Programación III

Ricardo Wehbe

UADE

30 de agosto de 2021

Programa

Las notaciones $\mathcal{O}$ y Θ

Las notaciones $\mathcal{O}$ y Θ

- Si $f : \mathbb{N} \to \mathbb{R}^+$, entonces $\mathcal{O}(f(n))$ (el "orden" de $f(n)$) es el conjunto de funciones $g : \mathbb{N} \to \mathbb{R}^+$ para las cuales existen constantes c_2 y n_0 tales que

$$0 \leq g(n) \leq c_2 f(n) \text{ para todo } n \geq n_0$$

Las notaciones $\mathcal{O}$ y Θ

- Si $f : \mathbb{N} \to \mathbb{R}^+$, entonces $\mathcal{O}(f(n))$ (el "orden" de $f(n)$) es el conjunto de funciones $g : \mathbb{N} \to \mathbb{R}^+$ para las cuales existen constantes c_2 y n_0 tales que

$$0 \leq g(n) \leq c_2 f(n) \ \text{ para todo } n \geq n_0$$

- Si $f : \mathbb{N} \to \mathbb{R}^+$, entonces $\Theta(f(n))$ (el "orden exacto" de $f(n)$) es el conjunto de funciones $g : \mathbb{N} \to \mathbb{R}^+$ para las cuales existen constantes c_1, c_2 y n_0 tales que

$$0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n) \ \text{ para todo } n \geq n_0$$

Un ejemplo con la notación $\mathcal{O}$

Un ejemplo con la notación $\mathcal{O}$

- Queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \mathcal{O}(n^3)$.

Un ejemplo con la notación $\mathcal{O}$

- Queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \mathcal{O}(n^3)$.
- Para ello, debemos encontrar las constantes c_2 y n_0 tales que

$$3n^3 + 5n^2 + 7 \leq c_2 n^3 \text{ para todo } n \geq n_0$$

Un ejemplo con la notación $\mathcal{O}$

- Queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \mathcal{O}(n^3)$.
- Para ello, debemos encontrar las constantes c_2 y n_0 tales que

$$3n^3 + 5n^2 + 7 \leq c_2 n^3 \text{ para todo } n \geq n_0$$

- Por ejemplo, $c_2 = 5$ y $n_0 = 3$.

Un ejemplo con la notación $\mathcal{O}$

- Queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \mathcal{O}(n^3)$.
- Para ello, debemos encontrar las constantes c_2 y n_0 tales que

$$3n^3 + 5n^2 + 7 \leq c_2 n^3 \text{ para todo } n \geq n_0$$

- Por ejemplo, $c_2 = 5$ y $n_0 = 3$.
- Observe que también tenemos $3n^3 + 5n^2 + 7 \in \mathcal{O}(n^4)$. Para mostrar esto, basta tomar las mismas constantes.

Un ejemplo con la notación Θ

Un ejemplo con la notación Θ

- Ahora queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.

Un ejemplo con la notación Θ

- Ahora queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.
- Para ello, debemos encontrar las constantes c_1, c_2 y n_0 tales que

$$c_1 n^3 \leq 3n^3 + 5n^2 + 7 \leq c_2 n^3 \text{ para todo } n \geq n_0$$

Un ejemplo con la notación Θ

- Ahora queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.
- Para ello, debemos encontrar las constantes c_1, c_2 y n_0 tales que

$$c_1 n^3 \leq 3n^3 + 5n^2 + 7 \leq c_2 n^3 \text{ para todo } n \geq n_0$$

- Por ejemplo, $c_1 = 1$, $c_2 = 5$ y $n_0 = 3$.

Un ejemplo con la notación Θ

- Ahora queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.
- Para ello, debemos encontrar las constantes c_1, c_2 y n_0 tales que

$$c_1 n^3 \leq 3n^3 + 5n^2 + 7 \leq c_2 n^3 \text{ para todo } n \geq n_0$$

- Por ejemplo, $c_1 = 1$, $c_2 = 5$ y $n_0 = 3$.
- Observe que ahora $3n^3 + 5n^2 + 7 \notin \Theta(n^4)$.

Un ejemplo con la notación Θ

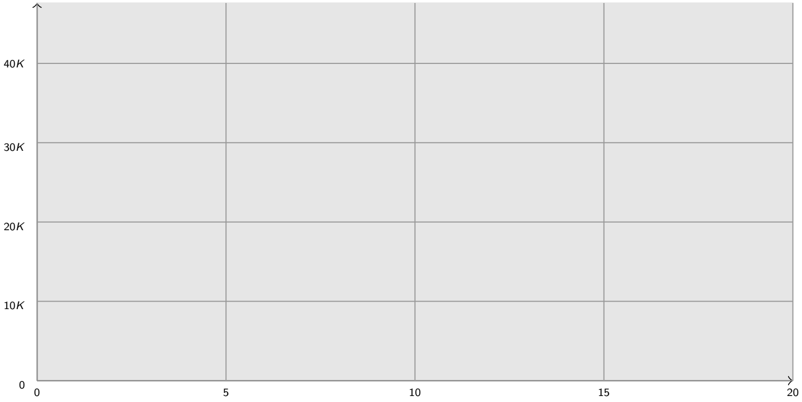
- Ahora queremos mostrar que $f(n) = 3n^3 + 5n^2 + 7 \in \Theta(n^3)$.
- Para ello, debemos encontrar las constantes c_1, c_2 y n_0 tales que

$$c_1 n^3 \leq 3n^3 + 5n^2 + 7 \leq c_2 n^3 \text{ para todo } n \geq n_0$$

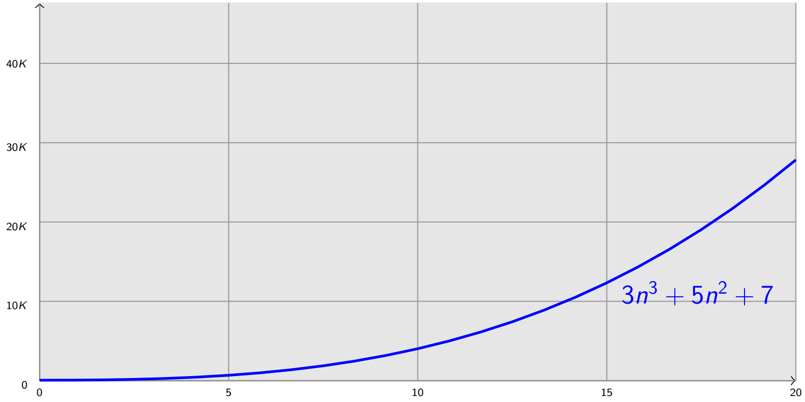
- Por ejemplo, $c_1 = 1$, $c_2 = 5$ y $n_0 = 3$.
- Observe que ahora $3n^3 + 5n^2 + 7 \notin \Theta(n^4)$.
- Esto se debe a que no podemos encontrar ninguna constante c_1 y n_0 tales que $c_1 n^4 \leq 3n^3 + 5n^2 + 7$ para todo $n \geq n_0$; por pequeña que sea la constante c_1, siempre habrá algún valor de n a partir del cual $c_1 n^4$ superará a $3n^3 + 5n^2 + 7$.

Un ejemplo. Interpretación gráfica de Θ

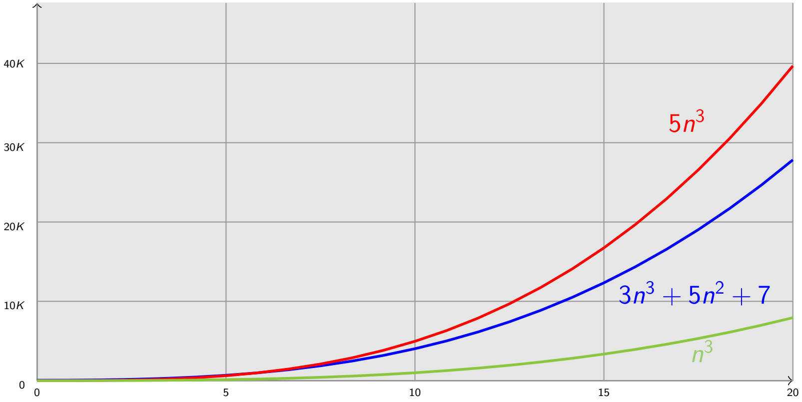
Un ejemplo. Interpretación gráfica de Θ



Un ejemplo. Interpretación gráfica de Θ



Un ejemplo. Interpretación gráfica de Θ



Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n \log n)$	casi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	casi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	casi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(2^n)$	exponencial
$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	casi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n!)$	factorial
$\mathcal{O}(2^n)$	exponencial
$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	casi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$\mathcal{O}(n^n)$	
$\mathcal{O}(n!)$	factorial
$\mathcal{O}(2^n)$	exponencial
$\mathcal{O}(n^3)$	cúbica
$\mathcal{O}(n^2)$	cuadrática
$\mathcal{O}(n \log n)$	casi lineal
$\mathcal{O}(n)$	lineal
$\mathcal{O}(\log n)$	logarítmica
$\mathcal{O}(1)$	constante

Una jerarquía (parcial) de casos de complejidad

$$
\begin{array}{rll}
 & \mathcal{O}(n^n) & \\
 & \mathcal{O}(n!) & \text{factorial} \\
 & \mathcal{O}(2^n) & \text{exponencial} \\
\text{tratable} & \left\{ \begin{array}{l} \mathcal{O}(n^3) \\ \mathcal{O}(n^2) \\ \mathcal{O}(n \log n) \\ \mathcal{O}(n) \\ \mathcal{O}(\log n) \\ \mathcal{O}(1) \end{array} \right. & \begin{array}{l} \text{cúbica} \\ \text{cuadrática} \\ \text{casi lineal} \\ \text{lineal} \\ \text{logarítmica} \\ \text{constante} \end{array}
\end{array}
$$

Una jerarquía (parcial) de casos de complejidad

intratable	$\mathcal{O}(n^n)$	
	$\mathcal{O}(n!)$	factorial
	$\mathcal{O}(2^n)$	exponencial
tratable	$\mathcal{O}(n^3)$	cúbica
	$\mathcal{O}(n^2)$	cuadrática
	$\mathcal{O}(n \log n)$	casi lineal
	$\mathcal{O}(n)$	lineal
	$\mathcal{O}(\log n)$	logarítmica
	$\mathcal{O}(1)$	constante

Resolución de recurrencias: el caso de substracción

Resolución de recurrencias: el caso de substracción

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

Resolución de recurrencias: el caso de substracción

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} \Theta\left(n^k\right) & \text{si } a < 1 \\ \Theta\left(n^{k+1}\right) & \text{si } a = 1 \\ \Theta\left(n^k a^{n/b}\right) & \text{si } a > 1 \end{cases}$$

Resolución de recurrencias: el caso de división

Resolución de recurrencias: el caso de división

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n/b) + p(n) & \text{si } n \geq b \end{cases}$$

Resolución de recurrencias: el caso de división

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n/b) + p(n) & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} \Theta\left(n^k\right) & \text{si } a < b^k \\ \Theta\left(n^k \log n\right) & \text{si } a = b^k \\ \Theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

(...) de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre.

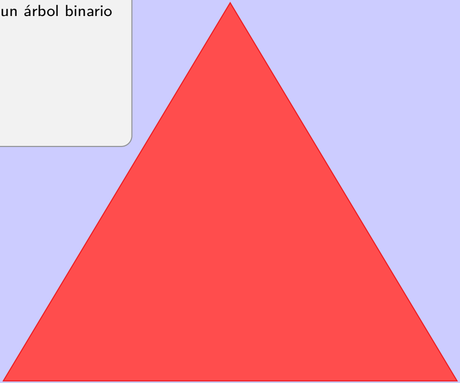
((...) dividir los problemas bajo examen en tantas partes como sea posible y necesario para resolverlos mejor.)

René Descartes, *Discours de la méthode*

Souvenir: ABB

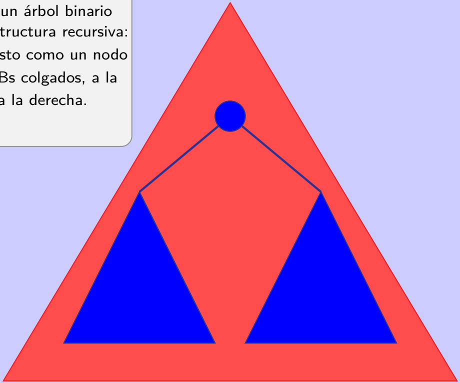
Souvenir: ABB

Un ABB es un árbol binario



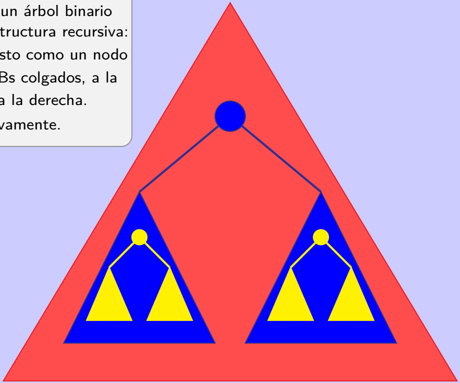
Souvenir: ABB

Un ABB es un árbol binario que tiene estructura recursiva: puede ser visto como un nodo con dos ABBs colgados, a la izquierda y a la derecha.



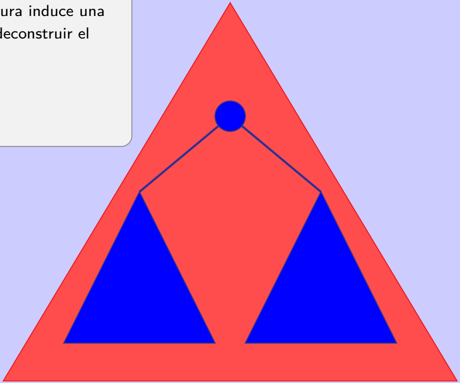
Souvenir: ABB

Un ABB es un árbol binario que tiene estructura recursiva: puede ser visto como un nodo con dos ABBs colgados, a la izquierda y a la derecha.
Y así sucesivamente.



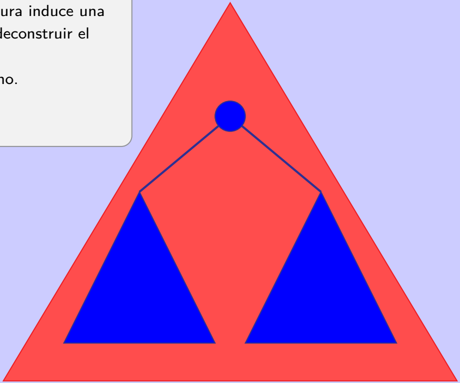
Souvenir: ABB

Esta estructura induce una manera de deconstruir el ABB.



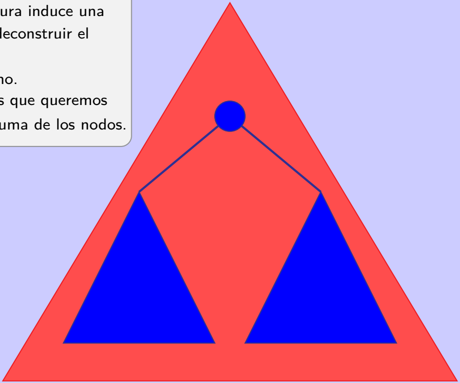
Souvenir: ABB

Esta estructura induce una manera de deconstruir el ABB.
Veamos cómo.



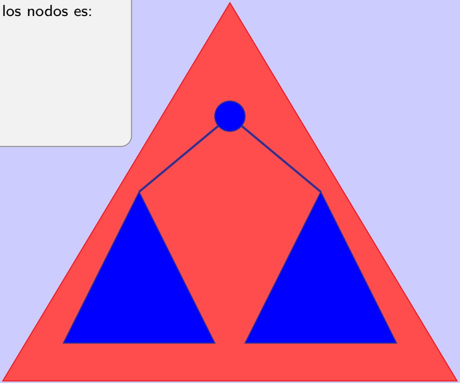
Souvenir: ABB

Esta estructura induce una manera de deconstruir el ABB.
Veamos cómo.
Supongamos que queremos calcular la suma de los nodos.

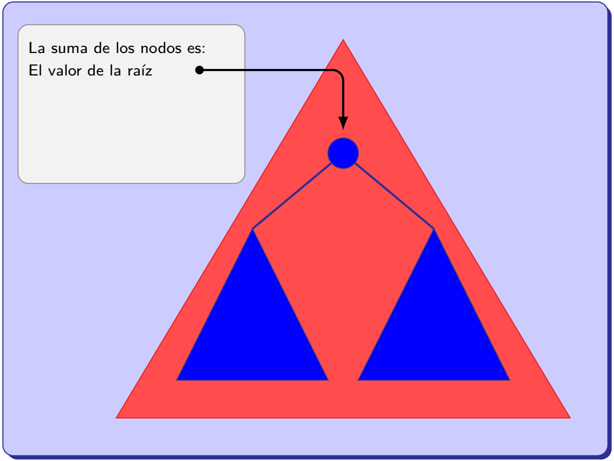


Souvenir: ABB

La suma de los nodos es:

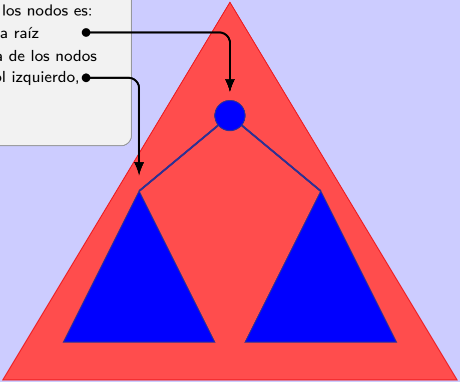


Souvenir: ABB

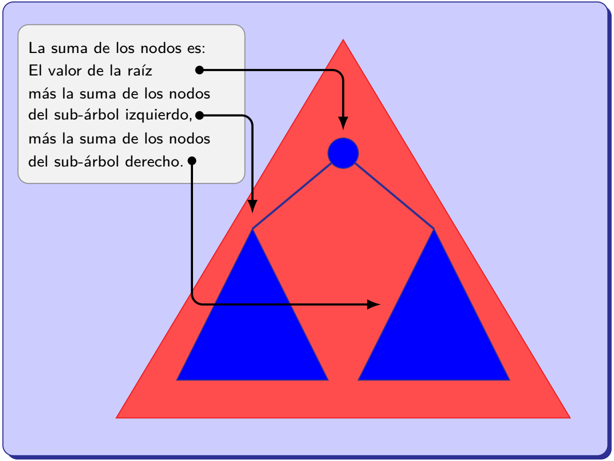


Souvenir: ABB

La suma de los nodos es:
El valor de la raíz
más la suma de los nodos
del sub-árbol izquierdo,



Souvenir: ABB

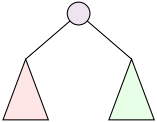


Un primer algoritmo *divide & conquer*

```
Algoritmo SumNodes (BST T)
    if (T.EmptyTree())          //base case
        return 0
    else
        return T.Value() + SumNodes(T.LeftSon() + SumNodes(T.RightSon()
```

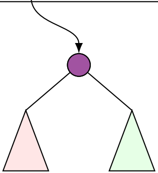
Un primer algoritmo *divide & conquer*

```
Algoritmo SumNodes (BST T)
    if (T.EmptyTree())        //base case
        return 0
    else
        return T.Value() + SumNodes(T.LeftSon() + SumNodes(T.RightSon()
```



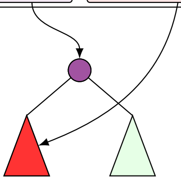
Un primer algoritmo *divide & conquer*

```
Algoritmo SumNodes (BST T)
    if (T.EmptyTree())          //base case
        return 0
    else
        return T.Value() + SumNodes(T.LeftSon() + SumNodes(T.RightSon())
```



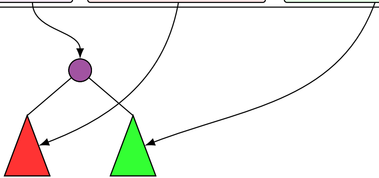
Un primer algoritmo *divide & conquer*

```
Algoritmo SumNodes (BST T)
    if (T.EmptyTree())        //base case
        return 0
    else
        return T.Value() + SumNodes(T.LeftSon() + SumNodes(T.RightSon()
```



Un primer algoritmo *divide & conquer*

```
Algoritmo SumNodes (BST T)
    if (T.EmptyTree())          //base case
        return 0
    else
        return T.Value() + SumNodes(T.LeftSon() + SumNodes(T.RightSon()
```



La técnica *divide & conquer*

La técnica *divide & conquer*

- Es una técnica que consiste en dividir un problema "grande" en una serie de problemas "pequeños" de resolución más simple y luego combinar las soluciones de los problemas "pequeños" para obtener una solución del problema "grande."

La técnica *divide & conquer*

- Es una técnica que consiste en dividir un problema "grande" en una serie de problemas "pequeños" de resolución más simple y luego combinar las soluciones de los problemas "pequeños" para obtener una solución del problema "grande."
- La técnica es recursiva: los problemas "pequeños" se subdividen a su vez hasta llegar a un problema "mínimo" de resolución trivial (el caso base de recurrencia).

La técnica *divide & conquer*

- Es una técnica que consiste en dividir un problema “grande” en una serie de problemas “pequeños” de resolución más simple y luego combinar las soluciones de los problemas “pequeños” para obtener una solución del problema “grande.”
- La técnica es recursiva: los problemas “pequeños” se subdividen a su vez hasta llegar a un problema “mínimo” de resolución trivial (el caso base de recurrencia).
- Se efectúan entonces los siguientes pasos: 1. dividir, 2. conquistar (resolver los problemas mínimos) y 3. combinar las soluciones obtenidas.

Un algoritmo *divide & conquer* genérico

Un algoritmo *divide & conquer* genérico

```
Algoritmo D&C (x)
    if isSmall(x) {
        return TrivialSolution (x)
    else
        ⟨x_1, ..., x_n⟩ ← decompose (x)
        for (i = 0; n < n; i++) {
            y_i ← D&C(x_i)
        }
        return combine(y_1, ..., y_n)
    }
```

Búsqueda binaria. Versión simplificada

Búsqueda binaria. Versión simplificada

- Partimos de un vector $u[0..n\text{-}1]$ de números ordenados ascendentemente y un número x. El programa debe determinar si x está en el vector o no.

Búsqueda binaria. Versión simplificada

- Partimos de un vector $u[0..n\text{-}1]$ de números ordenados ascendentemente y un número x. El programa debe determinar si x está en el vector o no.
- Por supuesto se puede resolver por fuerza bruta: se recorre secuencialmente el vector y se compara cada elemento $u[i]$ con x. Esto tiene complejidad $\mathcal{O}(n)$.

Búsqueda binaria. Versión simplificada

- Partimos de un vector $u[0..n\text{-}1]$ de números ordenados ascendentemente y un número x. El programa debe determinar si x está en el vector o no.
- Por supuesto se puede resolver por fuerza bruta: se recorre secuencialmente el vector y se compara cada elemento $u[i]$ con x. Esto tiene complejidad $\mathcal{O}(n)$.
- Una solución más eficiente es la *búsqueda binaria*: comparamos x con el elemento central $u[k]$. Si x es mayor, limitamos nuestra búsqueda a $u[k+1..n]$ y, si es menor, a $u[1..k-1]$

Búsqueda binaria. Versión simplificada

- Partimos de un vector $u[0..n\text{-}1]$ de números ordenados ascendentemente y un número x. El programa debe determinar si x está en el vector o no.
- Por supuesto se puede resolver por fuerza bruta: se recorre secuencialmente el vector y se compara cada elemento $u[i]$ con x. Esto tiene complejidad $\mathcal{O}(n)$.
- Una solución más eficiente es la *búsqueda binaria*: comparamos x con el elemento central $u[k]$. Si x es mayor, limitamos nuestra búsqueda a $u[k+1..n]$ y, si es menor, a $u[1..k-1]$
- Los casos base son: si el índice inicial queda a la derecha del final, entonces no hemos encontrado el número y debe retornarse `false`; si el vector tiene un solo elemento, es trivial determinar si x pertenece al vector o no al vector. Y si $u[k]=x$, la respuesta es obviamente `true`.

El algoritmo de búsqueda binaria. Versión simplificada

```
Algoritmo BinSearch (int [] u[0..n-1], int ini, fin, x)
   if (ini > fin) {                    // primer caso base
     return false;
   } else if (ini == fin) {            // segundo caso base
     return (u[ini] == x);
   } else {                            // quedan dos o más elementos
     mid = (ini + fin) / 2;
     if x == u[mid] {                  // tercer caso base
       return true;
     } else if (x > u[mid]) {          // mitad izquierda
       return BinSearch(u, mid+1, fin, x)
     } else {                          // mitad derecha
       return BinSearch(u, ini, mid-1, x)
     }
   }
```

Complejidad de la búsqueda binaria

Complejidad de la búsqueda binaria

- Observemos que se trata de un algoritmo recursivo con división. Tiene la forma

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T(n/2) + c & \text{si } n > 1 \end{cases}$$

Complejidad de la búsqueda binaria

- Observemos que se trata de un algoritmo recursivo con división. Tiene la forma

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T(n/2) + c & \text{si } n > 1 \end{cases}$$

- Tenemos entonces $a = 1$, $b = 2$ y $k = 0$.

Complejidad de la búsqueda binaria

- Observemos que se trata de un algoritmo recursivo con división. Tiene la forma

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T(n/2) + c & \text{si } n > 1 \end{cases}$$

- Tenemos entonces $a = 1$, $b = 2$ y $k = 0$.
- Por lo tanto, $a = b^k$ (ya que $1 = 2^0$).

Complejidad de la búsqueda binaria

- Observemos que se trata de un algoritmo recursivo con división. Tiene la forma

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T(n/2) + c & \text{si } n > 1 \end{cases}$$

- Tenemos entonces $a = 1$, $b = 2$ y $k = 0$.
- Por lo tanto, $a = b^k$ (ya que $1 = 2^0$).
- Entonces $T(n) \in \Theta(n^k \log n) = \Theta(\log n)$.

Cálculo de potencias de la forma $a^{(2^k)}$ con $k > 0$

```
1.  int Algoritmo Potencia (int a, n)
2.     if (n == 2) {                                   // Caso base
3.        return a * a;
4.     } else {
5.        return Potencia(a, n/2)*Potencia(a, n/2)
6.     }
```

Cálculo de potencias de la forma $a^{(2^k)}$ con $k > 0$

```
int Algoritmo Potencia (int a, n)
    if (n == 2) {                                   // Caso base
        return a * a;
    } else {
        return Potencia(a, n/2)*Potencia(a, n/2)
    }
```

- Aquí tenemos $a = 2$, $b = 2$ y $k = 0$.

Cálculo de potencias de la forma $a^{(2^k)}$ con $k > 0$

```
int Algoritmo Potencia (int a, n)
    if (n == 2) {                                   // Caso base
        return a * a;
    } else {
        return Potencia(a, n/2)*Potencia(a, n/2)
    }
```

- Aquí tenemos $a = 2$, $b = 2$ y $k = 0$.

Cálculo de potencias de la forma $a^{(2^k)}$ con $k > 0$

```
int Algoritmo Potencia (int a, n)
    if (n == 2) {                                   // Caso base
        return a * a;
    } else {
        return Potencia(a, n/2)*Potencia(a, n/2)
    }
```

- Aquí tenemos $a = 2$, $b = 2$ y $k = 0$.
- Por lo tanto, estamos en el caso $a > b^k$, ya que $2 > 2^0$.

Cálculo de potencias de la forma $a^{(2^k)}$ con $k > 0$

```
int Algoritmo Potencia (int a, n)
    if (n == 2) {                                  // Caso base
        return a * a;
    } else {
        return Potencia(a, n/2)*Potencia(a, n/2)
    }
```

- Aquí tenemos $a = 2$, $b = 2$ y $k = 0$.
- Por lo tanto, estamos en el caso $a > b^k$, ya que $2 > 2^0$.
- Entonces $Potencia(a, n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Cálculo de potencia, mejorado

Considere la siguiente versión del programa:

```
int Algoritmo Potencia_2 (int a, n)
    if (n == 2) {                        // Base case
        return a * a;
    } else {
        int quad = Potencia_2(a, n/2);
        return quad*quad;
    }
```

Cálculo de potencia, mejorado

Considere la siguiente versión del programa:

```
1.  int Algoritmo Potencia_2 (int a, n)
2.     if (n == 2) {                          // Base case
3.          return a * a;
4.      } else {
5.          int quad = Potencia_2(a, n/2);
6.          return quad*quad;
7.      }
```

Aquí tenemos *Potencia_2*$(a, n) \in \Theta(\log n)$. ¿Por qué?

Algunos métodos de ordenamiento ineficientes

Algunos métodos de ordenamiento ineficientes

- Ordenamiento por burbujeo: consiste en encontrar en la secuencia el mayor elemento y colocarlo en la última posición; se busca luego el mayor de los elementos restantes y se lo coloca en la penúltima posición, y así sucesivamente.

Algunos métodos de ordenamiento ineficientes

- Ordenamiento por burbujeo: consiste en encontrar en la secuencia el mayor elemento y colocarlo en la última posición; se busca luego el mayor de los elementos restantes y se lo coloca en la penúltima posición, y así sucesivamente.
- Ordenamiento por inserción: en cada paso i se coloca el elemento de la posición i en su posición correspondiente entre las primeras i posiciones.

Algunos métodos de ordenamiento ineficientes

- Ordenamiento por burbujeo: consiste en encontrar en la secuencia el mayor elemento y colocarlo en la última posición; se busca luego el mayor de los elementos restantes y se lo coloca en la penúltima posición, y así sucesivamente.
- Ordenamiento por inserción: en cada paso i se coloca el elemento de la posición i en su posición correspondiente entre las primeras i posiciones.
- En el primer caso, al final del paso i, las últimas i posiciones están ordenadas; en el segundo, al final del paso i, las primeras i posiciones están ordenadas.

Ordenamiento por burbujeo

```
Algoritmo Bubble (int [] u[0..n-1])
    for (i = n-1; i > 0; i--) {
        for (j = 0; j < i; j++) {
            if (u[i] < u[j]) {
                int aux = u[i]
                u[i] = u[j]
                u[j] = aux
            }
        }
    }
```

Ordenamiento por burbujeo

```
Algoritmo Bubble (int [] u[0..n-1])
    for (i = n-1; i > 0; i--) {
        for (j = 0; j < i; j++) {
            if (u[i] < u[j]) {
                int aux = u[i]
                u[i] = u[j]
                u[j] = aux
            }
        }
    }
```

$\mathcal{O}(n)$ $\mathcal{O}(n)$

Ordenamiento por burbujeo

```
Algoritmo Bubble (int [] u[0..n-1])
    for (i = n-1; i > 0; i--) {
        for (j = 0; j < i; j++) {
            if (u[i] < u[j]) {
                int aux = u[i]
                u[i] = u[j]
                u[j] = aux
            }
        }
    }
```

$\mathcal{O}(n)$ $\mathcal{O}(n)$

$$\mathcal{O}(n^2)$$

Ordenamiento por inserción

```
Algoritmo Insertion (int [] u[0..n-1])
    for (i = 2; i < n; i++) {
        for (j = i; j > 0, j--) {
            if (u[j] < u[j - 1]) {
                int aux = u[j - 1]
                u[j - 1] = u[j]
                u[j] = aux
            }
        }
    }
```

Ordenamiento por inserción

```
Algoritmo Insertion (int [] u[0..n-1])
    for (i = 2; i < n; i++) {
        for (j = i; j > 0, j--) {
            if (u[j] < u[j-1]) {
                int aux = u[j-1]
                u[j-1] = u[j]
                u[j] = aux
            }
        }
    }
```

$\mathcal{O}(n)$ $\mathcal{O}(n)$

Ordenamiento por inserción

```
Algoritmo Insertion (int [] u[0..n-1])
    for (i = 2; i < n; i++) {
        for (j = i; j > 0, j--) {
            if (u[j] < u[j-1]) {
                int aux = u[j-1]
                u[j-1] = u[j]
                u[j] = aux
            }
        }
    }
```

$\mathcal{O}(n)$ $\mathcal{O}(n)$

$$\mathcal{O}(n^2)$$

Métodos de ordenamiento *divide & conquer*

Métodos de ordenamiento *divide & conquer*

- Dos métodos de ordenamiento *divide & conquer*: *MergeSort* y *QuickSort*.

Métodos de ordenamiento *divide & conquer*

- Dos métodos de ordenamiento *divide & conquer*: *MergeSort* y *QuickSort*.
- *MergeSort* divide la secuencia de entrada en dos mitades, ordena cada una de ellas y luego "mezcla" las mitades ordenadas en una nueva secuencia ordenada. El proceso se aplica recursivamente sobre cada mitad hasta llegar a un caso trivial. Los casos triviales son los vectores de un elemento.

Métodos de ordenamiento *divide & conquer*

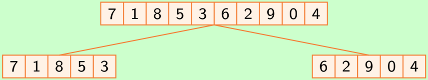
- Dos métodos de ordenamiento *divide & conquer*: *MergeSort* y *QuickSort*.
- *MergeSort* divide la secuencia de entrada en dos mitades, ordena cada una de ellas y luego "mezcla" las mitades ordenadas en una nueva secuencia ordenada. El proceso se aplica recursivamente sobre cada mitad hasta llegar a un caso trivial. Los casos triviales son los vectores de un elemento.
- En el *QuickSort* se elige un elemento (el "pivot"), a partir del cual se divide el vector en dos partes: los elementos mayores al pivot pasan a la derecha del pivot y los menores a su izquierda. Al final de este proceso, el pivot está en su posición definitiva y el proceso continúa recursivamente sobre cada una de las mitades hasta llegar a un caso trivial. El caso trivial es un vector de un elemento.

Ejemplo de funcionamiento de *MergeSort*

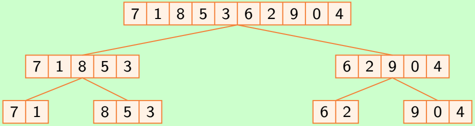
Ejemplo de funcionamiento de *MergeSort*

7	1	8	5	3	6	2	9	0	4

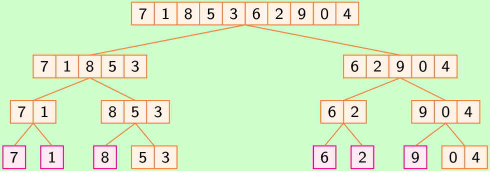
Ejemplo de funcionamiento de *MergeSort*



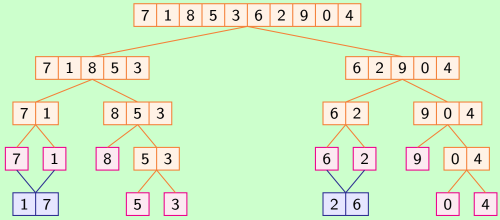
Ejemplo de funcionamiento de *MergeSort*



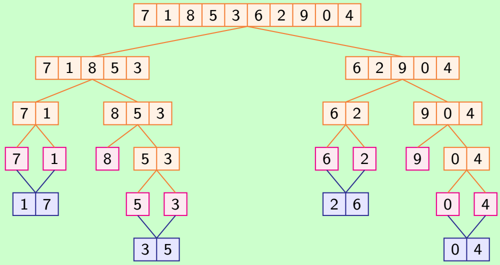
Ejemplo de funcionamiento de *MergeSort*



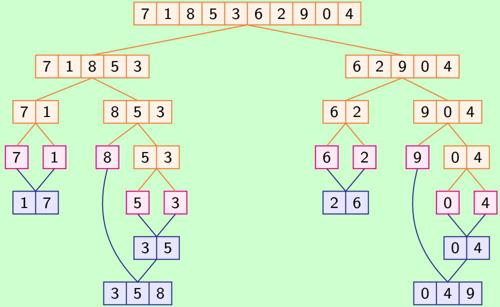
Ejemplo de funcionamiento de *MergeSort*



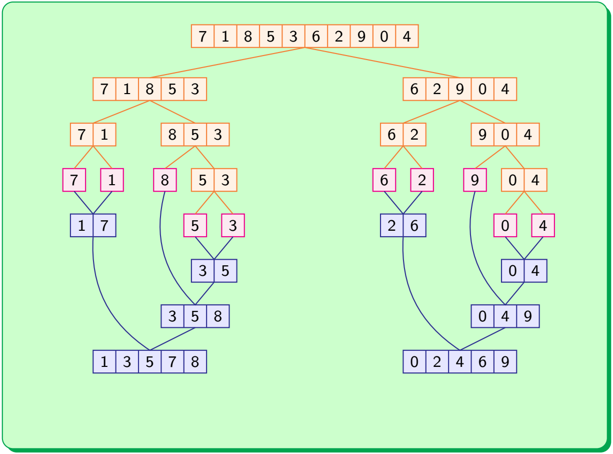
Ejemplo de funcionamiento de *MergeSort*



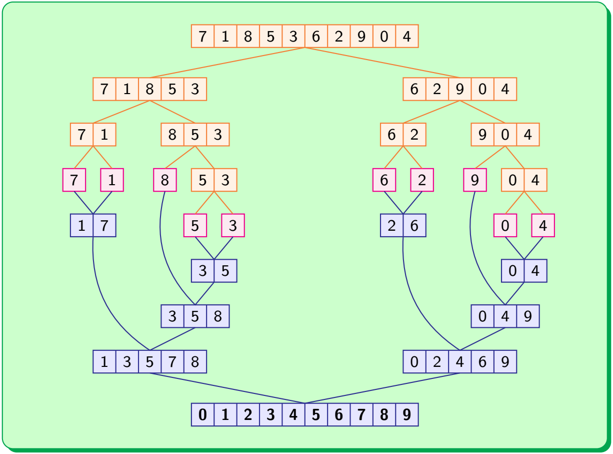
Ejemplo de funcionamiento de *MergeSort*



Ejemplo de funcionamiento de *MergeSort*



Ejemplo de funcionamiento de *MergeSort*



El algoritmo *MergeSort*

El algoritmo *MergeSort*

```
Algoritmo MergeSort (int [] u[0..n-1], int ini, fin)
    if (ini < fin) {
        int mid = (ini + fin)/2
        MergeSort (u, ini, mid)
        MergeSort (u, mid+1, fin)
        Merge (u, ini, fin)
    }
```

El algoritmo *Merge*

El algoritmo *Merge*

```
Algoritmo Merge (int [] u[0..n-1], int ini, fin)
    w = int [0..fin-inicio+1];
    int mid = (ini + fin)/2;
    i = ini;
    j = mid + 1;
    for (k = 0; k < fin − inicio + 1; k + +) {
        if ((j > fin) || (u[i] ≤ u[j] && i < mid + 1)) {
            w[k] = u[i];
            i + +;
        } else {
            w[k] = u[j];
            j + +;
        }
    }
    for (k = 0; k ≤ fin − ini; k + +) {
        u[ini + k] = w[k];
    }
}
```

Complejidad de *MergeSort*

Complejidad de *MergeSort*

- Para analizar la complejidad de MergeSort, observamos que cada llamada recursiva produce dos llamadas con el argumento dividido por dos. Por lo tanto, $a = 2$ y $b = 2$.

Complejidad de *MergeSort*

- Para analizar la complejidad de MergeSort, observamos que cada llamada recursiva produce dos llamadas con el argumento dividido por dos. Por lo tanto, $a = 2$ y $b = 2$.
- Para determinar k, debemos analizar la complejidad de Merge.

Complejidad de *MergeSort*

- Para analizar la complejidad de MergeSort, observamos que cada llamada recursiva produce dos llamadas con el argumento dividido por dos. Por lo tanto, $a = 2$ y $b = 2$.
- Para determinar k, debemos analizar la complejidad de Merge.
- La complejidad de *Merge* está dada por un ciclo de a los sumo n. Por lo tanto, tenemos $\mathcal{O}(n)$ y entonces $k = 1$.

Complejidad de *MergeSort*

- Para analizar la complejidad de MergeSort, observamos que cada llamada recursiva produce dos llamadas con el argumento dividido por dos. Por lo tanto, $a = 2$ y $b = 2$.
- Para determinar k, debemos analizar la complejidad de Merge.
- La complejidad de *Merge* está dada por un ciclo de a los sumo n. Por lo tanto, tenemos $\mathcal{O}(n)$ y entonces $k = 1$.
- Entonces estamos en el caso $a = b^k$ y la complejidad es $\Theta(n \log n)$.

Ejemplo de funcionamiento de *QuickSort*

Ejemplo de funcionamiento de *QuickSort*

7	1	8	5	3	6	2	9	0	4

Ejemplo de funcionamiento de *QuickSort*

7	1	8	5	3	6	2	9	0	4

Ejemplo de funcionamiento de *QuickSort*

7	1	8	5	3	6	2	9	0	4

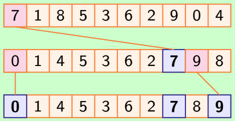
0	1	4	5	3	6	2	**7**	9	8

Ejemplo de funcionamiento de *QuickSort*

7	1	8	5	3	6	2	9	0	4

0	1	4	5	3	6	2	**7**	9	8

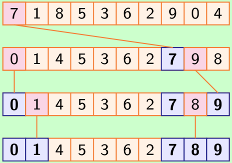
Ejemplo de funcionamiento de *QuickSort*



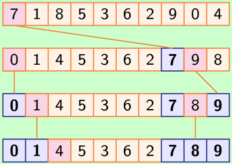
Ejemplo de funcionamiento de *QuickSort*

7	1	8	5	3	6	2	9	0	4
0	1	4	5	3	6	2	**7**	9	8
0	1	4	5	3	6	2	**7**	8	**9**

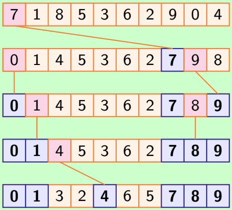
Ejemplo de funcionamiento de *QuickSort*



Ejemplo de funcionamiento de *QuickSort*



Ejemplo de funcionamiento de *QuickSort*



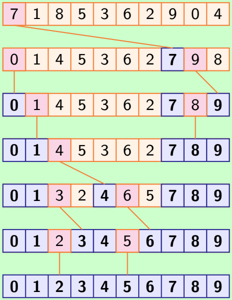
Ejemplo de funcionamiento de *QuickSort*

7	1	8	5	3	6	2	9	0	4
0	1	4	5	3	6	2	**7**	9	8
0	1	4	5	3	6	2	**7**	8	**9**
0	**1**	4	5	3	6	2	**7**	**8**	**9**
0	**1**	3	2	**4**	6	5	**7**	**8**	**9**

Ejemplo de funcionamiento de *QuickSort*

Ejemplo de funcionamiento de *QuickSort*

Ejemplo de funcionamiento de *QuickSort*



El algoritmo *QuickSort*

El algoritmo *QuickSort*

```
Algoritmo QuickSort (int [] u[1..n], int ini, fin)
    if (ini < fin) {
        integer p ← Pivot(ini, fin)
        QuickSort(u, ini, p − 1)
        QuickSort (u, p + 1, fin)
    }
```

El algoritmo *Pivot*

```
Algoritmo Pivot (int [] u[1..n], int ini, fin)
    p = u[ini];
    i = ini + 1;
    j = fin;
    while (u[i] ≤ p && i ≤ fin) {
        i + +;                                  i "junta" valores ≤ p
    }
    while (u[j] > p) {
        j − −;                                  j "junta" valores > p
    }
    while (i < j) {
        int aux = u[j];
        u[j] = u[i];
        u[i] = aux;
        while (u[i] ≤ p && i ≤ fin) {
            i + +;                              i "junta" valores ≤ p
        }
        while (u[j] > p) {
            j − −;                              j "junta" valores ≥ p
        }
    }
    u[ini] = u[j];
    u[j] = p;
    return j;
}
```

Complejidad de *Pivot*

Complejidad de *Pivot*

- La complejidad está dada por el número de iteraciones de los ciclos.

Complejidad de *Pivot*

- La complejidad está dada por el número de iteraciones de los ciclos.
- Los dos primeros ciclos iteran en el peor de los casos n veces.

Complejidad de *Pivot*

- La complejidad está dada por el número de iteraciones de los ciclos.
- Los dos primeros ciclos iteran en el peor de los casos n veces.
- Los ciclos anidados iteran en conjunto en el peor de los casos n veces.

Complejidad de *Pivot*

- La complejidad está dada por el número de iteraciones de los ciclos.
- Los dos primeros ciclos iteran en el peor de los casos n veces.
- Los ciclos anidados iteran en conjunto en el peor de los casos n veces.
- Por lo tanto, tenemos una complejidad $\mathcal{O}(n)$.

Complejidad de *QuickSort*

Complejidad de *QuickSort*

- Consideramos dos casos-límite: cuando el pivot está en la mitad exacta y cuando está en un extremo.

Complejidad de *QuickSort*

- Consideramos dos casos-límite: cuando el pivot está en la mitad exacta y cuando está en un extremo.
- Caso límite 1: tenemos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + p(n) & \text{si } n > 1 \end{cases}$$

Es decir, $a = 2$, $b = 2$ y $k = 1$. Entonces $T(n) \in \Theta(n \log n)$.

Complejidad de *QuickSort*

- Consideramos dos casos-límite: cuando el pivot está en la mitad exacta y cuando está en un extremo.
- Caso límite 1: tenemos

$$T(n)=\begin{cases} 1 & \text{si } n=1 \\ 2T(n/2)+p(n) & \text{si } n>1 \end{cases}$$

  Es decir, $a=2$, $b=2$ y $k=1$. Entonces $T(n)\in\Theta(n\log n)$.
- Caso límite 2: tenemos

$$T(n)=\begin{cases} 1 & \text{si } n=1 \\ T(n-1)+p(n) & \text{si } n>1 \end{cases}$$

  Es decir, $a=1$, $b=1$ y $k=1$. Entonces $T(n)\in\Theta(n^2)$.

Algunas consideraciones sobre el pivot

Algunas consideraciones sobre el pivot

- La complejidad de *QuickSort* es en el mejor de los casos equivalente a la de MergeSort ($\mathcal{O}(n \log n)$)...

Algunas consideraciones sobre el pivot

- La complejidad de *QuickSort* es en el mejor de los casos equivalente a la de MergeSort ($\mathcal{O}(n \log n)$)...
- ...y en el peor de los casos comparable a los casos de burbujeo o inserción ($\mathcal{O}(n^2)$).

Algunas consideraciones sobre el pivot

- La complejidad de *QuickSort* es en el mejor de los casos equivalente a la de MergeSort ($\mathcal{O}(n \log n)$)...
- ...y en el peor de los casos comparable a los casos de burbujeo o inserción ($\mathcal{O}(n^2)$).
- La eficiencia de *QuickSort* depende de cuán cerca está el pivot de la media del vector.

Algunas consideraciones sobre el pivot

- La complejidad de *QuickSort* es en el mejor de los casos equivalente a la de MergeSort ($\mathcal{O}(n \log n)$)...
- ...y en el peor de los casos comparable a los casos de burbujeo o inserción ($\mathcal{O}(n^2)$).
- La eficiencia de *QuickSort* depende de cuán cerca está el pivot de la media del vector.
- Existen algunas técnicas un poco más elaboradas para seleccionar el pivot.

Algunas consideraciones sobre el pivot

- La complejidad de *QuickSort* es en el mejor de los casos equivalente a la de MergeSort ($\mathcal{O}(n \log n)$)...
- ...y en el peor de los casos comparable a los casos de burbujeo o inserción ($\mathcal{O}(n^2)$).
- La eficiencia de *QuickSort* depende de cuán cerca está el pivot de la media del vector.
- Existen algunas técnicas un poco más elaboradas para seleccionar el pivot.
- Sin embargo, en la práctica a menudo *QuickSort* se desempeña mejor que *MergeSort*. ¿Puede imaginarse por qué?

Algunas conclusiones

Algunas conclusiones

- La idea central de la técnica *divide & conquer* es “deconstruir” un problema en componentes menores que puedan ser más fácilmente resueltos y posteriormente construir una solución combinando las soluciones parciales que se encontraron.

Algunas conclusiones

- La idea central de la técnica *divide & conquer* es “deconstruir” un problema en componentes menores que puedan ser más fácilmente resueltos y posteriormente construir una solución combinando las soluciones parciales que se encontraron.
- Éste es usualmente un proceso recursivo, que se aplica a los componentes menores, que por lo tanto son también “deconstruidos” en componentes aún menores.

Algunas conclusiones

- La idea central de la técnica *divide & conquer* es “deconstruir” un problema en componentes menores que puedan ser más fácilmente resueltos y posteriormente construir una solución combinando las soluciones parciales que se encontraron.
- Éste es usualmente un proceso recursivo, que se aplica a los componentes menores, que por lo tanto son también “deconstruidos” en componentes aún menores.
- Por supuesto, necesitamos un caso base como sucede siempre con la recurrencia.

Algunas conclusiones

- La idea central de la técnica *divide & conquer* es “deconstruir” un problema en componentes menores que puedan ser más fácilmente resueltos y posteriormente construir una solución combinando las soluciones parciales que se encontraron.
- Éste es usualmente un proceso recursivo, que se aplica a los componentes menores, que por lo tanto son también “deconstruidos” en componentes aún menores.
- Por supuesto, necesitamos un caso base como sucede siempre con la recurrencia.
- Los algoritmos de ordenamiento *divide & conquer* están entre los más eficientes.

Ejercicios propuestos 1

1. La municipalidad de Fraile Muerto tiene un registro de los n productores de soja de la región (como todo el mundo sabe, Fraile Muerto está en el corazón de la región sojera de Córdoba.) Los referidos productores están registrados con números secuenciales ($0, 1, 2, \ldots, n-1$). Estos datos están almacenados en un vector P. Pero alguien se percata de que en el vector hay un elemento repetido, es decir que tiene $n+1$ elementos en lugar de tener n, unque el último número es, correctamente, $n-1$.

   Se pide elaborar una estrategia y escribir el programa correspondiente para encontrar la posición del elemento repetido detallando las diferentes partes que componen la solución y su relación con el esquema general.

2. Calcule la raíz cuadrada entera de un número $n > 0$ utilizando *divide-and-conquer*. Recuerde que la raíz cuadrada entera de un número n es el máximo valor entero u tal que $u^2 \leq n$. Por ejemplo, la raíz entera de 18 es 4 y la de 9 es 3.

Ejercicios propuestos 2

3. El problema de la moneda falsa. El banco de Fraile Muerto recibió su primera remesa de monedas de cinco pesos. Dentro de un lote de n monedas se sabe que hay una falsa. La moneda falsa no se distingue de las otras sino por su peso: es un poco más pesada. Se dispone de una balanza de platillos que no da el peso cuantitativo; sólo dice si un grupo de monedas en un platillo pesa más, menos, o lo mismo que otro grupo de monedas en el otro platillo.

   Disponiendo de este *hardware*, más bien modesto, determine el número mínimo de pesadas (en el peor caso) para identificar la moneda falsa.

Ejercicios propuestos 3

4. Picos en un vector. Dado un vector u, un *elemento pico* es un elemento que no es menor que sus vecinos inmediatos. Encuentre un algoritmo eficiente (mejor que $\mathcal{O}(n)$) que encuentre algún elemento pico en un vector de n posiciones. Observe que puede haber varios elementos pico; basta encontrar uno de ellos (siempre hay por lo menos uno.)

   Ejemplo. Si la entrada fuera $[0, 1, 3, 2, 5, 1, 0]$ la respuesta podría ser 3 o 5. Si la entrada fuera $[7, 2, 3, 4, 5, 6]$ la respuesta podría ser 7 o 6. Si la entrada fuera $[1, 2, 3]$ la única respuesta sería 3.

5. Suponga que v es un vector de n dígitos binarios ordenados. Encuentre un algoritmo eficiente (mejor que $\mathcal{O}(n)$) que determine la cantidad de unos en v.

   Ejemplo. Si $v = [0, 0, 0, 1, 1, 1, 1, 1]$ (aquí $n = 8$). El programa debería retornar 5, que es la cantidad de unos en v.