Análisis y diseño de algoritmos

Semana 11

Logro de la sesión

Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos con la técnica BackTracking

Backtracking – Un ejemplo sencillo

Deseamos sentar a tres personas en tres sillas, ¿cuántas formas existen para ubicar a las personas en las 3 sillas?

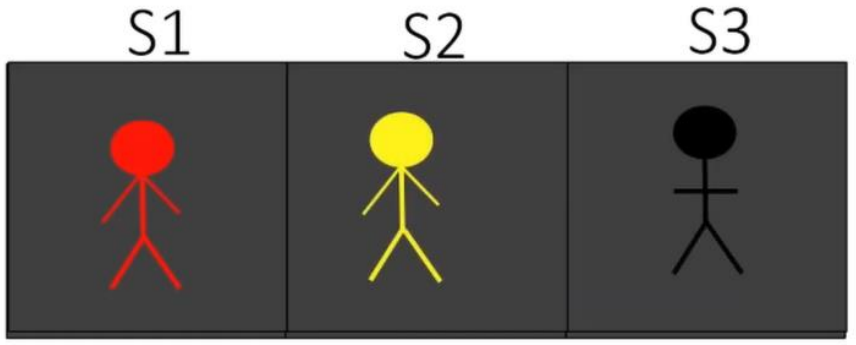
Este es un ejercicio de permutaciones:

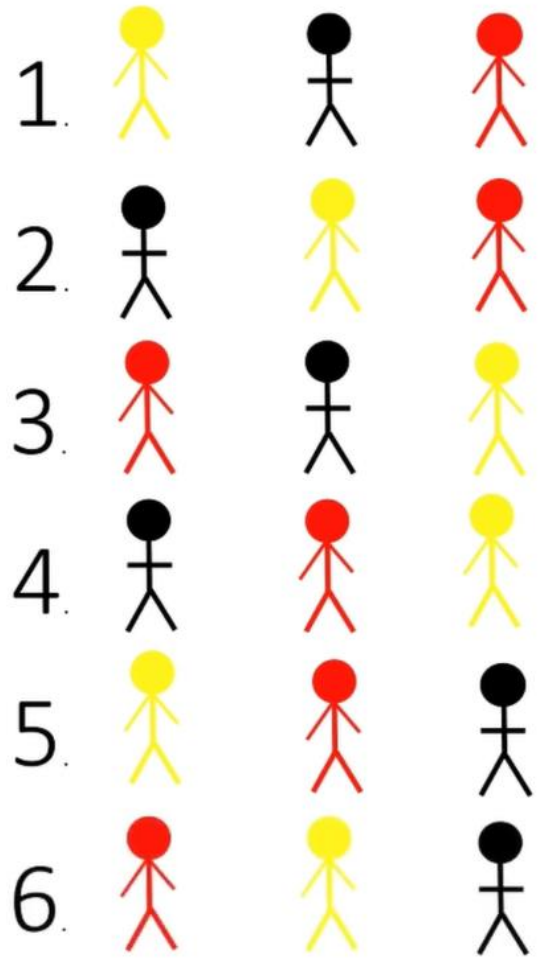
Permutaciones
Cuando elegimos k de n objetos y el orden importa, el número de permutaciones es

$$n(n-1)(n-2)\dots(n-k+1)$$

La solución es (n=k=3): 3! = 3*2*1 = 6

Si lo hacemos con un algoritmo de "fuerza bruta", representaríamos todas las combinaciones posibles y llegaríamos a la respuesta.



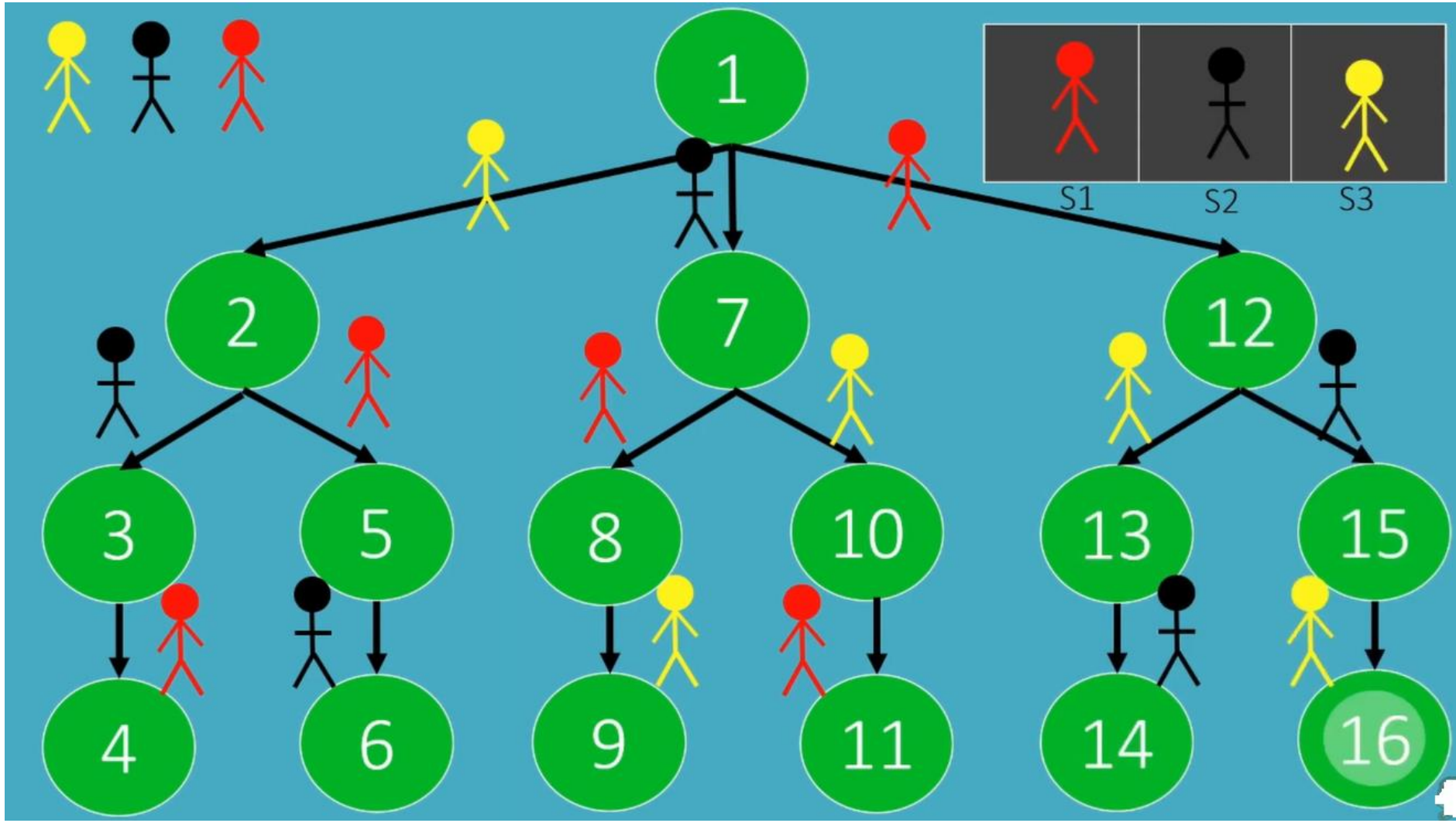


Backtracking – Un ejemplo sencillo

¿Y si lo hacemos con un algoritmo de “Backtracking”’?: En este caso, utilizamos un árbol y hacemos búsqueda en profundidad

El recorrido desde un nodo raíz hasta un nodo hoja sería una solución

En este caso el algoritmo de fuerza bruta y el algoritmo backtracking no tienen ninguna diferencia

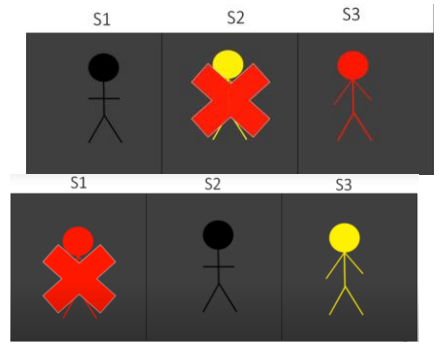


Backtracking – Un ejemplo sencillo

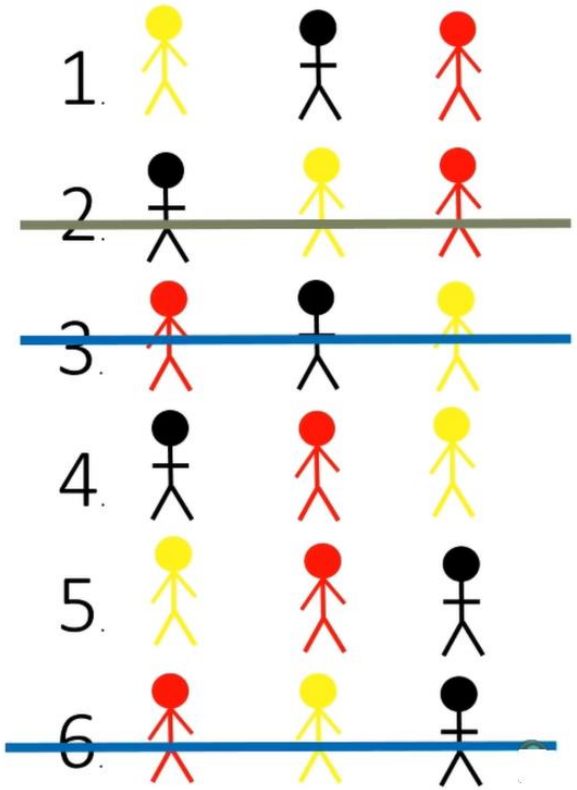
Pero ahora consideremos las siguientes restricciones:

La persona amarilla no puede sentarse en el asiento 2

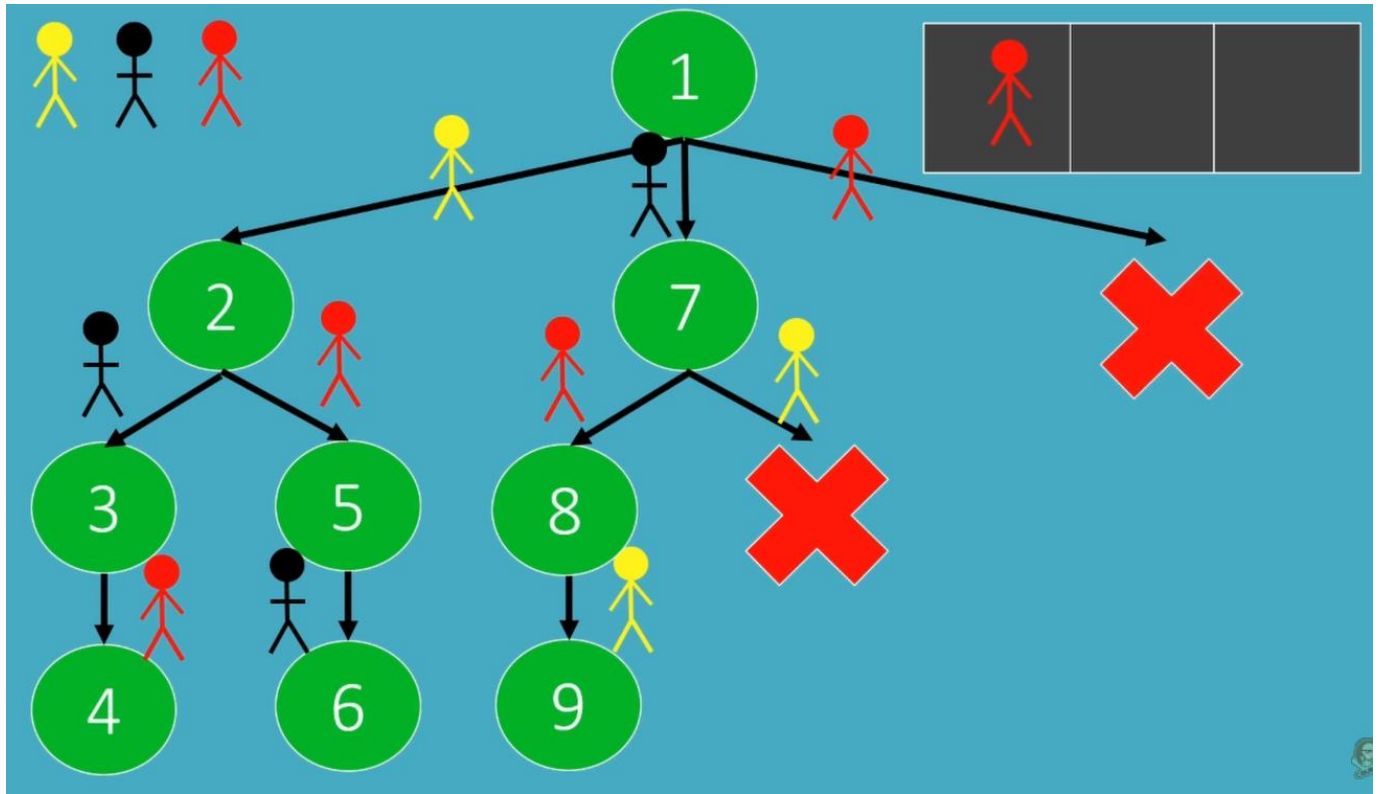
La persona roja no se puede sentar en el primer asiento



Con algoritmo "fuerza bruta"



Con algoritmo "Backtracking"



Búsqueda Exhaustiva: Introducción

- Muchos problemas de optimización sólo pueden resolverse de manera exacta explorando todas las posibles combinaciones de elementos y tratando de encontrar aquellas que sean solución, eligiendo entre ellas una cualquiera o aquella que optimice una determinada función objetivo.
- Ciertos problemas admiten estrategias de selección heurística de elementos que conducen a soluciones óptimas. Es lo que hacen los algoritmos voraces.
- Pero las estrategias voraces conducen, en la mayor parte de los casos, a soluciones subóptimas. Recuérdese el caso del problema de la mochila (discreto), o el problema del viajante.
- Aparece entonces la necesidad de **volver atrás**, es decir, deshacer decisiones previas que se ha demostrado que no conducían a una solución (por el contrario, los algoritmos voraces nunca deshacen una decisión previamente tomada).

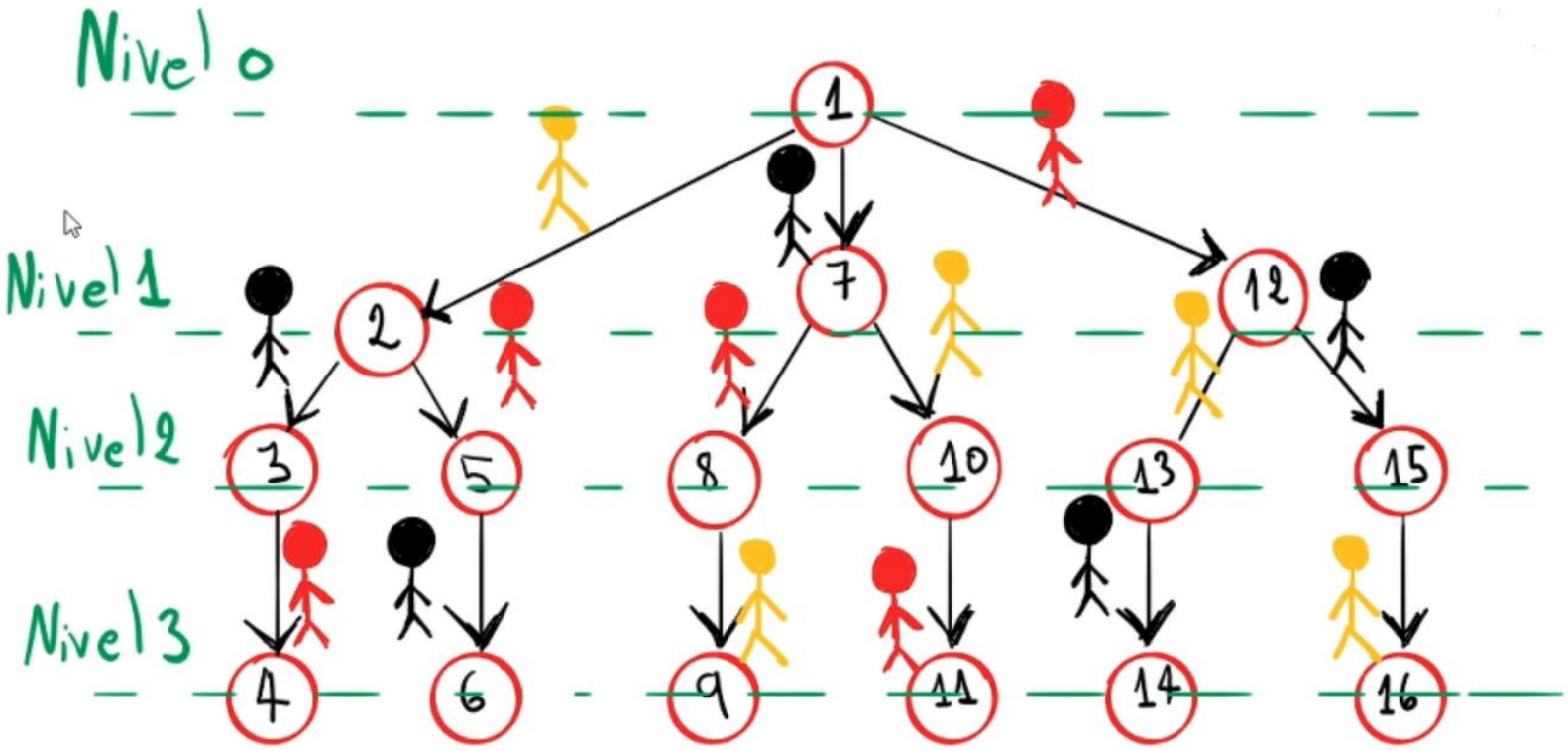
Búsqueda Exhaustiva: Introducción

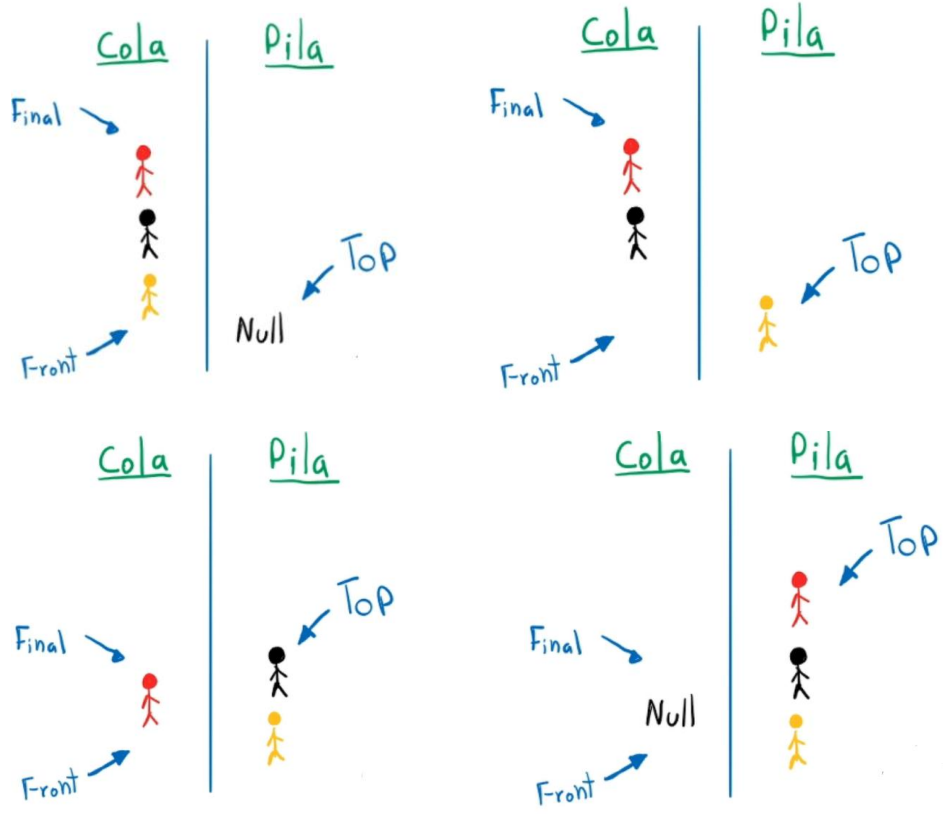
- Los algoritmos que exploran todas las posibles combinaciones de elementos se conocen como **algoritmos de fuerza bruta** o algoritmos de búsqueda exhaustiva.
- A pesar de su nombre, estos algoritmos raramente exploran todas las posibles combinaciones de elementos, sino que expanden las combinaciones más prometedoras, de acuerdo a un cierto criterio relacionado con la función objetivo que se pretende optimizar.
- La técnica de búsqueda más sencilla, conocida como **backtracking** o **vuelta atrás**, opera de manera recursiva, explorando en profundidad el árbol virtual de combinaciones de elementos y evaluando cada nuevo nodo para comprobar si es terminal (es decir, si es una posible solución).
- En ciertos casos, si un nodo terminal es solución, el algoritmo termina, retornando la combinación de elementos correspondiente a dicho nodo.
- En otros casos, se busca **la mejor solución**, de modo que el algoritmo evalúa la función objetivo en el nodo terminal, comprueba si es mejor que el óptimo almacenado hasta la fecha y sigue buscando.

Backtracking – Un ejemplo sencillo

Vamos a escribir el código, consideraremos que al ir del nodo "x" al nodo "y", se sienta una persona, esta acción se repite una y otra vez, así que hablamos de una función recursiva. La función recursiva terminaría cuando llegamos a un nodo hoja

También debemos considerar que podemos movernos de un nodo a otro no sólo hacia abajo sino a un costado en el mismo nivel en el que nos encontramos, como en el caso de cuando vamos del nodo 2 al nodo 3 o del nodo 2 al nodo 5 (en el nivel 2), esto lo implementaremos con una estructura repetitiva.

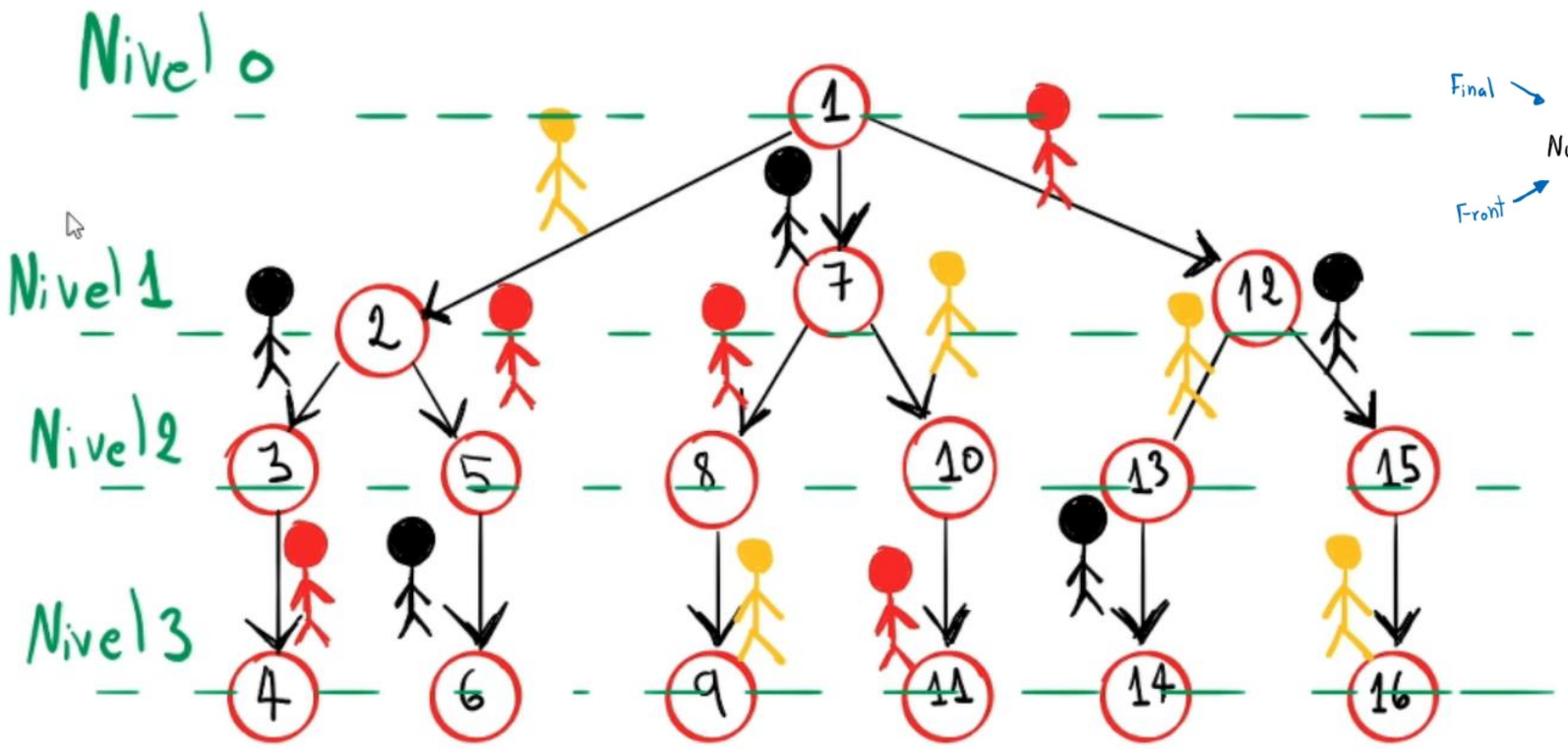


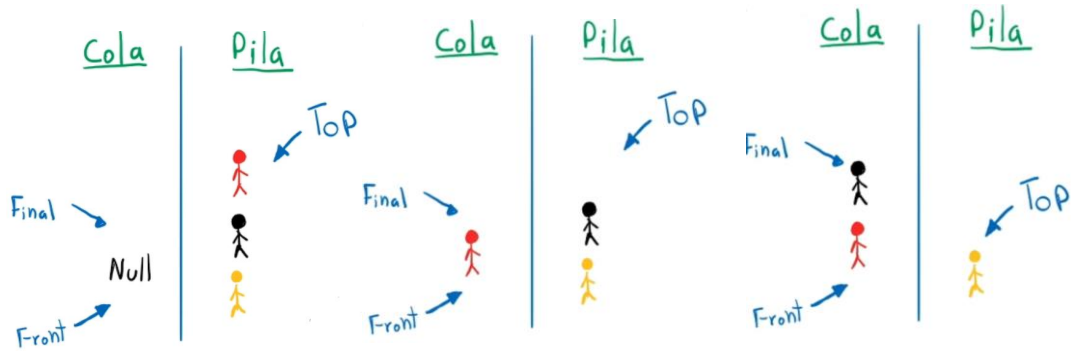


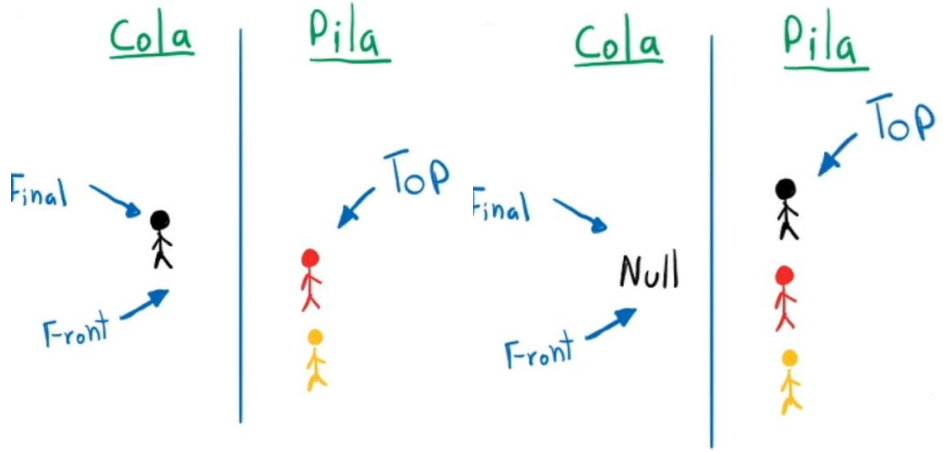
Backtracking – Un ejemplo sencillo

Vamos a escribir el código, consideraremos que al ir del nodo "x" al nodo "y", se sienta una persona, esta acción se repite una y otra vez, así que hablamos de una función recursiva. La función recursiva terminaría cuando llegamos a un nodo hoja

También debemos considerar que podemos movernos de un nodo a otro no sólo hacia abajo sino a un costado en el mismo nivel en el que nos encontramos, como en el caso de cuando vamos del nodo 2 al nodo 3 o del nodo 2 al nodo 5 (en el nivel 2), esto lo implementaremos con una estructura repetitiva.







Backtracking – Un ejemplo sencillo

```
#include<iostream>
#include<queue>
#include<stack>
using namespace std;
int cont=0;
void mostrarPila(stack <char> s) {
  stack<char> aux;
  while (!s.empty()){
    cout << '\t' << s.top();
    aux.push(s.top());
    s.pop();
  }
  while (!aux.empty()){
    s.push(aux.top());
    aux.pop();
  }
  cout << '\n';
}
void back(int i,int n,stack<char> pila,queue<char> cola){
  //verificamos si ya hay tres personas sentadas
  if(i==n){
    cont++;
    cout<<cont<<". ";
    mostrarPila(pila);
    //imprimir();
  }
  else{
    //con cada for nos movemos hacia los costados
    for(int j=i;j<n;j++){
      //Verificamos que se cumpla las restriciones
      if(!((i==1 and cola.front()=='a') or
       (i==0 and cola.front()=='r'))){
        //metemos un objeto a la pila
        pila.push(cola.front());
        cola.pop();
        //con cada llamada recursiva nos movemos hacia
        //abajo del árbol osea en profundidad
        back(i+1,n,pila,cola);
        //sacamos un objeto de la pila
        cola.push(pila.top());
        pila.pop();
      }else{
        cola.push(cola.front());
        cola.pop();
      }
    }
  }
}
int main (int argc, char *argv[]) {
  queue<char> cola;
  stack<char> pila;
  cola.push('a');
  cola.push('n');
  cola.push('r');
  //cola.push('v');
  //cola.push('m');
  int i=0;
  back(i,cola.size(),pila,cola);
}
```

Backtacking: Esquema de diseño

Los algoritmos de vuelta atrás operan como sigue:

- La solución se construye de manera incremental, añadiendo un nuevo elemento en cada paso.
- El proceso de construcción de la solución es recursivo y equivale a recorrer en profundidad un árbol virtual en el que ciertos nodos representan respuestas parciales (incompletas) y otros (los que cumplen ciertas restricciones) son respuestas completas, es decir, potenciales soluciones al problema.
- Si un nodo respuesta s es solución, hay tres posibilidades:
  (1) el algoritmo termina, retornando s;
  (2) el algoritmo añade s al conjunto de soluciones y continúa el proceso de búsqueda;
  (3) el algoritmo evalúa la función objetivo en dicho nodo, $f(s)$, actualiza la solución óptima s_{opt} y continúa el proceso de búsqueda.
- Si un nodo respuesta no es solución, el algoritmo continúa el proceso de búsqueda. Ello implica deshacer decisiones previas (esto es, volver a una llamada recursiva anterior) y tomar en su lugar otras decisiones, que conducen a otras ramas del árbol y finalmente a nuevos nodos respuesta, que serán evaluados, etc.

Backtacking: Esquema de diseño

Backtracking rápido (encontrar una solución)

```
def backtracking(nodo):
    if EsSolucion(nodo):
        return nodo
    for v in Expandir(nodo):
        if EsFactible(v):
            s = backtracking(v)
            if s!=None:
                return s
    return None
```

Backtacking: Esquema de diseño

Backtracking exhaustivo (encontrar todas las soluciones)

```
def backtracking(nodo):
    lista_soluciones = []
    if EsSolucion(nodo):
        lista_soluciones.append(nodo)
    for v in Expandir(nodo):
        if EsFactible(v):
            ls = backtracking(v)
            lista_soluciones.extend(ls)
    return lista_soluciones
```

Backtacking: Esquema de diseño

Backtracking para optimización (encontrar la mejor solución)

```
def backtracking(nodo):
    if EsSolucion(nodo):
        mejor_sol = nodo
    else:
        mejor_sol = None
    for v in Expandir(nodo):
        if EsFactible(v):
            ms = backtracking(v)
            if mejor_sol==None or f(ms)>f(mejor_sol):
                mejor_sol = ms
    return mejor_sol
```

Backtacking: Esquema de diseño

Elementos principales del esquema de diseño

- `nodo`: almacena el conjunto de elementos (o decisiones) que conforman una rama del árbol (virtual) de búsqueda. Corresponde a una respuesta parcial o completa al problema planteado.
- `Expandir(nodo)`: devuelve la lista de nodos que resulta al añadir un nuevo elemento al conjunto de elementos representado por `nodo`. Se supone que existe un número finito de elementos con el que expandir cada nodo (en caso contrario, esta estrategia no sería viable computacionalmente).
- `EsFactible(nodo)`: devuelve un booleano que indica si es posible construir una solución al problema a partir del conjunto de elementos representado por `nodo` (en muchas ocasiones, los elementos de `nodo` serán incompatibles con las restricciones impuestas a una solución).
- `EsSolucion(nodo)`: devuelve un booleano indicando si el conjunto de elementos representado por `nodo` es solución al problema planteado.
- `f(nodo)`: función objetivo. Sin pérdida de generalidad, se supone que la mejor solución es aquella que maximiza el valor de `f`.

Ejercicio

Caso 1

Desarrollar el algoritmo de los asientos en Python, o en el lenguaje de su preferencia siguiendo el esquema de diseño presentado

Ejercicio

Caso 2

El problema de suma objetivo consiste en un conjunto de números enteros, sin repeticiones, donde el problema tiene por finalidad, encontrar el conjunto o subconjunto que al sumar los números de como resultado un número buscado, por ejemplo dado un conjunto de números enteros {2,3,6,1,5}, encontrar los subconjuntos cuya suma sea exactamente 9.

Ejemplo de suma objetivo 9:

2 6 1

2 1 6

3 6

3 1 5

6 2 1

6 3

6 1 2

El problema de la suma de conjuntos de enteros

Enunciado: Considerese un vector $v = (v_1, v_2, \ldots, v_n)$ con n enteros positivos distintos. El problema de la suma consiste en encontrar todos los subconjuntos de v cuya suma sea exactamente M.

- Las soluciones tendrán forma de tupla: $x = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$, y deberán cumplir: $\sum_{i=1}^{n} x_i v_i = M$
- Para resolver este problema, se aplicará un backtracking exhaustivo, que irá considerando todas las posibilidades (incluir/no incluir cada elemento en la solución: $x_i = 1$ y $x_i = 0$) a partir de una tupla vacía, hasta llegar a una tupla de tamaño n, que será evaluada y, si suma M, añadida al conjunto de soluciones.
- Dada una respuesta parcial $(x_1, x_2, \ldots, x_{k-1})$, y una decisión local x_k, con $k \leq n$, la función `EsFactible()` comprobará que no se ha sobrepasado y que aún es posible alcanzar la suma M:

$$\sum_{i=1}^{k-1} x_i v_i + x_k v_k \leq M$$

$$\sum_{i=1}^{k-1} x_i v_i + x_k v_k + \sum_{i=k+1}^{n} v_i \geq M$$

El problema de la suma de conjuntos de enteros: Implementación en lenguaje Python

Implementación en lenguaje Python

```
def precompute_table(v):
    global t        # t[k] = sum_{i=k+1-->len(v)-1} v[i]
    t=[0]
    for i in range(len(v)-1,0,-1):
        t.insert(0,t[0]+v[i])
def suma_enteros(x,v,suma,M):
    global t        # t[k] = sum_{i=k+1-->len(v)-1} v[i]
    k = len(x)
    n = len(v)
    if k == n:      # es solucion: suma = M
        return [x]
    ls = []
    # if EsFactible x[k]=1
    if suma+v[k] <= M and suma+v[k]+t[k] >= M:
        ls = ls + suma_enteros(x[:]+[1],v,suma+v[k],M)
    # if EsFactible x[k]=0
    if suma+t[k] >= M:
        ls = ls + suma_enteros(x[:]+[0],v,suma,M)
    return ls
```

Características de los Algoritmos de BackTracking

El método de backtracking es una técnica de resolución de problemas que realiza una búsqueda exhaustiva, sistemática y organizada sobre el espacio de búsqueda del problema

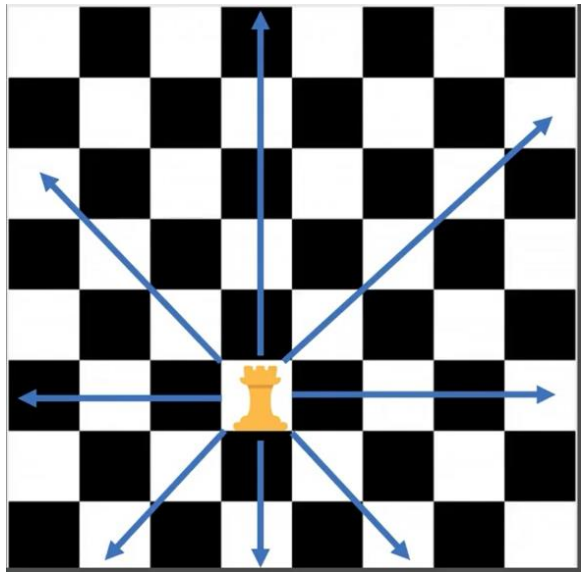
Funciona en problemas cuyas soluciones se pueden construir por etapas. La solución maximiza, minimiza o satisface una función criterio.

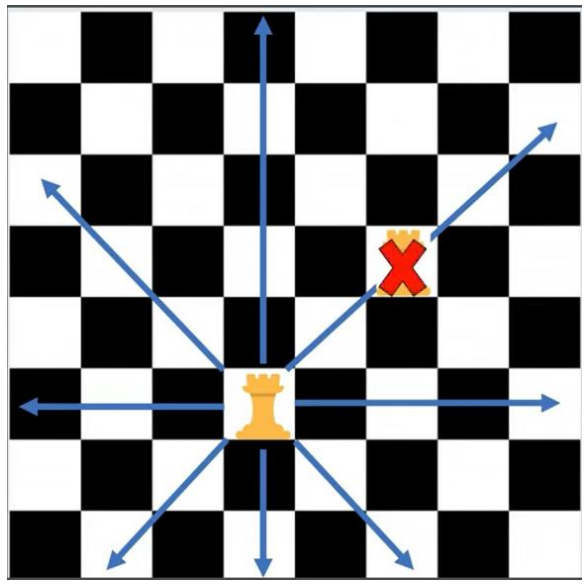
Se utiliza para implementar algunos juegos. Por ejemplo el ajedrez, el 3 en raya y los laberintos. Se usa también en los análisis gramaticales (también llamados parsers) y en procesos de calendarización (scheduling).

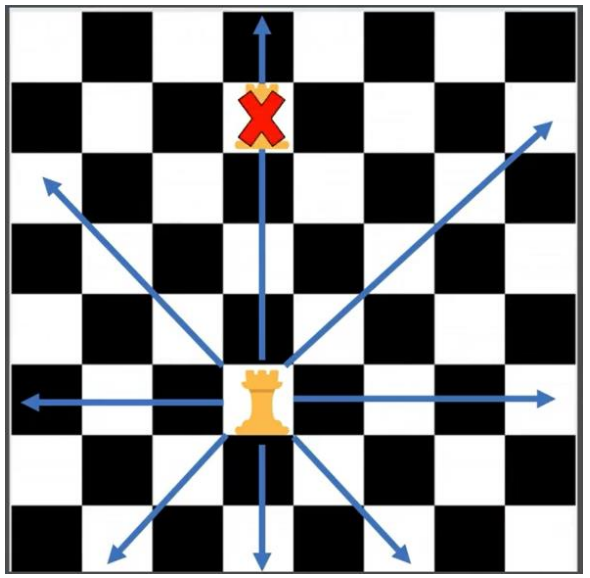
El espacio de posibles soluciones se estructura como un árbol de exploración, en el que en cada nivel se toma la decisión de la etapa correspondiente, y se le llama nodo estado a cualquier nodo del árbol de exploración.

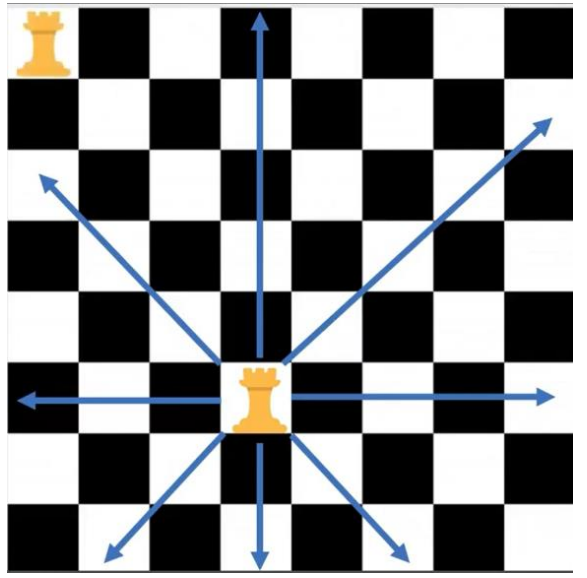
En cada etapa de búsqueda en el algoritmo backtracking, el recorrido del árbol de búsqueda se realiza en profundidad. Si una extensión de la solución parcial actual no es posible, se retrocede para intentar por otro camino, de la misma manera en la que se procede en un laberinto cuando se llega a un callejón sin salida.

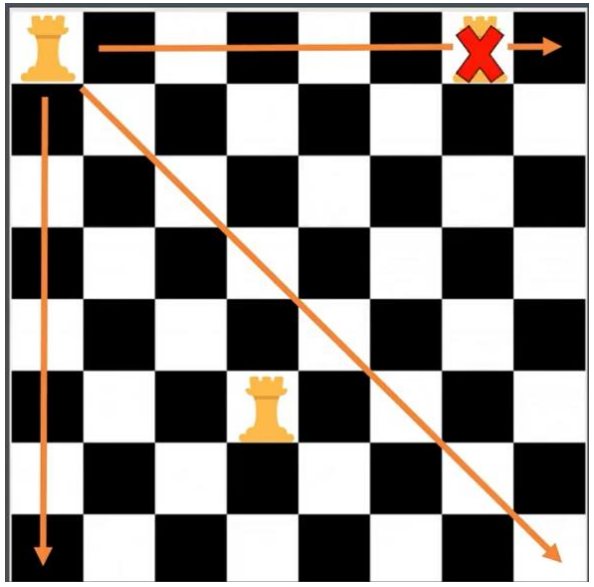
El problema de las n reinas

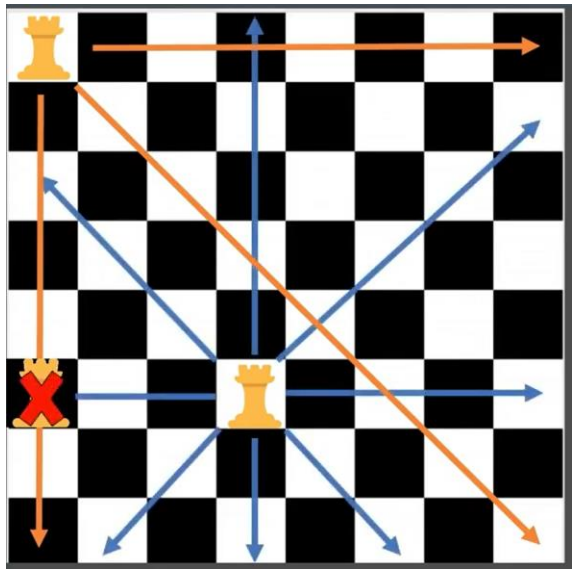




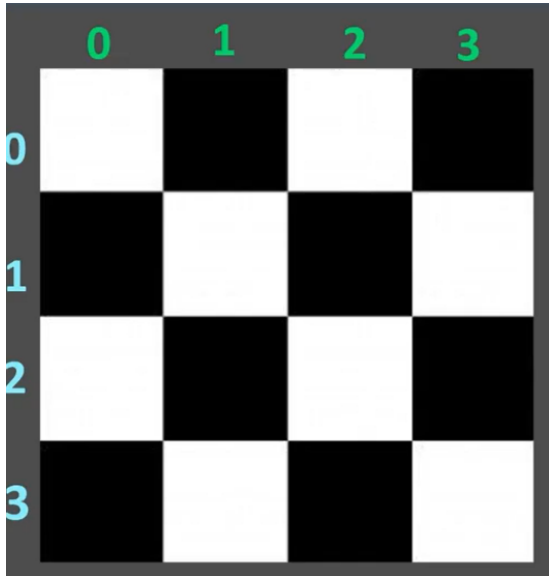


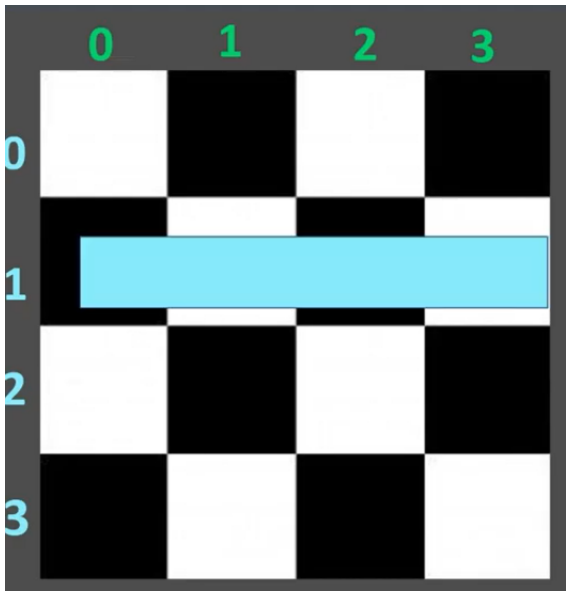


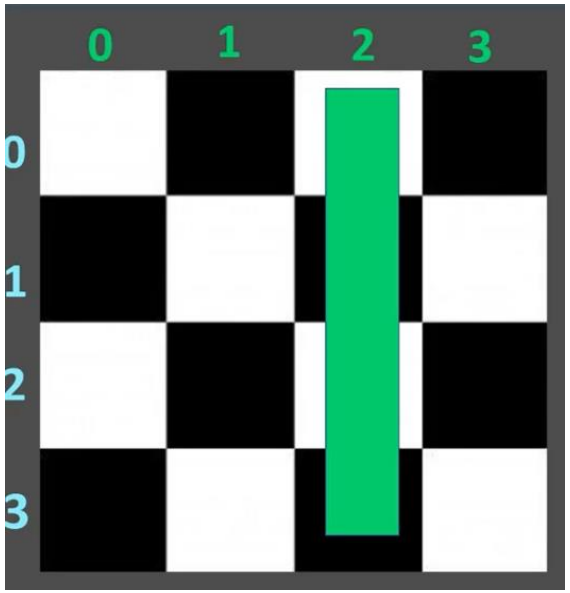


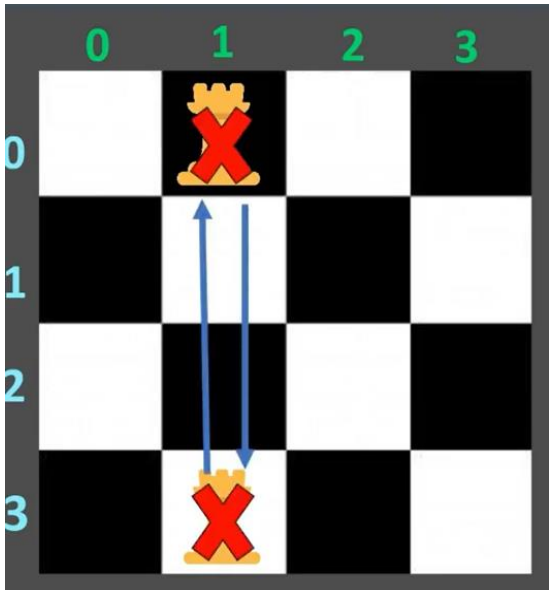


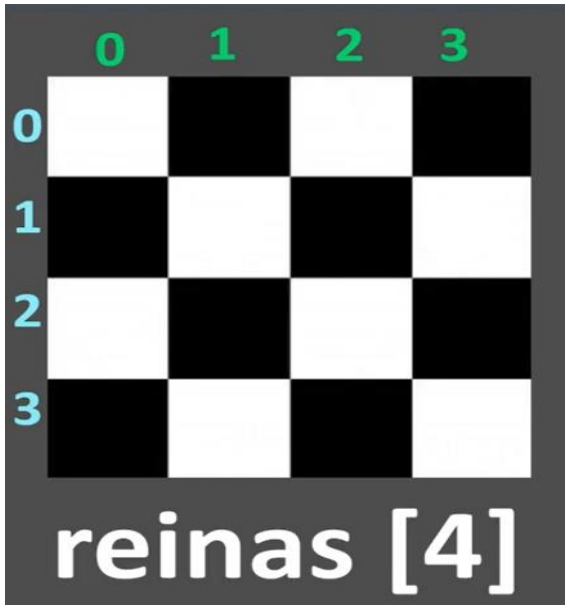
El problema de las n reinas

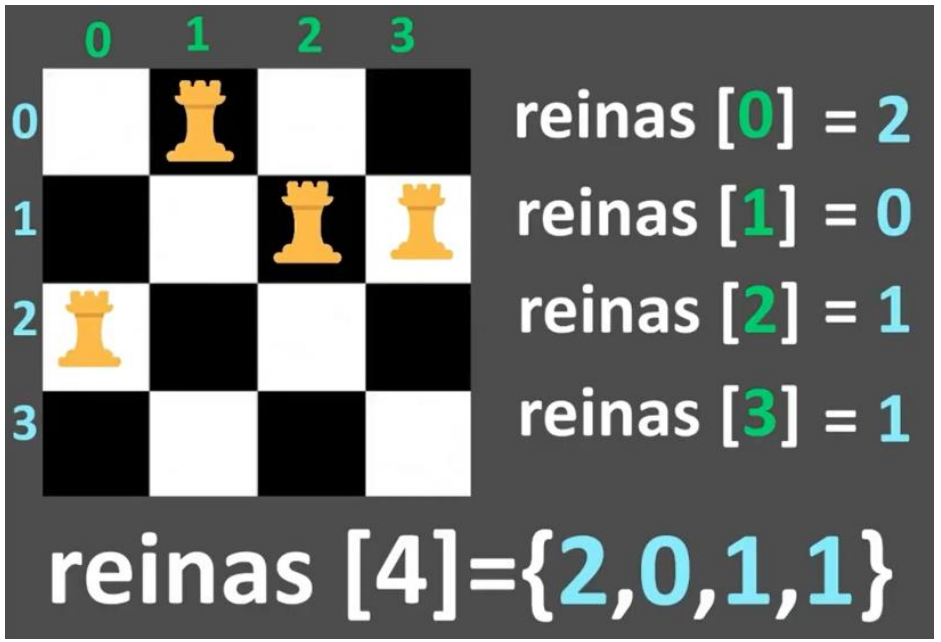




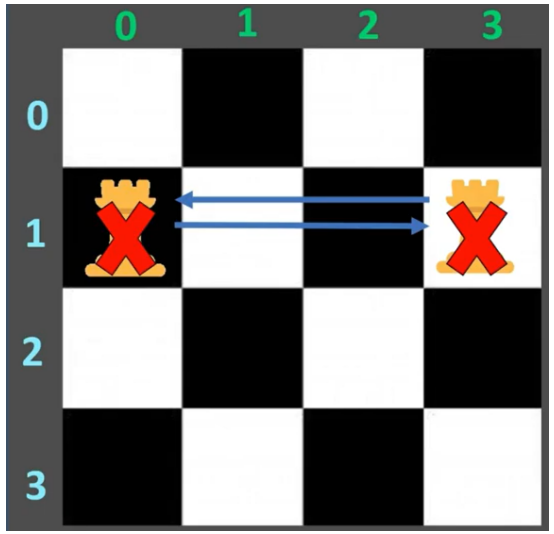


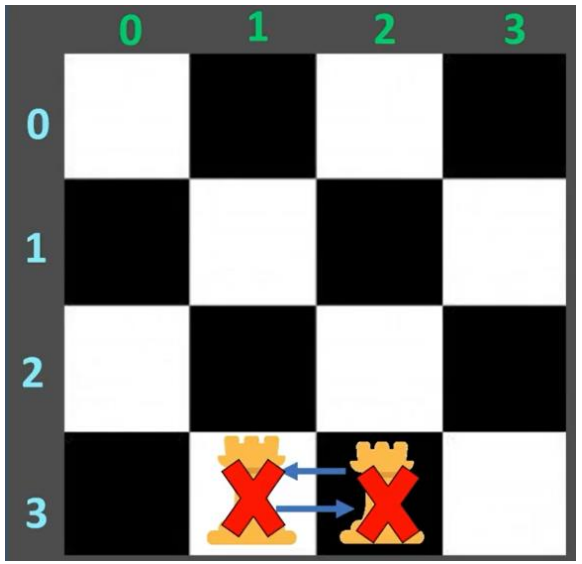


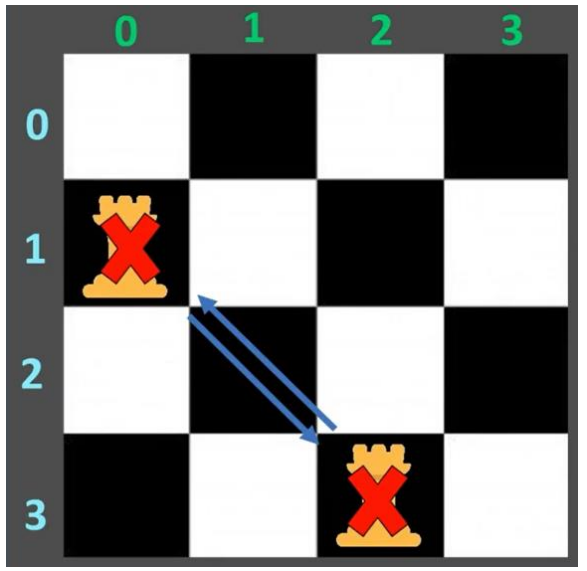


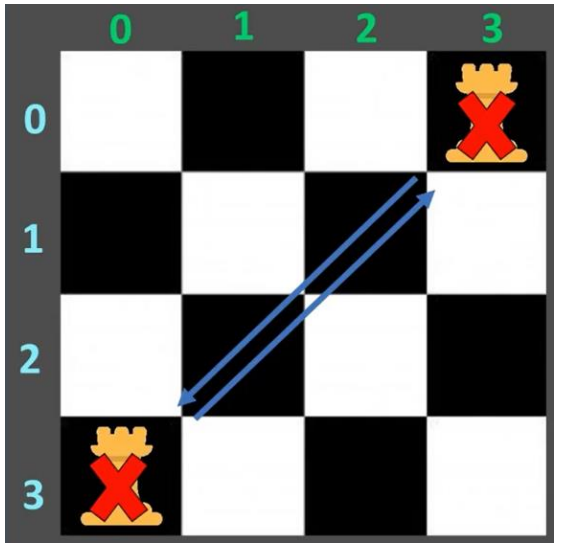


El problema de las n reinas









1 = 1

reinas[0] = reinas[3]

3 = 3

reinas[1] = reinas[2]

reinas[0] = 1
reinas[2] = 3

0-2 1-3
|-2| |-2|
2 2

reinas[3] = 0
reinas[0] = 3

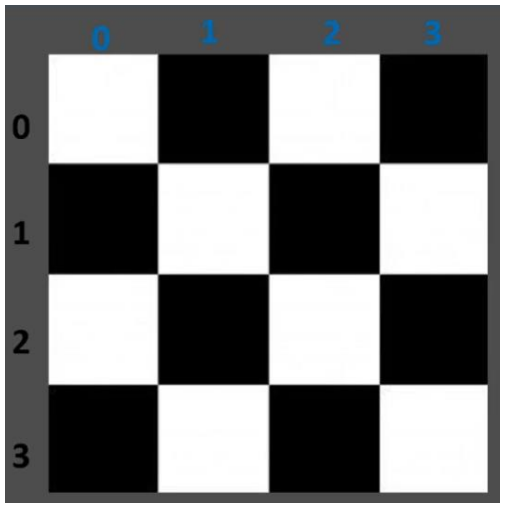
3-0 0-3
|3| |-3|
3 = 3

reinas[i] = reinas[k]

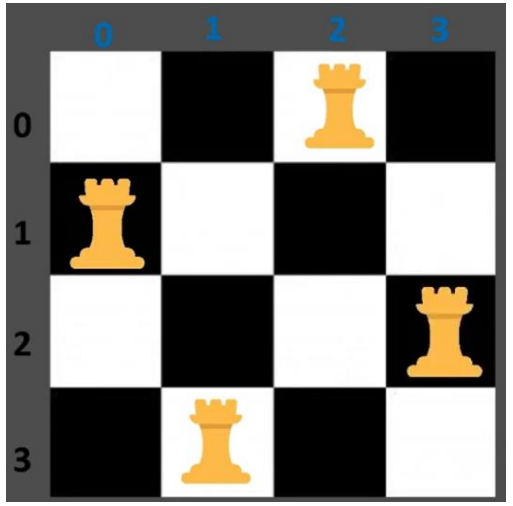
reinas[i] = x
reinas[k] = y

|i-k| = |x-y|

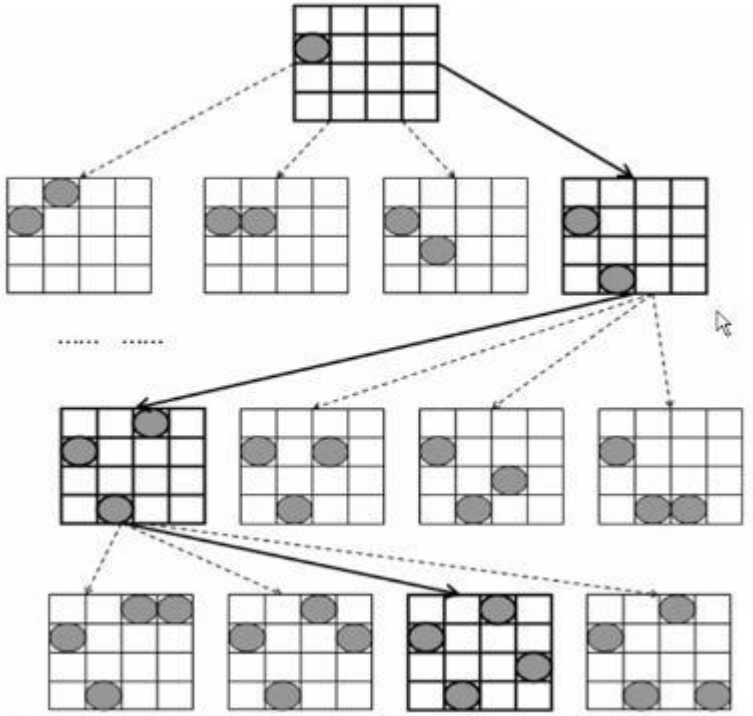
El problema de las n reinas

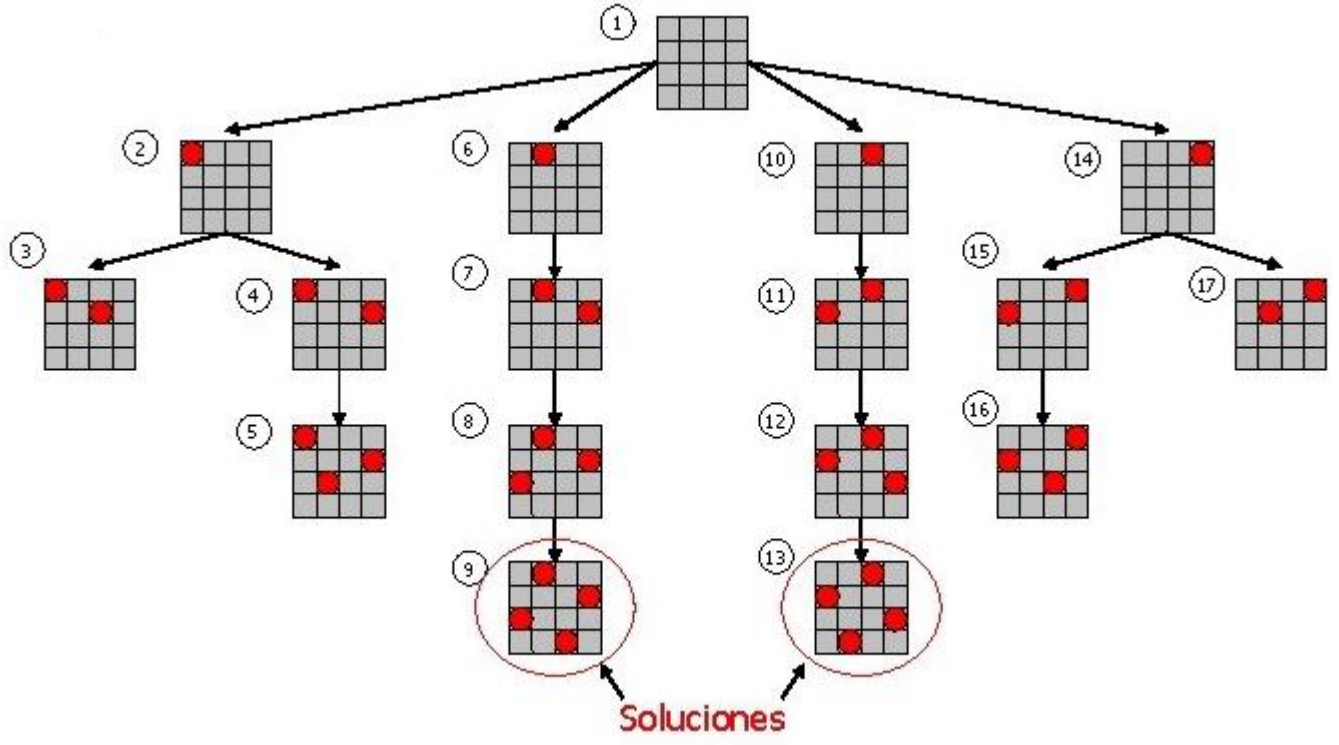


reinas[4] = {-1 ,-1 ,-1 ,-1 }

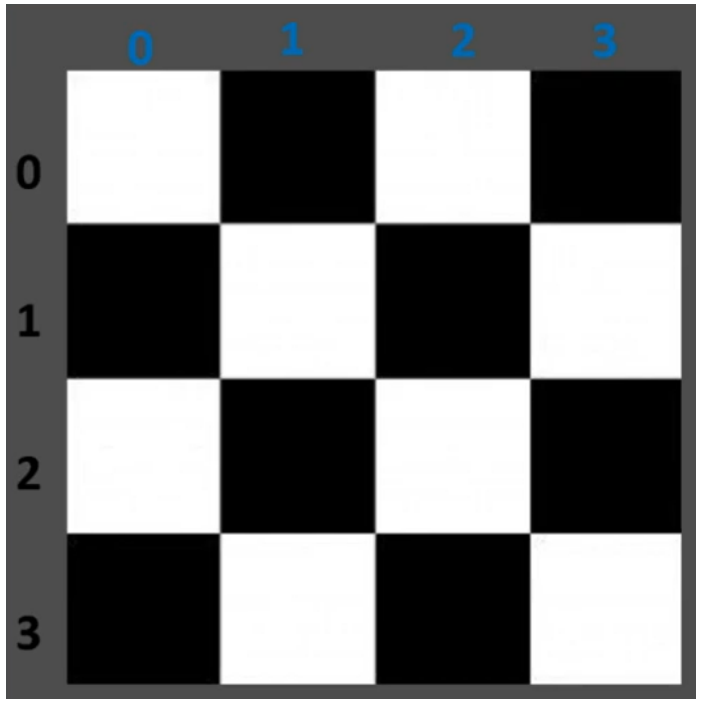


reinas[4] = { 1 , 3 , 0 , 2 }

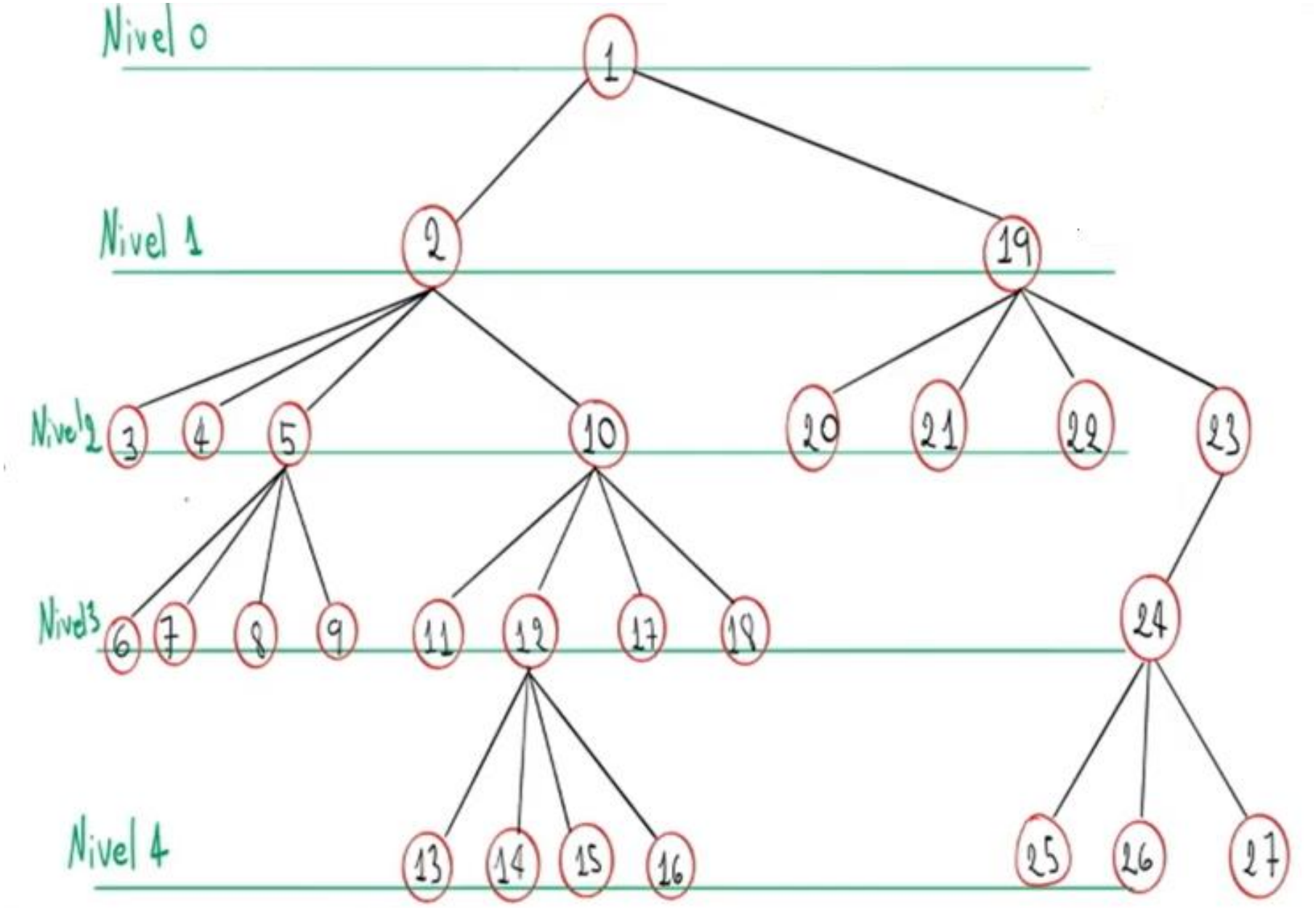




El problema de las n reinas

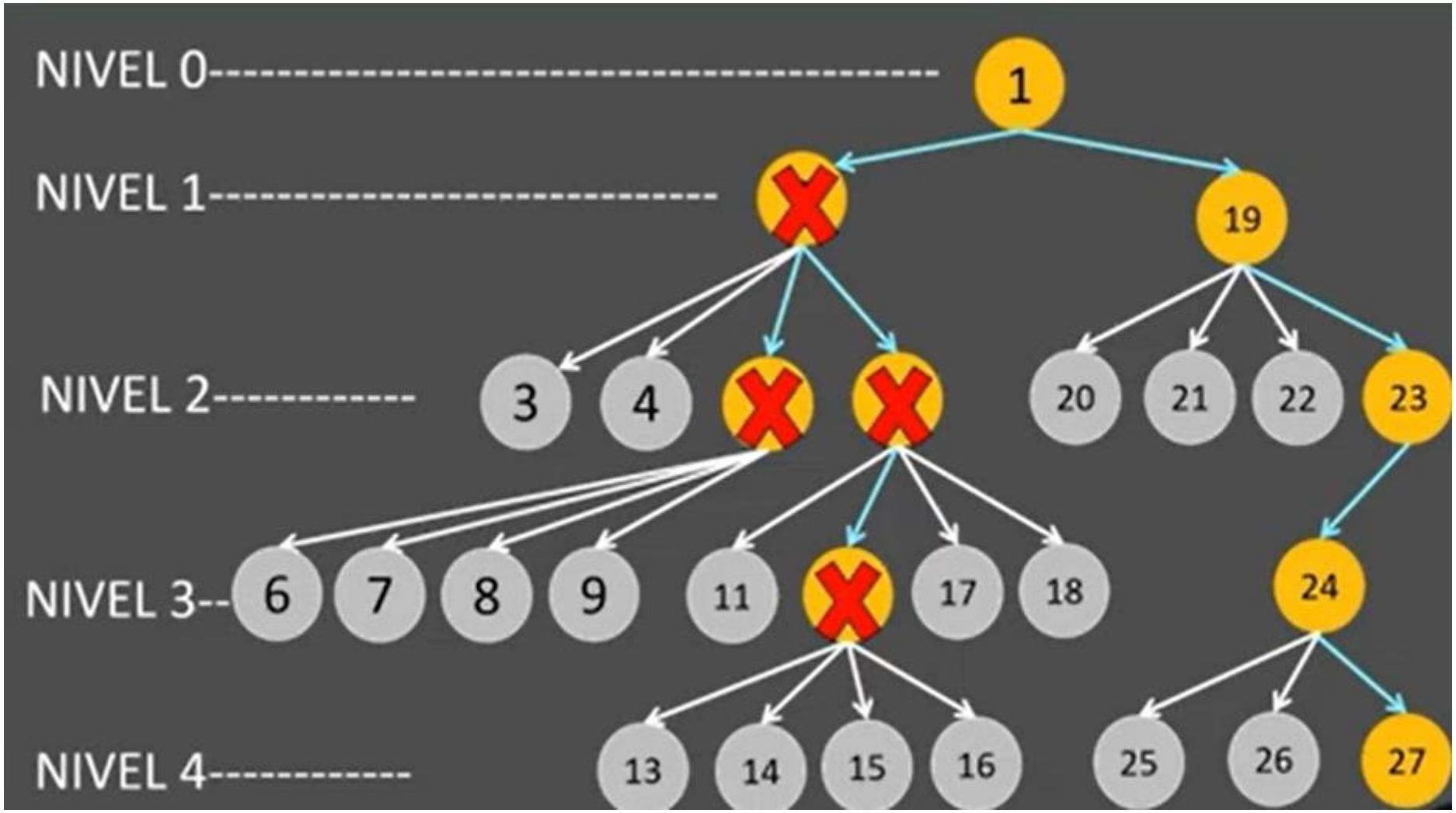


reinas[4] = {-1 ,-1 ,-1 ,-1 }



El problema de las n reinas - código

NIVEL 0 -> k=0 ----



NIVEL 4 -> k=4 ----

El problema de las n reinas - código

```
#include <iostream>
#include<stdio.h>
using namespace std;
int x=0;
//Acá comprobamos si la reina está colocada en una posición
//válida
//para que la posición sea válida las reinas no deben estar
en la
//misma columna
//y tampoco pueden estar en la misma fila, ni diagonal
bool comprobar(int reinas[],int n, int k){
  int i;
  for(i=0;i<k;i++){
    if( (reinas[i]==reinas[k]) or (abs(k-i) ==
abs(reinas[k]-reinas[i]))){
      return false;
    }
  }
  return true;
}
void Nreinas(int reinas[],int n, int k){
//este es el caso base que me indica que se ha encontrado
//una solución
//por lo cual ya no hay mas reinas por colocar. Hemos
//logrado poner
//todas las reinas en el tablero de ajedrez
  if(k==n){
    x++;
    cout<<"Solucion "<<x<<" : ";
    for(int i=0;i<n;i++){
      cout<<reinas[i]<<" , ";
    }
    cout<<endl;
  }
  //aun quedan reinas por colocar y el algoritmo debe
seguir buscando
  else{
    for(reinas[k]=0;reinas[k]<n;reinas[k]++){
      if(comprobar(reinas,n,k)){
        Nreinas(reinas,n,k+1);
      }
    }
  }
}
int main(int argc, char *argv[]) {
  int k=0;
  int cant;
  cout<<"Ingresar la cantidad de reinas : ";
  cin>>cant;
  int *reinas = new int[cant];
  for(int i=0;i<cant;i++){
    reinas[i]=-1;
  }
  Nreinas(reinas,cant,k);
  return 0;
}
```

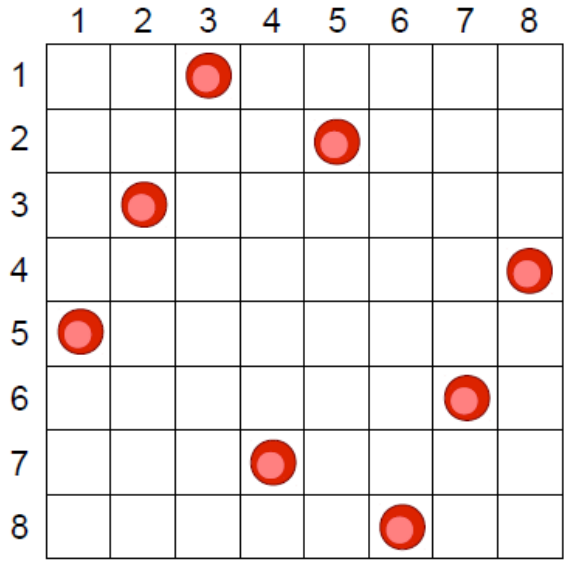
El problema de las n reinas

Enunciado: Considerese un tablero de ajedrez de $n \times n$ casillas (típicamente, $n = 8$). Se trata de colocar n reinas en el tablero sin que se amenacen entre sí. Basta con encontrar una solución (esquema de backtracking *rápido*).

- Un nodo consistirá en un vector c de tamaño $k \leq n$, tal que $c[j]$ será la columna que ocupa la reina en la fila j, $\forall j \in [1, k]$.
- La estrategia de construcción de la solución (función `Expandir()`) consistirá en expandir el vector c poniendo una reina en la siguiente fila del tablero.
- La función `EsFactible()` se encarga de comprobar que cada nueva reina se coloca de modo que no amenaza a las anteriores, evitando las columnas y diagonales ya cubiertas (esto reduce enormemente el coste).
- Por otra parte, un nodo será solución si $k = n$.
- Como el esquema es un backtracking rápido, el algoritmo termina en cuanto encuentra una solución.

El problema de las n reinas

Una de las soluciones para un tablero de 8×8



Implementación en lenguaje Python

```
def EsFactible(k,j,c):
    for i in range(k):
        if c[i]==j or (i-c[i])==(k-j) or (i+c[i])==(k+j):
            return False
    return True

def reinas(c,n):
    k = len(c)
    if k == n:
        return c
    for j in range(n):
        if EsFactible(k,j,c):
            v = c[:] + [j]
            s = reinas(v,n)
            if s != None:
                return s
    return None
```

Si se generasen todas las posiciones, la complejidad sería del orden de n^n.
Pero, ¿cuál es el coste en la práctica? Por ejemplo, para $n = 8$, $n^n = 16{,}777{,}216$, pero son necesarias tan sólo 114 llamadas para encontrar una solución.

El problema de la mochila (discreto)

Enunciado: Considerese una mochila capaz de albergar un peso máximo M, y n elementos con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$. Se trata de encontrar qué combinación de elementos, representada mediante la tupla $x = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$, maximiza el beneficio:

$$f(x) = \sum_{i=1}^{n} b_i x_i \tag{1}$$

sin sobrepasar el peso máximo M:

$$\sum_{i=1}^{n} p_i x_i \leq M \tag{2}$$

- En este caso, se aplicará un esquema de backtracking para optimización, que explorará todas las posibles soluciones a partir de una tupla prefijo de tamaño k, con un peso restante r, y devolverá la tupla que maximiza la función objetivo (1).
- La función `EsFactible()` simplemente comprobará la condición (2).

El problema de la mochila (discreto)

Implementación en lenguaje Python

```
def mochila_d_bt(x,r,p,b):
    k = len(x)
    n = len(p)
    if k == n:
        return x,0
    max_b = 0
    mejor_sol = x[:] + [0]*(n-k)
    for i in range(k,n):
        if p[i] <= r:
            x_new = x[:] + [0]*(i-k) + [1]
            ms,b_ms = mochila_d_bt(x_new,r-p[i],p,b)
            if b[i] + b_ms > max_b:
                max_b = b[i] + b_ms
                mejor_sol = ms
    return mejor_sol,max_b
```

Ejercicio

Dibujar el árbol vinculado al algoritmo

Ejercicio

Caso 3

Considérese una lista s con n letras, todas ellas diferentes. Escribir en lenguaje Python una función que retorne la lista de todas las palabras formadas por m ≤ n letras distintas que puedan construirse con ellas, usando un esquema de búsqueda exhaustiva.

Lo que se pide es un problema clásico de combinatoria: encontrar las variaciones de n elementos tomados de m en m. La función deberá obtener el conjunto de soluciones que se derivan de un prefijo p de longitud k (donde todas las letras son diferentes), creando un nuevo prefijo q con cada una de las letras no presentes en p y añadiendo al conjunto solución de p todas las palabras cuyo prefijo es q. La función retornará el conjunto solución. Si el prefijo p tiene la longitud deseada m, el conjunto solución estará formado por el propio prefijo p

Ejercicio

Caso 4

El problema del caballo de ajedrez consiste en encontrar un recorrido de un tablero de ajedrez de n × n casillas que incluya exactamente una vez cada casilla, utilizando los movimientos del caballo a partir de una casilla inicial arbitraria.

Escribir en lenguaje Python un algoritmo de backtracking que obtenga una solución al problema. Escribir a continuación una versión mejorada que utilice el heurístico de Warnsdorff para ordenar las casillas a recorrer a partir de una dada (véase http://en.wikipedia.org/wiki/Knight's tour). ¿Se nota la diferencia?

Ramificación y Poda: Características generales

- **Branch and Bound** es una variante del esquema de *backtracking*
- Al igual que *backtracking*, se basa en el recorrido del árbol de expansión en busca de soluciones
- Se aplica sobre todo a **problemas de optimización**: búsqueda de la mejor solución a un problema, aunque también para buscar una o todas las soluciones a un problema
- La principal diferencia con *backtracking* es que la generación de nodos del árbol de expansión se puede realizar aplicando distintas estrategias $\rightarrow$ **estrategias de ramificación**
- Además se utilizan **cotas** que permiten podar ramas que no conducen a una solución óptima (se evita ramificar nodos) $\rightarrow$ **estrategias de poda**

Ramificación y Poda: Características generales

- En cuanto a la **estrategia de ramificación**
  - En el esquema de *backtracking*, el recorrido del árbol de expansión siempre es en profundidad
  - En ramificación y poda, la generación de los nodos del árbol de expansión puede seguir varias extrategias:
    - ★ en profundidad (LIFO)
    - ★ en anchura (FIFO)
    - ★ aquella que selecciona el nodo más prometedor

    El objetivo es utilizar la estrategia que permita encontrar la solución más rápidamente
- Se realiza una **estrategia de poda** en la que en cada nodo se calcula una **cota** del posible valor de aquellas soluciones que pudieren encontrarse más adelante en el árbol. Esto permite no explorar aquellas ramas que no conducen a una solución válida u óptima (en el caso de problemas de optimización)

El problema de la asignación de tareas

Enunciado: Dados n agentes y n tareas y una matriz de costes c de tamaño $n \times n$, tal que $c(a, t)$ es el coste de que el agente a lleve a cabo la tarea t, el problema consiste en **asignar una tarea a cada agente de manera que el coste total acumulado sea mínimo**. Cada agente realizará una sola tarea y todas las tareas deben ser realizadas.

- Existen $n!$ combinaciones distintas, demasiadas incluso para valores moderados de n.
- Se aplicará una estrategia de ramificación y poda, a partir de dos valores de referencia:
  1. *cma*: el **coste mínimo absoluto** alcanzable
  2. *cms*: el **coste de la mejor solución** obtenida hasta el momento
- Si la búsqueda encuentra una solución cuyo coste iguala *cma*, la búsqueda habrá finalizado, porque no es posible encontrar soluciones mejores.
- El valor de *cms* se va actualizando a medida que avanza la búsqueda y se utiliza para podar nodos v cuyo coste mínimo alcanzable $c_{min}(v)$ sea igual o superior a *cms*. Nótese que cada vez que se actualiza *cms* es necesario volver a revisar los nodos vivos para ver cuáles de ellos deben ser podados.
- Por otra parte, el **nodo más prometedor** (el nodo a expandir a continuación) será **aquel que tenga un valor de c_{min} más pequeño**.

El problema de la asignación de tareas: Ejemplo

Agentes	Tareas 1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	18

(1) $cma = 54$
- Suma de los mínimos de fila $\rightarrow$ 49
- Suma de los mínimos de columna $\rightarrow$ 54

Nos quedamos con el máximo de ambos, ya que todos los agentes deben realizar una tarea y todas las tareas deben ser realizadas.

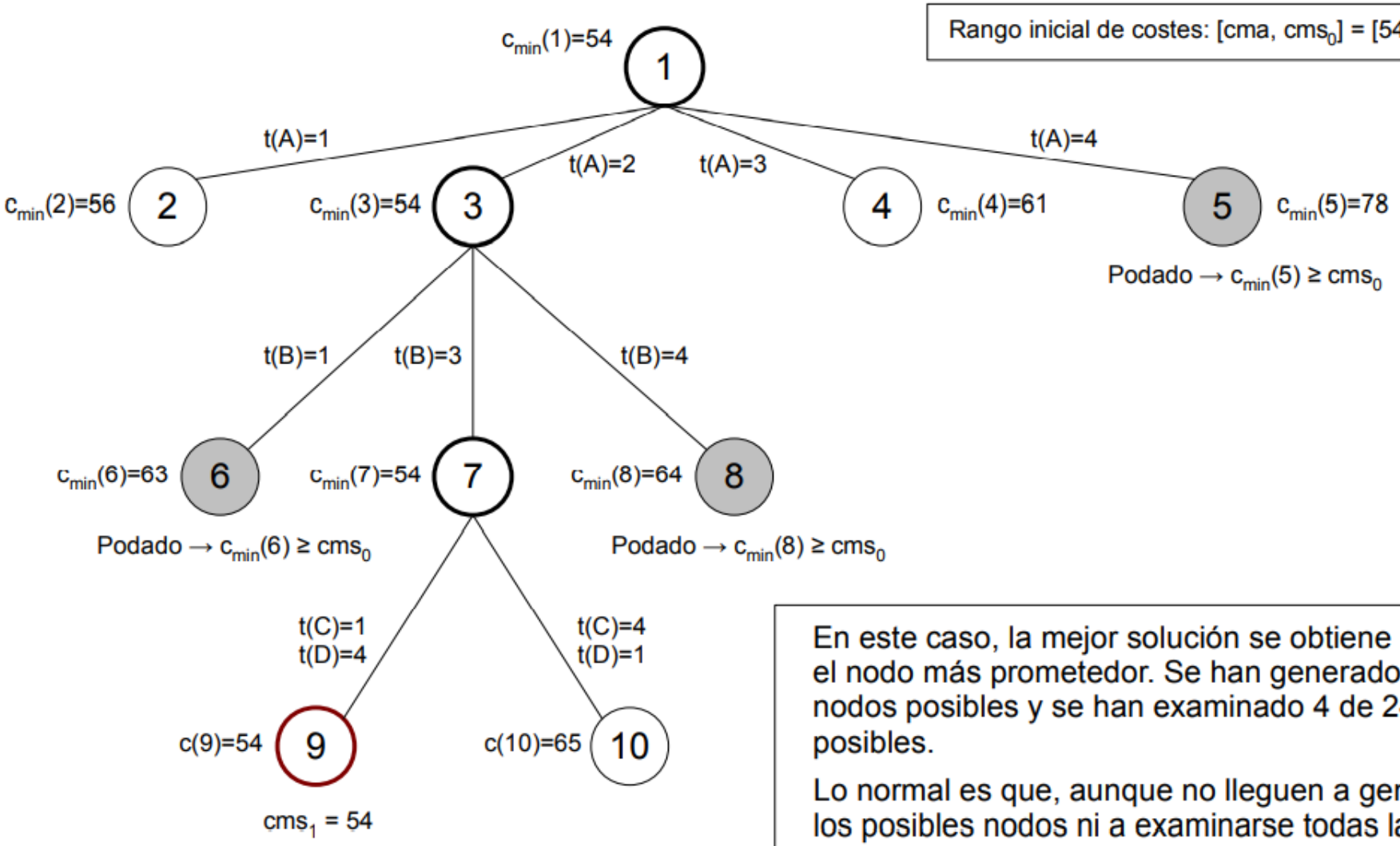
(2) Inicialización de $cms = 63$
- Diagonal principal $\rightarrow$ 63
- Diagonal opuesta $\rightarrow$ 87

Nos quedamos con la solución de coste mínimo.

(3) Inicialmente, el rango de costes es $[54, 63]$.

(4) El coste mínimo alcanzable a partir de un nodo v, $c_{min}(v)$, se calcula del mismo modo que cma:
- para los agentes y tareas que quedan por asignar en v, se suman los mínimos de fila por un lado: $s_{minf}(v)$, y los de columna por otro: $s_{minc}(v)$;
- $c_{min}(v)$ = coste de las asignaciones en v + máximo$(s_{minf}(v), s_{minc}(v))$.

El problema de la asignación de tareas: Ejemplo



En este caso, la mejor solución se obtiene siguiendo el nodo más prometedor. Se han generado 10 de 41 nodos posibles y se han examinado 4 de 24 soluciones posibles.

Lo normal es que, aunque no lleguen a generarse todos los posibles nodos ni a examinarse todas las posibles soluciones, la búsqueda implique un mayor número de pasos y la mejor solución aparezca en una rama heurísticamente subóptima.

El problema de la asignación de tareas: Implementación en Python

```
def asignar(c):
    n = len(c)
    ms, cms = init_ms(c)
    cma = compute_cma([],c)
    if cma<cms:
        nodo_raiz = [[],cma] # nodo raiz: [tupla vacia, cma global]
        lista = [nodo_raiz]  # lista de nodos vivos
    while len(lista)>0:      # mientras queden nodos vivos
        x, cma = lista.pop() # extraemos el nodo mas prometedor
        tareas = [t for t in range(n) if t not in x]
        for t in tareas:     # ramificacion
            x_new = x[:]+[t]
            cma_new = compute_cma(x_new,c)
            if cma_new<cms:  # nodo podado si cma_new >= cms
                if len(x_new)==n: # x_new es solucion
                    ms, cms = x_new, cma_new
                    # cms actualizado --> poda de nodos con cma>=cms
                    while len(lista)>0 and lista[0][1]>=cms:
                        lista.pop(0)
                else: # insercion en orden (de mayor a menor cma)
                    i=len(lista)-1
                    while i>=0 and lista[i][1]<cma_new:
                        i=i-1
                    lista.insert(i+1,[x_new,cma_new])
    return ms, cms
```

El problema de la asignación de tareas: Funciones auxiliares

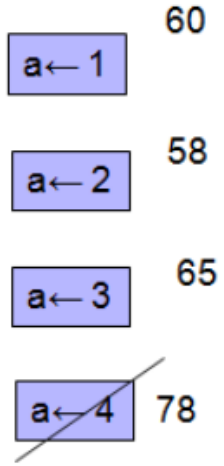
```
def init_ms(c):
    n = len(c)
    cdiag1 = 0
    cdiag2 = 0
    for i in range(n):
        cdiag1 += c[i][i]
        cdiag2 += c[i][n-i-1]
    if cdiag1<=cdiag2:
        return [i for i in range(n)], cdiag1
    else:
        return [n-i-1 for i in range(n)], cdiag2

def compute_cma(x,c):
    n = len(c)
    k = len(x)
    coste_x = 0
    for i in range(k):
        coste_x += c[i][x[i]]
    sum_minf = sum(min(c[i][j] for j in range(n) if j not in x) for i in range(k,n))
    sum_minc = sum(min(c[i][j] for i in range(k,n)) for j in range(n) if j not in x)
    return coste_x + max(sum_minf,sum_minc)
```

El problema de la asignación de tareas: Otro forma del árbol

Podemos obtener una cota superior al problema con el costo de asignar cualquier tarea a cualquier trabajador. Por ejemplo, el trabajador “a” la tarea 1, a “b” la tarea 2, a “c” la 3 y a “d” la 4, cuyo costo sería 11+15+19+28 = 73. Si tomáramos otra diagonal, haciendo a = 4, b = 3, c = 2, d = 1, el costo sería 40+13+17+17= 87, así que 73 es la mejor cota superior de estas dos, ya que es la menor. Para establecer una cota inferior, tomamos el menor costo de cada tarea, independientemente de quien la ejecute, así tenemos que 11+12+13+22 = 58 es la cota inferior. Otra opción es sumar el costo más pequeño de cada fila, asumiendo que cada trabajador tiene que hacer una tarea. En este caso 11+13+11+14 = 49 es otra cota inferior, sin embargo, 58 es una cota inferior más útil, porque es superior a 49 y así se acorta el intervalo de búsqueda. Entonces tenemos que la respuesta se encuentra entre 58 y 73.

Trabajador/ Tarea	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

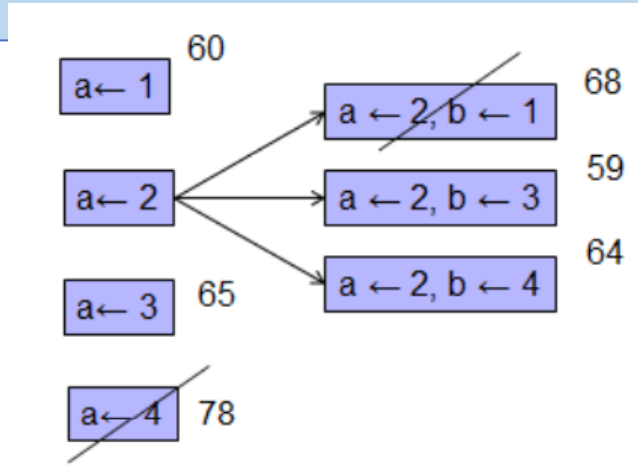


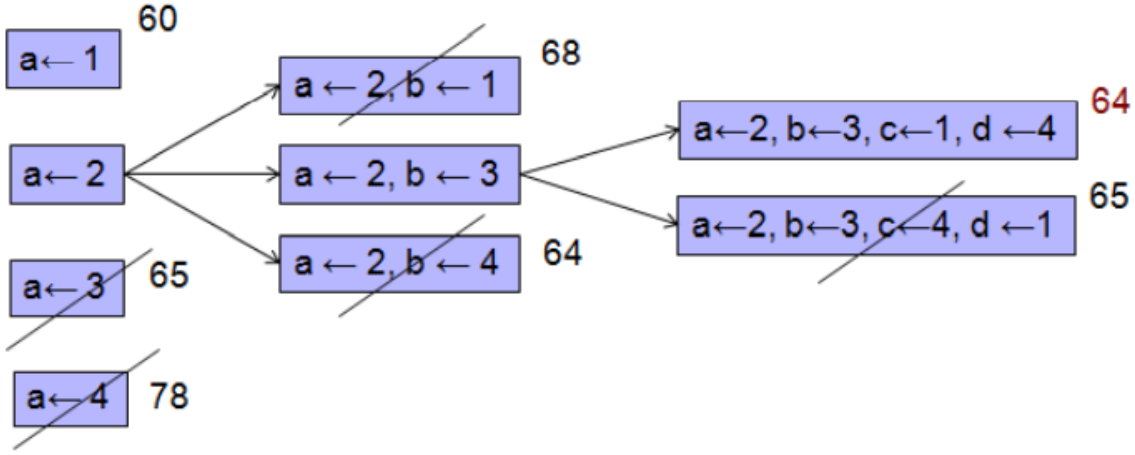
11+14+13+22 = 60

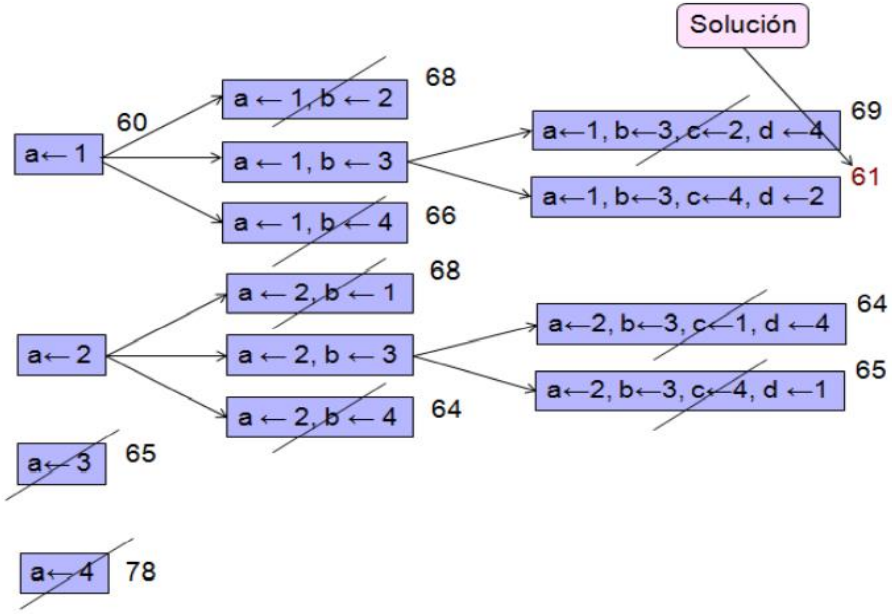
11+12+13+22 = 58

11+14+18+22 = 65

11+14+13+40 = 78







Ejercicio

Caso 3

Resuelve por ramificación y poda el siguiente problema de asignación de tareas

Trabajador/ Tarea	1	2	3	4
a	29	19	17	12
b	32	30	26	28
c	3	21	7	9
d	18	13	10	15

Esquema general de los algoritmos de backtracking

Para una determinada etapa, se elige una de las opciones dentro del árbol de búsqueda y se analiza para determinar si esta opción es aceptable como parte de la solución parcial. En caso de que no sea aceptable, se procede a elegir otra opción dentro de la misma etapa hasta encontrar una opción aceptable o bien agotar todas las opciones. Si se encuentra una solución aceptable, ésta se guarda y se profundiza en el árbol de búsqueda aplicando el mismo procedimiento Retroceso a la siguiente etapa. En caso de que Retroceso devuelva como resultado que no hubo éxito, entonces se retira la opción que se eligió y se intenta con otra, de lo contrario, cuando sí hubo éxito entonces Retroceso termina y regresa el control al procedimiento que lo llamó. Al final del procesamiento Retroceso termina cuando se encontró un caso exitoso o cuando se agotaron todas las opciones.

```
Función Retroceso( etapa )
  Inicializar_Opciones():
  Hacer
     opcion←SeleccionarOpcion
     Si aceptable(opcion) Entonces
      Guardar (opcion)
      Si incompleta(solucion) Entonces
        exito ←Retroceso( siguiente( etapa ) ) // recursión
        Si ( exito = FALSO ) Entonces
          retirar( opcion )
        Fin Si
      Si no  // solución completa u hoja del árbol
        exito ← VERDADERO
      Fin Si
     Fin Si
  mientras (exito = FALSO ) AND ( opcion != UltimaOpcion)

  regresar exito

Fin Retroceso
```

¿Preguntas?

FIN