Análisis de Algoritmos
Lectura: Técnicas de diseño de algoritmos

Rodolfo Conde
rodolfo.conde@itam.mx

Instituto Tecnológico Autónomo de México
Maestría en Ciencias de la Computación

29 de septiembre de 2020

Contenido

Diseño y arquitectura de algoritmos

En esta lectura aprenderemos algunas de las técnicas que existen para diseñar algoritmos. Para ello, introduciremos algunos problemas conocidos que nos permitiran utilizar estas técnicas para diseñar algoritmos para resolverlos. Haciendo esto, podremos seguir practicando el análisis de la correctez y el tiempo de ejecución de los algoritmos.

Nuestra referencia principal es [Cor+09].

A continuación

Técnicas para el diseño de algoritmos

Dividir y conquistar

Notas y ejercicios

Arquitectura y diseño de algoritmos

Aunque en general nadie nos obliga a seguir un método especifico para construir un algoritmo, sí existen varias metodologias que si bien no sirven para absolutamente todos los problemas, si son aplicables a muchos tipos de problemas de los cuales, siempre vamos a querer encontrar el algoritmo más eficiente posible.

Arquitectura y diseño de algoritmos

Aunque en general nadie nos obliga a seguir un método especifico para construir un algoritmo, sí existen varias metodologias que si bien no sirven para absolutamente todos los problemas, si son aplicables a muchos tipos de problemas de los cuales, siempre vamos a querer encontrar el algoritmo más eficiente posible. Veamos algunas de estas técnicas y su descripción general.

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Incremental El algoritmo construye la solución para una parte pequeña de la entrada y entonces va procesando "pedazos" pequeños de la entrada para enriquecer la solución parcial, para al final del proceso, obtener la solución completa

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Incremental El algoritmo construye la solución para una parte pequeña de la entrada y entonces va procesando "pedazos" pequeños de la entrada para enriquecer la solución parcial, para al final del proceso, obtener la solución completa (Ejemplo: Insertion-Sort).

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Dividir y Conquistar Es una técnica aplicada en muchos algoritmos recursivos, consta de tres pasos fundamentales:

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Dividir y Conquistar Es una técnica aplicada en muchos algoritmos recursivos, consta de tres pasos fundamentales:

- **Dividir** el problema en cierto número de subproblemas (disjuntos) que son instancias más pequeñas del problema a resolver.

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Dividir y Conquistar Es una técnica aplicada en muchos algoritmos recursivos, consta de tres pasos fundamentales:

- **Dividir** el problema en cierto número de subproblemas (disjuntos) que son instancias más pequeñas del problema a resolver.
- **Conquistar** los subproblemas resolviendolos de forma recursiva. Si los subproblemas son suficientemente pequeños, se pueden resolver de forma directa (casos base).

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Dividir y Conquistar Es una técnica aplicada en muchos algoritmos recursivos, consta de tres pasos fundamentales:

- **Combinar** las soluciones de los subproblemas en la solución del problema original.

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Programación[1] Dinámica Técnica similar a la de Dividir y Conquistar, excepto que en esta, los subproblemas en los que se dividen al problema original, pueden tener "subsubproblemas.en común. Las soluciones a estos subsubproblemas se guardan en una tabla que el algoritmo puede consultar para no volver a resolverlos. Esta técnica se aplica principalmente a Problemas de optimización.

[1]"Programación"se refiere en este contexto a un método tabular

Algoritmos: Técnicas básicas

Las técnicas para el diseño de algoritmos que veremos son:

Algoritmos ambiciosos En esta técnica, se construye el algoritmo de tal formal que este siempre busca hacer la elección que en ese momento luce como "la mejor solución". Esto puede ser interpretado también como hacer la elección de la mejor solución "local", con la esperanza de que esta nos llevara a la mejor solución global (final). Los algoritmos ambiciosos no siempre nos llevan a la mejor solución, pero en algunos casos sí lo hacen.

A continuación

Técnicas para el diseño de algoritmos

Dividir y conquistar

Notas y ejercicios

Técnica: Dividir y Conquistar

Comenzaremos nuestro estudio de técnicas de diseño de algoritmos con la de Dividir y Conquistar[2].

[2] De seguro nacida de la famosa técnica de guerra "Divide y Venceras".

Técnica: Dividir y Conquistar

En la técnica de Dividir y Conquistar, solucionamos un problema dado de forma recursiva, aplicando tres pasos en cada nivel de la recursión:

Dividir el problema en cierto número de subproblemas (disjuntos) que son instancias más pequeñas del problema a resolver.

Conquistar los subproblemas resolviendolos de forma recursiva. Si los subproblemas son suficientemente pequeños, se pueden resolver de forma directa (casos base).

Combinar las soluciones de los subproblemas en la solución del problema original.

Dividir y Conquistar: El problema del subarreglo máximo

Veamos un ejemplo de la aplicación de esta técnica. Primero presentamos el problema del ejemplo. El problema del subarreglo máximo, definido así:

Entrada: Un arreglo numérico A de tamaño $n \geqslant 1$.

Salida: Una tupla de números (i, j, s), tal que s es la suma de los elementos del subarreglo $A[i \ldots j]$ y esta suma es máxima entre todas las sumas de todos los posibles subarreglos de A.

Subarreglo máximo: Aplicaciones

Algunas aplicaciones de este problema son:

- En economia (Compra-venta de bienes, comprar primero, despues vender. Se puede conocer los precios a futuro, hay que máximizar la ganacia).

Subarreglo máximo: Aplicaciones

Análisis y primeras soluciones (En pizarron).

Dividir y Conquistar vs Subarreglo máximo

Ahora analicemos este problema tratando de aplicar la técnica de Dividir y Conquistar (DyC).

Dividir y Conquistar vs Subarreglo máximo

DyC nos recomienda intentar dividir el problema global en dos subproblemas ajenos.

Dividir y Conquistar vs Subarreglo máximo

DyC nos recomienda intentar dividir el problema global en dos subproblemas ajenos. ¿Podemos hacer eso con subarreglo máximo?

Dividir y Conquistar vs Subarreglo máximo

Si partimos el arreglo de entrada a la mitad

Dividir y Conquistar vs Subarreglo máximo

Si partimos el arreglo de entrada a la mitad ¿Donde queda un posible subarreglo máximo?

Dividir y Conquistar vs Subarreglo máximo

Si partimos el arreglo de entrada a la mitad ¿Donde queda un posible subarreglo máximo?

- En la primera mitad

Dividir y Conquistar vs Subarreglo máximo

Si partimos el arreglo de entrada a la mitad ¿Donde queda un posible subarreglo máximo?

- En la primera mitad
- En la segunda mitad

Dividir y Conquistar vs Subarreglo máximo

Si partimos el arreglo de entrada a la mitad ¿Donde queda un posible subarreglo máximo?

- En la primera mitad
- En la segunda mitad
- Entre la primera y la segunda mitad

Dividir y Conquistar vs Subarreglo máximo

¡¡ Las tres respuestas anteriores son correctas !!

Dividir y Conquistar vs Subarreglo máximo

¡¡ Las tres respuestas anteriores son correctas !!

No al mismo tiempo

Dividir y Conquistar vs Subarreglo máximo

¡¡ Las tres respuestas anteriores son correctas !!

¿Estrategia recursiva?

Dividir y Conquistar vs Subarreglo máximo

Lo anterior sugiere, a grandes rasgos, la siguiente estrategia, dado un arreglo de entrada A:

Dividir y Conquistar vs Subarreglo máximo

Lo anterior sugiere, a grandes rasgos, la siguiente estrategia, dado un arreglo de entrada A:

- Calcular el punto medio *imedio* de A.

Dividir y Conquistar vs Subarreglo máximo

Lo anterior sugiere, a grandes rasgos, la siguiente estrategia, dado un arreglo de entrada A:

- Calcular el punto medio *imedio* de A.
- Encontrar el subarreglo máximo de $A\,[0, \ldots imedio]$

Dividir y Conquistar vs Subarreglo máximo

Lo anterior sugiere, a grandes rasgos, la siguiente estrategia, dado un arreglo de entrada A:

- Calcular el punto medio *imedio* de A.
- Encontrar el subarreglo máximo de $A\,[0, \ldots imedio]$
- Encontrar el subarreglo máximo de $A\,[imedio + 1, \ldots A.length - 1]$

Dividir y Conquistar vs Subarreglo máximo

Lo anterior sugiere, a grandes rasgos, la siguiente estrategia, dado un arreglo de entrada A:

- Calcular el punto medio *imedio* de A.
- Encontrar el subarreglo máximo de $A\,[0, \ldots imedio]$
- Encontrar el subarreglo máximo de $A\,[imedio + 1, \ldots A.length - 1]$
- Encontrar el subarreglo máximo de A, pero que cruza el punto medio *imedio*

Dividir y Conquistar vs Subarreglo máximo

Lo anterior sugiere, a grandes rasgos, la siguiente estrategia, dado un arreglo de entrada A:

- Calcular el punto medio *imedio* de A.
- Encontrar el subarreglo máximo de $A\,[0, \ldots imedio]$
- Encontrar el subarreglo máximo de $A\,[imedio + 1, \ldots A.length - 1]$
- Encontrar el subarreglo máximo de A, pero que cruza el punto medio *imedio*
- Comparar los 3 subarreglos maximos para ver cual es el más grande y declararlo el subarreglo máximo.

Dividir y Conquistar vs Subarreglo máximo

- Encontrar el subarreglo máximo de $A[0, \ldots imedio]$
- Encontrar el subarreglo máximo de $A[imedio + 1, \ldots A.length - 1]$

Estos son instancias del subarreglo máximo mas pequeñas que el original.

Dividir y Conquistar vs Subarreglo máximo

- Encontrar el subarreglo máximo de A, pero que cruza el punto medio *imedio*

¿Y este problema, es el mismo?

Dividir y Conquistar vs Subarreglo máximo: Nuevo subproblema

Resulta ser que

- El problema de encontrar un subarreglo máximo que cruce el punto medio, no es el mismo problema original,
- Necesitamos resolver ese problema cada vez que llamemos de forma recursiva al método principal de encontrar el subarreglo máximo.
- Este comportamiento es muy común en los prblemas que se resuelven con la técnica de DyC.

Dividir y Conquistar vs Subarreglo máximo: Nuevo subproblema

Resulta ser que

- El problema de encontrar un subarreglo máximo que cruce el punto medio, no es el mismo problema original,
- Necesitamos resolver ese problema cada vez que llamemos de forma recursiva al método principal de encontrar el subarreglo máximo.
- Este comportamiento es muy común en los prblemas que se resuelven con la técnica de DyC.

Llamaremos a este problema El subarreglo máximo que cruza el punto medio.

Solución para Subarreglo máximo que cruza el punto medio I

```
function FIND-MAX-CROSSING-SUBARRAY(A, ibajo, imedio, ialto)
    sumaIzquierda = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaIzquierda then
            sumaIzquierda = suma
            maxIzquierdo = i
        end if
    end for
    sumaDerecha = −∞
    suma = 0
    for j = imedio + 1 to ialto do
        suma = suma + A[j]
```

Solución para Subarreglo máximo que cruza el punto medio II

if $suma > sumaDerecha$ **then**
    $sumaDerecha = suma$
    $maxDerecho = j$
end if
end for
return $(maxIzquierdo, maxDerecho, sumaIzquierda + sumaDerecha)$
end function

Solución para Subarreglo máximo que cruza el punto medio I

Con la solución del problema del Subarreglo máximo que cruza el punto medio, podemos escribir la solución recursiva del problema del Subarreglo máximo

DyC vs Subarreglo máximo: Batalla final I

function FIND-MAXIMUM-SUBARRAY(A, ibajo, ialto)
    if $ibajo == ialto$ **then return** $(ibajo, ialto, A[ibajo])$ // Caso base es un elemento
    else
        $imedio = \lfloor (ibajo + ialto) \div 2 \rfloor$
        $(izqBajo, izqAlto, sumaIzq) =$ Find-Maximum-Subarray$(A, ibajo, imedio)$
        $(derBajo, derAlto, sumaDer) =$ Find-Maximum-Subarray$(A, imedio + 1, ialto)$
        $(cruzBajo, cruzAlto, sumaCruz) =$ Find-Maximum-Crossing-Subarray$(A, ibajo, imedio, ialto)$
        if $sumaIzq \geq sumaDer$ and $sumaIzq \geq sumaCruz$ **then**
            return $(izqBajo, izqAlto, sumaIzq)$
        else if $sumaDer \geq sumaIzq$ and $sumaDer \geq sumaCruz$ **then**
            return $(derBajo, derAlto, sumaDer)$

DyC vs Subarreglo máximo: Batalla final II

```
        else
            return (cruzBajo, cruzAlto, sumaCruz)
        end if
    end if
end function
```

DyC vs Subarreglo máximo: Batalla final

Analisis de correctez (Platicado, EJERCICIO MORAL)

DyC vs Subarreglo máximo: Batalla final

Hasta este momento, vamos bien. Find-Maximum-Subarray calcula de manera correcta el subarreglo máximo de un arreglo cualquiera.

DyC vs Subarreglo máximo: Batalla final

Pero ¿ Cual es la complejidad de tiempo de ejecución?

DyC vs Subarreglo máximo: Batalla contra reloj

Para contestar esto, debemos analizar:

- Complejidad del algoritmo Find-Max-Crossing-Subarray.
- Complejidad del algoritmo Find-Maximum-Subarray.

Subarreglo máximo que cruza: Tiempo de ejecución

Para contestar esto, debemos analizar:

- Complejidad del algoritmo Find-Max-Crossing-Subarray.
- Complejidad del algoritmo Find-Maximum-Subarray.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Tres elementos fundamentales para determinar la complejidad del tiempo de ejecución:

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Tres elementos fundamentales para determinar la complejidad del tiempo de ejecución:

- Tamaño de la entrada

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Tres elementos fundamentales para determinar la complejidad del tiempo de ejecución:

- Tamaño de la entrada
- ¿Cuánto tarda el primero ciclo?

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Tres elementos fundamentales para determinar la complejidad del tiempo de ejecución:

- Tamaño de la entrada
- ¿Cuánto tarda el primero ciclo?
- ¿Cuánto tarda el segundo ciclo?

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI +
sumaD)
        end function
```

El tamaño de la porción del arreglo a procesar esta determinado por los indices *ibajo* e *ialto*.

Subarreglo máximo que cruza: Tiempo de ejecución

function F-M-C-S(A, ibajo, imedio, ialto)
    $sumaI = -\infty$
    $suma = 0$
    for $i = imedio$ **downto** $ibajo$ **do**
        $suma = suma + A[i]$
        if $suma > sumaI$ **then**
            $sumaI = suma$
            $maxI = i$
        end if
    end for
        $sumaD = -\infty$
        $suma = 0$
        for $j = imedio + 1$ **to** $ialto$ **do**
            $suma = suma + A[j]$
            if $suma > sumaD$ **then**
                $sumaD = suma$
                $maxD = j$
            end if
        end for **return** $(maxI, maxD, sumaI + sumaD)$
        end function

El tamaño de la porción es $n = ialto - ibajo + 1$.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Subarreglo máximo que cruza: Tiempo de ejecución

function F-M-C-S(A, ibajo, imedio, ialto)
    $sumaI = -\infty$
    $suma = 0$
    for $i = imedio$ **downto** $ibajo$ **do**
        $suma = suma + A[i]$
        if $suma > sumaI$ **then**
            $sumaI = suma$
            $maxI = i$
        end if
    end for
        $sumaD = -\infty$
        $suma = 0$
        for $j = imedio + 1$ **to** $ialto$ **do**
            $suma = suma + A[j]$
            if $suma > sumaD$ **then**
                $sumaD = suma$
                $maxD = j$
            end if
        end for **return** $(maxI, maxD, sumaI + sumaD)$
        end function

- El total de iteraciones del primer ciclo es $imedio - ibajo + 1$.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

- El total de iteraciones del primer ciclo es $imedio - ibajo + 1$.
- Cada operación dentro del ciclo es elemental.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI +
sumaD)
        end function
```

- El total de iteraciones del primer ciclo es $imedio - ibajo + 1$.
- Cada operación dentro del ciclo es elemental.
- Cada iteración toma $\Theta(1)$ tiempo.

Subarreglo máximo que cruza: Tiempo de ejecución

function F-M-C-S(A, ibajo, imedio, ialto)
  $sumaI = -\infty$
  $suma = 0$
  for $i = imedio$ **downto** $ibajo$ **do**
    $suma = suma + A[i]$
    if $suma > sumaI$ **then**
      $sumaI = suma$
      $maxI = i$
    end if
  end for
    $sumaD = -\infty$
    $suma = 0$
    for $j = imedio + 1$ **to** $ialto$ **do**
      $suma = suma + A[j]$
      if $suma > sumaD$ **then**
        $sumaD = suma$
        $maxD = j$
      end if
    end for **return** $(maxI, maxD, sumaI + sumaD)$
    end function

- El total de iteraciones del primer ciclo es $imedio - ibajo + 1$.
- Cada operación dentro del ciclo es elemental.
- Cada iteración toma $\Theta(1)$ tiempo.
- El ciclo toma $\Theta(imedio - ibajo + 1)$ tiempo.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Por observaciones similares a las del primer ciclo, para el segundo ciclo:

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = -∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = -∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI +
sumaD)
        end function
```

Por observaciones similares a las del primer ciclo, para el segundo ciclo:

- Iteraciones: *ialto* − *imedio*.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Por observaciones similares a las del primer ciclo, para el segundo ciclo:

- Iteraciones: *ialto* − *imedio*.
- Cada iteración: $\Theta(1)$ tiempo.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Por observaciones similares a las del primer ciclo, para el segundo ciclo:

- Iteraciones: $ialto - imedio$.
- Cada iteración: $\Theta(1)$ tiempo.
- El ciclo toma $\Theta(ialto - imedio)$ tiempo.

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Finalmente, el tiempo de ejecución total sera:

Subarreglo máximo que cruza: Tiempo de ejecución

```
function F-M-C-S(A, ibajo, imedio, ialto)
    sumaI = −∞
    suma = 0
    for i = imedio downto ibajo do
        suma = suma + A[i]
        if suma > sumaI then
            sumaI = suma
            maxI = i
        end if
    end for
        sumaD = −∞
        suma = 0
        for j = imedio + 1 to ialto do
            suma = suma + A[j]
            if suma > sumaD then
                sumaD = suma
                maxD = j
            end if
        end for return (maxI, maxD, sumaI + sumaD)
        end function
```

Finalmente, el tiempo de ejecución total sera:

$$\Theta(imedio - ibajo + 1) + \Theta(ialto - imedio) = \Theta(ialto - ibajo + 1) = \Theta(n)$$

Subarreglo máximo: Tiempo de ejecución

Sabiendo el tiempo de ejecución del método Find-Maximum-Crossing-Subarray, podemos calcular el tiempo de ejecución del método Find-Maximum-Subarray.

Subarreglo máximo: Tiempo de ejecución

function F-M-S(A, ibajo, ialto)
  if $ibajo == ialto$ **then**
    return $(ibajo, ialto, A[ibajo])$
  else
    $imedio = \lfloor (ibajo + ialto) \div 2 \rfloor$
    $(izB, izA, sIz) =$ F-M-S$(A, ibajo, imedio)$
    $(deB, deA, sDe) =$
F-M-S$(A, imedio + 1, ialto)$
    $(crB, crA, sCr) =$
F-M-C-S$(A, ibajo, imedio, ialto)$
    if $sIz \geq sDe$ and $sIz \geq sCr$ **then**
      return (izB, izA, sIz)
    else if $sDe \geq sIz$ and $sDe \geq sCr$ **then**
      return (deB, deA, sDe)
    else
      return (crB, crA, sCr)
    end if
  end if
end function

¿Pero Como?

Subarreglo máximo: Tiempo de ejecución

```
function F-M-S(A, ibajo, ialto)
    if ibajo == ialto then
        return (ibajo, ialto, A[ibajo])
    else
        imedio = ⌊(ibajo + ialto) ÷ 2⌋
        (izB, izA, sIz) = F-M-S(A, ibajo, imedio)
        (deB, deA, sDe) =
F-M-S(A, imedio + 1, ialto)
        (crB, crA, sCr) =
F-M-C-S(A, ibajo, imedio, ialto)
        if sIz ≥ sDe  and  sIz ≥ sCr then
            return (izB, izA, sIz)
        else if sDe ≥ sIz  and  sDe ≥ sCr then
            return (deB, deA, sDe)
        else
            return (crB, crA, sCr)
        end if
    end if
end function
```

¡¡ Al Pizarrón !!

Subarreglo máximo: Notas adicionales

Por cierto

- Existe un algoritmo de tiempo lineal para el problema del subarreglo máximo.
- DyC: Buen algoritmo, pero no el mejor.
- Muchas veces, DyC si nos da el mejor algoritmo posible.
- Problema con solución similar:

Subarreglo máximo: Notas adicionales

Por cierto

- Existe un algoritmo de tiempo lineal para el problema del subarreglo máximo.
- DyC: Buen algoritmo, pero no el mejor.
- Muchas veces, DyC si nos da el mejor algoritmo posible.
- Problema con solución similar:
  - Ordenamiento

Subarreglo máximo: Notas adicionales

Por cierto

- Existe un algoritmo de tiempo lineal para el problema del subarreglo máximo.
- DyC: Buen algoritmo, pero no el mejor.
- Muchas veces, DyC si nos da el mejor algoritmo posible.
- Problema con solución similar:
  - Ordenamiento
  - Merge-Sort

A continuación

Técnicas para el diseño de algoritmos

Dividir y conquistar
- Recurrencias

Notas y ejercicios

Dividir y Conquistar: Recurrencias

En los algoritmos diseñados con la técnica de DyC es muy común que aparezcan funciones de complejidad del tiempo de ejecución de la siguiente forma:

$$T(n) = aT(n/b) + f(n) \tag{1}$$

Dividir y Conquistar: Recurrencias

En los algoritmos diseñados con la técnica de DyC es muy común que aparezcan funciones de complejidad del tiempo de ejecución de la siguiente forma:

$$T(n) = aT(n/b) + f(n) \tag{1}$$

n es el tamaño de la entrada

Dividir y Conquistar: Recurrencias

En los algoritmos diseñados con la técnica de DyC es muy común que aparezcan funciones de complejidad del tiempo de ejecución de la siguiente forma:

$$T(n) = aT(n/b) + f(n) \tag{1}$$

n es el tamaño de la entrada
$a \geq 1$ y $b > 1$

Dividir y Conquistar: Recurrencias

Una recurrencia de esta forma nos describe un algoritmo construido con la técnica de DyC tal que

$$T(n) = aT(n/b) + f(n)$$

Dividir y Conquistar: Recurrencias

Una recurrencia de esta forma nos describe un algoritmo construido con la técnica de DyC tal que

$$T(n) = aT(n/b) + f(n)$$

- construye a subproblemas

Dividir y Conquistar: Recurrencias

Una recurrencia de esta forma nos describe un algoritmo construido con la técnica de DyC tal que

$$T(n) = aT(n/b) + f(n)$$

- construye a subproblemas
- cada subproblema es de tamaño $\frac{1}{b}$ con respecto al tamaño original del problema.

Dividir y Conquistar: Recurrencias

Una recurrencia de esta forma nos describe un algoritmo construido con la técnica de DyC tal que

$$T(n) = aT(n/b) + f(n)$$

- construye a subproblemas
- cada subproblema es de tamaño $\frac{1}{b}$ con respecto al tamaño original del problema.
- Los pasos de dividir y combinar juntos, toman tiempo de ejecución de $f(n)$.

Recurrencias: Formas de encontrar soluciones

Existen tres métodos para resolver recurrencias:

- Substitución.
- El método del árbol.
- El Teorema Maestro.

Recurrencias: Formas de encontrar soluciones

Teorema 1 (El método maestro)

Sean $a \geq 1, b > 1$ *constantes,* $f : \mathbb{N} \to \mathbb{N}$ *una función y* $T : \mathbb{N} \to \mathbb{N}$ *definida por la recurrencia*

$$T(n) = aT(n/b) + f(n),$$

donde interpretamos al término n/b *como el piso ó el techo del número* n/b*. Entonces* $T(n)$ *tiene las siguientes cotas asintóticas:*

1. *Si* $f(n) = O(n^{\log_b a - \epsilon})$ *para* $\epsilon > 0$*, entonces* $T(n) = \Theta(n^{\log_b a})$*.*
2. *Si* $f(n) = \Theta(n^{\log_b a})$*, entonces* $T(n) = \Theta(n^{\log_b a} \log n)$*.*
3. *Si* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *para* $\epsilon > 0$*, y si se cumple que* $af(n/b) \leq cf(n)$ *para* $c < 1$ *y* n *suficientemente grande, entonces* $T(n) = \Theta(f(n))$*.*

A continuación

Técnicas para el diseño de algoritmos

Dividir y conquistar

Notas y ejercicios

Ejercicios del capítulo I

EJERCICIO 1

Formaliza el enunciado para definir el problema del subarreglo máximo que cruza el punto medio.

EJERCICIO 2

Construye un algoritmo de tiempo lineal para el problema Subarreglo-Maximo.

EJERCICIO 3

Ejercicios del capítulo II

Demuestra como multiplicar los números complejos $a + bi$ y $c + di$ usando unicamente 3 multiplicaciones de números reales. Tu algoritmo debe tomar como entrada los 4 números reales que representan los dos complejos y dar como resultado la parte real $ac - bd$ y la parte imaginaria $ad + bc$.

EJERCICIO 4

Demuestra que la solución para la recurrencia

$$T(n) = T(n/2) + \Theta(1).$$

es $T(n) = \log n$.

Ejercicios del capítulo III

EJERCICIO 5

Da una cota asintótica para la recurrencia $T(n) = 4T(n/2) + n^2 \log n$.

Bibliografía I

Thomas H. Cormen y col. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.