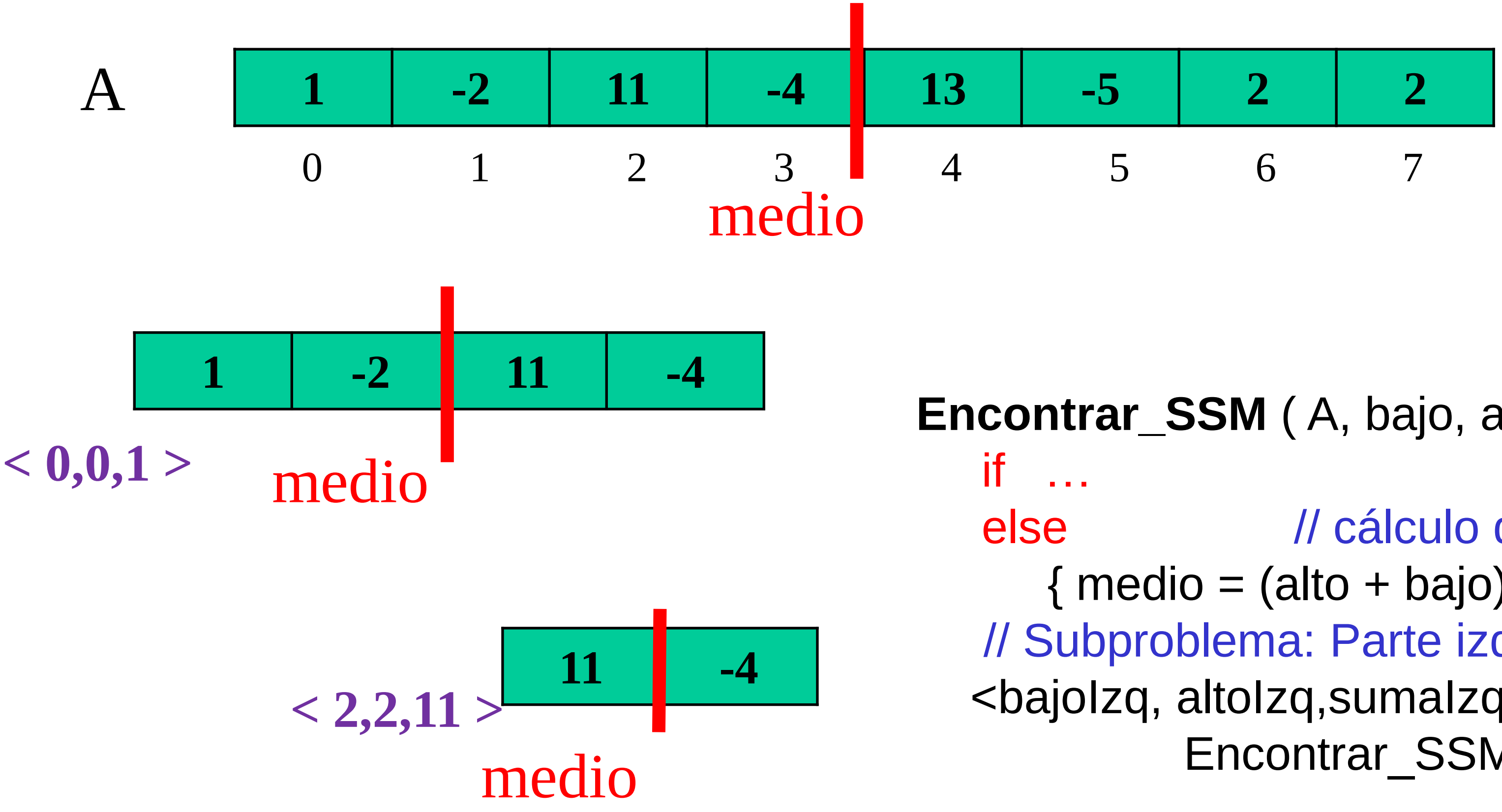
```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)              // caso base
        return  < bajo, alto, A[bajo] >;
…
```

Subsecuencia de suma máxima



```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)              // caso base
        return  < bajo, alto, A[bajo] >;
…
```

Subsecuencia de suma máxima



A

1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

medio

1	-2	11	-4

< 0,0,1 >

medio

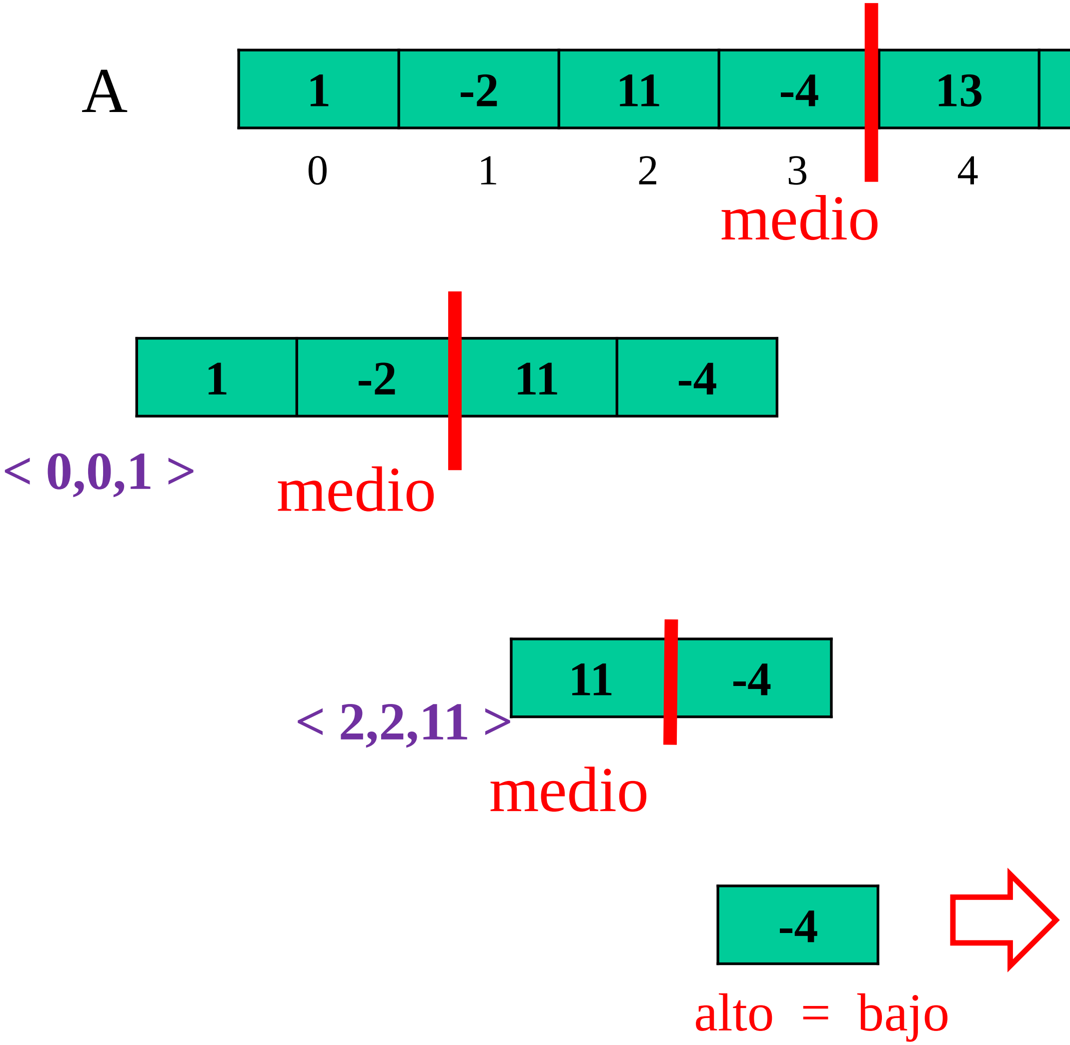
11	-4

< 2,2,11 >

medio

```
Encontrar_SSM ( A, bajo, alto ) {
    if   …
    else                  // cálculo de subproblemas
        { medio = (alto + bajo) / 2;
    // Subproblema: Parte izquierda
    <bajoIzq, altoIzq,sumaIzq> =
                 Encontrar_SSM (A, bajo, medio);

    // Subproblema: Parte derecha
    <bajoDer, altoDer,sumaDer> =
                Encontrar_SSM (A, medio+1, alto);
```

Subsecuencia de suma máxima



1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

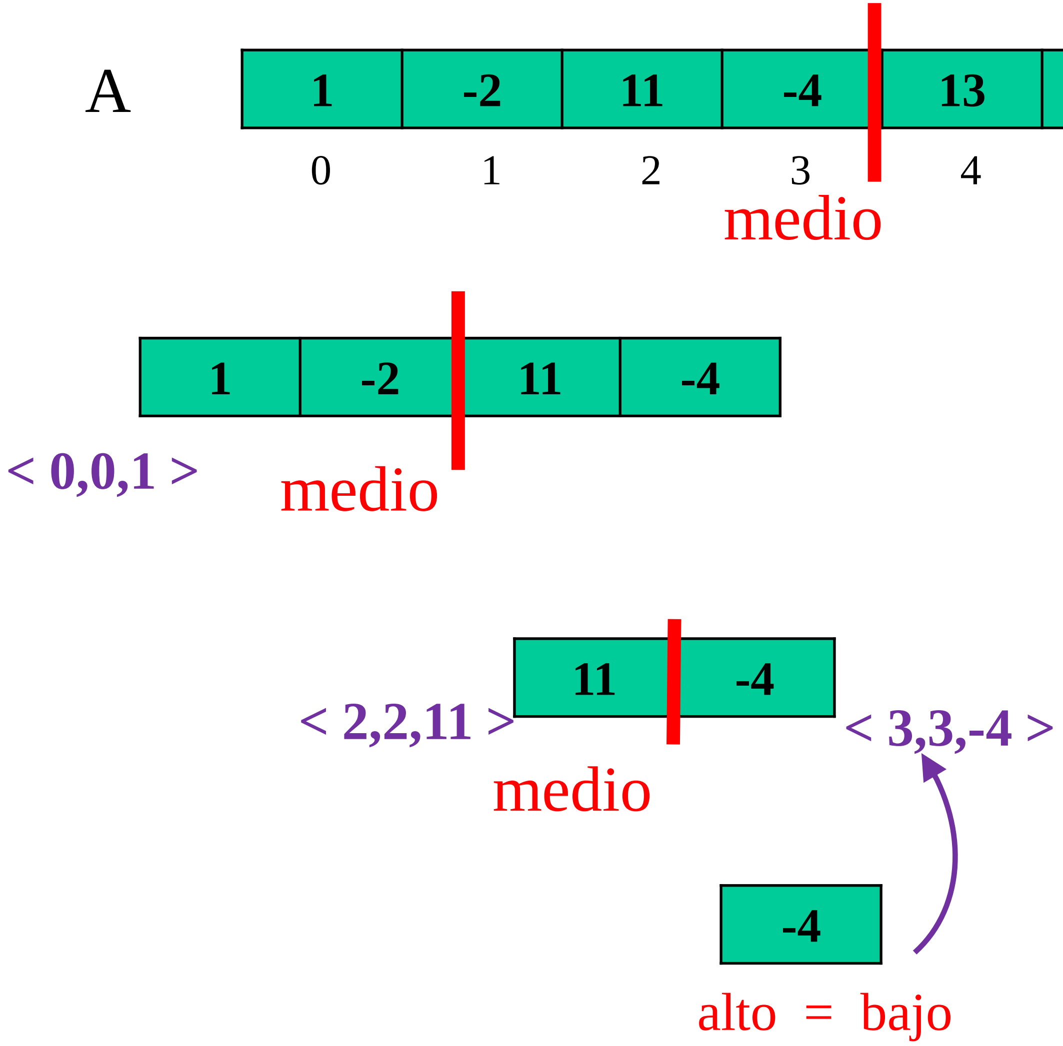
1	-2	11	-4

11	-4

-4

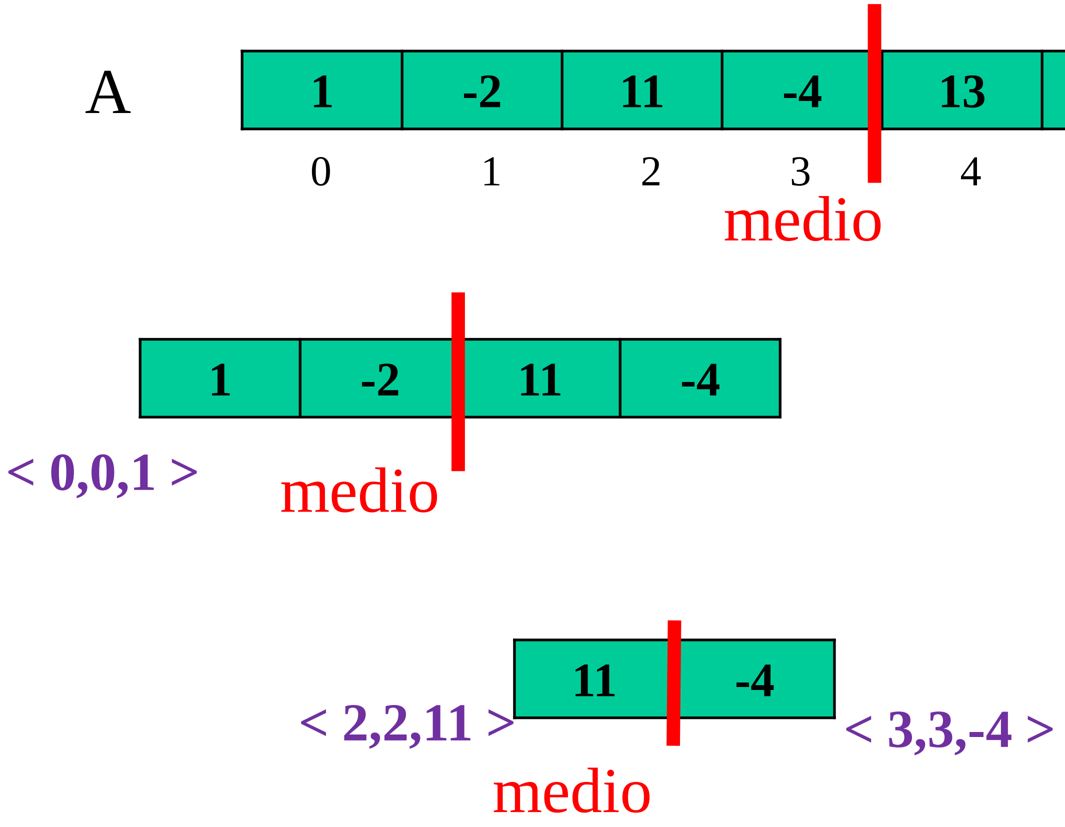
```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)     // caso base
        return  < bajo, alto, A[bajo] >;
…
```

Subsecuencia de suma máxima



```
Encontrar_SSM ( A, bajo, alto ) {
    if (alto = = bajo)     // caso base
        return  < bajo, alto, A[bajo] >;
…
```

Subsecuencia de suma máxima



A

1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

medio

1	-2	11	-4

< 0,0,1 >

medio

11	-4

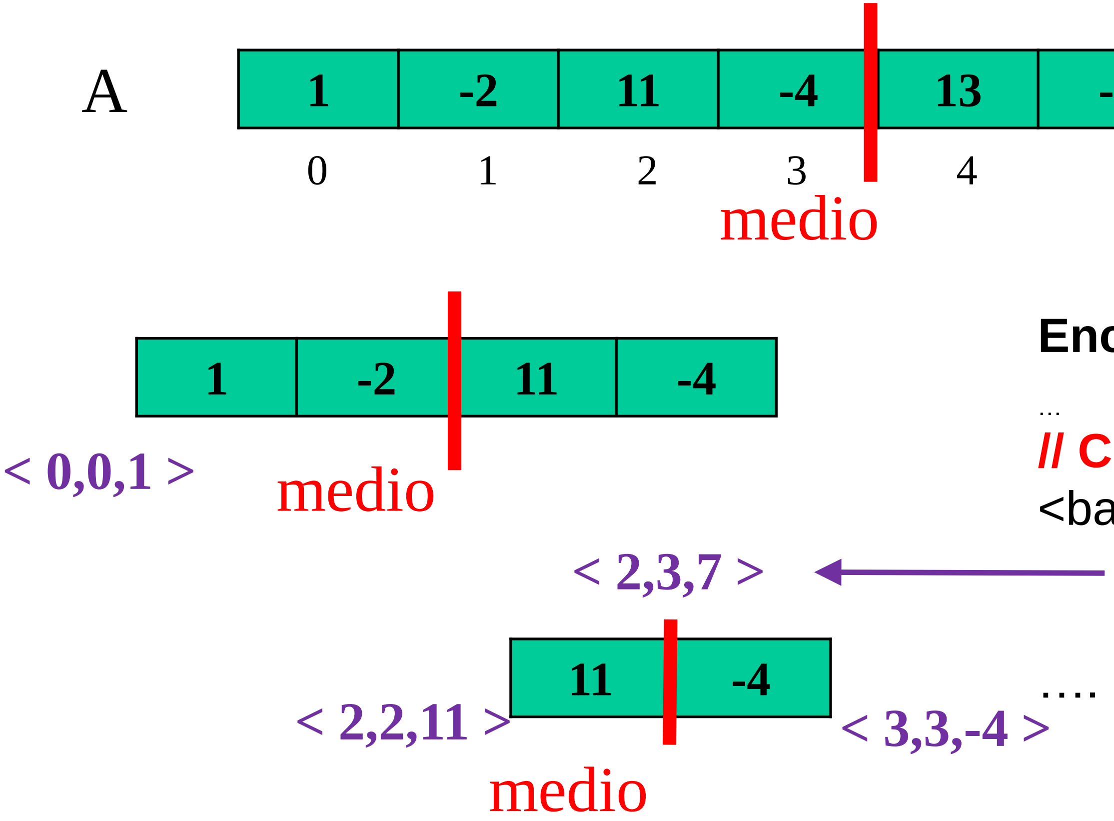
< 2,2,11 >

< 3,3,-4 >

medio

```
Encontrar_SSM ( A, bajo, alto ) {
    …
// Subproblema: Parte izquierda


// Subproblema: Parte derecha
…
// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
          SolucionMedio( A, bajo, medio, alto);

….
```

Subsecuencia de suma máxima



A

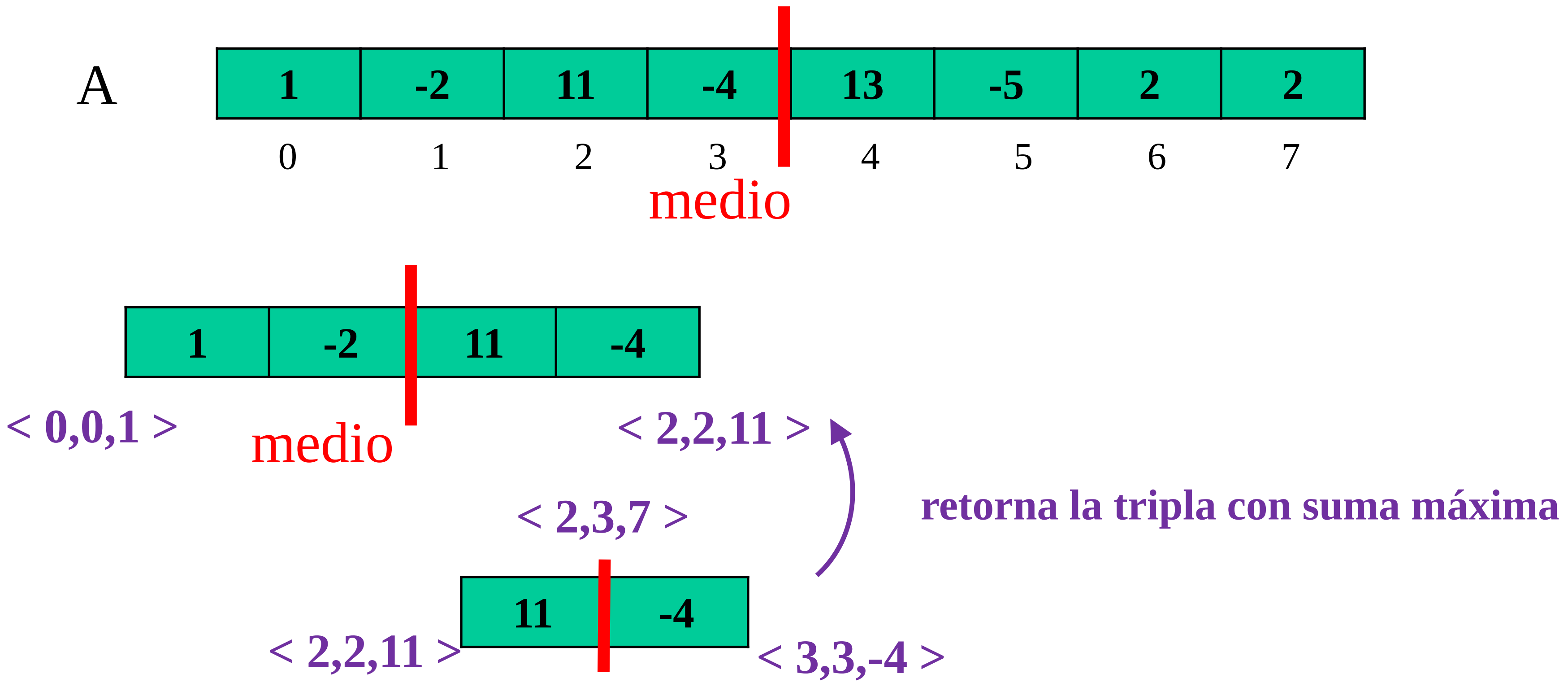
1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

1	-2	11	-4

11	-4

```
Encontrar_SSM ( A, bajo, alto ) {
…
// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
        SolucionMedio( A, bajo, medio, alto);
```

Subsecuencia de suma máxima



A

1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

medio

1	-2	11	-4

< 0,0,1 >

medio

< 2,2,11 >

< 2,3,7 >

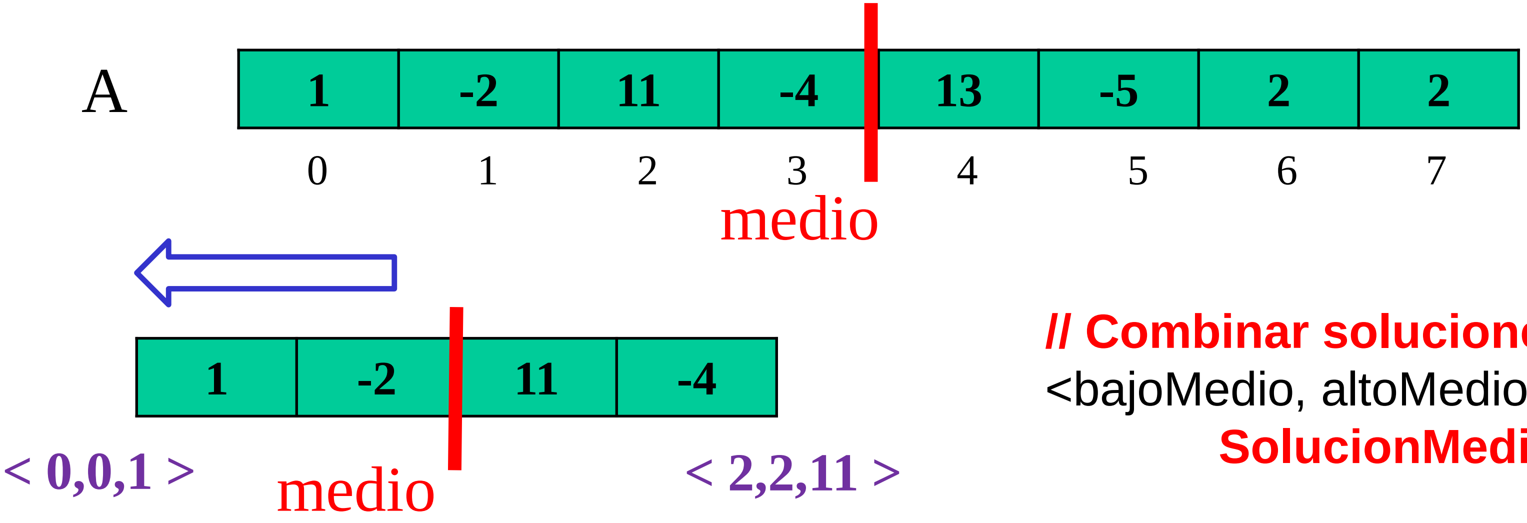
retorna la tripla con suma máxima

11	-4

< 2,2,11 >

< 3,3,-4 >

Subsecuencia de suma máxima



A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
	0	1	2	3	4	5	6	7

medio

1	**-2**	**11**	**-4**

< 0,0,1 > medio **< 2,2,11 >**

// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
SolucionMedio(A, bajo, medio, alto);

Suma = 0

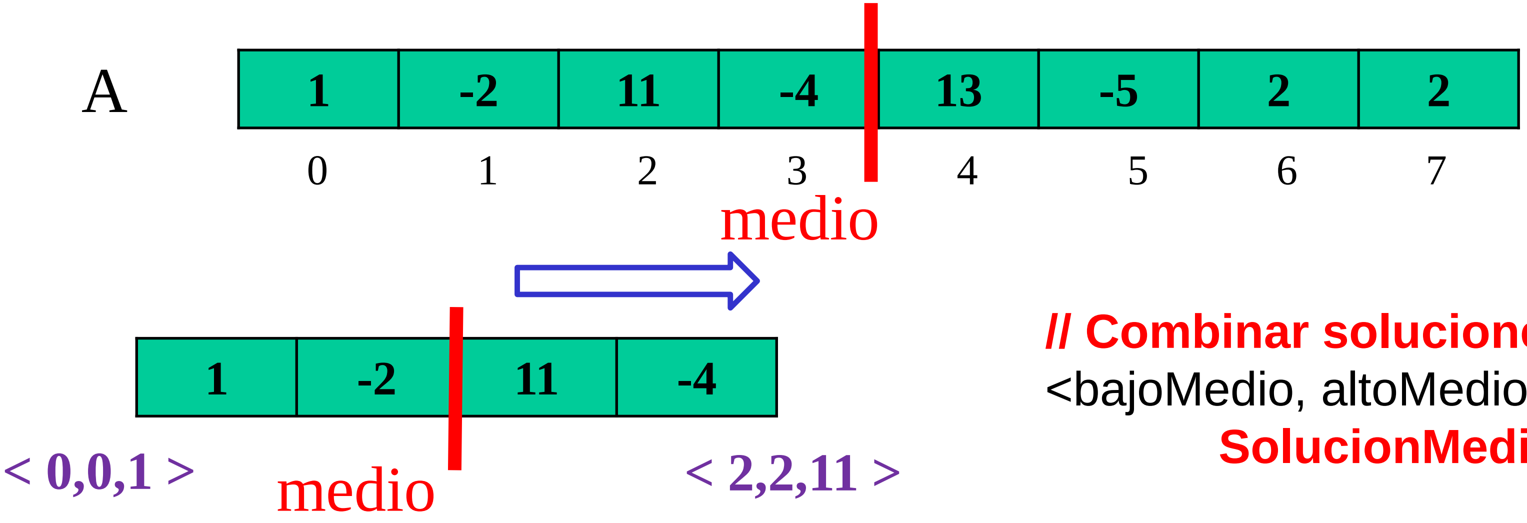
Suma + A[medio] = -2

Suma + A[0] = -2 +1 = -1

SumaIzq = -1

IndiceIzq = 0

Subsecuencia de suma máxima



A

1	-2	11	-4	13	-5	2	2
0	1	2	3	4	5	6	7

medio

1	-2	11	-4

< 0,0,1 >

medio

< 2,2,11 >

// Combinar soluciones

<bajoMedio, altoMedio, sumaMedio> =

SolucionMedio(A, bajo, medio, alto);

Suma = 0

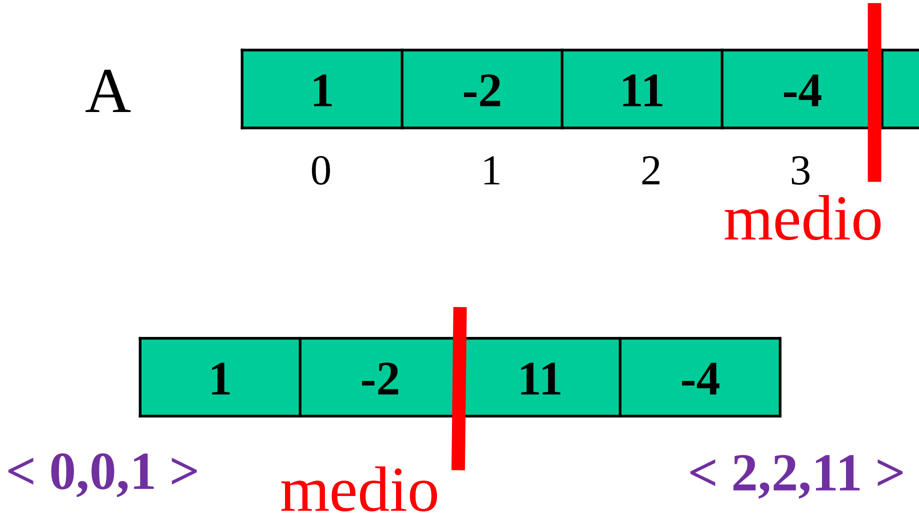
Suma + A[medio+1] = 11

Suma + A[3] = 11-4 = 7

SumaDer = 11

IndiceDer = 2

Subsecuencia de suma máxima



A

A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
	0	1	2	3	4	5	6	7

medio

1	**-2**	**11**	**-4**

< 0,0,1 >

medio

< 2,2,11 >

// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
SolucionMedio(A, bajo, medio, alto);

SumaIzq = -1

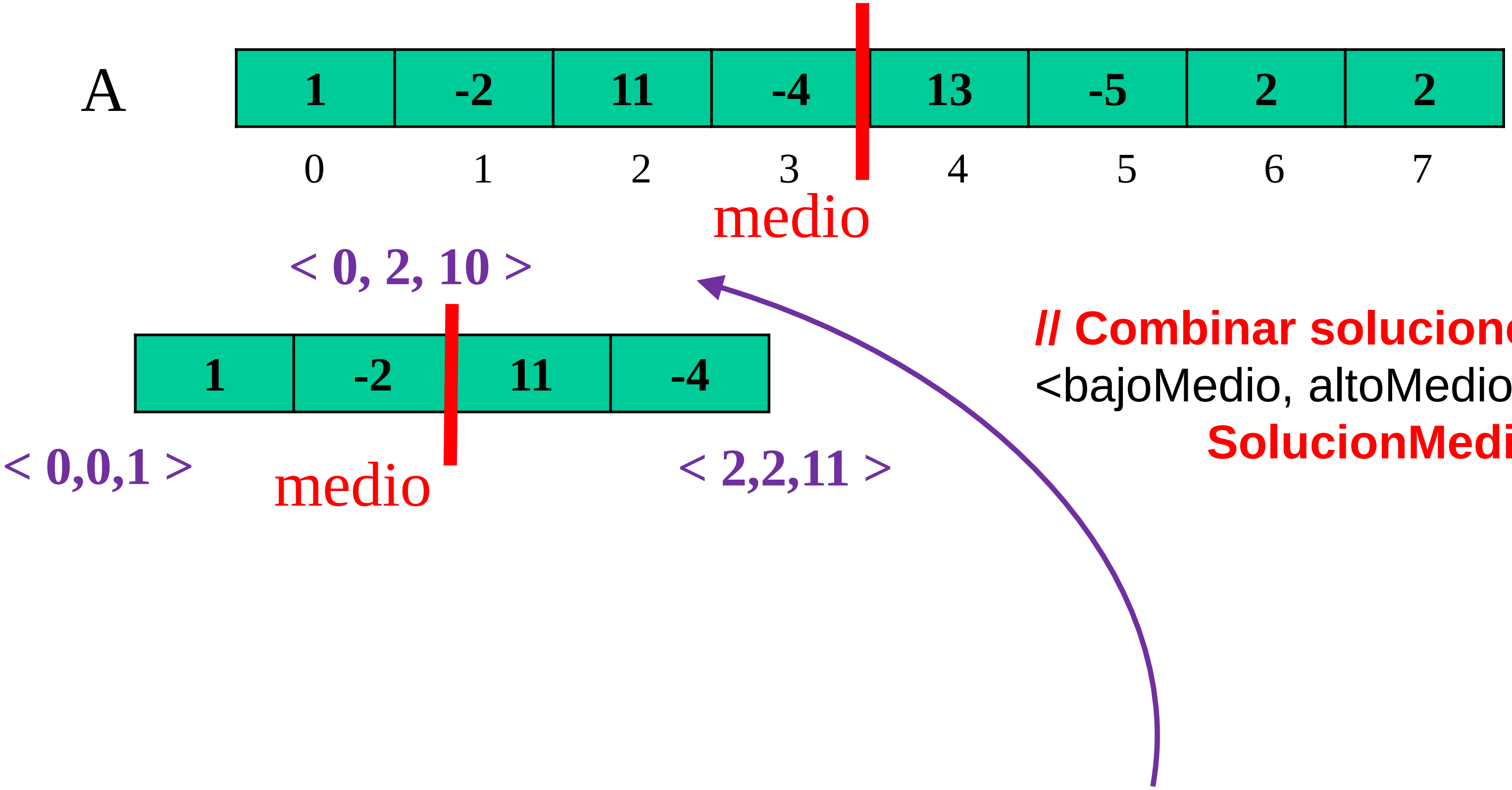
IndiceIzq = 0

SumaDer = 11

IndiceDer = 2

return < IndiceIzq, IndiceDer, sumaIzq + sumaDer > **< 0, 2, 10 >**

Subsecuencia de suma máxima



A

1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
0	1	2	3	4	5	6	7

medio

< 0, 2, 10 >

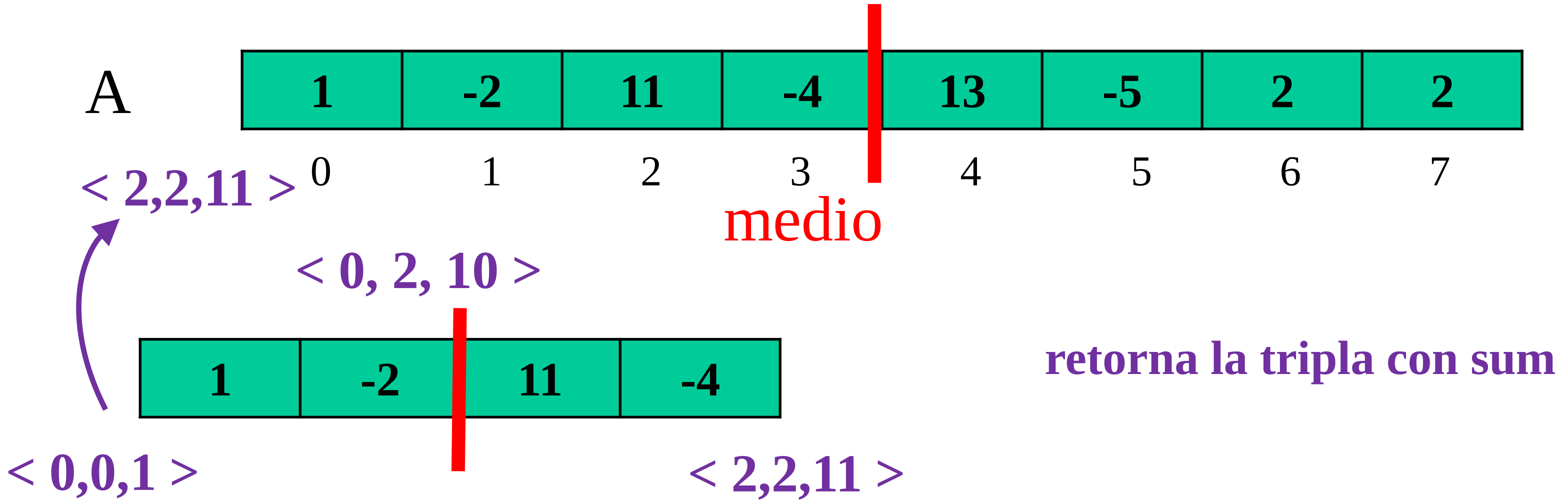
1	**-2**	**11**	**-4**

< 0,0,1 > medio **< 2,2,11 >**

// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
SolucionMedio(A, bajo, medio, alto);

return < IndiceIzq, IndiceDer, sumaIzq + sumaDer > **< 0, 2, 10 >**

Subsecuencia de suma máxima

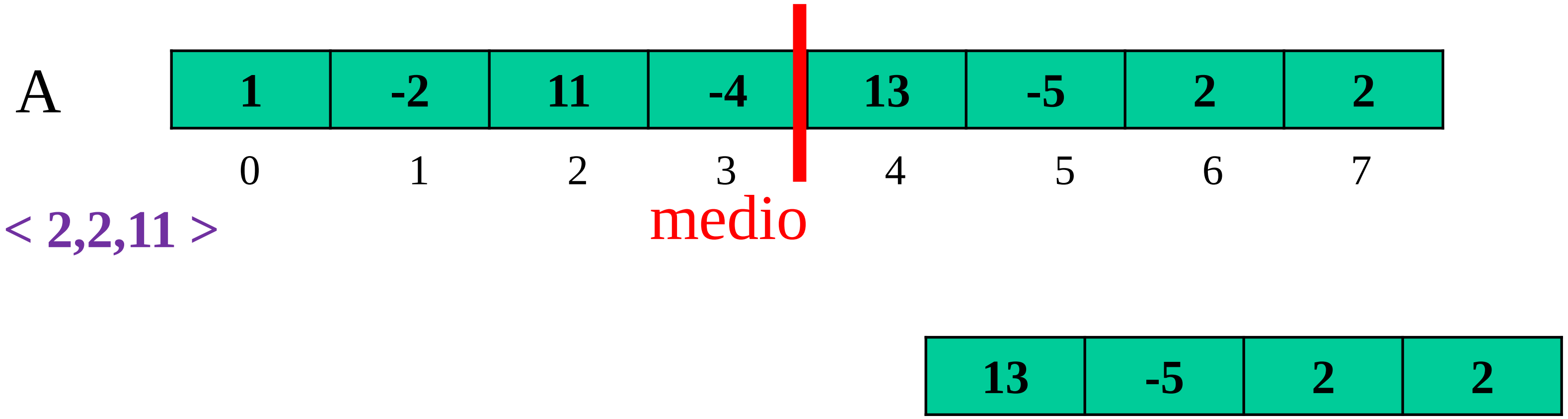


A	1	-2	11	-4	13	-5	2	2
	0	1	2	3	4	5	6	7

1	-2	11	-4

retorna la tripla con suma máxima

Subsecuencia de suma máxima



A	1	-2	11	-4	13	-5	2	2
	0	1	2	3	4	5	6	7

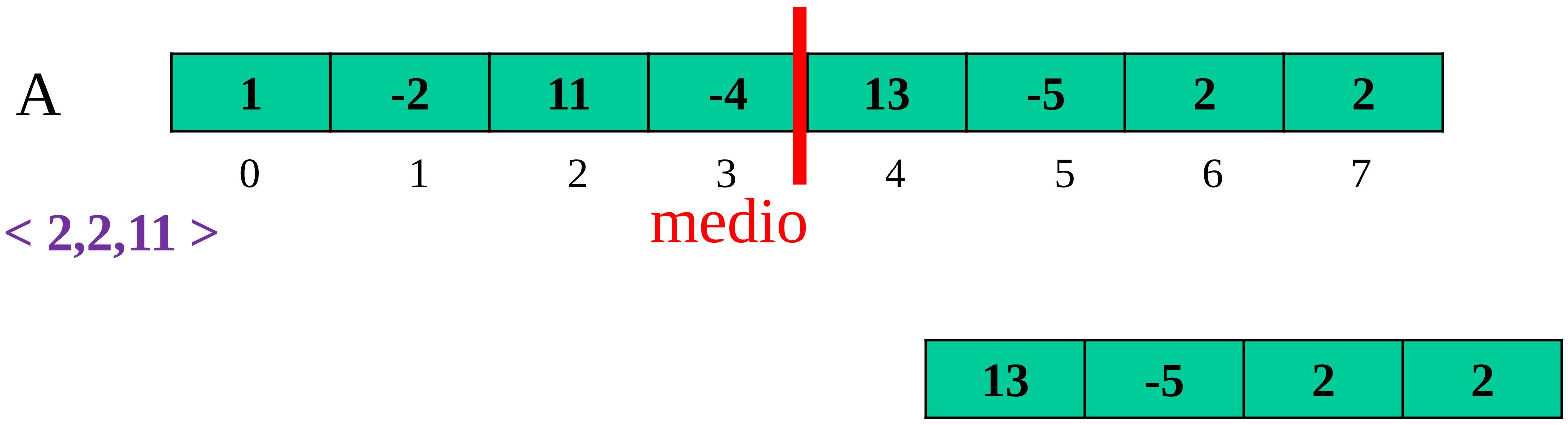
medio

< 2,2,11 >

13	-5	2	2

resuelto el subproblema de la parte Izquierda, nos resta resolver el subproblema de la parte Derecha

Subsecuencia de suma máxima



A	1	-2	11	-4	13	-5	2	2
	0	1	2	3	4	5	6	7

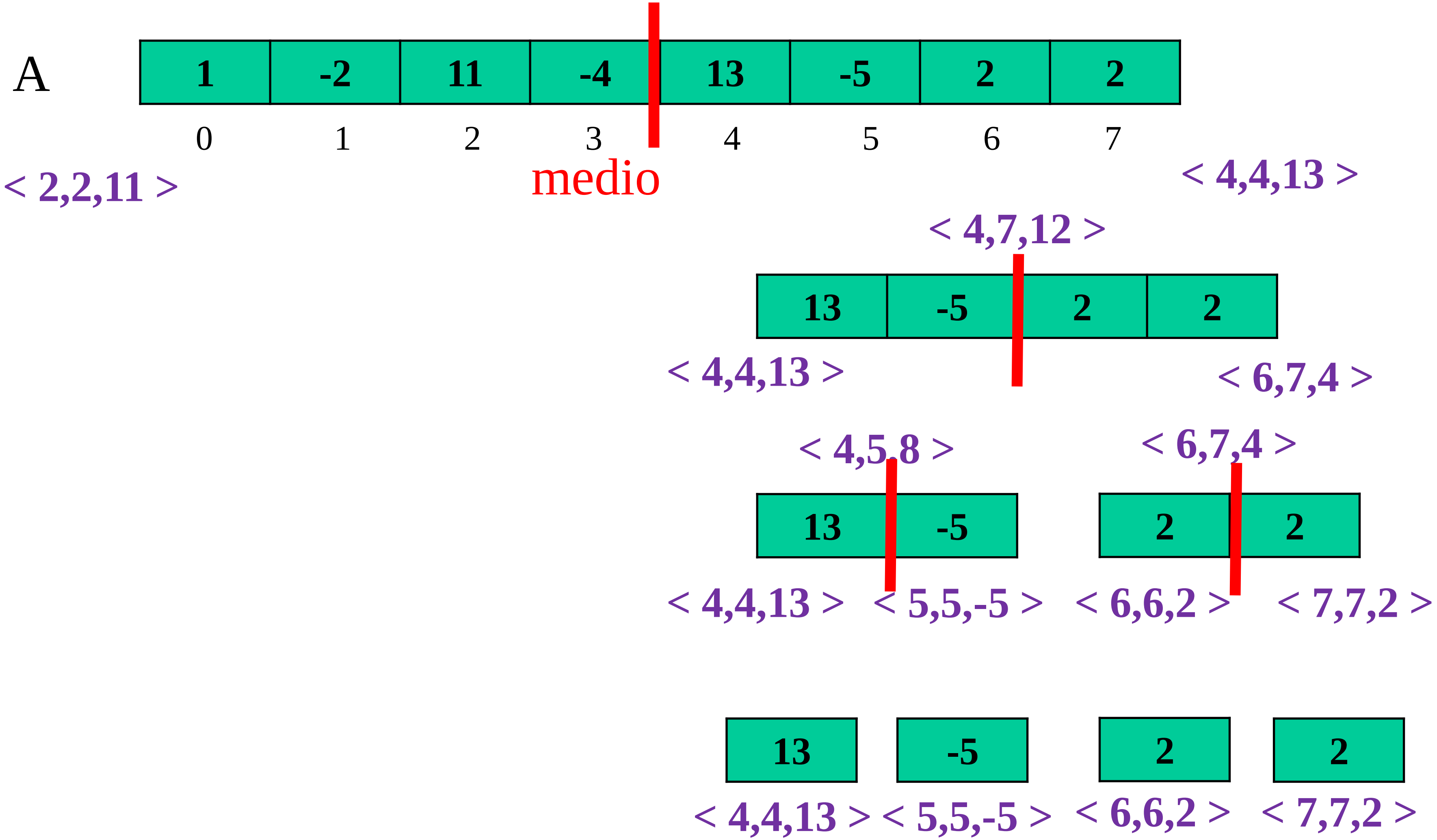
medio

< 2,2,11 >

13	-5	2	2

```
Encontrar_SSM ( A, bajo, alto ) {
…

 // Subproblema: Parte izquierda
…
// Subproblema: Parte derecha
<bajoDer, altoDer,sumaDer> = Encontrar_SSM (A, medio+1, alto);

// Combinar Soluciones
```

Subsecuencia de suma máxima



A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
	0	1	2	3	4	5	6	7

medio

< 2,2,11 >

< 4,4,13 >

< 4,7,12 >

13	**-5**	**2**	**2**

< 4,4,13 >

< 6,7,4 >

< 4,5,8 >

< 6,7,4 >

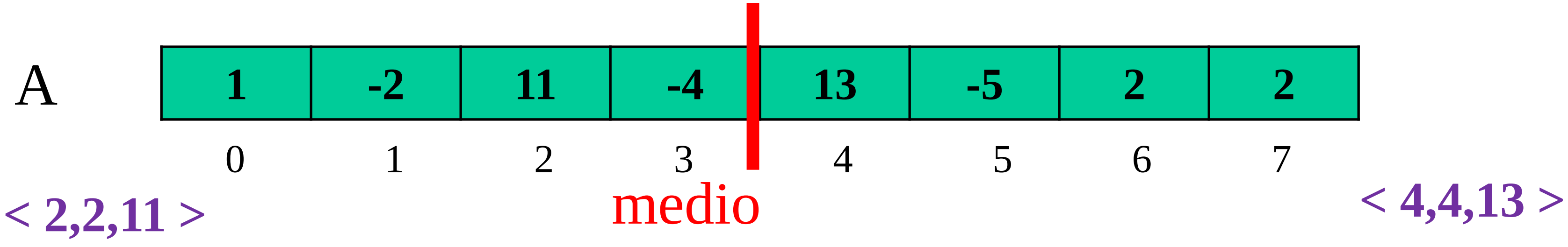
13	**-5**		**2**	**2**

< 4,4,13 > **< 5,5,-5 >** **< 6,6,2 >** **< 7,7,2 >**

13	**-5**	**2**	**2**

< 4,4,13 > **< 5,5,-5 >** **< 6,6,2 >** **< 7,7,2 >**

Subsecuencia de suma máxima



A

1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
0	1	2	3	4	5	6	7

< 2,2,11 >

medio

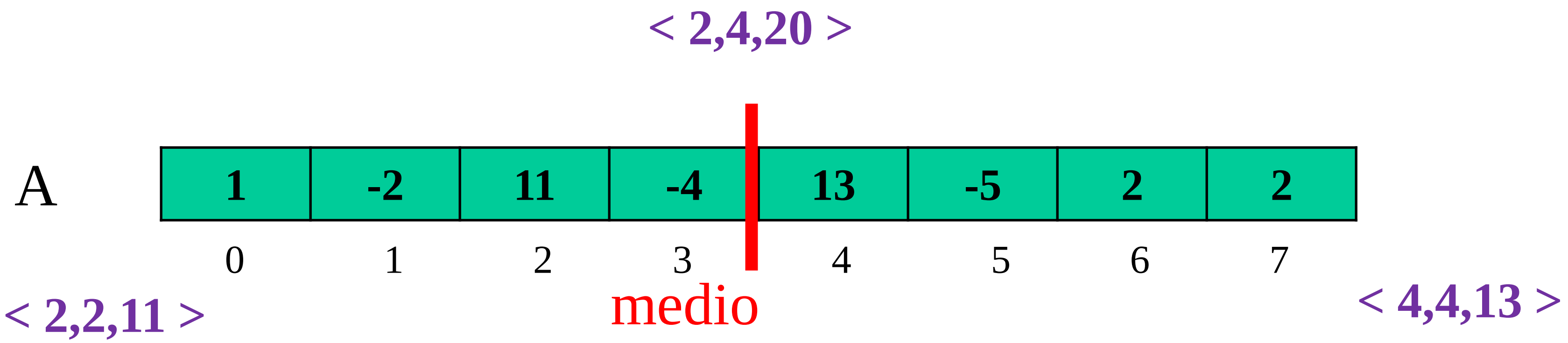
< 4,4,13 >

```
Encontrar_SSM ( A, bajo, alto ) {
…
// Combinar soluciones
<bajoMedio, altoMedio, sumaMedio> =
          SolucionMedio( A, bajo, medio, alto);
….
```

< 2,4,20 >

Ir a SolucionMedio

Subsecuencia de suma máxima



< 2,4,20 >

A

1	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
0	1	2	3	4	5	6	7

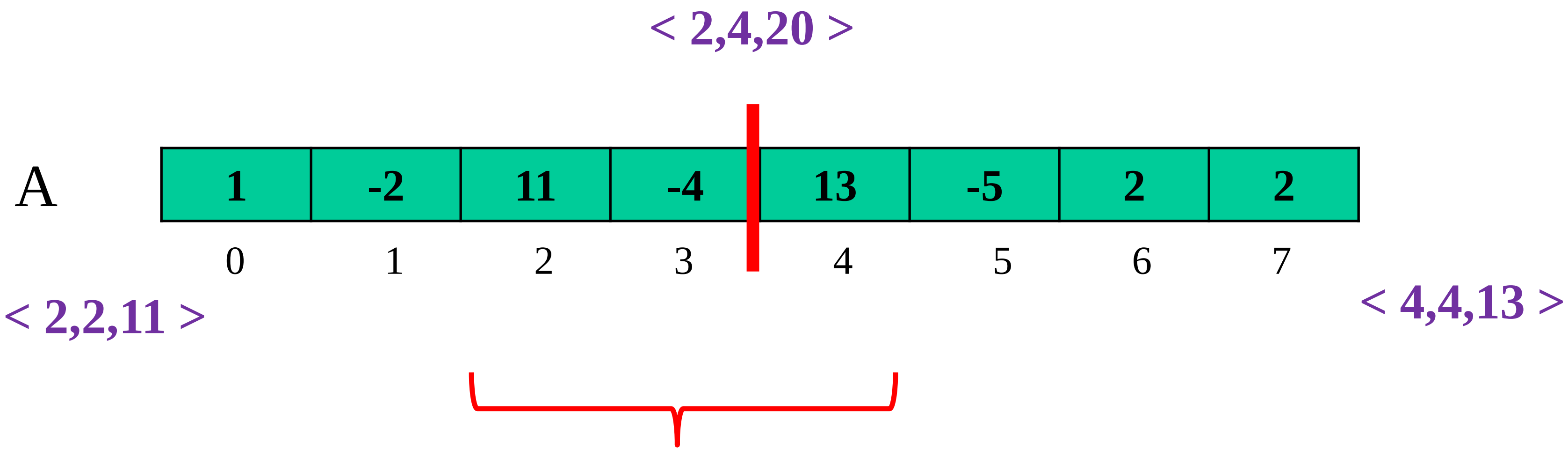
medio

< 2,2,11 >

< 4,4,13 >

```
Encontrar_SSM ( A, bajo, alto ) {
…
// Combinar soluciones
  … SolucionMedio …

If ((sumaIzq > sumaDer ) and ( sumaIzq > sumaMedio))
      return < bajoIzq, altoIzq, sumaIzq>;

else If ((sumaDer >= sumaIzq) and (sumaDer > = sumaMedio))
     return < bajoDer, altoDer, sumaDer>;

else return < bajoMedio, altoMedio, sumaMedio>
```

Subsecuencia de suma máxima



< 2,4,20 >

A	**1**	**-2**	**11**	**-4**	**13**	**-5**	**2**	**2**
	0	1	2	3	4	5	6	7

< 2,2,11 >

< 4,4,13 >

Subsecuencia de suma máxima → suma = 20

Subsecuencia de suma máxima

Complejidad temporal:

- Algoritmo por fuerza bruta → **$O(n^2)$**
- Algoritmo por Divide y Conquista

$$T(n) = \begin{cases} co & n=1 \\ 2\,T(n/2) + \underbrace{n\,c1}_{T_{\text{SolucionMedio}}} & n>1 \end{cases}$$

Subsecuencia de suma máxima

Complejidad temporal:

- Algoritmo por fuerza bruta → **O (n^2)**
- Algoritmo por Divide y Conquista → **O (n log n)**

$$T(n) = \begin{cases} co & n=1 \\ 2\,T(n/2) + \underbrace{n\,c1}_{T_{SolucionMedio}} & n>1 \end{cases}$$

Subsecuencia de suma máxima

Para pensar:

- ✓ Escriba un algoritmo no recursivo que en tiempo lineal resuelva el problema de hallar la subsecuencia de suma máxima

Pistas:

Ninguna secuencia de suma máxima comienza o termina con un número negativo.

Recorrer el arreglo de izquierda a derecha, guardando el subarreglo máximo encontrado hasta el momento.

Si conoce el subarreglo de suma máxima A[1..j], extienda la solución para encontrar un subarreglo máximo que termina en j+1, usando la siguiente información:

Para algún 1 <= i <= j+1, un subarreglo máximo de A[1..j+1] es

- o un subarreglo máximo de A[1..j]
- o un subarreglo máximo de A[i…j+1]

BIBLIOGRAFÍA

- Cormen, T.; Lieserson, C.; Rivest, R. **Introduction to Algorithms** Ed. The MIT Press. 2009.
- Horowitz, E.; Sahni, S.; Rajasekaran, S. **Computer Algorithms.** Computer Science Press.1998.
- Brassard, G.; Bratley, P. Prentice-Hall. **Fundamentos de Algoritmia**. 1997.