Programación III

Ricardo Wehbe

UADE

17 de octubre de 2021

Make an exhaustive list of everything you might do and do the last thing on the list.

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

Brian Eno, *Oblique Strategies*

Introducción

Introducción

- Hay problemas para los cuales no se conoce un algoritmo eficiente de resolución. En estos casos, la única solución posible es la exploración directa de todas las posibilidades (*brute force*).

Introducción

- Hay problemas para los cuales no se conoce un algoritmo eficiente de resolución. En estos casos, la única solución posible es la exploración directa de todas las posibilidades (*brute force*).
- La técnica de *backtracking* es un método de búsqueda de soluciones exhaustivo sobre grafos acíclicos. Este método puede optimizarse con la "poda" de alternativas que no conduzcan a una solución.

Introducción

- Hay problemas para los cuales no se conoce un algoritmo eficiente de resolución. En estos casos, la única solución posible es la exploración directa de todas las posibilidades (*brute force*).
- La técnica de *backtracking* es un método de búsqueda de soluciones exhaustivo sobre grafos acíclicos. Este método puede optimizarse con la "poda" de alternativas que no conduzcan a una solución.
- Los grafos representan las posibles alternativas por ser evaluadas.

Backtracking. Esquema general

Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).

Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).

Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).

Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada $(x_1, x_2, \ldots, x_n)$.

Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada $(x_1, x_2, \ldots, x_n)$.
- Cada elemento x_i de la lista se escoge entre un conjunto de candidatos. Cada lista representa un *estado*.

Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada $(x_1, x_2, \ldots, x_n)$.
- Cada elemento x_i de la lista se escoge entre un conjunto de candidatos. Cada lista representa un *estado*.
- A veces se buscan todas las soluciones; a veces, nos conformamos con una ("estado solución").

Backtracking. Esquema general

- Se representan todas las posibilidades en un árbol (como ya sabemos, decir que un grafo conexo es acíclico equivale a decir que es un árbol).
- Se busca la solución recorriendo el árbol (de una determinada forma según la estrategia).
- Hay partes del árbol (sub-árboles) que se evitan porque no pueden contener soluciones (poda).
- La solución del problema se presenta en una lista ordenada $(x_1, x_2, \ldots, x_n)$.
- Cada elemento x_i de la lista se escoge entre un conjunto de candidatos. Cada lista representa un *estado*.
- A veces se buscan todas las soluciones; a veces, nos conformamos con una ("estado solución").
- A veces no existe ninguna solución.

Elementos del algoritmo general de *backtracking*

Elementos del algoritmo general de *backtracking*

- Para aplicar *backtracking* debemos proveer los datos S de cada instancia particular y cinco parámetros: *root*, *reject*, *accept*, *first* y *next*.

Elementos del algoritmo general de *backtracking*

- Para aplicar *backtracking* debemos proveer los datos S de cada instancia particular y cinco parámetros: *root*, *reject*, *accept*, *first* y *next*.
- $root(T)$ devuelve el candidato parcial en la raíz del árbol de búsqueda T.
- $reject(T, c)$ devuelve *true* si no vale la pena completar el candidato c.
- $accept(T, c)$ devuelve *true* si el candidato c es una solución de T y *false* si no.
- $first(T, c)$ genera la primera extensión del candidato c.
- $next(T, c)$ genera la siguiente extensión luego de c.

El algoritmo general de *backtracking*

El algoritmo general de *backtracking*

Se comienza con la llamada $backtracking(T, root(T))$. El pseudo- código es el

siguiente:

El algoritmo general de *backtracking*

Se comienza con la llamada $backtracking(T, root(T))$. El pseudo- código es el siguiente:

```
1    algorithm backtracking(T, c)
2    if reject(T, c)
3        return
4    endif
5    if accept(T, c)
6        return (T, c)
7    endif
8    s ← first(T, c)
9    while s ≠ NULL
10       backtracking(T, s)
11       s ← next(T, c)
12   endwhile
```

El problema de las *n* damas

El problema de las *n* damas

- La dama (o reina) es la pieza más poderosa en el juego del ajedrez. Se puede mover a cualquier casilla situada en su misma fila, columna o diagonales.

El problema de las *n* damas

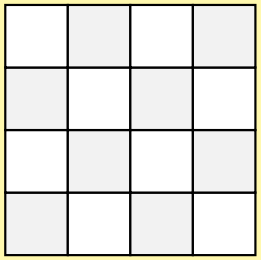
- La dama (o reina) es la pieza más poderosa en el juego del ajedrez. Se puede mover a cualquier casilla situada en su misma fila, columna o diagonales.
- El problema consiste en ubicar n damas en un tablero de $n \times n$ casillas (con $n \geq 4$) de tal manera que ninguna esté en el radio de acción de otra.

El problema de las *n* damas

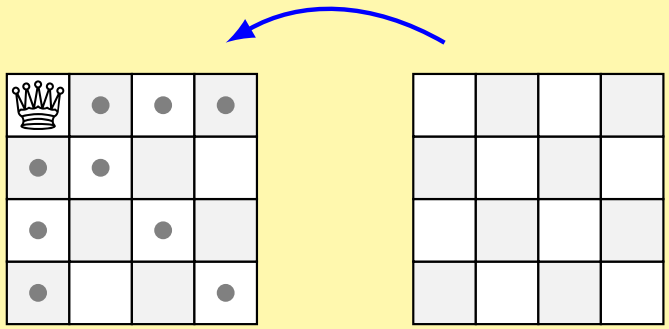
- La dama (o reina) es la pieza más poderosa en el juego del ajedrez. Se puede mover a cualquier casilla situada en su misma fila, columna o diagonales.
- El problema consiste en ubicar n damas en un tablero de $n \times n$ casillas (con $n \geq 4$) de tal manera que ninguna esté en el radio de acción de otra.
- Por lo tanto, las damas deben ubicarse una en cada fila y cada columna.

El problema de las n damas. Un ejemplo con $n = 4$

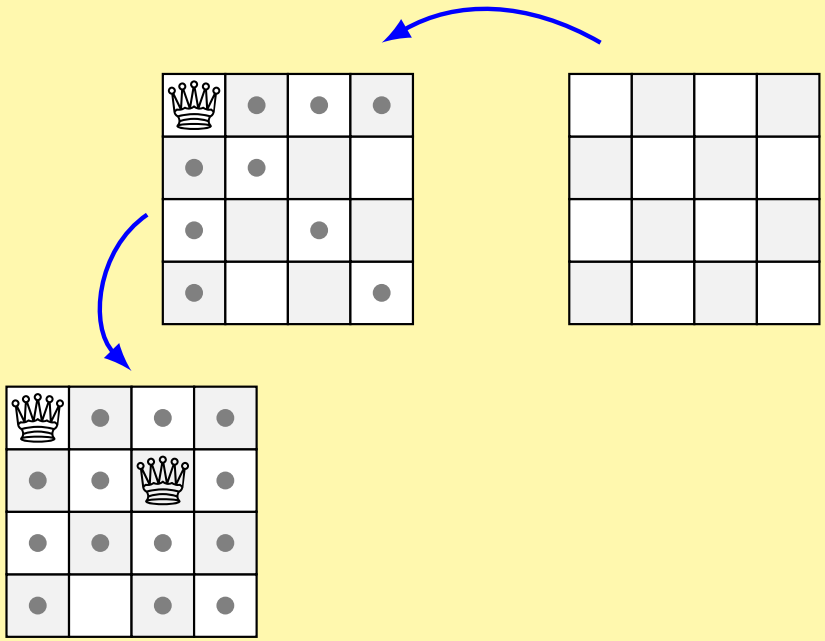
El problema de las n damas. Un ejemplo con $n = 4$



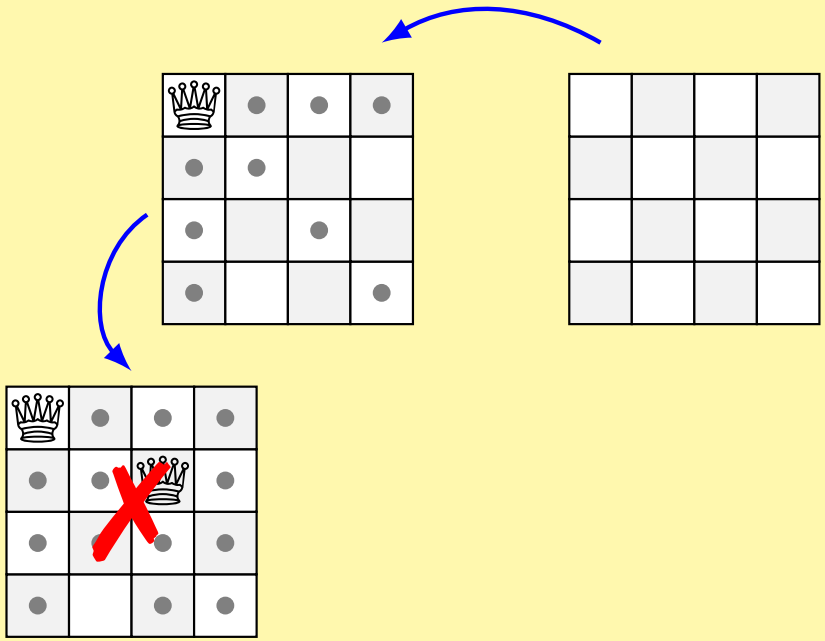
El problema de las n damas. Un ejemplo con $n = 4$



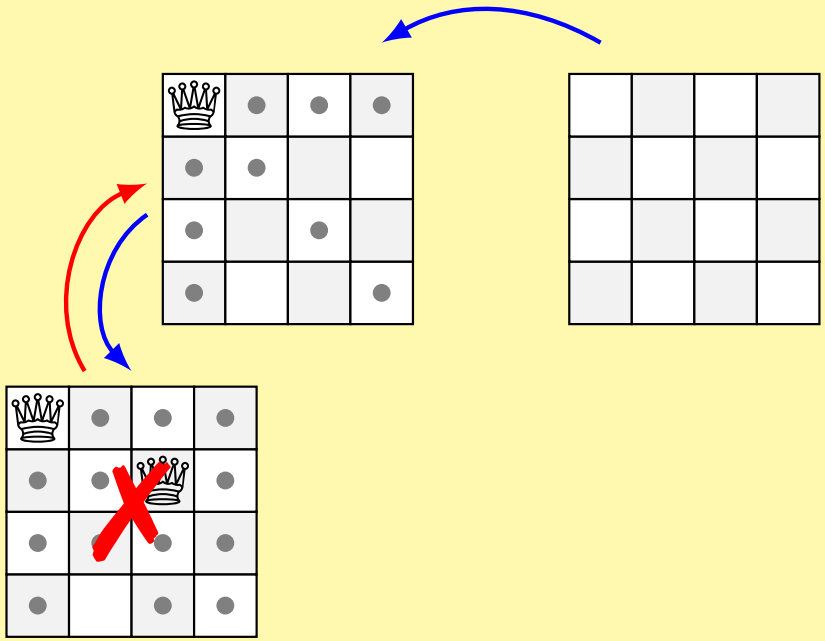
El problema de las n damas. Un ejemplo con $n = 4$



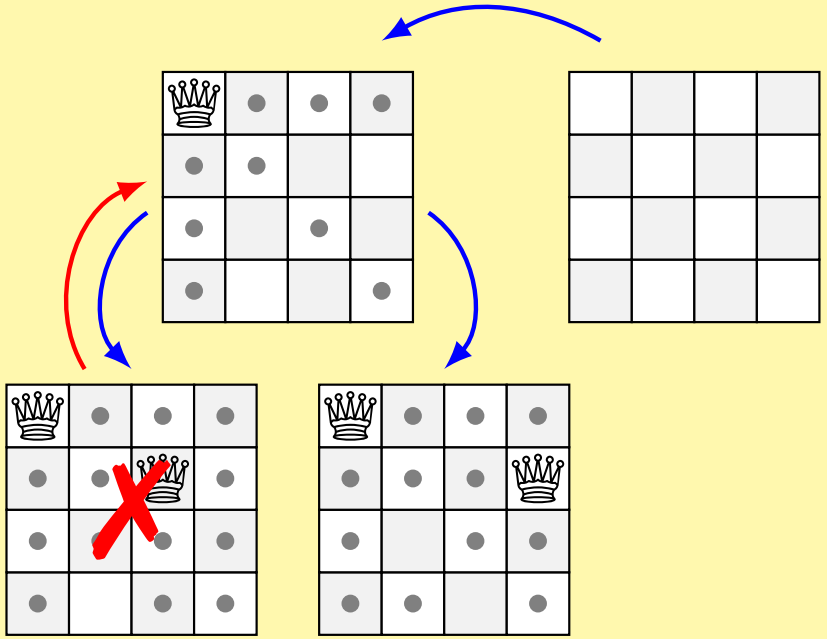
El problema de las n damas. Un ejemplo con $n = 4$



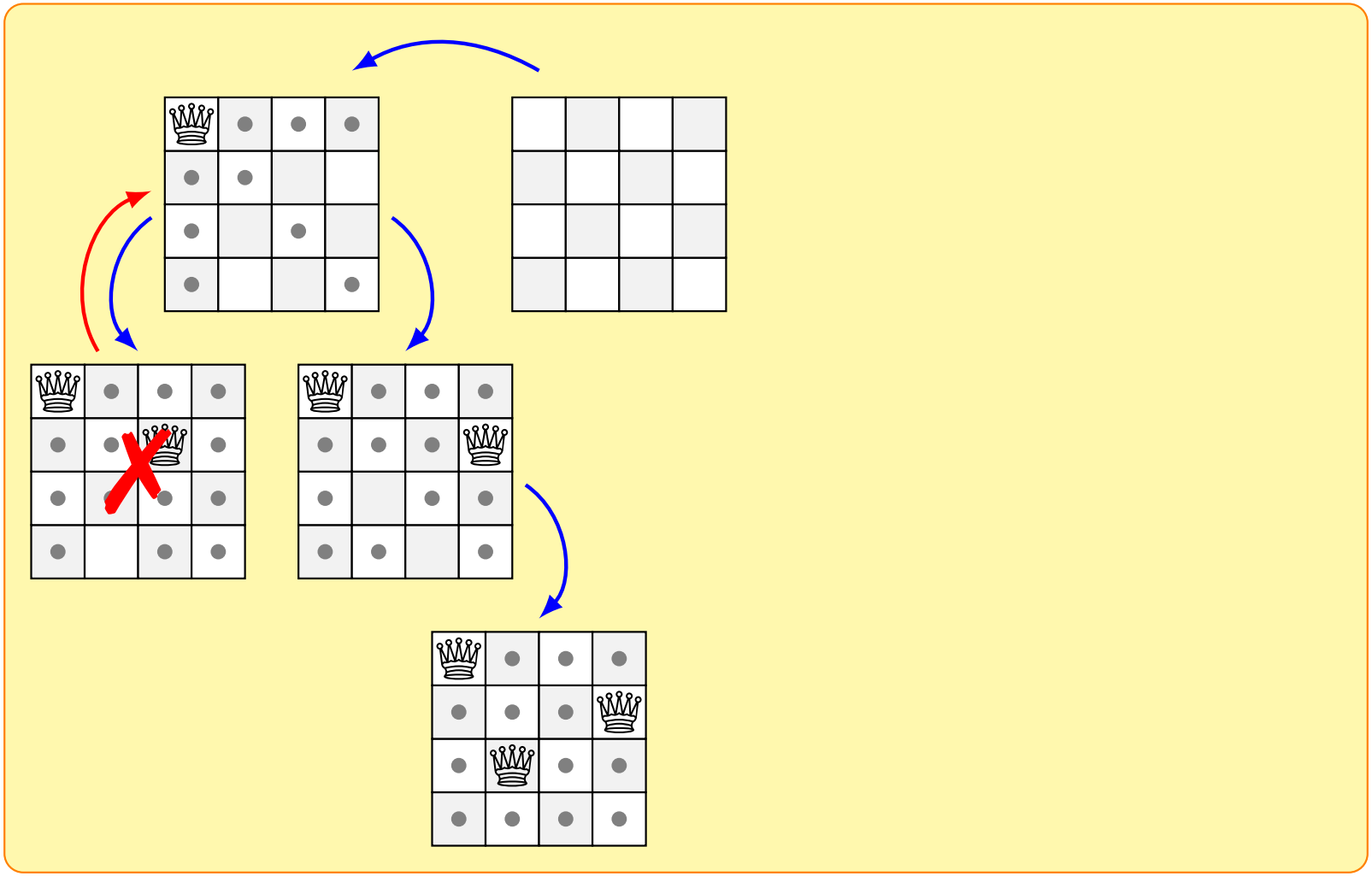
El problema de las n damas. Un ejemplo con $n = 4$



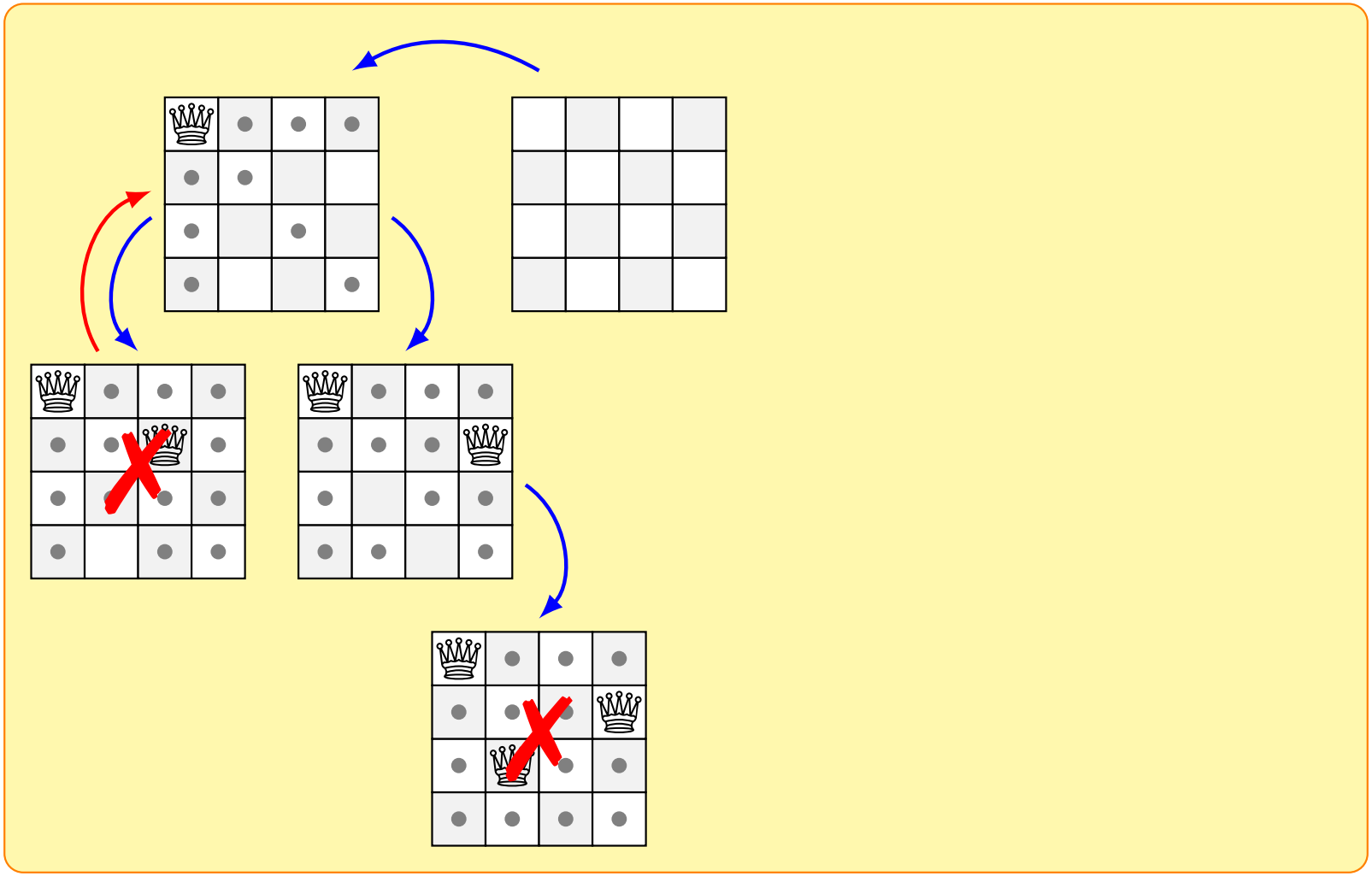
El problema de las n damas. Un ejemplo con $n = 4$



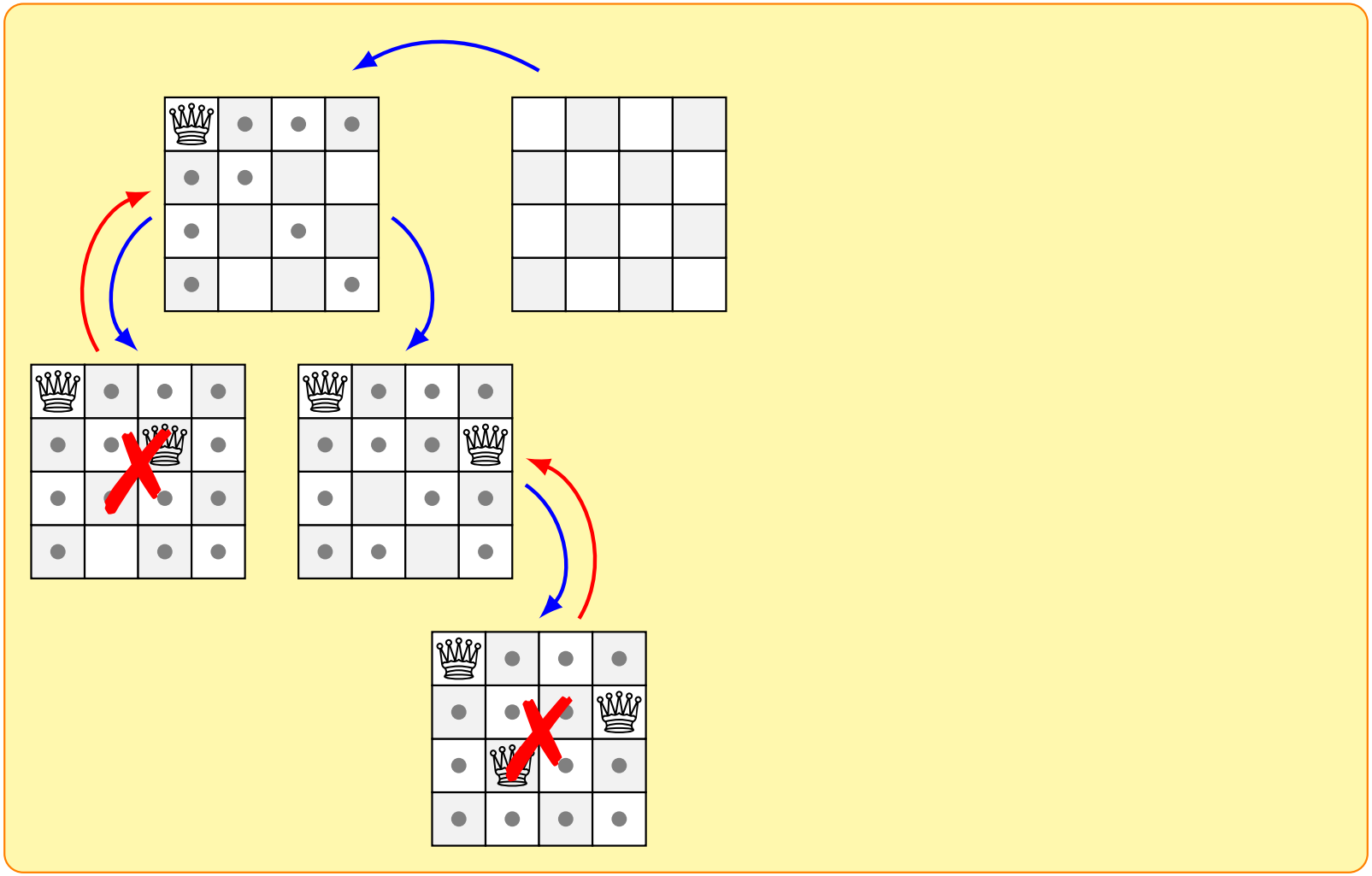
El problema de las n damas. Un ejemplo con $n = 4$



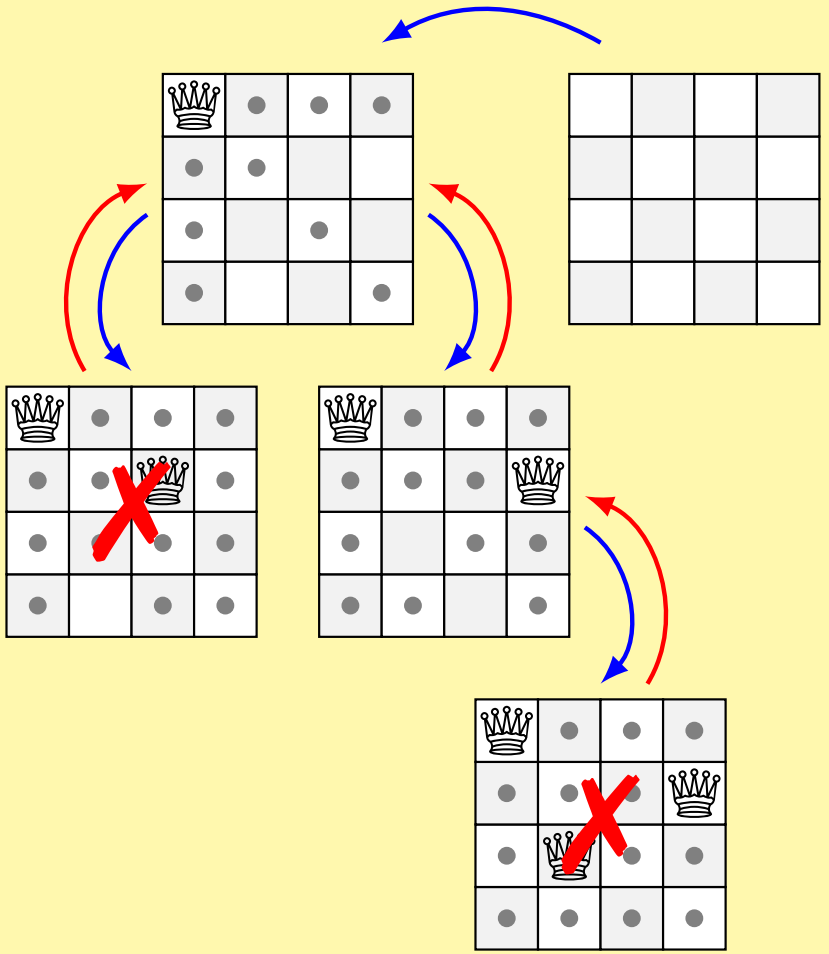
El problema de las n damas. Un ejemplo con $n = 4$



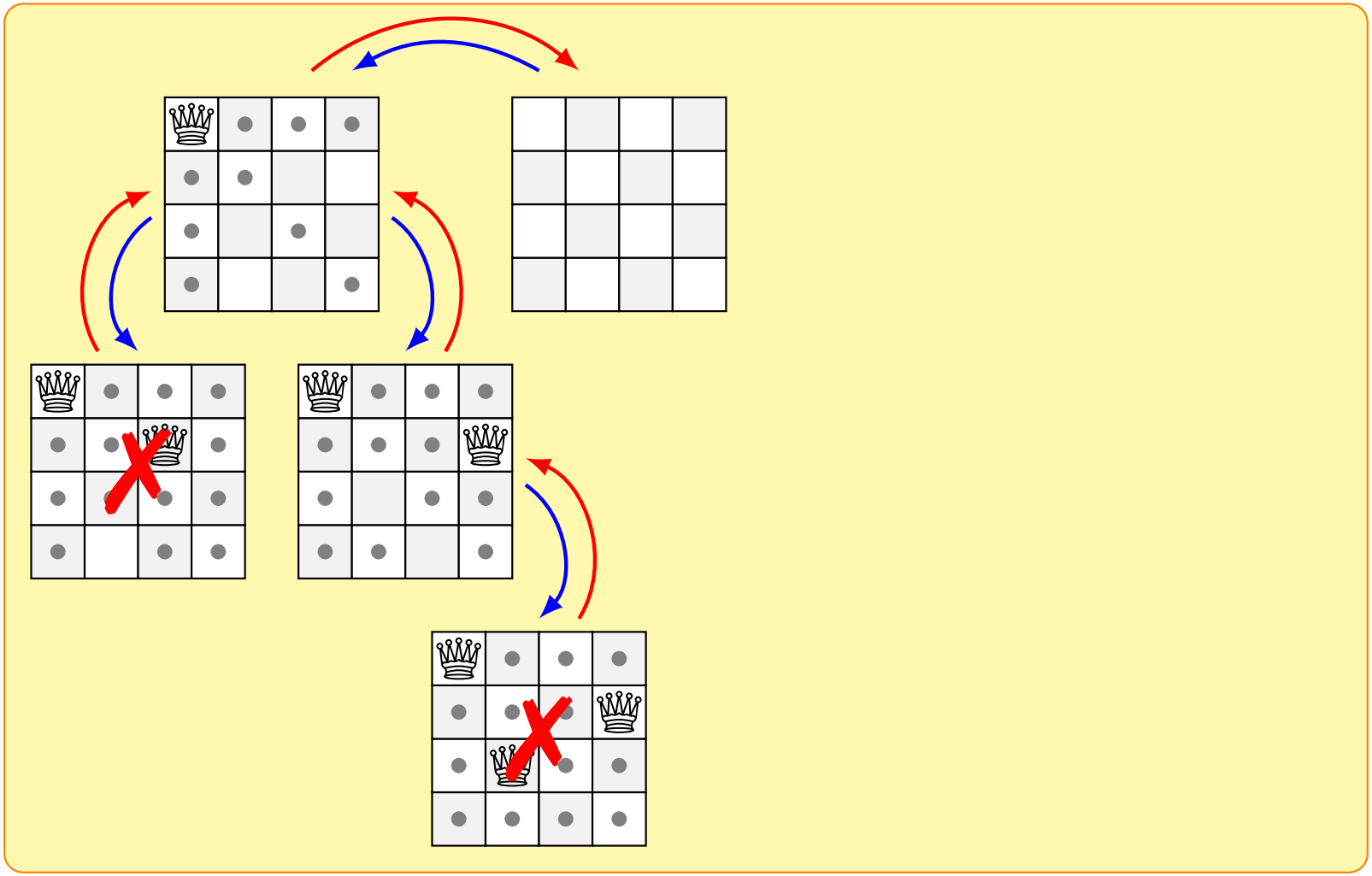
El problema de las n damas. Un ejemplo con $n = 4$



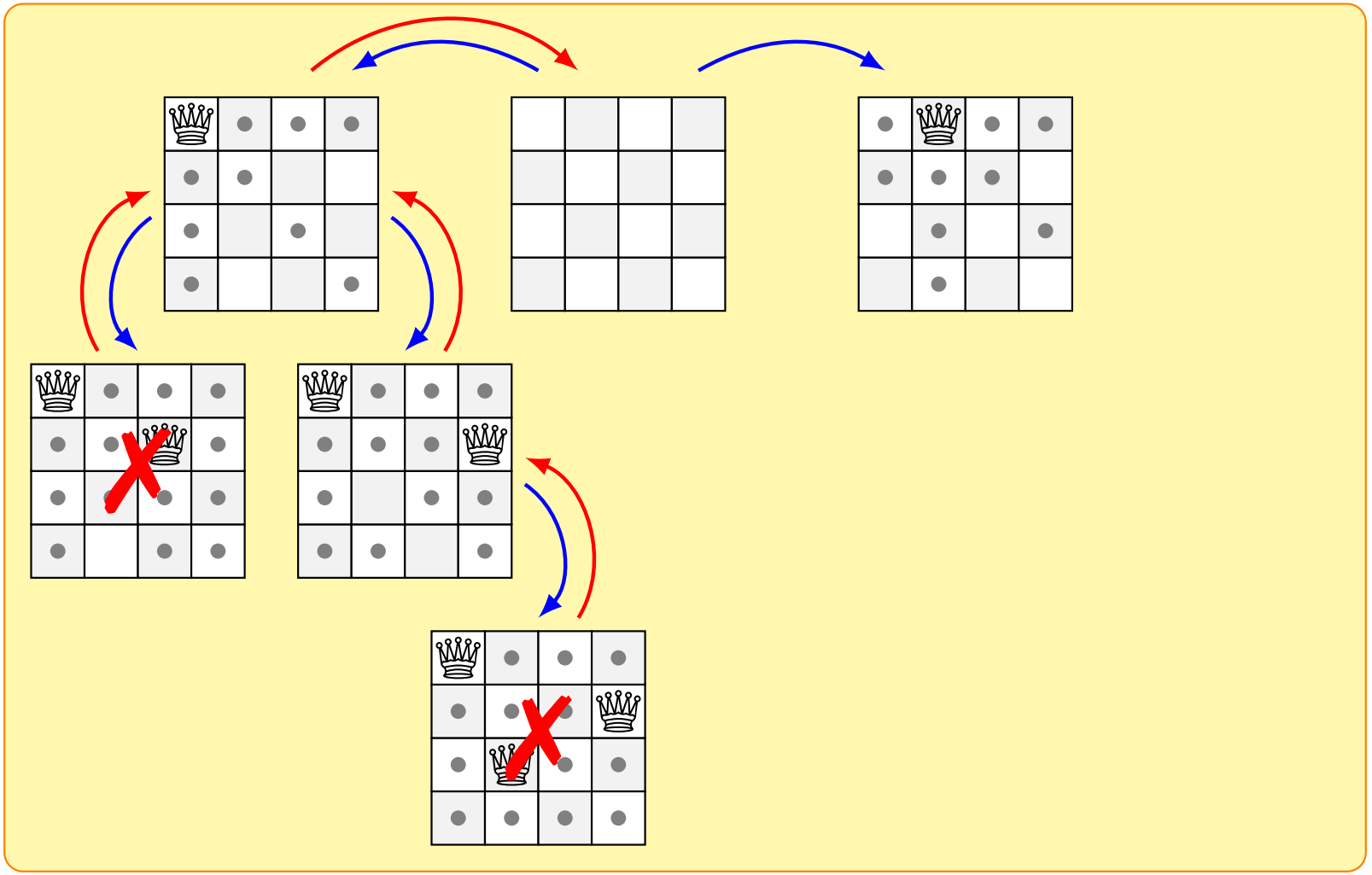
El problema de las n damas. Un ejemplo con $n = 4$



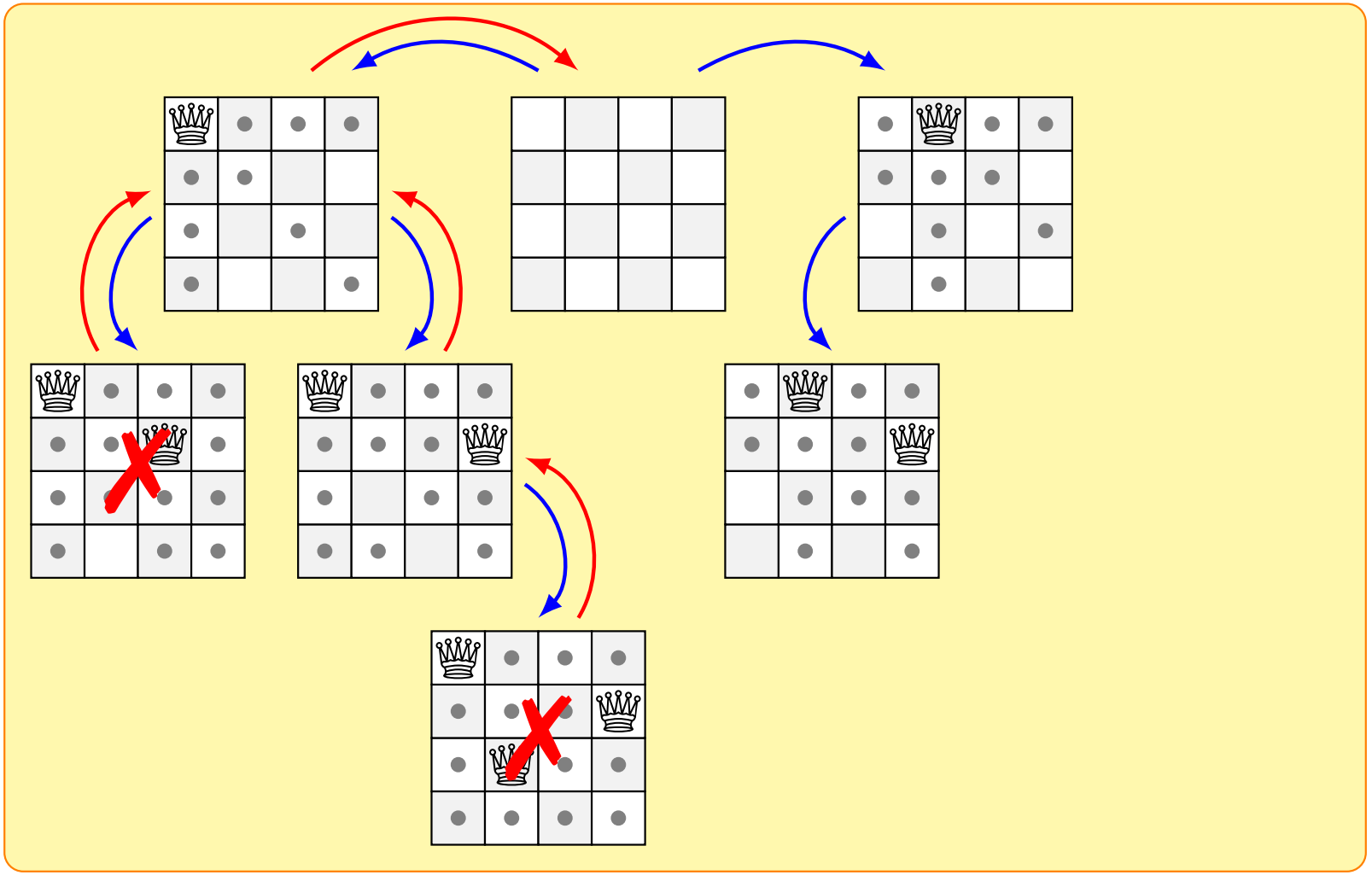
El problema de las n damas. Un ejemplo con $n = 4$



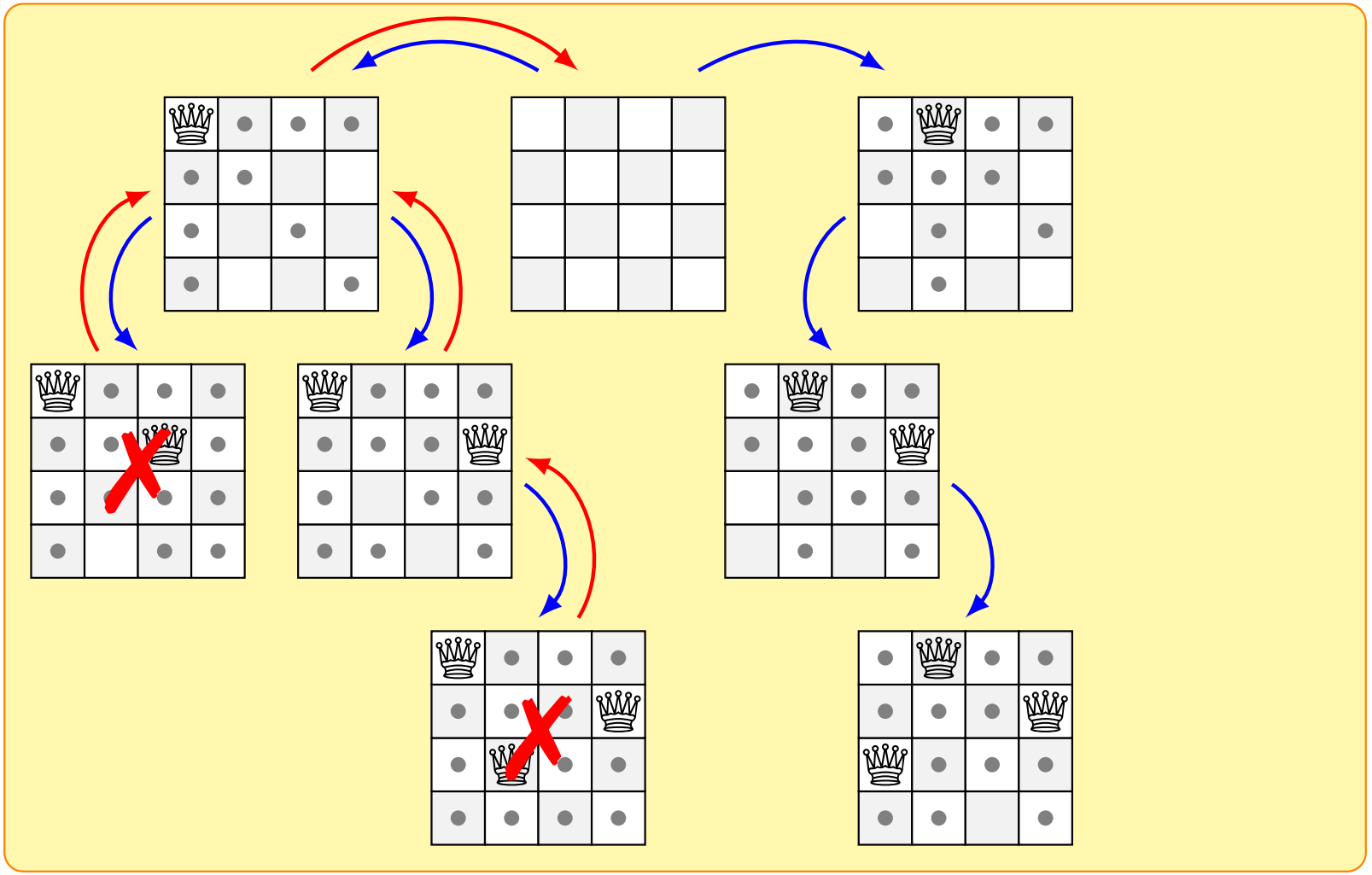
El problema de las n damas. Un ejemplo con $n = 4$



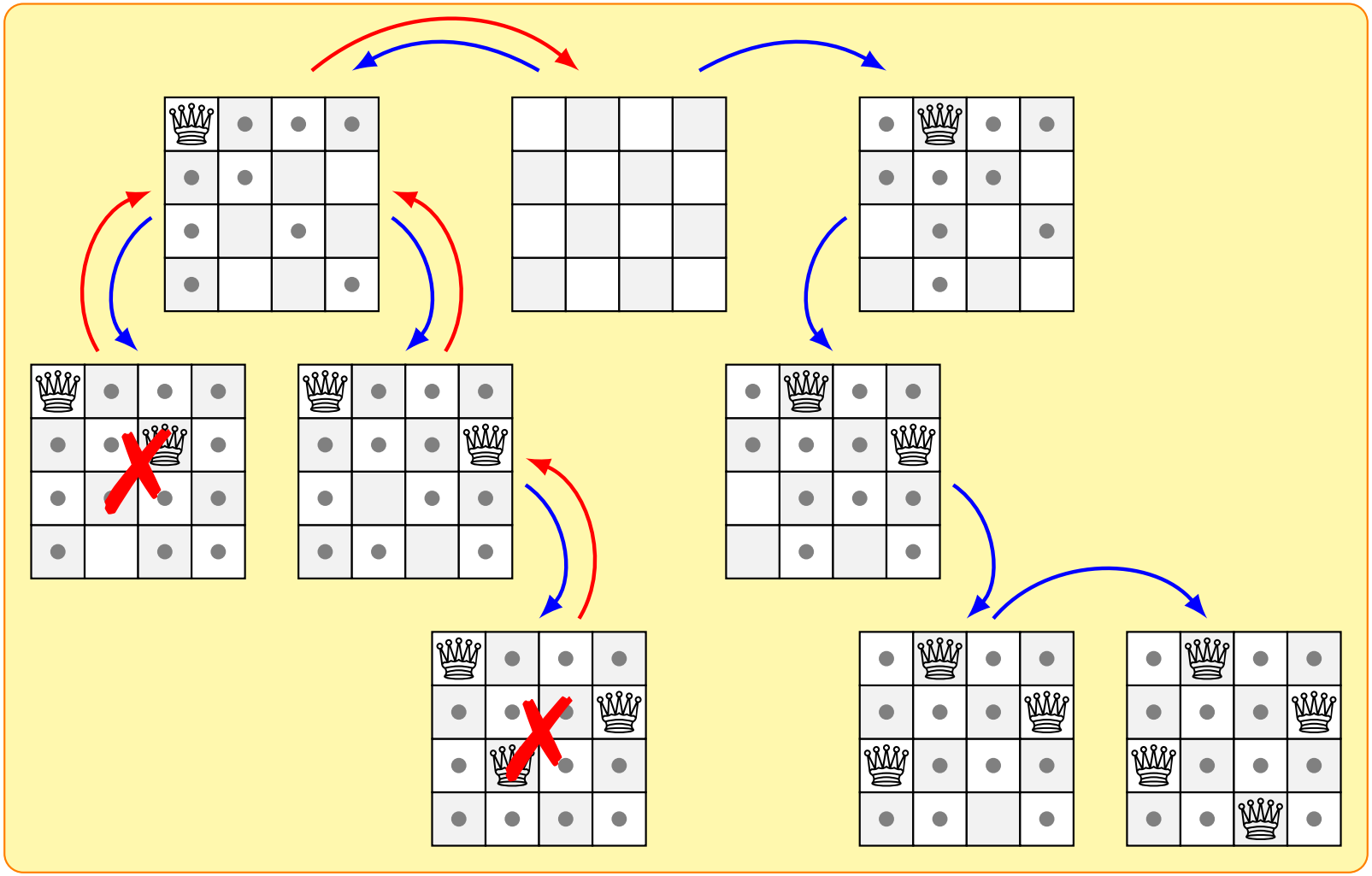
El problema de las n damas. Un ejemplo con $n = 4$



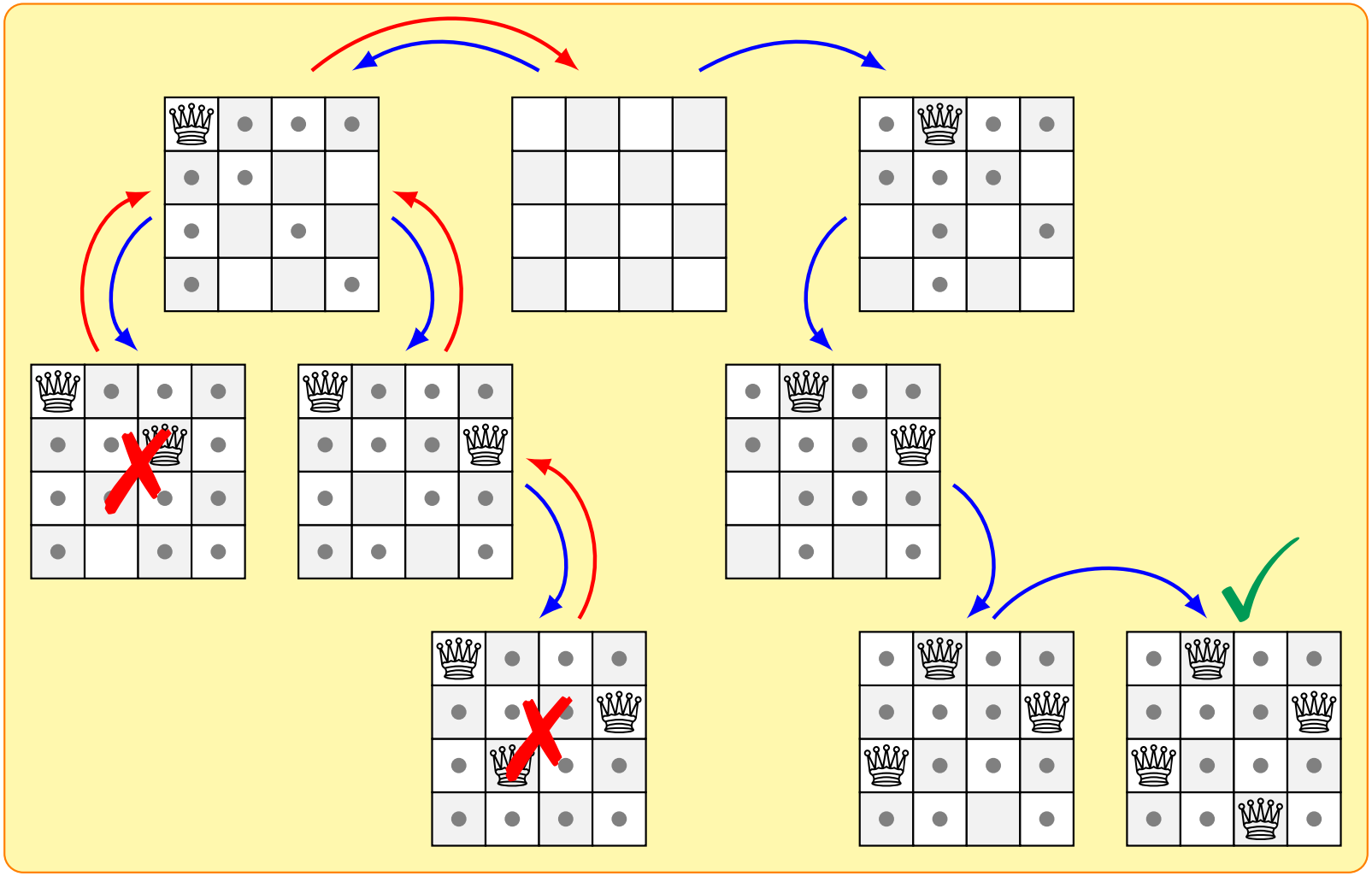
El problema de las n damas. Un ejemplo con $n = 4$



El problema de las n damas. Un ejemplo con $n = 4$



El problema de las n damas. Un ejemplo con $n = 4$



Un algoritmo para el problema de las *n* damas

Un algoritmo para el problema de las *n* damas

- Teniendo en cuenta que nunca puede haber más de una dama en una fila o en una columna, se puede expresar la solución en un vector de n posiciones donde se indique, para cada fila, en qué columna se encuentra la dama correspondiente. Cada posición contendrá entonces un número de 1 a n.

Un algoritmo para el problema de las *n* damas

- Teniendo en cuenta que nunca puede haber más de una dama en una fila o en una columna, se puede expresar la solución en un vector de n posiciones donde se indique, para cada fila, en qué columna se encuentra la dama correspondiente. Cada posición contendrá entonces un número de 1 a n.
- No es necesario verificar las filas, pues no puede haber más de una dama en cada posición del vector; para chequear las columnas, es necesario verificar que no haya repeticiones.

Un algoritmo para el problema de las n damas

- Teniendo en cuenta que nunca puede haber más de una dama en una fila o en una columna, se puede expresar la solución en un vector de n posiciones donde se indique, para cada fila, en qué columna se encuentra la dama correspondiente. Cada posición contendrá entonces un número de 1 a n.
- No es necesario verificar las filas, pues no puede haber más de una dama en cada posición del vector; para chequear las columnas, es necesario verificar que no haya repeticiones.
- Para verificar las diagonales, observemos que las casillas diagonales que "ve" una dama situada en la fila i y la columna j son las que se encuentran en las filas $i \pm k$ (en nuestro caso sólo nos interesarán los valores mayores) y columnas $j \pm k$.

Un algoritmo para el problema de las *n* damas. Parte 1

```
1.   boolean algorithm Queens (int [n] S, int e)
2.   boolean ok = false
3.   S[e] = 1
4.   while ((S[e] < n + 1 && !(ok))) {
5.      if DamaOK(S, e) {
6.         if (e = n) {
7.            ok = true
8.         } else {
9.            ok = Queens(S, e + 1)
10.        }
11.     }
12.     S[e] = S[e] + 1
13.  }
14.  return ok
```

Un algoritmo para el problema de las *n* damas. Parte 2

```
1.  boolean algorithm DamaOK (int [n] S, int e)
2.  for (i = 1; i ≤ e − 1; i + +) {
3.     if (S[i] == S[e] || |S[i] − S[e]| = |i − e|) {
4.       return false
5.     }
6.  }
7.  return true
```

Un algoritmo para el problema de las n damas. Parte 2

```
1.  boolean algorithm DamaOK (int [n] S, int e)
2.  for (i = 1; i ≤ e − 1; i + +) {
3.     if (S[i] == S[e] || |S[i] − S[e]| = |i − e|) {
4.       return false
5.     }
6.  }
7.  return true
```

Observe: aquí tenemos $\mathcal{O}(n)$; tenemos además $a = n$, $b = 1$; Por lo tanto, estamos en $\mathcal{O}(n^n)$. Horrible.

El problema de la suma del subconjunto

El problema de la suma del subconjunto

- El problema se puede plantear así: dado un conjunto de n números v y un valor m, mostrar todos los subconjuntos de v en los cuales la suma de sus elementos sea m.

El problema de la suma del subconjunto

- El problema se puede plantear así: dado un conjunto de n números v y un valor m, mostrar todos los subconjuntos de v en los cuales la suma de sus elementos sea m.
- Puesto que en una solución cada elemento de v está o no está, la solución puede expresarse como un vector booleano con n posiciones.

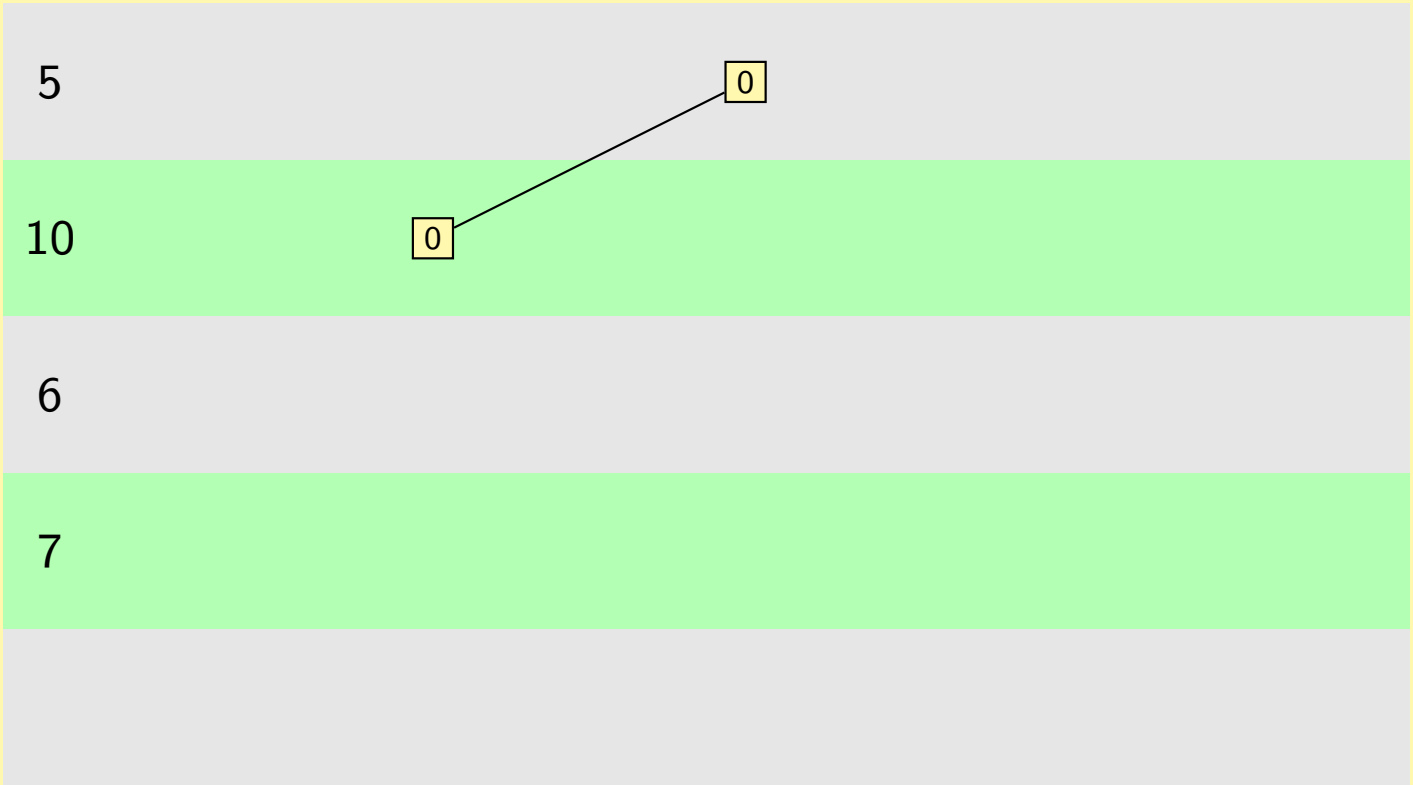
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$

5	0
10	
6	
7	

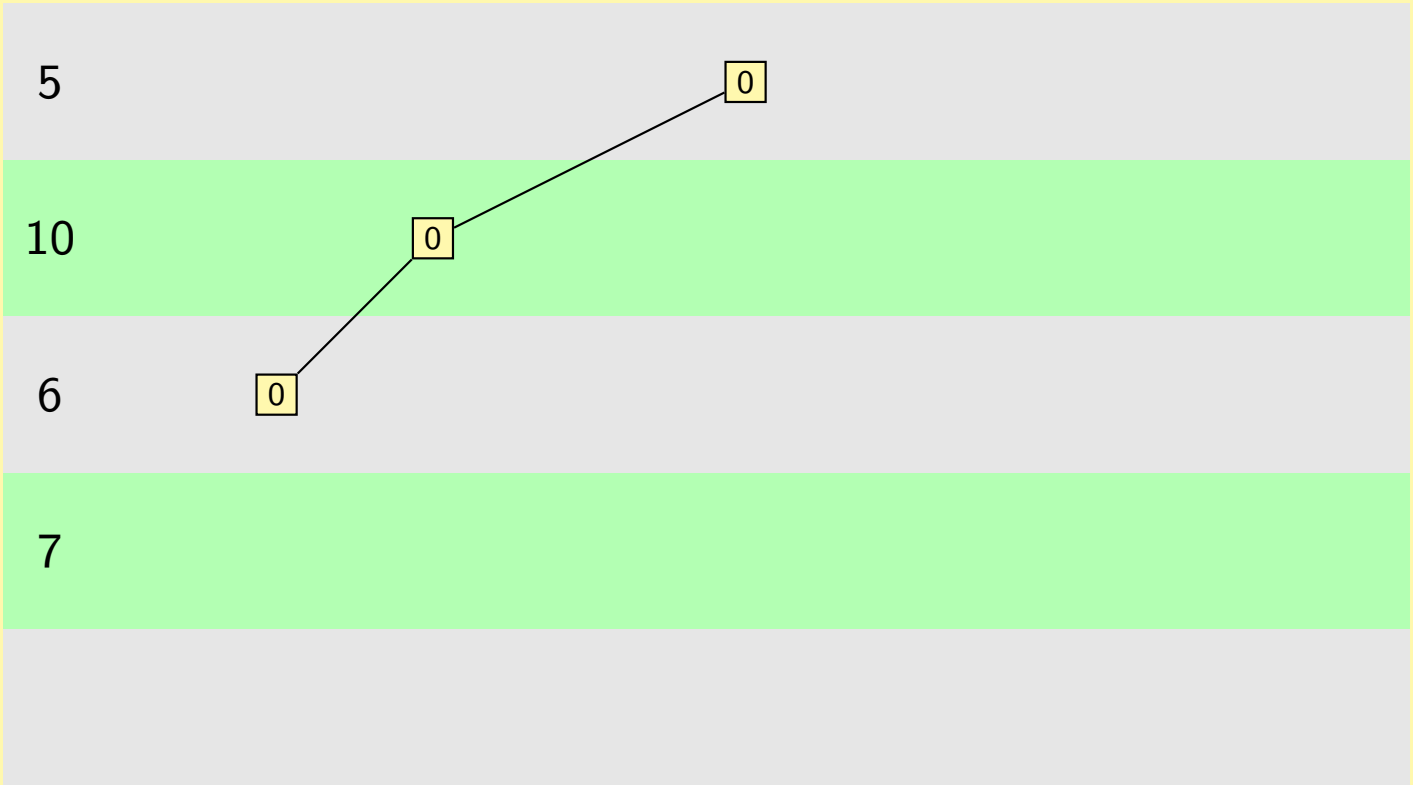
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



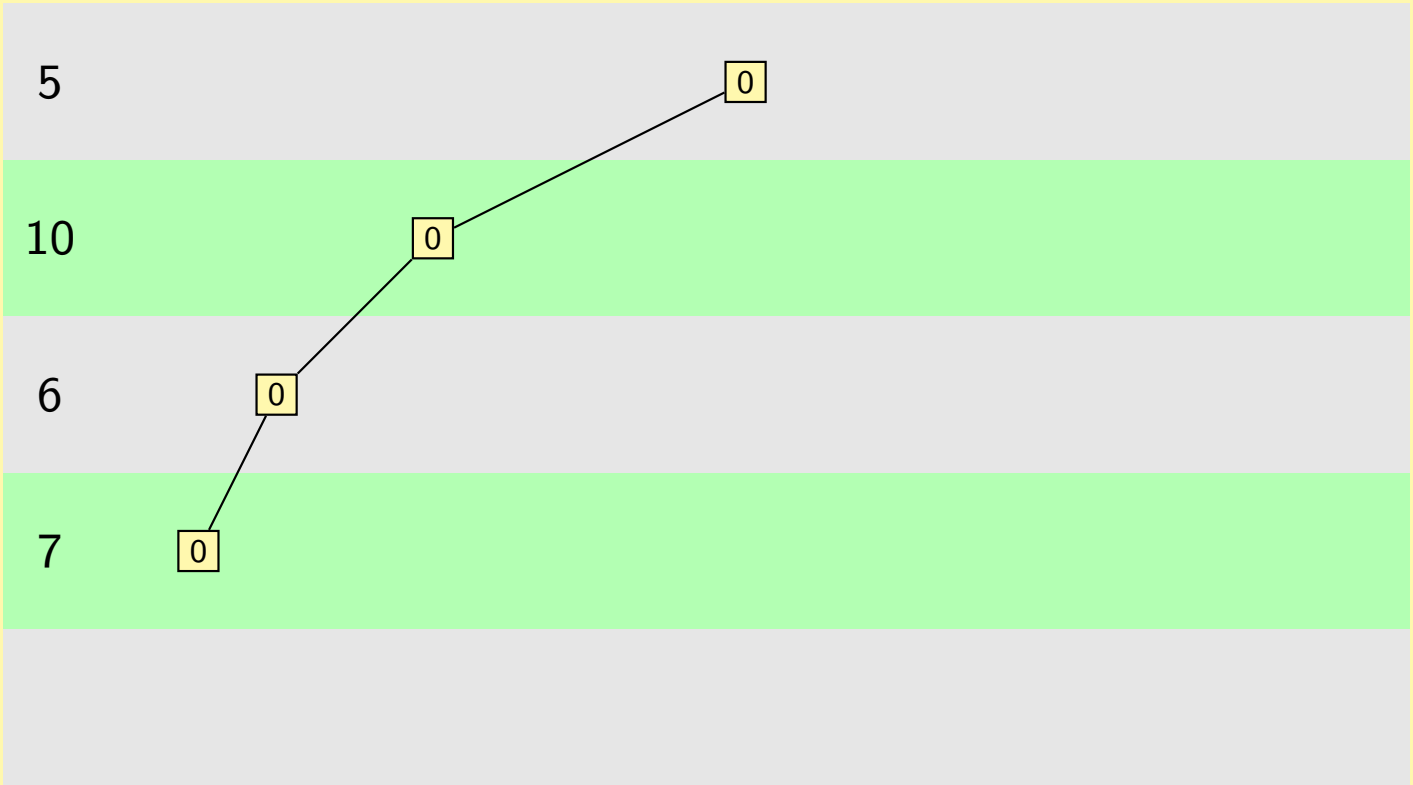
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



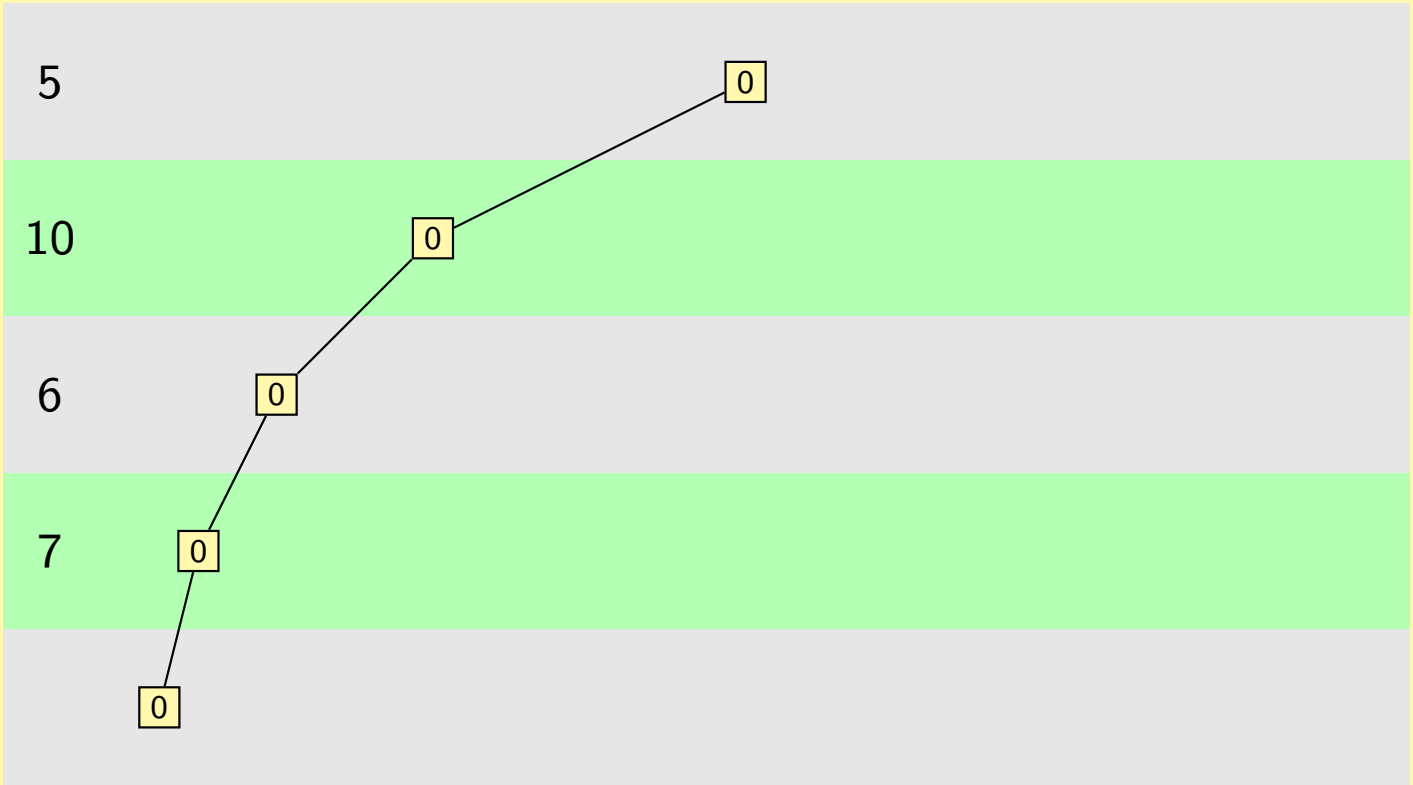
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



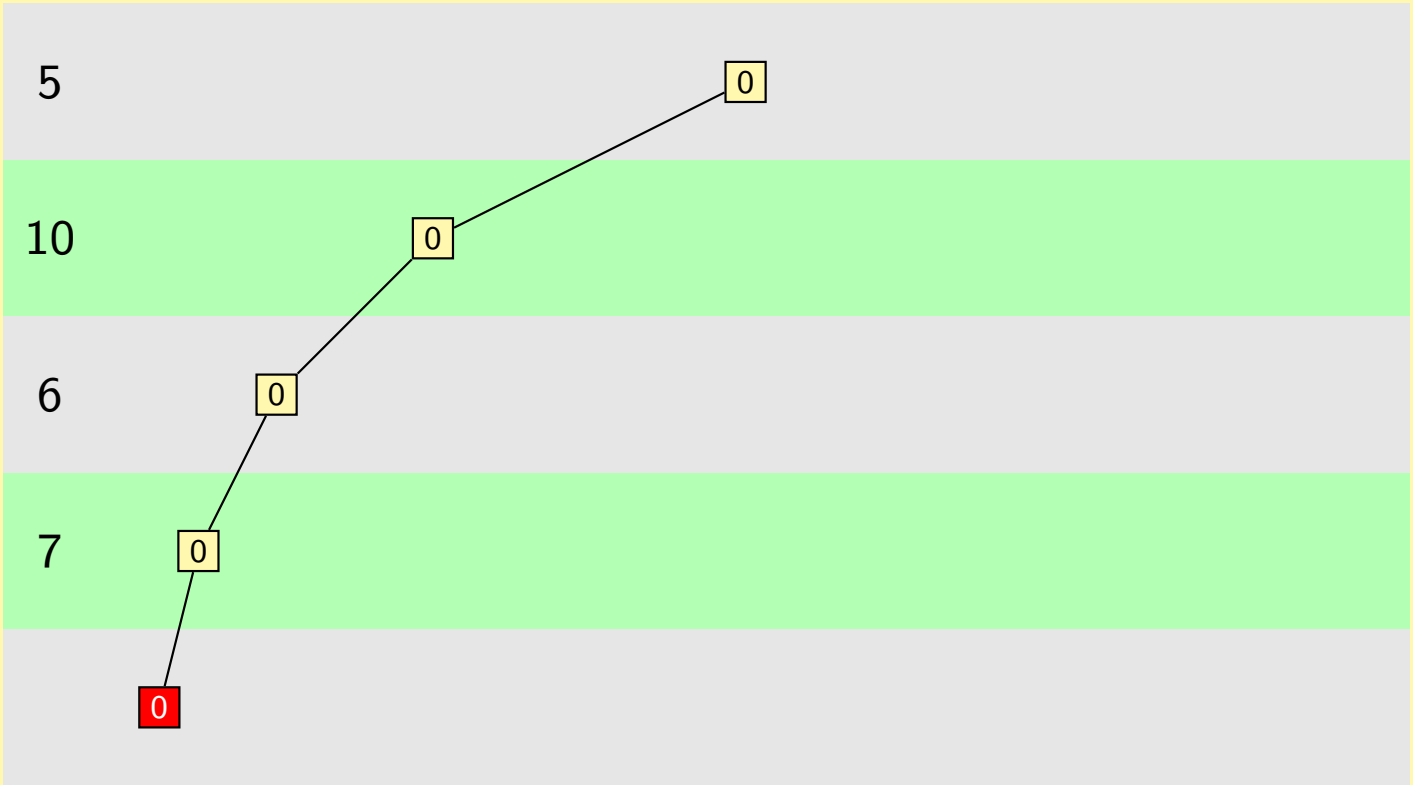
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



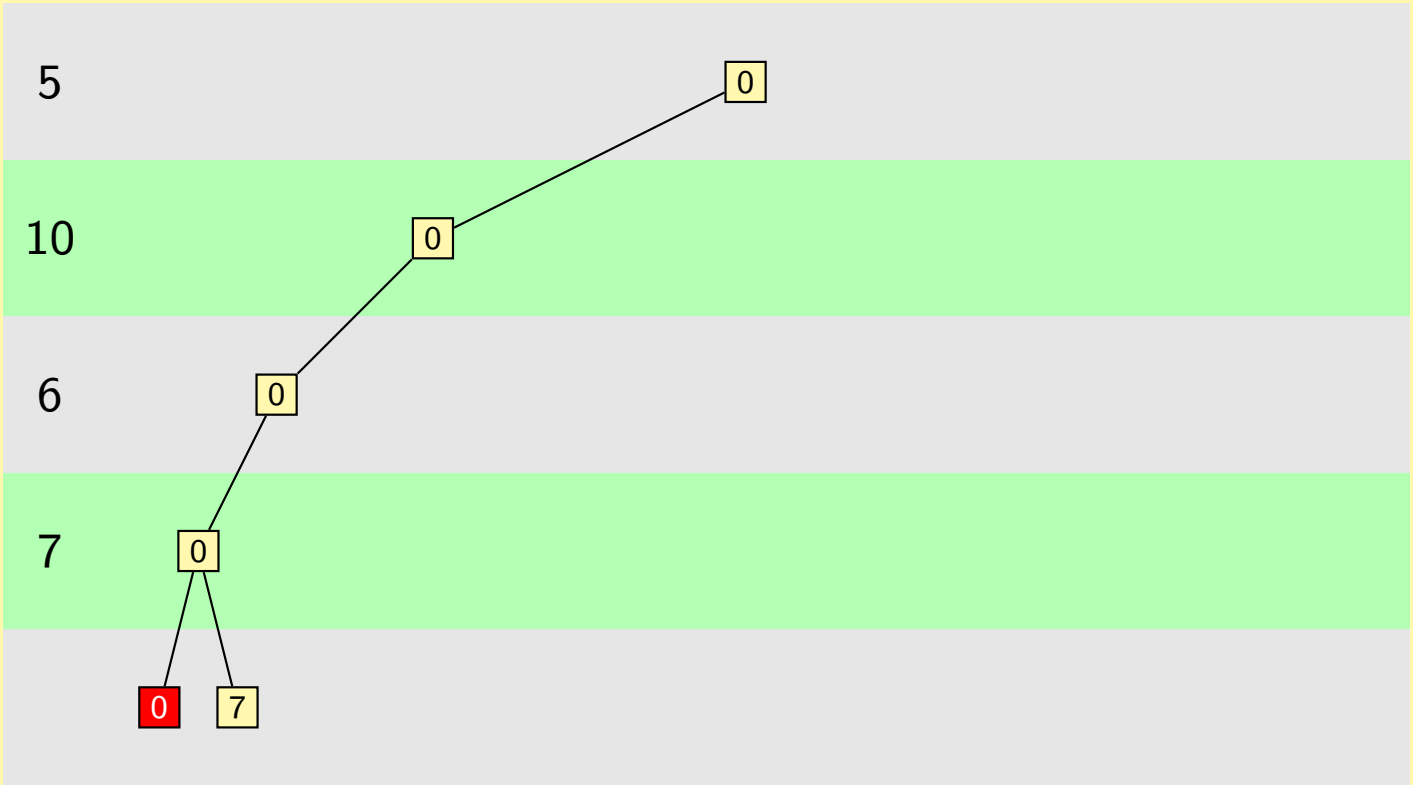
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



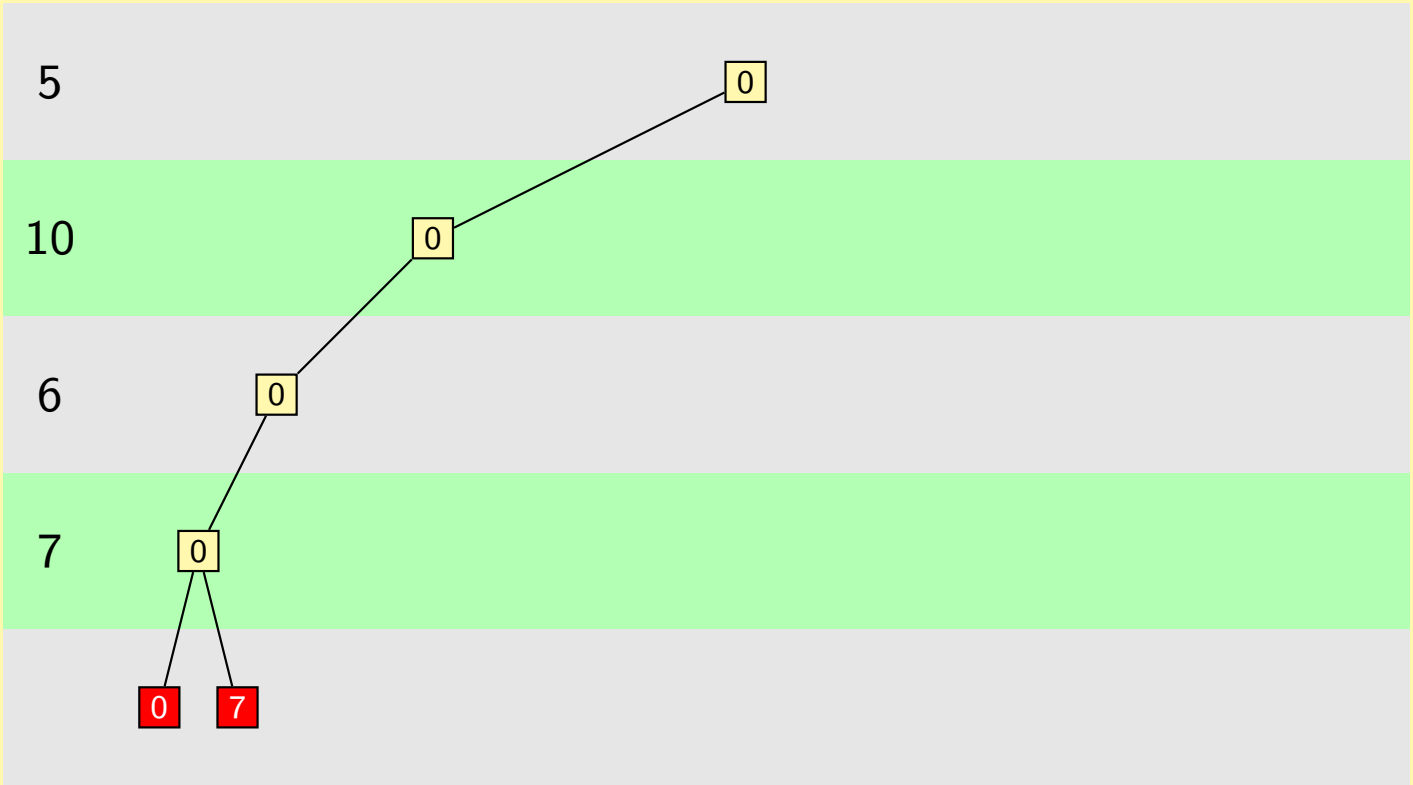
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



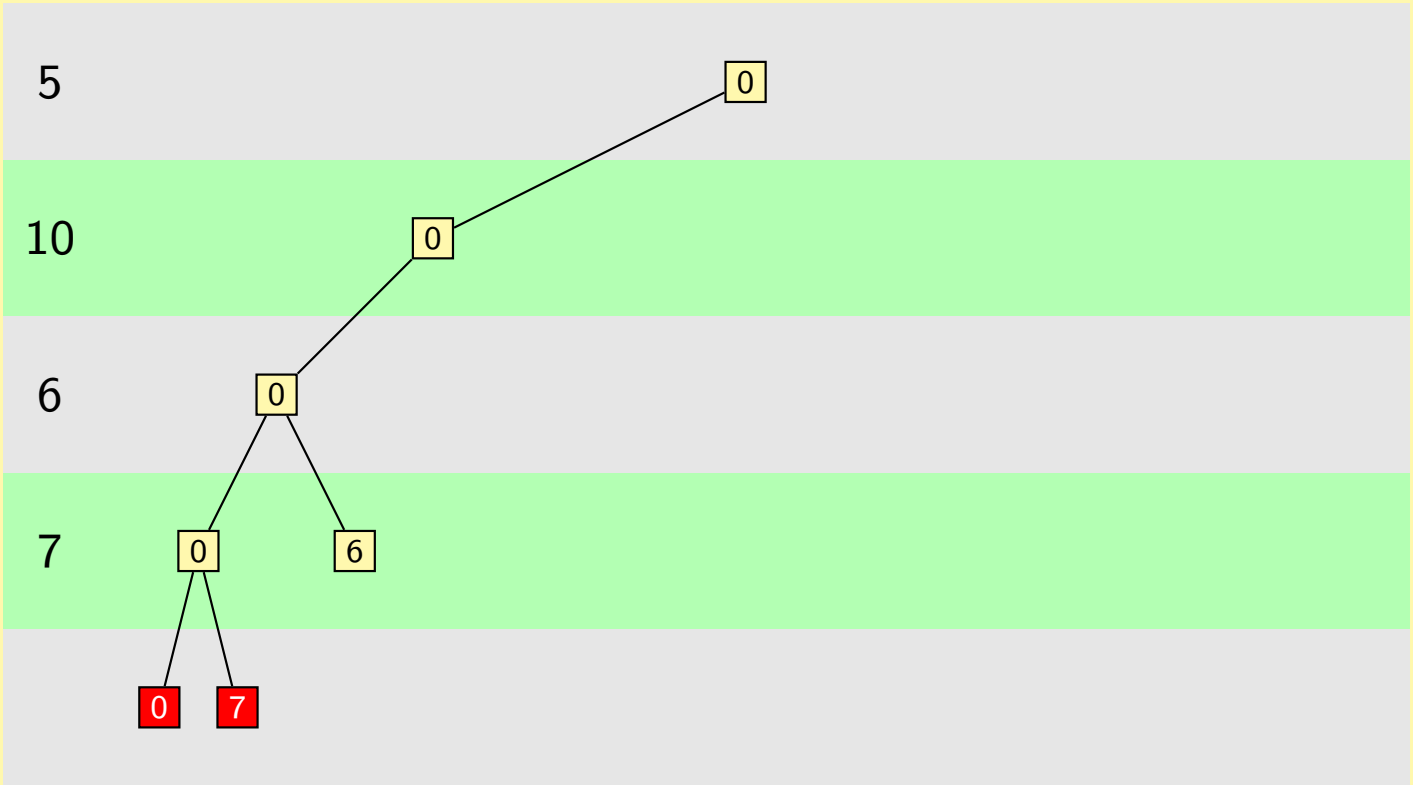
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



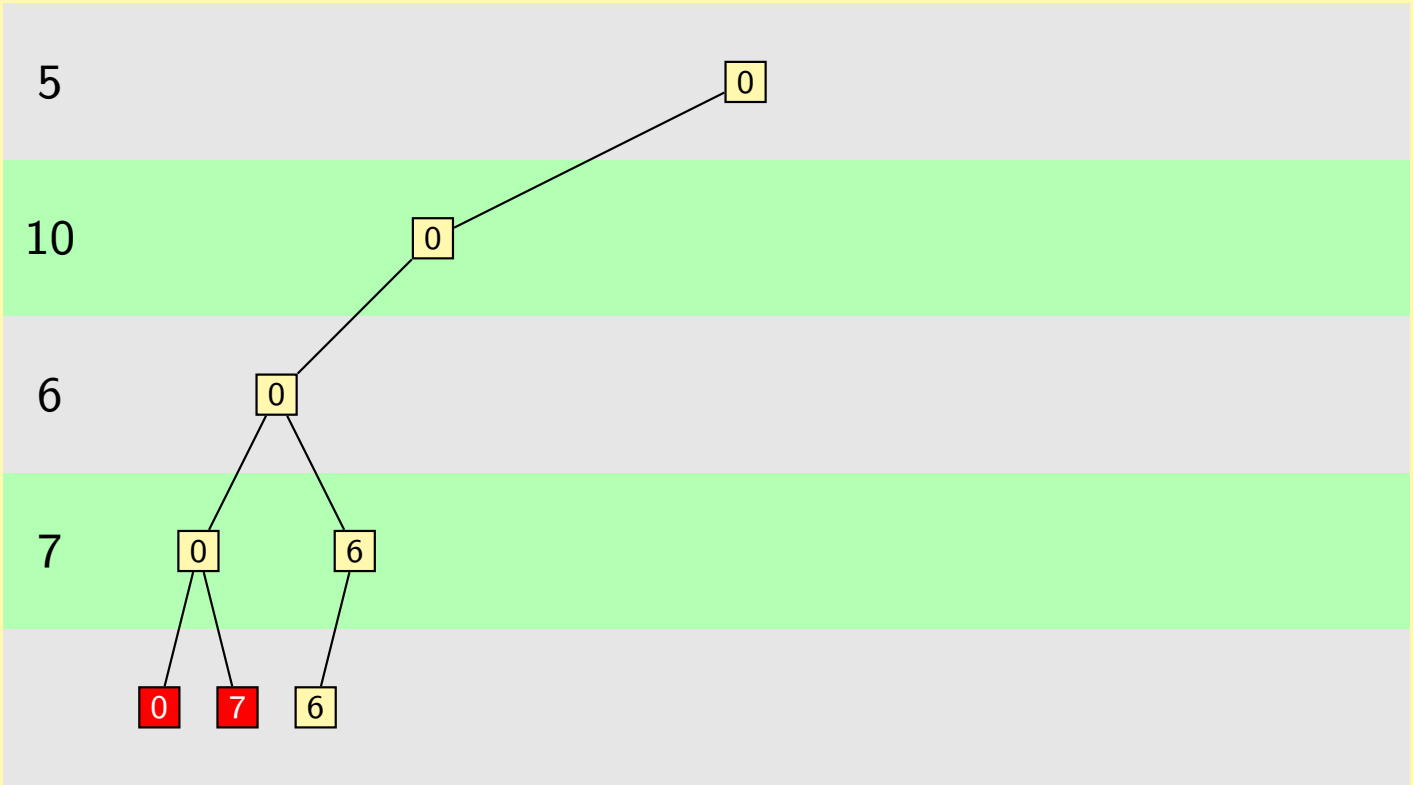
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



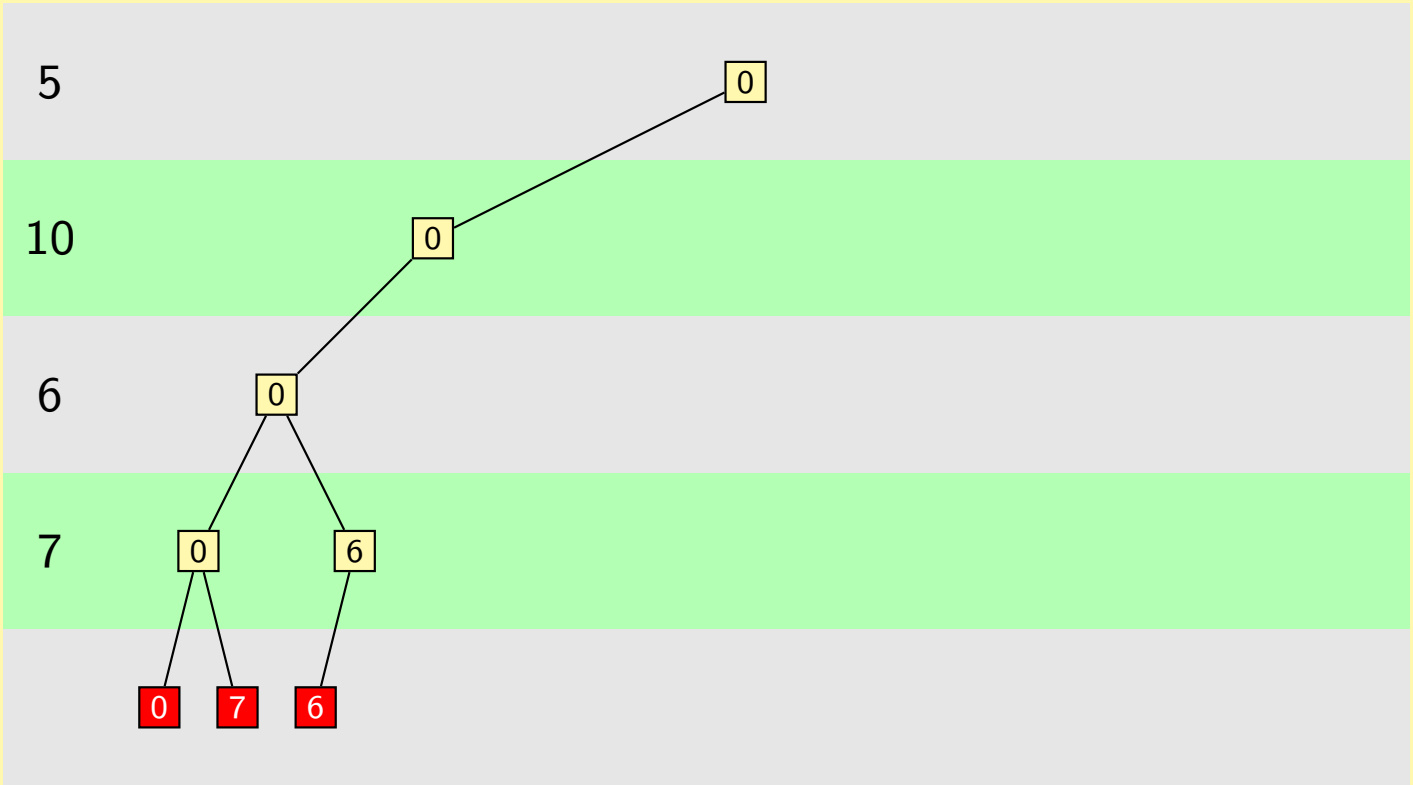
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



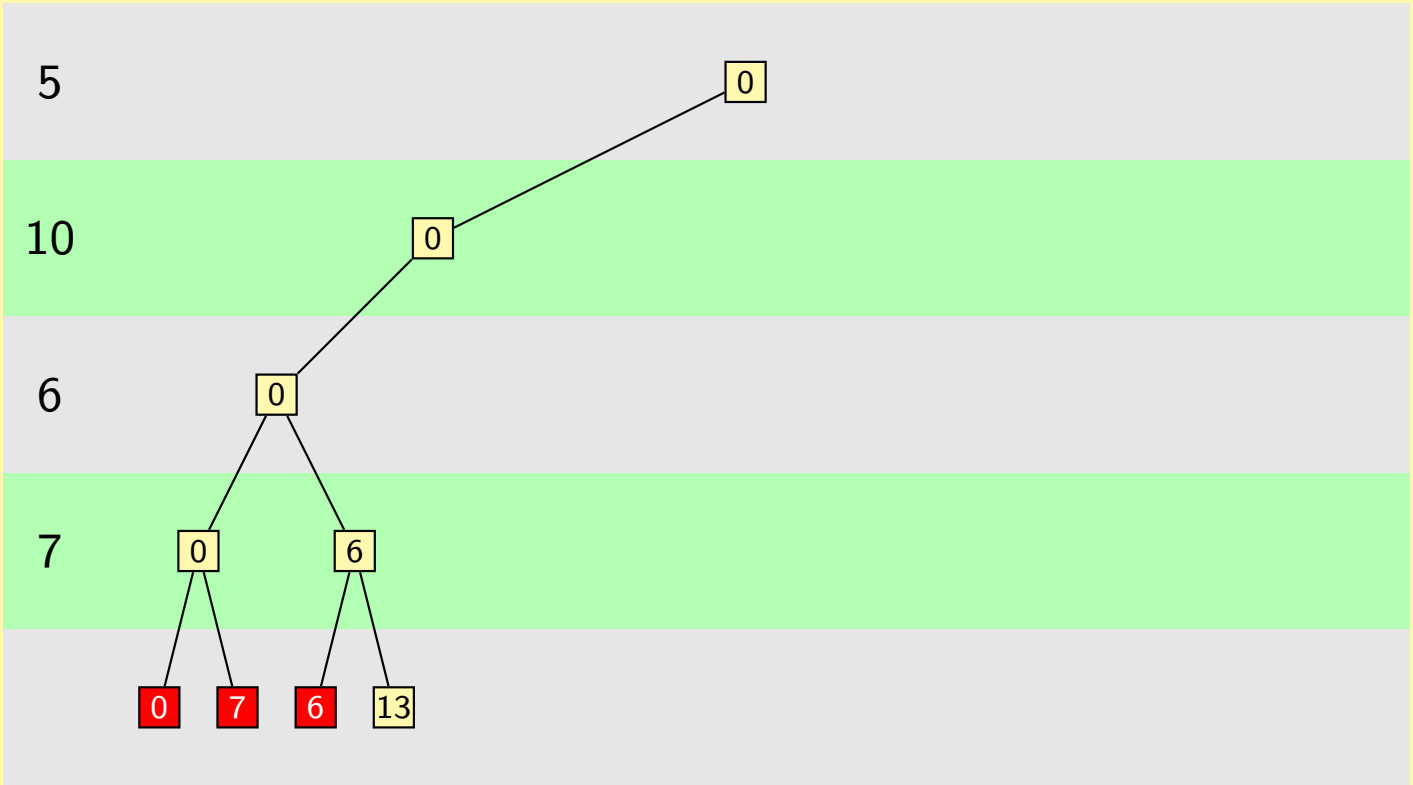
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



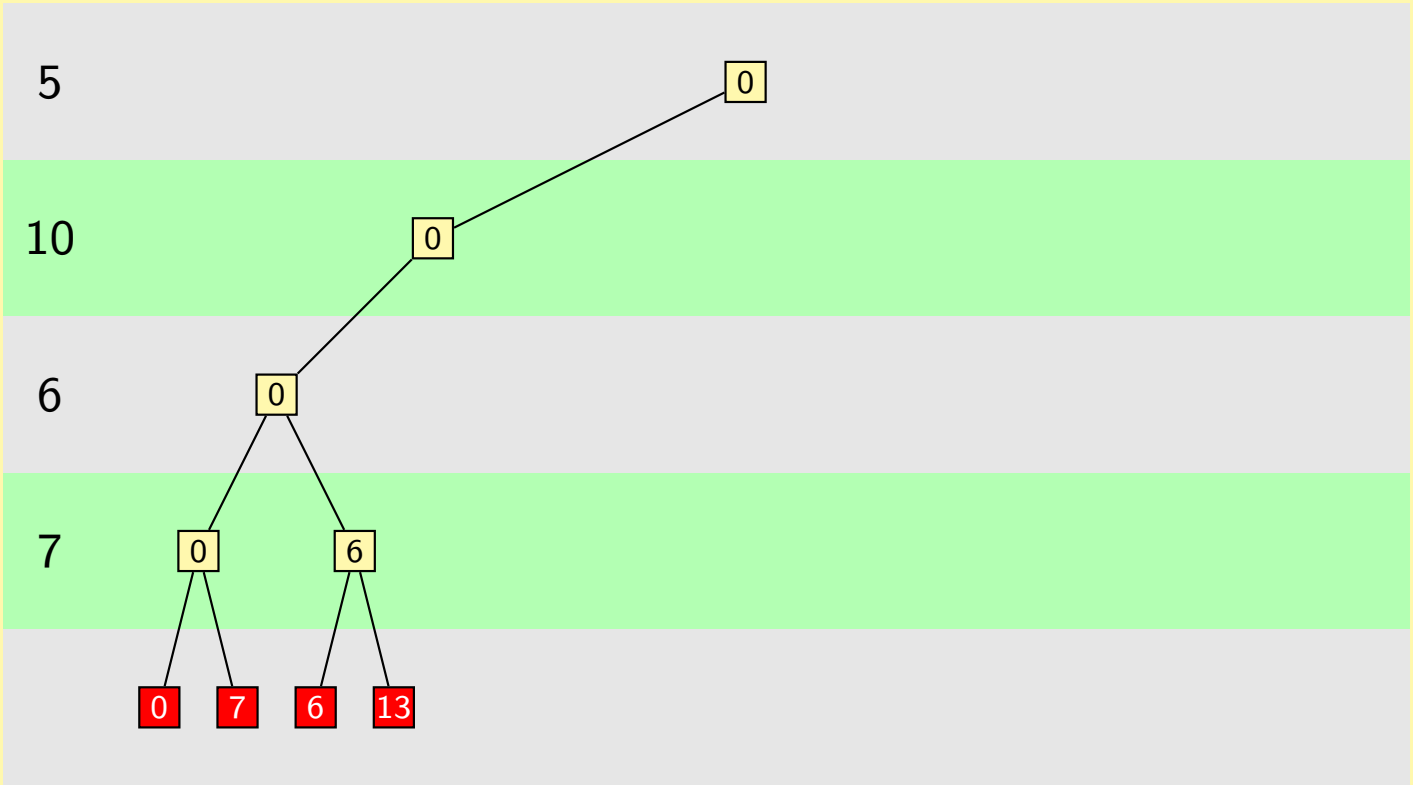
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



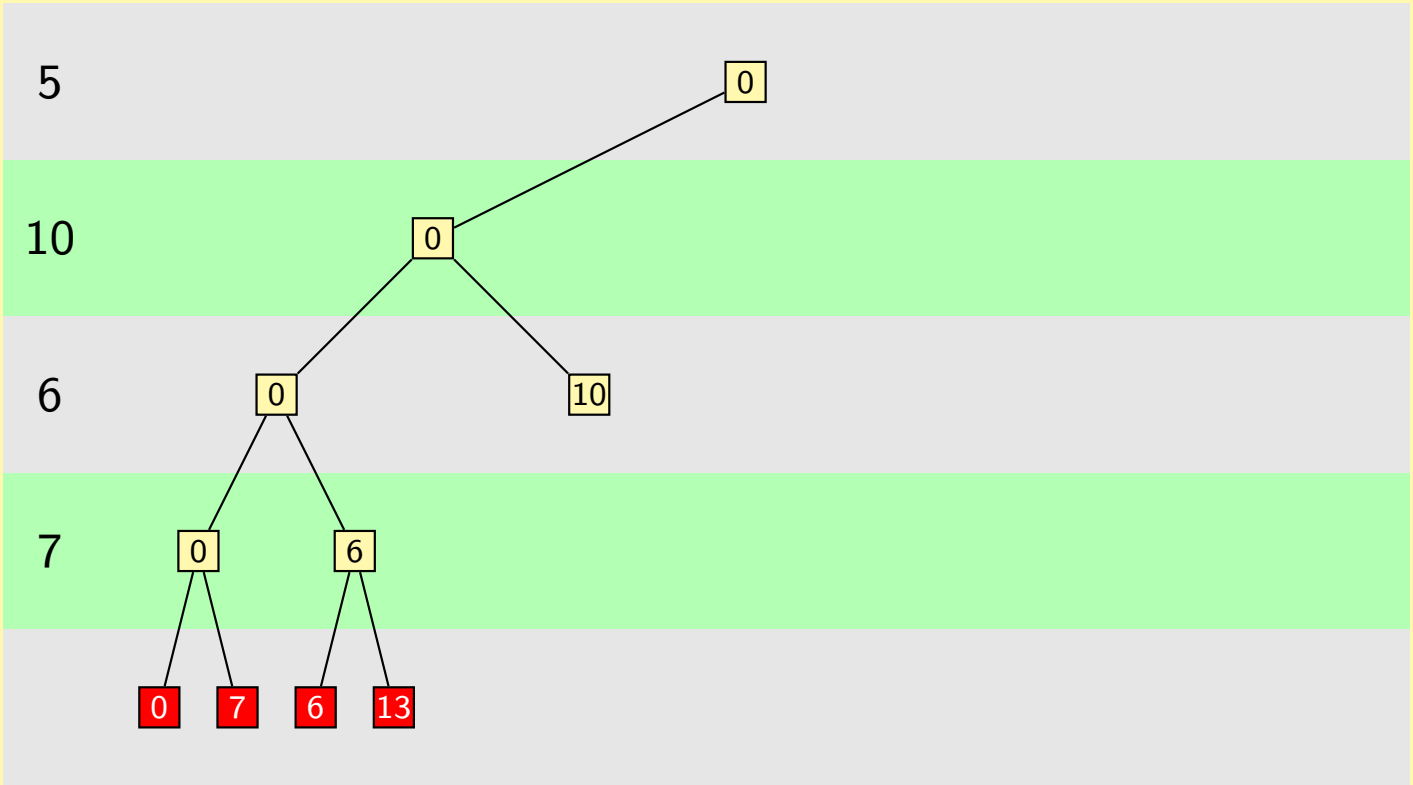
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



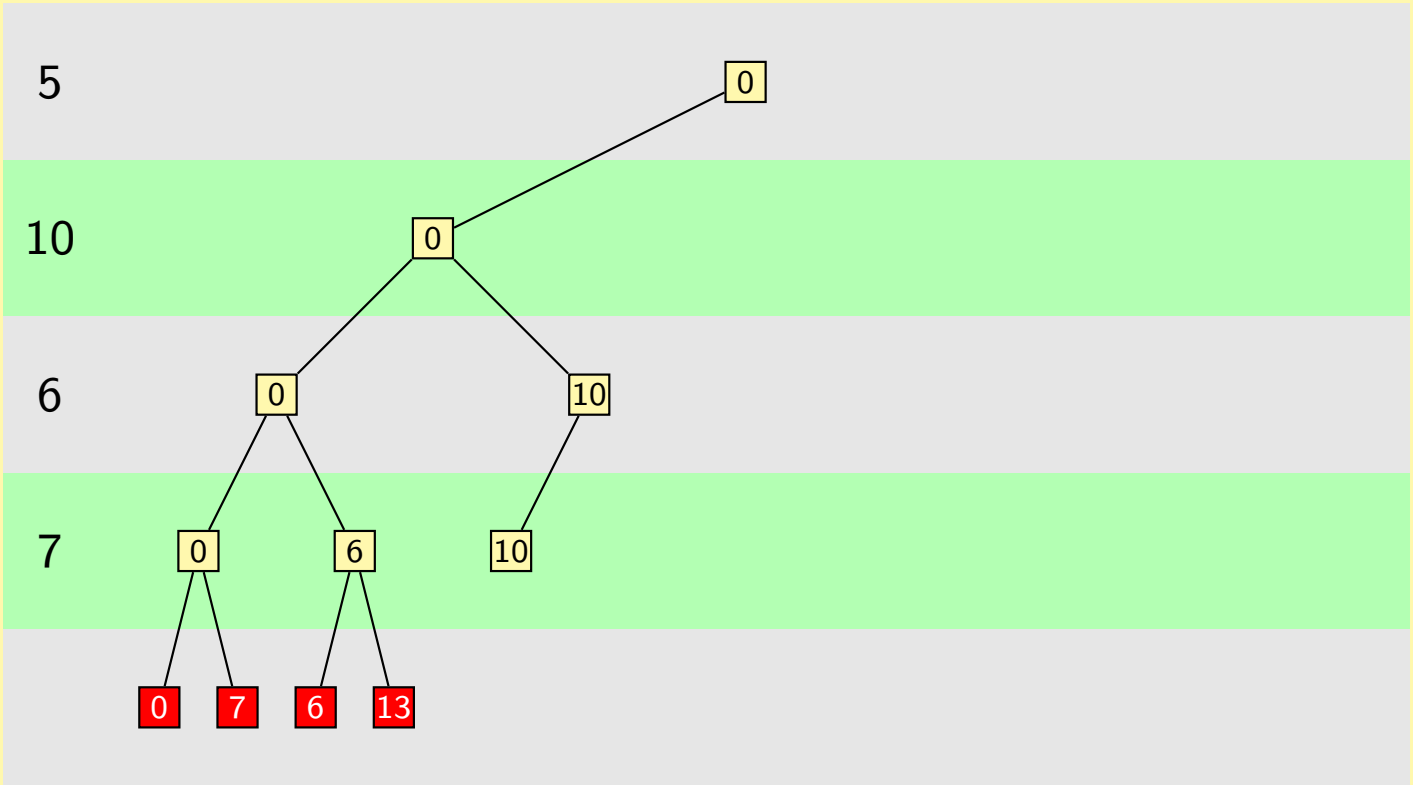
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



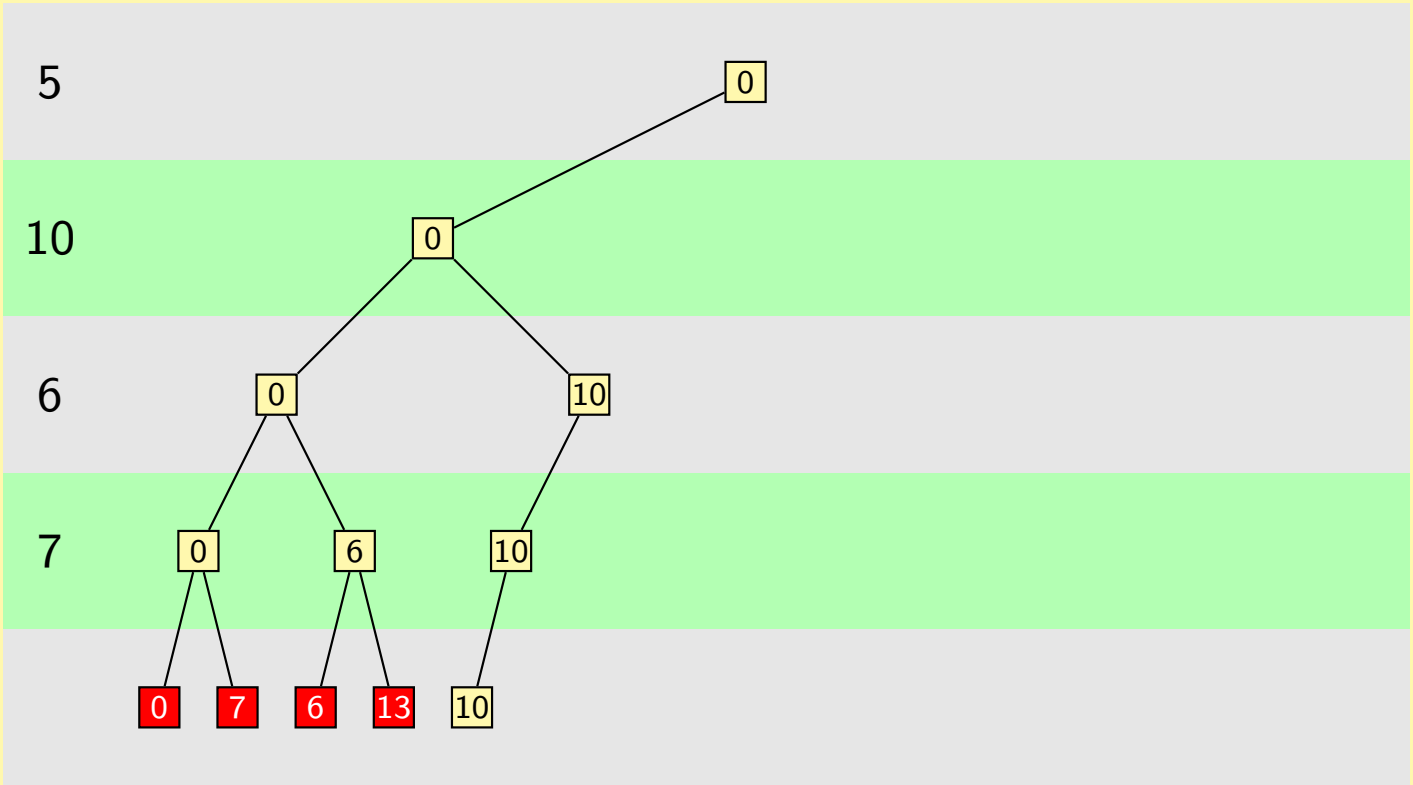
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



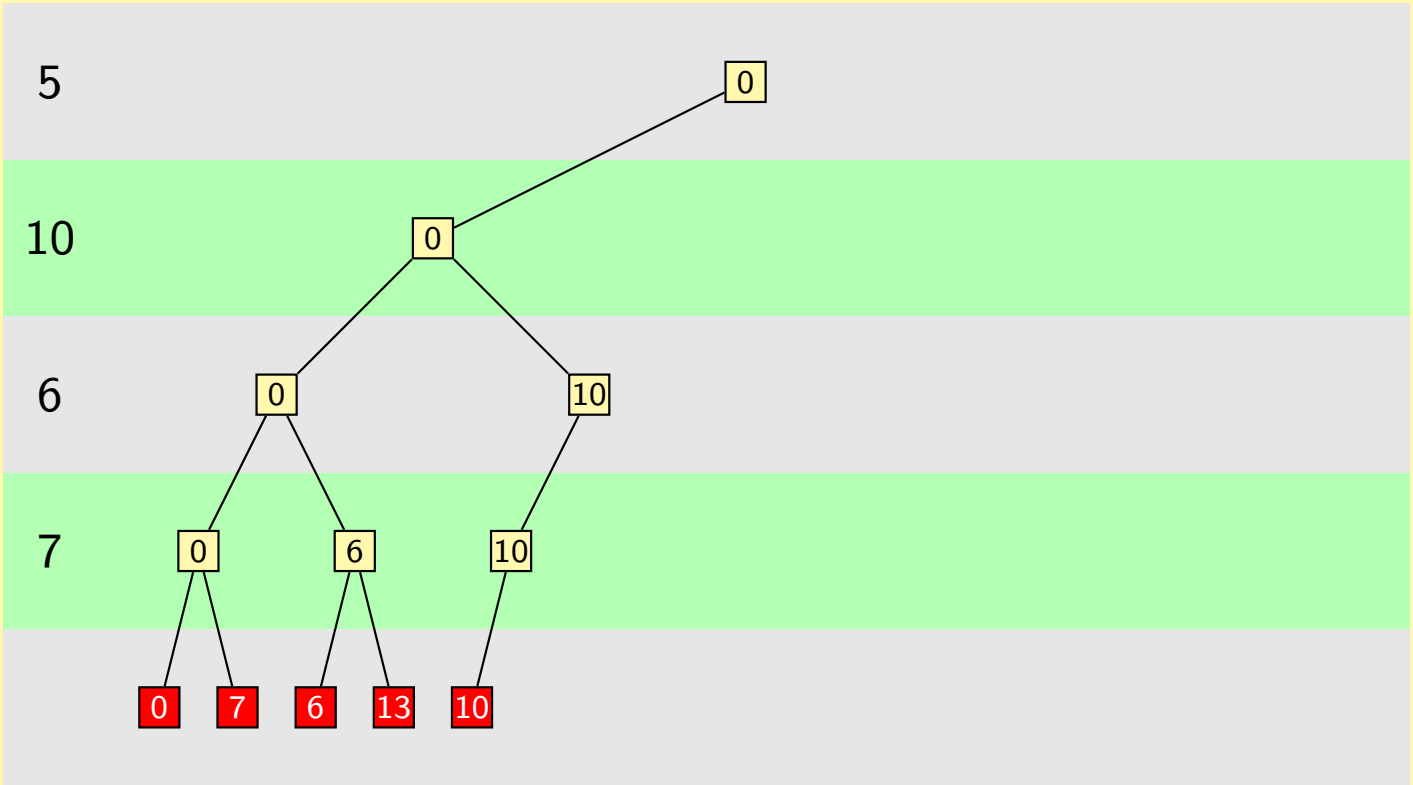
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



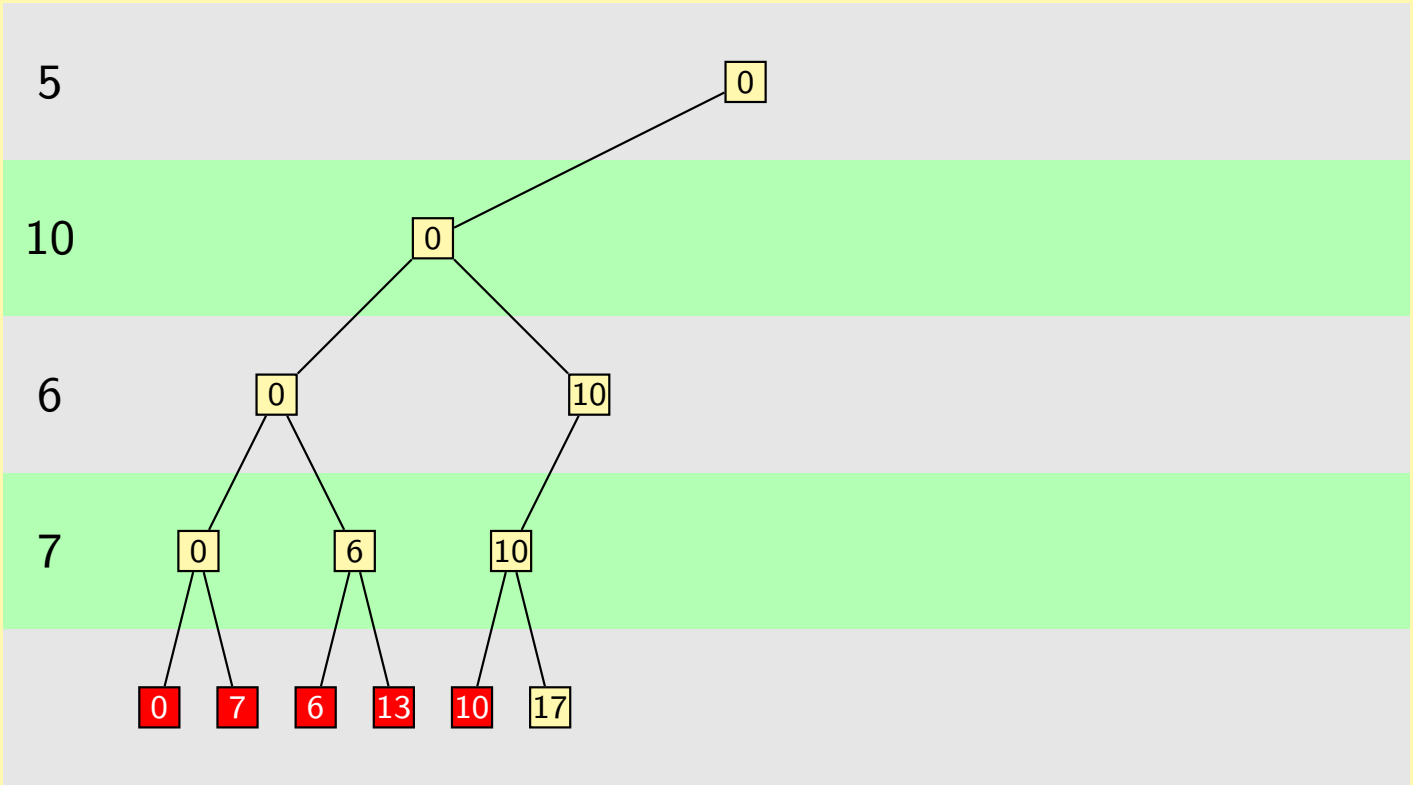
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



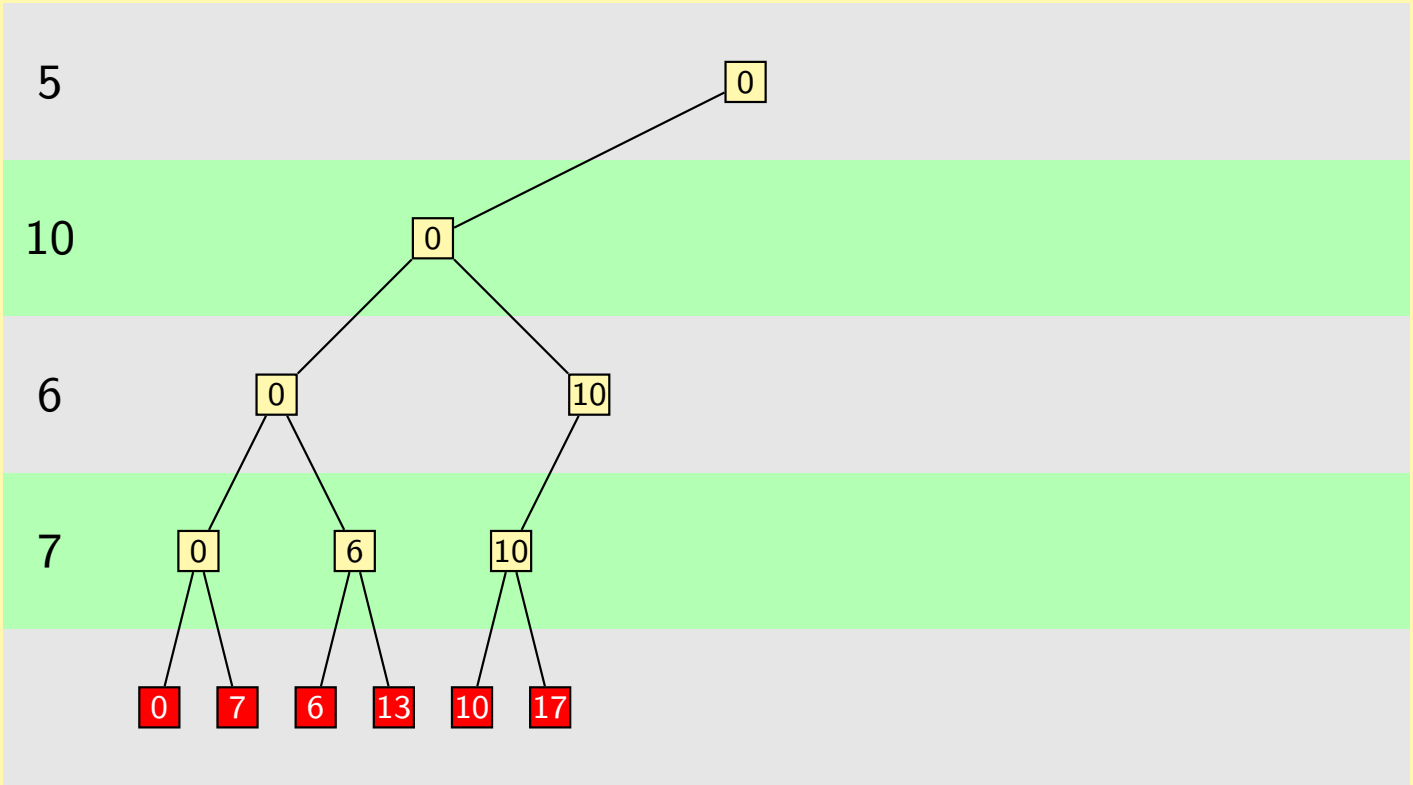
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



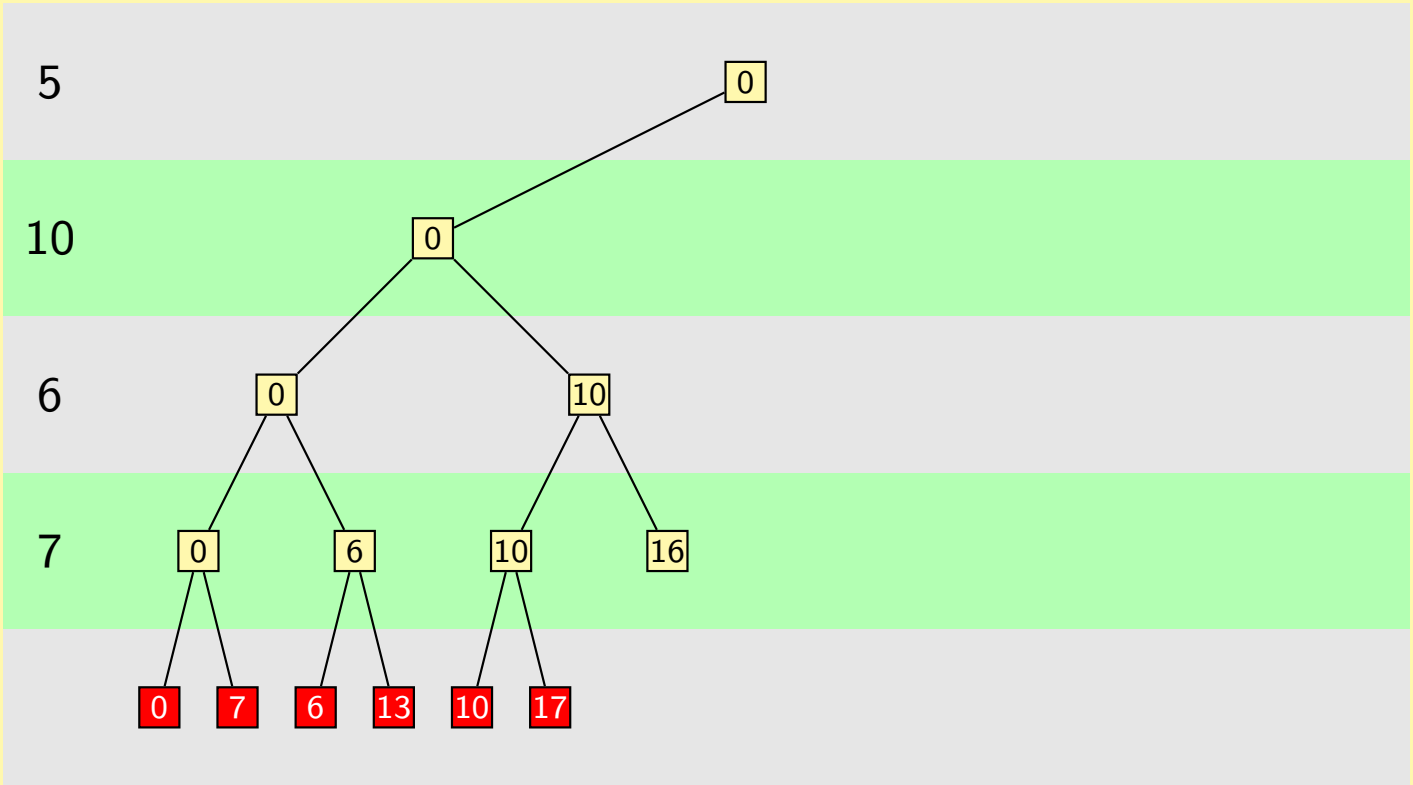
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



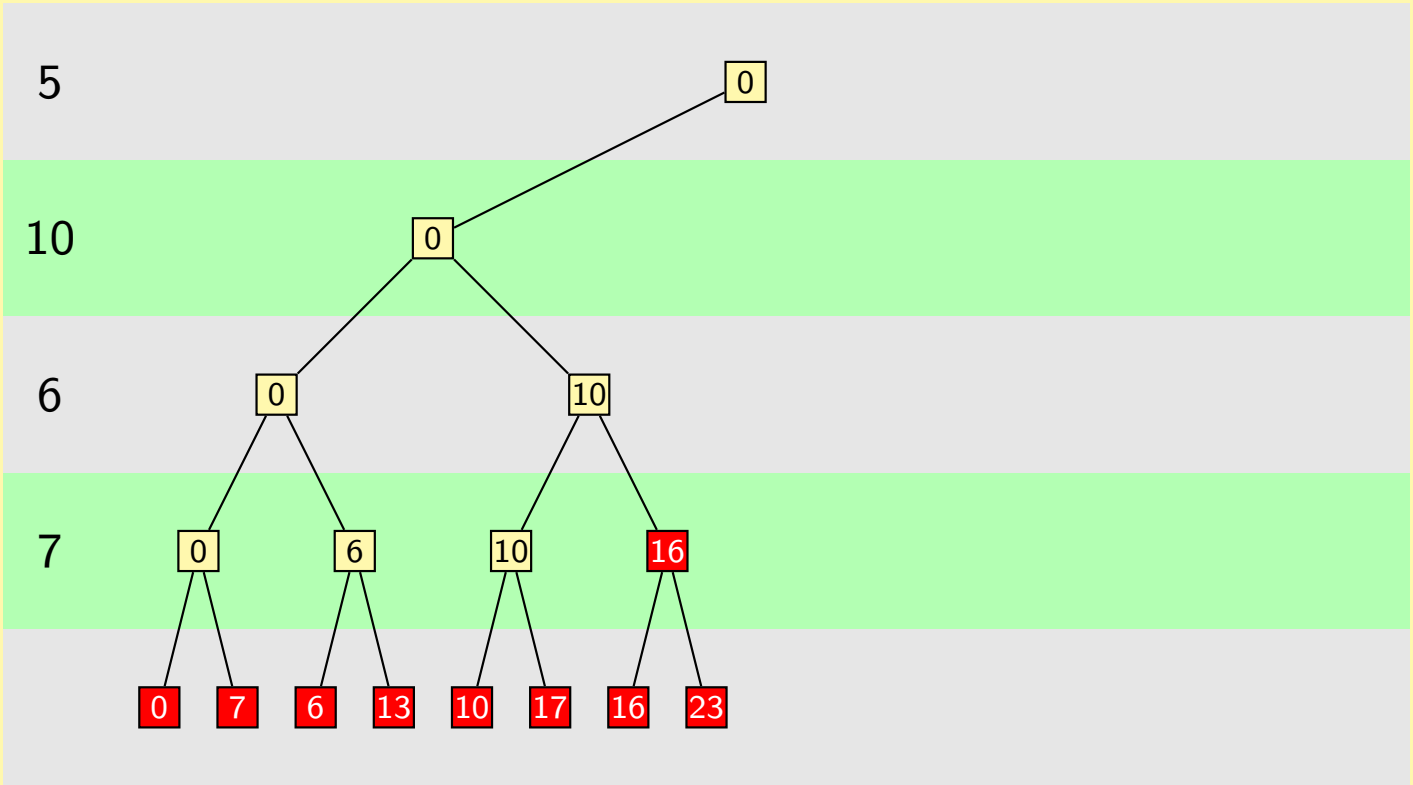
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



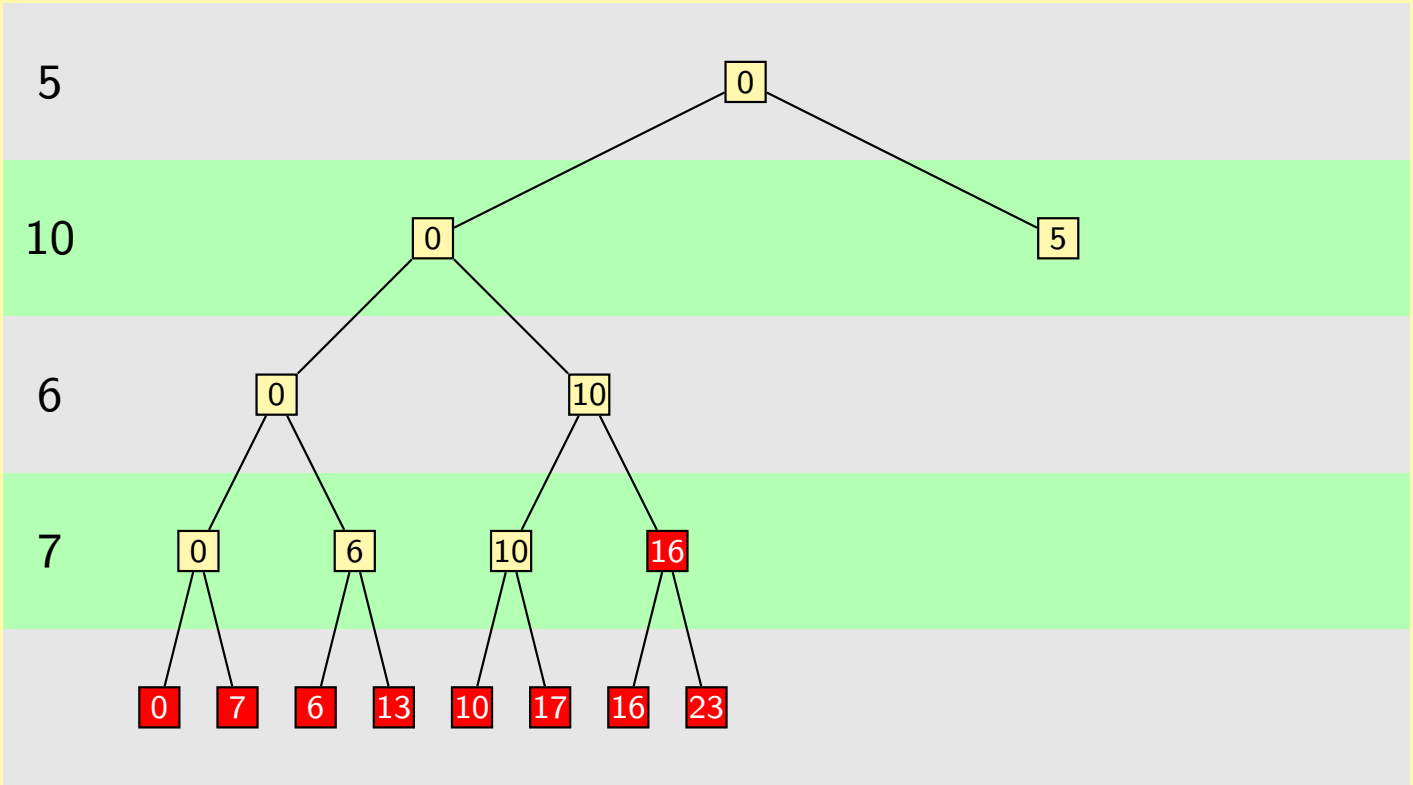
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



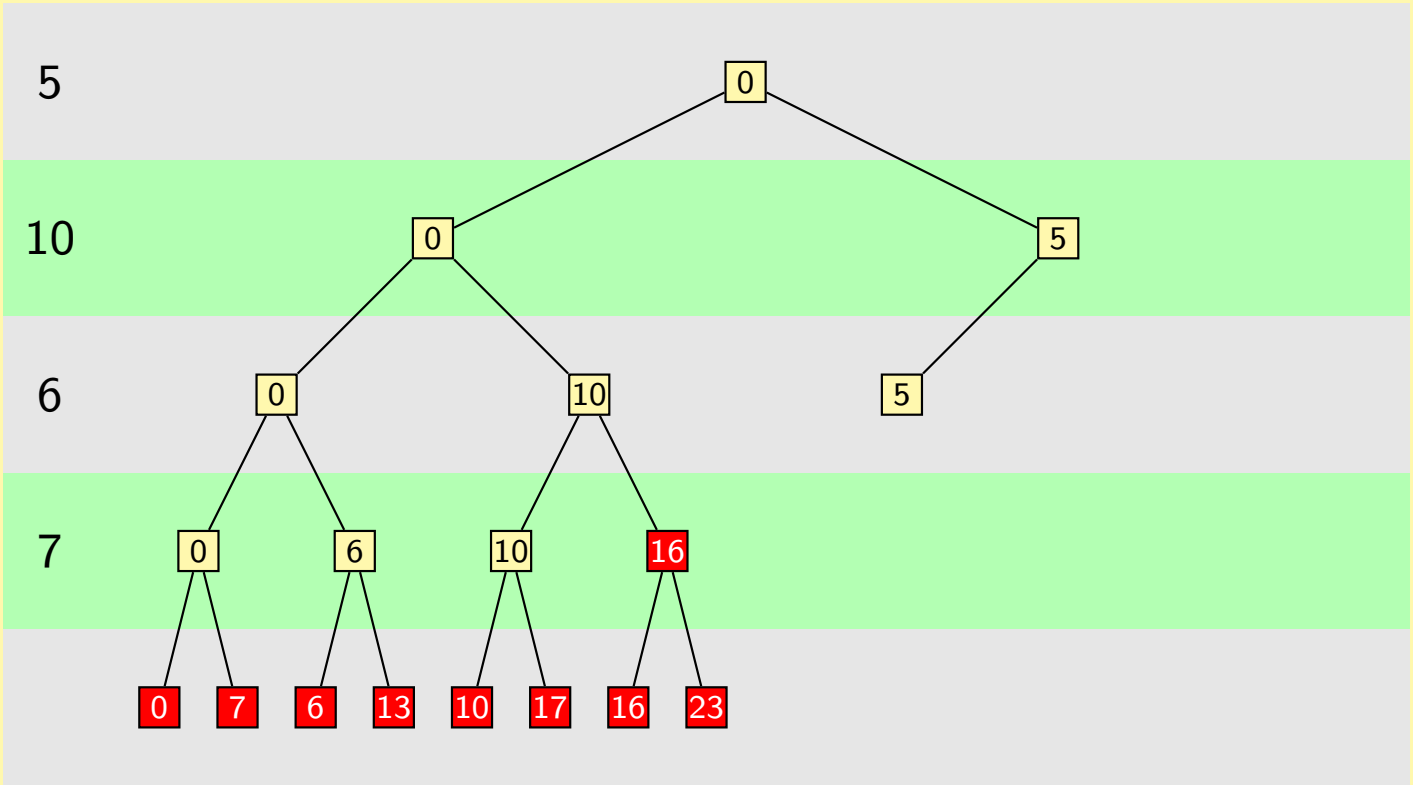
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



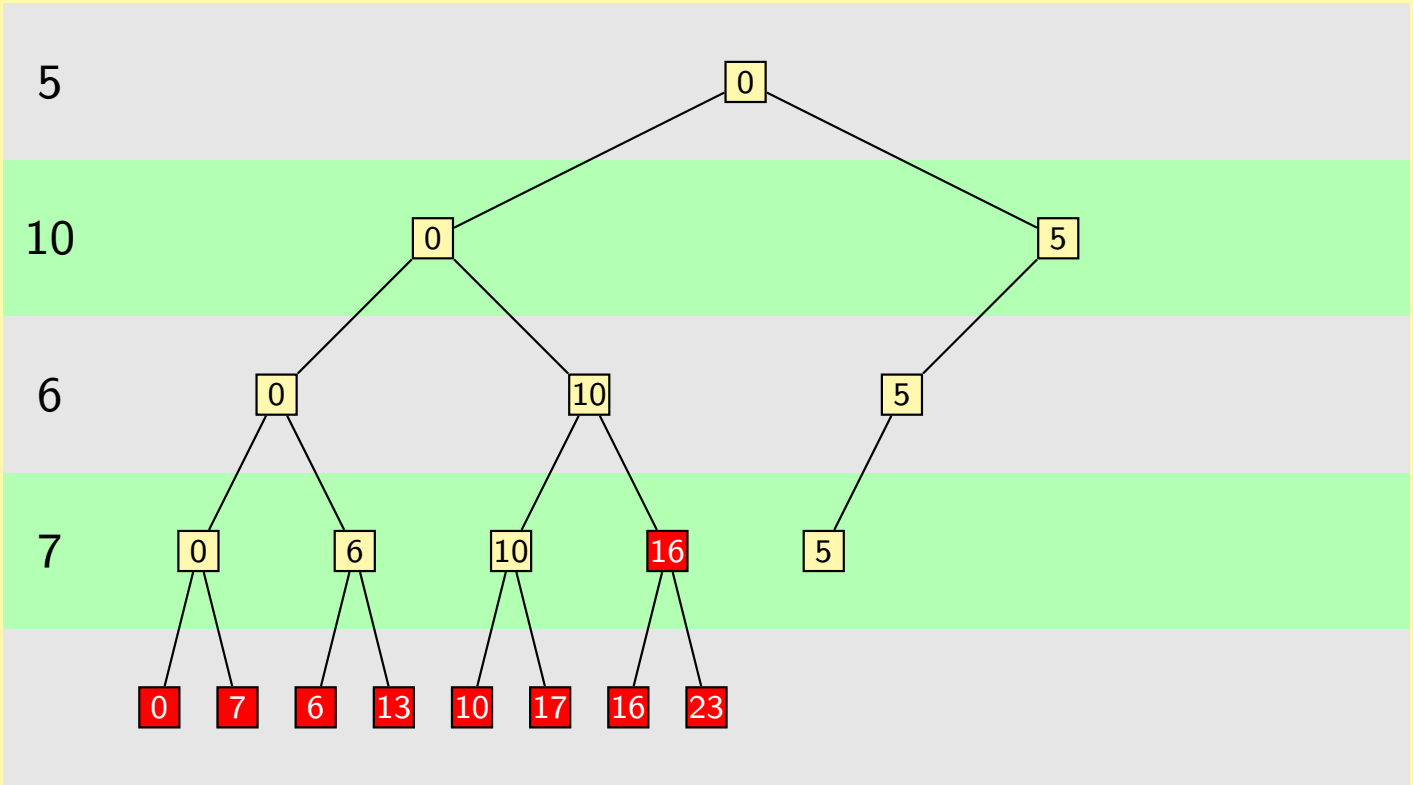
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



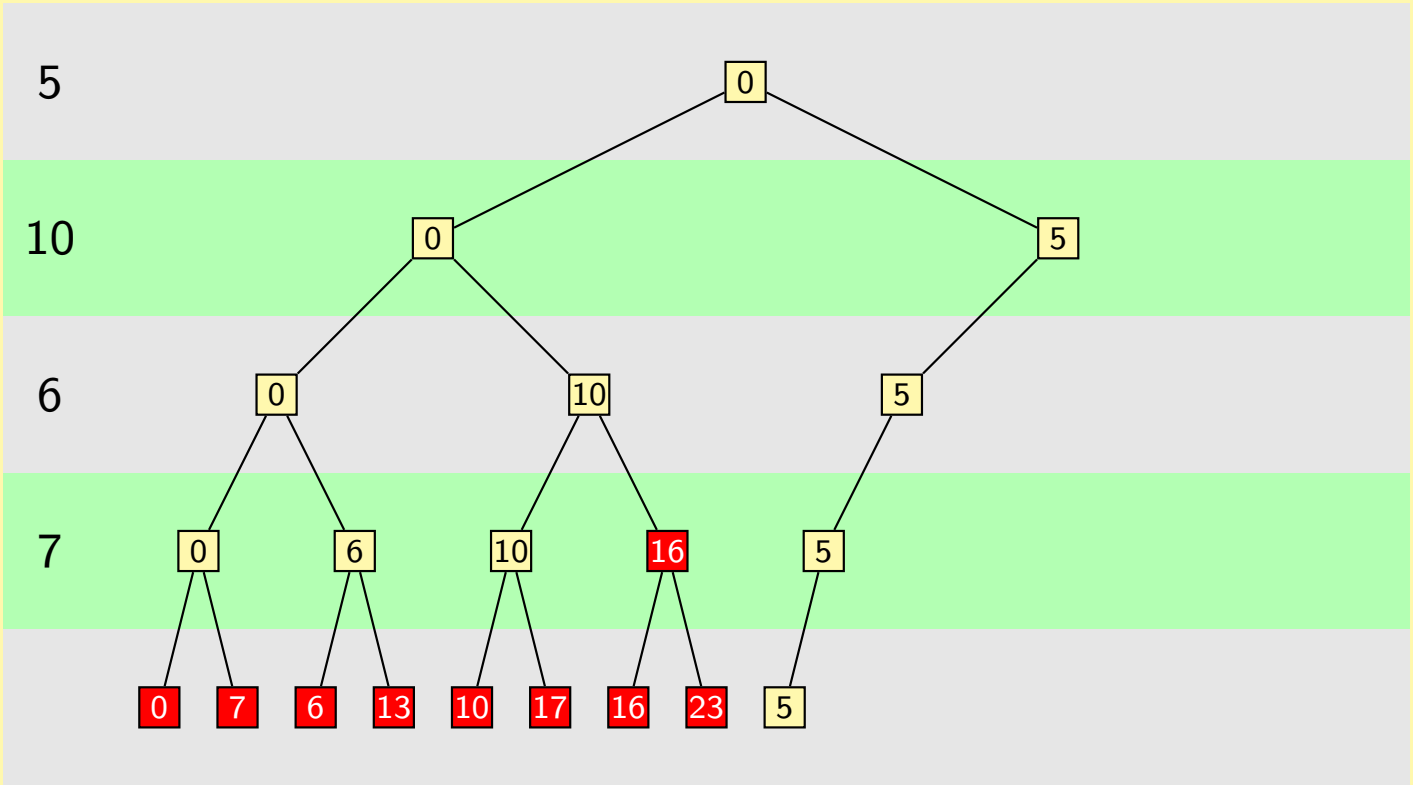
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



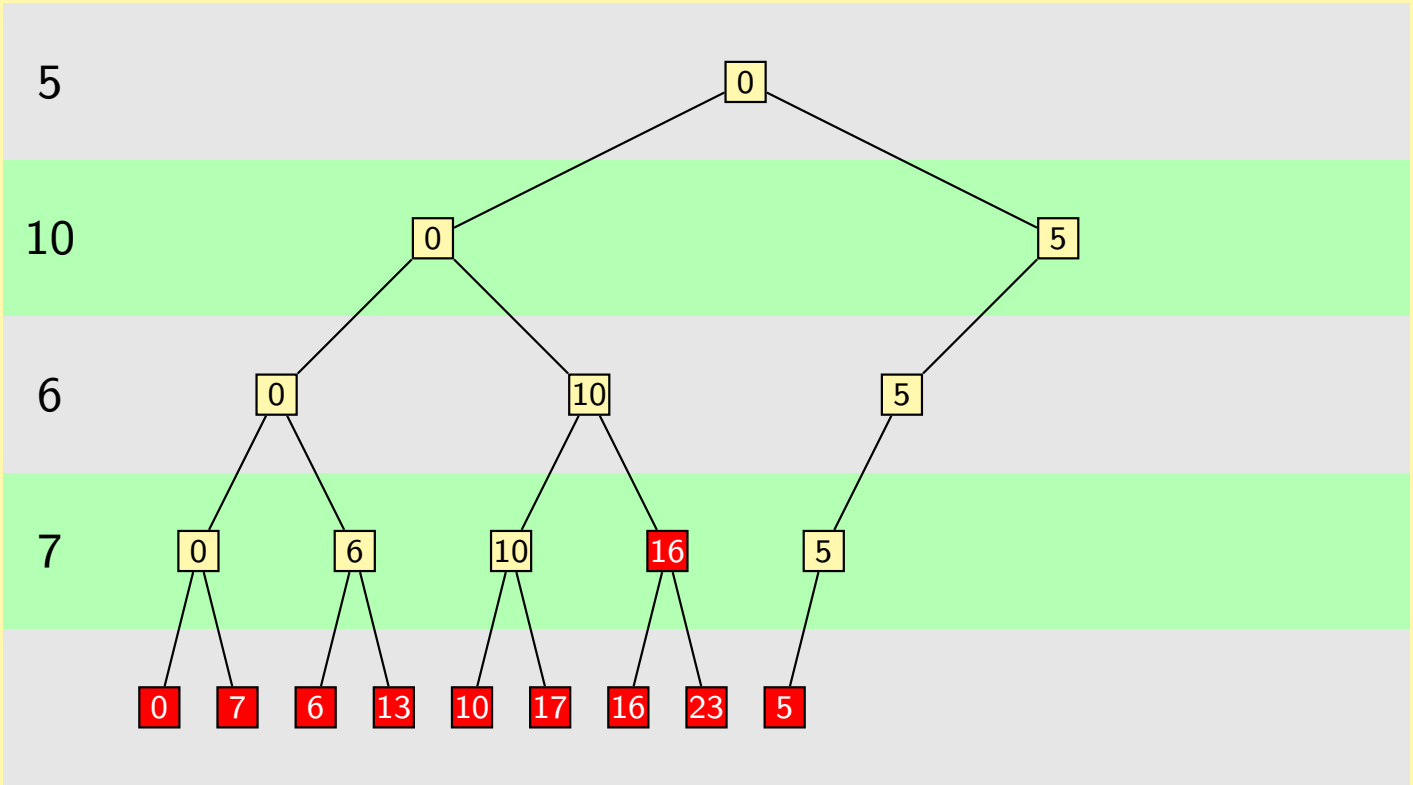
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



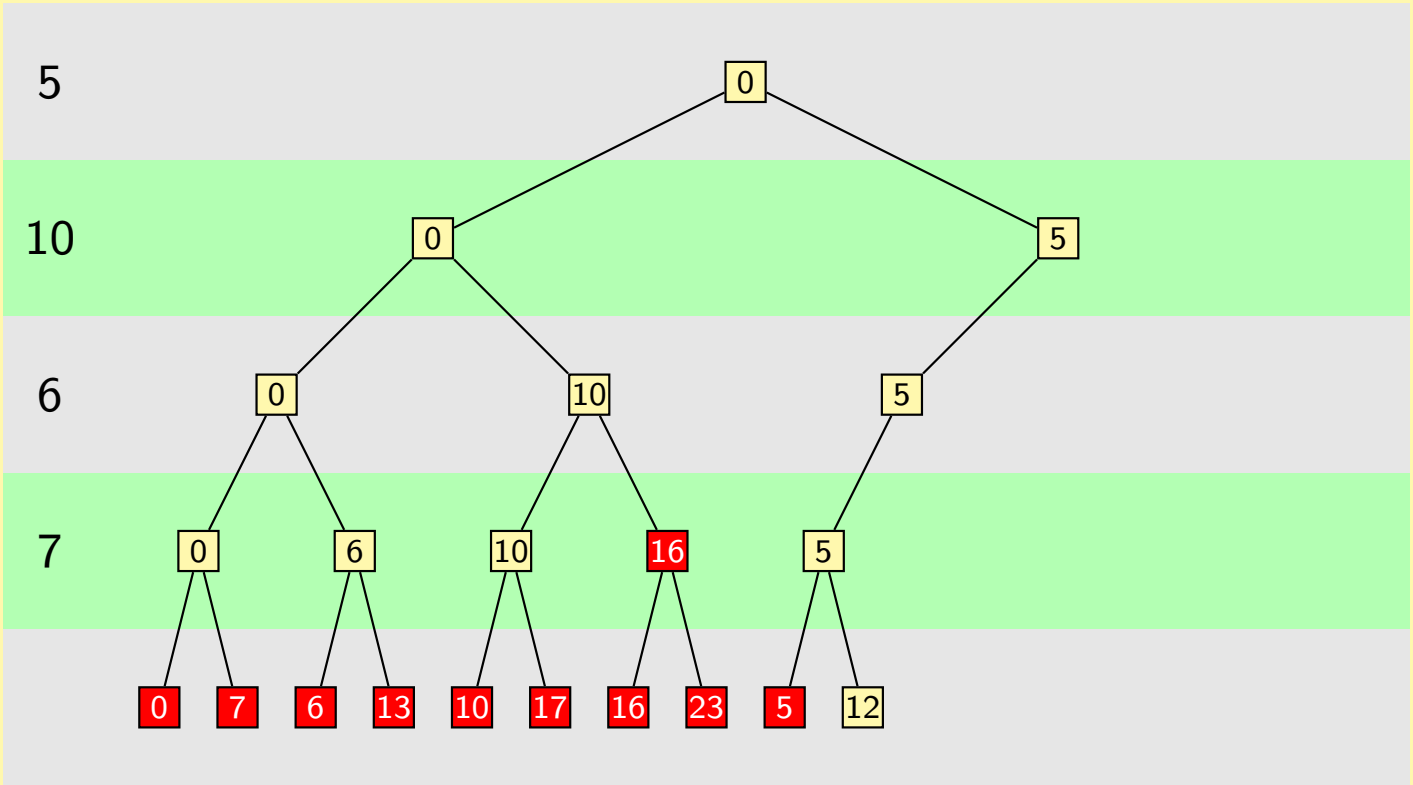
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



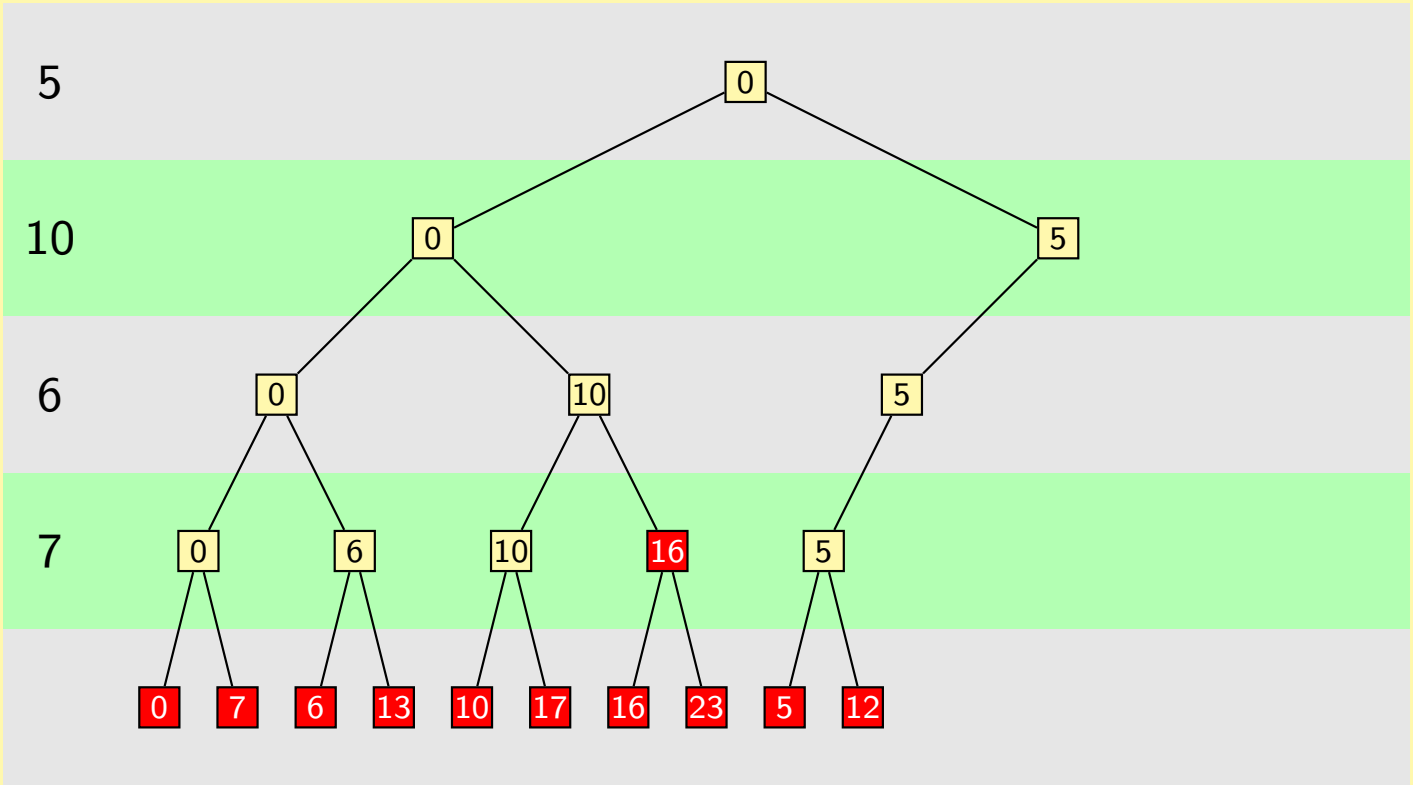
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



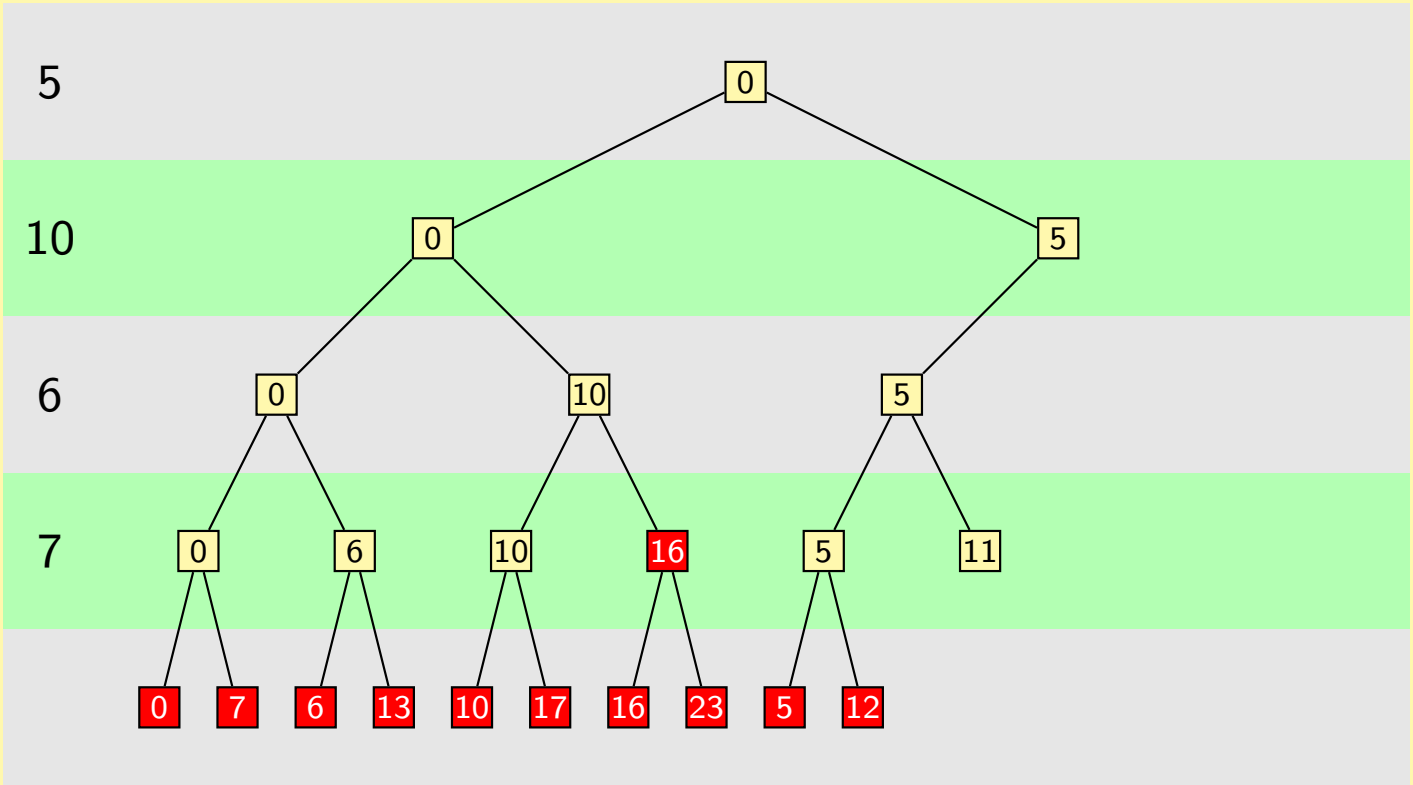
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



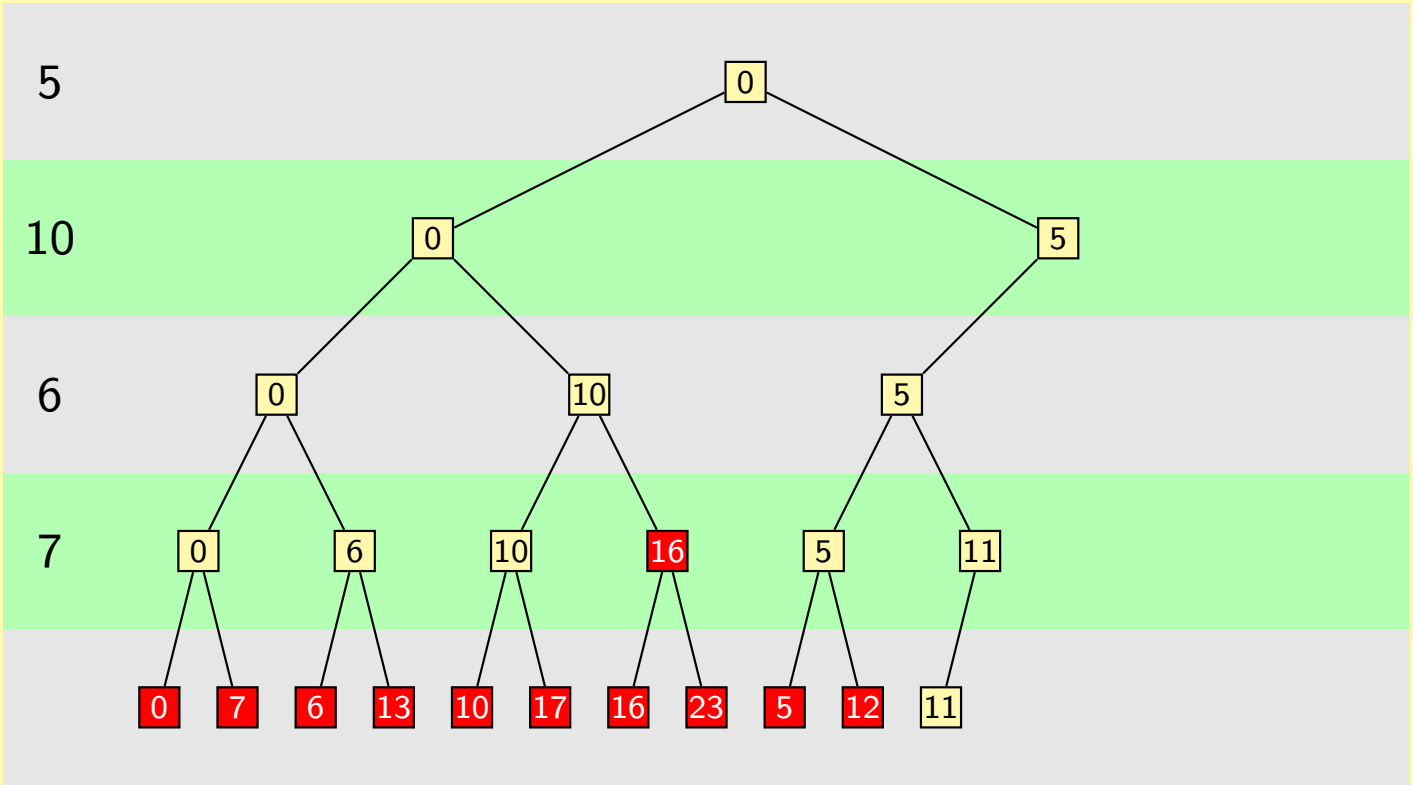
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



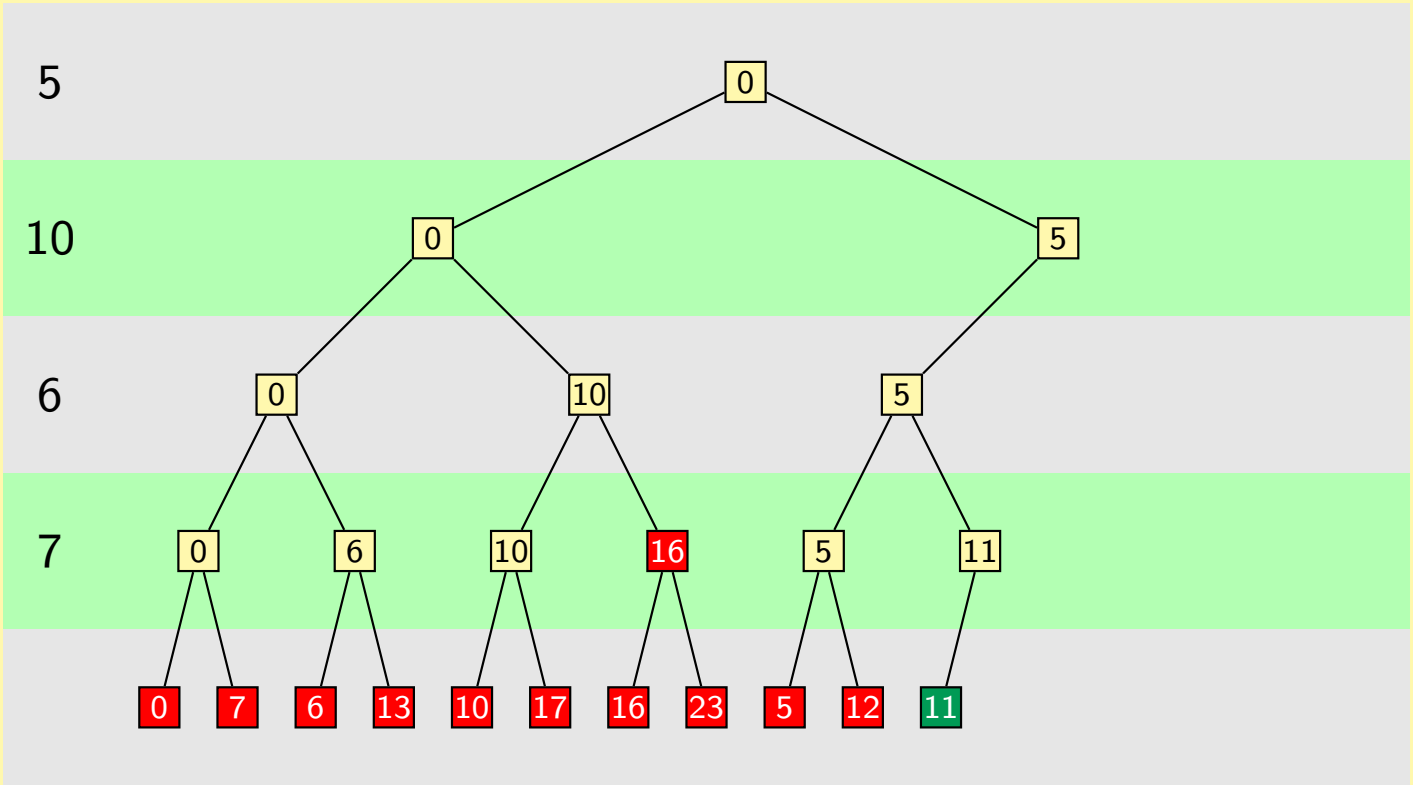
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



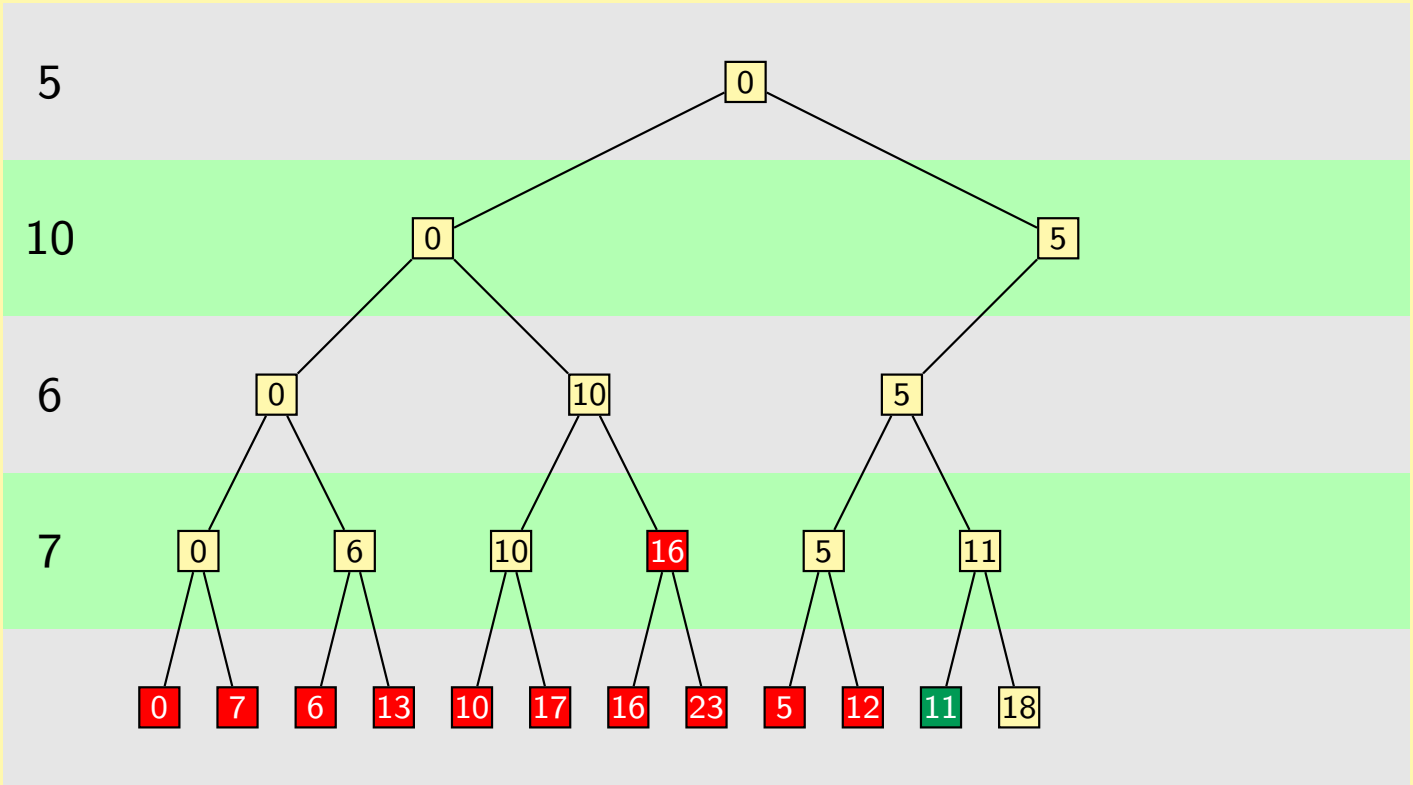
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



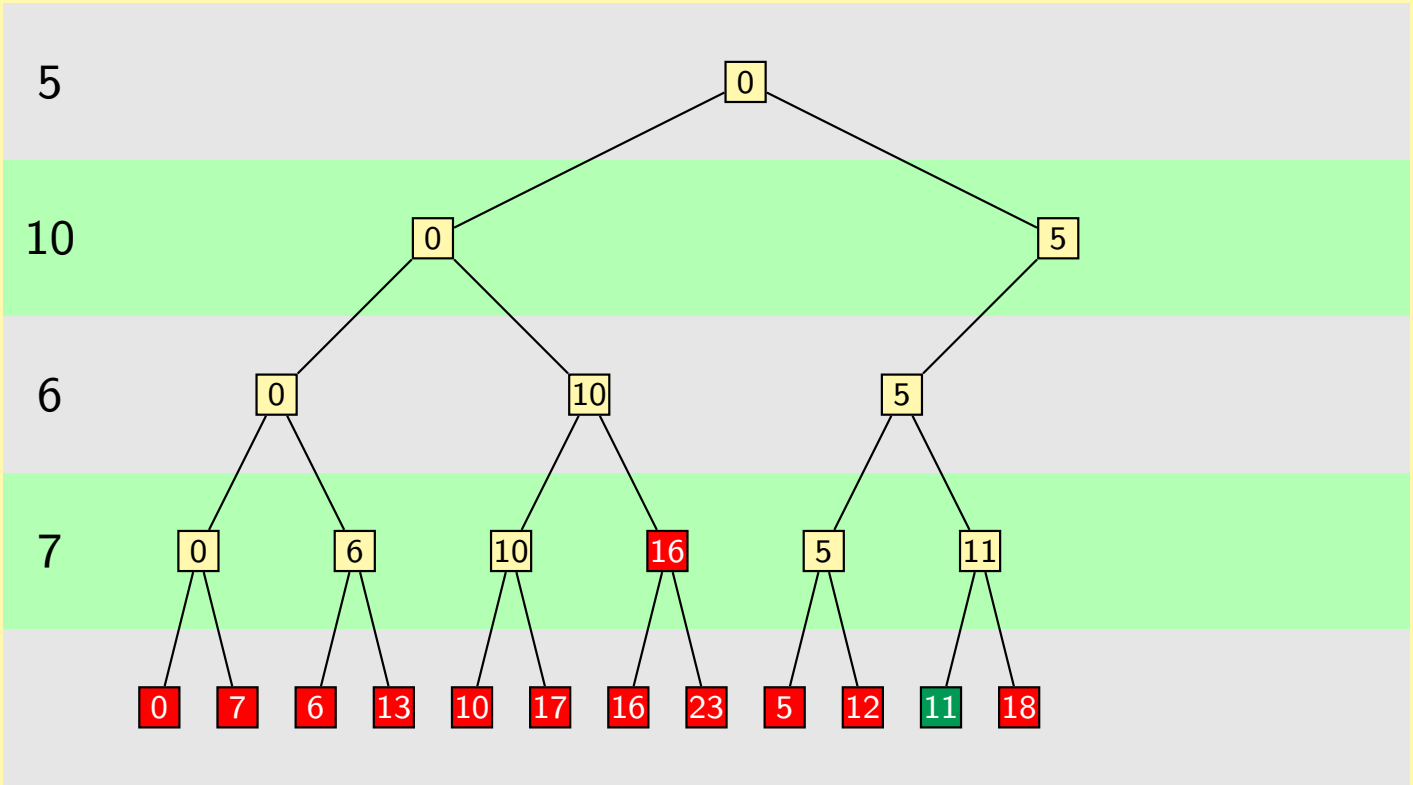
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



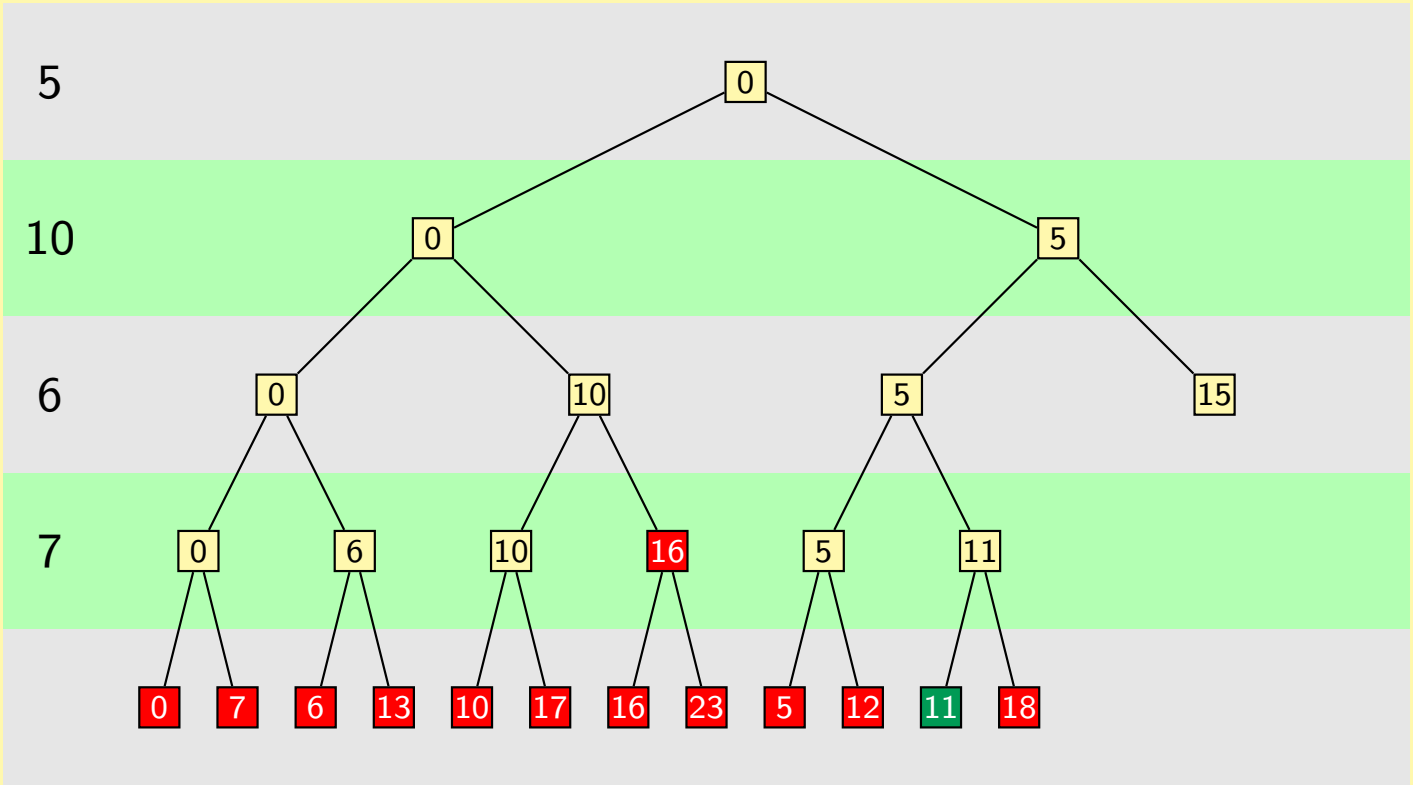
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



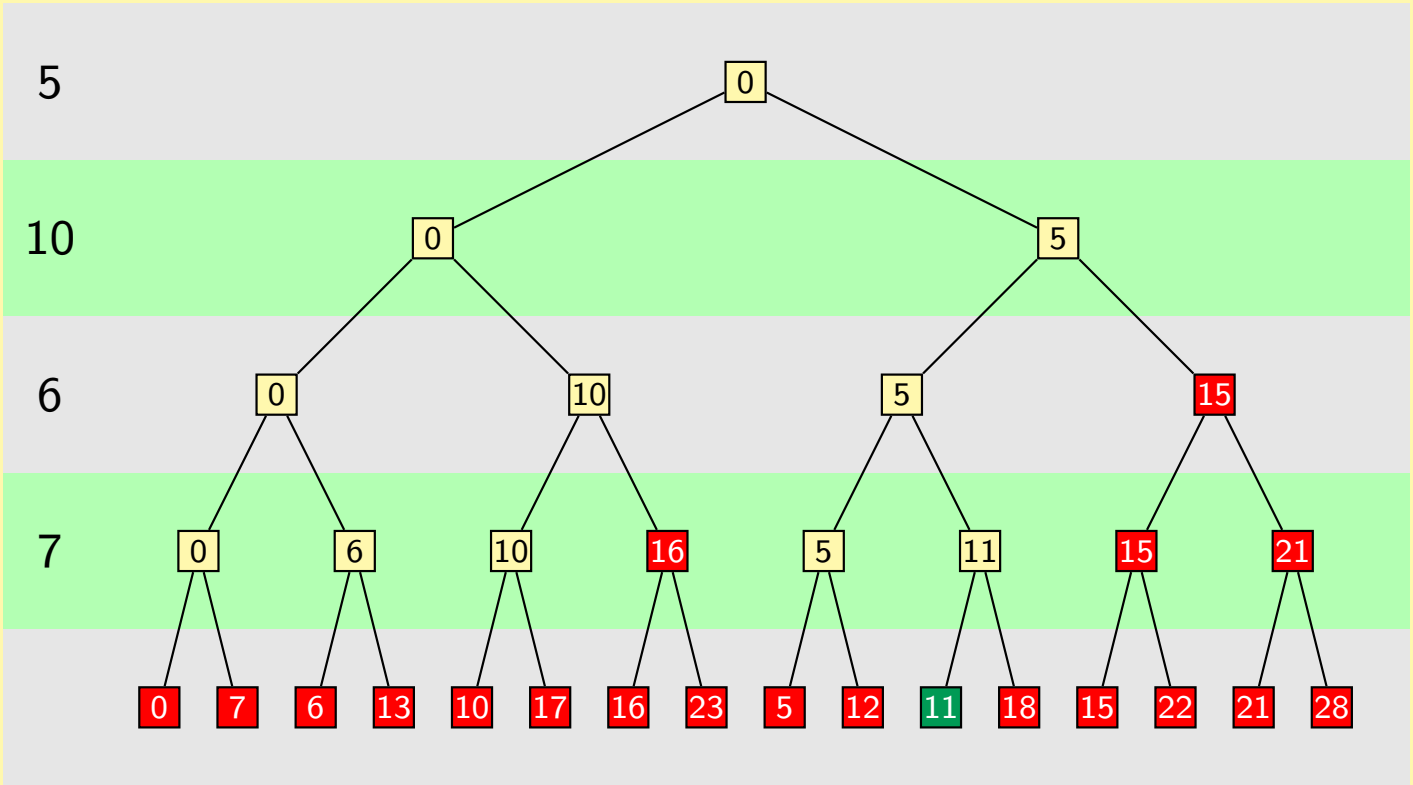
Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



Antes de seguir. Un ejemplo

$V = \{5, 10, 6, 7\}$
$m = 11$



El algoritmo para la suma del subconjunto

```
1   algoritmo SubSetSum
2   input v, ActSol: array [1..n] of integer;
3         m, etapa, ActSum: integer
4   for i = 0 to 1
5      ActSol[etapa] ← i
6      ActSum ← ActSum + v[etapa] * i
7      if etapa = n
8        if ActSum = m
9          display (ActSol)
10       endif
11     else
12       if ActSum ≤ m
13         SubSetSum(v, ActSol, m, etapa + 1, ActSum)
14       endif
15     endif
16  endfor
```

El algoritmo para la suma del subconjunto

```
1   algoritmo SubSetSum
2   input v, ActSol: array [1..n] of integer;
3         m, etapa, ActSum: integer
4   for i = 0 to 1
5      ActSol[etapa] ← i
6      ActSum ← ActSum + v[etapa] * i
7      if etapa = n
8        if ActSum = m
9          display (ActSol)
10       endif
11     else
12       if ActSum ≤ m
13         SubSetSum(v, ActSol, m, etapa + 1, ActSum)
14       endif
15     endif
16  endfor
```

Observe que $a = 2$; $b = 1$ y $k = 0$. Tenemos entonces $\mathcal{O}(2^n)$.

Ejercicios propuestos 1

1. Se tiene un conjunto $\mathbb{N}_k$ con los números naturales del 1 al k. Escriba un programa que muestre por pantalla todos los posibles subconjuntos $U \subseteq \mathbb{N}_k$.
2. Se tiene un grafo dirigido $G = (V, E)$ representado como una matriz de adyacencia. Escriba un programa que enumere todos los caminos de un vértice $s \in V$ cualquiera que pasen a lo sumo una vez por cada vértice.
3. Se tiene un conjunto de salas numeradas del 0 hasta n. Las salas están comunicadas entre sí a través de puertas que se abren solamente en el sentido de la numeración creciente (si una puerta permite pasar de la sala i a la sala j, entonces $i < j$.) Todas las salas tienen un puerta saliente, excepto, por supuesto, la salas n. Construya un algoritmo que permita ir desde la sala 0 a la sala n atravesando la máxima cantidad de salas.

Ejercicios propuestos 2

4. Se tiene un tablero de tamaño $n \times n$ y un rey en una casilla arbitraria (x_0, y_0). Por supuesto, $x_0, y_0 \in \{1, \ldots, n\}$. Cada casilla $(x,)$ del tablero tiene asignado un peso $T(x, y)$, de tal forma que a cada recorrido $\tau = (x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k)$ se le puede asignar un valor $V(R)$ que está determinado por la siguiente expresión:

$$V(R) = \sum_{i=0}^{k} i \cdot T(x_i, y_i)$$

El problema consiste en diseñar un algoritmo que proporcione el recorrido de peso mínimo que visite todas las casillas del tablero sin repetir ninguna. Recuerde que un rey de ajedrez puede mover a cualquier casilla vecina en cualquier dirección.