Programación III

Ricardo Wehbe

UADE

31 de agosto de 2021

Programa

1. Repaso de la clase anterior
2. Divide & conquer. Algunos ejemplos
    - Palíndromos
    - Fibonacci
    - El elemento mayoritario
    - Las torres de Hanoi
    - El torneo de Bridge
    - Suma parcial máxima
3. Ejercicios propuestos

Divide & conquer

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."

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).

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

```
Algoritmo D&C (x)
    if isSmall(x) {
        return TrivialSolution (x)
    else
        ⟨x1, . . . , xn⟩ ← decompose (x)
        for (i = 0; i < n; i++) {
            yi ← D&C(xi)
        }
        return combine(y1, . . . , yn)
    }
```

MergeSort y *QuickSort*

MergeSort y *QuickSort*

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

MergeSort y *QuickSort*

- 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.

MergeSort y *QuickSort*

- 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.
- La complejidad de *MergeSort* está en $\mathcal{O}(n \log n)$.

MergeSort y *QuickSort*

- 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.
- La complejidad de *MergeSort* está en $\mathcal{O}(n \log n)$.
- 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.

MergeSort y *QuickSort*

- 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.
- La complejidad de *MergeSort* está en $\mathcal{O}(n \log n)$.
- 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.
- *QuickSort* está entre $\mathcal{O}(n \log n)$ (mejor caso) y $\mathcal{O}(n^2)$ (peor caso.)

Palíndromos

Palíndromos

- Un *palíndromo* es un vector que tiene la misma secuencia de elementos en ambos sentidos (por ejemplo, los vectores $[1, 2, 3, 2, 1]$ y $[1]$ son palíndromos; el vector $[1, 2, 1, 1]$ no lo es.)

Palíndromos

- Un *palíndromo* es un vector que tiene la misma secuencia de elementos en ambos sentidos (por ejemplo, los vectores $[1, 2, 3, 2, 1]$ y $[1]$ son palíndromos; el vector $[1, 2, 1, 1]$ no lo es.)
- Buscamos un algoritmo que devuelva *true* si un vector es palíndromo y *false* en caso contrario.

Palíndromos

- Un *palíndromo* es un vector que tiene la misma secuencia de elementos en ambos sentidos (por ejemplo, los vectores $[1, 2, 3, 2, 1]$ y $[1]$ son palíndromos; el vector $[1, 2, 1, 1]$ no lo es.)
- Buscamos un algoritmo que devuelva *true* si un vector es palíndromo y *false* en caso contrario.
- El algoritmo natural para atacar este problema consiste en comparar el primer elemento del vector con el último y, en caso de ser iguales, proseguir recursivamente con el resto del vector.

Algoritmo para la determinación de palíndromos

```
boolean Algoritmo Palindrome (int [] u[1..n], int ini, fin) {
    if (ini ≥ fin) {
        return true;
    } else if (u[ini] ≠ u[fin]) {
        return false;
    } else {
        return Palindrome(u, ini + 1, fin − 1);
    }
}
```

Palíndromos. Cálculo de la complejidad

Palíndromos. Cálculo de la complejidad

- Tenemos una recurrencia con substracción.

Palíndromos. Cálculo de la complejidad

- Tenemos una recurrencia con substracción.
- Los valores son $a = 1$, $b = 2$ y $k = 0$. Estamos entonces en el caso $a = 1$.

Palíndromos. Cálculo de la complejidad

- Tenemos una recurrencia con substracción.
- Los valores son $a = 1$, $b = 2$ y $k = 0$. Estamos entonces en el caso $a = 1$.
- Estamos entonces en el caso $\Theta(n^{k+1}) = \Theta(n)$.

La función de Fibonacci

La función de Fibonacci

- La función de Fibonacci, que es una función $F : \mathbb{N}_0 \to \mathbb{N}_0$ se define como sigue:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

La función de Fibonacci

- La función de Fibonacci, que es una función $F : \mathbb{N}_0 \to \mathbb{N}_0$ se define como sigue:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

- Una implementación directa de esto es la siguiente:

```
int Algoritmo Fib (int n) {
    } if (n ≤ 1) {                    // casos base
        return n;
    } else {
        return Fib(n − 1)+Fib(n − 2);
    }
}
```

La función de Fibonacci. Complejidad

La función de Fibonacci. Complejidad

- Aquí tenemos un caso de recurrencia por substracción.

La función de Fibonacci. Complejidad

- Aquí tenemos un caso de recurrencia por substracción.
- Tenemos $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en $\Theta(n^k a^{n/b}) = \Theta(2^n)$.

La función de Fibonacci. Complejidad

- Aquí tenemos un caso de recurrencia por substracción.
- Tenemos $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en $\Theta(n^k a^{n/b}) = \Theta(2^n)$.
- ¿Por qué tanto?

La función de Fibonacci. Complejidad

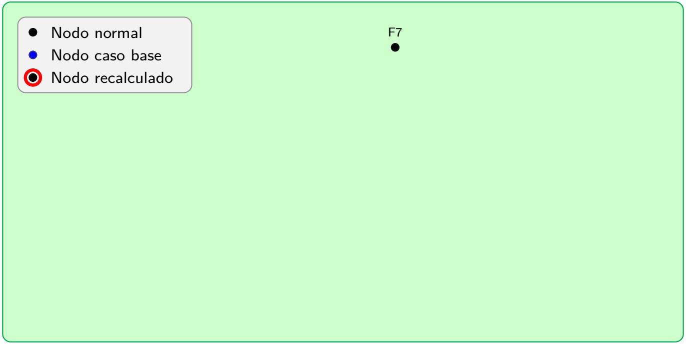
- Aquí tenemos un caso de recurrencia por substracción.
- Tenemos $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en $\Theta(n^k a^{n/b}) = \Theta(2^n)$.
- ¿Por qué tanto?
- Hay demasiadas cosas que se calculan una y otra vez. Los sub-problemas no son independientes.

La función de Fibonacci. Complejidad

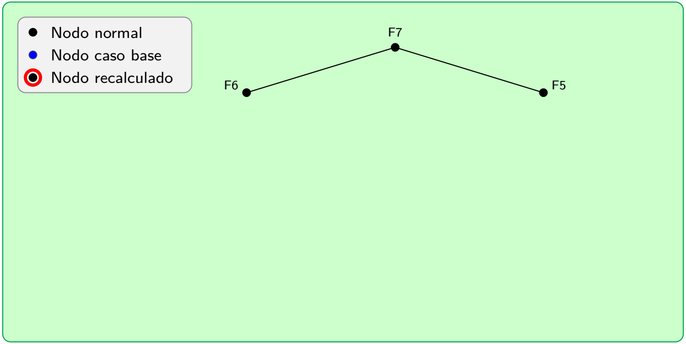
- Aquí tenemos un caso de recurrencia por substracción.
- Tenemos $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en $\Theta(n^k a^{n/b}) = \Theta(2^n)$.
- ¿Por qué tanto?
- Hay demasiadas cosas que se calculan una y otra vez. Los sub-problemas no son independientes.
- Una solución más eficiente queda pendiente hasta que veamos programación dinámica.

Fibonacci. Cálculos redundantes

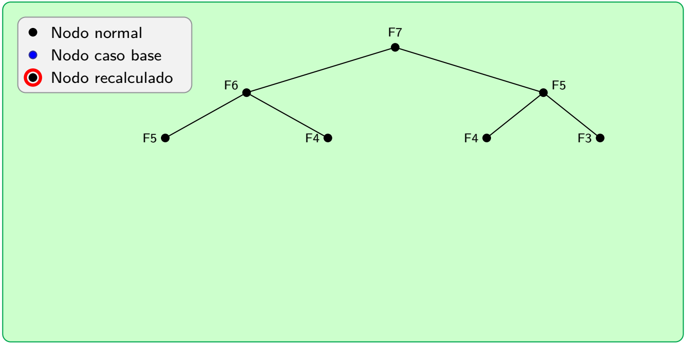
Fibonacci. Cálculos redundantes



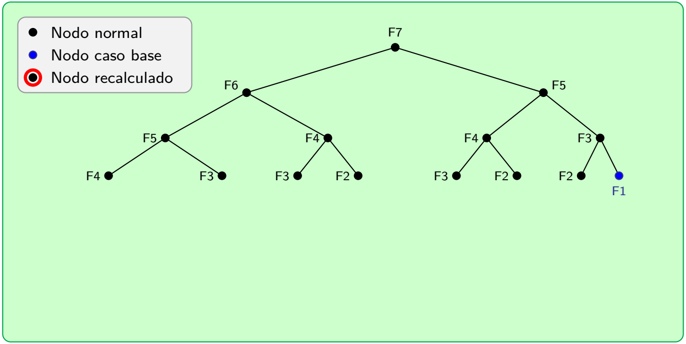
Fibonacci. Cálculos redundantes



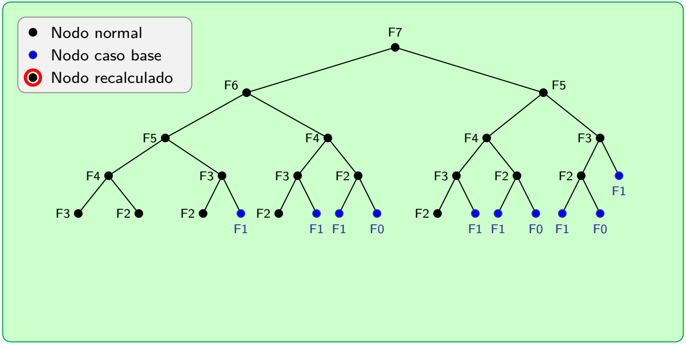
Fibonacci. Cálculos redundantes



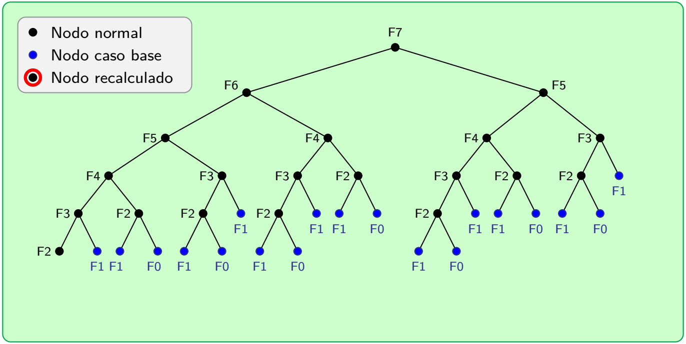
Fibonacci. Cálculos redundantes



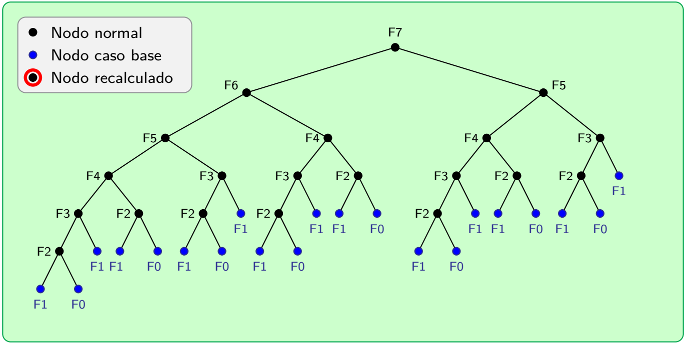
Fibonacci. Cálculos redundantes



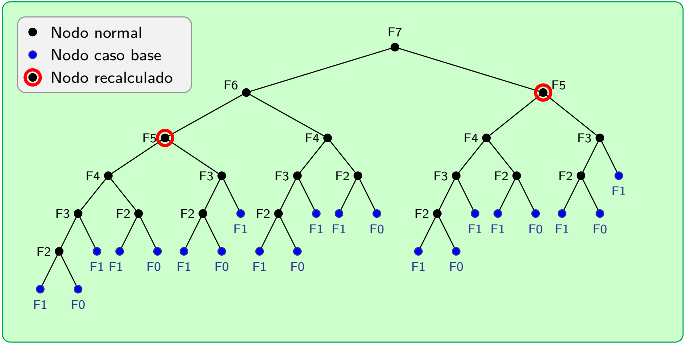
Fibonacci. Cálculos redundantes



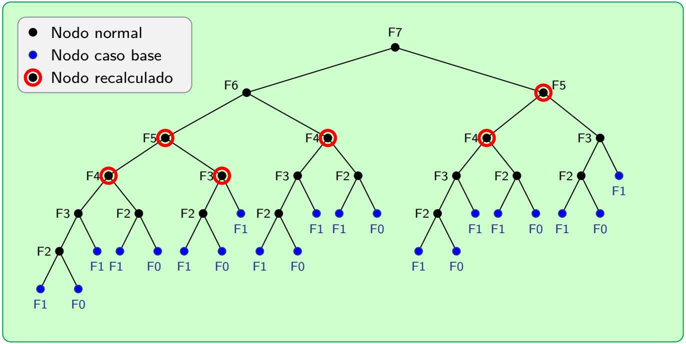
Fibonacci. Cálculos redundantes



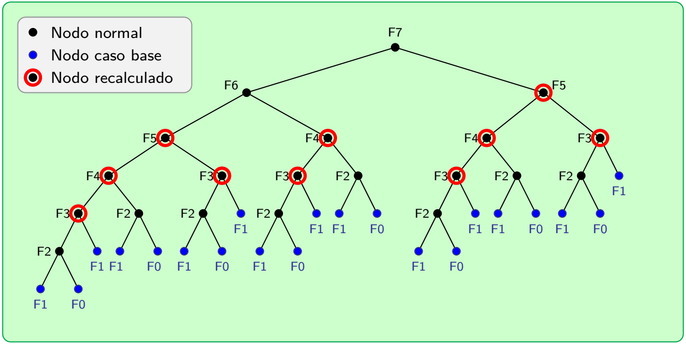
Fibonacci. Cálculos redundantes



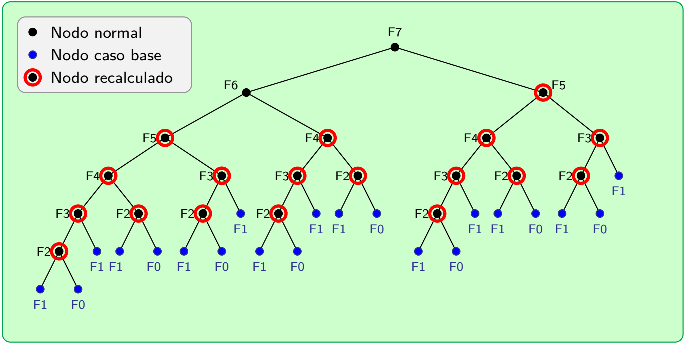
Fibonacci. Cálculos redundantes



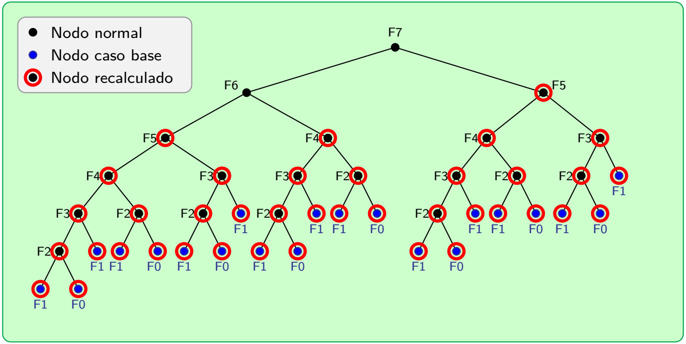
Fibonacci. Cálculos redundantes



Fibonacci. Cálculos redundantes



Fibonacci. Cálculos redundantes



El problema del elemento mayoritario

El problema del elemento mayoritario

- Tenemos un vector $u[1..n]$. Un elemento x de u es *mayoritario* en u si aparece por lo menos $n/2 + 1$ veces en u.

El problema del elemento mayoritario

- Tenemos un vector $u[1..n]$. Un elemento x de u es *mayoritario* en u si aparece por lo menos $n/2+1$ veces en u.
- Ejemplo: 7 es mayoritario en $[7,1,7]$ y 2 en $[2,1,2,2]$. No hay ningún elemento mayoritario en $[1,2,1,3,3,2,1,1]$.

El problema del elemento mayoritario

- Tenemos un vector $u[1..n]$. Un elemento x de u es *mayoritario* en u si aparece por lo menos $n/2+1$ veces en u.
- Ejemplo: 7 es mayoritario en $[7,1,7]$ y 2 en $[2,1,2,2]$. No hay ningún elemento mayoritario en $[1,2,1,3,3,2,1,1]$.
- La estrategia *naïf* consiste en revisar el número de ocurrencias de cada elemento hasta encontrar uno que supere las $n/2+1$ ocurrencias (si lo hubiere.) Esto tiene complejidad $\mathcal{O}(n^2)$.

El problema del elemento mayoritario

- Tenemos un vector $u[1..n]$. Un elemento x de u es *mayoritario* en u si aparece por lo menos $n/2+1$ veces en u.
- Ejemplo: 7 es mayoritario en $[7,1,7]$ y 2 en $[2,1,2,2]$. No hay ningún elemento mayoritario en $[1,2,1,3,3,2,1,1]$.
- La estrategia *naïf* consiste en revisar el número de ocurrencias de cada elemento hasta encontrar uno que supere las $n/2+1$ ocurrencias (si lo hubiere.) Esto tiene complejidad $\mathcal{O}(n^2)$.
- Una estrategia un poco más elaborada consiste en ordenar el vector ($\mathcal{O}(n\log n)$.) Si x es mayoritario, debe estar en la posición $n/2$ del vector ordenado. Basta entonces contar la cantidad de veces que $u[n/2]$ aparece en u (costo $\mathcal{O}(n)$.)

El problema del elemento mayoritario

- Tenemos un vector $u[1..n]$. Un elemento x de u es *mayoritario* en u si aparece por lo menos $n/2+1$ veces en u.
- Ejemplo: 7 es mayoritario en $[7,1,7]$ y 2 en $[2,1,2,2]$. No hay ningún elemento mayoritario en $[1,2,1,3,3,2,1,1]$.
- La estrategia *naïf* consiste en revisar el número de ocurrencias de cada elemento hasta encontrar uno que supere las $n/2+1$ ocurrencias (si lo hubiere.) Esto tiene complejidad $\mathcal{O}(n^2)$.
- Una estrategia un poco más elaborada consiste en ordenar el vector ($\mathcal{O}(n\log n)$.) Si x es mayoritario, debe estar en la posición $n/2$ del vector ordenado. Basta entonces contar la cantidad de veces que $u[n/2]$ aparece en u (costo $\mathcal{O}(n)$.)
- Tendríamos entonces $\mathcal{O}(n\log n)+\mathcal{O}(n)=\mathcal{O}(n\log n)$.

El problema del elemento mayoritario

- Tenemos un vector $u[1..n]$. Un elemento x de u es *mayoritario* en u si aparece por lo menos $n/2+1$ veces en u.
- Ejemplo: 7 es mayoritario en $[7,1,7]$ y 2 en $[2,1,2,2]$. No hay ningún elemento mayoritario en $[1,2,1,3,3,2,1,1]$.
- La estrategia *naïf* consiste en revisar el número de ocurrencias de cada elemento hasta encontrar uno que supere las $n/2+1$ ocurrencias (si lo hubiere.) Esto tiene complejidad $\mathcal{O}(n^2)$.
- Una estrategia un poco más elaborada consiste en ordenar el vector ($\mathcal{O}(n\log n)$.) Si x es mayoritario, debe estar en la posición $n/2$ del vector ordenado. Basta entonces contar la cantidad de veces que $u[n/2]$ aparece en u (costo $\mathcal{O}(n)$.)
- Tendríamos entonces $\mathcal{O}(n\log n)+\mathcal{O}(n)=\mathcal{O}(n\log n)$.
- ¿Podremos mejorar esto?

Intermezzo. El algoritmo *pseudo-M*

Intermezzo. El algoritmo *pseudo-M*

- Nos conformaremos por ahora con diseñar un algoritmo que nos devuelve un candidato a elemento mayoritario.

Intermezzo. El algoritmo *pseudo-M*

- Nos conformaremos por ahora con diseñar un algoritmo que nos devuelve un candidato a elemento mayoritario.
- Este algoritmo toma como entrada un vector $u[1..n]$ y devuelve una salida (r, n, c_x, x), donde n es la cantidad de elementos e u, tal que:

Intermezzo. El algoritmo *pseudo-M*

- Nos conformaremos por ahora con diseñar un algoritmo que nos devuelve un candidato a elemento mayoritario.
- Este algoritmo toma como entrada un vector $u[1..n]$ y devuelve una salida (r, n, c_x, x), donde n es la cantidad de elementos e u, tal que:
  - Si no hay elemento mayoritario en u, el algoritmo devuelve $(\texttt{false}, n, 0, 0)$.

Intermezzo. El algoritmo *pseudo-M*

- Nos conformaremos por ahora con diseñar un algoritmo que nos devuelve un candidato a elemento mayoritario.
- Este algoritmo toma como entrada un vector $u[1..n]$ y devuelve una salida (r, n, c_x, x), donde n es la cantidad de elementos e u, tal que:
  - Si no hay elemento mayoritario en u, el algoritmo devuelve $(\texttt{false}, n, 0, 0)$.
  - Si hay un candidato x, el algoritmo devuelve $(\texttt{true}, n, c_x, x)$ donde x aparece *a lo sumo* $n \geq c_x \geq n/2 + 1$ veces y cualquier otro elemento $y \neq x$ aparece *a lo sumo* $n - c_x$ veces.

Intermezzo. El algoritmo *pseudo-M*

- Nos conformaremos por ahora con diseñar un algoritmo que nos devuelve un candidato a elemento mayoritario.
- Este algoritmo toma como entrada un vector $u[1..n]$ y devuelve una salida (r, n, c_x, x), donde n es la cantidad de elementos e u, tal que:
  - Si no hay elemento mayoritario en u, el algoritmo devuelve $(\texttt{false}, n, 0, 0)$.
  - Si hay un candidato x, el algoritmo devuelve $(\texttt{true}, n, c_x, x)$ donde x aparece *a lo sumo* $n \geq c_x \geq n/2 + 1$ veces y cualquier otro elemento $y \neq x$ aparece *a lo sumo* $n - c_x$ veces.
- Observe que si r_x es $true$, entonces sólo x (el candidato) puede ser un elemento mayoritario en u.

Intermezzo. El algoritmo *pseudo-M*

- Nos conformaremos por ahora con diseñar un algoritmo que nos devuelve un candidato a elemento mayoritario.
- Este algoritmo toma como entrada un vector $u[1..n]$ y devuelve una salida (r, n, c_x, x), donde n es la cantidad de elementos e u, tal que:
  - Si no hay elemento mayoritario en u, el algoritmo devuelve $(\mathtt{false}, n, 0, 0)$.
  - Si hay un candidato x, el algoritmo devuelve $(\mathtt{true}, n, c_x, x)$ donde x aparece *a lo sumo* $n \geq c_x \geq n/2 + 1$ veces y cualquier otro elemento $y \neq x$ aparece *a lo sumo* $n - c_x$ veces.
- Observe que si r_x es *true*, entonces sólo x (el candidato) puede ser un elemento mayoritario en u.
- El desafío es entonces escribir este programa siguiendo la metodología *divide & conquer*.

Estrategia para el algoritmo *pseudo-M*

Estrategia para el algoritmo *pseudo-M*

- Tenga en cuenta que el algoritmo produce un candidato; éste podría no ser el elemento mayoritario. Pero mientras un elemento pueda ser mayoritario, debe mantenérselo como candidato.

Estrategia para el algoritmo *pseudo-M*

- Tenga en cuenta que el algoritmo produce un candidato; éste podría no ser el elemento mayoritario. Pero mientras un elemento pueda ser mayoritario, debe mantenérselo como candidato.
- El valor de c uede por lo tanto estar sobreeestimado; nunca puede estar subestimado.

Estrategia para el algoritmo *pseudo-M*

- Tenga en cuenta que el algoritmo produce un candidato; éste podría no ser el elemento mayoritario. Pero mientras un elemento pueda ser mayoritario, debe mantenérselo como candidato.
- El valor de c uede por lo tanto estar sobreeestimado; nunca puede estar subestimado.
- El único caso base es $\textit{pseudo-M}([x]) = (\textit{true}, 1, 1, x)$.

Estrategia para el algoritmo *pseudo-M*

- Tenga en cuenta que el algoritmo produce un candidato; éste podría no ser el elemento mayoritario. Pero mientras un elemento pueda ser mayoritario, debe mantenérselo como candidato.
- El valor de c uede por lo tanto estar sobreeestimado; nunca puede estar subestimado.
- El único caso base es $\textit{pseudo-M}([x]) = (\textit{true}, 1, 1, x)$.
- ¿Cómo se combinan los resultados para u_1 y u_1?

Estrategia para el algoritmo *pseudo-M*

- Tenga en cuenta que el algoritmo produce un candidato; éste podría no ser el elemento mayoritario. Pero mientras un elemento pueda ser mayoritario, debe mantenérselo como candidato.
- El valor de c uede por lo tanto estar sobreeestimado; nunca puede estar subestimado.
- El único caso base es $\textit{pseudo-M}([x]) = (\textit{true}, 1, 1, x)$.
- ¿Cómo se combinan los resultados para u_1 y u_1?
- Debemos considerar todos los posibles casos.

Estrategia para el algoritmo *pseudo-M*

- Tenga en cuenta que el algoritmo produce un candidato; éste podría no ser el elemento mayoritario. Pero mientras un elemento pueda ser mayoritario, debe mantenérselo como candidato.
- El valor de c uede por lo tanto estar sobreeestimado; nunca puede estar subestimado.
- El único caso base es $\textit{pseudo-M}([x]) = (\textit{true}, 1, 1, x)$.
- ¿Cómo se combinan los resultados para u_1 y u_1?
- Debemos considerar todos los posibles casos.
- En el siguiente *slide* hay una hoja de ruta de la aplicación de este algoritmo.

Una hoja de ruta para el algoritmo *Pseudo-M*

-

Una hoja de ruta para el algoritmo *Pseudo-M*

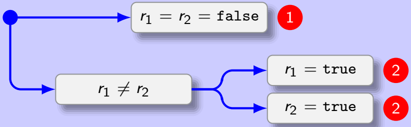
$\bullet \longrightarrow$ $\boxed{r_1 = r_2 = \texttt{false}}$ 1

Una hoja de ruta para el algoritmo *Pseudo-M*

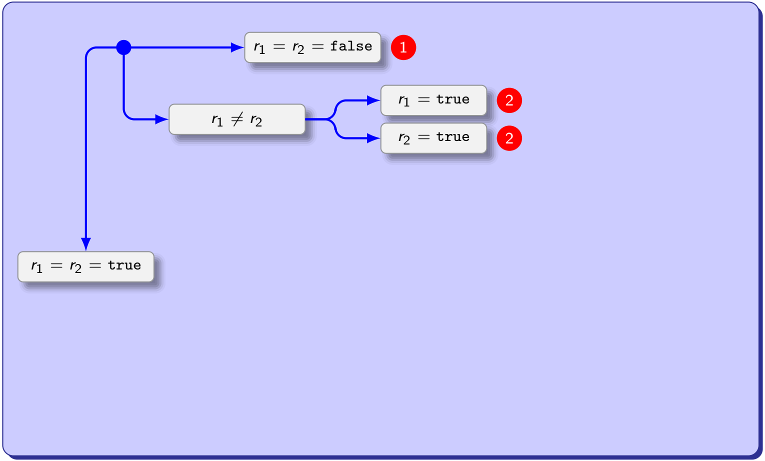
$r_1 = r_2 =$ `false` 1

$r_1 \neq r_2$

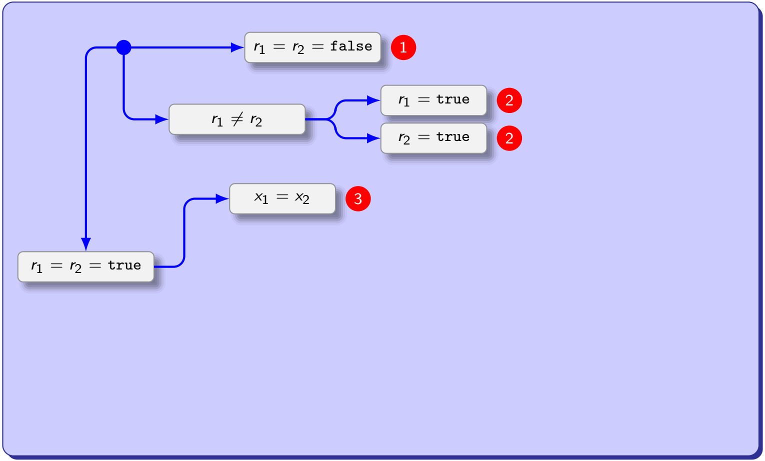
Una hoja de ruta para el algoritmo *Pseudo-M*



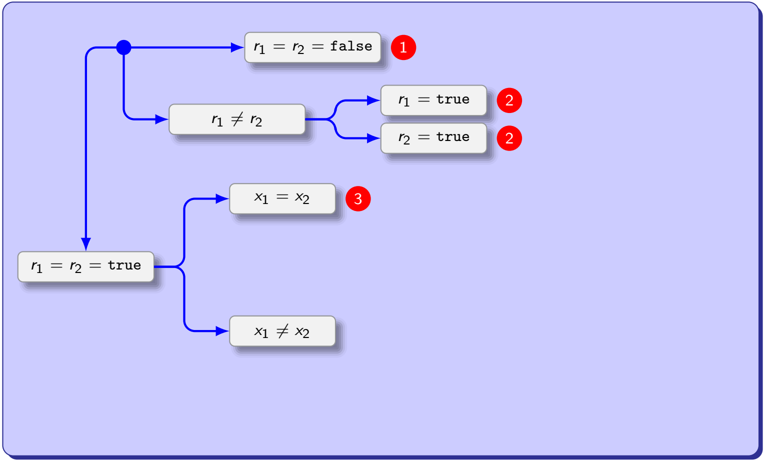
Una hoja de ruta para el algoritmo *Pseudo-M*



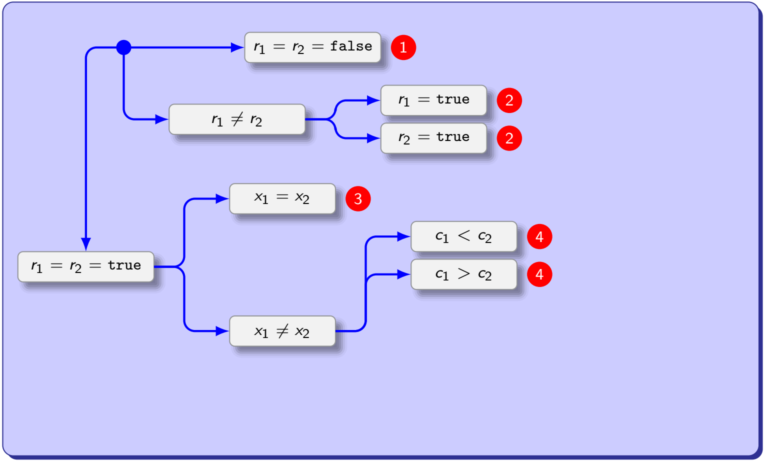
Una hoja de ruta para el algoritmo *Pseudo-M*



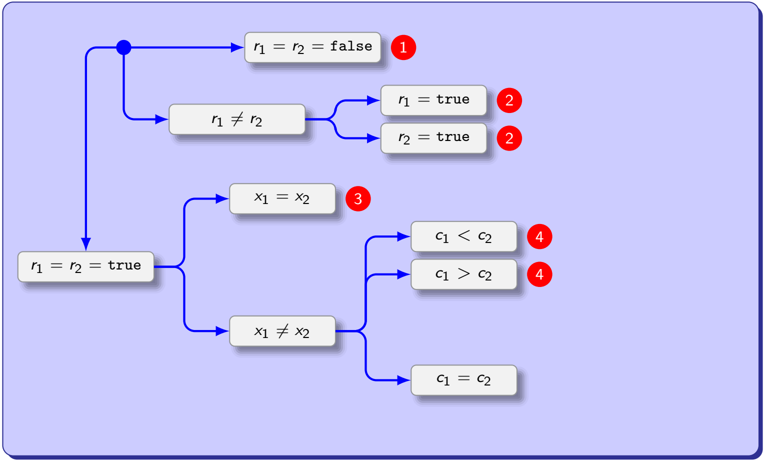
Una hoja de ruta para el algoritmo *Pseudo-M*



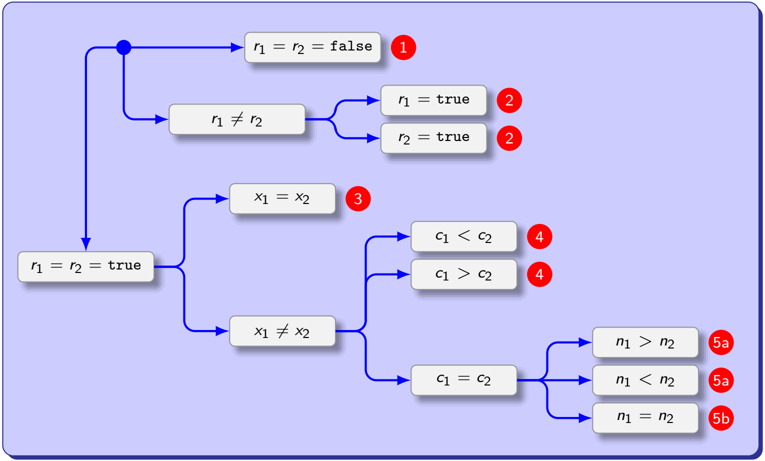
Una hoja de ruta para el algoritmo *Pseudo-M*



Una hoja de ruta para el algoritmo *Pseudo-M*



Una hoja de ruta para el algoritmo *Pseudo-M*



Análisis de las combinaciones. Estrategia

Análisis de las combinaciones. Estrategia

- Recuerde que tenemos un sub-arreglo u de longitud n.

Análisis de las combinaciones. Estrategia

- Recuerde que tenemos un sub-arreglo u de longitud n.
- Por lo tanto, el resultado $(\mathtt{true}, n, c_x, x)$ significa que en u hay un elemento x que aparece a lo sumo c_x veces con $n \geq c_x \geq n/2 + 1$ y que cualquier otro elemento aparece a lo sumo $n - c_x$ veces, y no puede por lo tanto ser elemento mayoritario.

Análisis de las combinaciones. Estrategia

- Recuerde que tenemos un sub-arreglo u de longitud n.
- Por lo tanto, el resultado $(\mathtt{true}, n, c_x, x)$ significa que en u hay un elemento x que aparece a lo sumo c_x veces con $n \geq c_x \geq n/2 + 1$ y que cualquier otro elemento aparece a lo sumo $n - c_x$ veces, y no puede por lo tanto ser elemento mayoritario.
- El resultado $(\mathtt{false}, n, 0, 0)$ significa que ningún elemento aparece más de $n/2$ veces.

Análisis de las combinaciones. Estrategia

- Recuerde que tenemos un sub-arreglo u de longitud n.
- Por lo tanto, el resultado $(\texttt{true}, n, c_x, x)$ significa que en u hay un elemento x que aparece a lo sumo c_x veces con $n \geq c_x \geq n/2 + 1$ y que cualquier otro elemento aparece a lo sumo $n - c_x$ veces, y no puede por lo tanto ser elemento mayoritario.
- El resultado $(\texttt{false}, n, 0, 0)$ significa que ningún elemento aparece más de $n/2$ veces.
- Si combinamos los resultados de dos sub-arreglos u_1 y u_2, debemos determinar si en su concatenación sucede alguna de las siguientes situaciones:
  - No hay ningún elemento mayoritario y el resultado para la concatenación es $(\texttt{false}, n_1 + n_2, 0, 0)$.
  - Hay un elemento x que aparece a lo sumo c veces $(n \geq c \geq n/2 + 1)$ y ningún otro elemento aparece más de $n/2$ veces. El resultado es entonces $(\texttt{true}, n_1 + n_2, c, x)$.

Combinaciones de resultados. Caso 1

No hay cancidato en ningún arreglo: $r_1 = r_2 =$ `false`

Combinaciones de resultados. Caso 1

No hay cancidato en ningún arreglo: $r_1 = r_2 =$ `false`

u_1

u_2

Combinaciones de resultados. Caso 1

No hay cancidato en ningún arreglo: $r_1 = r_2 =$ `false`

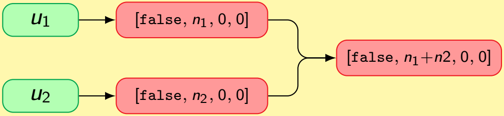
$u_1 \longrightarrow$ [`false`, n_1, 0, 0]

$u_2 \longrightarrow$ [`false`, n_2, 0, 0]

Ningún elemento aparece más de $n_1/2$ veces en u_1.
Ningún elemento aparece más de $n_2/2$ veces en u_2.
Por lo tanto, ningún elemento aparece más de $(n_1+n_2)/2$ veces en u_1+u_2.

Combinaciones de resultados. Caso 1

No hay cancidato en ningún arreglo: $r_1 = r_2 =$ `false`



Ningún elemento aparece más de $n_1/2$ veces en u_1.
Ningún elemento aparece más de $n_2/2$ veces en u_2.
Por lo tanto, ningún elemento aparece más de $(n_1+n_2)/2$ veces en u_1+u_2.
No hay entonces ningún candidato.

Combinaciones de resultados. Caso 1

No hay cancidato en ningún arreglo: $r_1 = r_2 = \texttt{false}$

$u_1 \rightarrow [\texttt{false}, n_1, 0, 0]$

$u_2 \rightarrow [\texttt{false}, n_2, 0, 0]$

$\rightarrow [\texttt{false}, n_1{+}n2, 0, 0]$

Ningún elemento aparece más de $n_1/2$ veces en u_1.
Ningún elemento aparece más de $n_2/2$ veces en u_2.
Por lo tanto, ningún elemento aparece más de $(n_1{+}n_2)/2$ veces en $u_1{+}u_2$.
No hay entonces ningún candidato.

Ejemplo 1:

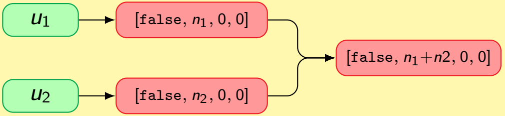
1,2 → [false,2,0,0]

2,1 → [false,2,0,0]

→ [false,4,0,0]

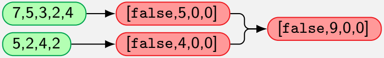
Combinaciones de resultados. Caso 1

No hay cancidato en ningún arreglo: $r_1 = r_2 =$ `false`



Ningún elemento aparece más de $n_1/2$ veces en u_1.
Ningún elemento aparece más de $n_2/2$ veces en u_2.
Por lo tanto, ningún elemento aparece más de $(n_1+n_2)/2$ veces en u_1+u_2.
No hay entonces ningún candidato.

Ejemplo 2:



Combinaciones de resultados. Caso 2

Hay un candidato en uno de los arreglos: $r_1 \neq r_2$

Combinaciones de resultados. Caso 2

Hay un candidato en uno de los arreglos: $r_1 \neq r_2$

u_1

u_2

Combinaciones de resultados. Caso 2

Hay un candidato en uno de los arreglos: $r_1 \neq r_2$

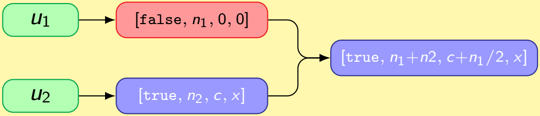
$u_1 \rightarrow$ [$\texttt{false}, n_1, 0, 0$]

$u_2 \rightarrow$ [$\texttt{true}, n_2, c, x$]

El elemento x aparece a lo sumo c veces en u_2 y a lo sumo c veces en u_1. Cualquier otro elemento aparece a lo sumo $n_1/2$ veces en u_1 y a lo sumo n_1-$c \leq n_1/2$ veces en u_2.

Combinaciones de resultados. Caso 2

Hay un candidato en uno de los arreglos: $r_1 \neq r_2$



El elemento x aparece a lo sumo c veces en u_2 y a lo sumo c veces en u_1. Cualquier otro elemento aparece a lo sumo $n_1/2$ veces en u_1 y a lo sumo n_1-$c \leq n_1/2$ veces en u_2.
El candidato es entonces x, que aparece a lo sumo $c+n_1/2 \geq (n_1+n_2)/2+1$ veces en u_1+u_2.

Combinaciones de resultados. Caso 2

Hay un candidato en uno de los arreglos: $r_1 \neq r_2$

$u_1 \rightarrow$ [`false`, n_1, 0, 0]

$u_2 \rightarrow$ [`true`, n_2, c, x]

$\rightarrow$ [`true`, $n_1{+}n2$, $c{+}n_1/2$, x]

El elemento x aparece a lo sumo c veces en u_2 y a lo sumo c veces en u_1. Cualquier otro elemento aparece a lo sumo $n_1/2$ veces en u_1 y a lo sumo n_1-$c \leq n_1/2$ veces en u_2.
El candidato es entonces x, que aparece a lo sumo $c{+}n_1/2 \geq (n_1{+}n_2)/2{+}1$ veces en $u_1{+}u_2$.

Ejemplo 1:

1,2 $\rightarrow$ [`false`,2,0,0]

3 $\rightarrow$ [`true`,1,1,3]

$\rightarrow$ [`true`,3,2,3]

Combinaciones de resultados. Caso 2

Hay un candidato en uno de los arreglos: $r_1 \neq r_2$

$u_1 \rightarrow$ [`false`, n_1, 0, 0]

$u_2 \rightarrow$ [`true`, n_2, c, x]

$\rightarrow$ [`true`, n_1+n2, $c+n_1/2$, x]

El elemento x aparece a lo sumo c veces en u_2 y a lo sumo c veces en u_1. Cualquier otro elemento aparece a lo sumo $n_1/2$ veces en u_1 y a lo sumo n_1-$c \leq n_1/2$ veces en u_2.
El candidato es entonces x, que aparece a lo sumo $c+n_1/2 \geq (n_1+n_2)/2+1$ veces en u_1+u_2.

Ejemplo 2:

1,1 $\rightarrow$ [`true`,2,2,1]

1,2 $\rightarrow$ [`false`,2,0,0]

$\rightarrow$ [`true`,4,3,1]

Combinaciones de resultados. Caso 3

The same candidate appears on both sub-arrays.

Combinaciones de resultados. Caso 3

The same candidate appears on both sub-arrays.

u_1

u_2

Combinaciones de resultados. Caso 3

The same candidate appears on both sub-arrays.

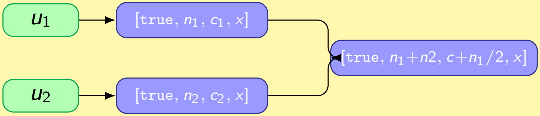
$u_1 \longrightarrow [\texttt{true}, n_1, c_1, x]$

$u_2 \longrightarrow [\texttt{true}, n_2, c_2, x]$

El elemento x aparece a lo sumo c_1 veces en u_1 y a lo sumo c_2 veces en u_2. Cualquier otro elemento aparece a lo sumo n_1-$c_1 \leq n_1/2$ veces en u_1 y a sumo n_2-$c_2 \leq n_2/2$ veces en u_2.

Combinaciones de resultados. Caso 3

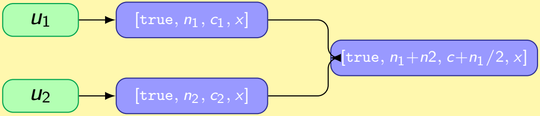
The same candidate appears on both sub-arrays.



El elemento x aparece a lo sumo c_1 veces en u_1 y a lo sumo c_2 veces en u_2. Cualquier otro elemento aparece a lo sumo n_1-$c_1 \leq n_1/2$ veces en u_1 y a sumo n_2-$c_2 \leq n_2/2$ veces en u_2.
El único candidato es entonces x, que aparece a lo sumo $c_1+c_2 \geq (n_1+n_2)/2+1$ veces en u_1+u_2.

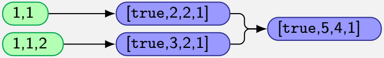
Combinaciones de resultados. Caso 3

The same candidate appears on both sub-arrays.



El elemento x aparece a lo sumo c_1 veces en u_1 y a lo sumo c_2 veces en u_2. Cualquier otro elemento aparece a lo sumo n_1-$c_1 \leq n_1/2$ veces en u_1 y a sumo n_2-$c_2 \leq n_2/2$ veces en u_2.
El único candidato es entonces x, que aparece a lo sumo $c_1{+}c_2 \geq (n_1{+}n_2)/2{+}1$ veces en $u_1{+}u_2$.

Ejemplo 1:



Combinaciones de resultados. Caso 3

The same candidate appears on both sub-arrays.

u_1 → [`true`, n_1, c_1, x]

u_2 → [`true`, n_2, c_2, x]

→ [`true`, n_1+n2, $c+n_1/2$, x]

El elemento x aparece a lo sumo c_1 veces en u_1 y a lo sumo c_2 veces en u_2. Cualquier otro elemento aparece a lo sumo n_1-$c_1 \leq n_1/2$ veces en u_1 y a sumo n_2-$c_2 \leq n_2/2$ veces en u_2.
El único candidato es entonces x, que aparece a lo sumo $c_1+c_2 \geq (n_1+n_2)/2+1$ veces en u_1+u_2.

Ejemplo 2:

2,8,3,8,8 → [`true`,5,3,8]

8,2,8,8 → [`true`,4,3,8]

→ [`true`,9,6,8]

Combinaciones de resultados. Caso 4

Hay diferentes candidatos en ambos arreglos, $c_1 > c_2$

Combinaciones de resultados. Caso 4

Hay diferentes candidatos en ambos arreglos, $c_1 > c_2$

u_1

u_2

Combinaciones de resultados. Caso 4

Hay diferentes candidatos en ambos arreglos, $c_1 > c_2$

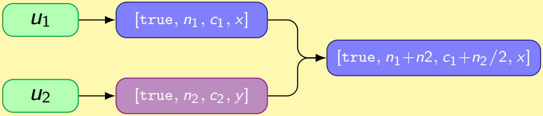
$u_1 \longrightarrow [\texttt{true}, n_1, c_1, x]$

$u_2 \longrightarrow [\texttt{true}, n_2, c_2, y]$

El elemento x aparece a lo sumo c_1 veces en u_1 y a sumo n_2-c_2 veces en u_2. Aparece a lo sumo $n_2+(c_1\text{-}c_2) \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, y aparece a lo sumo $n_1+(c_2\text{-}c_1) \leq (n_1+n_2)/2$ veces en u_1+u_2

Combinaciones de resultados. Caso 4

Hay diferentes candidatos en ambos arreglos, $c_1 > c_2$



El elemento x aparece a lo sumo c_1 veces en u_1 y a sumo n_2-c_2 veces en u_2. Aparece a lo sumo $n_2+(c_1\text{-}c_2) \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, y aparece a lo sumo $n_1+(c_2\text{-}c_1) \leq (n_1+n_2)/2$ veces en u_1+u_2
El único candidato es entonces x.

Combinaciones de resultados. Caso 4

Hay diferentes candidatos en ambos arreglos, $c_1 > c_2$

$$\begin{tikzcd}
u_1 \arrow[r] & {[\texttt{true}, n_1, c_1, x]} \arrow[dr] & \\
 & & {[\texttt{true}, n_1+n2, c_1+n_2/2, x]} \\
u_2 \arrow[r] & {[\texttt{true}, n_2, c_2, y]} \arrow[ur] &
\end{tikzcd}$$

El elemento x aparece a lo sumo c_1 veces en u_1 y a sumo n_2-c_2 veces en u_2. Aparece a lo sumo $n_2+(c_1\text{-}c_2) \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, y aparece a lo sumo $n_1+(c_2\text{-}c_1) \leq (n_1+n_2)/2$ veces en u_1+u_2
El único candidato es entonces x.

Ejemplo 1:

$$\begin{tikzcd}
4,4,4 \arrow[r] & {[\texttt{true},3,3,4]} \arrow[dr] & \\
 & & {[\texttt{true},6,4,4]} \\
4,1,1 \arrow[r] & {[\texttt{true},3,2,1]} \arrow[ur] &
\end{tikzcd}$$

Combinaciones de resultados. Caso 4

Hay diferentes candidatos en ambos arreglos, $c_1 > c_2$

$u_1 \rightarrow$ [true, n_1, c_1, x]

$u_2 \rightarrow$ [true, n_2, c_2, y]

$\rightarrow$ [true, n_1+$n2$, c_1+$n_2/2$, x]

El elemento x aparece a lo sumo c_1 veces en u_1 y a sumo n_2-c_2 veces en u_2. Aparece a lo sumo $n_2+(c_1\text{-}c_2) \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, y aparece a lo sumo $n_1+(c_2\text{-}c_1) \leq (n_1+n_2)/2$ veces en u_1+u_2
El único candidato es entonces x.

Ejemplo 2:

3,7,6,6,6 $\rightarrow$ [true,5,3,6]

4,4,6,4,4 $\rightarrow$ [true,5,4,4]

$\rightarrow$ [true,10,6,4]

Combinaciones de resultados. Caso 5a

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 > n_2$

Combinaciones de resultados. Caso 5a

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 > n_2$

u_1

u_2

Combinaciones de resultados. Caso 5a

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 > n_2$

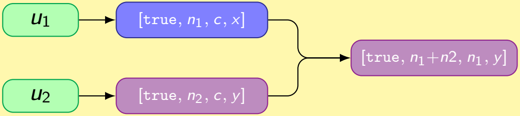
$u_1 \rightarrow [\texttt{true}, n_1, c, x]$

$u_2 \rightarrow [\texttt{true}, n_2, c, y]$

El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 \leq (n_1+n_2)/2$ veces en u_1+u_2.

Combinaciones de resultados. Caso 5a

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 > n_2$



El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 \leq (n_1+n_2)/2$ veces en u_1+u_2.
El único candidato es entonces y.

Combinaciones de resultados. Caso 5a

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 > n_2$

$u_1 \rightarrow$ [true, n_1, c, x]

$u_2 \rightarrow$ [true, n_2, c, y]

$\rightarrow$ [true, n_1+n2, n_1, y]

El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 \leq (n_1+n_2)/2$ veces en u_1+u_2.

El único candidato es entonces y.

Ejemplo 1:

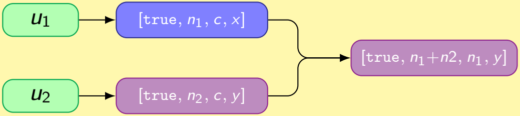
4,4,4 $\rightarrow$ [true,3,3,4]

4,1,1,1 $\rightarrow$ [true,4,3,1]

$\rightarrow$ [true,7,4,4]

Combinaciones de resultados. Caso 5a

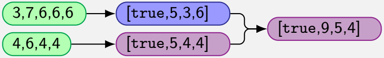
Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 > n_2$



El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 \geq (n_1+n_2)/2+1$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 \leq (n_1+n_2)/2$ veces en u_1+u_2.

El único candidato es entonces y.

Ejemplo 2:



Combinaciones de resultados. Caso 5b

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 = n_2$

Combinaciones de resultados. Caso 5b

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 = n_2$

u_1

u_2

Combinaciones de resultados. Caso 5b

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 = n_2$

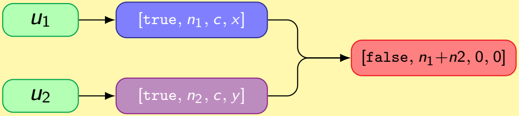
$u_1 \rightarrow [\texttt{true}, n_1, c, x]$

$u_2 \rightarrow [\texttt{true}, n_2, c, y]$

El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 = (n_1+n_2)/2$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 = (n_1+n_2)/2$ veces en u_1+u_2.

Combinaciones de resultados. Caso 5b

Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 = n_2$

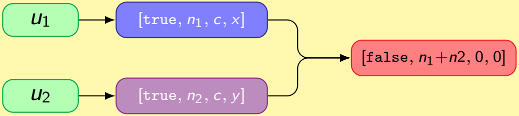


El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 = (n_1+n_2)/2$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 = (n_1+n_2)/2$ veces en u_1+u_2.

No hay entonces ningún candidato.

Combinaciones de resultados. Caso 5b

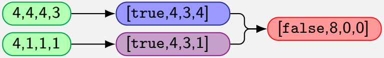
Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 = n_2$



El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 = (n_1+n_2)/2$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 = (n_1+n_2)/2$ veces en u_1+u_2.

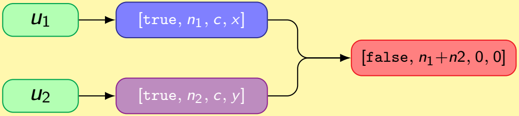
No hay entonces ningún candidato.

Ejemplo 1:



Combinaciones de resultados. Caso 5b

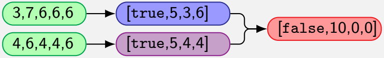
Hay diferentes candidatos en ambos arreglos, $c_1 = c_2$, $n_1 = n_2$



El elemento y aparece a lo sumo c veces en u_2 y a lo sumo n_1-c veces en u_1. Aparece a lo sumo $n_1+(c\text{-}c) = n_1 = (n_1+n_2)/2$ veces en u_1+u_2. Análogamente, x aparece a lo sumo $n_2 = (n_1+n_2)/2$ veces en u_1+u_2.

No hay entonces ningún candidato.

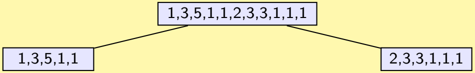
Ejemplo 2:



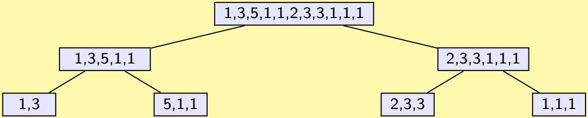
El algoritmo *pseudo-M*: un ejemplo

1,3,5,1,1,2,3,3,1,1,1

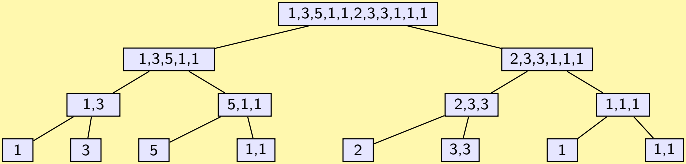
El algoritmo *pseudo-M*: un ejemplo



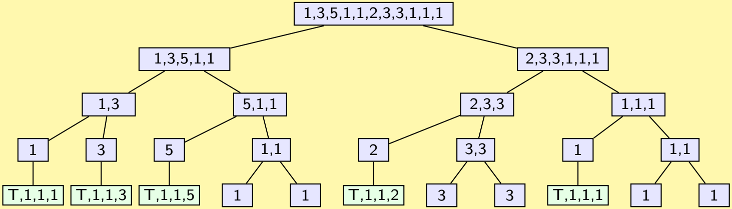
El algoritmo *pseudo-M*: un ejemplo



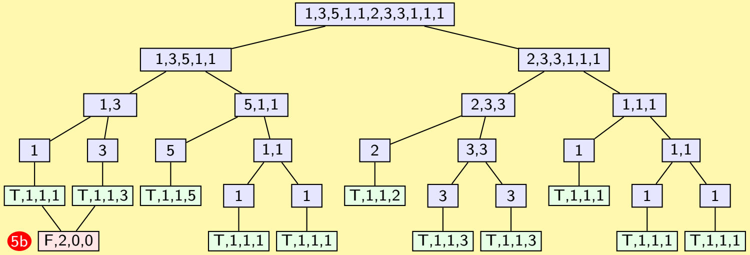
El algoritmo *pseudo-M*: un ejemplo



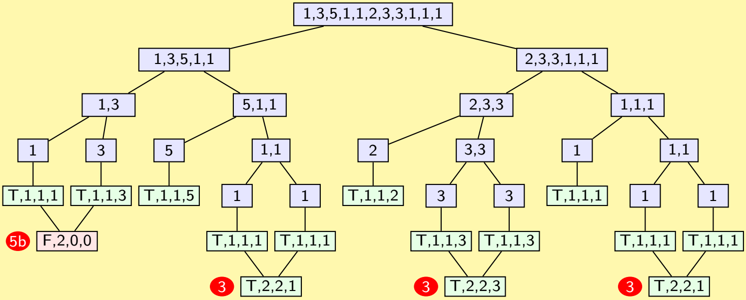
El algoritmo *pseudo-M*: un ejemplo



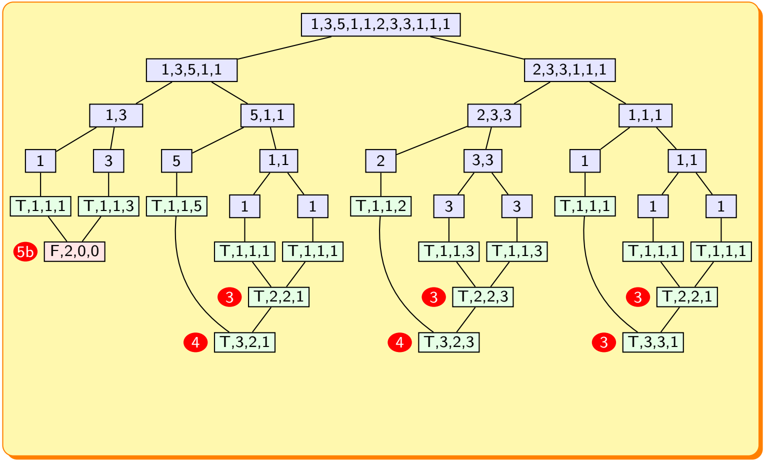
El algoritmo *pseudo-M*: un ejemplo



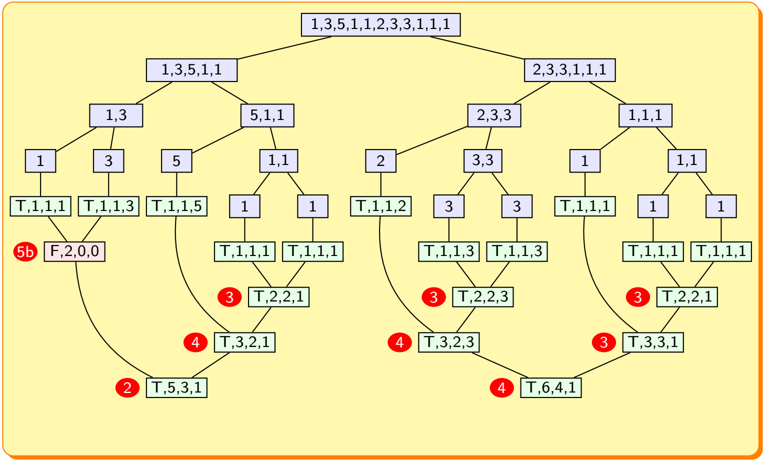
El algoritmo *pseudo-M*: un ejemplo



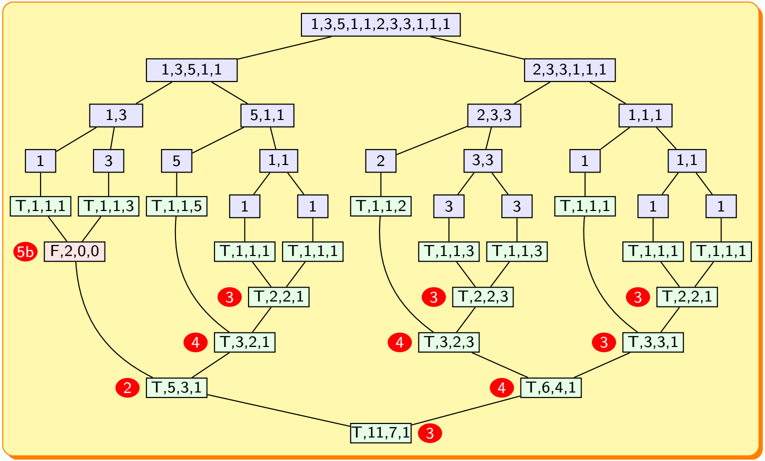
El algoritmo *pseudo-M*: un ejemplo



El algoritmo *pseudo-M*: un ejemplo



El algoritmo *pseudo-M*: un ejemplo



El algoritmo *pseudo-M*, parte 1

```
class obj {
    boolean r;
    int n;
    int c;
    int x;
}
obj Pseudo-M(int [] u[1..n], int i,j) {
    if (i == j) {                                   // Caso base
        return obj(true, 1, 1, u[i]);
    } else {
        obj o1 = Pseudo-M(u,i, (i+j)/2)
        obj o1 = Pseudo-M(u,(i+j)/2+1,j)
    }
    if (!o1.r && !o1.r) {                           // Caso 1: no hay candidatos
        return obj(false,o1.n+o2.n,0,0);
    } else if (o1.r && !o1.r) {                     // Caso 2a: un candidato (o1)
        return obj(true,o1.n+o2.n, o1.c+o2.n/2,o1.x)
    } else if (!o1.r && o1.r) {                     // Caso 2b: un candidato (o2)
        return obj(true,o1.n+o2.n, o2.c+o1.n/2,o2.x)
```

El algoritmo *pseudo-M*, parte 2

```
        } else if (o1.r && o2.r) {          // En adelante: diferentes candidatos
            if (o1.x == o1.x) {             // Caso 3: mismo candidato
                return (true,o1.n+o2.n, o1.c+o2.c, o1.x)
            } else if (o1.c > o2.c) {       // Caso 4a con o1
                return (true,o1.n+o2.n, o1.c-o2.c, o1.x)
            } else if (o1.c > o2.c) {       // Caso 4b con o2
                return (true,o1.n+o2.n, o2.c-o1.c, o2.x)
                                            // En adelante: diferente candidato, mismo c
            } else if (o1.n < o2.n) {       // Caso 5a1 con o1
                return (true,o1.n+o2.n, o1.c+o2.n/2, o1.x)
            } else if (o1.n < o2.n) {       // Caso 5a2 con o2
                return (true,o1.n+o2.n, o2.c+o1.n/2, o2.x)
            } else {                        // Caso 5b con todo igual
                return (false,o1.n+o2.n, 0,0)
            }
        }
    }
```

Complejidad de *Pseudo-M*

Complejidad de *Pseudo-M*

- Tenemos recurrencia con división. Para este algoritmo tenemos $a = 2$, $b = 2$ y $k = 0$.

Complejidad de *Pseudo-M*

- Tenemos recurrencia con división. Para este algoritmo tenemos $a = 2$, $b = 2$ y $k = 0$.
- Estamos en el caso $a = 2 > b^k = 2^0 = 1$. Por lo tanto, tenemos $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Complejidad de *Pseudo-M*

- Tenemos recurrencia con división. Para este algoritmo tenemos $a = 2$, $b = 2$ y $k = 0$.
- Estamos en el caso $a = 2 > b^k = 2^0 = 1$. Por lo tanto, tenemos $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$.
- Malas noticias: no hemos resuelto el problema del elemento mayoritario. Apenas si hemos encontrado un candidato.

Complejidad de *Pseudo-M*

- Tenemos recurrencia con división. Para este algoritmo tenemos $a = 2$, $b = 2$ y $k = 0$.
- Estamos en el caso $a = 2 > b^k = 2^0 = 1$. Por lo tanto, tenemos $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$.
- Malas noticias: no hemos resuelto el problema del elemento mayoritario. Apenas si hemos encontrado un candidato.
- ¿Podemos utilizar el algoritmo *Pseudo-M* para resolver el problema del elemento mayoritario?

Complejidad de *Pseudo-M*

- Tenemos recurrencia con división. Para este algoritmo tenemos $a = 2$, $b = 2$ y $k = 0$.
- Estamos en el caso $a = 2 > b^k = 2^0 = 1$. Por lo tanto, tenemos $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$.
- Malas noticias: no hemos resuelto el problema del elemento mayoritario. Apenas si hemos encontrado un candidato.
- ¿Podemos utilizar el algoritmo *Pseudo-M* para resolver el problema del elemento mayoritario?
- Sí; basta encontrar la cantidad de veces que el candidato aparece en u. Esto tiene complejidad $\Theta(n)$.

Complejidad de *Pseudo-M*

- Tenemos recurrencia con división. Para este algoritmo tenemos $a = 2$, $b = 2$ y $k = 0$.
- Estamos en el caso $a = 2 > b^k = 2^0 = 1$. Por lo tanto, tenemos $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$.
- Malas noticias: no hemos resuelto el problema del elemento mayoritario. Apenas si hemos encontrado un candidato.
- ¿Podemos utilizar el algoritmo *Pseudo-M* para resolver el problema del elemento mayoritario?
- Sí; basta encontrar la cantidad de veces que el candidato aparece en u. Esto tiene complejidad $\Theta(n)$.
- Por lo tanto, tendremos complejidad total $\Theta(n) + \Theta(n) = 2\Theta(n) = \Theta(n)$.

De regreso al problema del elemento mayoritario.

```
int Algoritmo M(int [] u[0..n-1]) {
    obj o= Pseudo-M(u)
    if (!o.r {                          // No hay candidato
        return -99
    } else if Majority(u, x) {
        return x
    } else {
        return -99
    }
}
boolean Algoritmo Majority(int [] u[0..n-1], int x) {
    int c = 0
    for (i = 0; i < n; i++) {
        if (x == u[i]) {
            c++
        }
    }
    if (c > n/2) {
        return true
    } else {
        return false
    }
}
```

De regreso al problema del elemento mayoritario.

```
      int Algoritmo M(int [] u[0..n-1]) {
Θ(n)      obj o= Pseudo-M(u)
          if (!o.r {                        // No hay candidato
              return -99
          } else if Majority(u, x) {
              return x
          } else {
              return -99
          }
      }
      boolean Algoritmo Majority(int [] u[0..n-1], int x) {
          int c = 0
          for (i = 0; i < n; i++) {
              if (x == u[i]) {
                  c++
Θ(n)          }
          }
          if (c > n/2) {
              return true
          } else {
              return false
          }
      }
```

Las torres de Hanoi

Las torres de Hanoi

- Se trata de un problema clásico de recurrencia.

Las torres de Hanoi

- Se trata de un problema clásico de recurrencia.
- Hay tres clavijas, A, B y C, y n discos perforados de diferentes diámetros que se insertan en ellas. Los llamamos $1, 2, \ldots, n$ (de menor a mayor.)

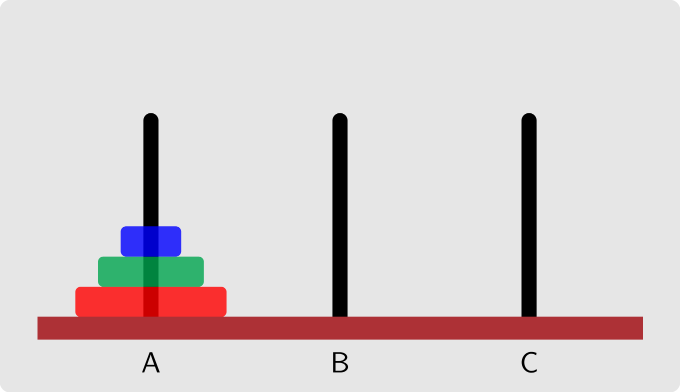
Las torres de Hanoi

- Se trata de un problema clásico de recurrencia.
- Hay tres clavijas, A, B y C, y n discos perforados de diferentes diámetros que se insertan en ellas. Los llamamos $1, 2, \ldots, n$ (de menor a mayor.)
- Al comienzo del juego, todos los discos están en A apilados de mayor a menor (es decir, al fondo de la pila está n, luego $n - 1$ y así hasta llegar a 1 que está en el tope de la pila.)

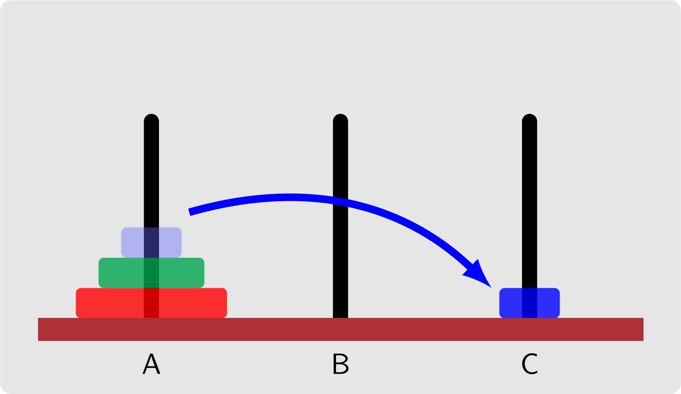
Las torres de Hanoi

- Se trata de un problema clásico de recurrencia.
- Hay tres clavijas, A, B y C, y n discos perforados de diferentes diámetros que se insertan en ellas. Los llamamos $1, 2, \ldots, n$ (de menor a mayor.)
- Al comienzo del juego, todos los discos están en A apilados de mayor a menor (es decir, al fondo de la pila está n, luego $n-1$ y así hasta llegar a 1 que está en el tope de la pila.)
- El objetivo del juego es mover todos los discos a la clavija C utilizando la clavija B como auxiliar y respetando las siguientes reglas:
  - Sólo se puede mover un disco por vez de una clavija a otra.
  - Sólo el tope de cada pila puede desplazarse.
  - Nunca puede haber un disco apilado sobre uno de diámetro menor.

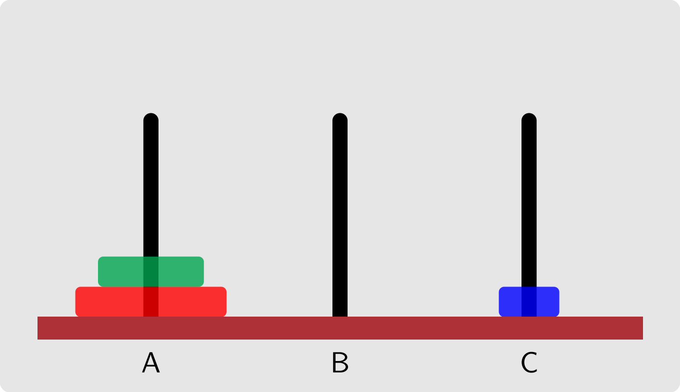
Las torres de Hanoi. Un ejemplo



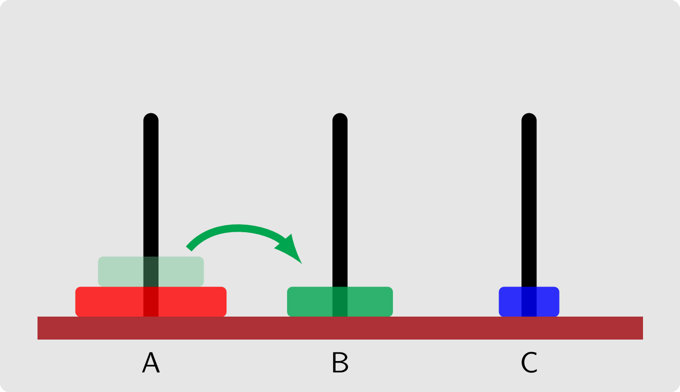
Las torres de Hanoi. Un ejemplo



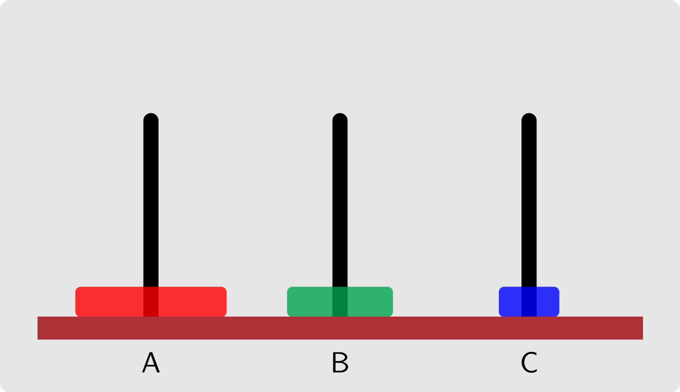
Las torres de Hanoi. Un ejemplo



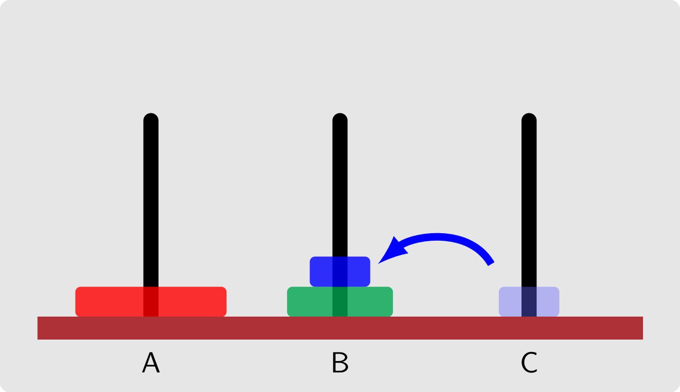
Las torres de Hanoi. Un ejemplo



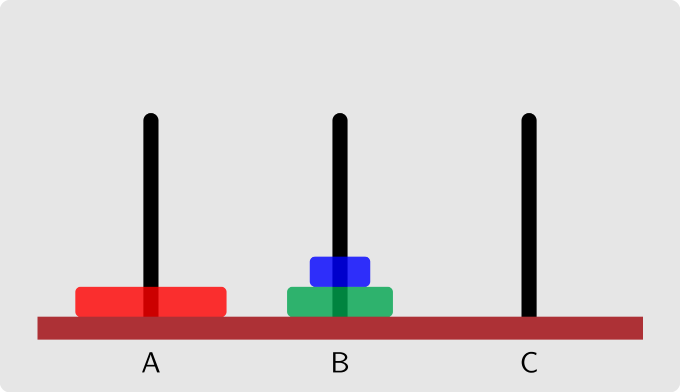
Las torres de Hanoi. Un ejemplo



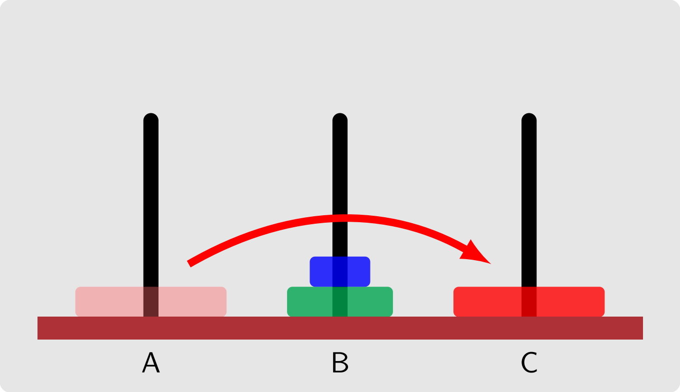
Las torres de Hanoi. Un ejemplo



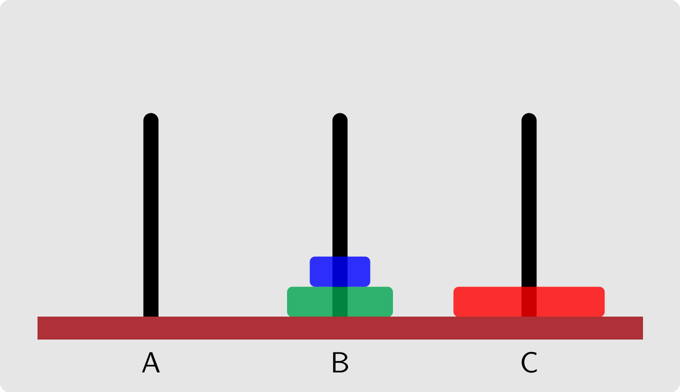
Las torres de Hanoi. Un ejemplo



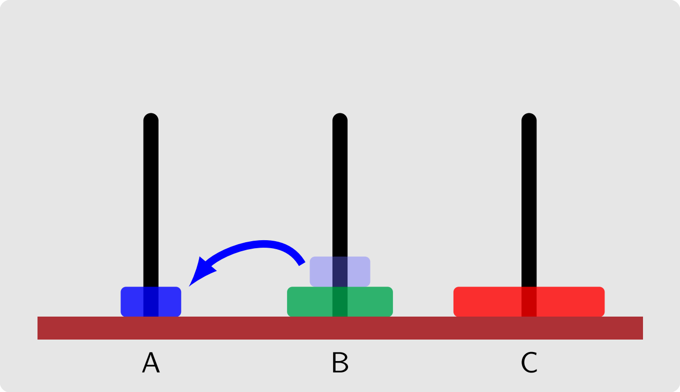
Las torres de Hanoi. Un ejemplo



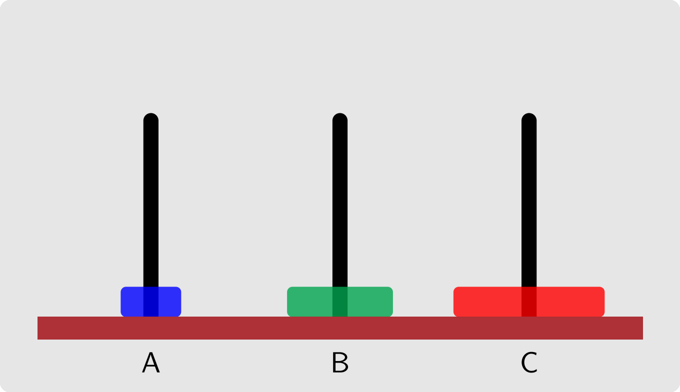
Las torres de Hanoi. Un ejemplo



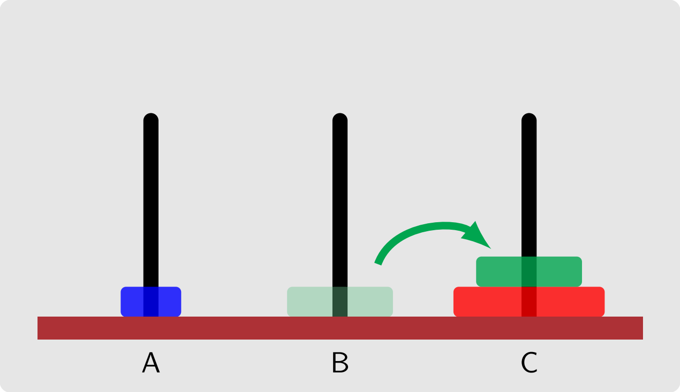
Las torres de Hanoi. Un ejemplo



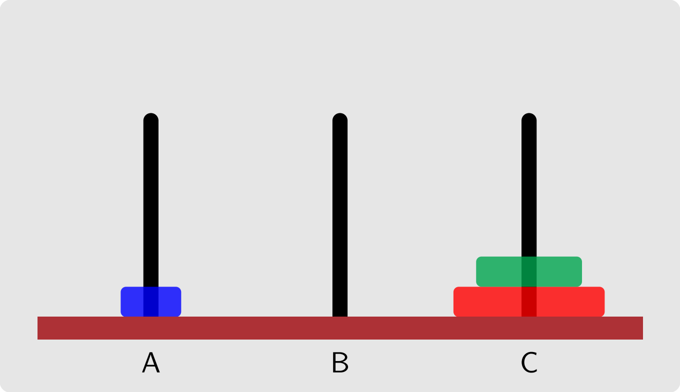
Las torres de Hanoi. Un ejemplo



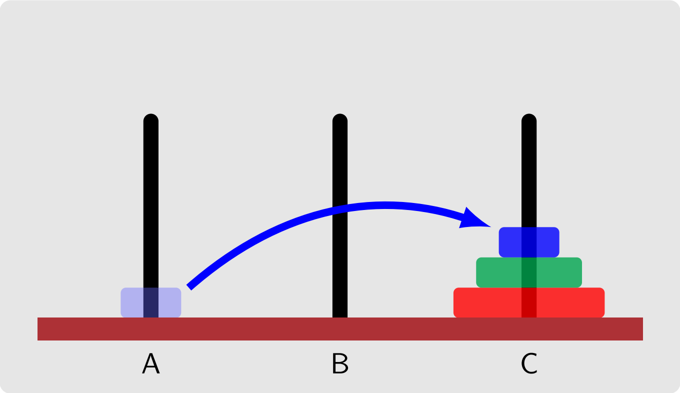
Las torres de Hanoi. Un ejemplo



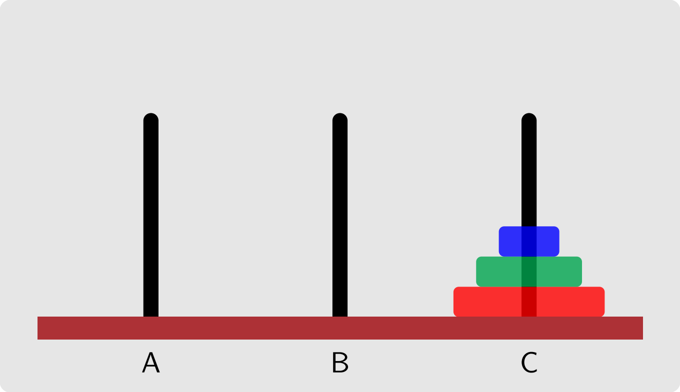
Las torres de Hanoi. Un ejemplo



Las torres de Hanoi. Un ejemplo



Las torres de Hanoi. Un ejemplo



Las torres de Hanoi. Estrategia

Las torres de Hanoi. Estrategia

- Observe que al comienzo del juego sólo hay dos movimientos posibles: trasladar 1 de A a B o a C

Las torres de Hanoi. Estrategia

- Observe que al comienzo del juego sólo hay dos movimientos posibles: trasladar 1 de A a B o a C
- Si se tiene un número par de elementos, hay que comenzar por mover 1 a B; sino a C.

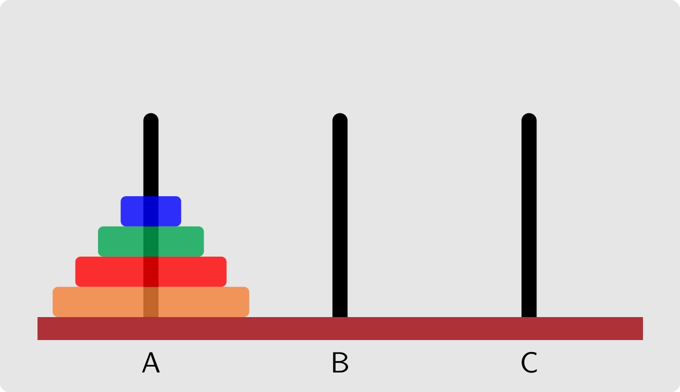
Las torres de Hanoi. Estrategia

- Observe que al comienzo del juego sólo hay dos movimientos posibles: trasladar 1 de A a B o a C
- Si se tiene un número par de elementos, hay que comenzar por mover 1 a B; sino a C.
- Observemos lo siguiente: si tenemos que desplazar n discos de A a C (denotamos este problema $Hanoi(n, A, B, C)$), podemos comenzar resolviendo el problema de trasladar $n-1$ discos de A a B ($Hanoi(n-1, A, C, B)$), luego pasar el disco restante de A a C y finalmente resolver el problema de trasladar la pila de B a C ($Hanoi(n, B, A, C)$).

Las torres de Hanoi. Estrategia

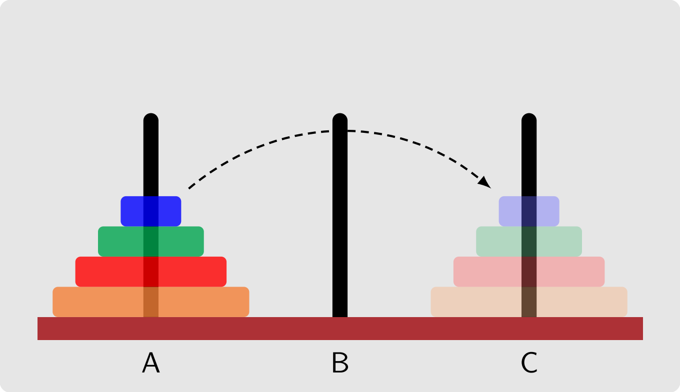
- Observe que al comienzo del juego sólo hay dos movimientos posibles: trasladar 1 de A a B o a C
- Si se tiene un número par de elementos, hay que comenzar por mover 1 a B; sino a C.
- Observemos lo siguiente: si tenemos que desplazar n discos de A a C (denotamos este problema $Hanoi(n, A, B, C)$), podemos comenzar resolviendo el problema de trasladar $n-1$ discos de A a B ($Hanoi(n-1, A, C, B)$), luego pasar el disco restante de A a C y finalmente resolver el problema de trasladar la pila de B a C ($Hanoi(n, B, A, C)$).
- El caso base es cuando tenemos un único disco ($Hanoi(1, B, A, C)$), que se resuelve simplemente moviendo el disco de B a C. Esto sugiere un esquema de recurrencia con substracción.

Las torres de Hanoi. Estrategia



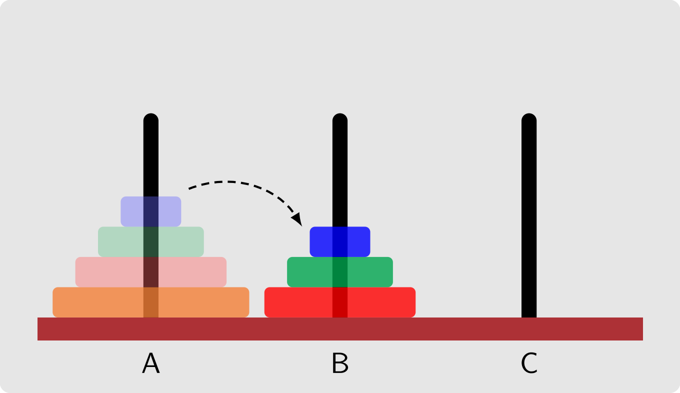
Queremos resolver *Hanoi*(4,A,B,C)

Las torres de Hanoi. Estrategia



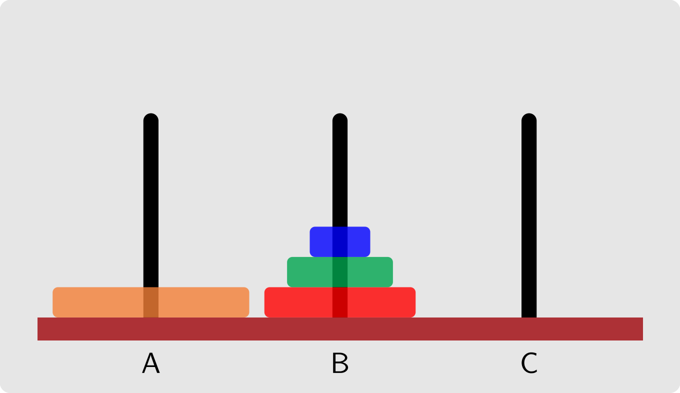
Queremos resolver *Hanoi*(4,A,B,C)

Las torres de Hanoi. Estrategia



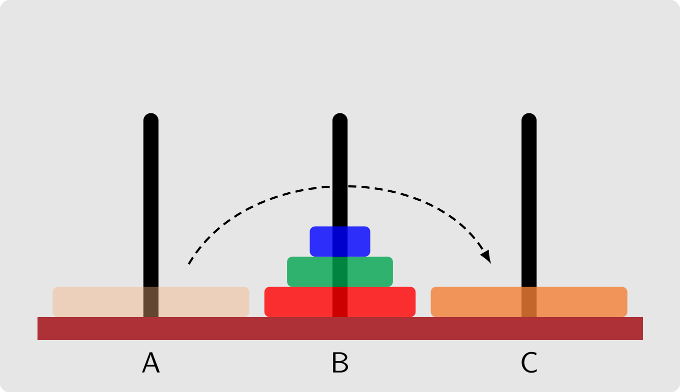
Comenzamos por resolver *Hanoi*(3,A,C,B)

Las torres de Hanoi. Estrategia



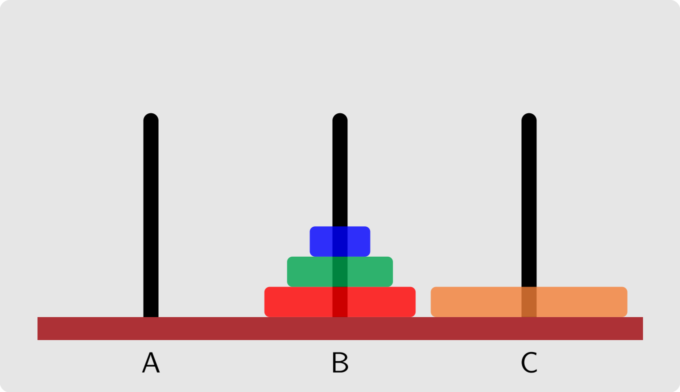
Comenzamos por resolver *Hanoi*(3,A,C,B)

Las torres de Hanoi. Estrategia



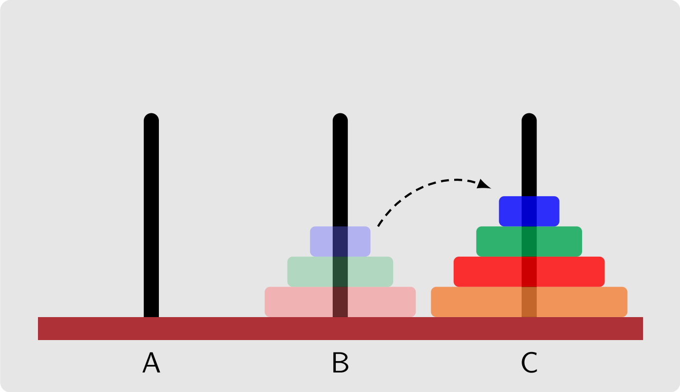
Pasamos el disco restante a C

Las torres de Hanoi. Estrategia



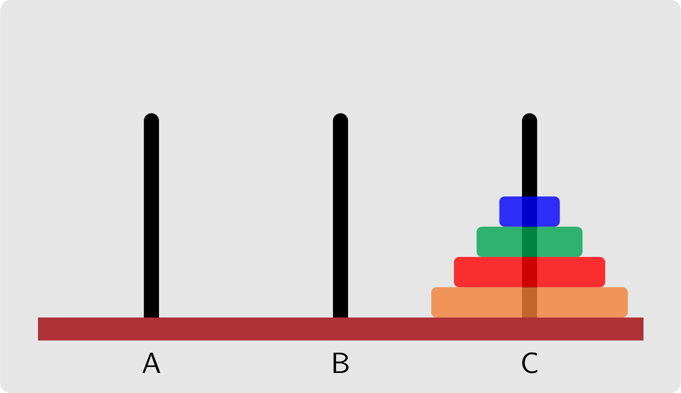
Pasamos el disco restante a C

Las torres de Hanoi. Estrategia



Y resolvemos *Hanoi*(3,B,A,C)

Las torres de Hanoi. Estrategia



Y resolvemos *Hanoi*(3,B,A,C)

Algoritmo de las torres de Hanoi

```
void Algoritmo Hanoi(int n, poste A, B, C) {
    if (n == 1) {
        move A → C
    } else {
        Hanoi(n-1, A, C, B)
        move A → C
        Hanoi(n-1, B, A, C)
    }
}
```

Algoritmo de las torres de Hanoi

```
void Algoritmo Hanoi(int n, poste A, B, C) {
    if (n == 1) {
        move A → C
    } else {
        Hanoi(n-1, A, C, B)
        move A → C
        Hanoi(n-1, B, A, C)
    }
}
```

Complejidad: se trata de un caso de substracción con $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en el caso $a > 1$ y tenemos $\Theta(n^k a^{n \div b}) = \Theta(n^0 2^{n \div 1}) = \Theta(2^n)$.

El torneo de Bridge

El torneo de Bridge

- El Club Social de Fraile Muerto ha decidido organizar su torneo anual de bridge. Como es tradicional, el número n de parejas participantes es una potencia de 2. Cada pareja debe competir exactamente una vez con todas las demás (*round-robin* y cada pareja juega exactamente un match diario.

El torneo de Bridge

- El Club Social de Fraile Muerto ha decidido organizar su torneo anual de bridge. Como es tradicional, el número n de parejas participantes es una potencia de 2. Cada pareja debe competir exactamente una vez con todas las demás (*round-robin* y cada pareja juega exactamente un match diario.
- Queremos diseñar un algoritmo que permita que el torneo concluya en $n - 1$ días.

El torneo de Bridge

- El Club Social de Fraile Muerto ha decidido organizar su torneo anual de bridge. Como es tradicional, el número n de parejas participantes es una potencia de 2. Cada pareja debe competir exactamente una vez con todas las demás (*round-robin* y cada pareja juega exactamente un match diario.
- Queremos diseñar un algoritmo que permita que el torneo concluya en $n-1$ días.
- Se puede ver la solución como una matriz T en la que cada fila representa una pareja y cada columna una fecha del torneo. Así, por ejemplo $T[2,3]=5$ significa que la pareja 2 se enfrenta con la pareja 5 en la cuarta fecha (comenzamos a contar desde 0.)

El torneo de Bridge

- El Club Social de Fraile Muerto ha decidido organizar su torneo anual de bridge. Como es tradicional, el número n de parejas participantes es una potencia de 2. Cada pareja debe competir exactamente una vez con todas las demás (*round-robin* y cada pareja juega exactamente un match diario.
- Queremos diseñar un algoritmo que permita que el torneo concluya en $n-1$ días.
- Se puede ver la solución como una matriz T en la que cada fila representa una pareja y cada columna una fecha del torneo. Así, por ejemplo $T[2,3]=5$ significa que la pareja 2 se enfrenta con la pareja 5 en la cuarta fecha (comenzamos a contar desde 0.)
- La matriz tendrá entonces n filas (una por cada pareja participante) y $n-1$ columnas (una para cada ronda del torneo.)

Estrategia para el problema del fixture

Estrategia para el problema del fixture

- Partimos de $n = 2^k$ con $k \geq 1$. El caso base es $k = 1$, donde hay dos parejas que se enfrentan entre sí una vez. El torneo para $n = 2$ acaba en un día.

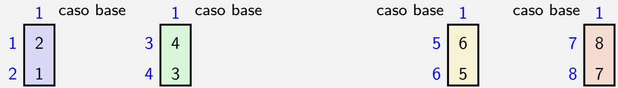
Estrategia para el problema del fixture

- Partimos de $n = 2^k$ con $k \geq 1$. El caso base es $k = 1$, donde hay dos parejas que se enfrentan entre sí una vez. El torneo para $n = 2$ acaba en un día.
- Si $k > 1$, se divide en dos grupos de $n/2 = 2^{k-1}$ parejas. Se elabora un sub-calendario para cada grupo.

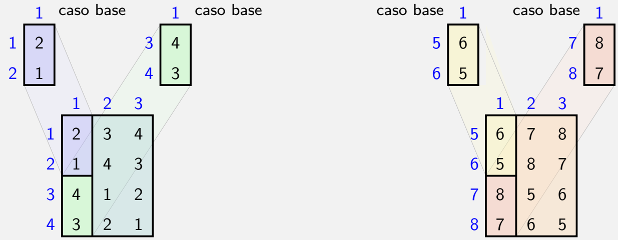
Estrategia para el problema del fixture

- Partimos de $n = 2^k$ con $k \geq 1$. El caso base es $k = 1$, donde hay dos parejas que se enfrentan entre sí una vez. El torneo para $n = 2$ acaba en un día.
- Si $k > 1$, se divide en dos grupos de $n/2 = 2^{k-1}$ parejas. Se elabora un sub-calendario para cada grupo.
- Finalmente se cruzan las parejas de cada grupo. La pareja 1 enfrenta en días sucesivos a las parejas $k+1, k+2, \dots, n$; la pareja 2 enfrenta en días sucesivos a las parejas $k+2, k+3, \dots, n, k+1$. Finalmente, la pareja $k-1$ enfrenta en días sucesivos a las parejas $n, k+1, \dots, k+n-1$.

Un ejemplo de fixture



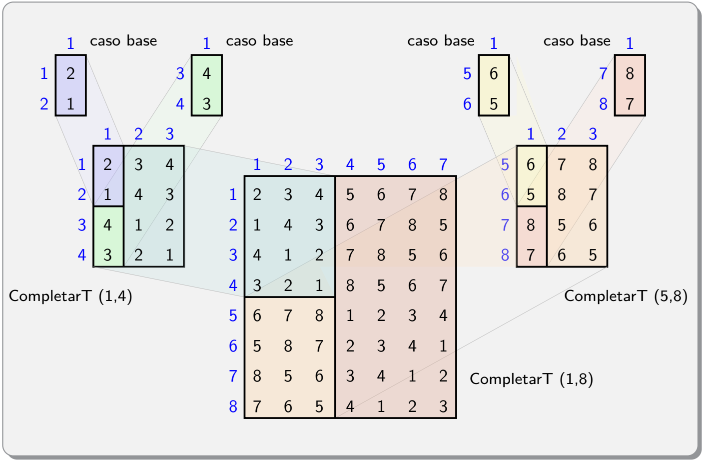
Un ejemplo de fixture



CompletarT (1,4)

CompletarT (5,8)

Un ejemplo de fixture



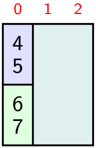
Un algoritmo para producir el fixture

```
Algoritmo Fixture (int T[1..n, 0..n − 2], int inf, sup)
    if inf = sup − 1 {
        T[inf, 0] ← sup
        T[sup, 0] ← inf
    } else {
        mid = (inf + sup)/2
        Fixture(T, inf, mid)
        Fixture(T, mid + 1, sup)
        CompletarT(T, inf, sup)
    }
```

El algoritmo *CompletarT*

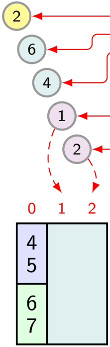
```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j++) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i++)
        for (j = FirstDay; j ≤ LastDay; j++) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
```

El algoritmo *CompletarT*



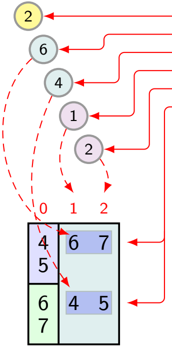

```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j++) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i++)
        for (j = FirstDay; j ≤ LastDay; j++) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
```

El algoritmo *CompletarT*



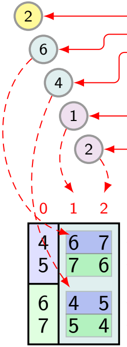

```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j++) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i++)
        for (j = FirstDay; j ≤ LastDay; j++) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
```

El algoritmo *CompletarT*




```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j++) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i++)
        for (j = FirstDay; j ≤ LastDay; j++) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
```

El algoritmo *CompletarT*




```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j++) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i++)
        for (j = FirstDay; j ≤ LastDay; j++) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
```

Complejidad del algoritmo *CompletarT*

```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j++) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i++) {
        for (j = FirstDay; j ≤ LastDay; j++) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
    }
```

Complejidad del algoritmo *CompletarT*

```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j + +) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i + +) {
        for (j = FirstDay; j ≤ LastDay; j + +) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
    }
```

$\Theta(n)$

$\Theta(n)$ $\Theta(n)$

Complejidad del algoritmo *CompletarT*

$\Theta(n^2)$

```
Algoritmo CompletarT (int [] T[0..n-1, 0..n-2], int First, Last)
    int Len = (Last - First + 1) / 2
    int FirstAdversary = First + Len
    int SecondAdversary = First
    int FirstDay = Len - 1
    int LastDay = FirstDay + Len - 1
    for (j = FirstDay; j ≤ LastDay; j++) {
        T[First, j] = FirstAdversary
        T[First + Len, j] = SecondAdversary
        FirstAdversary++
        SecondAdversary++
    }
    for (i = FirstDay + 1; i ≤ Last − Len; i++) {
        for (j = FirstDay; j ≤ LastDay; j++) {
            if (j == LastDay) {
                T[i][j] = T[i − 1][FirstDay]
                T[i + Len][j] = T[i + Len − 1][FirstDay]
            } else {
                T[i][j] = T[i − 1][j + 1]
                T[i + Len][j] = T[i + Len − 1][j + 1]
            }
        }
    }
```

$\Theta(n)$

$\Theta(n)$ $\Theta(n)$

Complejidad de *Fixture*

Complejidad de *Fixture*

- Tenemos un caso de recurrencia por división. Para determinar k, debemos analizar la complejidad del procedimiento *CompletarT*.

Complejidad de *Fixture*

- Tenemos un caso de recurrencia por división. Para determinar k, debemos analizar la complejidad del procedimiento *CompletarT*.
- En *CompletarT* tenemos un primer ciclo que en el peor de los casos se ejecuta n veces y un par de ciclos anidados que en el peor de los casos se ejecutan $\left(\frac{n}{2}\right)^2$ veces. O sea, $\Theta(n) + \Theta(n^2) = \Theta(n^2)$. Por lo tanto, $k = 2$.

Complejidad de *Fixture*

- Tenemos un caso de recurrencia por división. Para determinar k, debemos analizar la complejidad del procedimiento *CompletarT*.
- En *CompletarT* tenemos un primer ciclo que en el peor de los casos se ejecuta n veces y un par de ciclos anidados que en el peor de los casos se ejecutan $\left(\frac{n}{2}\right)^2$ veces. O sea, $\Theta(n) + \Theta(n^2) = \Theta(n^2)$. Por lo tanto, $k = 2$.
- Por otro lado, tenemos $a = 2$, $b = 2$. Estamos en el caso $a = 2 < b^k = 2^2$ y en consecuencia tenemos $\Theta(n^k) = \Theta(n^2)$.

El problema de la suma parcial máxima (MPS) 1

El problema de la suma parcial máxima (MPS) 1

- El *MPS* problema se define cono sigue: dado un arreglo $A[0..n\text{-}1]$ de enteros, encontrar valores de i y j con $1 \leq i \leq j \leq n-1$ tales que la suma

$$\sum_{k=i}^{j} A[k]$$

sea máxima.

El problema de la suma parcial máxima (MPS) 1

- El *MPS* problema se define cono sigue: dado un arreglo $A[0..n\text{-}1]$ de enteros, encontrar valores de i y j con $1 \leq i \leq j \leq n-1$ tales que la suma

$$\sum_{k=i}^{j} A[k]$$

  sea máxima.
- Por ejemplo, para el arreglo $[3, -5, 5, 9, 7, -10, 4]$, la solución del *MPS* es $i = 3$ y $j = 5$ (suma 21).

El problema de la suma parcial máxima (MPS) 1

- El *MPS* problema se define cono sigue: dado un arreglo $A[0..n\text{-}1]$ de enteros, encontrar valores de i y j con $1 \leq i \leq j \leq n-1$ tales que la suma

$$\sum_{k=i}^{j} A[k]$$

sea máxima.

- Por ejemplo, para el arreglo $[3, -5, 5, 9, 7, -10, 4]$, la solución del *MPS* es $i = 3$ y $j = 5$ (suma 21).
- Obviamente, si todos los números del arreglo son positivos, la solución del *MPS* es la suma de todos los números, o sea que $i = 0$ y $j = n\text{-}1$; si todos los números son negativos, la solución del *MPS* sería el máximo número.

El problema de la suma parcial máxima (MPS) 2

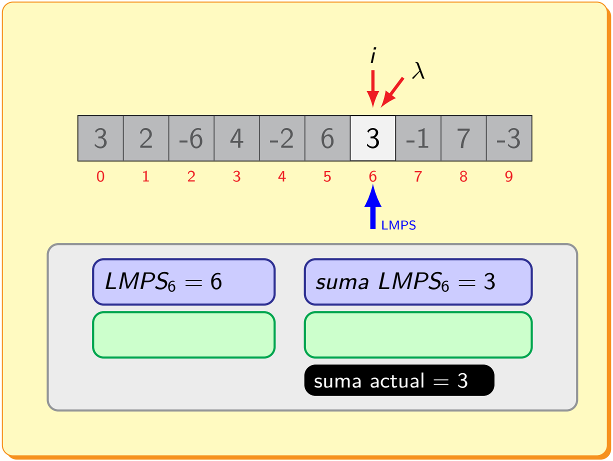
El problema de la suma parcial máxima (MPS) 2

- Para poder encontrar una solución eficiente para este problema, definimos dos sub-problemas:
  - El problema $LMPS_\lambda$, que consiste en encontrar un valor i, $0 \leq i \leq \lambda$, tal que la suma
  $$\sum_{k=i}^{\lambda} A[k]$$
  es máxima (la posición izquierda λ queda fija.)
  - El problema $RMPS_\rho$ que consiste en encontrar un valor j, $\rho \leq j \leq n\text{-}1$, tal que la suma
  $$\sum_{k=\rho}^{j} A[k]$$
  es máxima (la posición derecha ρ queda fija.)

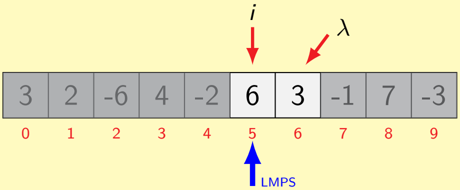
Los problemas $LMPS_\lambda$ y $RMPS_\rho$

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

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



Los problemas $LMPS_\lambda$ y $RMPS_\rho$

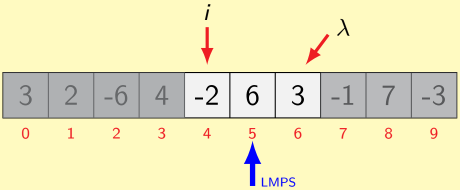


$LMPS_6 = 5$

$suma\ LMPS_6 = 9$

suma actual $= 9$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$

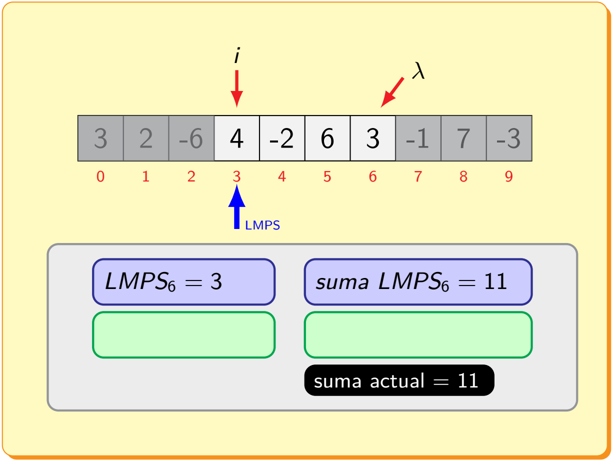


$LMPS_6 = 5$

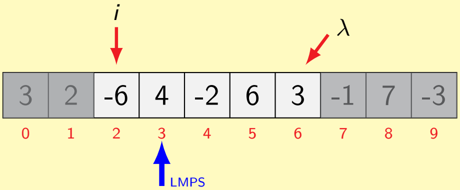
$suma\ LMPS_6 = 9$

suma actual $= 7$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



Los problemas $LMPS_\lambda$ y $RMPS_\rho$

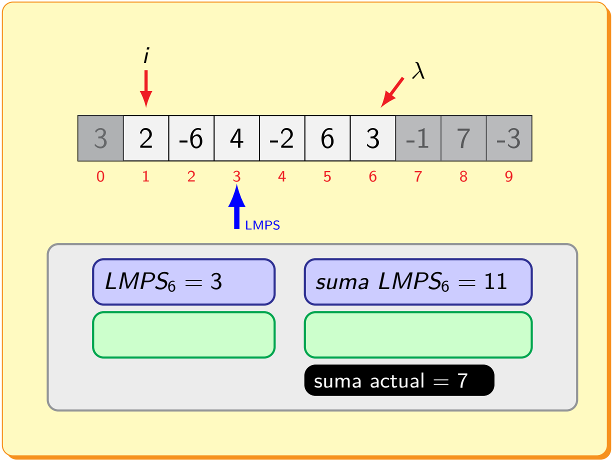


$LMPS_6 = 3$

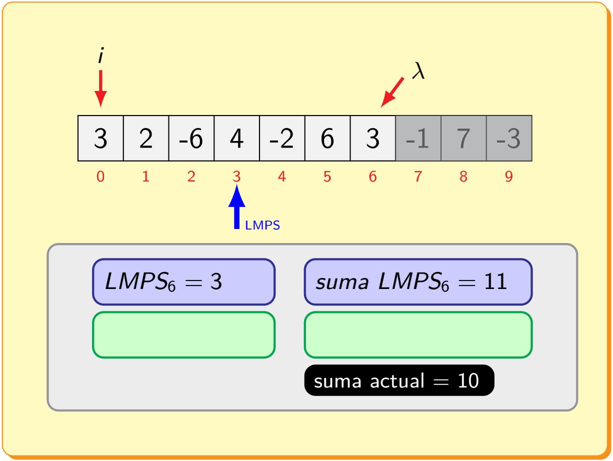
$suma\ LMPS_6 = 11$

suma actual $= 5$

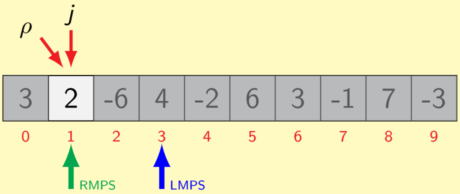
Los problemas $LMPS_\lambda$ y $RMPS_\rho$



Los problemas $LMPS_\lambda$ y $RMPS_\rho$



Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

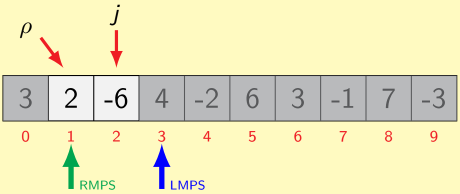
$suma\ LMPS_6 = 11$

$RMPS_1 = 1$

$suma\ RMPS_6 = 2$

suma actual $= 2$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

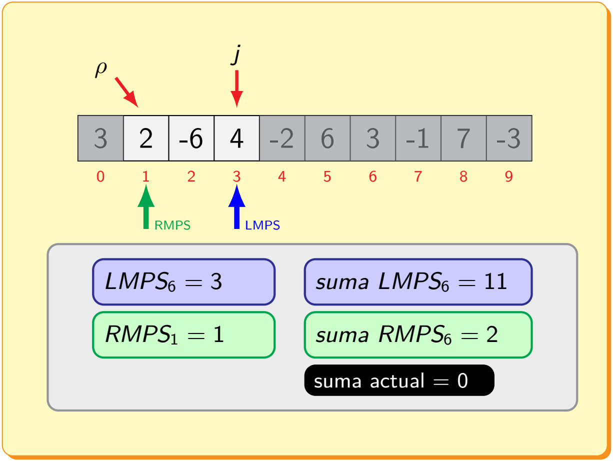
$suma\ LMPS_6 = 11$

$RMPS_1 = 1$

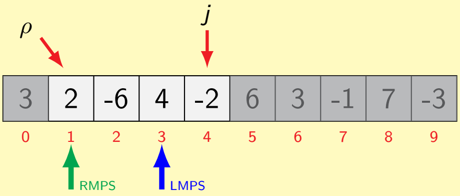
$suma\ RMPS_6 = 2$

suma actual = -4

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

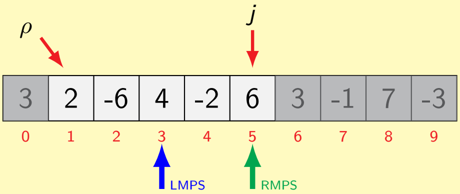
$RMPS_1 = 1$

$suma\ LMPS_6 = 11$

$suma\ RMPS_6 = 2$

suma actual = -2

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

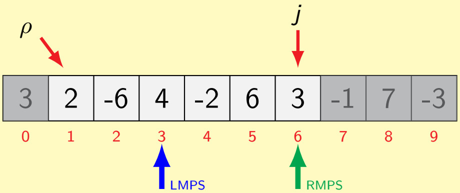
$suma\ LMPS_6 = 11$

$RMPS_1 = 5$

$suma\ RMPS_6 = 4$

suma actual $= 4$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

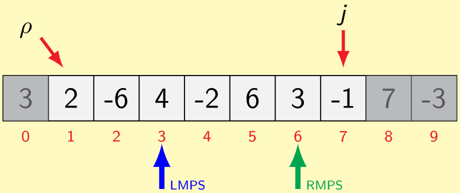
$suma\ LMPS_6 = 11$

$RMPS_1 = 6$

$suma\ RMPS_6 = 7$

suma actual $= 7$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

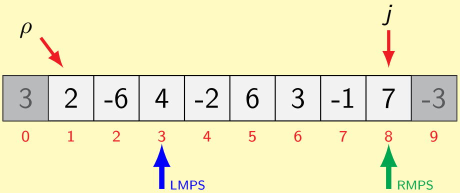
$suma\ LMPS_6 = 11$

$RMPS_1 = 6$

$suma\ RMPS_6 = 7$

suma actual $= 6$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

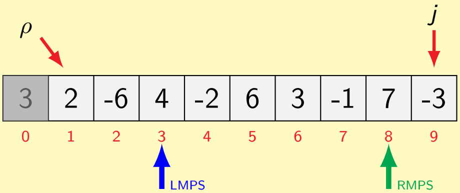
$RMPS_1 = 8$

$suma\ LMPS_6 = 11$

$suma\ RMPS_6 = 13$

suma actual $= 13$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

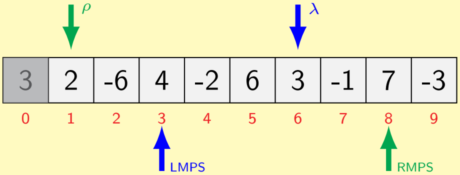
$suma\ LMPS_6 = 11$

$RMPS_1 = 8$

$sum\ RMPS_6 = 13$

suma actual $= 10$

Los problemas $LMPS_\lambda$ y $RMPS_\rho$



$LMPS_6 = 3$

$suma\ LMPS_6 = 11$

$RMPS_1 = 8$

$sum\ RMPS_6 = 13$

Algunas consideraciones estratégicas

Algunas consideraciones estratégicas

- Los métodos $LMPS_\lambda$ y $RMPS_\rho$ están ambos en $\mathcal{O}(n)$.

Algunas consideraciones estratégicas

- Los métodos $LMPS_\lambda$ y $RMPS_\rho$ están ambos en $\mathcal{O}(n)$.
- Una posible estrategia: calcular para cada posición i del arreglo $LMPS_i$ o $RMPS_i$ y quedarse con el máximo.

Algunas consideraciones estratégicas

- Los métodos $LMPS_\lambda$ y $RMPS_\rho$ están ambos en $\mathcal{O}(n)$.
- Una posible estrategia: calcular para cada posición i del arreglo $LMPS_i$ o $RMPS_i$ y quedarse con el máximo.
- Esta estrategia resolvería el problema en MPS problema en $\mathcal{O}(n^2)$.

Algunas consideraciones estratégicas

- Los métodos $LMPS_\lambda$ y $RMPS_\rho$ están ambos en $\mathcal{O}(n)$.
- Una posible estrategia: calcular para cada posición i del arreglo $LMPS_i$ o $RMPS_i$ y quedarse con el máximo.
- Esta estrategia resolvería el problema en MPS problema en $\mathcal{O}(n^2)$.
- No obstante, es posible encontrar una estrategia mejor utilizando *divide & conquer*.

Algunas consideraciones estratégicas

- Los métodos $LMPS_\lambda$ y $RMPS_\rho$ están ambos en $\mathcal{O}(n)$.
- Una posible estrategia: calcular para cada posición i del arreglo $LMPS_i$ o $RMPS_i$ y quedarse con el máximo.
- Esta estrategia resolvería el problema en MPS problema en $\mathcal{O}(n^2)$.
- No obstante, es posible encontrar una estrategia mejor utilizando *divide & conquer*.
- ¿Cómo? Piénselo. Queda como ejercicio.

Algunas consideraciones estratégicas

- Los métodos $LMPS_\lambda$ y $RMPS_\rho$ están ambos en $\mathcal{O}(n)$.
- Una posible estrategia: calcular para cada posición i del arreglo $LMPS_i$ o $RMPS_i$ y quedarse con el máximo.
- Esta estrategia resolvería el problema en MPS problema en $\mathcal{O}(n^2)$.
- No obstante, es posible encontrar una estrategia mejor utilizando *divide & conquer*.
- ¿Cómo? Piénselo. Queda como ejercicio.
- Suponga que conoce MPS para cada una de las mitades de un arreglo. Imagine también que se sabe que un elemento $A[j]$ está en el MPS. ¿Cómo se podría usar este conocimiento?

Ejercicios propuestos 1

1. En este ejercicio consideramos una función *monótonamente decreciente* $f : \mathbb{N} \to \mathbb{Z}$, esto es, una función definida sobre los números naturales que devuelve valores enteros, de manera que es $f(i) > f(i+1))$. Asumiendo que podemos evaluar f en cualquier punto i en tiempo constante, queremos encontrar $n = \min\{i \in \mathbb{N} | f(i) \leq 0\}$. En otras palabras, queremos encontrar el valor en el que f se vuelve negativa.

   Por supuesto, es posible resolver el problema en tiempo $\mathcal{O}(n)$ evaluando $f(1)$, $f(2)$, $f(3)$, $\ldots f(n)$. Describa un algoritmo que lo resuelva en $\mathcal{O}(\log n)$.

   Ayuda: evalúe f en $\mathcal{O}(\log n)$ valores $x \leq n$ cuidadosamente elegidos y tal vez en algún valores entre n y $2n$ — pero tenga en cuenta que usted no conoce el valor n inicialmente.

Ejercicios propuestos 2

2. Se le da un arreglo infinito A en el que las primeras n posiciones contienen enteros ordenados y el resto de las posiciones contienen ∞. Usted no conoce el valor de n. Describa un algoritmo que recibe como entrada un entero x y encuentra la posición del arreglo que contiene a x, si tal posición existe, en tiempo $\mathcal{O}(\log n)$.

   Si la idea de un arreglo infinito no es de su agrado, puede asumir como opción que el arreglo tiene longitud n, pero usted no conoce este valor (ni cuenta, por supuesto, con un método que se lo devuelva) y que la implementación del arreglo en su lenguaje de programación devuelve el mensaje de error ∞ siempre que se trata de acceder a un elemento $A[i]$ con $i > n$.

   Ayuda: puede buscar inspiración en la estrategia del ejercicio anterior.

Ejercicios propuestos 3

3. Una variante del ejercicio previo. Se le da un arreglo infinito A en el que las primeras n posiciones ontienen números enteros en cualquier orden y el resto de ls posiciones contienen ∞. Encentre el número n en tiempo $\mathcal{O}(\log n)$.

4. Suponga que nos dan un arreglo $A[0..n\text{-}1]$ orenado de enteros que ha sido *desplazado circularmente* k posiciones hacia la derecha. Por ejemplo, $[35, 42, 5, 15, 27, 29]$ es un arreglo ordenado desplazado circularmente $k = 2$ posiciones, mientras que $[27, 29, 35, 42, 5, 15]$ ha sido desplazado $k = 4$ posiciones; $[1, 3, 5, 6, 7, 15]$ ha sido desplazado $k = 0$.

   Queremos encontrar el mayor elemento del arreglo en A . Obviamente, podemos encontrarlo en tiempo $\mathcal{O}(n)$. Describa un algoritmo que resuelva el problema en tiempo $\mathcal{O}(\log n)$.

5. Dado un arreglo ordenado con números todos diferentes $A[0..n\text{-}a]$, usted quiere encontrar si existe un índice i tal que $A[i] = i$. Dé un algoritmo *divide & conquer* que resuelva este problema en tiempo $\mathcal{O}(\log n)$.

Ejercicios propuestos 4

6. Una operación de *merge* múltiple. Suponga que tiene k arreglos ordenados, cada uno con n elementos, y que usted quiere combinarlos en un único arreglo ordenado de kn elementos.
    1. He aquí una estrategia: usando el procedimiento de *merge* que ya conocemos, combine los dos primeros arreglos, combine el resultado con el tercero, luego con el cuarto y así sucesivamente. ¿Cuál es la complejidad temporal de este algoritmo en términos de k y n?
    2. Dé un algoritmo más eficiente usando la técnica *divide & conquer*.