# Estrategias algorítmicas

Tema 3(III)

Algorítmica y Modelos de Computación

1. Introducción.
2. Algoritmos divide y vencerás.
3. Algoritmos voraces.
4. Programación dinámica.
5. Algoritmos Bactracking (vuelta atrás).
6. Ramificación y poda.

# 5. Algoritmos vuelta atrás (Bactracking).

1. Introducción. Características generales.
2. Esquema general.
3. Análisis de tiempos de ejecución.
4. Ejemplos de aplicación.

    4.1. Problema de la mochila 0-1.

    4.2. Problema de la asignación.

    4.3. Resolución de juegos.

# 5. Algoritmos Bactracking. Introducción. Características generales.

- El **backtracking** (**retroceso** o **vuelta atrás**) es una técnica general de resolución de problemas aplicable en problemas de **optimización**, **juegos** y otros tipos.
- Las técnicas vistas hasta ahora intentan construir la solución basándose en ciertas propiedades de esta. Sin embargo, ciertos problemas no pueden solucionarse con ninguna de las técnicas anteriores: la única manera de resolver estos problemas es a través de un **estudio exhaustivo** de un conjunto de posibles soluciones.
- Los algoritmos exhaustivos pueden generar y explorar todas las posibles combinaciones (incluidas aquellas que no llevan a una solución).
- **Inconveniente**: Tardan mucho en encontrar la solución ya que pierden el tiempo explorando combinaciones inútiles que no conducen a ninguna solución.
- La técnica de backtracking permite realizar este estudio exhaustivo evitando tener que generar todas las posibles combinaciones ya que cada vez que se toma una decisión permite deshacerla (vuelta atrás) si no lleva a una solución o, en el caso, de optimización, la solución a la que lleva no mejora la que tenemos hasta ahora.
- **Ventaja**: Se alcanza antes la solución (al evitar tener que explorar aquellas inútiles).

# 5. Algoritmos Bactracking. Introducción. Características generales.

- El **backtracking** realiza una **búsqueda exhaustiva y sistemática en el espacio de soluciones**. Por ello, suele resultar muy ineficiente.
- Se puede entender como “opuesto” a avance rápido:
  - **Avance rápido:** añadir elementos a la solución y no deshacer ninguna decisión tomada.
  - **Backtracking:** añadir y quitar todos los elementos. **Probar todas las combinaciones.**
- Cada **solución es el resultado de una secuencia de decisiones**
  - Pero a diferencia del método voraz, las decisiones pueden deshacerse ya sea porque no lleven a una solución o porque se quieran explorar todas las soluciones (para obtener la solución óptima)
- Existe una **función objetivo** que debe ser satisfecha u optimizada por cada selección

## 5. Algoritmos Bactracking. Características generales.

- Las etapas por las que pasa el algoritmo se pueden representar mediante un **árbol de expansión** (o ***árbol del espacio de estados***).
- El árbol de expansión no se construye realmente, sino que está implícito en la ejecución del algoritmo
- Cada **nivel** del árbol representa una etapa de la secuencia de decisiones
- Una **solución** se puede expresar como una tupla: $\mathbf{(x_1, x_2, ..., x_n)}$, satisfaciendo unas restricciones y tal vez optimizando cierta función objetivo.
- En cada momento, el algoritmo se encontrará en cierto nivel **k**, con una solución parcial $(x_1, ..., x_k)$.
  - Si se puede añadir un nuevo elemento a la solución $\mathbf{x_{k+1}}$, se genera y se avanza al nivel **k+1**.
  - Si no, se prueban otros valores para $\mathbf{x_k}$.
  - Si no existe ningún valor posible por probar, entonces se retrocede al nivel anterior **k-1**.
  - Se sigue hasta que la solución parcial sea una solución completa del problema, o hasta que no queden más posibilidades por probar.

## 5. Algoritmos Bactracking. Características generales.

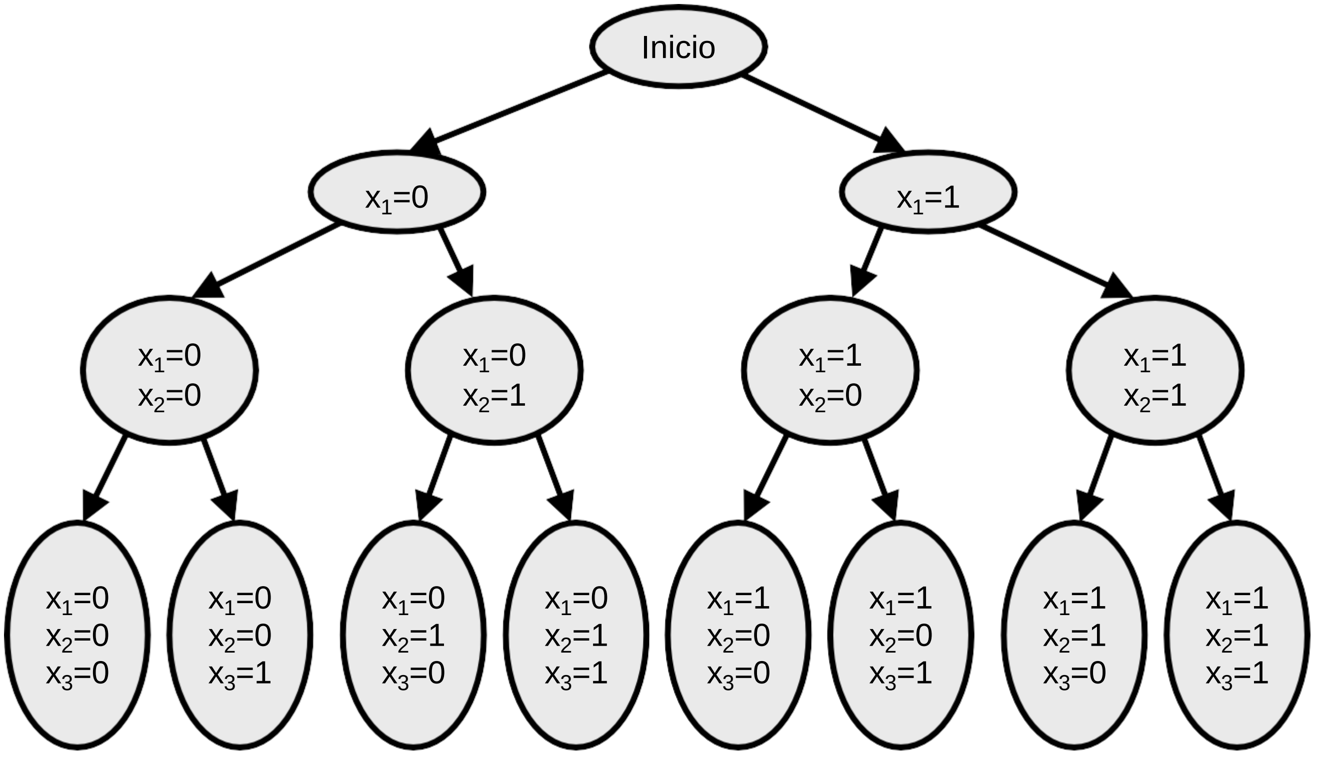
- Las **soluciones** del problema se pueden representar como una n-tupla: $\mathbf{(x_1, x_2, ..., x_n)}$
- El **objetivo** consiste en encontrar soluciones factibles.
- La idea consiste en **construir el vector solución elemento a elemento** usando una **función factible** modificada para estimar si una solución parcial o incompleta tiene alguna posibilidad de éxito.
- Cada una de las tuplas $\mathbf{(x_1, x_2, ..., x_i}$; ?) donde $i \leq n$ se denomina un **estado** y denota un conjunto de soluciones.
- Un estado puede ser:
  - estado **terminal** o solución: conjunto con un solo elemento (solución);
  - estado **no terminal** o solución parcial: representa implícitamente un conjunto de varias soluciones;
- Se dice que un **estado no terminal** es **factible** o prometedor cuando no se puede descartar que contenga alguna solución factible.
- El conjunto de estados se organiza formando un **árbol de estados**.

## Resumen:

- Los algoritmos **backtracking** determinan las **soluciones**, representadas como una n-tupla: **($x_1$, $x_2$, …, $x_n$)**, realizando una **búsqueda exhaustiva y sistemática** en el espacio de soluciones del problema.
- Esta **búsqueda** se puede representar en el **árbol de soluciones** asociado al conjunto de soluciones del problema.
- Cada tupla **($x_1$, $x_2$, ..., $x_i$**; ?) $i \leq n$ se denomina un **estado** y representa cada uno de los **nodos del árbol**.
- Un estado puede ser:
  - estado **terminal** o solución: Aquel estado (nodo) del problema (árbol) para el que el camino desde la raíz hasta el nodo representa una n-tupla (solución) que satisface todas las restricciones implícitas del problema;
  - estado **no terminal** o solución parcial: representa implícitamente un conjunto de varias soluciones;
- Se dice que un **estado no terminal** es **factible** o prometedor cuando no se puede descartar que contenga alguna solución factible.
- El conjunto de estados se organiza formando un **árbol de estados**.

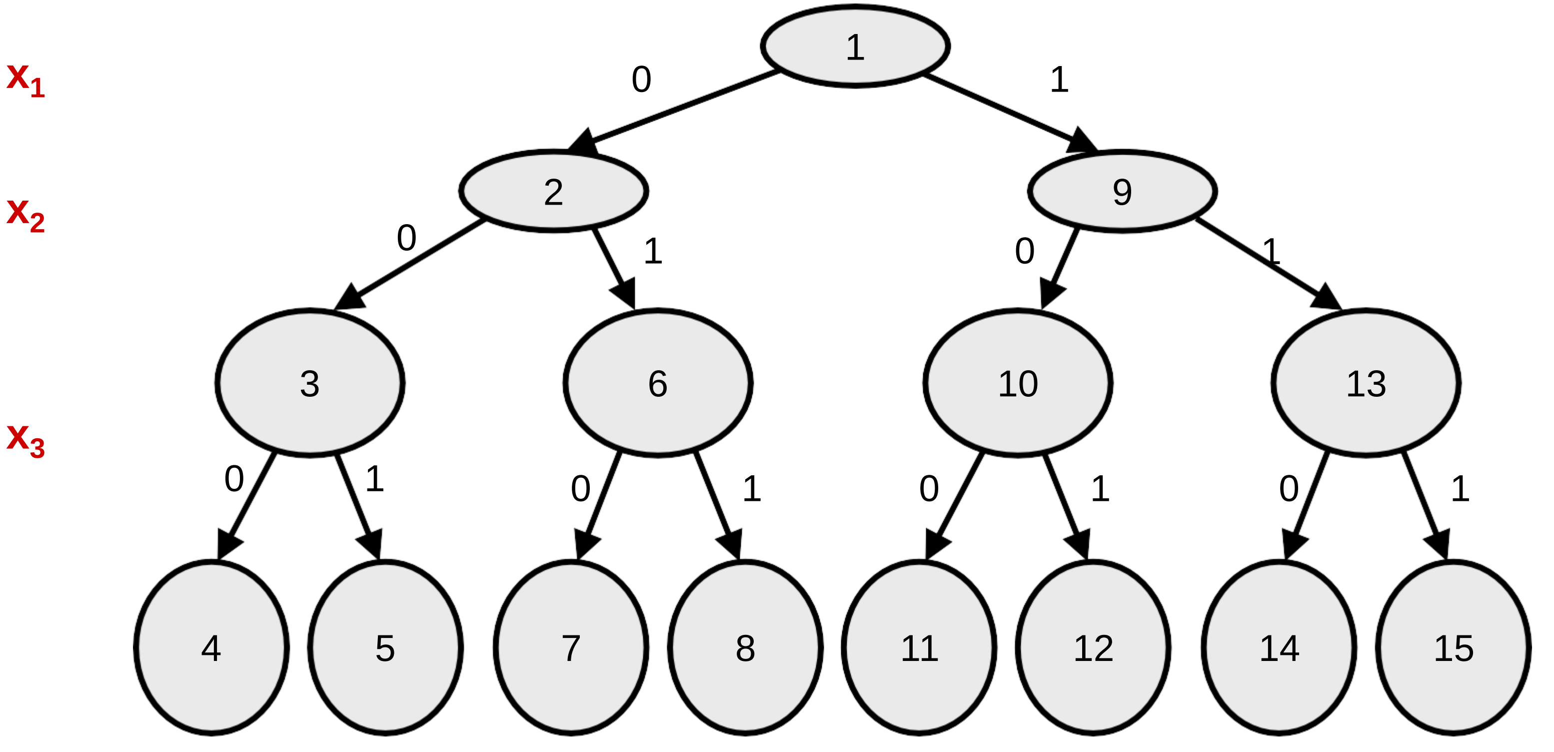
# 5. Algoritmos Bactracking. Características generales.

- El resultado es equivalente a hacer un **recorrido en profundidad en el árbol de soluciones** (árbol de expansión o árbol del espacio de estados).



# 5. Algoritmos Bactracking. Características generales.

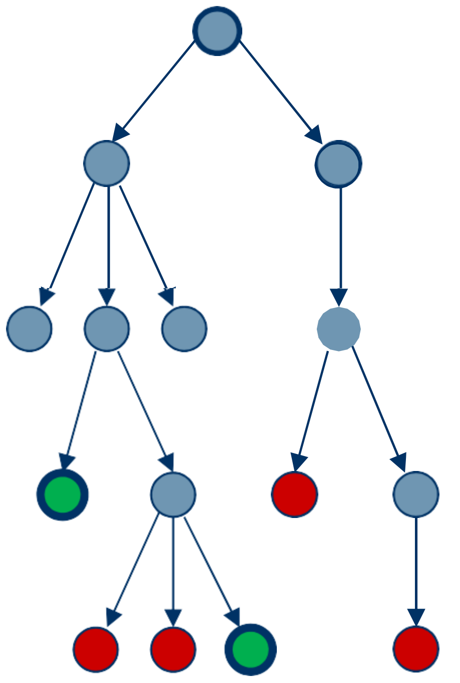
- **Representación simplificada del árbol.**



# 5. Algoritmos Bactracking. Características generales.

- **Generación de estados.**

Cuando se ha concebido el árbol de estados para un problema, podemos resolver el problema generando sistemáticamente sus **estados**, determinando cuáles de éstos son **estados solución** y, finalmente, comprobando qué estados solución son **estados solución óptimos**



- **Nodo vivo:** Estado del problema que ya ha sido generado, pero del que aún no se han generado todos sus hijos.
- **Nodo muerto:** Estado del problema que ya ha sido generado y, o bien se ha podado, o bien ya se han generado todos sus descendientes.
- **Nodo de expansión:** Nodo vivo del que actualmente se están generando sus descendientes.

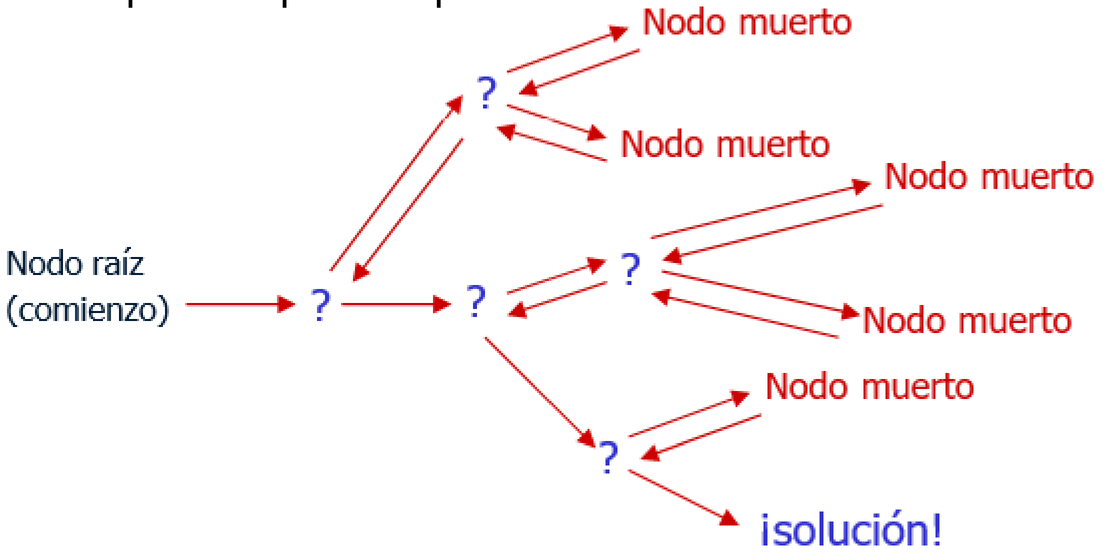
# 5. Algoritmos Bactracking. Características generales.

## Generación de estados.

- Para generar todos los estados de un problema, comenzamos con un nodo raíz a partir del cual se generan otros nodos.
- Al ir generando estados del problema, mantenemos una lista de nodos vivos.
- Se usan funciones de acotación para **podar** nodos vivos y evitar tener que generar todos sus nodos hijos.
- Existen distintas formas de generar los estados de un problema (en función de cómo exploremos el árbol de estados).
  - backtracking (en profundidad)
  - branch & bound: ramificación y poda (en anchura)

## Generación de estados usando backtracking.

- **Backtracking** corresponde a una generación primero en profundidad de los estados del problema.
- Tan pronto como un nuevo hijo H del **Nodo de expansión** en curso C ha sido generado, este hijo se convierte en un nuevo **Nodo de expansión**.
- El nodo C se convertirá de nuevo en **Nodo de expansión** cuando el subárbol H haya sido explorado por completo.



# 5. Algoritmos Bactracking. Características generales.

- **Árboles de backtracking:**
  - El árbol es simplemente una forma de representar la ejecución del algoritmo.
  - Es **implícito**, no almacenado (no necesariamente).
  - El recorrido es en **profundidad**, normalmente de izquierda a derecha.
  - La primera decisión para aplicar backtracking: ¿cómo es la forma del árbol?
  - **Preguntas relacionadas:** ¿Qué significa cada valor de la tupla solución $(x_1, ..., x_n)$? ¿Cómo es la representación de la solución al problema?

- Tipos comunes de árboles de backtracking:
  - Árboles binarios.
  - Árboles n-arios.
  - Árboles permutacionales.
  - Árboles combinatorios.

> - Cuando las tuplas solución tienen el mismo tamaño el algoritmo backtracking genera una de los 3 primeros tipos de árboles y los estados solución son los nodos hojas.
> - Cuando las tuplas solución pueden tener distinto tamaño el algoritmo backtraking genera árboles combinatorios y cada nodo del árbol (no sólo las hojas) es un estado solución

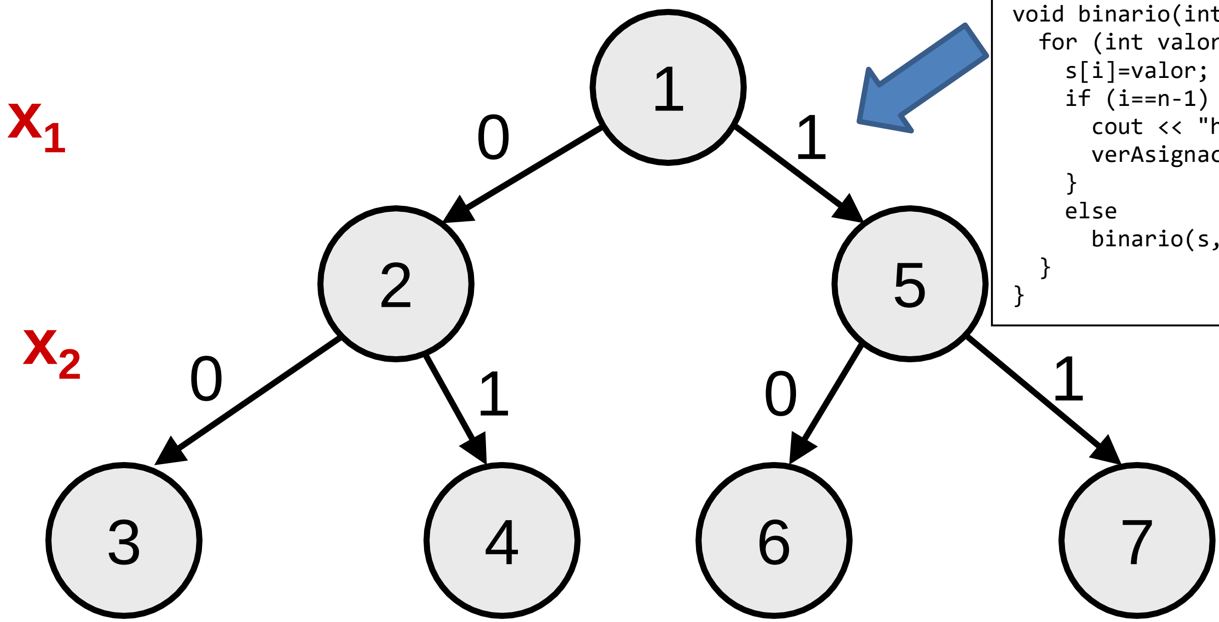
**NOTA**: En todos los algoritmos de recorrido de grafos supondremos que el grafo está implementado con listas de adyacencia.

# 5. Algoritmos Bactracking. Características generales.

- **Árboles binarios: s**= ($x_1$, $x_2$, ..., $x_n$), con $x_i \in \{0, 1\}$

```
//llamada inicial binario 2 niveles
int s[2];
binario(s,2,0);

void binario(int s[], int n, int i) {
  for (int valor=0; valor<2; valor++) {
    s[i]=valor;
    if (i==n-1) {
      cout << "hoja: ";
      verAsignacion(s, n);
    }
    else
      binario(s, n, i+1);
  }
}
```




**(*) sólo los nodos hojas del árbol son estados solución. Todas las tuplas s solución tienen el mismo tamaño (n)**

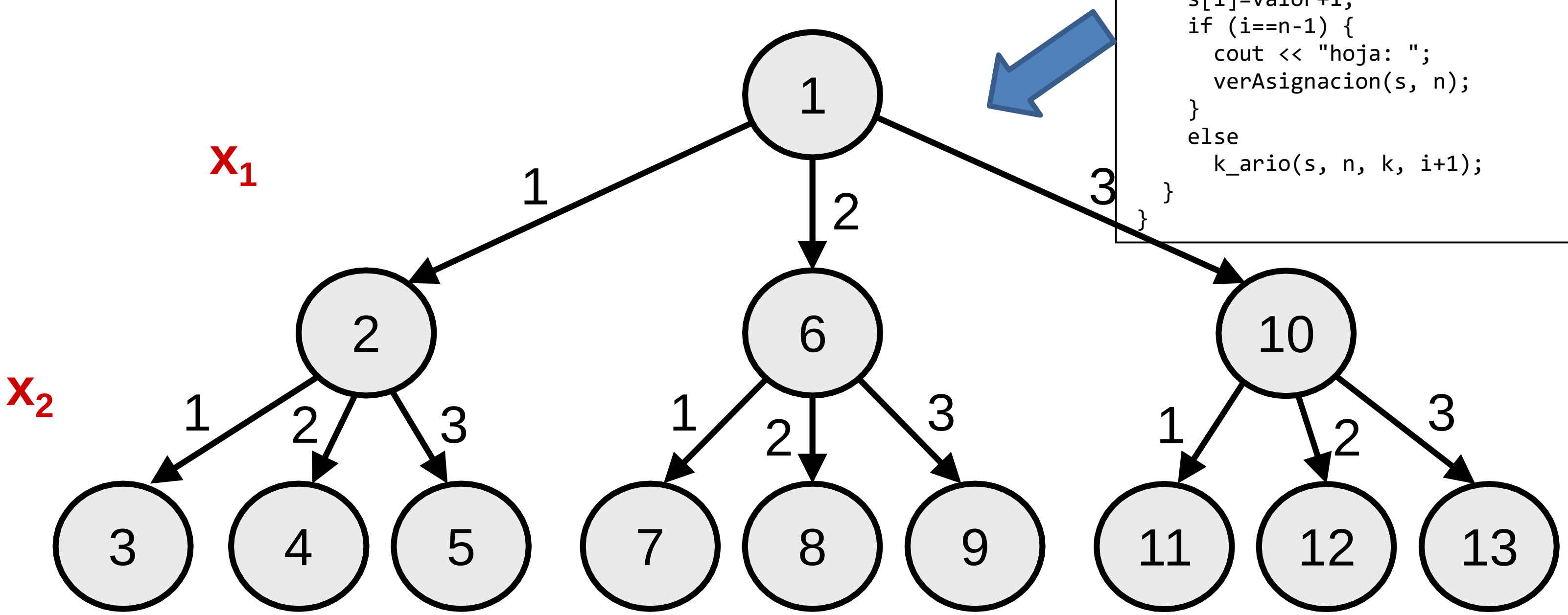
- **Tipo de problemas:** elegir ciertos elementos de entre un conjunto, sin importar el orden de los elementos.
  - Problema de la mochila 0/1.
  - Encontrar un subconjunto de {12, 23, 1, 8, 33, 7, 22} que sume exactamente 50.

# 5. Algoritmos Bactracking. Características generales.

- **Árboles k-arios: s=** ($x_1, x_2, \ldots, x_n$), con $x_i \in \{1,..,k\}$

```
int s[3];      //3_ario de 2 niveles
k_ario(s,2, 3,0); //llamada inicial

void k_ario(int s[], int n, int k, int i) {
  for (int valor=0; valor<k; valor++) {
    s[i]=valor+1;
    if (i==n-1) {
      cout << "hoja: ";
      verAsignacion(s, n);
    }
    else
      k_ario(s, n, k, i+1);
  }
}
```




**(*) sólo los nodos hojas del árbol son estados solución. Todas las tuplas s solución tienen el mismo tamaño (n)**

- **Tipo de problemas:** varias opciones para cada $x_i$.
  - Problema del cambio de monedas.
  - Problema de las **n** reinas.

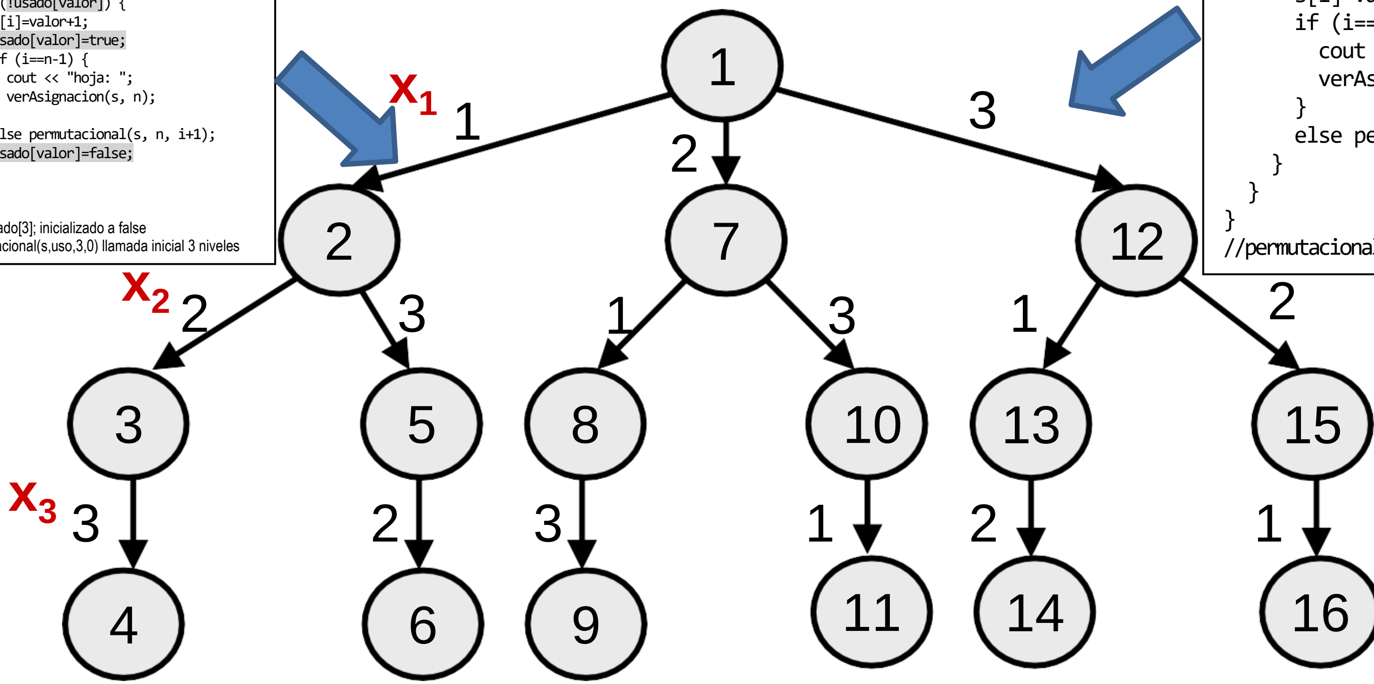
```
void verAsignacion(int asignacion[], int n) {
  for (int i = 0; i < n; i++)
    cout << asignacion[i] << " ";
  cout << endl;
}
```

# 5. Algoritmos Bactracking. Características generales.

- **Árboles permutacionales: s=** $(x_1, x_2, ..., x_n)$, con $x_i \in \{1,..,n\}$ y $x_i \neq x_j$

Permutaciones(1,2,3) = { 123, 132, 213, 231, 312, 321 }

```
void permutacional(int s[], bool usado[],
int n, int i) {
  for (int valor=0; valor<n; valor++) {
    if (!usado[valor]) {
      s[i]=valor+1;
      usado[valor]=true;
      if (i==n-1) {
        cout << "hoja: ";
        verAsignacion(s, n);
      }
      else permutacional(s, n, i+1);
      usado[valor]=false;
    }
  }
}
//bool usado[3]; inicializado a false
//permutacional(s,uso,3,0) llamada inicial 3 niveles
```




```
void permutacional(int s[], int n, int i) {
  for (int valor=0; valor<n; valor++) {
    if (!Repetido(s,i,valor)) {
      s[i]=valor+1;
      if (i==n-1) {
        cout << "hoja: ";
        verAsignacion(s, n);
      }
      else permutacional(s, n, i+1);
    }
  }
}
//permutacional(s,3,0) llamada inicial 3 niveles
```

**(*) sólo los nodos hojas del árbol son estados solución. Todas las tuplas s solución tienen el mismo tamaño (n)**

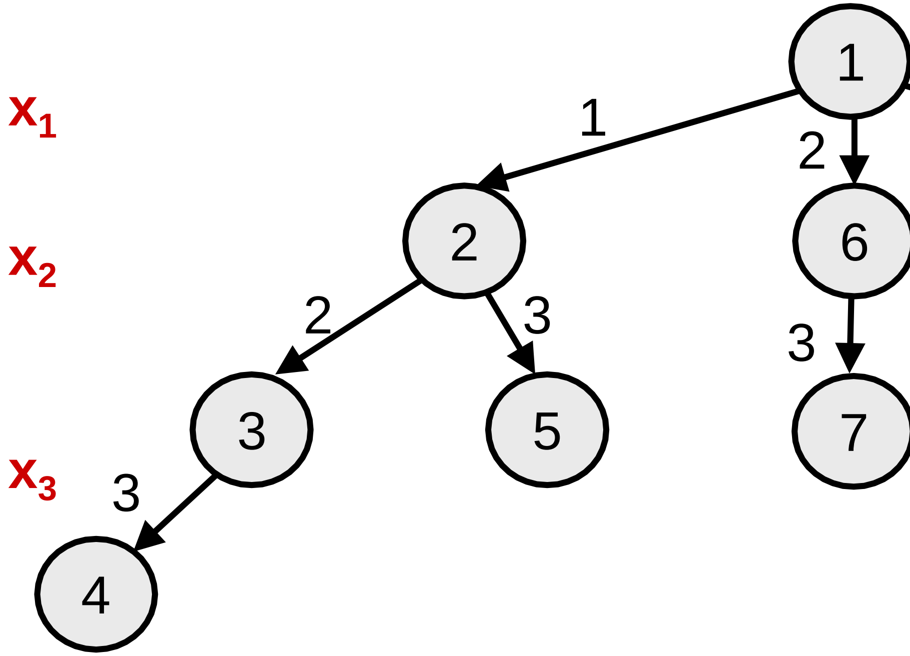
- **Tipo de problemas:** los $x_i$ no se pueden repetir.
  - Generar todas las permutaciones de (1, ..., n).
  - Asignar **n** trabajos a **n** personas, asignación uno-a-uno.

```
bool Repetido(int s[], int pos, int valor) {
  for(int i=0; i<pos; i++) {
    if (s[i]==valor)
      return true;
  }
  return false;
}
```

# 5. Algoritmos Bactracking. Características generales.

- **Árboles combinatorios: s**= ($x_1$, $x_2$, ..., $x_m$), con m≤n, $x_i \in \{1,..,n\}$ y $x_i < x_{i+1}$

Combinaciones(1,2,3) = { 1, 12, 123, 13, 2, 23, 3 }




```
//combinatorio(s,3,0) llamada inicial 3 niveles

void combinatorio(int s[], int n, int i) {
  for (int valor=i; valor<n; valor++) {
   if (i==0 || valor>s[i-1]) {
    s[i]=valo+1r;
    verAsignacion(s, i+1);
    combinatorio(s, n, i+1);
   }
  }
}
```

**(*) cada nodo del árbol (no sólo las hojas) son estados solución. Las tuplas s solución tienen distinto tamaño (1 ≤ m ≤** n)

- **Tipo de problemas:** los mismos que con árboles binarios.
  - Binario: (0, 1, 0, 1, 0, 0, 1, 0) → Combinatorio: (2, 4, 7)

```
void verAsignacion(int asignacion[], int n) {
  for (int i = 0; i < n; i++)
    cout << asignacion[i] << " ";
  cout << endl;
}
```

# 5. Algoritmos Bactracking. Características generales.

```
int s[3];         //4_ario de 3 niveles
k_ario(s,3,4,0); //llamada inicial

void k_ario(int s[], int n, int k, int i) {
  for (int valor=0; valor<k; valor++) {
    s[i]=valor;
    if (i==n-1) {
      cout << "hoja: ";
      verAsignacion(s, n);
    }
    else
      k_ario(s, n, k, i+1);
  }
}
```

```
int s[3];         //4_ario de 3 niveles
k_ario(s,3,4,0); //llamada inicial

void k_ario(int s[], int n, int k, int i) {     //i=0
  for (int valor=0; valor<k; valor++) {
    s[i]=valor;
    for (int valor=0; valor<k; valor++) {       //i+1=1
      s[i+1]=valor;
      for (int valor=0; valor<k; valor++) {     //i+2=2
        s[i+2]=valor;
        cout << "hoja: ";
        verAsignacion(s, n);
      }
    }
  }
}
```

Crea un árbol 4-ario de 3 niveles

000, 001, 002, 003, 010, 011, 012, 013, 020, 021, 022, 023, 030, 031, 032, 033
100, 101, 102, 103, 110, 111, 112, 113, 120, 121, 122, 123, 130, 131, 132, 133
200, 201, 202, 203, 210, 211, 212, 213, 220, 221, 222, 223, 230, 231, 232, 233
300, 301, 302, 303, 310, 311, 312, 313, 320, 321, 322, 323, 330, 331, 332, 333

```
int s[4];  //permutacional de 4 niveles
permutacional(s,4,0); //llamada inicial

void permutacional(int s[], int n, int i) {
  for (int valor=0; valor<n; valor++) {
    if (!Repetido(s,i,valor)) {
      s[i]=valor+1;
      if (i==n-1) {
        cout << "hoja: ";
        verAsignacion(s, n);
      }
      else permutacional(s, n, i+1);
    }
  }
}
```

```
int s[3];  //permutacional de 3 niveles
permutacional(s,3,0); //llamada inicial

void permutacional(int s[], int n, int i) {     //i=0
  for (int valor=0; valor<k; valor++)
   if (!Repetido(s,i,valor)) {
    s[i]=valor+1;
    for (int valor=0; valor<k; valor++)         //i+1=1
     if (!Repetido(s,i,valor)) {
      s[i+1]=valor+1;
      for (int valor=0; valor<k; valor++)       //i+2=2
       if (!Repetido(s,i,valor)) {
        s[i+2]=valor+1;
        for (int valor=0; valor<k; valor++)     //i+3=3
         if (!Repetido(s,i,valor)) {
          s[i+3]=valor+1;
          cout << "hoja: ";
          verAsignacion(s, n);
         }
        }
       }
     }
}
```

Crea un árbol permutacional de 4 niveles

1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431
3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321

## 5. Algoritmos Bactracking. Características generales. Ejemplo de backtracking.

- **Ejemplo:** Diseñar un algoritmo que permita obtener un subconjunto de números dentro del conjunto *datos*= {17, 11, 3} cuya suma sea 20

1. **Representación de la solución**: tuplas mismo tamaño (árbol binario) vs tuplas distinto tamaño (árbol combinatorio)

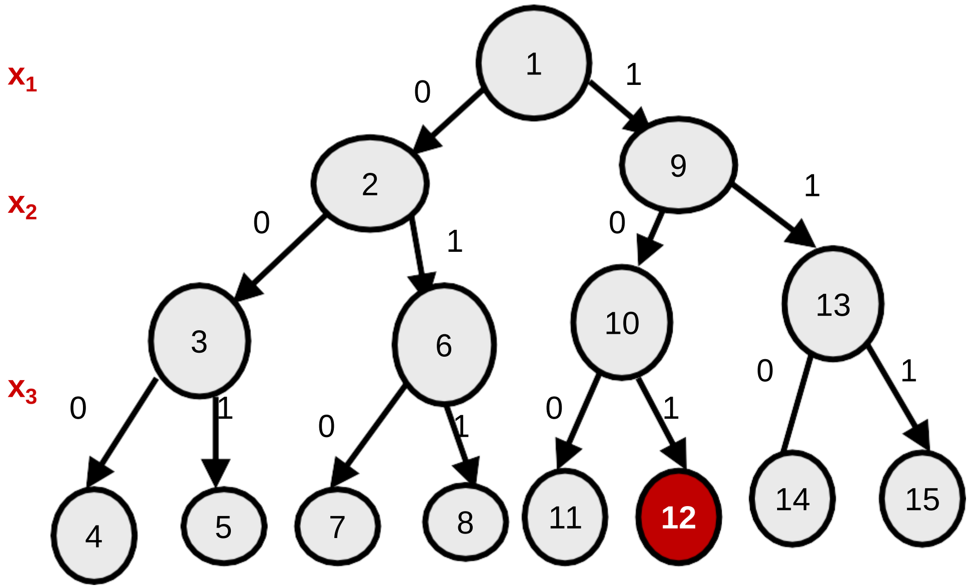
**1.1.** Tuplas del mismo tamaño → vector de 3 elementos [$x_1$, $x_2$, $x_3$] con $x_i \in \{0, 1\}$

- Restricciones explicitas: indican que valores pueden tomar los componentes de la solución
  - $\mathbf{x_i = 0}$ indica que el elemento i **no** está en la solución
  - $\mathbf{x_i = 1}$ indica que el elemento i **si** está en la solución
  - La solución parcial debe cumplir que $\sum_{i=1}^{3} x_i \, datos_i \leq 20$
- Restricciones implícitas: indican que tuplas pueden dar lugar a soluciones válidas
  - La solución debe cumplir el siguiente objetivo: $\sum_{i=1}^{3} x_i datos_i = 20$
- Para este problema existe una única solución: x=[1, 0, 1]

**2. Árbol de expansión.**

Para la representación elegida (tuplas del mismo tamaño) existen dos formas posibles:

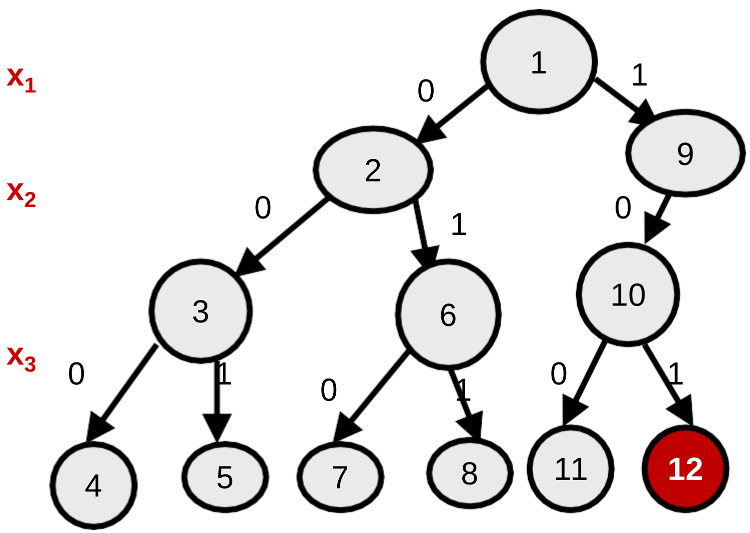
**2.1. Árbol binario** ⇒ Generar todas las combinaciones posibles y escoger aquellas que sean solución




- Cada camino de la raíz a las hojas define una posible solución
- El árbol de expansión tiene $2^3$ hojas (8 posibles soluciones)
- Se generan tuplas que no son soluciones ⇒ ineficiente

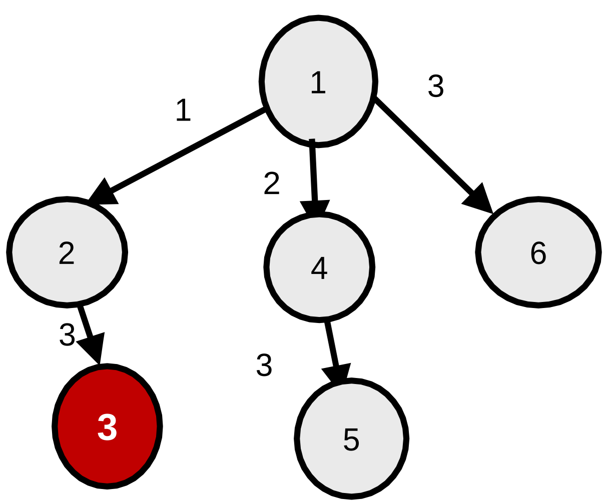
**2. Árbol de expansión.** (cont.)

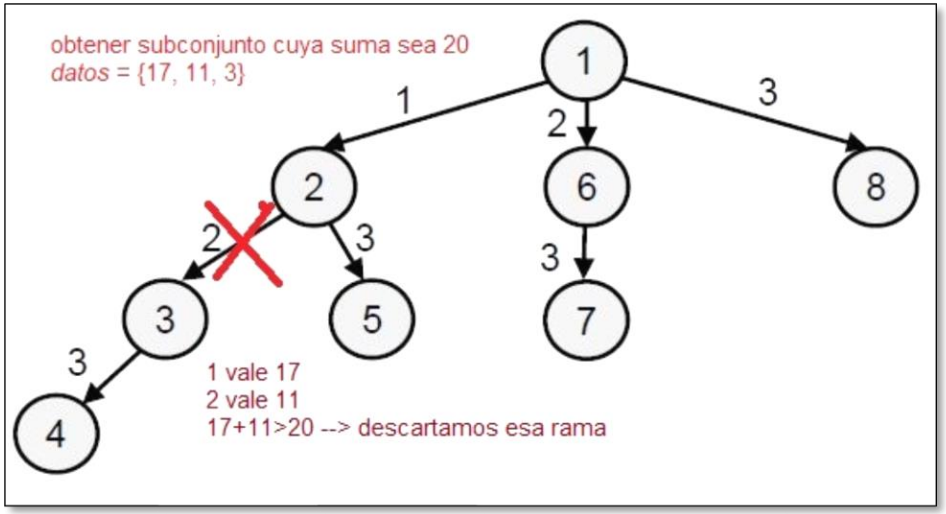
**2.2. Árbol binario utilizando la técnica de backtracking** ⇒ a medida que se construye la tupla, se comprueba si esta puede llegar a ser una solución al problema. En caso negativo, se ignora y se vuelve al estado anterior.



# 5. Algoritmos Bactracking. Características generales. Ejemplo.

- **1.2.** Tuplas de distinto tamaño (representación de la solución con árbol combinatorio):
  - Tuplas de distinto tamaño → vector de a lo sumo 3 elementos ordenados con valores entre 1 y 3 (los índices de los elementos del conjunto anterior)
    - 1 indica que el elemento 1 (con valor 17) está en la solución
    - 2 indica que el elemento 2 (con valor 11) está en la solución
    - 3 indica que el elemento 3 (con valor 3) está en la solución
  - Para este problema existe una única solución: [1, 3]

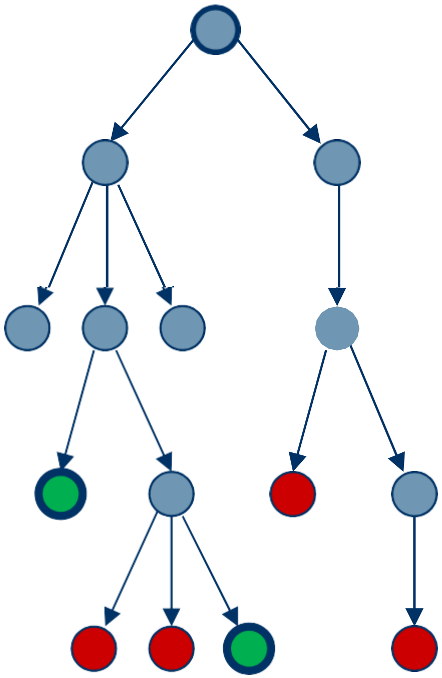


# 5. Algoritmos Bactracking. Esquema general.

**Cuestiones a resolver antes de programar:**

- Qué tipo de árbol es adecuado para el problema.
  - → Cómo es la representación de la solución
  - → Cómo es la tupla solución (mismo o distinto tamaño). Qué indica cada $\mathbf{x}_i$ y qué valores puede tomar.
- Cómo generar un recorrido según ese árbol
  - → Generar un nuevo nivel.
  - → Generar los hermanos de un nivel.
  - → Retroceder en el árbol.



- Qué ramas se pueden descartar por no conducir a soluciones del problema.
  - → Poda por restricciones del problema.
  - → Poda según el criterio de la función objetivo.

# 5. Algoritmos Bactracking. Esquema general.

- La **técnica de backtracking** es un **recorrido en profundidad (preorden) del árbol de expansión**

1. En cada momento el algoritmo se encontrará en un cierto **nivel K**
2. En el nivel K se tiene una **solución parcial ($x_1$, ..., $x_k$),**
3. Se comprueba si se puede **añadir** un nuevo elemento **$x_{k+1}$** a la solución
   - En caso afirmativo, **($x_1$, ..., $x_{k+1}$)** es prometedor $\Rightarrow$ se genera la solución parcial **($x_1$, ..., $x_{k+1}$)** y se avanza al nivel **K+1**
   - En otro caso $\Rightarrow$ se prueban otros valores de **$x_k$**
4. Si ya no existen más valores para **$x_k$** , se retrocede (se **vuelve atrás**-backtrack) al nivel anterior **K-1**
5. El algoritmo continúa hasta que la solución parcial sea una solución completa del algoritmo, o hasta que no queden más posibilidades

## Esquema general de backtracking recursivo

```
función backtrackingRec (x[1..n], solucion[1..n], k)
  IniciarValores(x,k) <-genera todas los valores posibles para el elemento x[k]
  para cada valor x[k] <-considera todos los valores posibles
    si Alcanzable(x[k]) entonces <-comprueba si el valor x[k] puede formar parte de solucion
      aux=solucion[k] <-salvo el valor que tenía en aux (por si deshacemos la decisión tomada)
      solucion[k]=x[k] <-añade el valor x[k] al vector solución
      si SolucionIncompleta(solucion) entonces
        backtrackingRec(x, solucion, k+1)
      sino si EsSolucion(solucion) entonces <-indica si solucion es una solución del problema
        escribir(solucion)
      fsi
      solucion[k]=aux <-elimina el último valor añadido al vector solucion (y lo dejamos como estaba antes)
    fsi
  fpara
ffunción
```

# 5. Algoritmos Bactracking. Esquema general NO recursivo

- **Esquema general (no recursivo).** Problema de satisfacción de restricciones.

```
procedimiento backtracking(var solucion[1..n])
  nivel:=1<-indica el nivel actual en el que se encuentra el algoritmo
  solucion:= solucionINICIAL <-valor de inicialización de la solución parcial que tenemos en cada momento/etapa
  fin:=false<-Valdrá true cuando hayamos encontrado alguna solución
  repetir
    Generar(nivel,solucion)/*solucion[nivel]←generar(nivel,solucion)*/ <-genera
    el siguiente hermano, o el primero, para el nivel actual. Devuelve el valor a añadir a la solución parcial actual
    si EsSolucion(nivel,solucion) entonces<-comprueba si la solución (s[1],…,s[nivel]) calculada
      fin:=true                                          hasta el momento es una solución válida
    sino
      si Criterio(nivel, solucion) entonces<-comprueba si a partir de la solución parcial actual
        nivel:=nivel + 1                                 (s[1], ..., s[nivel]) se puede alcanzar una válida.
      sino                                               Si no, se rechazan todos sus descendientes (poda)
        mientras not(HayMasHermanos(nivel,solucion)) hacer<-HayMasHermanos() devuelve
        true si el nodo actual tiene hermanos que aún no han sido generados
          Retroceder(nivel,solucion) <-retrocede un nivel en el árbol de soluciones. nivel-- y posiblemente
        fmientras                     tendrá que actualizar la solución actual, quitando los elementos retrocedidos
      fsi
    fsi
  hasta fin=true
fprocedimiento
```

## 5. Algoritmos Bactracking. Esquema general. **Ejemplo (1/4)**

- **Ejemplo:** Encontrar un subconjunto del conjunto T= $\{t_1, t_2, ..., t_n\}$ que sume exactamente P.

- **Variables**:
  - Representación de la solución con un **árbol binario** (todas las tuplas solución tienen el mismo tamaño n y son las hojas del árbol)
  - **s**: array [1..n] de {-1, 0, 1}
    - s[i] = 0 → el número i-ésimo no se utiliza
    - s[i] = 1 → el número i-ésimo sí se utiliza
    - s[i] = -1 → valor de inicialización (número i-ésimo no estudiado)
  - $\mathbf{s_{INICIAL}}$: (-1, -1, ..., -1)
  - **fin**: Valdrá **true** cuando se haya encontrado solución.
  - **tact:** Suma acumulada hasta ahora (inicialmente 0).

- La solución parcial debe cumplir que $\sum_{i=1}^{n} x_i \, datos_i \leq 20$ y la solución que $\sum_{i=1}^{n} x_i \, datos_i = 20$

- **Algoritmo recursivo1** (muestra solo la 1ª solución, eliminando lo amarillo muestra todas)
  Llamada inicial: **sumaP(t, x, sol, 0, P, 0)** /* pesoIni = 0, k=0 */

```
proc sumaP(t[1..n], x[1..n], sol[1..n], pesoIni, P, k)
  encontrado:=false
  para valor:= 0 hasta 1 hacer // árbol binario: posibles valores 0 y 1
    si encontrado=false entonces
      x[k]:=valor;   pesoAc:= pesoIni + valor * t[k]
      si (pesoAc ≤ P) entonces
        si k = n entonces // si es una hoja (k = n → es hoja) vemos si es o no solucion
          si pesoAc = P entonces /* solución */
            escribir(x);  sol:=x /* Se asigna el vector completo */
            encontrado:=true
          fsi
        sino // si no es una hoja (k<>n → no es hoja) hace llamada recursiva
          encontrado=sumaP(t, x, sol, pesoAc, P, k+1) /* recursión */
        fsi
      fsi
    fsi
  fpara
  return encontrado
fproc
```

# 5. Algoritmos Bactracking. Esquema general. **Ejemplo (3/4)**

- **Algoritmo recursivo2** (muestra solo la 1ª solución, eliminando lo amarillo muestra todas)
  Llamada inicial: **sumaP(t, x, sol, 0, ∑t[i], P, 0)** /* pesoIni = 0, pesoRestoIni=∑t[i], k=0 */
- si alcanza solución, poda aunque no haya llegado a una hoja del árbol binario
- poda anticipadamente si los pesos que quedan no alcanzan la solución

```
proc sumaP(t[1..n], x[1..n], sol[1..n], pesoIni, pesoRestoIni, P, k)
  encontrado:=false
  pesoRestoAc:= pesoRestoIni – t[k]
  para x[k]:= 0 hasta 1 hacer // árbol binario: posibles valores 0 y 1
    si encontrado=false entonces
      pesoAc:= pesoIni + x[k] * t[k]
      si (pesoAc = P) entonces //solución → no hace más llamadas recursivas (poda) aunque no sea hoja (hoja → k=n)
        para i:= k+1 hasta n hacer x[i]:=0 //rellena los que faltan con ceros
        escribir(x);  sol:=x /* Se asigna el vector completo */
        encontrado:=true
      sino si (pesoAc ≤ P Y pesoAc+pesoRestoAc>= P) entonces
        si k < n entonces // si no es una hoja (k<n → no es hoja) hace llamada recursiva
           encontrado=sumaP(t, x, sol, pesoAc, pesoRestoAc, P, k+1) /* recursión */
        fsi
      fsi
    fsi
  fpara
  return encontrado
fproc
```

Este algoritmo mejora al anterior ya que realiza muchas más podas

# 5. Algoritmos Bactracking. Esquema general. **Ejemplo (4/4)**

- **Funciones**:
  - **Generar (nivel, s)**

    s[nivel]:= s[nivel] + 1

    **si** s[nivel]==1 **entonces** tact:= tact + $t_{nivel}$
  - **EsSolución (nivel, s)**

    **devolver** (nivel==n) Y (tact==P)
  - **Criterio (nivel, s)**

    **devolver** (nivel<n) Y (tact≤P)
  - **HayMasHermanos (nivel, s)**

    **devolver** s[nivel] < 1
  - **Retroceder (nivel, s)**

    tact:= tact – $t_{nivel}$*s[nivel]

    s[nivel]:= -1

    nivel:= nivel – 1

- **Algoritmo Iterativo:** (el mismo que el esquema general)

**Ejemplo:** Encontrar un subconjunto del conjunto T= {$t_1$, $t_2$, ..., $t_n$} que sume exactamente P

**Variables**:

Representación de la solución con un **árbol binario**

**s**: array [1..n] de {-1, 0, 1}

s[i] = 0 → el número i-ésimo no se utiliza

s[i] = 1 → el número i-ésimo sí se utiliza

s[i] = -1 → número i-ésimo no estudiado

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= sINICIAL          sINICIAL: (-1, -1, ..., -1)
    fin:= false
    tact:= 0
    repetir
        Generar (nivel, s)
            [ s[nivel]:= s[nivel] + 1
              si s[nivel]==1 entonces tact:= tact + t_nivel ]
        si EsSolución (nivel, s)  (nivel==n) Y (tact==P)  entonces
            fin:= true
        sino si Criterio (nivel, s)  (nivel<n) Y (tact≤P)  entonces
            nivel:= nivel + 1
        sino
            mientras NOT HayMasHermanos (nivel, s)  s[nivel] < 1  hacer
                Retroceder (nivel, s)
                    [ tact:= tact - t_nivel * s[nivel]
                      s[nivel]:= -1
                      nivel:= nivel - 1 ]
        finsi
    hasta fin
```

- **Variaciones** del esquema general:

1. Si no es seguro que exista una solución
2. Si se quiere almacenar todas las soluciones (no sólo una)
3. Si el problema es de optimización (maximizar o minimizar)

- **Caso 1)** Puede que no exista ninguna solución.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s) entonces
            fin:= true
        sino si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT HayMasHermanos (nivel, s) AND (nivel>0) hacer
                Retroceder (nivel, s)
        finsi
    hasta fin OR (nivel==0)
```

Para poder generar todo el árbol de backtracking

- **Caso 2)** Se quiere almacenar todas las soluciones.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s) entonces
            Almacenar (nivel, s)
        si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT HayMasHermanos (nivel, s) AND (nivel>0) hacer
                Retroceder (nivel, s)
        finsi
    hasta nivel==0
```

- En algunos problemas los nodos intermedios pueden ser soluciones
- O bien, retroceder después de encontrar una solución

- **Caso 3)** Problema de **optimización** (maximización).

**Backtracking (var s: TuplaSolución)**

nivel:= 1

s:= $s_{INICIAL}$

**voa:= -∞; soa:= Ø**

**voa**: valor óptimo actual

**soa:** solución óptima actual

**repetir**

Generar (nivel, s)

**si** EsSolución (nivel, s) **AND Valor(s) > voa entonces**

**voa:= Valor(s); soa:= s**

**si** Criterio (nivel, s) **entonces**

nivel:= nivel + 1

**sino**

**mientras** NOT HayMasHermanos (nivel, s) **AND (nivel>0) hacer**

Retroceder (nivel, s)

**finsi**

**hasta nivel==0**

# 5. Algoritmos Bactracking. Variaciones del Esquema general: **Ejemplo.**

- **Ejemplo:** Encontrar un subconjunto del conjunto T= $\{t_1, t_2, ..., t_n\}$ que sume exactamente P, usando el **menor número** posible de elementos.

- **Funciones**:
  - **Valor(s)**

    **devolver** s[1] + s[2] + ... + s[n]
  - Todo lo demás no cambia !!

- **Otra posibilidad:** incluir una nueva variable:

  **vact: entero**. Número de elementos en la tupla actual.
  - **Inicialización** (añadir)**:** vact:= 0
  - **Generar** (añadir)**:** vact:= vact + s[nivel]
  - **Retroceder** (añadir)**:** vact:= vact - s[nivel]

- **Ejemplo:** Encontrar un subconjunto del conjunto T= {$t_1$, $t_2$, ..., $t_n$} que sume exactamente P, usando el **menor número** posible de elementos.
- **Caso 3)** Problema de **optimización** (minimización). Usando función **Valor(s)**

**Backtracking (var s: TuplaSolución)** **s**: array [1..n] de {-1, 0, 1}

nivel:= 1; s:= $s_{INICIAL}$ **$s_{INICIAL}$: {-1, -1, ..., -1}**

tact:=0;

**voa:= ∞; soa:= Ø**

**voa**: valor óptimo actual

**soa:** solución óptima actual

**repetir**

Generar (nivel, s)

s[nivel]:= s[nivel] + 1
**si** s[nivel]==1 **entonces** tact:= tact + $t_{nivel}$

**si** ~~EsSolución (nivel, s)~~ (nivel==n) Y (tact==P) **AND Valor(s) < voa entonces**

**voa:= Valor(s);** **soa:= s**

**función Valor(s)**
**devolver** s[1] + s[2] + ... + s[n]
**ffunción**

**si** ~~Criterio (nivel, s)~~ (nivel<n) Y (tact≤P) **entonces**

nivel:= nivel + 1

**sino**

**mientras** NOT ~~HayMasHermanos (nivel, s)~~ s[nivel]<1 **AND (nivel>0) hacer**

Retroceder (nivel, s)

tact:= tact - $t_{nivel}$ * s[nivel]
s[nivel]:= -1
nivel:= nivel - 1

**finsi**

**hasta nivel==0**

- **Ejemplo:** Encontrar un subconjunto del conjunto T= {$t_1$, $t_2$, ..., $t_n$} que sume exactamente P, usando el **menor número** posible de elementos.
- **Caso 3)** Problema de **optimización** (minimización). Usando nueva variable **vact**

**Backtracking (var s: TuplaSolución)** **s**: array [1..n] de {-1, 0, 1}

nivel:= 1; s:= $s_{INICIAL}$ **$s_{INICIAL}$: {-1, -1, ..., -1}**

tact:=0; **vact:= 0;**

**voa:= ∞; soa:= Ø**

**voa**: valor óptimo actual

**soa:** solución óptima actual

**repetir**

Generar (nivel, s)

s[nivel]:= s[nivel] + 1; vact:= vact + s[nivel]
**si** s[nivel]==1 **entonces** tact:= tact + $t_{nivel}$

**si** ~~EsSolución (nivel, s)~~ (nivel==n) Y (tact==P) **AND vact < voa entonces**

**voa:= vact; soa:= s**

**si** ~~Criterio (nivel, s)~~ (nivel<n) Y (tact≤P) **entonces**

nivel:= nivel + 1

**sino**

**mientras** NOT ~~HayMasHermanos (nivel, s)~~ s[nivel]<1 **AND (nivel>0) hacer**

Retroceder (nivel, s)

tact:= tact - $t_{nivel}$ * s[nivel]
vact:= vact - s[nivel]
s[nivel]:= -1
nivel:= nivel - 1

**finsi**

**hasta nivel==0**

## 5. Algoritmos Bactracking. Análisis de tiempos de ejecución.

### Observaciones

- La representación de las soluciones determina la forma del árbol de expansión
  - Cantidad de descendientes de un nodo
  - Profundidad del árbol
  - Cantidad de nodos del árbol
- La representación de las soluciones determina, como consecuencia, la eficiencia del algoritmo ya que el tiempo de ejecución depende del número de nodos generados
- El árbol tendrá (como mucho) tantos niveles como valores tenga la secuencia solución
- En cada nodo se debe poder determinar:
  - Si es solución o posible solución del problema
  - Si tiene hermanos sin generar
  - Si a partir de este nodo se puede llegar a una solución

## 5. Algoritmos Bactracking. Análisis de tiempos de ejecución.

- En general se obtienen ordenes de complejidad exponencial y factorial
- El orden de complejidad depende del número de nodos generados y del tiempo requerido para cada nodo (que podemos considerar constante)
- Si la solución es de la forma $(\mathbf{x_1, x_2, \ldots, x_n})$, donde $\mathbf{x_i}$ admite $\mathbf{m_i}$ valores
- En el caso peor, se generarán todas las posibles combinaciones para cada $\mathbf{x_i}$

| Nivel 1 | $\mathbf{m_1}$ **nodos** |
|---|---|
| Nivel 2 | $\mathbf{m_1} * \mathbf{m_2}$ nodos |
| ….. | …. |
| Nivel n | $\mathbf{m_1} * \mathbf{m_2} *\ldots * \mathbf{m_n}$ nodos |

- Para el ejemplo planteado al principio, $\mathbf{m_i} = 2$

$$\boldsymbol{T(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2}$$

**Tiempo exponencial**

- Cada caso depende de cómo se realice la poda del árbol, y de la instancia del problema

# 5. Algoritmos Bactracking. Análisis de tiempos de ejecución.

- Para el problema de calcular todas las permutaciones de (1, 2,…, **n**)
- Se representa la solución como una tupla $\{x_1, x_2, ..., x_n\}$,
- Restricciones explícitas: $x_i \in \{i, ..., n\}$,
- En el nivel 1, hay *n* posibilidades, en el nivel 2 *n* - 1

| **Nivel 1** | ***n* nodos** |
|---|---|
| Nivel 2 | ***n**** (***n*-1**) nodos |
| ….. | …. |
| Nivel n | ***n** ** (***n*-1**) *…..** * **1** nodos |

- **Tiempo factorial**

$$T(n) = n + n*(n-1) + ... + n! \in O(n!)$$

## Resumen

- Normalmente, el tiempo de ejecución se puede obtener multiplicando dos factores:
  - Número de nodos del árbol.
  - Tiempo de ejecución de cada nodo.

  siempre que el tiempo en cada nodo sea del mismo orden.
- Las podas eliminan nodos a estudiar, pero su efecto suele ser más impredecible.
- En general, los algoritmos de backtracking dan lugar a tiempos de órdenes factoriales o exponenciales $\Rightarrow$ No usar si existen otras alternativas más rápidas.

- "**Mochila 0-1**". Como el problema de la mochila, pero los objetos **no** se pueden partir (se cogen enteros o nada)
- **Datos del problema:**
  - **n**: número de objetos disponibles.
  - **M**: capacidad de la mochila.
  - $\mathbf{p = (p_1, p_2, ..., p_n)}$ pesos de los objetos.
  - $\mathbf{b = (b_1, b_2, ..., b_n)}$ beneficios de los objetos.
- **Formulación matemática:**

**Maximizar** $\sum_{i=1}^{n} x_i b_i$ sujeto a la **restricción** $\sum_{i=1}^{n} x_i p_i \le M$ con $x_i \in \{1,0\}$

- Veremos la solución con los esquemas:

1. Backtracking recursivo
2. Backtracking NO recursivo

## 1. Implementación con backtracking recursivo.

- La solución se puede representar, con un árbol binario, como una tupla

  **s= {$x_1$, $x_2$, ..., $x_n$}**
- Restricciones explicitas: $x_i \in \{0,1\}$
  - $x_i = 0 \rightarrow$ el objeto ***i*** no se introduce en la mochila
  - $x_i = 1 \rightarrow$ el objeto ***i*** no se introduce en la mochila
- Restricciones implícitas: $\sum_{i=1}^{n} x_i\, p_i \leq M$
- El objetivo es maximizar la función $\sum_{i=1}^{n} x_i b_i$
- Se almacenará una **solución parcial** que se irá actualizando al encontrar una nueva solución con mayor beneficio
- Solo los nodos terminales del árbol de expansión pueden ser solución al problema
- La función `alcanzable(Criterio)` comprueba que los pesos acumulados hasta el momento no excedan la capacidad de la mochila. Esta función permite la **poda** de nodos

## 5. **Backtracking recursivo**. Ejemplo. 1_Problema de la mochila 0/1.

(**benMax** por referencia, no por valor …o poner **benMax** como variable global y que no sea un parámetro más)

```
/*elem[1..n] vector de estructuras con dos campos: beneficio y peso*/
proc mochila(elem[1..n], x[1..n], sol[1..n], benIni, benMax, pesoIni, k)
  para valor:= 0 hasta 1 hacer // árbol binario: posibles valores 0 y 1
    x[k]:=valor
    benAc:= benIni + valor * elem[k].beneficio
    pesoAc:= pesoIni + valor * elem[k].peso
    si (pesoAc ≤ M) entonces
      si k = n entonces
        si benAc > benMax entonces
          sol:= x /* Se asigna el vector completo */
          benMax:= benAc
        fsi
      sino
        mochila(elem, x, sol, benAc, benMax, pesoAc, k+1) // recursión
      fsi
    fsi
  fpara
fproc
```

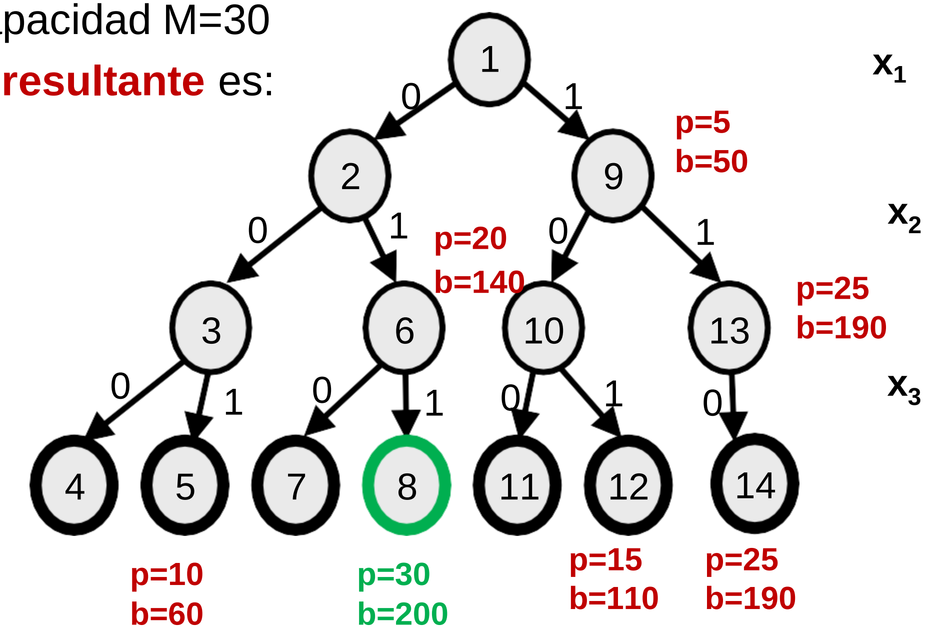
## 5. **Backtracking recursivo**. Ejemplo. 1_Problema de la mochila 0/1.

- El procedimiento que invoca al algoritmo es:

```
proc invoca_mochila(elem[1..n],sol[1..n])
  crear solAct[1..n]
  mochila(elem,solAct,sol,0,-∞,0,1)
fproc
```

- **Ejemplo**: Con una mochila de capacidad M=30 y los siguientes objetos, el **árbol resultante** es:

| objeto | A | B | C |
|---|---|---|---|
| peso | 5 | 20 | 10 |
| valor | 50 | 140 | 60 |



## 2. Implementación con backtracking NO recursivo:

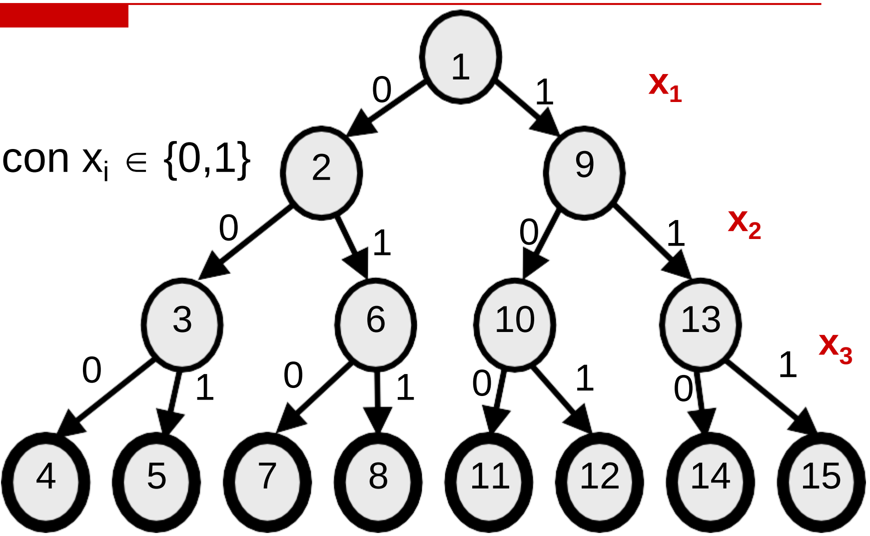
- Aplicación de backtracking NO recursivo (proceso metódico):

1. Determinar cómo es la forma del árbol de backtracking ⇔ cómo es la representación de la solución.
2. Elegir el esquema de algoritmo adecuado, adaptándolo en caso necesario.
3. Diseñar las funciones genéricas para la aplicación concreta: según la forma del árbol y las características del problema.
4. Posibles mejoras: usar variables locales con “valores acumulados”, hacer más podas del árbol, etc.

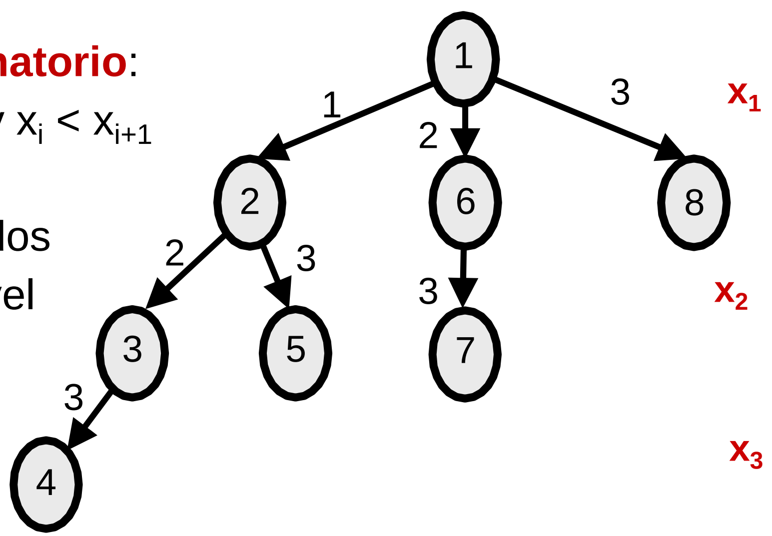
# 5. Algoritmos Bactracking. Ejemplos de aplicación. 1_Problema de la mochila 0/1

## 1. Representación de la solución.

- Con un **árbol binario**: **s= ($x_1$, $x_2$, ..., $x_n$)**, con $x_i \in \{0,1\}$
  - $x_i = 0 \rightarrow$ No se coge el objeto **i**
  - $x_i = 1 \rightarrow$ Sí se coge el objeto **i**
  - $x_i = -1 \rightarrow$ Objeto **i** no estudiado
  - En el nivel **i** se estudia el objeto **i**
  - Las soluciones están en nivel **n**




- También es posible usar un **árbol combinatorio**:
  **s= ($x_1$, $x_2$, ..., $x_m$)**, con $m \leq n$, $x_i \in \{1,..,n\}$ y $x_i < x_{i+1}$
  - $x_i \rightarrow$ Número de objeto escogido
  - m $\rightarrow$ Número total de objetos escogidos
  - Las soluciones están en cualquier nivel



**2. Elegir el esquema de algoritmo: caso optimización** (maximización).

**voa**: valor óptimo actual
**soa:** solución óptima actual

**pact**: Peso actual
**bact:** Beneficio actual

```
Backtracking (var s: array [1..n] de entero)
    nivel:= 1;  s:= sINICIAL
    voa:= -∞; soa:= Ø
    pact:= 0; bact:= 0
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s)   AND (bact > voa) entonces
            voa:= bact;   soa:= s
        si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT HayMasHermanos (nivel, s) AND (nivel>0) hacer
                Retroceder (nivel, s)
        finsi
    hasta nivel==0
```

## 3. Funciones genéricas del esquema.

- **Generar (nivel, s)** → Probar primero 0 y luego 1

  s[nivel]:= s[nivel] + 1

  pact:= pact + p[nivel]*s[nivel]

  bact:= bact + b[nivel]*s[nivel]

  **si** s[nivel]==1 **entonces**
  pact:= pact + p[nivel]
  bact:= bact + b[nivel]
  **finsi**

- **EsSolución (nivel, s)**

  **devolver** (nivel==n) AND (pact≤M)

- **Criterio (nivel, s)**

  **devolver** (nivel<n) AND (pact≤M)

- **HayMasHermanos (nivel, s)**

  **devolver** s[nivel] < 1

- **Retroceder (nivel, s)**

  pact:= pact – p[nivel]*s[nivel]

  bact:= bact – b[nivel]*s[nivel]

  s[nivel]:= -1

  nivel:= nivel – 1

**algoritmo: caso optimización** (maximización). **Poda según el criterio de peso**

**Backtracking (var s: array[1..n] de entero)** **s**: array [1..n] de {-1, 0, 1}

nivel:= 1; s:= $s_{INICIAL}$ **$s_{INICIAL}$: {-1, -1, …, -1}**

**pact:=0; bact:= 0;**

**voa:= -∞; soa:= Ø**

**repetir**

Generar (nivel, s)

```
s[nivel]:= s[nivel] + 1;
pact:= pact + p[nivel] * s[nivel]
bact:= bact + b[nivel] * s[nivel]
```

→

```
si s[nivel]==1 entonces
    pact:= pact + p[nivel]
    bact:= bact + b[nivel]
finsi
```

**si** ~~EsSolución (nivel, s)~~ (nivel==n) Y (pact ≤ M) **AND bact > voa entonces**

**voa:= bact; soa:= s**

**si** ~~Criterio (nivel, s)~~ (nivel<n) Y (pact ≤ M) **entonces** //poda según el criterio de peso

nivel:= nivel + 1

**sino**

**mientras** NOT ~~HayMasHermanos (nivel, s)~~ s[nivel]<1 **AND (nivel>0) hacer**

Retroceder (nivel, s)

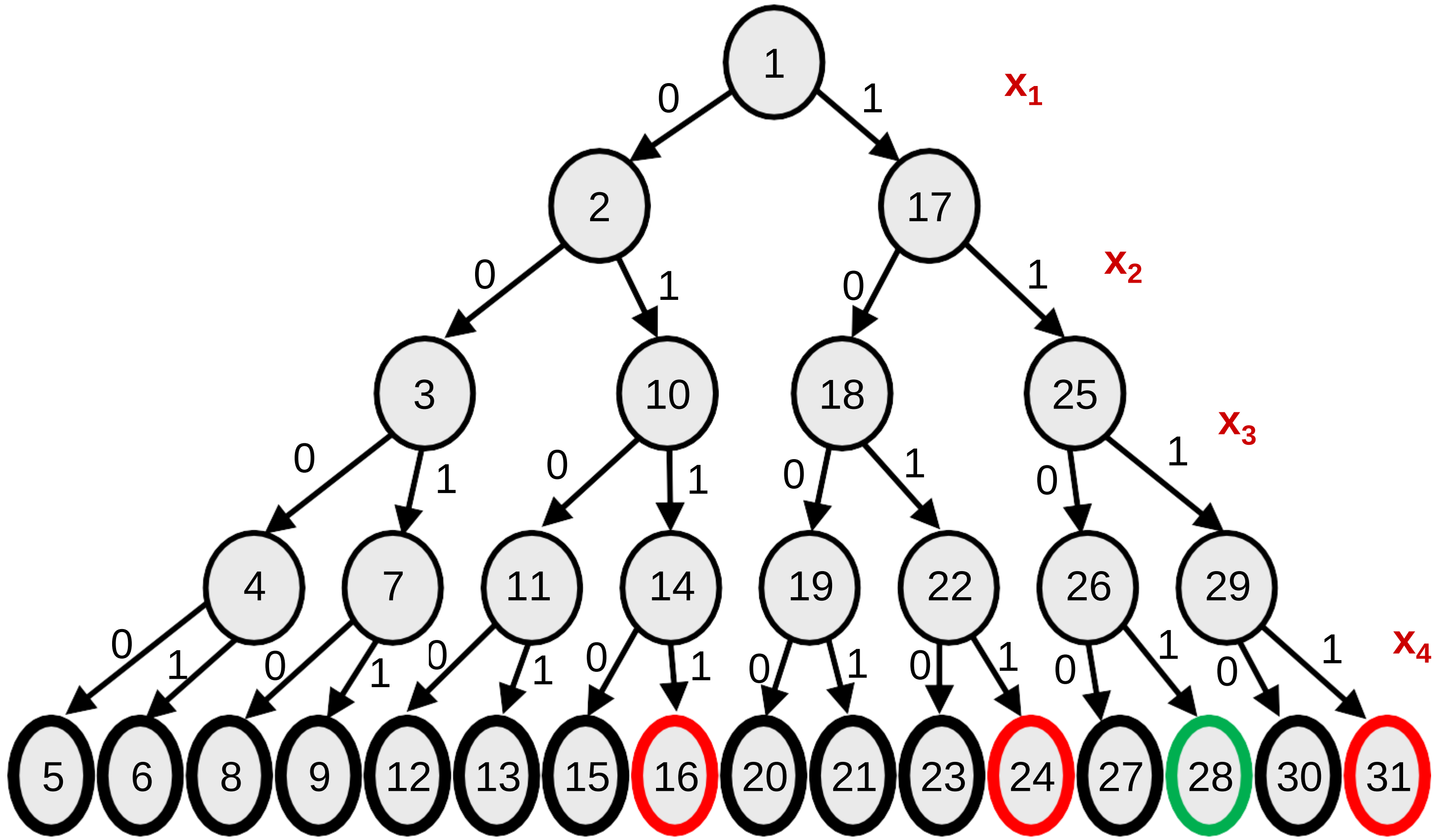
```
pact:= pact - p[nivel] * s[nivel]
bact:= bact - b[nivel] * s[nivel]
s[nivel]:= -1
nivel:= nivel - 1
```

**finsi**

**hasta nivel==0**

| | |
|---|---|
| **pact**: Peso actual | **voa**: valor óptimo actual |
| **bact:** Beneficio actual | **soa:** solución óptima actual |

- **Ejemplo:** n = 4; M = 7;  b = (2, 3, 4, 5); p = (1, 2, 3, 4)




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **pact**: | 0 | 4 | **7** | **9** | **8** | **7** | **10** |
| **bact**: | 0 | 5 | 9 | 12 | 11 | **10** | 14 |

- El algoritmo resuelve el problema, encontrando la solución óptima, pero:
  - Es muy ineficiente. ¿Cuánto es el orden de complejidad?
  - **Problema adicional:** en el ejemplo, se generan todos los nodos posibles, no hay ninguna poda. La función **Criterio** es siempre cierta (excepto para algunos nodos hoja).
- **Solución:** Mejorar la poda con una función **Criterio** más restrictiva.
- Incluir una poda adicional **según el criterio de optimización**.
  - **Poda según el criterio de peso:** si el peso actual es mayor que M podar el nodo (incluido en la página anterior).
  - **Poda según el criterio de optimización:** si el beneficio actual no puede mejorar el **voa** podar el nodo.

    **bmax: entero**. Suma valores de los objetos aun no considerados
    - **Inicialización** (añadir)**:** bmax:= b[2]+ … + b[n]
    - nivel:= nivel + 1 (añadir)**:** bmax:= bmax - b[nivel]
    - nivel:= nivel - 1 (añadir)**:** bmax:= bmax + b[nivel]

**algoritmo: caso optimización** (maximización). **Poda según el criterio de peso y optimización**

**Backtracking (var s: array[1..n] de entero)** **s**: array [1..n] de {-1, 0, 1}

nivel:= 1; s:= $s_{INICIAL}$ **$s_{INICIAL}$: {-1, -1, …, -1}**

**pact:=0; bact:= 0;** bmax:= b[2]+ … + b[n]

**voa:= -∞; soa:= Ø**

**repetir**

Generar (nivel, s)

s[nivel]:= s[nivel] + 1;
pact:= pact + p[nivel] * s[nivel]
bact:= bact + b[nivel] * s[nivel]

**si** s[nivel]==1 **entonces**
pact:= pact + p[nivel]
bact:= bact + b[nivel]

**si** ~~EsSolución (nivel, s)~~ (nivel==n) Y (pact ≤ M) **AND bact > voa entonces**

**voa:= bact;** **soa:= s**

**si** Criterio (nivel, s) **entonces**

//poda según el criterio de peso y optimización
(nivel<n) Y (pact ≤ M) Y (bact + bmax > voa)

nivel:= nivel + 1; bmax:= bmax - b[nivel]

**sino**

**mientras** NOT ( ~~HayMasHermanos (nivel, s)~~ s[nivel]<1 Y Criterio (nivel, s) )

**AND (nivel>0) hacer**

Retroceder (nivel, s)

pact:= pact - p[nivel] * s[nivel]
bact:= bact - b[nivel] * s[nivel]
s[nivel]:= -1
bmax:= bmax + b[nivel]
nivel:= nivel - 1

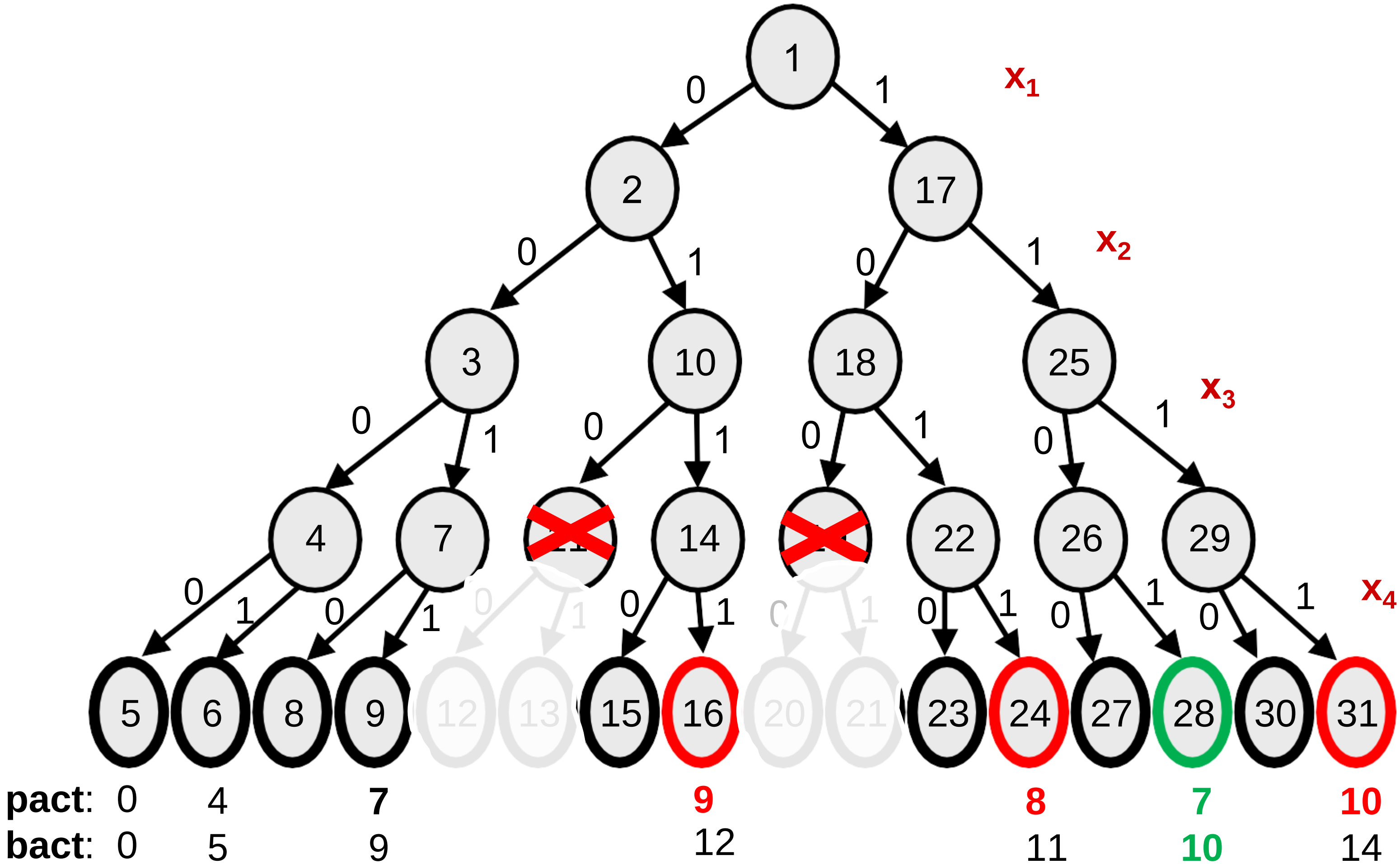
**finsi**

**hasta nivel==0**

**pact**: Peso actual **voa**: valor óptimo actual

**bact:** Beneficio actual **soa:** solución óptima actual

- **Ejemplo:** n = 4; M = 7;  b = (2, 3, 4, 5); p = (1, 2, 3, 4)



**algoritmo: caso optimización** (maximización). **Poda según el criterio de peso y optimización**

**Mejora: Generar primero el 1 y luego el 0**

```
Backtracking (var s: array[1..n] de entero)     s: array [1..n] de {0, 1, 2}
    nivel:= 1;  s:= s_INICIAL     s_INICIAL: {2, 2, …, 2}
    pact:=0; bact:= 0; bmax:= b[2]+ … + b[n]
    voa:= -∞; soa:= Ø
    repetir
        Generar (nivel, s)
        si ~~EsSolución (nivel, s)~~ (nivel==n) Y (pact ≤ M)  AND bact > voa entonces
            voa:= bact;     soa:= s
        si Criterio (nivel, s)   entonces
            nivel:= nivel + 1; bmax:= bmax - b[nivel]
        sino
            mientras NOT ( ~~HayMasHermanos (nivel, s)~~ s[nivel]>0 Y Criterio (nivel, s) )
                        AND (nivel>0) hacer
                Retroceder (nivel, s)
        finsi
    hasta nivel==0
```

Generar (nivel, s):

```
s[nivel]:= s[nivel] - 1;
si s[nivel]==1 entonces
    pact:= pact + p[nivel]
    bact:= bact + b[nivel]
finsi
```

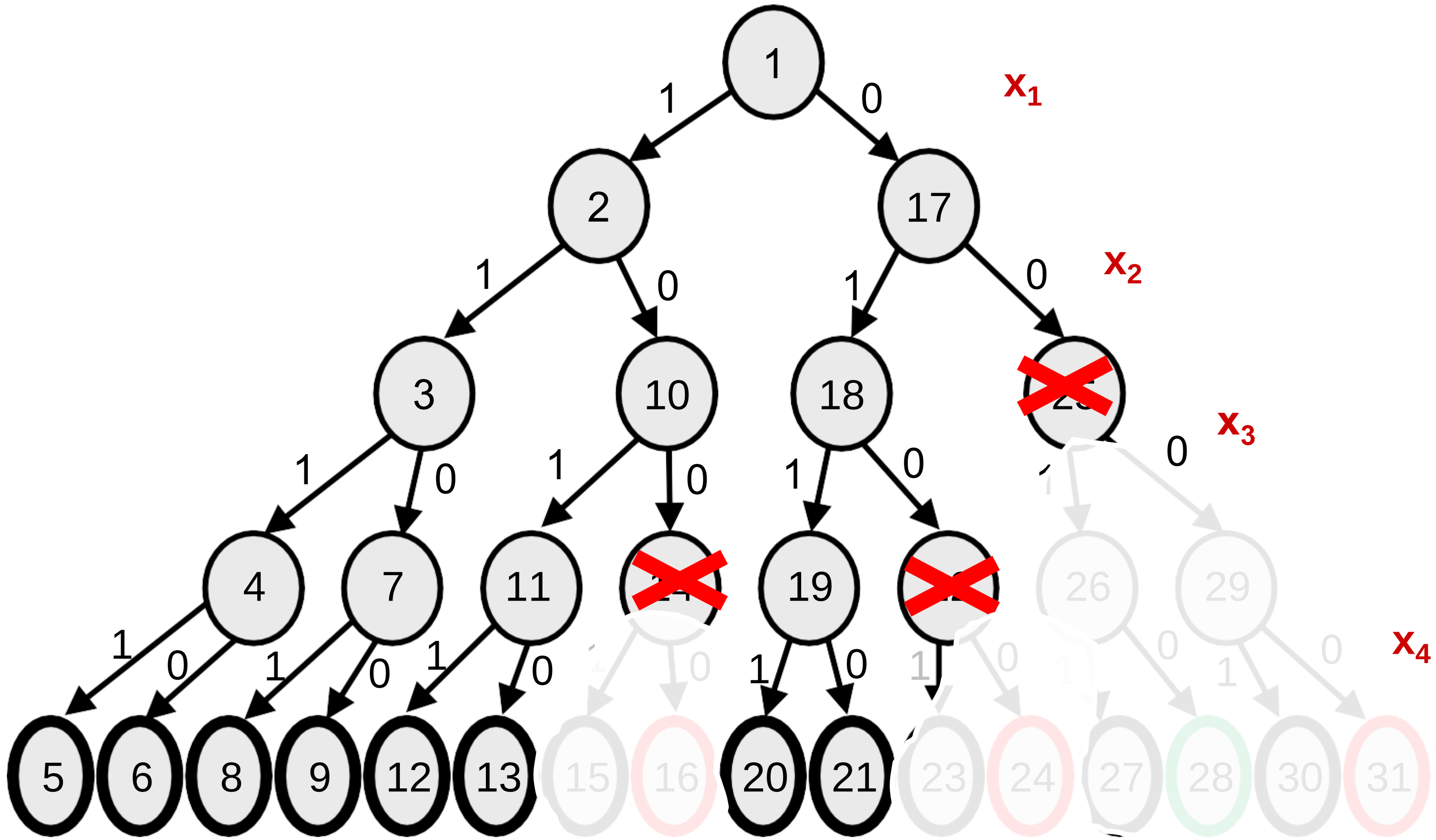
Criterio (nivel, s):

```
//poda según el criterio de peso y optimización
(nivel<n) Y (pact ≤ M) Y (bact + bmax > voa)
```

Retroceder (nivel, s):

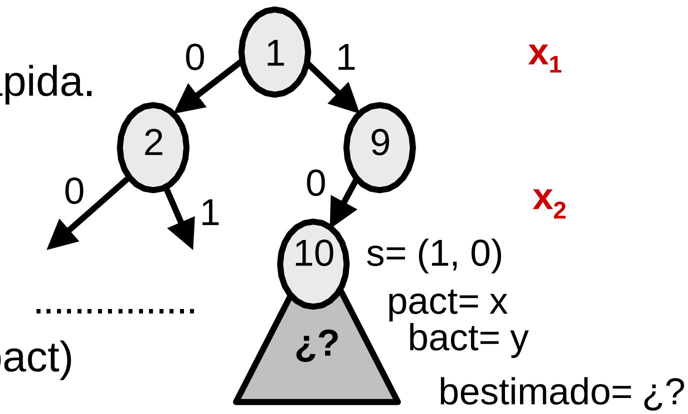
```
si s[nivel]==1 entonces
    pact:= pact - p[nivel]
    bact:= bact - b[nivel]
finsi
s[nivel]:= 2
bmax:= bmax + b[nivel]
nivel:= nivel  - 1
```

- **Ejemplo:** n = 4; M = 7;  b = (2, 3, 4, 5); p = (1, 2, 3, 4)




**pact**: 10 6 7 3 4 5
**bact**: 14 9 10 5 6 7

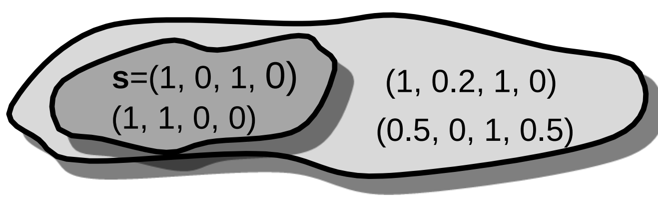
- ¿Cómo calcular una cota superior del beneficio que se puede obtener a partir del nodo actual, es decir ($x_1$, ..., $x_k$)?
- La estimación debe poder realizarse de forma rápida.




- La **estimación del beneficio** para el nivel y nodo actual será:
  bestimado:= bact + Estimacion (nivel+1, n, M - pact)

- **Estimacion (k, n, Q)**: Estimar una cota superior para el problema de la mochila 0/1, usando los objetos **k..n**, con capacidad máxima **Q**.
- ¿Cómo? **Idea:** el resultado del problema de la mochila (no 0/1) es una cota superior válida para el problema de la mochila 0/1.
- **Demostración:**

**Soluciones mochila 0/1**




**Soluciones mochila no 0/1**

- Sea **s** la solución óptima de mochila 0/1. **s** es válida para la mochila (no 0/1). Por lo tanto, la solución óptima de la mochila (no 0/1) será **s** o mayor.

- **Estimacion (k, n, Q)**: Aplicar el algoritmo voraz para el problema de la mochila[1] (no 0/1), con los objetos **k..n**. Si devuelve un resultado float y en mochila 0/1 los beneficios son enteros, nos quedamos con la parte entera del resultado.
- ¿Qué otras partes se deben modificar?
- **Criterio (nivel, s)** /* poda si **pact > M** o si **bestimado ≤ voa** */

  **si** (pact > M) OR (nivel == n) **entonces devolver** FALSO /* poda si pact > M */

  **sino**

  bestimado:= bact + $\lfloor$MochilaVoraz (nivel+1, n, M - pact)$\rfloor$

  **devolver** bestimado > voa /* poda si bestimado ≤ voa */

  **finsi**
- En el algoritmo principal:

  **.....**

  **mientras** (NOT HayMasHermanos (nivel, s) OR

  NOT Criterio (nivel, s)) AND (nivel > 0) **hacer**

  Retroceder (nivel, s)

  **…..**

[1] dados b y p, el voraz coge 1º los objetos con mayor b/p (entero si cabe o la fracción que quepa, que será el último de los metidos)

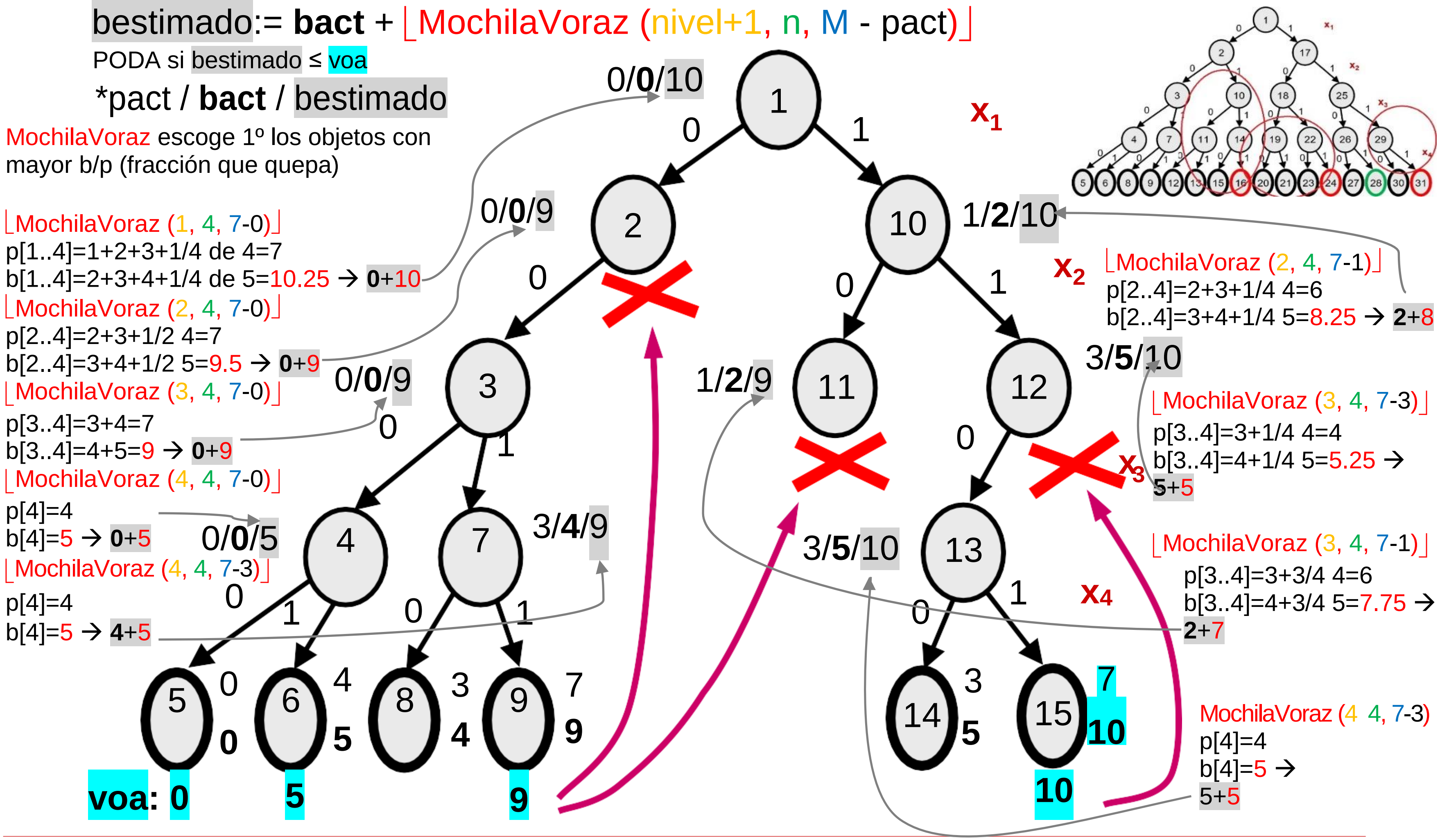
- **Ejemplo:** n = 4; M = 7; b = (2, 3, 4, 5); p = (1, 2, 3, 4) ; b/p=(2, 1. 5, 1.3, 1.25)

  bestimado:= **bact** + ⌊MochilaVoraz (nivel+1, n, M - pact)⌋

  PODA si bestimado ≤ voa

  *pact / **bact** / bestimado

MochilaVoraz escoge 1º los objetos con mayor b/p (fracción que quepa)



- Se eliminan nodos, pero a costa de aumentar el tiempo de ejecución en cada nodo.
- ¿Cuál será el tiempo de ejecución total?
- Suponiendo los objetos ordenados por $b_i/p_i$...
- Tiempo de la función **Criterio** en el nivel **i** (en el peor caso) es

  $T_{Criterio}$ =1 + Tiempo de la función **MochilaVoraz** = **1 + *n* - *i***
- **Idea intuitiva.** Tiempo en el peor caso (suponiendo todos los nodos): Número de nodos $O(2^n)$ ***** Tiempo de cada nodo(función criterio) $O(n)$.
- ¿Tiempo: $O(n \cdot 2^n)$? **NO**

$$t(n) = \sum_{i=1}^{n} 2^i \cdot (n - i + 1) = (n+1)\sum_{i=1}^{n} 2^i - \sum_{i=1}^{n} i \cdot 2^i = 2 \cdot 2^{n+1} - 2n - 4$$

## Conclusiones:

- El cálculo intuitivo no es correcto.
- En el peor caso, el orden de complejidad sigue siendo un $O(2^n)$.
- En promedio se espera que la poda elimine muchos nodos, reduciendo el tiempo total.
- Pero el tiempo sigue siendo muy malo. ¿Cómo mejorarlo?
- Posibilidades:
  - 1. Generar primero el 1 y luego el 0.
  - 2. Usar un árbol combinatorio.
  - ...

- **Modificación 1ª: Generar primero el 1 y luego el 0.**
- **Ejercicio:** Cambiar las funciones:

  Generar y HayMasHermanos.
- **Ejemplo:** n = 4; M = 7

  b = (2, 3, 4, 5)

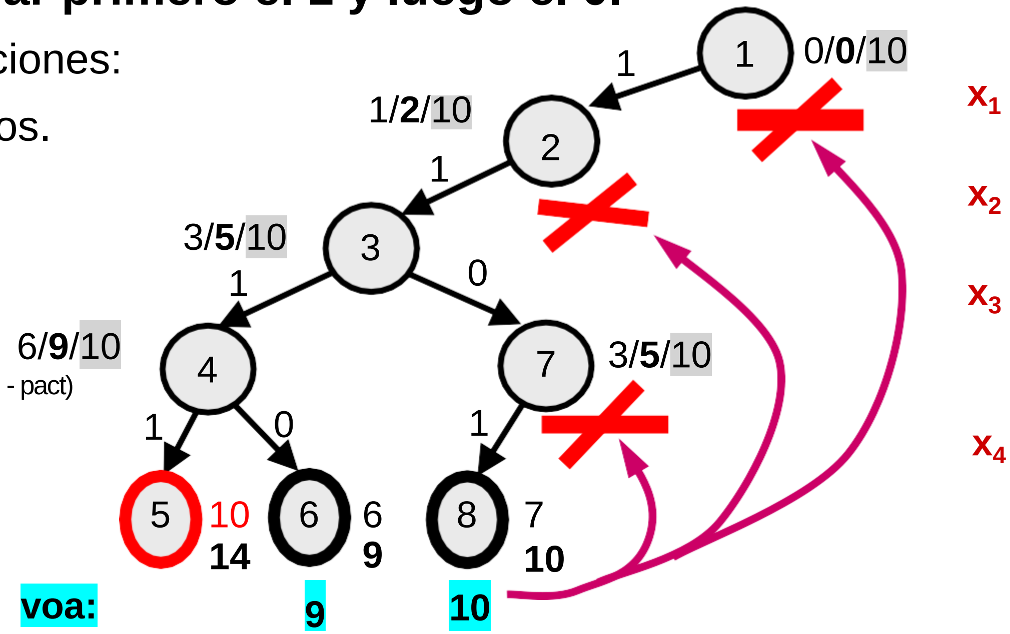
  p = (1, 2, 3, 4)

  b/p=(2, 1.5, 1.3, 1.25)

  bestimado:= **bact** + ⌊MochilaVoraz (nivel+1, n, M - pact)

  pact / **bact** / bestimado

  PODA si bestimado ≤ voa

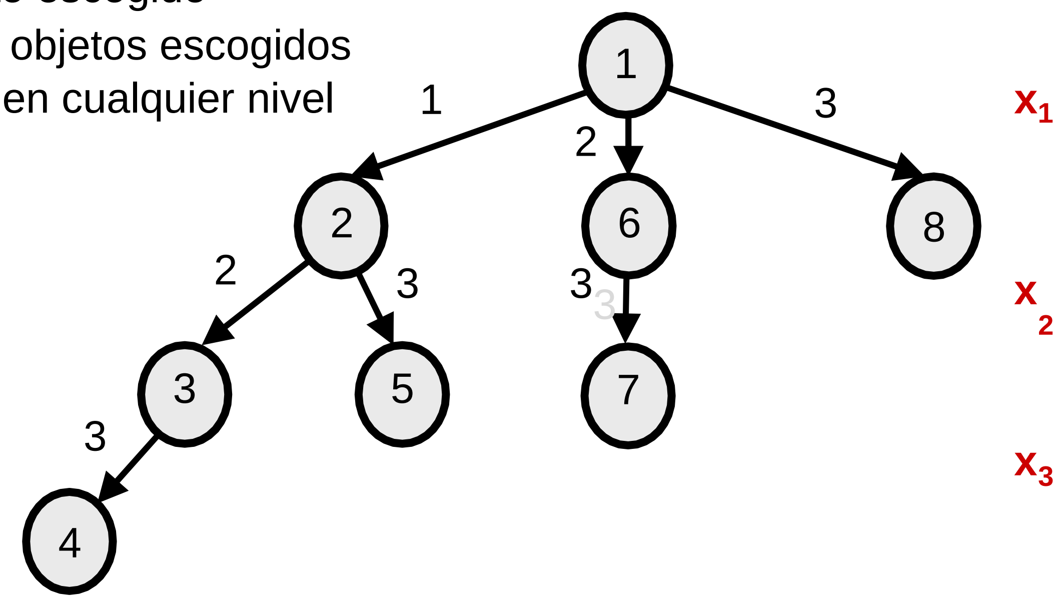



- En este caso es mejor la estrategia “primero el 1”, pero ¿y en general?
- Si la solución óptima es de la forma s = (1, 1, 1, X, X, 0, 0, 0) entonces se alcanza antes la solución generando primero 1 (y luego 0).
- Si es de la forma s = (0, 0, 0, X, X, 1, 1, 1) será mejor empezar por 0.
- **Idea:** es de esperar que la solución de la mochila 0/1 sea “parecida” a la de la mochila **no** 0/1. Si ordenamos los objetos por $b_i/p_i$ entonces tendremos una solución del primer tipo.

- **Modificación 2ª: Usar un árbol combinatorio.**
- **Representación de la solución:**

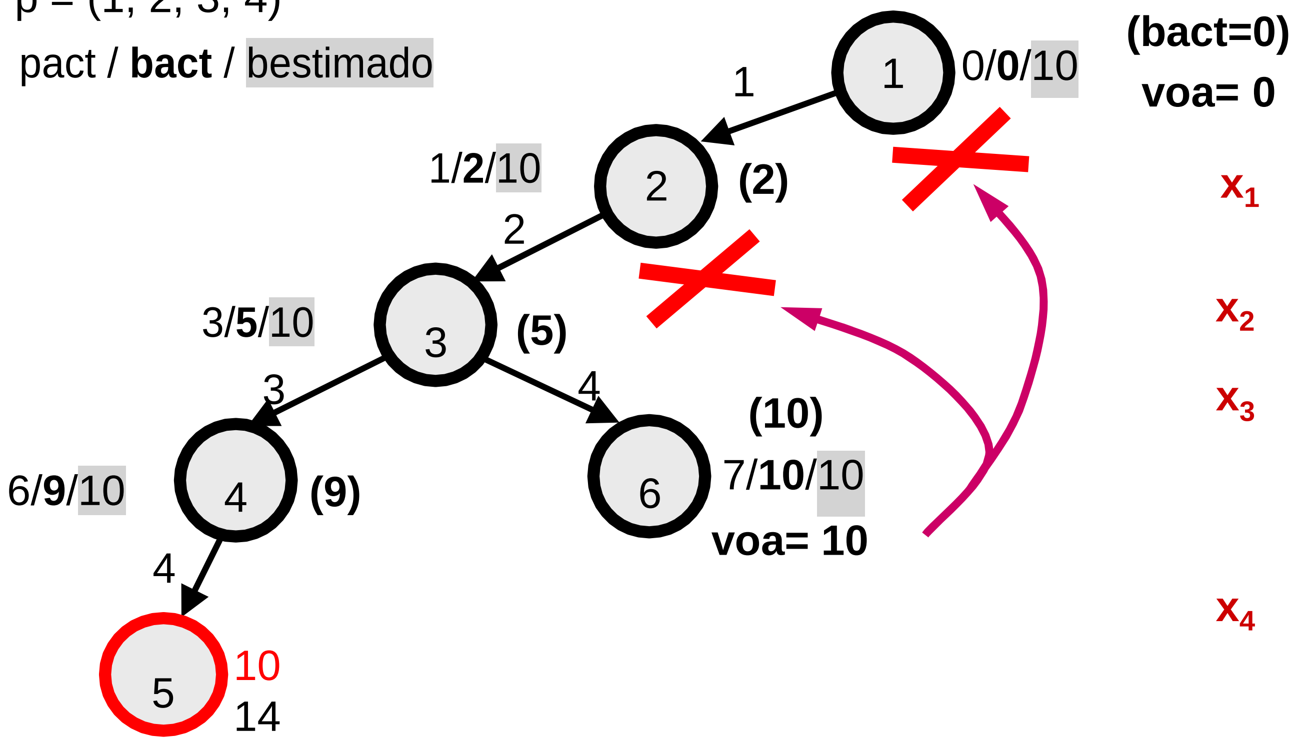
  **s= ($x_1$, $x_2$, ..., $x_m$)**, con $m \le n$, $x_i \in \{1,..,n\}$ y $x_i < x_{i+1}$

  - $x_i$ → Número de objeto escogido
  - m → Número total de objetos escogidos
  - Las soluciones están en cualquier nivel




- **Ejercicio:** Cambiar la implementación para generar este árbol.
  - Esquema del algoritmo: nos vale el mismo.
  - Modificar las funciones Generar, Solución, Criterio y HayMasHermanos.

- **Ejemplo:** n = 4; M = 7
  b = (2, 3, 4, 5)
  p = (1, 2, 3, 4)
  pact / **bact** / bestimado




PODA si bestimado ≤ **voa**

- **Resultado:** conseguimos reducir el número de nodos.
- ¿Mejorará el tiempo de ejecución y el orden de complejidad?

## 5. Algoritmos Bactracking. Ejemplos. 2_Problema de asignación de tareas.

- Existen **n** empleados y **n** tareas a realizar.
- Se representa mediante una tabla M de tamaño n x n, **M[i, j]**, el coste de realizar la tarea j por el empleado i , para i , j = 1,…, n.
- **Objetivo:** asignar una tarea a cada trabajador (asignación uno-a-uno), de manera que se **minimice el coste total**.

**Tareas** (columnas) / **Empleados** (filas)

| M | 1 | 2 | 3 |
|---|---|---|---|
| **1** | 3 | 5 | 1 |
| **2** | 10 | 10 | 1 |
| **3** | 8 | 5 | 5 |

- **Ejemplo 1.** s= {T1,T3,T2} /* (E1,T1), (E2,T3), (E3,T2) */

  $M_{TOTAL}$= 3+1+5= **9**
- **Ejemplo 2.** s= {T2,T1,T3} /* (E1,T2), (E2,T1), (E3,T3)*/

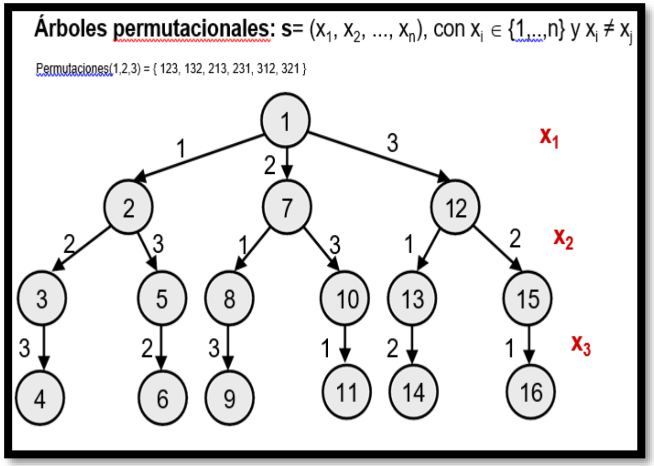
  $M_{TOTAL}$= 5+10+5= **30**

> El **problema de asignación** es un problema **NP-completo** clásico.

- **Otras variantes y enunciados:**
  - Problema de los **matrimonios estables**.
  - Problemas con **distinto número** de tareas y personas. Ejemplo: problema de los árbitros.
  - Problemas de **asignación de recursos**: fuentes de oferta y de demanda. Cada fuente de oferta tiene una capacidad O[i] y cada fuente de demanda una D[j].
  - **Isomorfismo de grafos:** la matriz de pesos varía según la asignación realizada.

- **Datos del problema:**
  - **n:** número de empleados y de tareas disponibles.
  - **M: array** [1..n, 1..n] **de** entero. Coste (rendimiento) de cada asignación. **M[i, j]** = coste de realizar la tarea **j** por el empleado **i**.
  - El problema consiste en asignar a cada operario i una tarea j de forma que se minimice el coste total.
- La **solución** se puede representar como una tupla

  **s= {x1, x2, ..., x$_n$}**




- Restricciones explicitas: $x_i \in \{1,\dots,n\}$
  - $\mathbf{x_i}$ es la **tarea** asignada al ***i-ésimo*** empleado
- Restricciones implícitas: $\mathbf{x_i \neq x_j}$, $\forall$ **i≠j**
- El objetivo es minimizar la función $\sum_{i=1}^{n} M[i, x_i]$
- Por cada solución que encuentre el algoritmo, se anotará su coste y se comparará con el coste de la mejor solución encontrada hasta el momento

## Algoritmo de backtracking recursivo versión 1: Usando función noRepetida()

(**coste** por referencia, no por valor …o poner **coste** como variable global y que no sea un parámetro más)

```
proc Tareas(M[1..n,1..n], tarea[1..n], mejorX[1...n], costeAcIni, coste, empleado)
  tarea[empleado]:= 0
  repetir
    tarea[empleado] := tarea[empleado] + 1
    si noRepetida(tarea, empleado) entonces
      costeAc := costeAcIni + M[empleado, tarea[empleado]]
      si (costeAc ≤ coste) entonces
        si empleado < n entonces
          tareas(M, tarea, mejorX, costeAc, coste, etapa+1)
        sino
          mejorX := tarea
          coste :=  costeAc
        fsi
      fsi
    fsi
  hasta tarea[empleado]=n
fproc
```

**El procedimiento que invoca al algoritmo es:**

```
proc invoca_tareas(M[1..n,1..n],X[1..n],C)
  crear XAct[1..n]
  C:= ∞
  Tareas(M, XAct, X, 0, C, 1)
fproc
```

```
//falso si Asignada[n] mismo valor que alguna anterior
//xi ≠ xj, ∀ i≠j
función noRepetida(Asignadas, n)
  para i := 1 hasta n-1 hacer
    si Asignadas[i] = Asignadas[n] entonces
        devolver falso
    fsi
  fpara
  devolver cierto
ffun
```

## Algoritmo de backtracking recursivo versión 2: Usando array usada[1..n]

(**coste** por referencia, no por valor ...o poner **coste** como variable global y que no sea un parámetro más)

```
proc Tareas(M[1..n,1..n], usada[1..n], tarea[1..n], mejorX[1...n], costeAcIni,
coste, empleado)
  tarea[empleado]:= 0
  repetir
    tarea[empleado] := tarea[empleado] + 1
    si usada[tarea[empleado]]=false entonces
      costeAc := costeAcIni + M[empleado, tarea[empleado]]
      usada[tarea[empleado]] := true
      si (costeAc ≤ coste) entonces
         si empleado < n entonces
            tareas(M, tarea, mejorX, costeAc, coste, etapa+1)
         sino
            mejorX := tarea
            coste :=  costeAc
         fsi
      fsi
      usada[tarea[empleado]] := false
    fsi
  hasta tarea[empleado]=n
fproc
```

**El procedimiento que invoca al algoritmo es:**

```
proc invoca_tareas(M[1..n,1..n],X[1..n],C)
  crear XAct[1..n] // enteros
  crear asignadas[1..n] //booleanos
  para i:= 1 hasta n hacer
    asignadas[i] := false
  C:= ∞
  Tareas(M, asignadas, XAct, X, 0, C, 1)
fproc
```

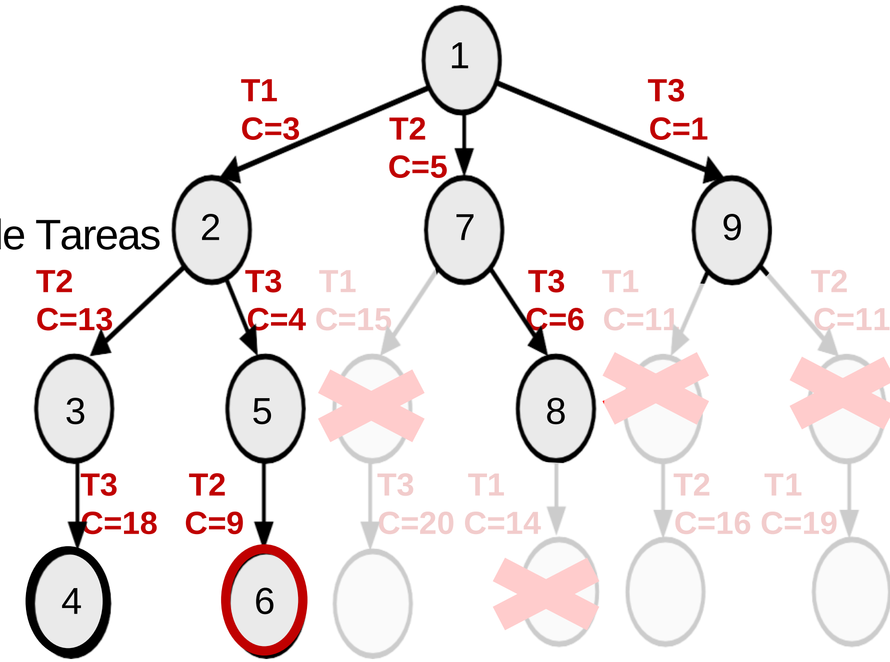
- **Ejemplo**:

Tareas

Empleados

| M | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 3 | 5 | 1 |
| 2 | 10 | 10 | 1 |
| 3 | 8 | 5 | 5 |

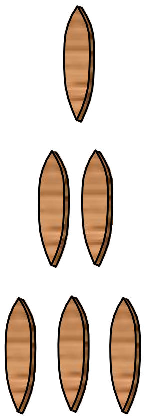
- **Secuencia de llamadas** para la matriz M de Tareas




```
tareas(XAct, mejorX, costeAcIni, coste, etapa)
 tareas([0,0,0], [0,0,0], 0, ∞, 1)
    tareas([1,0,0], [0,0,0], 3, ∞, 2)
      tareas([1,2,0], [0,0,0], 13, ∞, 3)
      tareas([1,3,0], [1,2,3], 4, 18, 3)
    tareas([2,0,0], [1,3,2], 5, 9, 2)
      tareas([2,3,0], [1,3,2], 6, 9, 3)
    tareas([3,0,0], [1,3,2], 1, 9, 2)
```

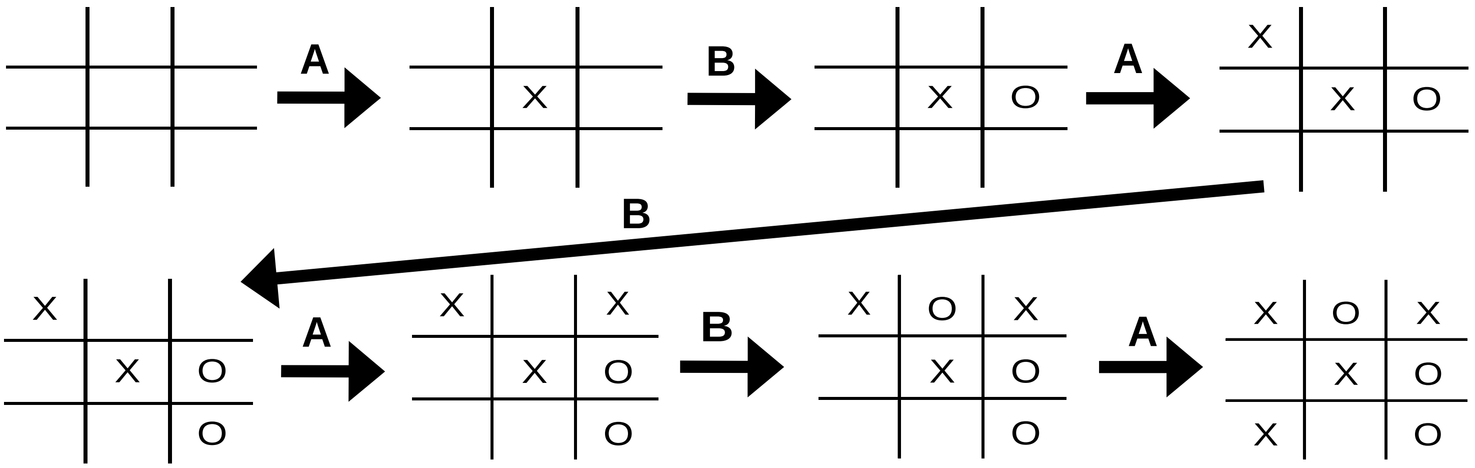
- El algoritmo realiza podas en el árbol de expansión eliminando aquellos nodos que no van a llevar a la solución optima
- **Ejercicio**: Realizar el ejemplo con el esquema NO recursivo

- La idea de backtracking (recorrido exhaustivo del árbol de un problema) se puede aplicar en **problemas de juegos**.
  - **Objetivo final:** decidir el movimiento óptimo que debe realizar el jugador que empieza moviendo.
- **Características (juegos de inteligencia):**
  - En el juego participan dos jugadores, **A** y **B**, que mueven alternativamente (primero **A** y luego **B**).
  - En cada movimiento un jugador puede elegir entre un número finito de posibilidades.
  - El resultado del juego puede ser: gana **A**, gana **B** o hay empate. El **objetivo** de los dos jugadores es **ganar**.
  - Supondremos juegos en los que no influye el azar.
  - **Ejemplos.** Las tres en raya, las damas, el ajedrez, el NIM, el juego de los palillos, etc.



- **Ejemplo.** El juego de los palillos.
  - Tres filas de palillos (en general **n**).
  - Cada jugador debe quitar uno o varios palillos, pero siempre de la misma fila.
  - Pierde el que quite el último palillo.

- **Ejemplo.** El juego de las tres en raya.




Gana **A**

- Una partida es una **secuencia** de movimientos.
- Si representamos todas las partidas (todos los posibles movimientos) tenemos **un árbol.**

## Árboles de juegos:

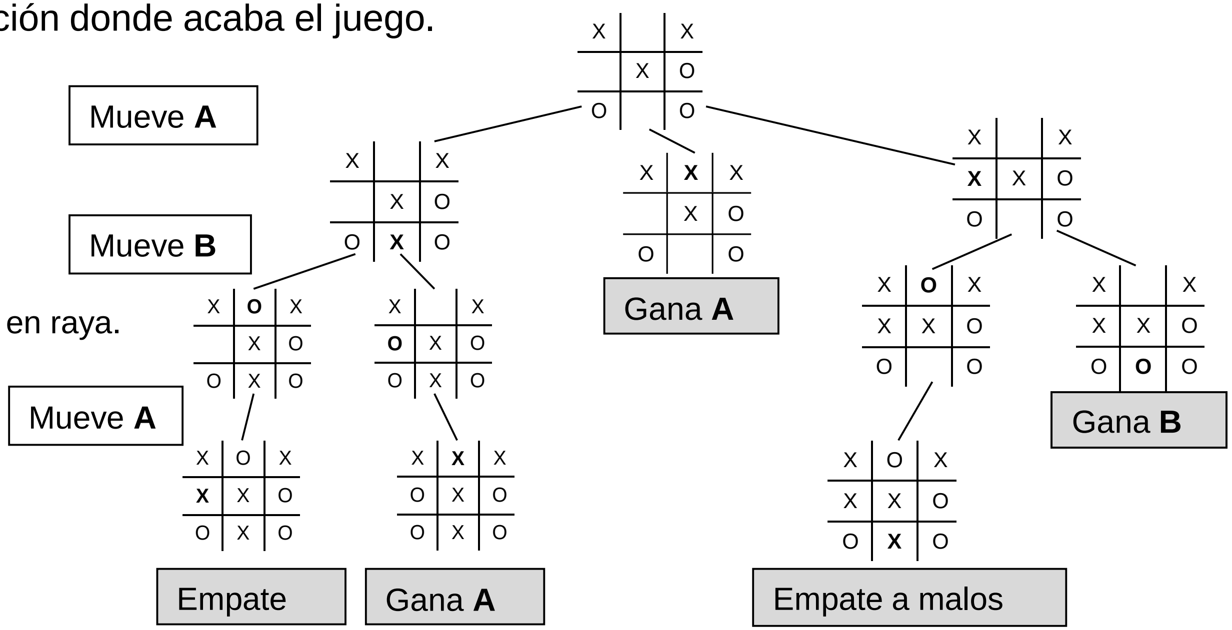
- Cada nodo del árbol representa un posible estado del juego.
- La **raíz** representa el comienzo de una partida.
- Los **descendientes** de un nodo dado son los movimientos posibles de cada jugador.
- En el nivel 1 mueve el jugador **A**.
- En el nivel 2 mueve el jugador **B**.
- En el nivel 3 mueve el jugador **A**.
- En el nivel 4 mueve el jugador **B**.
- ...
- Una hoja es una situación donde acaba el juego.

## Ejemplo.

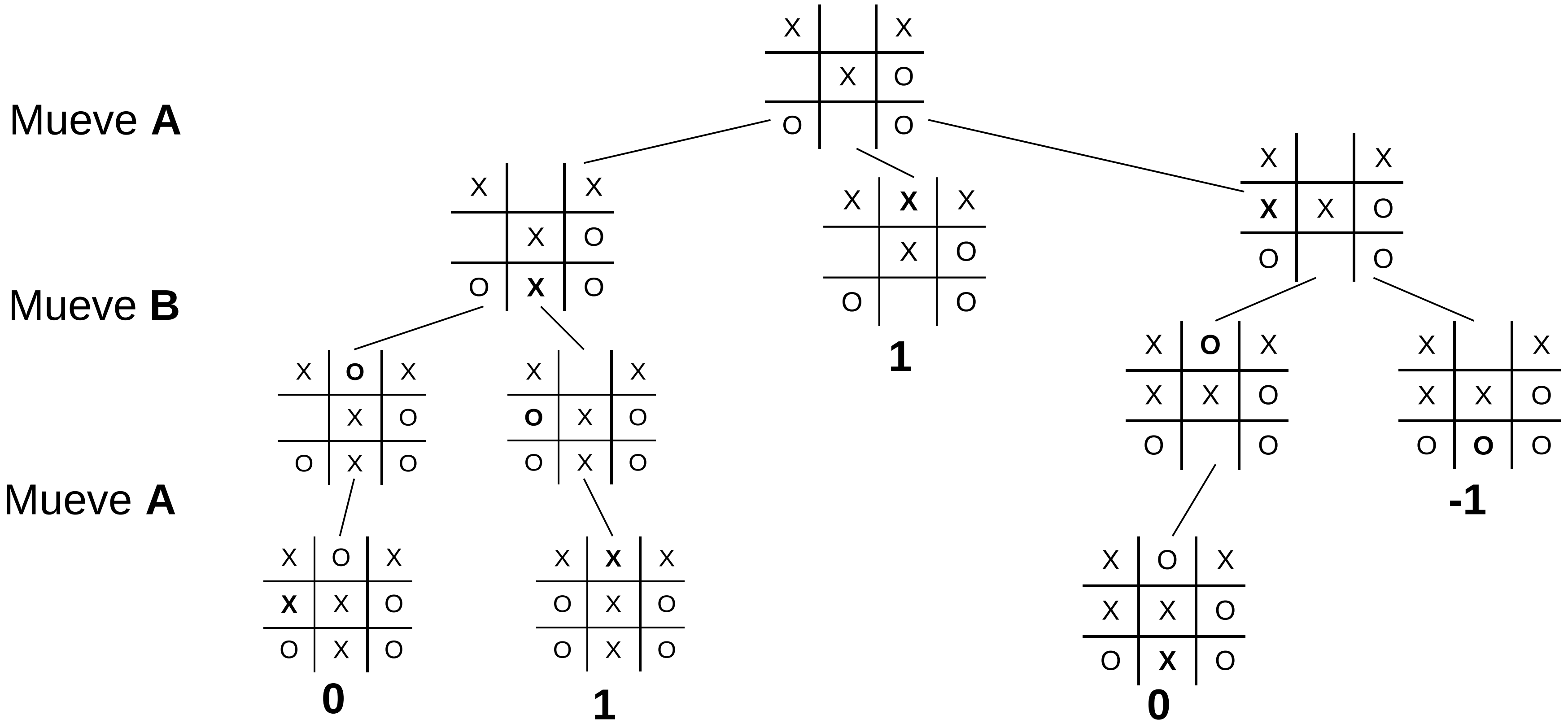
Parte del árbol de juego tres en raya.

**A → X**

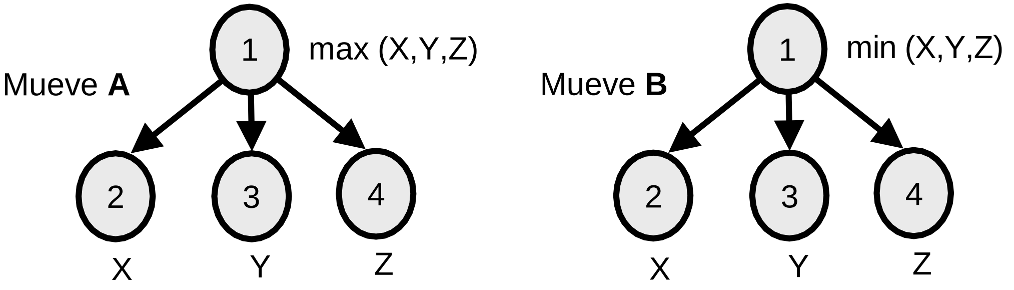
**B → O**



- Etiquetamos cada hoja con un número, que valdrá:
  - **1** → Si el juego finaliza con victoria de **A**.
  - **-1** → Si acaba con victoria de **B**.
  - **0** → Si se produce un empate.



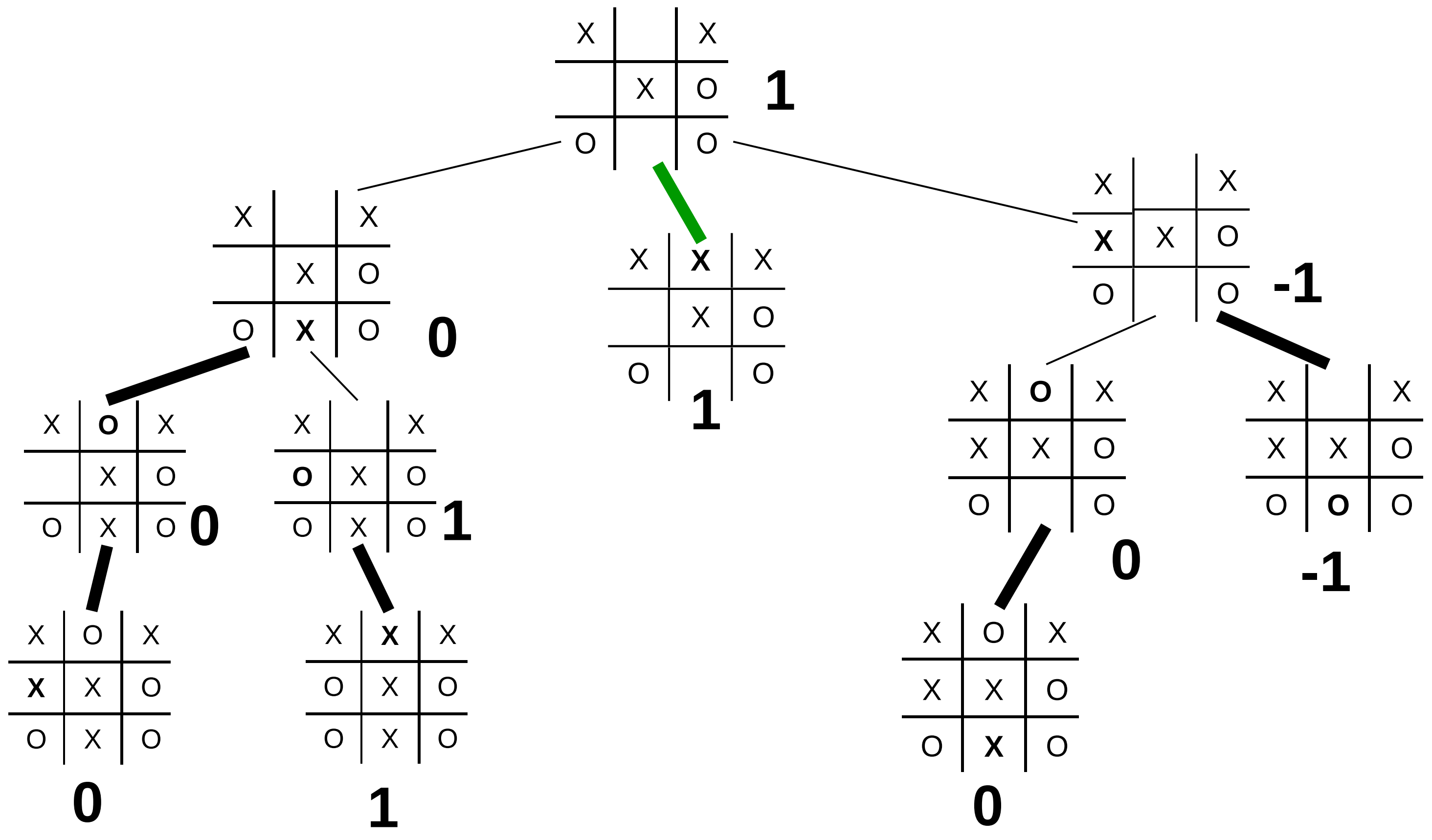
- El objetivo para **A**: encontrar un camino en el árbol que le lleve hasta una hoja con valor 1.
- Pero, ¿qué pasa si a partir de la situación inicial no se llega a un nodo hoja con valor 1? →
  - → En los movimientos de **B**, el jugador **B** intentará llegar a hojas con valor -1 (ó en caso de no existir, de valor 0).
  - → En los movimientos de **A**, el jugador **A** intentará llegar a hojas con valor 1 (ó en caso de no existir, de valor 0).
- De esta manera se define una forma de propagar el valor de los hijos hacia los padres: **estrategia minimax**.
- **Estrategia minimax.** Los valores de las hojas se **propagan** a los padres de la siguiente forma:
  - En los movimientos de **A**, el valor del nodo padre será el **máximo** de los valores de los nodos hijos.
  - En los movimientos de **B**, el valor del nodo padre será el **mínimo** de los valores de los nodos hijos.
  - Se repite hasta llegar al nodo raíz (situación de partida).



Mueve **A**
**MAX**

Mueve **B**
**MIN**

Mueve **A**
**MAX**




- **Movimiento óptimo:** aquel que conduzca al máximo. O si el primer nivel es un MIN, el que conduzca al mínimo.

- En general, tendremos una **función de utilidad**.
- **Función de utilidad:** para cada nodo hoja devuelve un valor numérico, indicando cómo de buena es esa situación para el jugador **A.**
- Si el árbol del juego es muy grande o infinito (por ejemplo, en el ajedrez) entonces la función de utilidad debe poder aplicarse sobre situaciones no terminales.
- En ese caso, la función de utilidad es una medida heurística: cómo es de prometedora la situación para **A.**

- **Proceso de resolución de juegos:**
  - Generar el árbol de juego hasta un nivel determinado. ¿Cuánto?
  - Aplicar la función de utilidad a los nodos hoja.
  - Propagar los valores de utilidad hasta la raíz, usando la **estrategia minimax**:
    - En los movimientos impares tomar el máximo de los hijos.
    - En los movimientos pares tomar el mínimo de los hijos.
  - **Solución final:** escoger el movimiento indicado por el hijo de la raíz con mayor valor.

- **Implementación:** Usar un backtracking recursivo.
- Backtracking → el recorrido será en profundidad.

- **Implementación de la estrategia minimax.**

**BuscaMinimax (B: TipoTablero; modo: (MAX, MIN)) : real**

**si EsHoja(B) entonces** Indica si el nodo es una situación terminal, o si estamos en el nivel máximo

**devolver Utilidad (B, modo)** Devuelve el valor de la función de utilidad para el tablero **B** en el **modo** indicado

**sino**

**si** modo == MAX entonces valoract:= -∞

**sino** valoract:= ∞

**para cada hijo C del tablero B hacer** Iterador para generar todos los movimientos a partir de una situación de partida **B**

**si** modo == MAX **entonces**

valoract:= max (valoract, BuscaMinimax(C, MIN))

**sino**

valoract:= min (valoract, BuscaMinimax(C, MAX))

**finsi**

**fpara**

**devolver** valoract

**fsi**

- **Tipos de datos:**
  - **TipoTablero:** Representación del estado del juego en un momento dado.
- **Funciones genéricas:**
  - **EsHoja (B):** Indica si el nodo es una situación terminal, o si estamos en el nivel máximo.
  - **Utilidad (B, modo):** Devuelve el valor de la función de utilidad para el tablero **B** en el **modo** indicado.
  - **para cada hijo C del tablero B:** Iterador para generar todos los movimientos a partir de una situación de partida **B**.

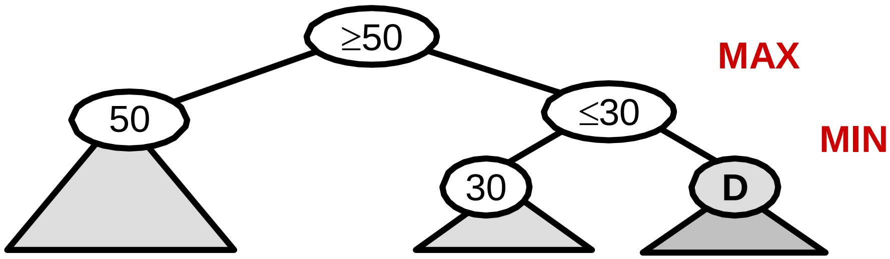
- **NOTA:** Faltaría devolver también el movimiento óptimo.

- **Ejemplo.** El juego de los palillos.

# 5. Algoritmos Bactracking. Ejemplos de aplicación. **3_Resolución de juegos.**

- Sobre los árboles de juegos se puede aplicar un tipo propio de poda, conocida como poda **alfa-beta**.
- **Poda Alfa:**

  Supongamos que en cierto momento de la evaluación llegamos a la siguiente situación.

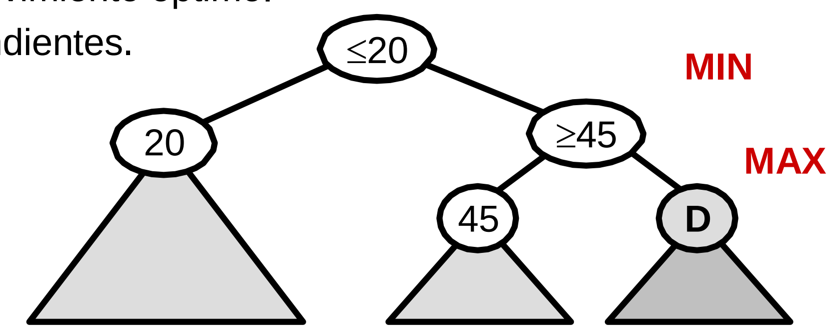



  - Haya lo que haya en **D**, nunca estará el movimiento óptimo.
  - **Conclusión:** podar el nodo **D** y sus descendientes.

- **Poda Beta:**

  Supongamos que en cierto momento de la evaluación llegamos a la siguiente situación.

  - Haya lo que haya en **D**, nunca estará el movimiento óptimo.
  - **Conclusión:** podar el nodo **D** y sus descendientes.



- **Implementación estrategia minimax, con poda alfa-beta.**

**BuscaMinimax (B: TipoTablero; valorPadre: real; modo: (MAX, MIN)) : real**

```
    si EsHoja(B) entonces
        devolver Utilidad (B)
    sino
        si modo == MAX entonces valoract:= -∞
        sino valoract:= ∞
        para cada hijo C del tablero B hacer
            si modo == MAX entonces
                valoract:= max (valoract, BuscaMinimax(C, valoract, MIN))
{ P. beta}      si valoract ≥ valorPadre entonces salir del para
            sino
                valoract:= min (valoract, BuscaMinimax(C, valoract, MAX))
{ P. alfa}      si valoract ≤ valorPadre entonces salir del para
            finsi
        finpara
        devolver valoract
    finsi
```

# 5. Algoritmos Bactracking. Conclusiones.

- **Backtracking:** Recorrido exhaustivo y sistemático en un árbol de soluciones.
- **Pasos para aplicarlo:**
  - Decidir la forma del árbol.
  - Establecer el esquema del algoritmo.
  - Diseñar las funciones genéricas del esquema.
- Relativamente fácil diseñar algoritmos que encuentren soluciones óptimas pero los algoritmos de backtracking son muy ineficientes.
- **Mejoras:** mejorar los mecanismos de poda, incluir otros tipos de recorridos (no solo en profundidad)

  $\Rightarrow$ Técnica de **Ramificación y Poda**.