Parte de Algoritmos
de la asignatura de Programación
Master de Bioinformática

Búsqueda exhaustiva

Web asignatura: http://dis.um.es/~domingo/algbio.html

E-mail profesor: domingo@um.es

Transparencias preparadas a partir de las del curso de Algoritmos y Estructuras de Datos II, del Grado de Ingeniería Informática y An Introduction to Bioinformatics Algorithms

Método general

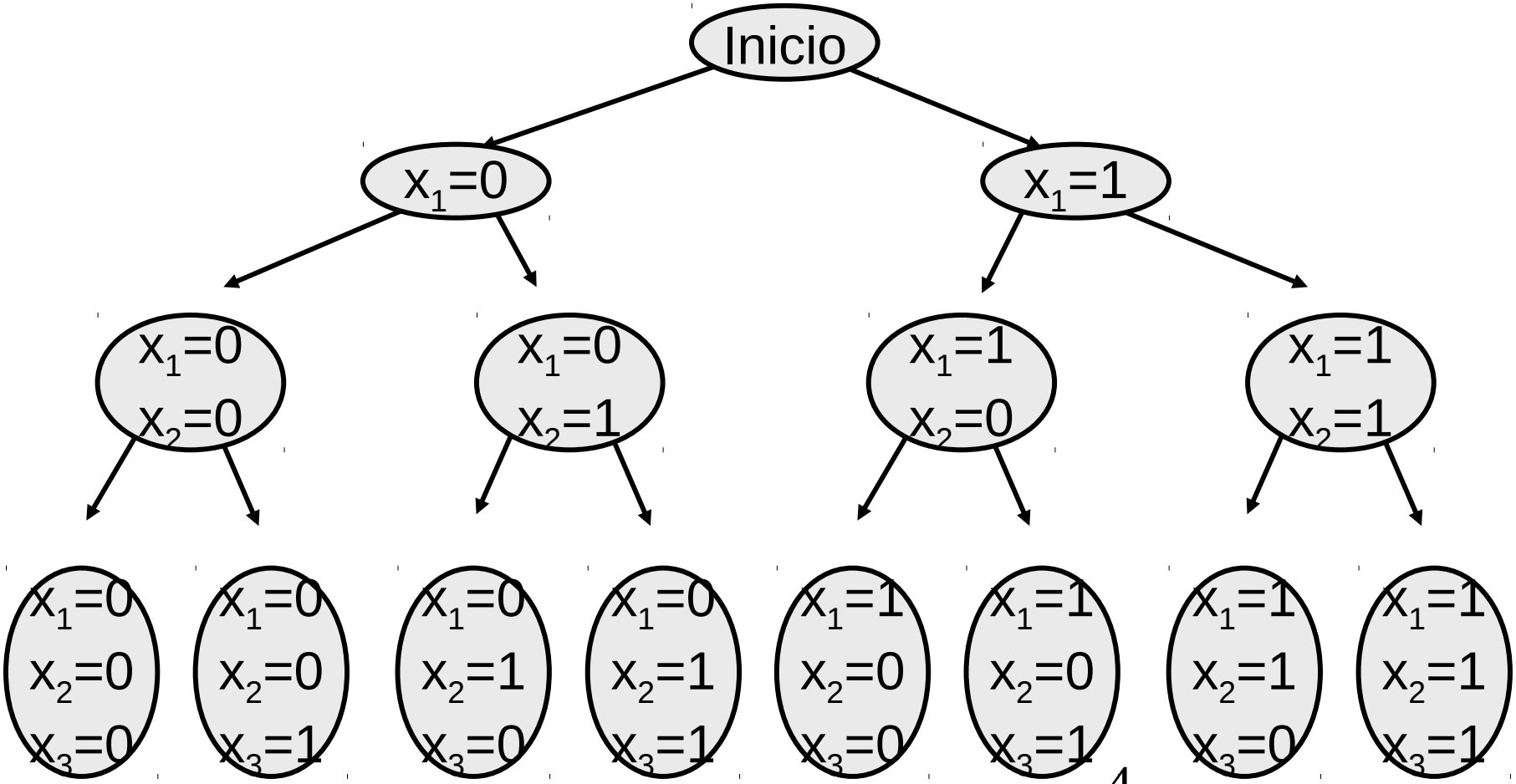
- La **búsqueda exhaustiva** es una técnica general de resolución de problemas.
- Se realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele resultar ineficiente.
- La búsqueda se suele realizar recorriendo un *árbol* con el que se representan las posibles soluciones.
- Hay algunos métodos de recorrido del árbol:
  - **Recorrido en anchura**.
  - **Backtracking**.
  - **Branch and Bound** (ramificación y acotación).

Método general

- La **solución** de un problema se puede expresar como una tupla (x_1, x_2, ..., x_n), satisfaciendo unas restricciones $P(x_1, x_2, ..., x_n)$ y tal vez optimizando una cierta función objetivo.
- En cada momento, el algoritmo se encontrará en un cierto nivel **k**, con una solución parcial (x_1, ..., x_k).
- Cada conjunto de posibles valores de la tupla representa un nodo del árbol de soluciones.
- Se sigue hasta que la solución parcial sea una solución completa del problema, o hasta que no queden más posibilidades por probar.

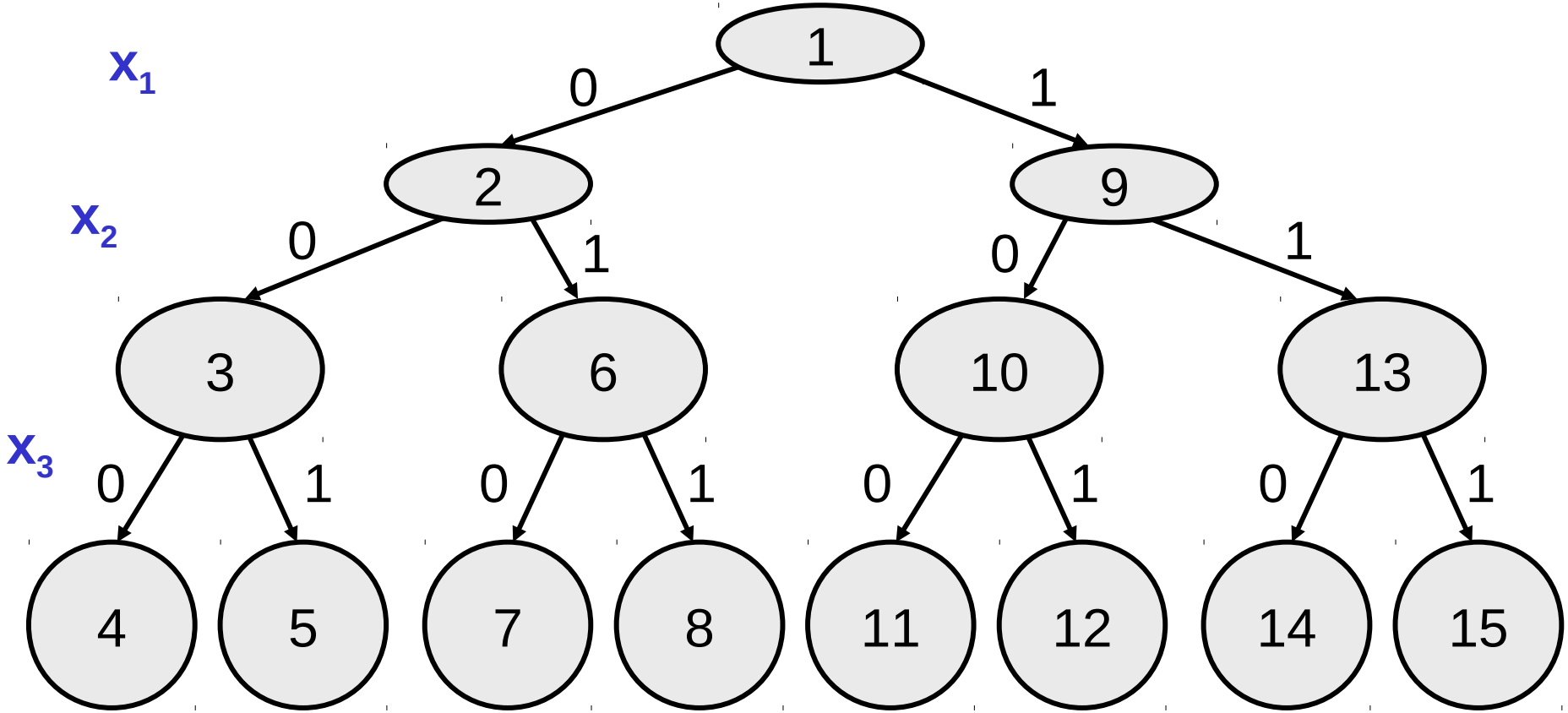
Método general

- Se recorre un árbol de soluciones. Sin embargo, este árbol es **implícito**, no se almacena en ningún lugar.



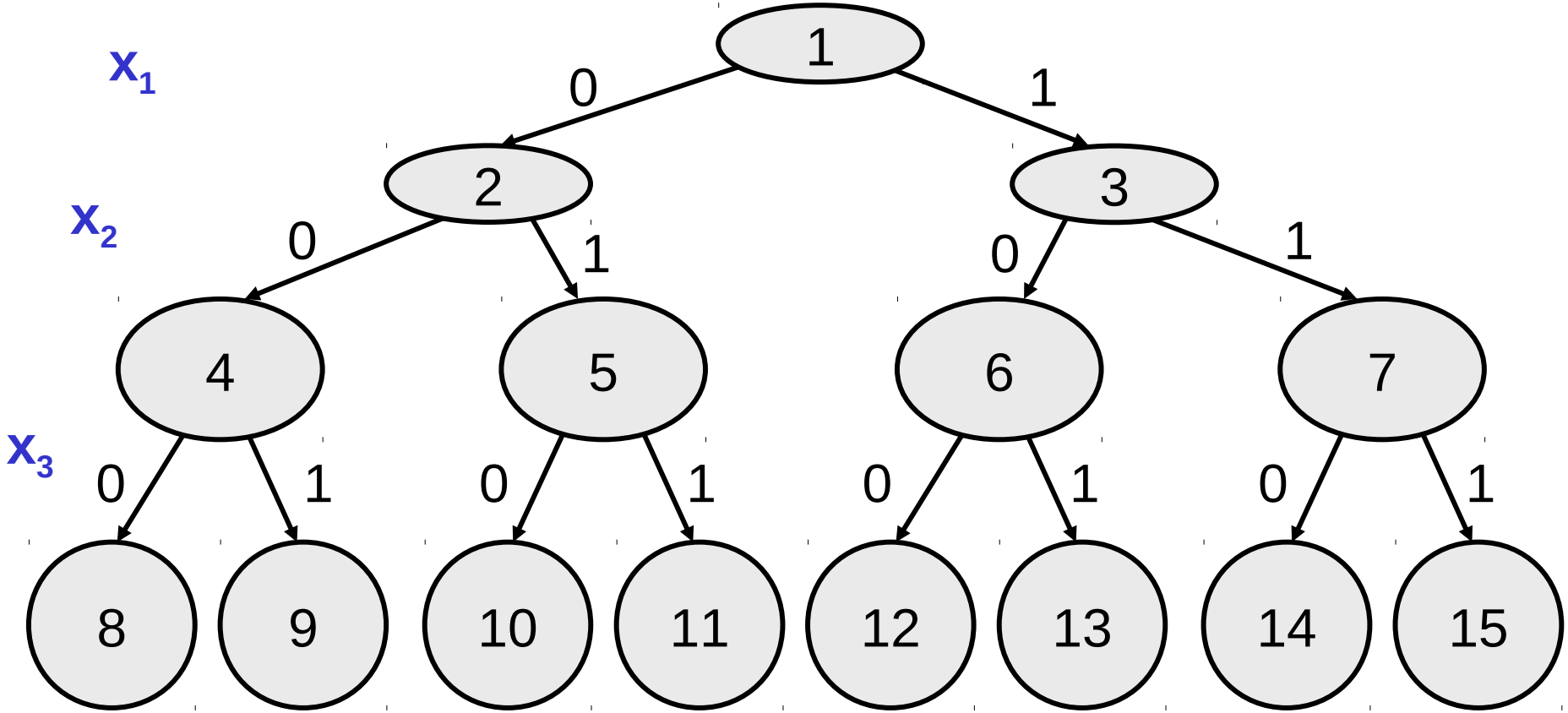
Método general

- El recorrido se hace en un cierto orden. Por ejemplo, con backtracking se hace en profundidad:



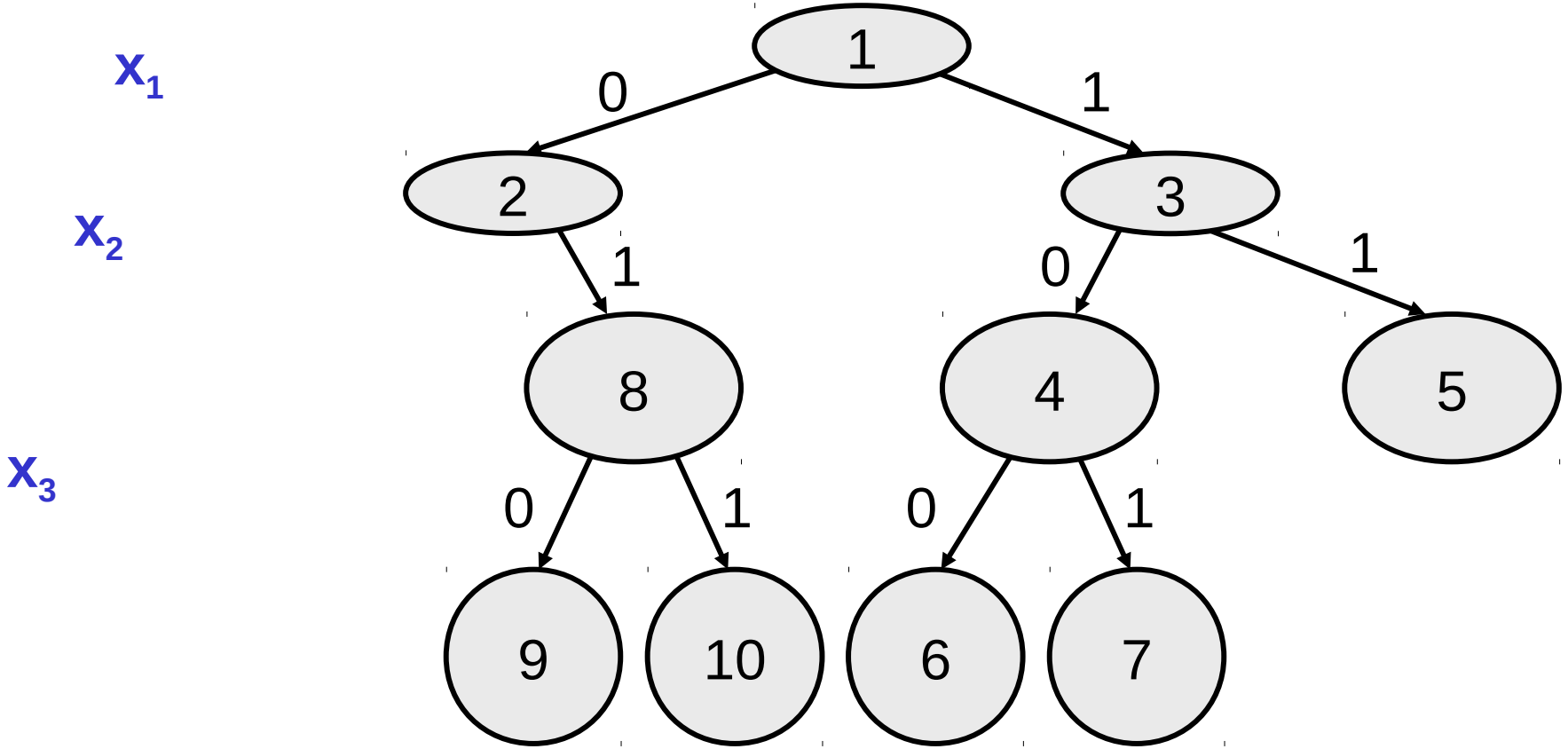
Método general

- Y en anchura:



Método general

- Con Branch and Bound se guía la búsqueda por algún criterio, y se intenta eliminar nodos:



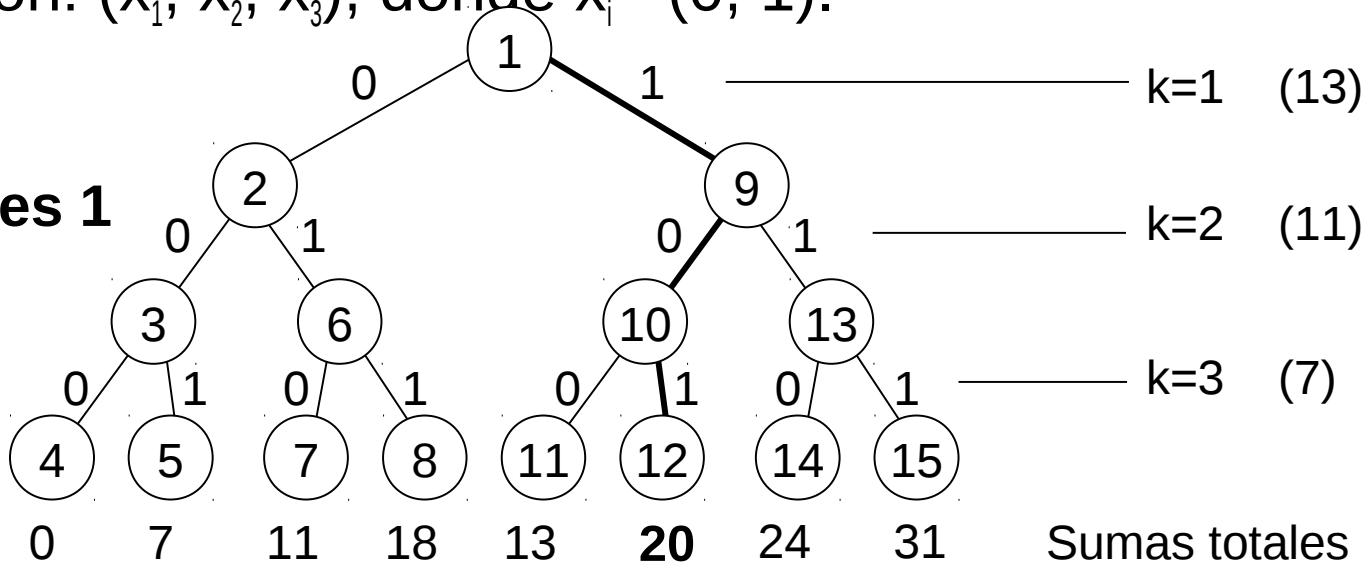
Ejemplo: suma de números

- **Ejemplo.** Dado un conjunto de números enteros {13, 11, 7}, encontrar si existe algún subconjunto cuya suma sea exactamente 20.
  - La primera decisión: ¿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?

Ejemplo: suma de números

- **Posibilidad 1)** Árbol binario: En cada nivel **i** decidir si el elemento **i** está o no en la solución. Representación de la solución: (x_1, x_2, x_3), donde x_i= (0, 1).

Árbol de soluciones 1

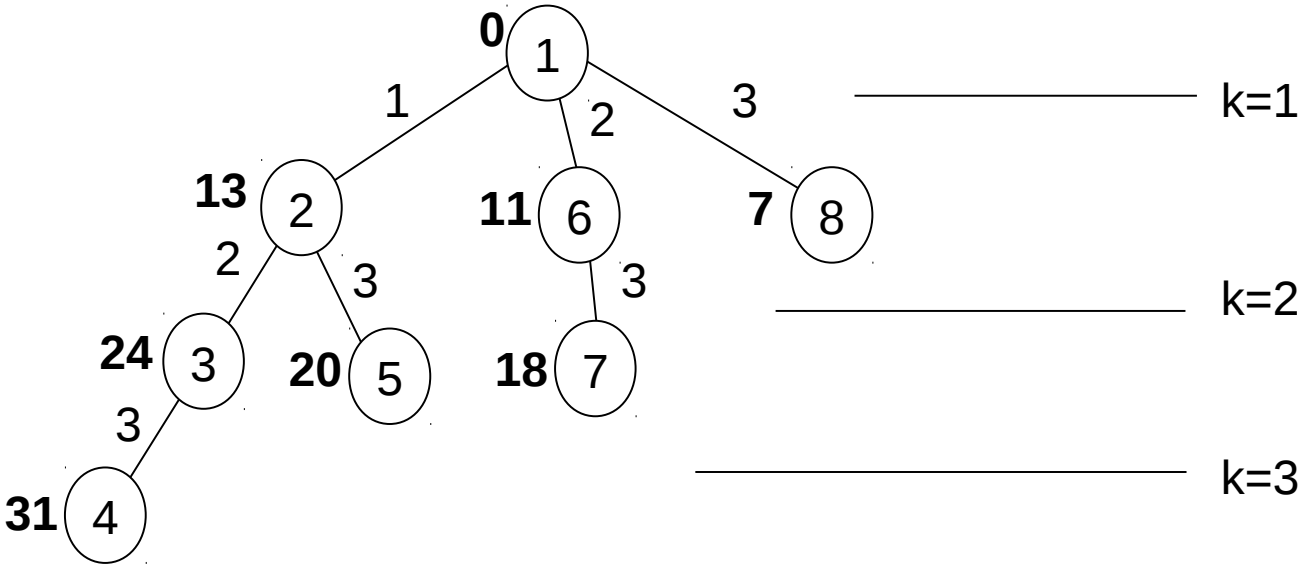


- Cada nodo representa un paso del algoritmo, una solución parcial en cada momento dado. El árbol indica un orden de ejecución (recorrido en profundidad) pero no se almacena en ningún lugar.
- Una solución es un nodo hoja con valor de suma 20.
- **Posible mejora**: En cada nodo llevamos el valor de la suma hasta ese punto. Si el valor es mayor que 20: retroceder al nivel anterior.

Ejemplo: suma de números

- **Posibilidad 2)** Árbol combinatorio: En cada nivel **i** decidir qué elemento se añade (1, 2 o 3). Representación de la solución (s_1, ..., s_m), donde $m \leq n$ y $s_i \in \{1, 2, 3\}$.

Árbol de soluciones 2



- Cada nodo es una posible solución. Será válida si la suma es 20.
- El recorrido es también en profundidad.
- Necesitamos funciones para generar los nodos, para descartar nodos y para saber si un nodo es solución.
- La eficiencia del algoritmo, depende del número de nodos, por lo que sería conveniente tener algún criterio para eliminar nodos.

Ejemplo: suma de números

El programa numeros.pl resuelve el problema de la suma de números.

$\rightarrow$ Cada una de las soluciones que se genera ¿a qué nodos de los árboles anteriores corresponden?

$\rightarrow$ ¿Qué coste teórico del tiempo de ejecución tiene? Comparar el tiempo de ejecución teórico con el experimental.

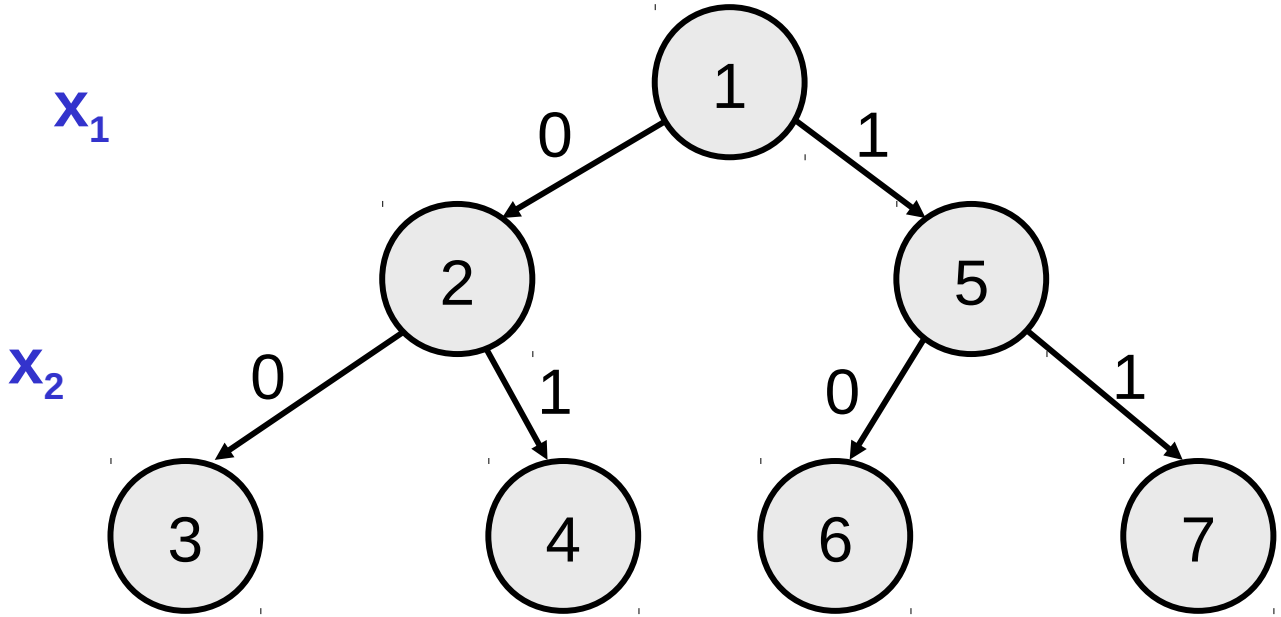
$\rightarrow$ ¿Qué posibilidades hay para reducir el tiempo de ejecución?

Tipos de árboles

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

Tipos de árboles

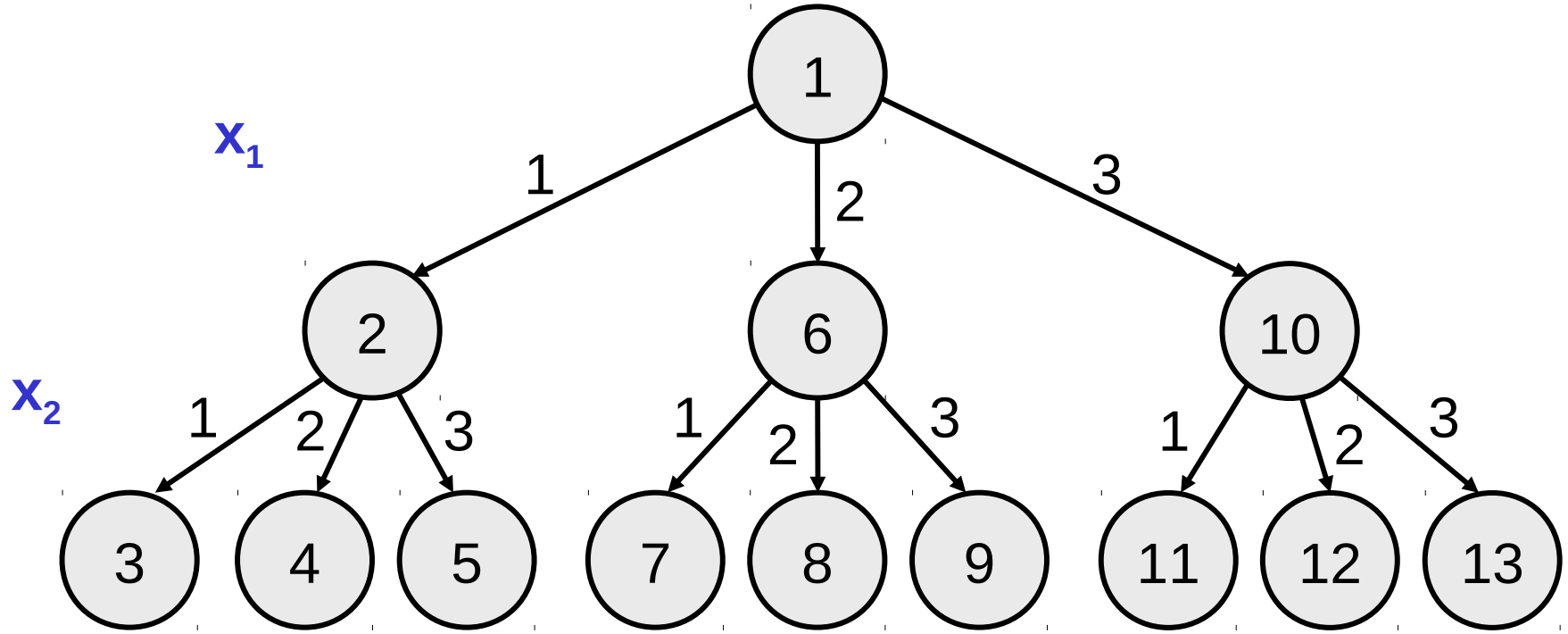
- **Árboles binarios: s**= (x_1, x_2, ..., x_n), con $x_i \in \{0, 1\}$



- **Tipo de problemas:** elegir ciertos elementos de entre un conjunto, sin importar el orden de los elementos.

Tipos de árboles

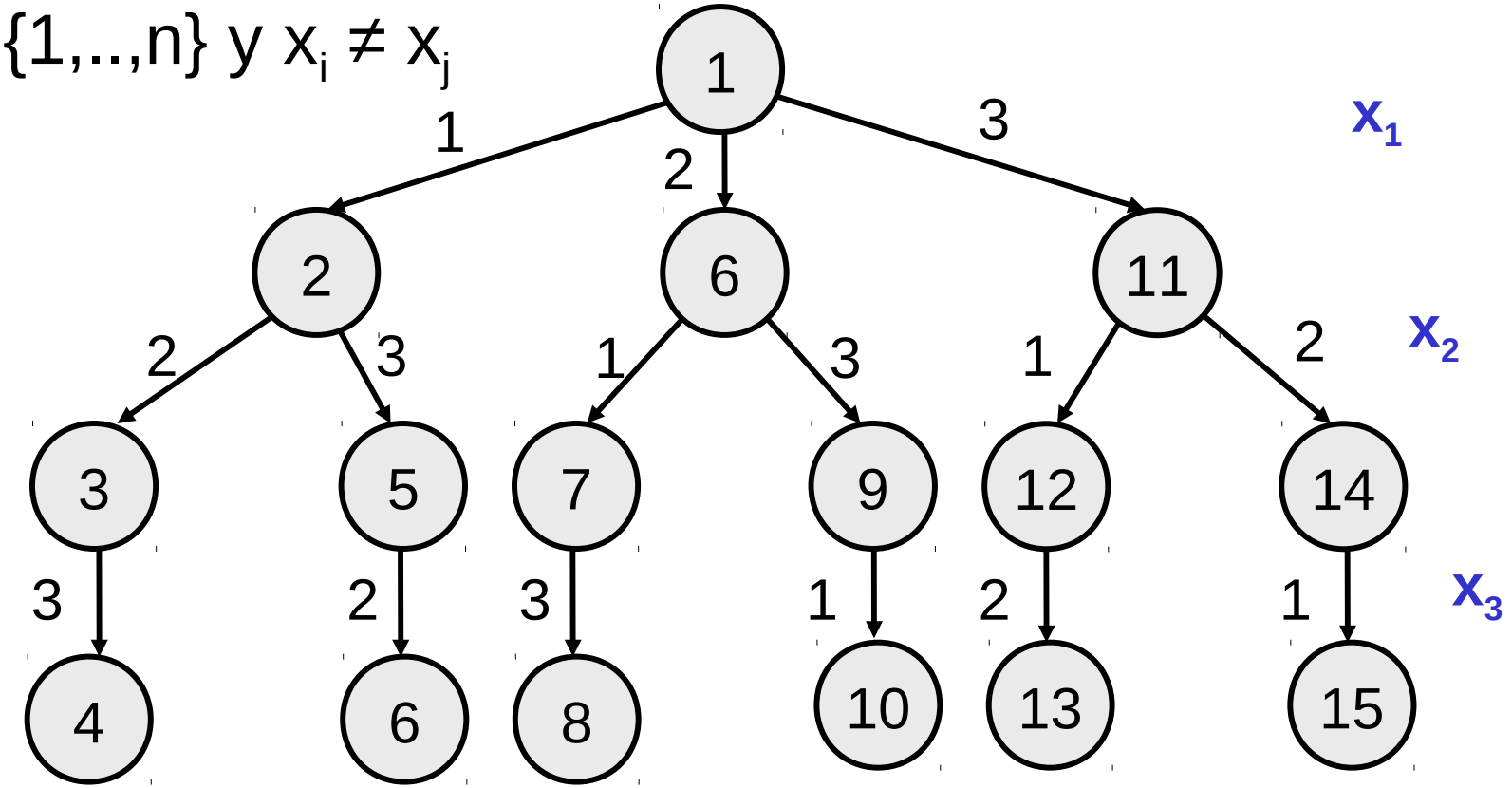
- **Árboles k-arios: s**= (x_1, x_2, ..., x_n), con $x_i \in$ {1,..,k}



- **Tipo de problemas:** varias opciones para cada x_i.

Tipos de árboles

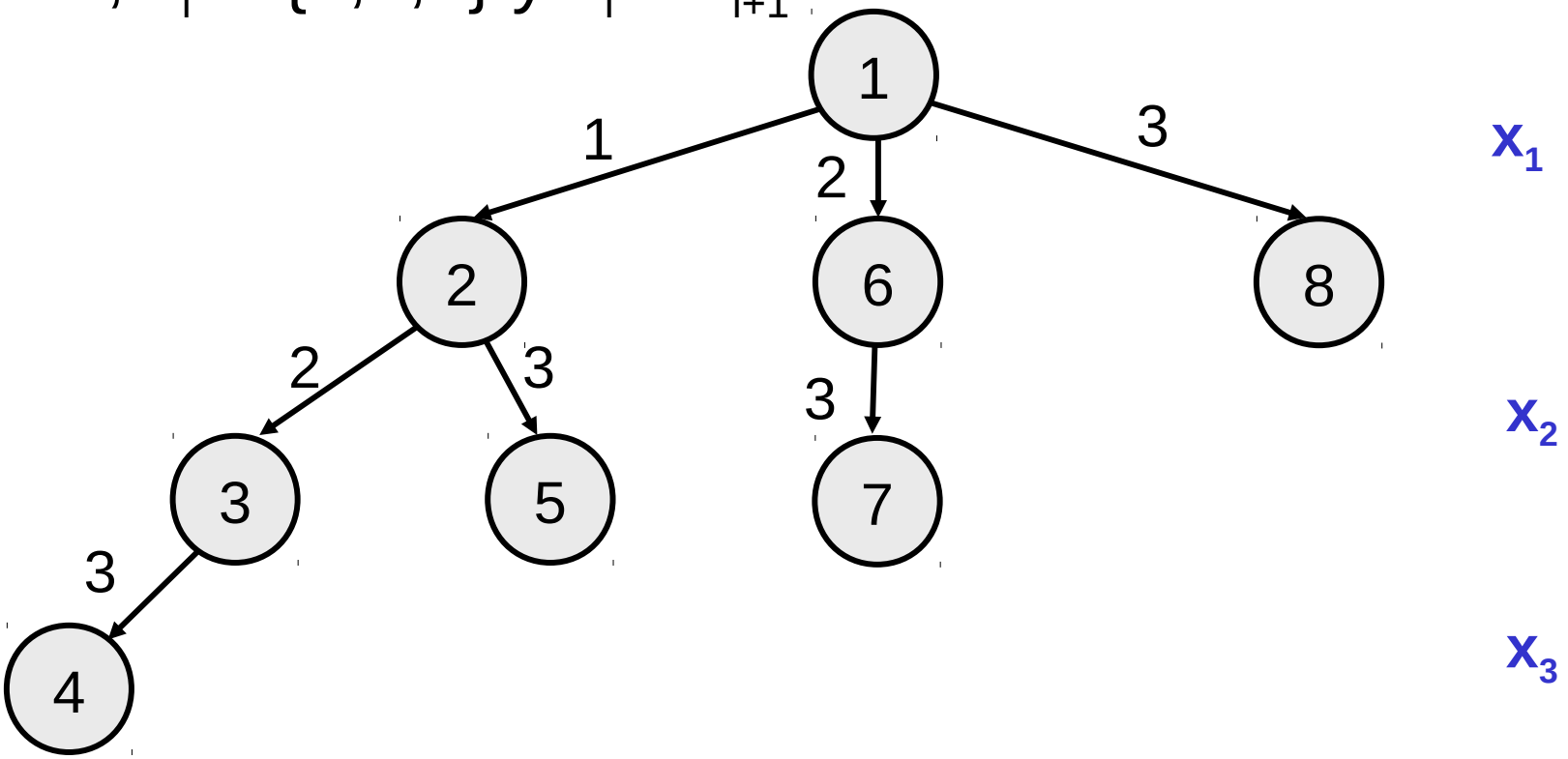
- **Árboles permutacionales: s**= (x_1, x_2, ..., x_n), con $x_i \in \{1,..,n\}$ y $x_i \neq x_j$



- **Tipo de problemas:** los x_i no se pueden repetir.

Tipos de árboles

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



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

Cuestiones a tener en cuenta

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? ¿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? (uso de variables auxiliares, que se modifican al moverse por el árbol)

  Poda por restricciones del problema.

  Poda según el criterio de la función objetivo.

Backtracking

- **Esquema general**. Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad, y se supone que existe alguna.

Backtracking (var s: TuplaSolución)

```
nivel = 1
s = s_INICIAL
fin = false
repetir
    Generar (nivel, s)
    si Solución (nivel, s) entonces
        fin = true
    sino si Criterio (nivel, s) entonces
        nivel = nivel + 1
    sino mientras NOT MasHermanos (nivel, s) hacer
        Retroceder (nivel, s)
hasta fin
```

Backtracking

- **Variables**:
  - **s**: almacena la solución parcial hasta cierto punto.
  - $\mathbf{s}_{\mathbf{INICIAL}}$: valor de inicialización.
  - **nivel**: indica el nivel actual en el que se encuentra el algoritmo.
  - **fin**: valdrá **true** cuando hayamos encontrado alguna solución.
  - Uso de variables auxiliares para eliminación de nodos o evitar cálculos, por ejemplo, suele ser común utilizar variables temporales con el valor actual (beneficio, peso, etc.) de la tupla solución.

Backtracking

- **Funciones**:
  - **Generar (nivel, s)**: genera el siguiente hermano, o el primero, para el **nivel** actual.
  - **Solución (nivel, s)**: comprueba si la tupla (s[1], ..., s[nivel]) es una solución válida para el problema.
  - **Criterio (nivel, s)**: comprueba si a partir de (s[1], ..., s[nivel]) se puede alcanzar una solución válida. En otro caso se rechazarán todos los descendientes (**poda**).
  - **MasHermanos (nivel, s)**: devuelve **true** si hay más hermanos del nodo actual que todavía no han sido generados.
  - **Retroceder (nivel, s)**: retrocede un nivel en el árbol de soluciones. Disminuye en 1 el valor de **nivel**, y posiblemente tendrá que actualizar la solución actual, quitando los elementos retrocedidos.

Backtracking: suma de números

- **Ejemplo,** problema de subconjunto de números que suma un valor 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 , valor de inicialización (número i-ésimo no estudiado)
  - $\mathbf{s}_{\mathbf{INICIAL}}$: (-1, -1, ..., -1)
  - **fin**: valdrá **true** cuando se haya encontrado solución.
  - **suma:** suma acumulada hasta ahora (inicialmente 0).

Backtracking: suma de números

Funciones:

- **Generar (nivel, s)**

  s[nivel] += 1

  si s[nivel]==1 **entonces** suma += num[nivel]

- **Solución (nivel, s)**

  devolver (nivel==n) Y (suma==P)

- **Criterio (nivel, s)**

  devolver (nivel<n) Y (suma≤P)

- **MasHermanos (nivel, s)**

  devolver s[nivel] < 1

- **Retroceder (nivel, s)**

  suma -= num[nivel]*s[nivel]

  s[nivel] = -1

  nivel -= 1

Backtracking: variaciones

Variaciones del esquema general:

1) ¿Y si no es seguro que exista una solución?

2) ¿Y si queremos almacenar todas las soluciones (no sólo una)?

3) ¿Y si el problema es de optimización (maximizar o minimizar)?

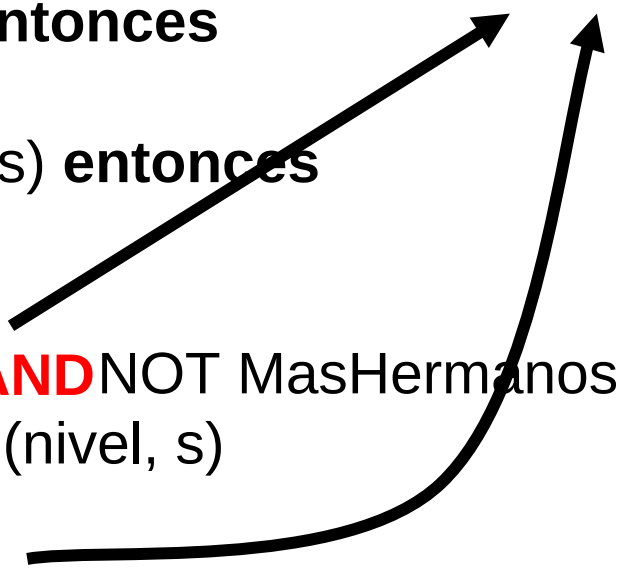
Backtracking: variaciones

- Si puede no existir solución:

Backtracking (var s: TuplaSolución)

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

Para poder generar todo el árbol de backtracking



Backtracking: variaciones

- Cuando queremos almacenar todas las soluciones:

Backtracking (var s: TuplaSolución)

```
nivel = 1
s = s_INICIAL
fin = false
repetir
    Generar (nivel, s)
    si Solución (nivel, s) entonces
        Almacenar (nivel, s)
     si Criterio (nivel, s) entonces
       nivel := nivel + 1
    sino
        mientras (nivel>0) AND  NOT MasHermanos (nivel, s)
         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

Backtracking: variaciones

- Problema de optimización (maximización):

Backtracking (var s: TuplaSolución)

nivel = 1

s = $s_{INICIAL}$

voa = -infinito; soa= Ø

repetir

Generar (nivel, s)

si Solución (nivel. s) **AND Valor(s) > voa entonces**

voa = Valor(s); soa = s

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

nivel = nivel + 1

sino

mientras **(nivel>0) AND** NOT MasHermanos (nivel, s)

hacer Retroceder (nivel, s)

finsi

hasta **nivel==0**

voa: valor óptimo actual

soa: solución óptima actual

Cuestiones

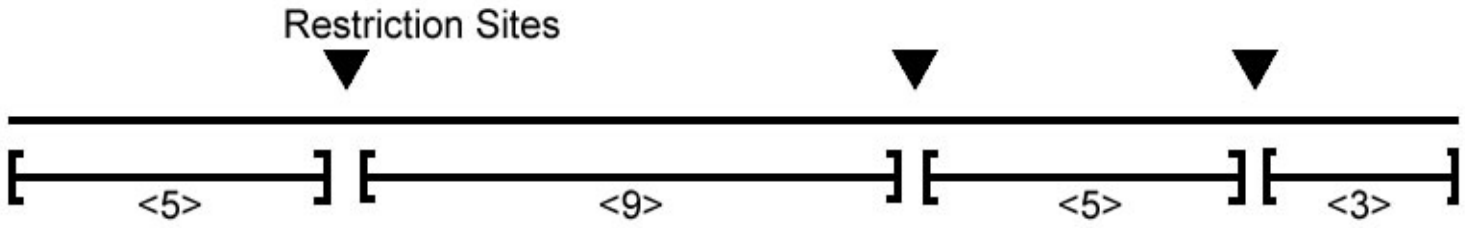
$\rightarrow$ Modificar el programa numeros.pl para que obtenga, de todos los subconjuntos que suman la cantidad dada, el que esté formado por menos elementos.

$\rightarrow$ Hacer un programa que resuelva esta versión de optimización del problema de la suma de números siguiendo un esquema de backtracking. Comparar experimentalmente el tiempo de ejecución de las dos versiones, justificando teóricamente las diferencias.

Problema del mapa de restricciones

En An Introduction to Bioinformatics Algorithms, capítulo 4.
Ver ahí el significado biológico.

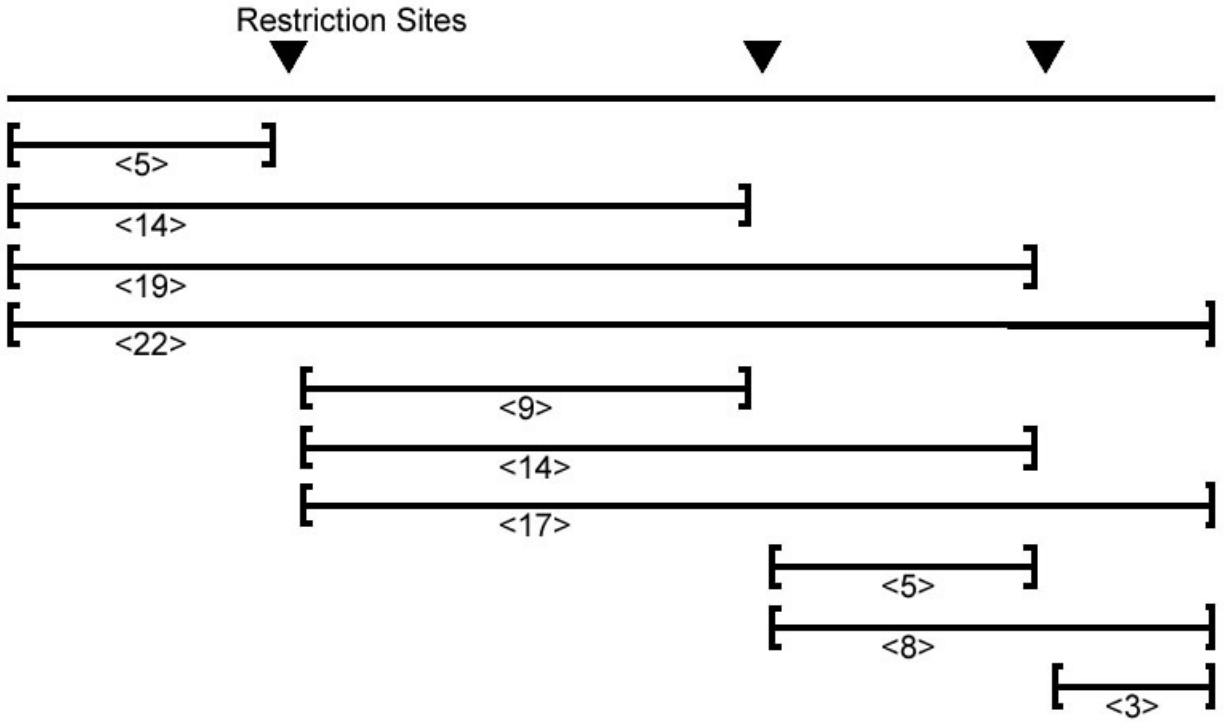
- Las “encimas de restricción” reconocen secuencias en ADN y cortan en esa secuencia, creando múltiples fragmentos:



Problema: reconstruir el orden de los fragmentos a partir de los tamaños: {3,5,5,9}

Problema del mapa de restricciones

- Por ejemplo, con tres cortes obtenemos un conjunto de 10 longitudes:



Problema del mapa de restricciones

Datos

n: Número total de cortes

X: Conjunto de ***n*** enteros que representan la posición de los cortes, incluyendo el inicio y el final

D*X*: Conjunto de enteros que representan las longitudes de los fragmentos producidos por los cortes

Problema del mapa de restricciones

X	0	2	4	7	10
0		2	4	7	10
2			2	5	8
4				3	6
7					3
10					

DX = {2, 2, 3, 3, 4, 5, 6, 7, 8, 10} se representa como una tabla bidimensional, con elementos de ***X*** = {0, 2, 4, 7, 10}, con elemento (*i*, *j*) con valor $x_j - x_i$, para $1 \leq i < j \leq n$.

Problema del mapa de restricciones

FORMULACIÓN

Dadas todas las distancias entre pares de puntos en una línea, se quiere obtener las posiciones de dichos puntos.

- Entrada: El conjunto de pares de distancias, L, que contiene $n(n-1)/2$ enteros
- Salida: Un conjunto X, de n enteros, tal que $DX = L$

Problema del mapa de restricciones

- No siempre es posible reconstruir $\boldsymbol{X}$ de forma única a partir de D$\boldsymbol{X}$:
- Por ejemplo:

  $X = \{0, 2, 5\}$ y $(X + 10) = \{10, 12, 15\}$

  producen D$\boldsymbol{X}$=$\{2, 3, 5\}$
- Y también $\{0,1,2,5,7,9,12\}$ y $\{0,1,5,7,8,10,12\}$ producen

  $\{1, 1, 2, 2, 2, 3, 3, 4, 4, 5, 5, 5, 6, 7, 7, 7, 8, 9, 10, 11, 12\}$

Problema del mapa de restricciones

- Obtener el fragmento de mayor longitud, ***M***, que será la longitud de la secuencia de ADN.
- Para cada posible conjunto

$$\boldsymbol{X}=\{0, x_2, \ldots, x_{n-1}, M\}$$

obtener D***X***

- Si D***X*** = ***L***, entonces ***X*** es el mapa de restricciones

Problema del mapa de restricciones

Se puede mejorar tomando subconjuntos de L:

- Obtener el fragmento de mayor longitud, M, que será la longitud de la secuencia de ADN.
- Para cada posible conjunto

$$X=\{0, x_2, \ldots, x_{n-1}, M\}$$

con x_i en L, obtener DX

- Si DX = L, entonces X es el mapa de restricciones

Problema del mapa de restricciones

$\rightarrow$ ¿Qué coste tendrán los dos algoritmos?

$\rightarrow$ Hacer un programa para resolver el problema con el algoritmo más eficiente.

$\rightarrow$ Estudiar el tiempo de ejecución experimental.

Otros problemas

→ En An Introduction to Bioinformatics Algorithms, capítulo 4, se puede consultar el problema de búsqueda de Motif. Explicarlo.