Divide and Conquer

Algoritmos y Estructuras de Datos II

Christian G. Cossio-Mercado

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

3 de julio de 2020

Temario

Temario

Divide and Conquer

Es una **estrategia algorítmica** que consiste en

1. **Dividir** el problema en k subproblemas del mismo tipo, pero más chicos
2. **Conquistar** resolviendo los subproblemas, recursivamente o directamente (si son lo suficientemente fáciles o chicos)
3. **Combinar** las soluciones obtenidas para resolver el problema original

Los problemas pueden tener varias soluciones, pero en este contexto esperamos que puedan pensar en una solución D&C

Esquema General

DC(X)

---

if X es chico (o simple) **then**
    Retornar solución ad hoc de X
else
    Descomponer X en subinstancias X_1, X_2,...,X_k
    for $i \in [1..k]$ **do**
        $Y_i =$ **DC(X_i)**
    Combinar las soluciones Y_i para construir una solución para X

Merge Sort

Es un algoritmo que usa D&C para ordenar un arreglo
Donde, Merge fusiona de forma ordenada dos arreglos ordenados, con costo O(n)

1. **Dividir** el arreglo en dos mitades
2. **Conquistar:** Si los arreglos son de un elemento ya están ordenados, sino, hacer recursión sobre ellos y los obtengo ordenados
3. **Combinar:** Merge para obtener el arreglo ordenado a partir de las dos mitades ordenadas

Merge Sort

MergeSort(A)

if $|A| \leq 1$ **then**
  return A

A1 $\leftarrow$ MergeSort(A[0 .. $\frac{n}{2}$)) ▷ T($\frac{n}{2}$)
A2 $\leftarrow$ MergeSort(A[$\frac{n}{2}$..n)) ▷ T($\frac{n}{2}$)

A $\leftarrow$ Merge(A1, A2) ▷ $\Theta(n)$

return A

Complejidad: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Búsqueda Binaria

Podemos pensar a una búsqueda binaria como un algoritmo de D&C donde dividimos nuestro problema en un solo sub-problema

1. **Dividir** el arreglo en una mitad (elegida a partir de cuanto vale el elemento a la mitad del arreglo)
2. **Conquistar:** Si el arreglo tiene un elemento, me fijo si es el que estoy buscando, sino, hago recursión sobre la mitad en donde podría estar el elemento
3. **Combinar:** No hace falta

Búsqueda Binaria

Buscar(A, elem, l, r) $\rightarrow$ int — Buscamos en el rango [l, r)

```
if r − l = 1 then                          // Rango de un solo elemento
    if A[l] = elem then                                          ▷ Θ(1)
        return l
    else
        return -1

m ← (l + r)/2

if elem ≤ A[m] then
    return Buscar(A, elem, l, m)                                 ▷ T(n/2)
else
    return Buscar(A, elem, m, r)                                 ▷ T(n/2)
```

Complejidad: $T(n) = T(\frac{n}{2}) + \Theta(1)$

Temario

Complejidades

¿Cómo calculamos estas complejidades?

- **Merge Sort:** $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- **Búsqueda Binaria:** $T(n) = T(\frac{n}{2}) + \Theta(1)$

Recurrencias en D&C

Las recurrencias de D&C tienen la siguiente forma:

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ aT\left(\frac{n}{c}\right) + f(n) & n > k \end{cases}$$

Donde:

- k es tamaño del caso base, que en general vale 1
- a es la cantidad de subproblemas a resolver.
- c es la cantidad de particiones y $\frac{n}{c}$ es el tamaño de los subproblemas a resolver.
- $f(n)$ es el costo de todo lo que se hace en cada llamado además de los llamados recursivos, así, incluye el costo de la división ($D(n)$) y de la combinación de los resultados ($C(n)$)
- Estamos asumiendo que resolver un caso base cuesta $\Theta(1)$.

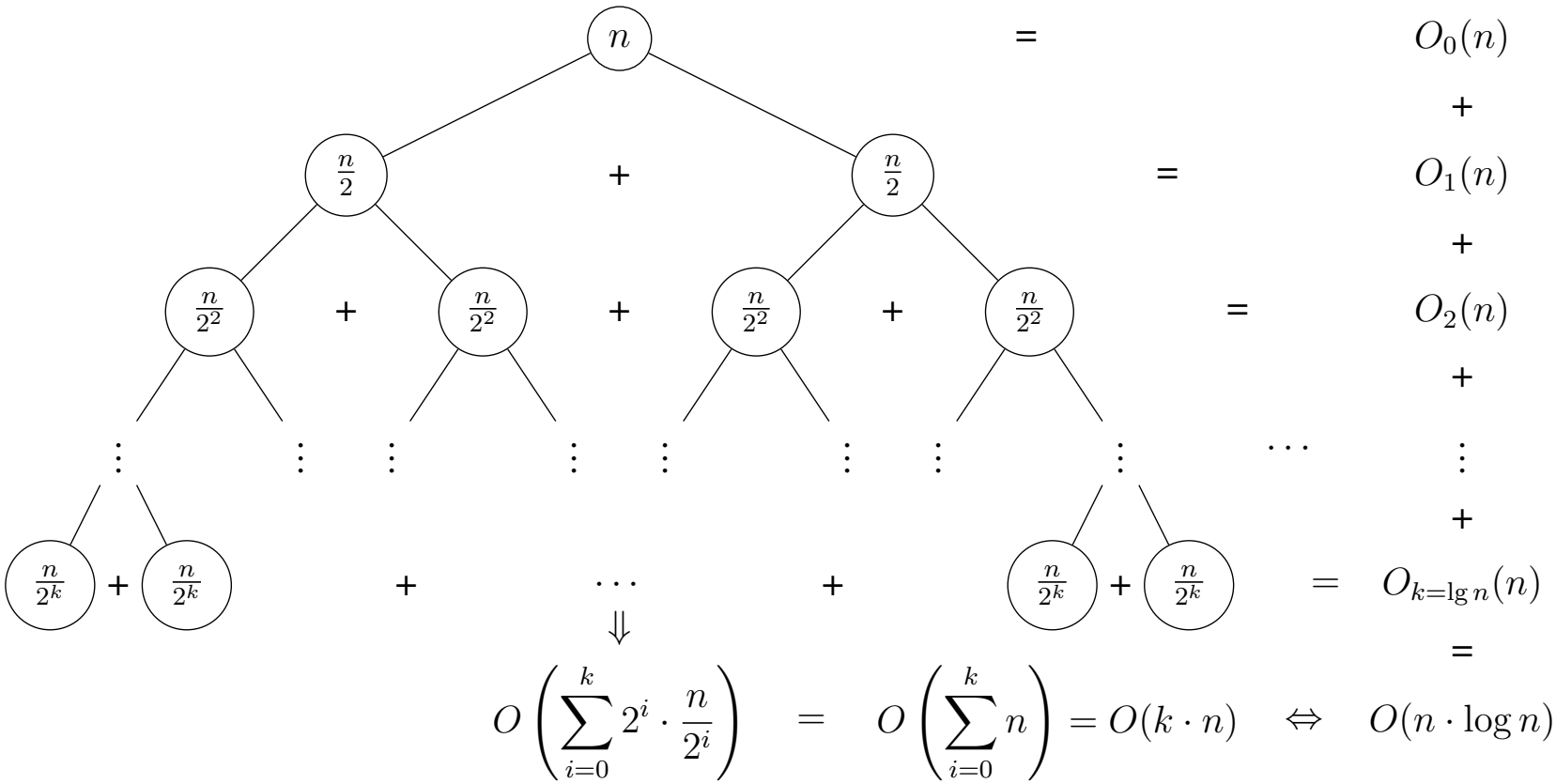
Cálculo de complejidad

- **Opción 1**: Dibujar el árbol de llamadas recursivas, calcular cuanto demora cada nodo y sumar para todos los nodos
- **Opción 2**: Adivinar cuanto va a dar y probar por inducción que tiene esa complejidad a partir de la recurrencia. Se lo conoce también como método de sustitución
- **Opción 3**: Usar el Teorema Maestro

Nota: No son las únicas opciones que existen

Árbol de recursión

$$T(n) = 2(n/2) + O(n)$$



Fuente: `https://courses.csail.mit.edu/6.006/spring11/rec/rec08.pdf`

Método de sustitución

Se basa en **proponer** una cota para T(n) y probarla por inducción.

Por ejemplo, si $T(n) = 2T(\frac{n}{2}) + n$, qvq $T(n) \in O(n \log n)$.
Por definición, qvq $T(n) \leq cn \log n$
Suponemos que vale para todos los $n' < n$, como, por ejemplo, $n' = \frac{n}{2}$.
Así, $T(n') = T(\frac{n}{2}) \leq c\ \frac{n}{2} \log(\frac{n}{2})$.

$$
\begin{aligned}
& T(n) = 2T(\tfrac{n}{2}) + n \\
\text{(sustituimos } & T(\tfrac{n}{2})) \ T(n) \leq 2\ c\ \tfrac{n}{2} \log(\tfrac{n}{2}) + n \\
& \leq c\ n \log(\tfrac{n}{2}) + n \\
& = c\ n(\log n - 1) + n \\
& = c\ n \log n - (c-1)n \\
& \leq c\ n \log n \text{ (tomando } c \geq 1)
\end{aligned}
$$

Así, para $n_0 = 2$, tenemos dos casos base ($n = 2$ y $n = 3$), y con $c \geq 2$ se cumplen los casos base. Luego se sigue con la demostración por inducción
Cuidado: Para que la demostración valga tenemos que llegar a *exactamente* lo que queríamos probar.

Teorema maestro

Si nuestra recurrencia es de la forma

$$T(n) = \begin{cases} aT\left(\frac{n}{c}\right) + f(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

(con a $\geq$ 1 y c $>$ 1) entonces:

$$T(n) = \begin{cases} \Theta(n^{\log_c a}) & \text{Si } \exists \varepsilon > 0 \text{ tal que } f(n) \in O(n^{\log_c a - \varepsilon}) \\ \Theta(n^{\log_c a} \log n) & \text{Si } f(n) \in \Theta(n^{\log_c a}) \\ \Theta(f(n)) & \text{Si } \exists \varepsilon > 0 \text{ tal que } f(n) \in \Omega(n^{\log_c a + \varepsilon}) \text{ y} \\ & \exists \delta < 1, \exists n_0 > 0 \text{ tal que } \forall n \geq n_0 \text{ se cumple: } a.f\left(\frac{n}{c}\right) \leq \delta f(n) \end{cases}$$

Teorema maestro - Merge Sort

Calculemos la complejidad de Merge Sort usando el teorema maestro.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- $a = 2$, cantidad de subproblemas
- $c = 2$, cantidad de particiones ($\frac{n}{2}$ el tamaño del subproblema)
- $\log_c a = \log_2 2 = 1$

$$f(n) \in \Theta(n^{\log_c a})$$

$$n \in \Theta(n^1)$$

Por el caso 2 del Teorema Maestro,

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

Teorema maestro - Búsqueda Binaria

Nuestra recurrencia es:

$$T(n) = 1T(\frac{n}{2}) + O(1)$$

- Tenemos que $\log_c a = \log_2 1 = 0$.
- Comparando f(n) con $n^{\log_c a} = n^0 = 1$ vemos que caemos en el segundo caso del teorema maestro, porque f(n) $\in \Theta(1)$.
- Entonces la complejidad es $\Theta(n^0 logn) = \Theta(logn)$

Temario

Máximo de una montaña

Ejercicio 1

Dado un arreglo montaña de longitud n, queremos encontrar al máximo. La complejidad del algoritmo que resuelva el problema debe ser $O(\log n)$

- Un arreglo de enteros *montaña* está compuesto por una secuencia estrictamente creciente seguida de una estrictamente decreciente
- Por ejemplo, para un arreglo [-1, 3, 8, 22, 30, 22, 8, 4, 2, 1], el máximo está en la posición 4 y vale 30
- Suponemos que hay al menos un elemento antes y después que el máximo, ya que las secuencias creciente y decreciente tienen al menos 2 elementos

Máximo de una montaña

A pensar

Dado A un arreglo montaña, pensemos los casos posibles...

- $A = [-1, 3, 8, 22, 30, 22, 8, 4, 2, 1]$
- $A = [10, 11, 12, -2, -100]$
- $A = [0, 48, 10, 8, -1]$
- $A = [-1, 48, 100, 100, 84, -10]$

Máximo de una montaña

Solución posible

Muy similar al algoritmo anterior. Acá considero rango [l, r].
Cuidado: Usar índices en vez de copiar los subarreglos.

Maximo(A, l, r) $\rightarrow$ int

```
if l = r then                          // Rango de un solo elemento
    return A[l]

m ← (l + r) / 2
// Si es creciente seguro el maximo está a la derecha de m
if A[m] < A[m + 1] then
    return Maximo(A, m+1, r);                                  ▷ T(n/2)
else
    return Maximo(A, l, m);                                    ▷ T(n/2)
```

Complejidad: O(log n). Misma justificación que en la búsqueda binaria.

Subsecuencia de suma máxima

Ejercicio 2

Dada una secuencia de n enteros, se desea encontrar el máximo valor que se puede obtener sumando elementos contiguos.

- Por ejemplo, para la secuencia [3, -1, 4, 8, -2, 2, -7, 5], este valor es 14, que se obtiene de la subsecuencia [3, -1, 4, 8]
- Si una secuencia tiene todos números negativos, se entiende que su subsecuencia de suma máxima es la vacía, por lo tanto el valor es 0
- Se desea hallar un algoritmo D&C que lo resuelva en $O(n \log n)$

Subsecuencia de suma máxima

A pensar

Dado un arreglo A, pensemos los casos posibles. . .

- $A = [3, -1, 4, 8, -2, 2, -7, 5]$

Subsecuencia de suma máxima

Solución posible

SumaSubsecuencia(A) $\rightarrow$ int

---

if $|A| = 1$ **then** // Rango de un solo elemento
  return max(0, A[0])

s1 $\leftarrow$ SumaSubsecuencia(A[0 .. $\frac{n}{2}$]) ▷ T($\frac{n}{2}$)
s2 $\leftarrow$ SumaSubsecuencia(A[$\frac{n}{2}$ .. n]) ▷ T($\frac{n}{2}$)
s3 $\leftarrow$ SumaAlMedio(A) ▷ O(n)

return max(s1, s2, s3)

SumaAlMedio(A) $\rightarrow$ int

---

s1 $\leftarrow$ SumaHaciaDerecha(A[$\frac{n}{2}$..n]) ▷ O(n)
s2 $\leftarrow$ SumaHaciaDerecha(reverso(A[0..$\frac{n}{2}$])) ▷ O(n)
return s1 + s2

Subsecuencia de suma máxima

Solución posible

SumaHaciaDerecha(A) $\rightarrow$ int

```
maxSuma ← 0                                                    ▷ O(1)
sumaAcumulada ← 0                                              ▷ O(1)

for i = 0 to n-1 do                                            ▷ O(n)
    sumaAcumulada += A[i]                                      ▷ O(1)
    maxSuma ← max(maxSuma, sumaAcumulada)                      ▷ O(1)

return maxSuma
```

- Recurrencia de **SumaSubsecuencia**: $T(n) = 2\ T(\frac{n}{2}) + O(n)$
- Observar que es la misma ecuación que MergeSort
- Podemos demostrar de igual forma que la complejidad es $O(n\ \log n)$

- **Bonus:** Se podía resolver en O(n) usando el algoritmo de Kadane

Matriz creciente

Ejercicio 3

Se tiene una matriz A de n x n números naturales, de manera que $A[i, j]$ representa al elemento en la fila i y columna j ($1 \leq i, j \leq n$). Se sabe que el acceso a un elemento cualquiera se realiza en tiempo $O(1)$. Se sabe también que todos los elementos de la matriz son distintos y que todas las filas y columnas de la matriz están ordenadas de forma creciente (es decir, $i < n \Rightarrow A[i, j] < A[i + 1, j]$ y $j < n \Rightarrow A[i, j] < A[i, j + 1]$).

1. Implementar, utilizando la técnica de dividir y conquistar, la función:

   está(in n: nat, in A: matriz(nat), in e: nat) $\rightarrow$ bool

   que decide si un elemento e dado aparece en alguna parte de la matriz. Se debe dar un algoritmo que tome tiempo estrictamente menor que $O(n^2)$. Notar que la entrada es de tamaño $O(n^2)$.

2. Calcular y justificar la complejidad del algoritmo propuesto. Para simplificar el cálculo, se puede suponer que n es potencia de dos.

Matriz creciente

- Dada M_1 de abajo, con $n = 4$, ¿Está 4?

1	4	35	157
5	19	118	334
9	64	464	1395
54	169	1295	5698

- Dada M_1 de abajo, con $n = 5$

2	14	70	318	3464
5	41	159	839	7269
53	239	596	1514	21901
151	1114	2969	8878	79094
412	4431	8971	35720	134696

  - ¿Está 500?
  - ¿Está 600?

Matriz creciente

A pensar

Pensemos los casos posibles. . .

Matriz creciente

Algunas ideas

- **Cuidado**: el tamaño de la entrada es $O(n^2)$
  - Si hacemos un algoritmo que recorra *todas* las posiciones no vamos a cumplir con la complejidad pedida
- Dividiendo la matriz en dos partes no vemos una forma clara de resolverlo.
  - No necesariamente tienen que ser siempre dos partes
- Suele ser conveniente prestar atencion a las características del problema.
- Notar en este caso que los valores están ordenados de una forma particular
- Puede resolverse sin D&C, lo que queda de ejercicio
  - **Pista:** Aprovechar que cada fila y columna están ordenadas y son estrictamente crecientes

Matriz creciente

Solución posible

esta?(n, A, e) $\rightarrow$ bool

```
return EstaM(A, e, 0, n, 0, n)
```

Esta(A, e, x_1, x_2, y_1, y_2) $\rightarrow$ bool

```
if (x1 + 1 = x2 and y1 + 1 = y2) then
    return e == A[x1][y1]
mx ← (x1 + x2)/2
my ← (y1 + y2)/2
if e ≤ A[mx][my] then
    return Esta(A, e, x1, mx, y1, my) or
           Esta(A, e, mx, x2, y1, my) or
           Esta(A, e, x1, mx, my, y2)
else
    return Esta(A, e, mx, x2, y1, my) or
           Esta(A, e, x1, mx, my, y2) or
           Esta(A, e, mx, x2, my, y2)
```

Matriz creciente

Complejidad

- Queremos calcular la complejidad en función del tamaño de la entrada. El problema es que el tamaño de la entrada es n^2
- Llamemos $m = n^2$ y calculemos la complejidad en función de m

$$T(m) = 3 * T(\frac{m}{4}) + O(1)$$

- O(1) $\subseteq$ O($m^{\log_4 3}$), caemos en el caso 1 del teorema maestro
- Entonces por el teorema, T(m) = $\Theta(m^{\log_4 3})$
- Como m = n^2, tenemos que T(n^2) = O($n^{2 \log_4 3}$) $\subseteq$ O($n^{1,60}$), que es estrictamente mejor que O(n^2)

Máxima suma en árbol

Enunciado

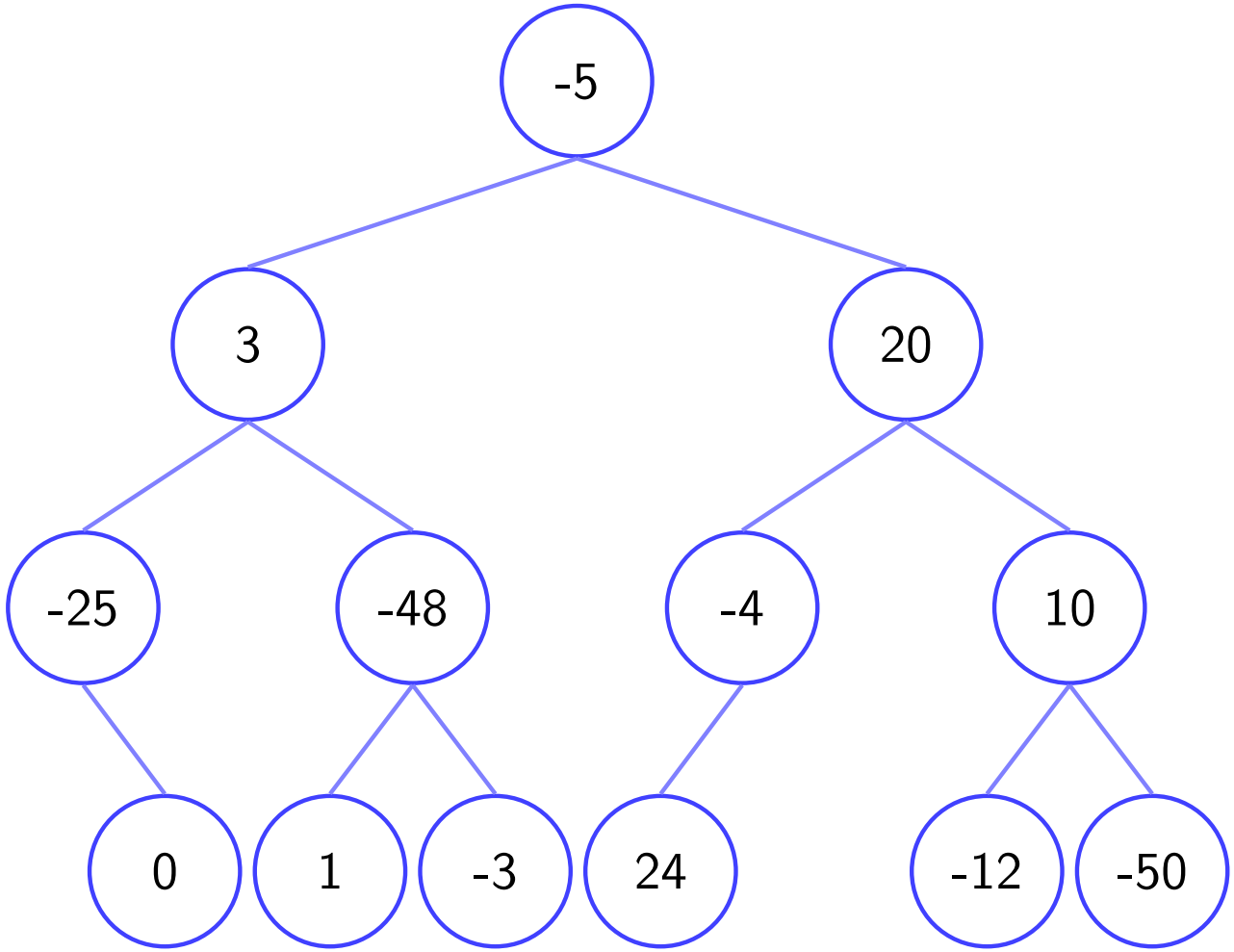
Dado un árbol binario de números enteros, se desea calcular la máxima suma de los nodos pertenecientes a un camino entre dos nodos cualesquiera del árbol

- Un camino entre dos nodos n_1 y n_2 está formado por todos los nodos que hay que atravesar en el árbol para llegar desde n_1 hasta n_2, incluyéndolos a ambos
- Un camino entre un nodo y sí mismo está formado únicamente por ese nodo
- Suponemos que el árbol está balanceado
- Dar un algoritmo $\text{MÁXIMASUMACAMINO}(a : \text{ab(int)}) \rightarrow \text{int}$ que resuelva el problema utilizando la técnica de *Divide & Conquer*

Máxima suma en árbol

Ejemplo

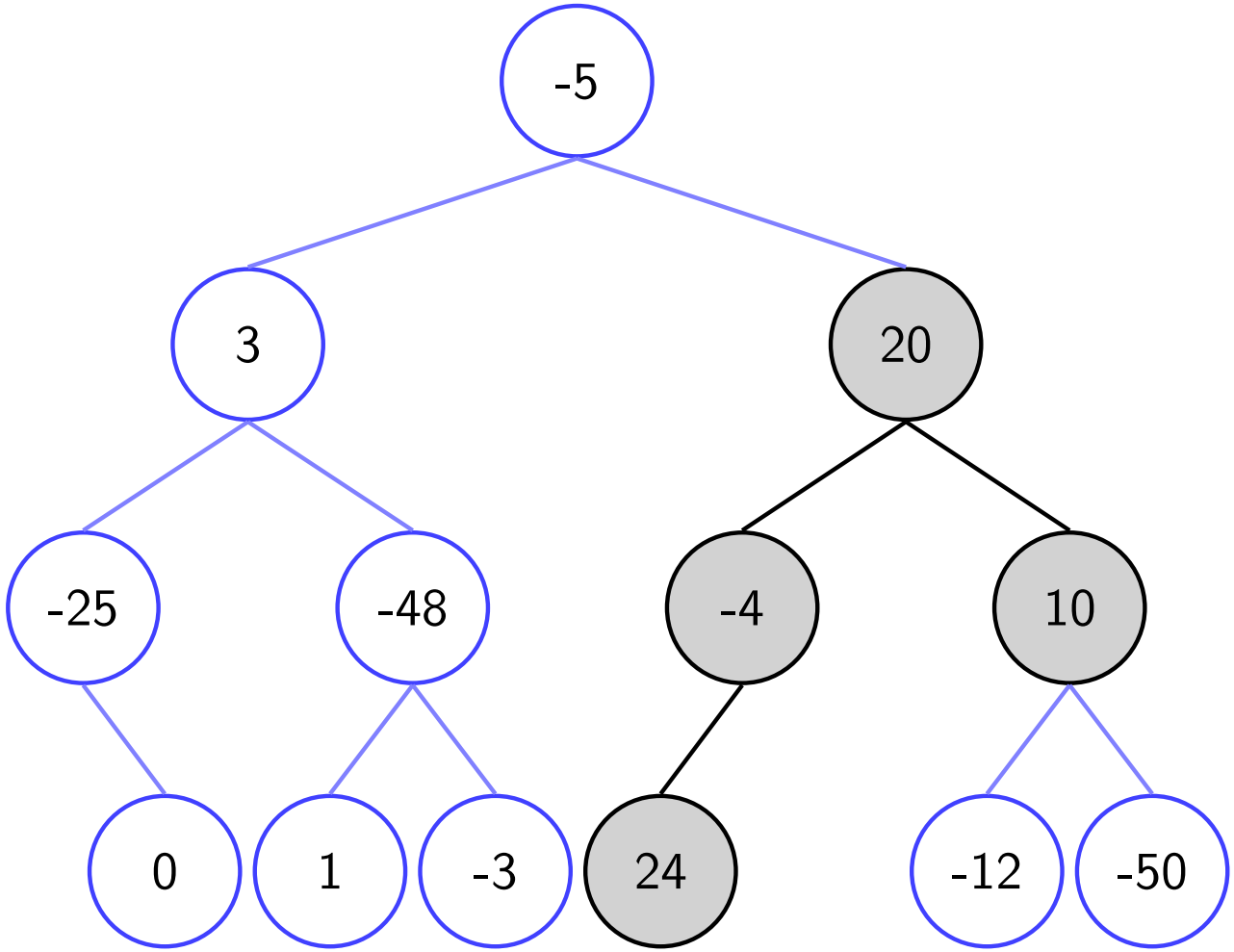
¿Cuál es el camino de máxima suma del árbol siguiente?



Máxima suma en árbol

Ejemplo

¿Cuál es el camino de máxima suma del árbol siguiente?



Suma total del camino$= 24 + (-4) + 20 + 10 = 50$

Máxima suma en árbol

Ejercicio a)

El algoritmo debe tener una complejidad temporal de peor caso igual o mejor que $O(n \log n)$ siendo n la cantidad de nodos del árbol

A pensar. . .

Máxima suma en árbol

Solución posible

Similar al problema de subsecuencia máxima, tenemos tres posibilidades: el máximo camino está en el subárbol izquierdo, en el subárbol derecho, o pasa por la raiz del árbol.

MáximaSumaCamino(A : ab(int)) $\rightarrow$ int

```
if nil?(A) then
    return 0                                                          ▷ O(1)

S1 ← MáximaSumaCamino( izq(A) )                                       ▷ T(n/2)
S2 ← MáximaSumaCamino( der(A) )                                       ▷ T(n/2)
S3 ← raiz(A) + MaxDesdeRaiz( izq(A) ) + MaxDesdeRaiz( der(A) )        ▷ O(n)

return max(S1, S2, S3)
```

Máxima suma en árbol

Solución posible

MaxDesdeRaiz es el camino más grande que empieza en la raiz. Podría ser un camino vacío.

MaxDesdeRaiz(A : ab(int)) $\rightarrow$ int

---

if nil?(A) **then**
  return 0 ▷ O(1)

C0 $\leftarrow$ 0 ▷ O(1)
C1 $\leftarrow$ raiz(A) ▷ O(1)
C2 $\leftarrow$ raiz(A) + MaxDesdeRaiz(izq(A)) ▷ T($\frac{n}{2}$)
C3 $\leftarrow$ raiz(A) + MaxDesdeRaiz(der(A)) ▷ T($\frac{n}{2}$)

return max(C0, C1, C2, C3)

Máxima suma en árbol

Complejidad

Teorema Maestro Si nuestra recurrencia es de la forma

$$T(n) = \begin{cases} aT\left(\frac{n}{c}\right) + f(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(n^{\log_c a}) & \text{Si } \exists \varepsilon > 0 \text{ tal que } f(n) \in O(n^{\log_c a - \varepsilon}) \\ \Theta(n^{\log_c a} \log n) & \text{Si } f(n) \in \Theta(n^{\log_c a}) \\ \Theta(f(n)) & \text{Si } \exists \varepsilon > 0 \text{ tal que } f(n) \in \Omega(n^{\log_c a + \varepsilon}) \text{ y} \\ & \exists \delta < 1, \exists n_0 > 0 \text{ tal que } \forall n \geq n_0 \text{ se cumple: } a.f\left(\frac{n}{c}\right) \leq \delta f(n) \end{cases}$$

- **MaxDesdeRaiz**: Tenemos que $T'(n) = 2\ T'\left(\frac{n}{2}\right) + O(1)$.
  Por el primer caso del teorema maestro la complejidad es $\Theta(n)$
- **MáximaSumaCamino**: su ecuación es $T(n) = 2\ T\left(\frac{n}{2}\right) + O(n)$
  Por el segundo caso del teorema su complejidad es $\Theta(n\ \log n)$

Máxima suma en árbol v2

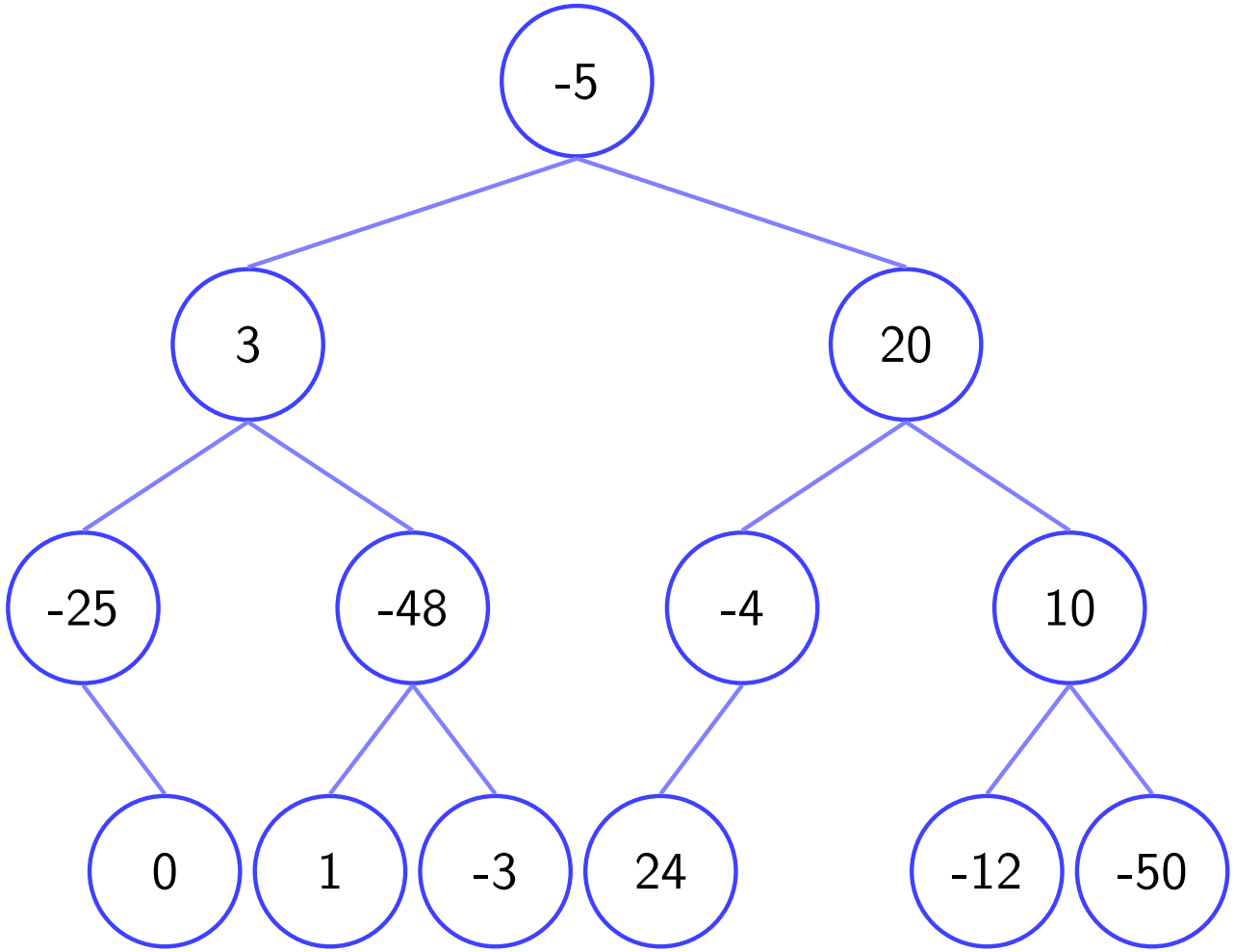
Ejercicio b)

El algoritmo debe tener una complejidad temporal de peor caso igual o mejor que $O(n)$ siendo n la cantidad de nodos del árbol.

A pensar. . .

Máxima suma en árbol v2

Ejemplo



Máxima suma en árbol v2

Solución posible

MaxCamino(A : ab(int)) $\rightarrow \langle sumaCamino : int, sumaCaminoRaiz : int \rangle$

---

if nil?(A) **then**
    return $\langle 0, 0 \rangle$ ▷ O(1)
dataIzq $\leftarrow$ MaxCamino(izq(A)) ▷ T($\frac{n}{2}$)
dataDer $\leftarrow$ MaxCamino(der(A)) ▷ T($\frac{n}{2}$)
S1 $\leftarrow$ dataIzq.sumaCamino ▷ O(1)
S2 $\leftarrow$ dataDer.sumaCamino ▷ O(1)
S3 $\leftarrow$ raiz(A) + dataIzq.sumaCaminoRaiz + dataDer.sumaCaminoRaiz ▷ O(1)
cam $\leftarrow$ max(S1, S2, S3) ▷ O(1)
desdeR $\leftarrow$ max(0, raiz(A),
    raiz(A) + dataIzq.sumaCaminoRaiz, raiz(A) + dataDer.sumaCaminoRaiz) ▷ O(1)

return $\langle cam, desdeR \rangle$

MáximaSumaCamino(A : ab(int)) $\rightarrow$ int

---

return MaxCamino(A).camino

Máxima suma en árbol v2

Complejidad

La complejidad de **MáximaSumaCamino** está dada por la de **MaxCamino**, cuya relación de recurrencia es:

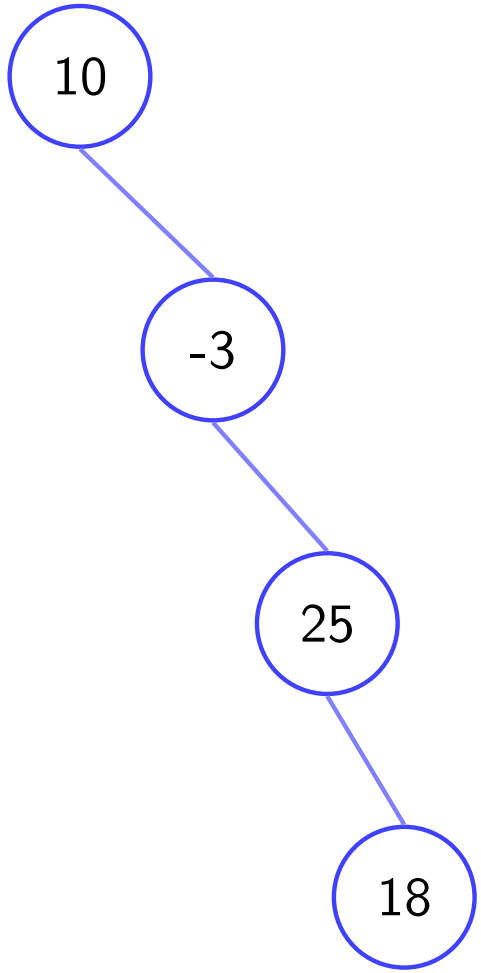
$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

Por el primer caso del teorema maestro, la complejidad es $\Theta(n)$

Máxima suma en árbol v3

Ejercicio c)

Supongamos que el árbol **NO** está balanceado. El algoritmo debe tener una complejidad temporal de peor caso igual o mejor que $O(n)$ siendo n la cantidad de nodos del árbol.
Por ejemplo, podríamos tener que resolver un árbol degenerado.



A pensar. . .

Máxima suma en árbol v3

Solución posible

- Podemos usar el mismo algoritmo que en b), pero **NO** podemos usar teorema maestro para justificar la complejidad, pues los llamados recursivos NO necesariamente son $T(\frac{n}{2})$.

- Para justificar que sigue siendo O(n) podemos hacer el cálculo de costo usando el árbol de la recursión. Cada uno de los n nodos son visitados **por única vez** y tenemos O(1) costo en cada nodo, por lo que el costo final termina siendo O(n).

Temario

Referencias y links útiles

- T. H. Cormen, C. E. Leiserson, R.L Rivest, and C. Stein. **"Introduction to Algorithms"**. 3rd edition.

- Más información y otras formas de resolver recurrencias. `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf`

- Clase del MIT. Aplicaciones: Convex Hull y Calcular Mediana en O(n) `https://www.youtube.com/watch?v=EzeYI7p9MjU`