# Backtracking

Diseño y Análisis de Algoritmos

Universidad
Rey Juan Carlos

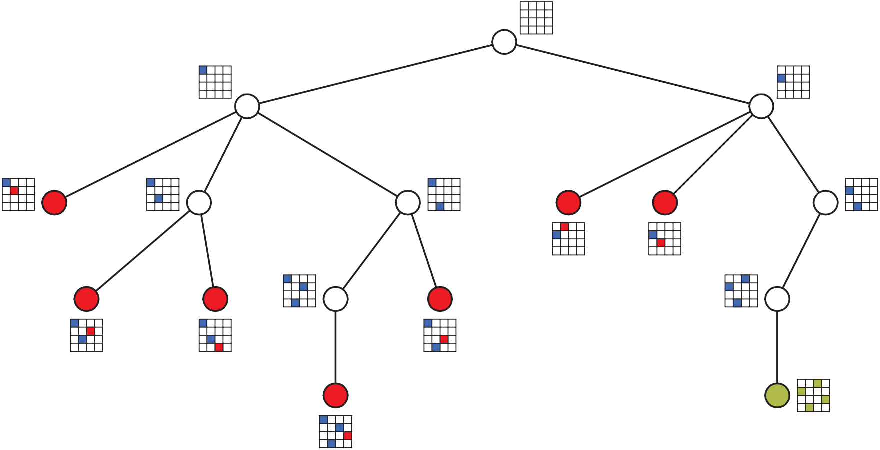
# Contenidos

# Introducción

# Backtracking - Vuelta atrás

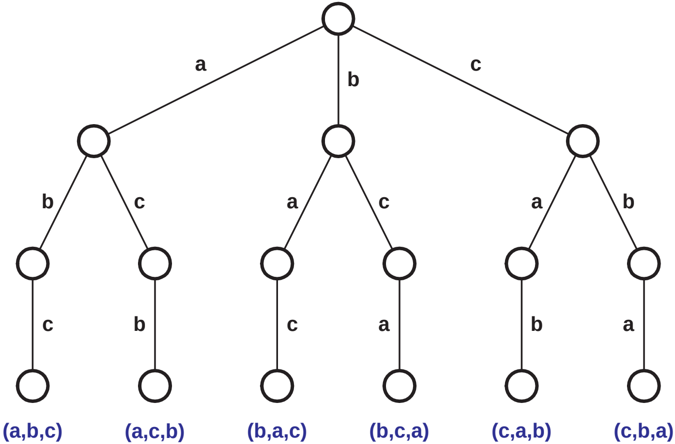
- Estrategia para encontrar soluciones a problemas con restricciones definidos sobre espacios discretos (de elevado tamaño)
- Construye soluciones parciales progresivamente, las cuales deben cumplir las restricciones del problema
- El recorrido (en profundidad) tiene éxito si, procediendo de esta forma, se puede definir por completo una solución (en una hoja de un árbol de recursión)
  - Puede detenerse al encontrar una solución o seguir hasta encontrar todas
- Si en alguna etapa la solución parcial construida hasta el momento no se puede completar, se **vuelve atrás** deshaciendo la solución parcial, hasta un punto donde puede seguir explorando posibles soluciones
- Es un método de "fuerza bruta" pero "inteligente"

# Ejemplo - 4 reinas

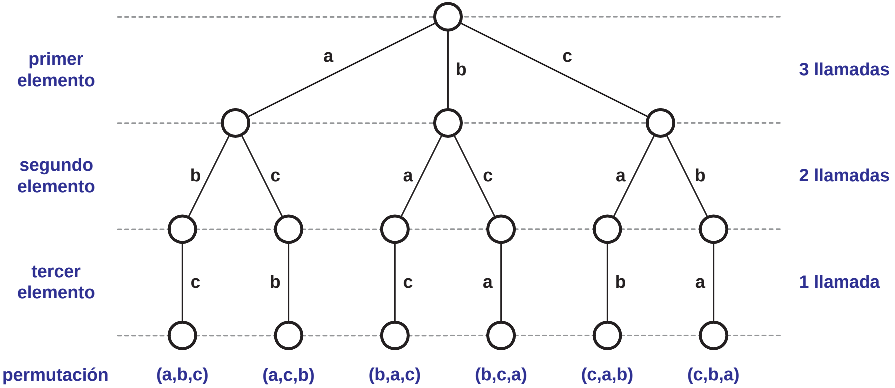


# Árboles de búsqueda

# Permutaciones de $\{a, b, c\}$



# Permutaciones de $\{a, b, c\}$



# Permutaciones de $\{a, b, c\}$

- ¿Cuántas llamadas recursivas se hacen en cada nivel?
  - 3, 2, 1 (se podrían controlar con bucles)
  - Pero lo más fácil es generar siempre 3 posibles llamadas, y usar un vector de valores booleanos para ver si realmente se debe realizar la llamada recursiva
  - El vector de booleanos se puede pasar por valor o referencia
    - Referencia: habrá que deshacer los cambios al retroceder (C o Java)
    - Valor: no será necesario deshacer cambios
- El nivel de la llamada indica la posición del nuevo elemento a añadir
- Al llegar al último nivel (a las hojas) tenemos completada la permutación

# Implementación - I

```
void permutaciones(int n){
  int[] perm = new int[n];
  boolean[] libres  = new boolean[n];

  for(int i=0; i<n; i++)
    libres[i] = true;

  perms(n, 0, perm, libres);
}

void imprimir(int[] v){
  for(int i=0; i<v.length; i++)
    System.out.print(v[i]+" ");

  System.out.println();
}
```

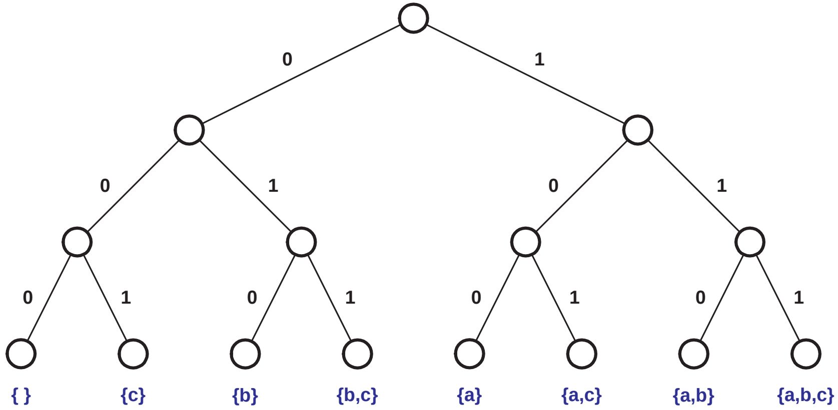
# Implementación - II

```
void perms(int n, int i, int[] solucion, boolean[] xs){
  for(int k=0; k<n; k++)
    if(xs[k]){
      solucion[i] = k;
      xs[k] = false;

      if(i==n-1)
        imprimir(solucion);
      else
        perms(n, i+1, solucion, xs);

      xs[k] = true;
    }
}
```

- `n`: número de elementos (profundidad del árbol)
- `i`: posición del elemento a insertar
- `solucion`: permutación parcial construida
- `xs`: indica los elementos incluidos en la solución parcial

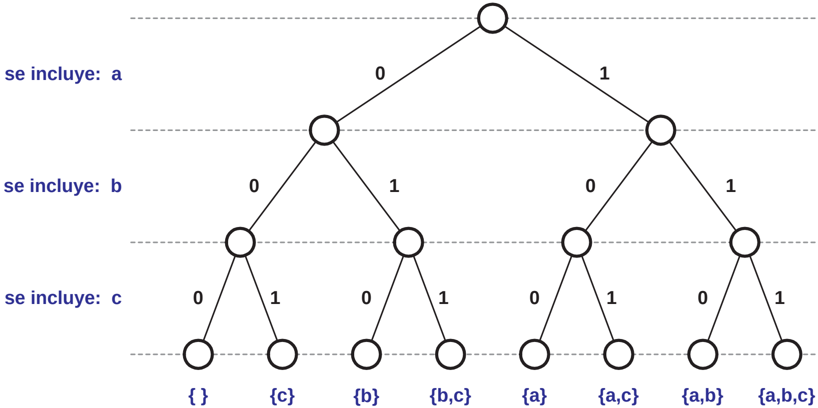
# Partes de $\{a, b, c\}$

## Solución - árbol binario



# Partes de $\{a, b, c\}$

## Solución - árbol binario



# Partes de $\{a, b, c\}$

## Solución - árbol binario

- Árbol binario
- En cada nodo decides si un elemento estará presente o no
- Simplemente crea un vector un vector de valores booleanos
  - $[1, 0, 1] = \{a, c\}$
- No es necesario deshacer los cambios

# Implementación - I

## Solución - árbol binario

```
void partesConj1(int[] c){
  boolean[] subc = new boolean[c.length];

  partes1(c.length, 0, c, subc);
}

void imprimir(int[] c, boolean[] v){
  for(int i=0; i<v.length; i++)
    if(v[i])
      System.out.print(c[i]+" ");

  System.out.println();
}
```

# Implementación - II

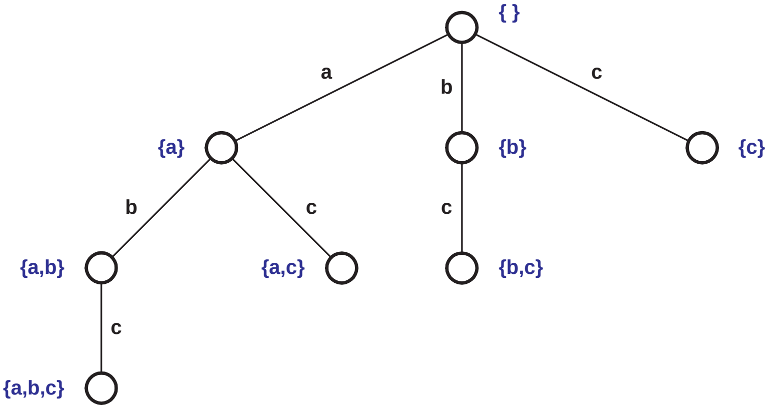
## Solución - árbol binario

```
void partes1(int n, int i, int[] c, boolean[] subc){
  for(int k=0; k<=1; k++){
    if(k==0)
      subc[i] = false;
    else
      subc[i] = true;

    if(i==n-1)
      imprimir(c,subc);
    else
      partes1(n, i+1, c, subc);
  }
}
```

- `n`: número de elementos (profundidad del árbol)
- `i`: elemento a considerar (nivel del árbol)
- `c`: vector de elementos
- `subc`: valores booleanos que definen los subconjuntos

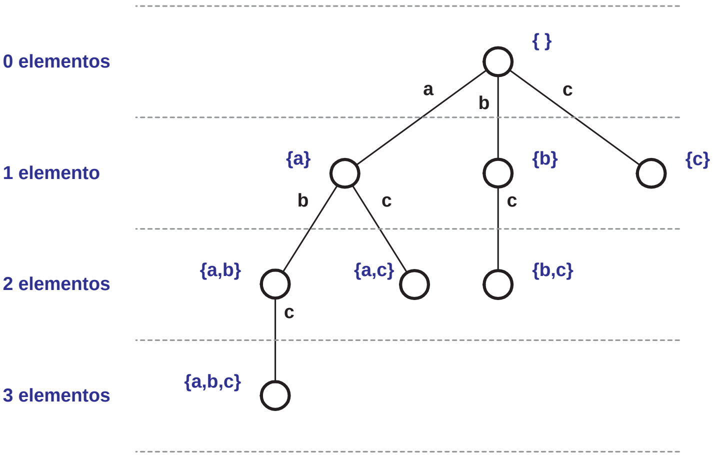
# Partes de $\{a, b, c\}$

## Solución - subconjunto en cada nodo



# Partes de $\{a, b, c\}$

## Solución - subconjunto en cada nodo




- La etiqueta de la rama indica qué elemento a insertar

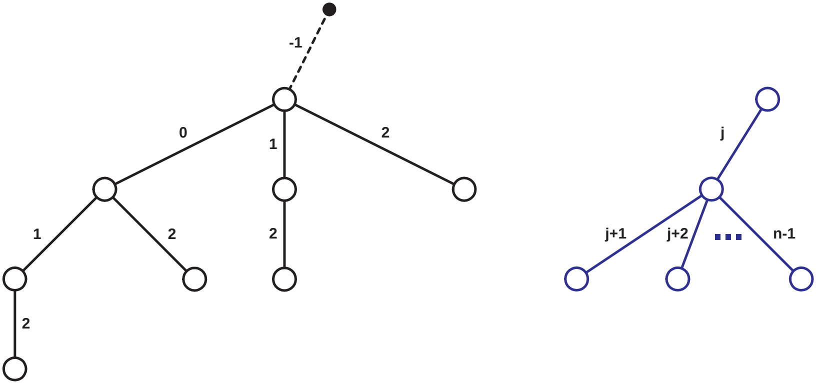
# Partes de $\{a, b, c\}$

## Solución - subconjunto en cada nodo

- El árbol de recursión no es binario
- En el nivel $i$ las soluciones tienen $i$ elementos
- Crea un vector con los índices de los elementos que se van incluyendo
  - $[0, 2] = \{a, c\}$
- Tiene la mitad de nodos (factor constante) que el árbol binario del primer algoritmo
- Se obtiene un subconjunto en cada nodo (no solo en las hojas)

# Partes de $\{a, b, c\}$

## Solución - subconjunto en cada nodo




- Ahora es necesario llevar dos índices
  - Posición en la solución del elemento a incluir
  - Índice del elemento a incluir (qué elemento se incluye)

# Implementación - I

## Solución - subconjunto en cada nodo

```
void partesConj2(int[] c){
  int[] subc = new int[c.length];

  partes2(c.length, 0, -1, c, subc);
}

void imprimir(int[] c, int[] subc, int fin){
  for(int i=0; i<=fin; i++)
    System.out.print (c[subc[i]]+" ");

  System.out.println();
}
```

# Implementación - II

## Solución - subconjunto en cada nodo

```
void partes2(int n, int i, int j, int[] c, int[] subc){
  imprimir(c, subc, i-1);

  for(int k=j+1; k<n; k++){
    subc[i] = c[k];

    partes2(n, i+1, k, c, subc);
  }
}
```

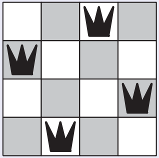
- `n`: número de elementos (profundidad máxima del árbol)
- `i`: posición en la solución del elemento a incluir
- `j`: índice del elemento a incluir (qué elemento se incluye)
- `c`: conjunto (vector) original
- `subc`: subconjuntos hallados

# N reinas

# N reinas

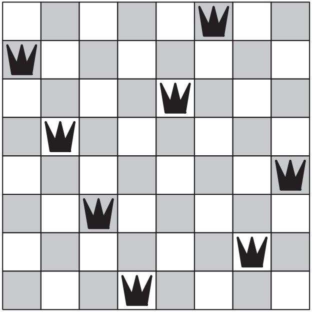
## Problema de las N reinas

- Dado un "tablero ajedrez" de $n \times n$ celdas, se pide ubicar $n$ reinas de modo que no se amenacen
  - No pueden estar en la misma fila
  - No pueden estar en la misma columna
  - No pueden estar en la misma diagonal (principal o secundaria)



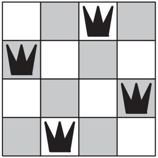
- Ejemplo más famoso de problema que puede resolverse aplicando *backtracking*

# 8 reinas



- Para $n = 8$ hay 92 soluciones posibles
- Aunque 12 únicas
  - Las demás pueden obtenerse aplicando simetrías, rotaciones y traslaciones
- El problema puede solicitar encontrar una solución o todas

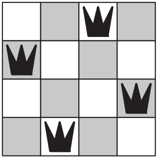
# N reinas



- Solución obvia pero absurda:
  - Probar las $2^{n^2}$ formas de colocar reinas en el tablero
    - $1{,}84 \cdot 10^{19}$ para $n = 8$
    - $65536$ para $n = 4$

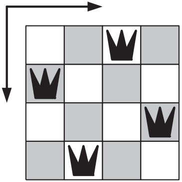
- Pero los conjuntos solución solo deben contener $n$ elementos
  - Esto reduciría nuestro espacio de búsqueda a $\binom{n^2}{n}$
    - $4{,}42 \cdot 10^{10}$ para $n = 8$
    - $1820$ para $n = 4$

# N reinas



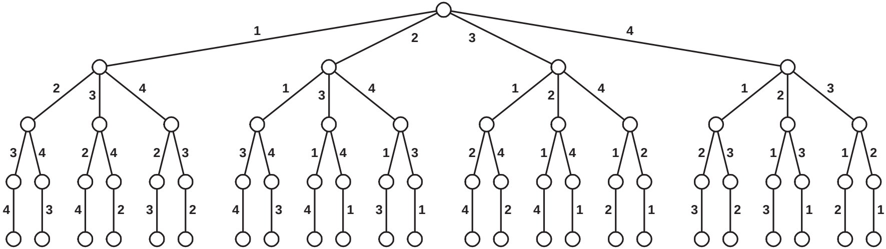
- Además solo puede haber una reina por cada columna:
  - Esto reduce las posibilidades a $n^n$ (hay $n$ formas de colocar una reina en una columna, y hay $n$ columnas)
    - 16777216 para $n = 8$
    - 256 para $n = 4$
- Pero además, no puede haber dos reinas en la misma fila
  - Esto convierte nuestro problema en la búsqueda de una permutación con $n!$ posibilidades (lo cual sigue siendo elevado)
    - 40320 para $n = 8$
    - 24 para $n = 4$

# N reinas



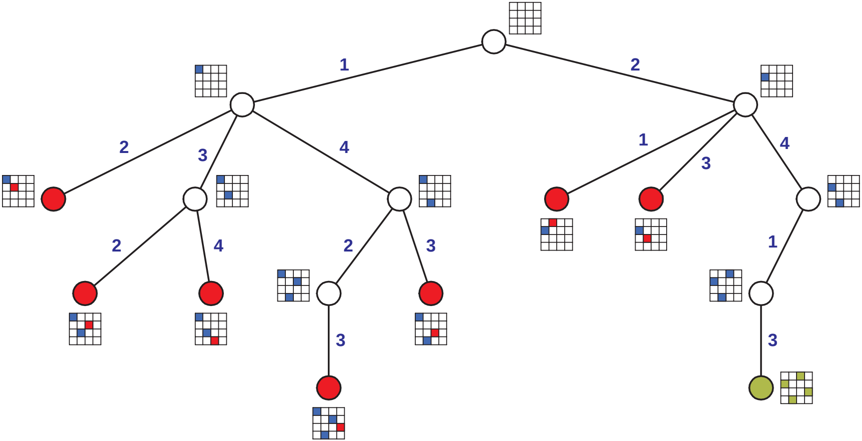
- Formato de la solución:
  - (fila de la columna $1$, fila de la columna $2$,...,fila de la columna $n$)
  - $(2, 4, 1, 3)$ en la figura
  - Todas las filas son diferentes y tienen que estar representadas
  - Tendremos que buscar las **permutaciones** válidas

# Árbol de búsqueda para $N = 4$



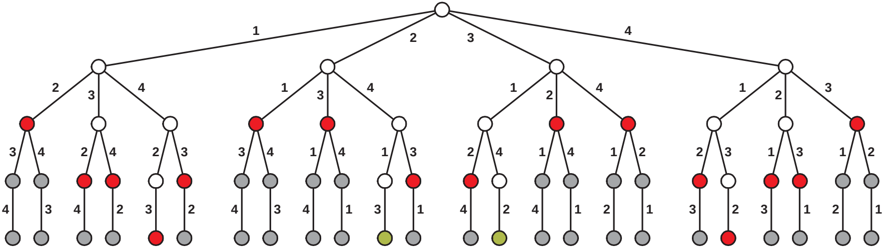

- Podemos emplear un algoritmo para buscar permutaciones
  - Al llegar a una hoja se "han colocado" las cuatro reinas y podemos probar si la solución es válida
  - Pero, podemos comprobar si la solución parcial puede llegar a ser solución final antes de llegar a una hoja
    - Podaríamos el árbol ahorrando cálculos

# Árbol de búsqueda para $N = 4$




- Solo se muestra el árbol hasta encontrar la primera solución válida

# Árbol de búsqueda podado para $N = 4$



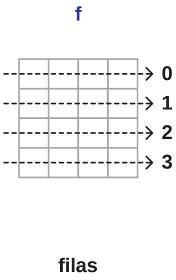

solución parcial válida

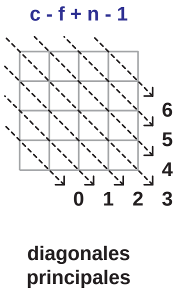
solución completa válida

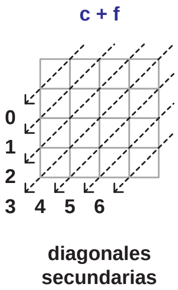
solución no válida

nodo no explorado

# Implementación




- Para verificar que una solución parcial es válida usamos:
  - `f`: filas libres
  - `dp`: diagonales principales libres ($columna - fila + n - 1$)
  - `ds`: diagonales secundarias libres ($columna + fila$)
- `c`: solución parcial

# Implementación - I

```
void n_reinas(int n){
  int[] c = new int[n];

  boolean[] f = new boolean[n];
  for(int i=0; i<n; i++)
    f[i] = true;

  boolean[] dp = new boolean[2*n-1];
  for(int i=0; i<2*n-1; i++)
    dp[i] = true;

  boolean[] ds = new boolean[2*n-1];
  for(int i=0; i<2*n-1; i++)
    ds[i] = true;

  buscarReinas(n, 0, c, f, dp, ds);
}
```

# Implementación - II

```
void buscarReinas(int n, int i, int[] solucion,
                  boolean[] f, boolean[] dp, boolean[] ds){
  for(int j=0; j<n; j++)
    if(f[j] && dp[i-j+n-1] && ds[i+j]){
      solucion[i] = j;

      f[j] = false;
      dp[i-j+n-1] = false;
      ds[i+j] = false;

      if(i==n-1)
        imprimir(solucion);
      else
        buscarReinas(n, i+1, solucion, f, dp, ds);

      f[j] = true;
      dp[i-j+n-1] = true;
      ds[i+j] = true;
    }
}
```

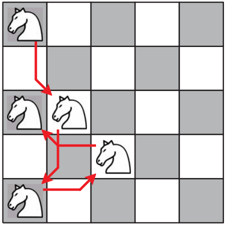
# Implementación - III

- Línea 3: se generan los candidatos
- Línea 4: se comprueba la validez del candidato
- Líneas $5-9$: se incluye el candidato en la solución, y se actualizan las estructuras de datos
- Línea 12: si se ha llegado a una solución válida se imprime
- Línea 14: en caso contrario se sigue buscando
- Líneas $16-18$: se borra el candidato de la solución, y se actualizan las estructuras de datos
  - Al modificar las estructuras de datos no es necesario modificar el vector que contiene la solución parcial

# Otros problemas

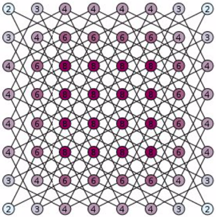
# Salto de caballo

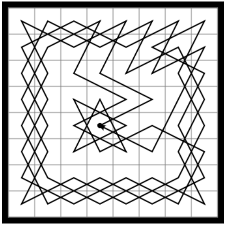
- No es requisito que pueda volver al punto de partida



| | | | | |
|---|---|---|---|---|
| 1 | 14 | 9 | 20 | 23 |
| 10 | 19 | 22 | 15 | 8 |
| 5 | 2 | 13 | 24 | 21 |
| 18 | 11 | 4 | 7 | 16 |
| 3 | 6 | 17 | 12 | 25 |

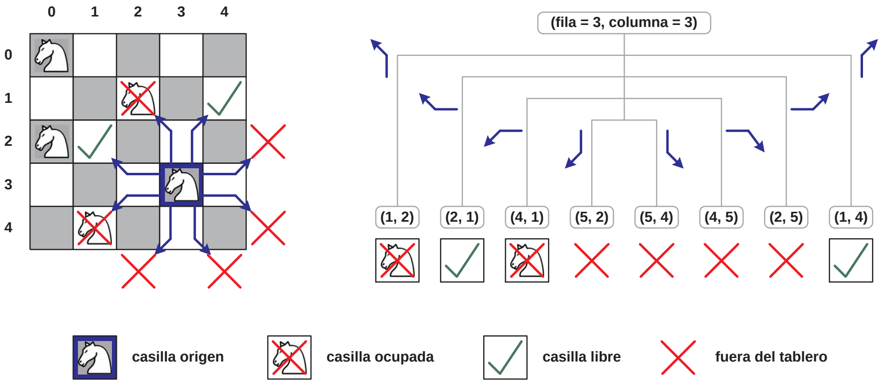
- Variante: **ciclo Hamiltoniano** (camino cerrado en un grafo)





# Salto de caballo

## Árbol de búsqueda




- Se puede encontrar una permutación de $n^2$ elementos (tablero de $n \times n$)
- Se va a seguir otra estrategia de *backtracking*, aunque la solución es similar
- Para tamaños mayores que $6 \times 6$ se puede tardar bastante en hallar una solución

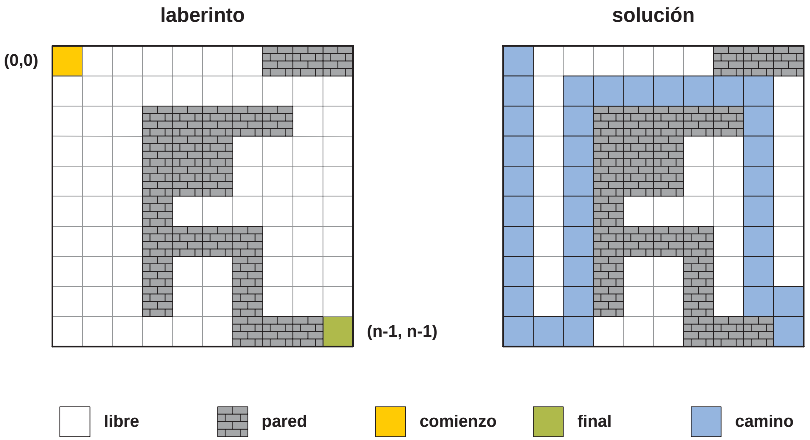
# Implementación - I

```
boolean saltoCaballo(int f, int c, int lado){
  int[][] tablero = new int[lado][lado];
  int[] incrX = new int[] {-1,  1,  2, 2, 1, -1, -2, -2};
  int[] incrY = new int[] {-2, -2, -1, 1, 2,  2,  1, -1};

  tablero[f][c] = 1;
  boolean hay = buscar(tablero.length*tablero.length,
                       tablero.length, 2, f, c, tablero, incrX, incrY);
  if(hay)
    imprimir(tablero);
  return hay;
}

void imprimir(int[][] tablero){
  for(int i=0; i<tablero.length; i++){
    for(int j=0; j<tablero.length; j++)
      System.out.print(((tablero[i][j]<10)?" ":"")+tablero[i][j]+"  ");

    System.out.println();
  }
}
```

# Implementación - II

```
boolean buscar(int n2, int n, int i, int f, int c,
               int[][] recorrido, int[] incrX, int[] incrY){
  boolean exito = false;
  for(int k=0; k<8 && !exito; k++){
    int nuevaF = f + incrY[k];
    int nuevaC = c + incrX[k];
    if(nuevaF>=0 && nuevaF<n && nuevaC>=0 && nuevaC<n)
      if(recorrido[nuevaC][nuevaF] == 0){
        recorrido[nuevaC][nuevaF] = i;
        if(i==n2)
          exito = true;
        else{
          exito = buscar (n2, n, i+1,
                          nuevaF, nuevaC, recorrido, incrX, incrY);
          if(!exito)
            recorrido[nuevaC][nuevaF] = 0;
        }
      }
  }
  return exito;
}
```

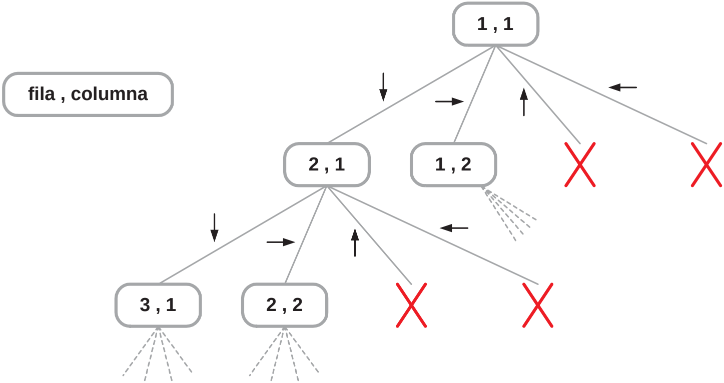
# Laberinto



laberinto
(0,0)
(n-1, n-1)
solución
libre
pared
comienzo
final
camino

# Laberinto

## Árbol de búsqueda




- No se debe hallar una permutación (el camino puede tener duración variable)
- No se debe hallar un subconjunto de celdas (el orden importa)

# Implementación - I

```
boolean hayCamino(char[][] laberinto){
  int[] incrX = new int[] {1, 0, -1,  0};
  int[] incrY = new int[] {0, 1,  0, -1};

  laberinto[0][0] = 'C';
  boolean exito =
          buscar(laberinto.length, 0, 0, laberinto, incrX, incrY);

  imprimir(laberinto);
  return exito;
}

void imprimir(char[][] laberinto){
  for(int i=0; i<laberinto.length; i++){
    for(int j=0; j<laberinto.length; j++)
       System.out.print (laberinto[i][j]+" ");

    System.out.println();
  }
}
```

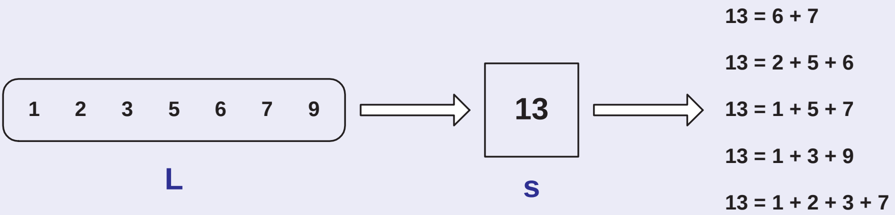
# Implementación - II

```
boolean buscar(int n, int x, int y, char[][] laberinto,
               int[] incrX, int[] incrY){
  boolean exito = false;
  for(int k=0; k<4 && !exito; k++){
    int coordX = x + incrX[k];
    int coordY = y + incrY[k];
    if(coordX>=0 && coordX<n && coordY>=0 && coordY<n)
      if(laberinto[coordY][coordX] == ' '){
        laberinto[coordY][coordX] = 'C';
        if (coordX==n-1 && coordY==n-1)
          exito = true;
        else{
          exito = buscar(n, coordX, coordY, laberinto, incrX, incrY);
          if(!exito)
            laberinto[coordY][coordX] = ' ';
        }
      }
  }
  return exito;
}
```

# Suma de subconjuntos

## Problema de la suma de subconjuntos

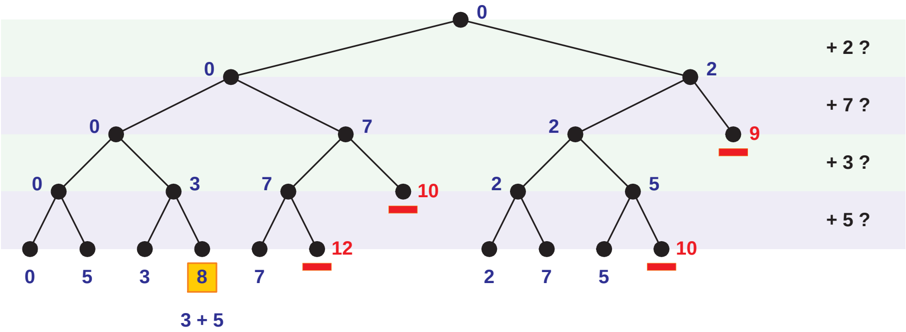
Dada una lista $L$ de números no negativos y otro número $s$, determinar los subconjuntos de $L$ que suman $s$




- Solución con partes de conjuntos (de tamaño fijo)
- La lista $L$ puede tener elementos repetidos

# Suma de subconjuntos

- $L = \langle 2, 7, 3, 5 \rangle, \quad s = 8$
- Se añade una condición para podar el árbol si la suma parcial es mayor que $s$



# Implementación - I

```
void main(String[] args){
  int[] L = new int[] {1, 2, 3, 5, 6, 7, 9};

  sumaSubconjuntos(L, 13);
}

void sumaSubconjuntos (int[] L, int s){
  buscarSubconjs(L.length, 0, 0, L, s, new boolean[L.length]);
}

void imprimir(int[] L, boolean[] v){
  for(int i=0; i<v.length; i++)
    if(v[i])
      System.out.print(L[i]+" ");

  System.out.println();
}
```

# Implementación - II

```
void buscarSubconjs (int n, int i, int p,
                     int[] L, int s, boolean[] subconj){
  for (int k=0; k<=1; k++){
    int nuevoP = p + k*L[i];
    if (nuevoP<=s){
      if(k==0)
        subconj[i]=false;
      else
        subconj[i]=true;

      if(i==n-1){
        if(nuevoP==s)
          imprimir (L, subconj);
      }
      else
        buscarSubconjs(n, i+1, nuevoP, L, s, subconj);
    }
    // nuevoP = p - k*L[i]; lo suprimimos por innecesario
  }
}
```

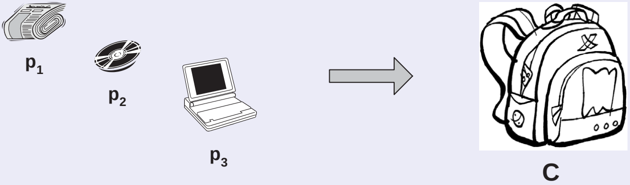
# Problema de la mochila 0-1

## Problema de la mochila 0-1

Dado un conjunto de $n$ objetos, cada uno con un peso $p_i$ y un valor $v_i$, $i = 1, \ldots, n$, y una mochila con capacidad $C$. Maximizar la suma de los valores asociados a los objetos que se introducen en la mochila, sin sobrepasar la capacidad $C$, sabiendo que los objetos **NO** pueden partirse en fracciones más pequeñas:

$$\underset{x}{maximizar} \quad \sum_{i=1}^{n} x_i v_i \qquad \text{sujeto a} \quad x_i \in \{0,1\} \qquad i = 1, \ldots, n$$

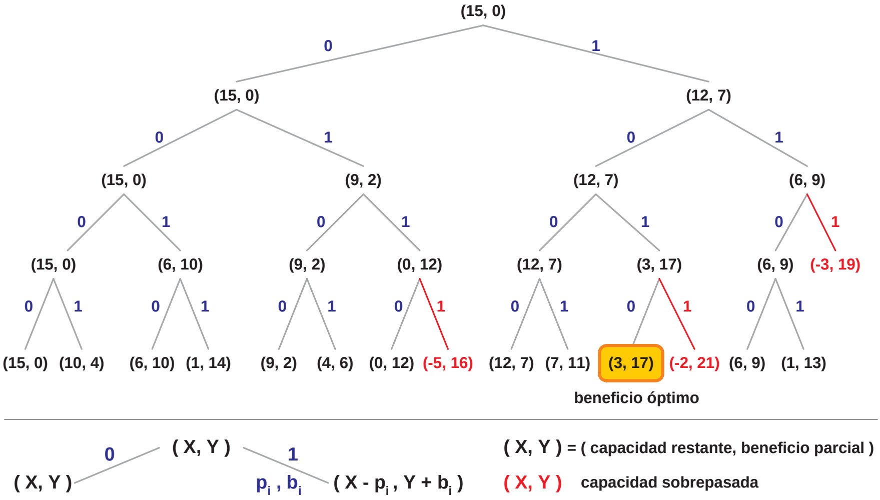
$$\sum_{i=1}^{n} x_i p_i \leq C$$



# Mochila 0-1

## Árbol de búsqueda

- pesos $= (3, 6, 9, 5)$, beneficios $= (7, 2, 10, 4)$, capacidad $= 15$



# Implementación - I

```
void main(String[] args){
  int[] ps= new int[] {3, 6, 9, 5};
  int[] bs= new int[] {7, 2, 10, 4};
  mochila_0_1(ps,bs, 15);
}

int mochila_0_1(int[] ps, int[] bs, int c){
  int[] solParcial = new int[ps.length];
  int[] solOptima = new int[ps.length];
  int bOpt = buscar01(ps.length, 0, c, 0, solParcial,
                      solOptima, -1, ps, bs, c);
  imprimir(solOptima, bOpt);
  return bOpt;
}

void imprimir(int[] v, int bOpt){
  for (int i=0; i<v.length; i++)
    System.out.print(v[i]+" ");

  System.out.println("= "+bOpt);
}
```

# Implementación - II

```
int buscar01(int n, int i, int p, int b, int[] solParc,
              int[] solOpt, int bOpt, int[] ps, int[] bs, int c){
  for(int k=0; k<=1; k++){
    if(k*ps[i]<=p){
      solParc[i] = k;
      int np = p - k*ps[i];   int nb = b + k*bs[i];
      if(i==n-1){
        if(nb>bOpt){
          bOpt = nb;
          for(int j=0; j<ps.length; j++)
            solOpt[j] = solParc[j];
        }
      }
      else
        bOpt = buscar01(n, i+1, np, nb, solParc, solOpt, bOpt, ps, bs, c);
        //int np = p + k*ps[i]; innecesaria
        //int nb = b - k*bs[i]; innecesaria
    }
  }
  return bOpt;
}
```

# Ramificación y poda

# Ramificación y poda

- Hasta ahora podábamos el árbol de recursión si una solución parcial no cumplía los requisitos del problema para llegar a ser una solución completa

- En **problemas de optimización** se pueden realizar podas adicionales:

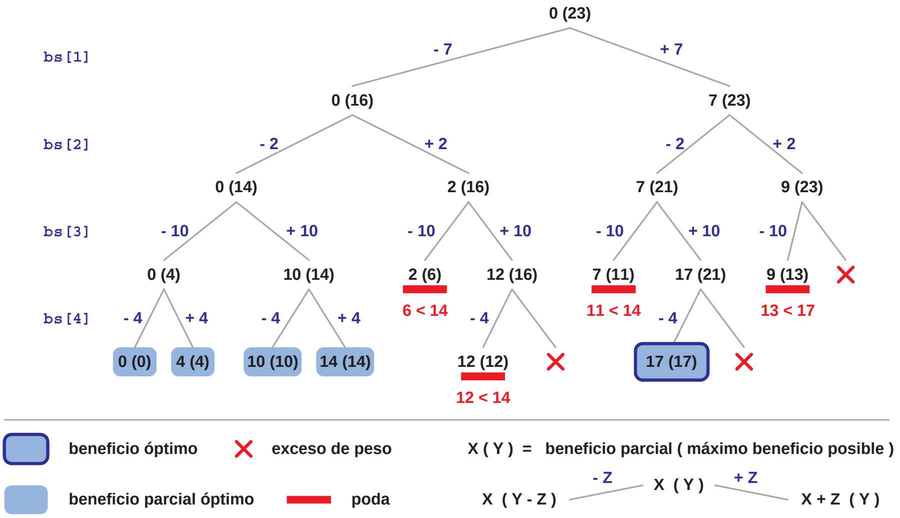
  *Habiendo conseguido hallar una solución válida con valor óptimo $v$, podemos descartar soluciones que nunca puedan alcanzar dicho valor óptimo*

- Es un método exacto de resolución de problemas de optimización

- Se aplica a problemas duros (desde un punto de vista computacional), para acelerar la búsqueda del óptimo global

# Ramificación y poda: Mochila 0-1

- pesos $= (3, 6, 9, 5)$, beneficios $= (7, 2, 10, 4)$, capacidad $= 15$



# Implementación - I

```
void main(String[] args){
  int[] ps= new int[] {3, 6, 9, 5};
  int[] bs= new int[] {7, 2, 10, 4};

  mochila_0_1(ps,bs,15);
}

void mochila_0_1(int[] ps, int[] bs, int c){
  int[] solParcial = new int[ps.length];
  int[] solOptima = new int[ps.length];

  int suma=0;
  for(int i=0; i<bs.length; i++)
    suma += bs[i];

  int bOpt = buscar01poda(ps.length, 0, c, 0, suma, solParcial, solOptima, -1, ps, bs, c);
  imprimir(solOptima, bOpt);
}

void imprimir(int[] v, int bOpt){
  for(int i=0; i<v.length; i++)
    System.out.print(v[i]+" ");

  System.out.println("= "+bOpt);
}
```

# Implementación - II

```
int buscar01poda(int n, int i, int p, int b, int bmax, int[] solParc,
              int[] solOpt, int bOpt, int[] ps, int[] bs, int c){

  for(int k=0; k<=1; k++){
    if(k*ps[i]<=p){
      solParc[i] = k;
      int np = p - k*ps[i];
      int nb = b + k*bs[i];
      int nbmax = bmax - (1-k)*bs[i];

      if(i==n-1){
        if(nb>bOpt){
          bOpt = nb;
          for(int j=0; j<ps.length; j++)
            solOpt[j] = solParc[j];
        }
      }
      else
        if(nbmax>bOpt)
          bOpt = buscar01poda(n, i+1, np, nb, nbmax, solParc, solOpt, bOpt, ps, bs, c);
    }
  }
  return bOpt;
}
```

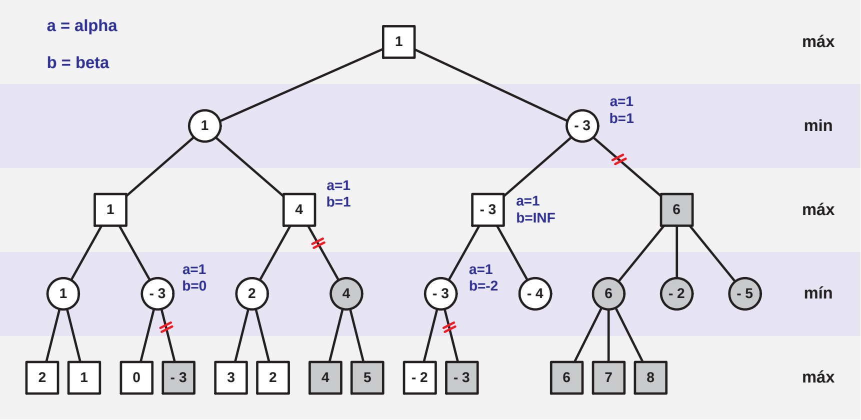
# Poda alfa-beta

# Poda alfa-beta

- Algoritmo minimax
  - Se utiliza en juegos de estrategia entre adversarios (p.e., ajedrez)
  - Cada jugador toma turnos
  - Se evalúa una función numérica acerca de la situación
    - Jaque-mate a favor ($\infty$), en contra ($-\infty$), comer reina contraria (valor muy alto), etc.
  - Un jugador trata de maximizar la función, el otro de minimizarla
- Poda alfa-beta
  - Estrategia de búsqueda para reducir el número de nodos evaluados por el algoritmo minimax

# Poda alfa-beta

- $\alpha =$ cota inferior de los máximos
- $\beta =$ cota superior de los mínimos
- Se poda si $\beta \leq \alpha$
- Se conocen valores heurísticos de los nodos hoja



# Pseudocódigo de la poda alfa-beta

```
alfabeta(nodo, n, a, b, jugador)
    si (n=0) o si el nodo es una hoja
       devuelve el valor heurístico del nodo
    si  jugador = jugadorMax
        para cada hijo del nodo
            a := max(a, alfabeta(hijo, n-1, a, b, not(jugador) ))
            si b=a
                break                             (* Poda Beta *)
        devuelve a
    en caso contrario
        para cada hijo del nodo
            b := min(b, alfabeta(hijo, n-1, a, b, not(jugador) ))
            si b=a
                break                             (* Poda Alfa *)
        devuelve b

(* Llamada inicial *)
n := profundidad del árbol
alfabeta(nodo origen, n, -infinito, +infinito, jugadorMax)
```

# Esquemas generales

# Pasos a seguir para diseñar algoritmos

- Diseño de la representación de la solución y del árbol de búsqueda
- Deducción de las comprobaciones de validez a realizar en cada nodo a partir de las restricciones globales de la solución
- Diseño de estructuras de datos auxiliares para la generación de candidatos o las comprobaciones de validez
- Codificación con ayuda de los **esquemas generales** de código

# Esquema para todas las soluciones - versión 1

```
void buscarTodas (int n, int i, Valor[ ] solucion){
  for(int k=0; k<n; k++){
    <<generar candidato k-ésimo>>;

    if(<<candidato válido>>){
      <<incluirlo en solucion >>;

      if(i==n-1)
        imprimirSolucion(solucion);
      else
        buscarTodas(n, i+1, solucion);

      <<borrarlo de solucion >>;
    }
  }
}
```

- Se ha usado, por ejemplo, en el problema de las N reinas

# Esquema para todas las soluciones - versión 2

```
void buscarTodas(int n, int i, Valor[ ] solucion){
  if(i==n)
    imprimirSolucion(solucion);
  else
    for(int k=0; k<n; k++){
      <<generar candidato k-ésimo>>;

      if(<<candidato válido>>){
        <<incluirlo en solucion>>;

        buscarTodas(n, i+1, solucion);

        <<borrarlo de solucion>>;
      }
    }
}
```

- Esquema alternativo (ahorras comparaciones, se comprueba `i==n` una sola vez)

# Esquema para una solución

```
boolean buscarUna(int n, int i, Valor[ ] solucion){
  boolean exito=false;
  for(int k=0; k<n && !exito; k++){
    <<generar candidato k-ésimo>>;
    if (<<candidato válido>>){
      <<incluirlo en solucion>>;

      if(i==n-1)
        exito = true;
      else{
        exito = buscarUna(n, i+1, solucion);
        if (!exito)
          <<borrarlo de solucion >>;
      }
    }
  }
  return exito;
}
```

# Esquema para la solución óptima

```
int buscarOptima(int n, int i,
                 Valor[ ] solActual, int valorActual,
                 Valor[ ] solOptima, int valorOptimo){
  int valor;
  for(int k=0; k<n; k++){
    <<generar candidato k-ésimo>>;
    if(<<candidato válido>>){
      <<incluirlo en solActual y actualizar valorActual>>;
      if(i==n-1)
        if(valorActual>valorOptimo){
          <<solOptima = solActual>>
          valorOptimo = valorActual;
        }
        else
          valorOptimo = buscarOptima(n, i+1, solActual,
                        valorActual, solOptima, valorOptimo);
          <<borrarlo de solActual y restaurar valorActual>>;
    }
  }
  return valorOptimo;
}
```