Universidad LOYOLA

Departamento de Métodos Cuantitativos

Tema 3. Diseño de Algoritmos

Antonio Manuel Durán Rosal (amduran@uloyola.es)

Programación Avanzada / Programación III

2º de Grado IITV / ISW

Curso 2025-2026

Noviembre 2025

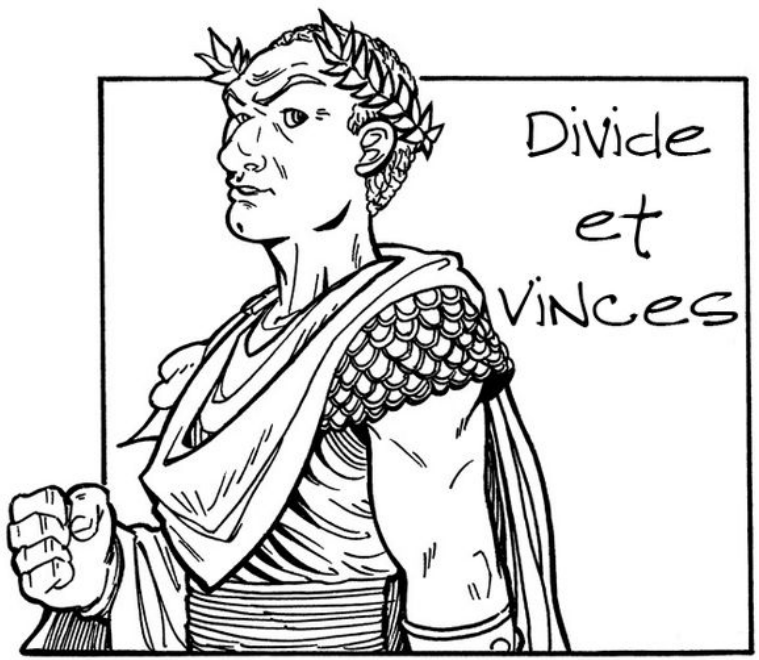
Índice de contenidos

Índice de contenidos

Divide y Vencerás

Divide y Vencerás (DyV) es una técnica de diseño de algoritmos que consiste en:

1. **Descomponer** la instancia del problema a resolver en un conjunto de instancias más pequeñas del mismo problema.
2. **Resolver** independientemente cada una de estas subinstancias.
3. **Combinar** estas soluciones en una solución a la instancia original.



Divide y Vencerás

Es importante determinar para cada problema:

1. **Cuáles son** las subinstancias del problema, y cómo se encuentran.
2. **Cómo solucionar** el problema de las subinstancias: aquí se puede aplicar nuevamente DyV hasta que se llegue a subsintancias de tamaño suficientemente pequeño para ser resueltas inmediatamente.
3. **Cómo combinar** las soluciones parciales.

Esquema algoritmo DyV simple

```
Función DyV (x)
    Si x es suficientemente simple Entonces
        Devuelve algoritmoBásico(x);
    Sino
        descomponer x en x[1], x[2], ..., x[s];
        Para i <-- 1 Hasta s Hacer
            y[i] <-- DyV(x[i]);
        FinPara
        combinar y[i] en una solución y a x;
        Devuelve y;
    FinSi
FinFunción
```

¿Siempre es bueno dividir?

Ejemplo:

$$T_s(n) = \begin{cases} n^2 & \text{si } n < n_0 \\ 3T\left(\frac{n}{2}\right) + 16n & eoc \end{cases}$$

n_0 es el umbral seleccionado a partir del cuál el algoritmo considera que el problema es lo suficientemente pequeño para ser resuelto por un algoritmo básico.

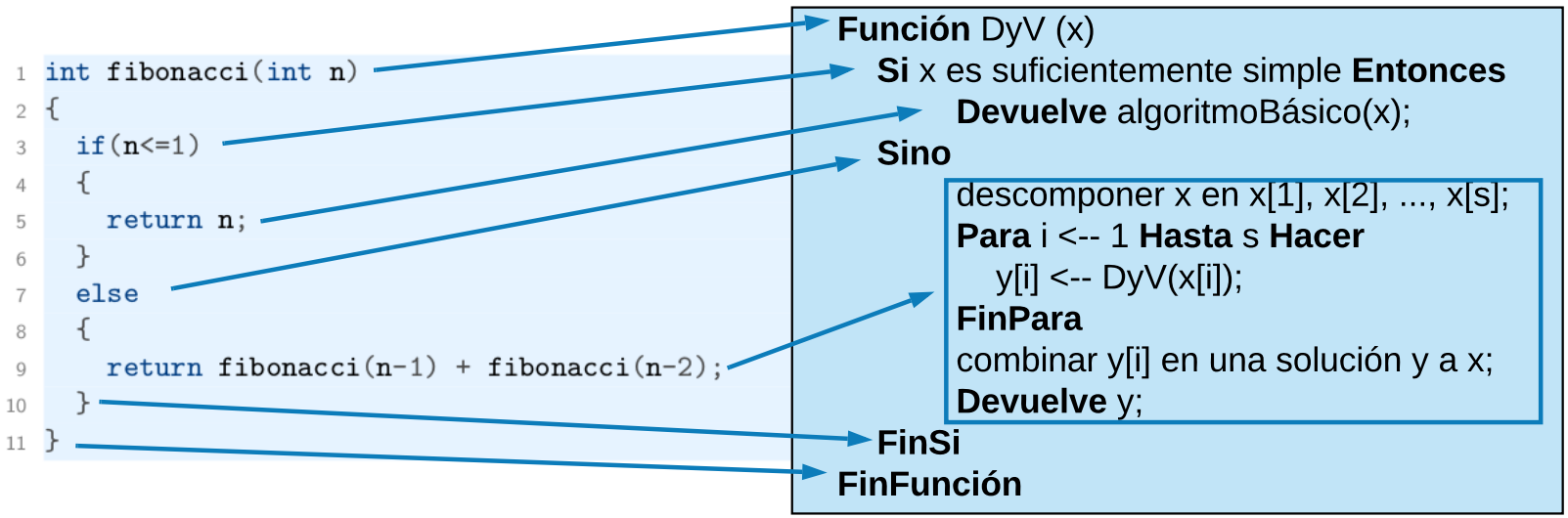
¿Cuál sería el tiempo de ejecución?

¿Siempre es bueno dividir?

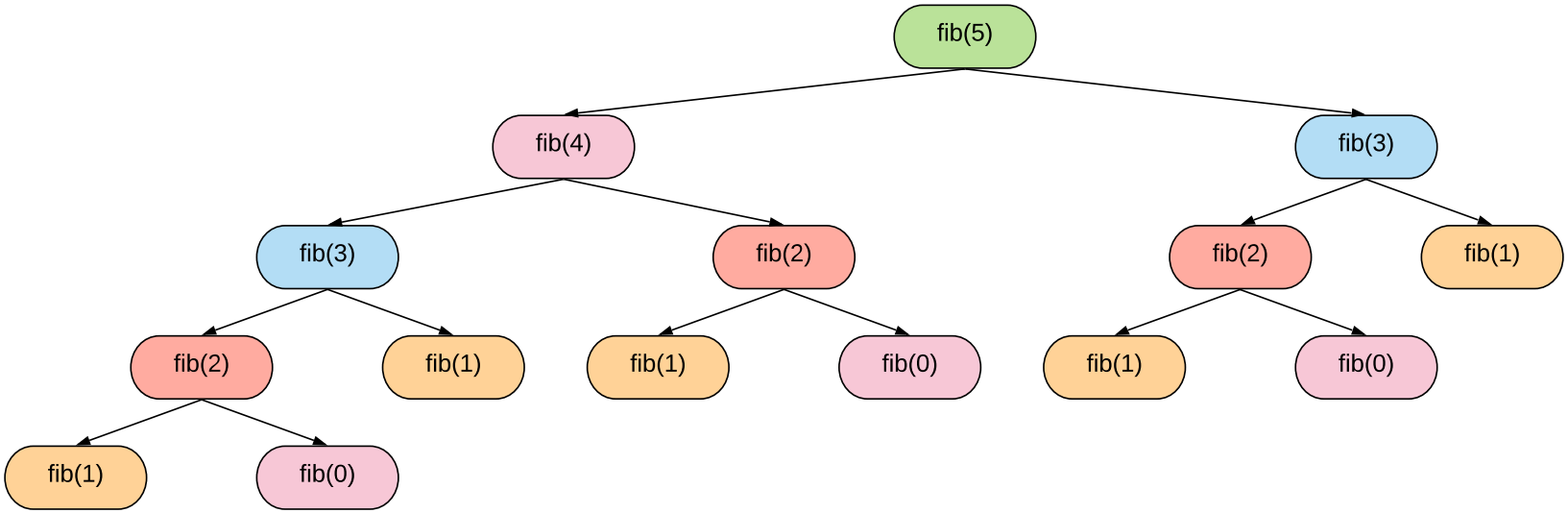
Para que esta técnica merezca la pena aplicarla es necesario que se cumplan tres condiciones:

- Se ha de seleccionar correctamente el umbral, es decir, cuando usar el algoritmo básico en lugar de seguir descomponiendo el problema.
  - Difícil y a veces imposible.
  - Puede cambiar de instancia en instancia.
  - El análisis empírico puede ser costoso.
  - Se debe tomar un valor en el cual tome aproximadamente el mismo tiempo el algoritmo directo que el DyV.
- Ha de ser posible la descomposición del problema en subproblemas y recomponer las soluciones a dichos problemas de una manera eficiente.
- Los subproblemas han de ser aproximadamente del mismo tamaño.

Problema con Fibonacci



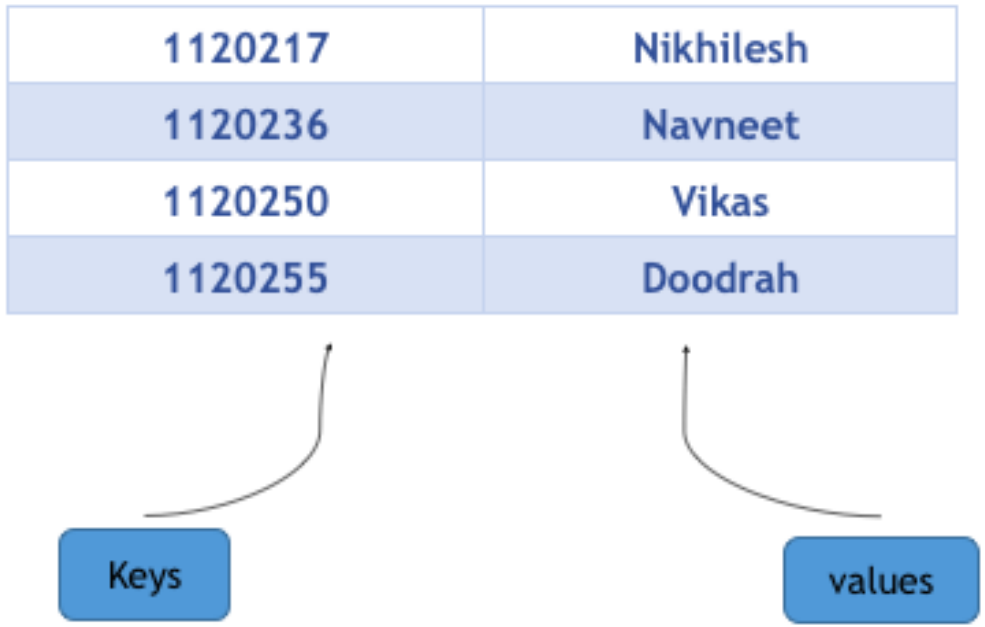
Problema con Fibonacci



Esquema DyV con memoria

```
Función DyV (x)
    Si x está resuelto (guardado en memoria) Entonces
        Devuelve solución y de x (recuperar solución)
    Sino Si x es suficientemente simple Entonces
        y <-- algoritmoBásico(x);
        guardar y en memoria
        Devuelve y;
    Sino
        descomponer x en x[1], x[2], ..., x[s];
        Para i <-- 1 Hasta s Hacer
            y[i] <-- DyV(x[i]);
        FinPara
        combinar y[i] en una solución y a x;
        guardar y en memoria
        Devuelve y;
    FinSi
FinFunción
```

Map de C++



https://en.cppreference.com/w/cpp/container/map

https://en.cppreference.com/w/cpp/container/unordered_map

Ejercicio 1

Implementar las siguientes versiones del algoritmo de **Fibonacci** y comparar los tiempos empleados por cada uno:

a) Recursiva (DyV) sin memoria.

b) Con memoria usando **map** y **unordered_map**.

c) Iterativa.

Ejercicio 2 (1/3)

Multiplicación de enteros grandes: Dados 2 enteros u y v naturales de n dígitos (por simplicidad n será potencia de 2), el algoritmo tradicional de multiplicación es de complejidad $O(n^2)$. Para mejorar dicho tiempo se implementará el método de **Karatsuba**, un algoritmo recursivo que divida los números u y v en dos partes cada uno:

$$u = 10^{n/2}a + b,$$

$$v = 10^{n/2}c + d,$$

siendo a, b, c y d números naturales de $n/2$ cifras. Usando esta división, el producto se calcularía como sigue:

$$uv = (10^{n/2}a + b)(10^{n/2}c + d) = ac10^n + (ad + bc)10^{n/2} + bd.$$

Ejercicio 2 (2/3)

Tenga en cuenta que, para multiplicar ac, ad, bc y bd, se utilizará este mismo algoritmo (de manera recursiva). Puede considerarse que la suma de enteros y la multiplicación por $10^{n/2}$ y 10^n son operaciones de orden lineal.
Para reducir más los cálculos, debe tenerse en cuenta que:

$$(ad + bc) = (a - b)(d - c) + ac + bd,$$

evitando tener que calcular ad y bc, y reutilizando así las multiplicaciones ac y bd que ya se tienen, pero añadiendo el cálculo $(a - b)(d - c)$.

Ejercicio 2 (3/3)

Se pide:

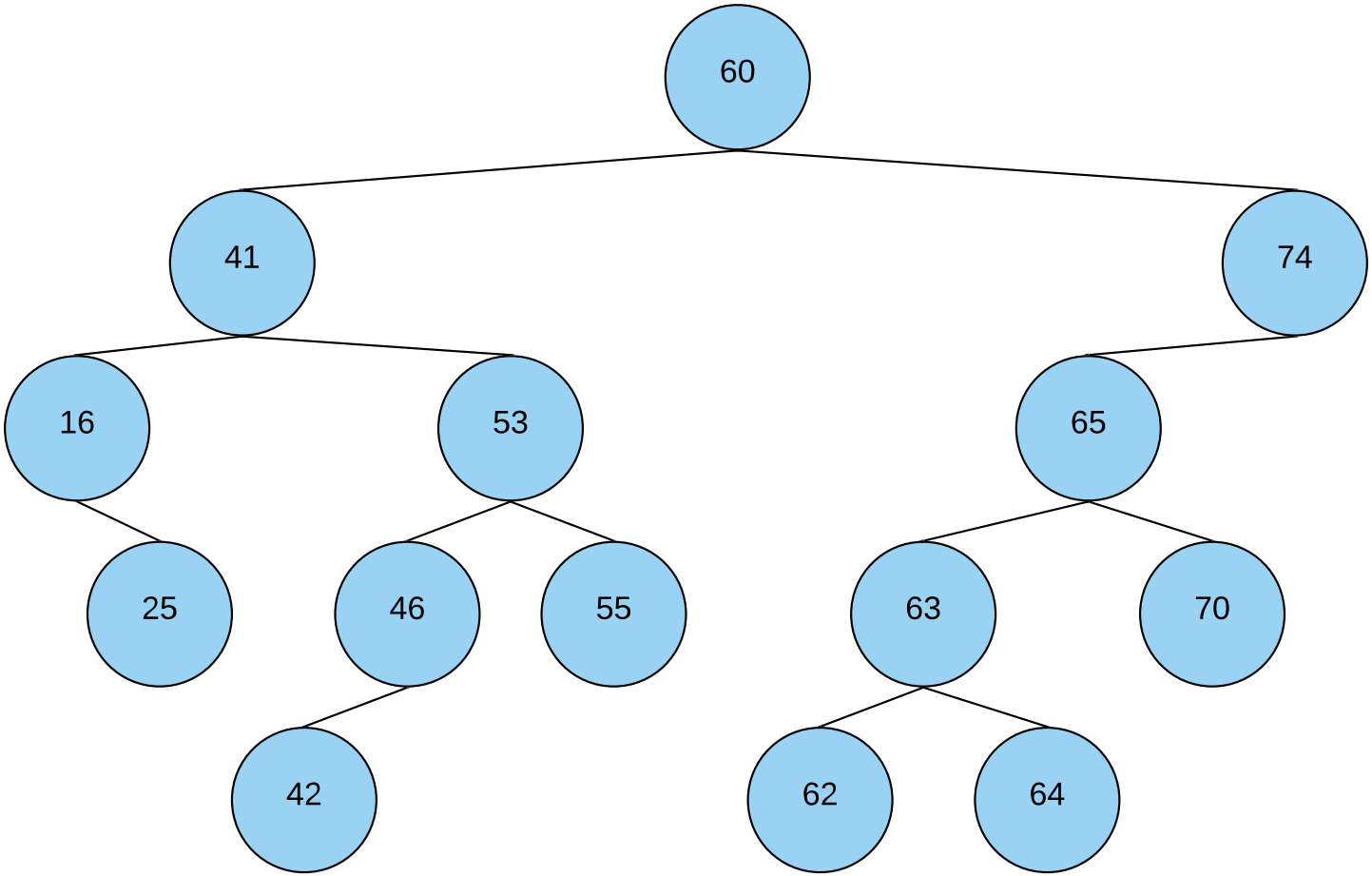
a) Diseñar e implementar el algoritmo de **Karatsuba**.

b) Calcular $T(n)$ y su complejidad.

$$
\begin{array}{rrrrrrr}
 & & & 4 & 1 & 0 & 3 \\
\times & & & & 2 & 5 & 4 \\
\hline
 & & 1 & 6 & 4 & 1 & 2 \\
 & 2 & 0 & 5 & 1 & 5 & \\
 & 8 & 2 & 0 & 6 & & \\
\hline
1 & 0 & 4 & 2 & 1 & 6 & 2
\end{array}
$$

Ejercicio 3 (1/3)

Búsqueda binaria:



Ejercicio 3 (2/3)

Búsqueda binaria: La compañía *MisProductosCIA* dispone de un portal en internet donde se venden diferentes productos de distintos propietarios. La web permite realizar búsquedas que entre otros aspectos, pueden ser filtradas por un rango de precios, es decir, **[precioMenor,precioMayor)**.
MisProductosCIA dispone de un gran sistema de información con miles de productos. Dicho sistema se encuentra **ordenado por precio de menor a mayor** y, además, permite el acceso a un producto concreto, dada su posición, en tiempo constante, es decir, independientemente de la posición del producto en el sistema.
La empresa está teniendo ciertos problemas con el algoritmo anterior, dado que en caso de existir demasiados productos, el tiempo de procesado es muy elevado. Para ello, aprovechando que los productos están ordenados por precios, se desea optimizar el algoritmo de búsqueda anterior empleando la técnica de búsqueda binaria que se describe a continuación.

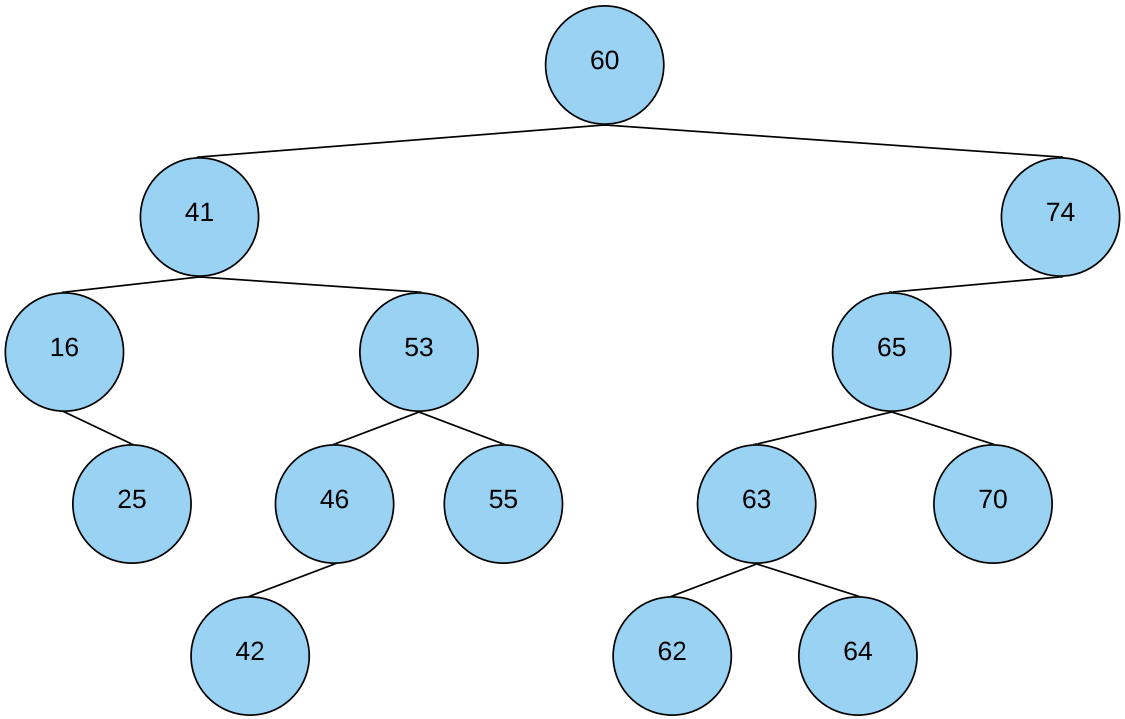
La función **Producto buscaBS(vector <Producto> v, int i, int j, float precio)** debe devolver el primer producto con un precio igual al precio pasado por parámetro en la lista ordenada. Se generaliza utilizando el rango $[i, j]$ dónde se buscará, siendo i la primera posición de la lista y j, la última. **En el caso particular en que el rango sea vacío, es decir, $j < i$ o que no se encuentre el producto se devuelve un objeto vacío.**

Ejercicio 3 (3/3)

Se pide:

a) Diseñar e implementar el algoritmo de **búsqueda binaria**.

b) Calcular $T(n)$ y su complejidad.

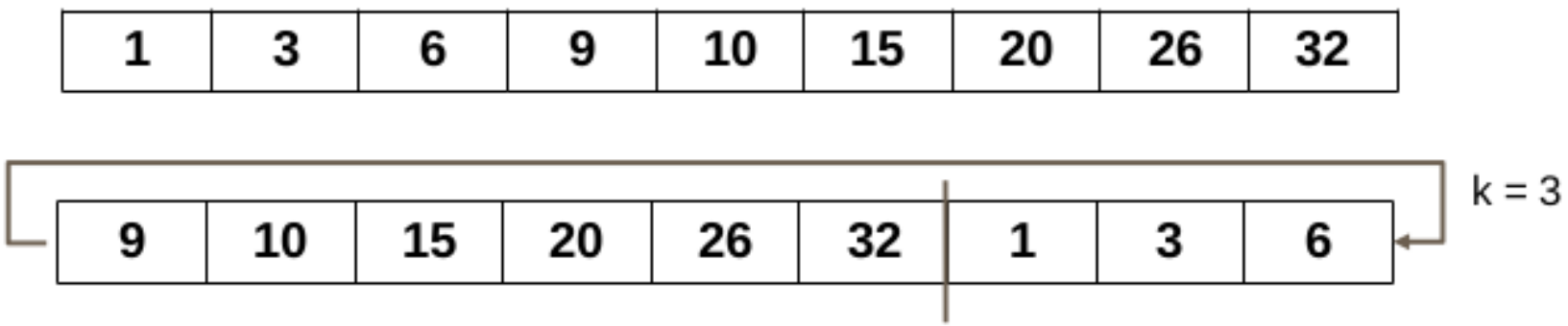


Ejercicio 4

Ejercicio k-ésimo: Dada una lista ordenada que ha sido rotada k veces (k desconocido). Se pide:

a) Diseñar e implementar el algoritmo que encuentre **el elemento mayor**.

b) Calcular $T(n)$ y su complejidad.



Ejercicio 5

Ejercicio máximo y mínimo: Dado un array, encontrar los elementos máximo y mínimo con respecto a un criterio. El problema se generaliza para encontrar el máximo y mínimo entras las posiciones i, j de la lista. Por ejemplo, tenemos un array de personas ordenadas alfabéticamente y queremos buscar al de menor edad y al de mayor edad. Se pide:

a) Diseñar e implementar el algoritmo que encuentre **la persona con edad mínima y la edad máxima**. La clase *Persona* tendrá dos campos: nombre y edad.

b) Calcular $T(n)$ y su complejidad.

Índice de contenidos

Algoritmos voraces

Es una técnica que se caracteriza por:

- **Problema de optimización:** cuya solución se construye partiendo de un conjunto de candidatos, para max/min una **función objetivo**.
- **Dos conjuntos:** a medida que avanzamos se construye un conjunto de candidatos evaluados y seleccionados, y otro de evaluados y rechazados.
- **Función de solución:** comprueba si los candidatos seleccionados constituyen una solución del problema.
- **Función de selección:** indica cual es el mejor de los candidatos restantes.
- **Función objetivo:** proporciona el valor de la solución encontrada.

Algoritmos voraces

Para resolver el problema se buscan un conjunto de candidatos que formen una solución:

1. Inicialmente el conjunto está vacío y sucesivamente se añade el mejor candidato por medio de la función de selección.
2. Si el conjunto no es viable, se rechaza el candidato.
3. Si es viable, se añade como seleccionado y forma parte de la solución final.
4. Una vez añadido se comprueba si el conjunto resultante es una solución del problema.

El nombre voraz proviene del hecho de seleccionar el bocado más apetecible en cada etapa, sin preocuparse por lo que pueda ocurrir más adelante.

Ejercicio 6 (1/2)

El problema del cambio: pagar a una persona una cantidad de dinero con el menor número de monedas posible.

Se pide determinar el número de monedas a devolver para un cambio de **2,94€**.

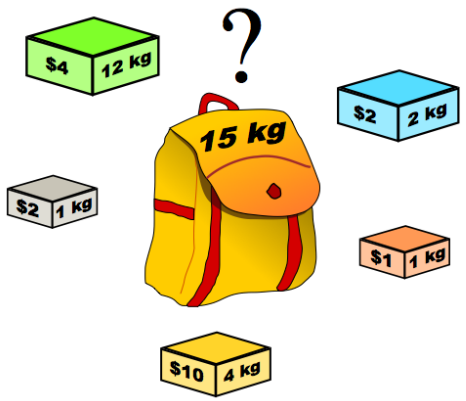
Ejercicio 6 (2/2)

¿Qué ocurre si introducimos una moneda de 4 céntimos?

Se pide determinar el número de monedas a devolver para un cambio de **0,08€**.

Ejercicio 7a (1/3)

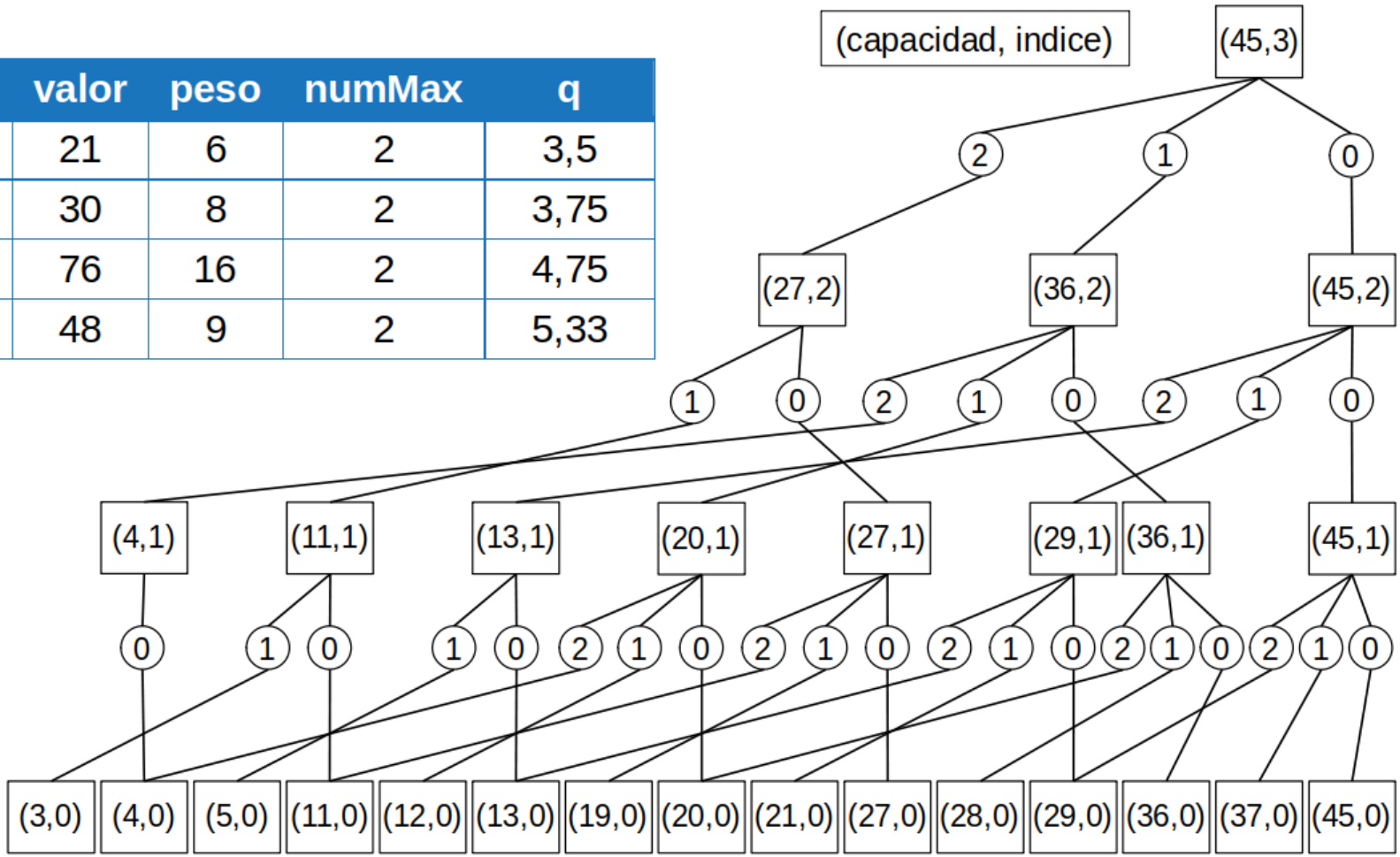
El problema de la mochila: tenemos un conjunto de objetos con un peso y un valor, con un número de ejemplares cada uno. Sea una mochila que puede almacenar un peso máximo:



- Se desea determinar los objetos que deben añadirse en la mochila que maximicen el valor total acumulado.
- La solución buscada es un multiconjunto de los objetos disponibles.

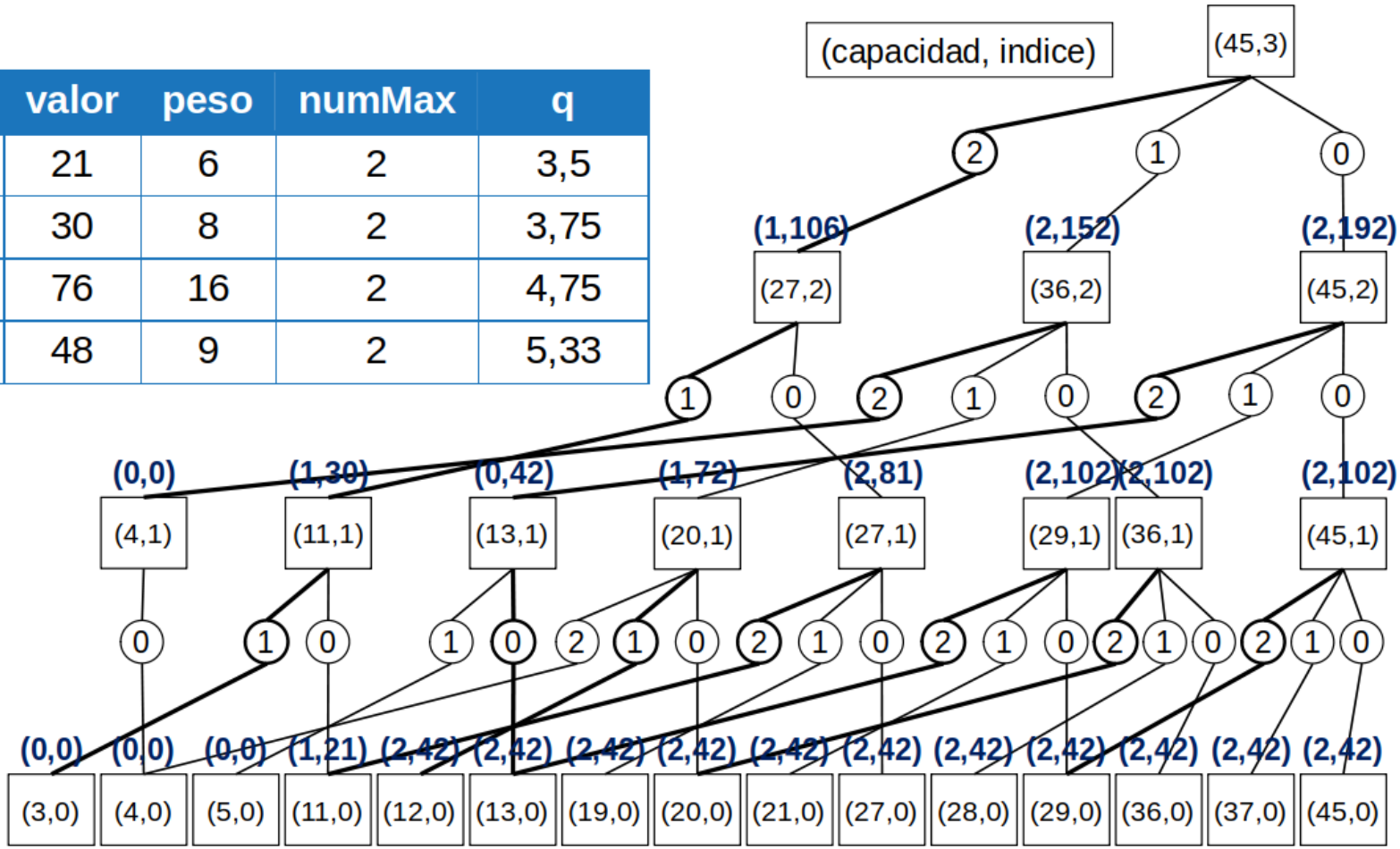
Ejercicio 7a (2/3)

Id	valor	peso	numMax	q
0	21	6	2	3,5
1	30	8	2	3,75
2	76	16	2	4,75
3	48	9	2	5,33



Ejercicio 7a (3/3)

Id	valor	peso	numMax	q
0	21	6	2	3,5
1	30	8	2	3,75
2	76	16	2	4,75
3	48	9	2	5,33



Esquema voraz iterativo

```
Función S voraz (T C)
   R s <-- vacío;
   V val <-- 0;
   Mientras NOT solución(s) AND NOT vacío(C) Hacer
      x <-- selección(C);
      C <-- C - x;
      Si factible(s U x) Entonces
         s <-- s U x;
      Sino
         Descartar candidato x;
      FinSi
      Si solución(s) Entonces
         val = objetivo(s);
      FinSi
   FinMientras
   Devolver (s+val)
FinFunción
```

- T: tipo del problema generalizado.
- S: tipo de la solución (R+V).
- R: tipo conjunto de valores de solución.
- V: valor de la solución con respecto a función objetivo. (Ambos pueden ser tuplas).
- **selección(C):** selección de uno de los candidatos.
- **factible($s \cap u$):** cumple las restricciones.
- **solución(s):** comprueba si es una solución final.
- **objetivo(s):** función que calcula el valor de la solución.

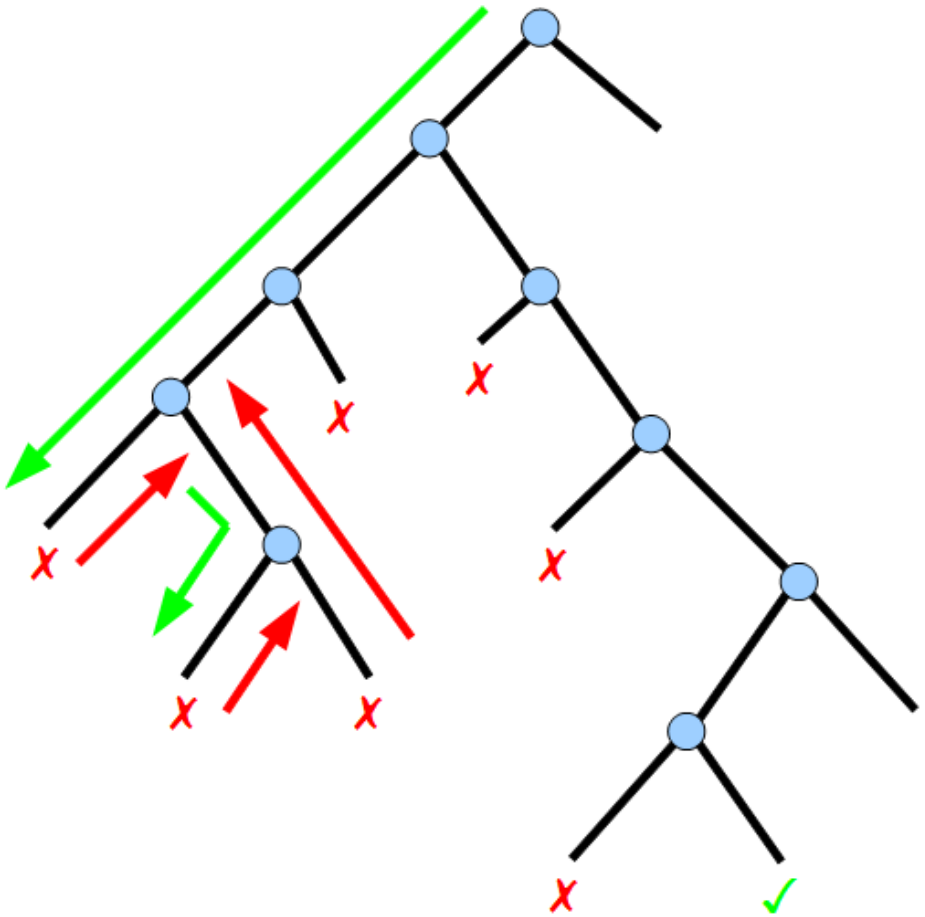
Índice de contenidos

Backtracking

Es una técnica muy útil cuando el cálculo de la solución se puede dividir en etapas o estados:

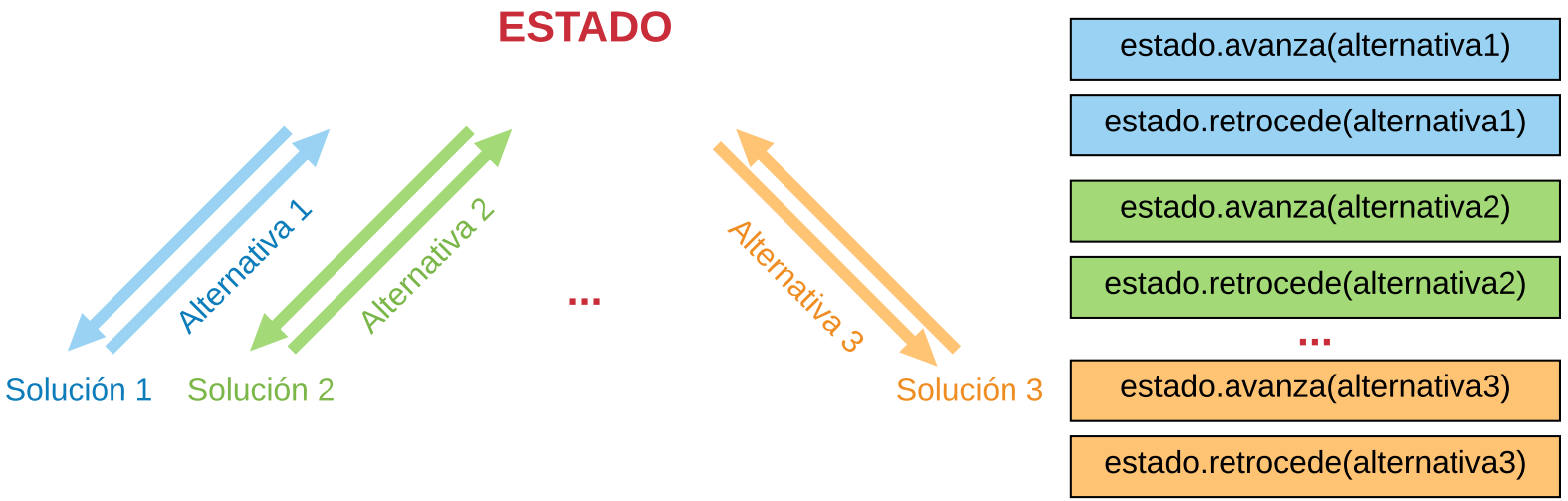
- Los algoritmos ***backtracking*** parten de un vértice (problema a resolver), van eligiendo alternativas y alcanzando nuevos vértices.
- Llamaremos **vértice inicial** al que tiene asociado el problema a resolver y **vértice actual** al alcanzado por el algoritmo en un momento.
- El **estado** guarda el vértice actual y la secuencia de alternativas que tiene como primer elemento la tomada en el vértice inicial y como último elemento la escogida para llegar al vértice actual.
- El estado debe ser diseñado para contener la información sobre el problema actual y la secuencia de alternativas elegidas para llegar hasta él desde el problema inicial.
- Contiene además los métodos necesarios para **añadir** y **eliminar** alternativas.
- El **estado inicial** debe tener información sobre el problema inicial y una secuencia vacía de alternativas escogidas.

Backtracking



Estado

En los algoritmos ***backtracking***, además de añadir, podemos eliminar la última alternativa (esto es para representar la vuelta al vértice padre).



Variante ramifica y poda

En la variante **ramifica y poda**, el algoritmo **filtra (poda)** el **conjunto de alternativas** disponible usando información de las decisiones tomadas y cotas del valor esperado.

Esquema *backtracking*

```
Función bt (E *e)
    Si e->esFinal() Entonces
        actualizaMejorSolucion(e);
    Sino
        P p <-- problema actual;
        list <A> alts <-- e->getAlternativas(p);
        Para cada alt en alts Hacer
            e->avanza(alt);
            bt(e);
            e->retrocede(alt);
        FinPara
    FinSi
FinFunción
```

Tipos:

- *E*: tipo del estado.
- *P*: tipo del problema actual (puede ser una tupla de valores).
- *A*: tipo de las alternativas.

Métodos del problema:

- **actualizaMejorSolucion(*e*):** actualiza la mejor solución encontrada hasta el momento.

Métodos del estado:

- **esFinal():** si se ha llegado a un estado final.
- **getAlternativas(P *p*):** devuelve las alternativas del problema actual.
- **avanza(Alt *alt*):** actualiza el estado añadiendo la alternativa *alt*.
- **retrocede(A *alt*):** actualiza el estado quitando la alternativa *alt*.

Esquema voraz recursivo

```
Función bt (E *e)
   Si e->esFinal() Entonces
         actualizaMejorSolucion(e);
   Sino
         P p <-- problema actual;
         list <A> alts <-- e->getAlternativas(p);
         A alt <-- e->getMejorAlternativa(p);
         Para cada alt en alts Hacer
            e->avanza(alt);
            bt(e);
            e->retrocede(alt);
         FinPara
   FinSi
FinFunción
```

Tipos:

- *E*: tipo del estado.
- *P*: tipo del problema actual (puede ser una tupla de valores).
- *A*: tipo de las alternativas.

Métodos del problema:

- **actualizaMejorSolucion(*e*):** actualiza la mejor solución encontrada hasta el momento.

Métodos del estado:

- **esFinal():** si se ha llegado a un estado final.
- **getMejorAlternativa(P *p*):** devuelve la mejor alternativa del problema actual.
- **avanza(Alt *alt*):** actualiza el estado añadiendo la alternativa *alt*.

Esquema ramifica y poda

```
Función bt (E *e)
   Si e->esFinal() Entonces
         actualizaMejorSolucion(e);
   Sino
         P p <-- problema actual;
         list <A> alts <-- e->getAlternativas(p);
         Para cada alt en alts Hacer
            e->avanza(alt);
            bt(e);
            e->retrocede(alt);
         FinPara
   FinSi
FinFunción
```

Tipos:

- *E*: tipo del estado.
- *P*: tipo del problema actual (puede ser una tupla de valores).
- *A*: tipo de las alternativas.

Métodos del problema:

- **actualizaMejorSolucion(*e*):** actualiza la mejor solución encontrada hasta el momento.

Métodos del estado:

- **esFinal():** si se ha llegado a un estado final.
- **getAlternativas(P *p*):** devuelve las alternativas del problema actual que **alcancen la cota**.
- **avanza(Alt *alt*):** actualiza el estado añadiendo la alternativa *alt*.
- **retrocede(A *alt*):** actualiza el estado quitando la alternativa *alt*.

¿Qué pretendemos?

La estructura de nuestros algoritmos de ***Backtracking*** y **Voraz** en su versión recursiva tendrá cuatro clases:

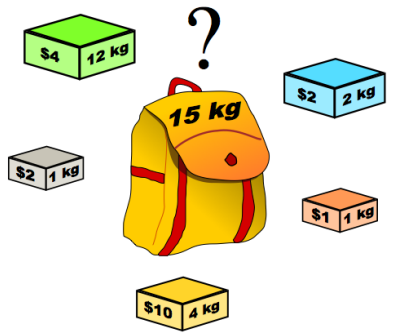
Representación de un **OBJETO** del problema	Representación de una **SOLUCIÓN** del problema	Representación de un **ESTADO** del problema	Representación del **PROBLEMA**
Métodos para el tratamiento del **OBJETO** del problema	Métodos para el tratamiento de la **SOLUCIÓN** del problema	esFinal getAlternativas getMejorAlternativa avanza retrocede	ejecutaBacktracking bt ejecutaVoraz vorazR actualizaMejorSolución

Índice de contenidos

Ejercicio 7

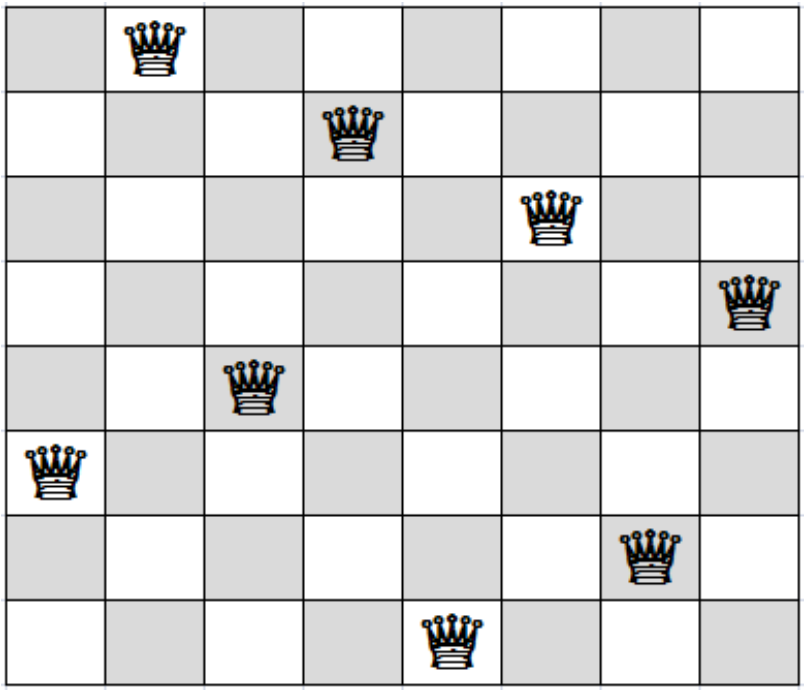
Problema de la mochila

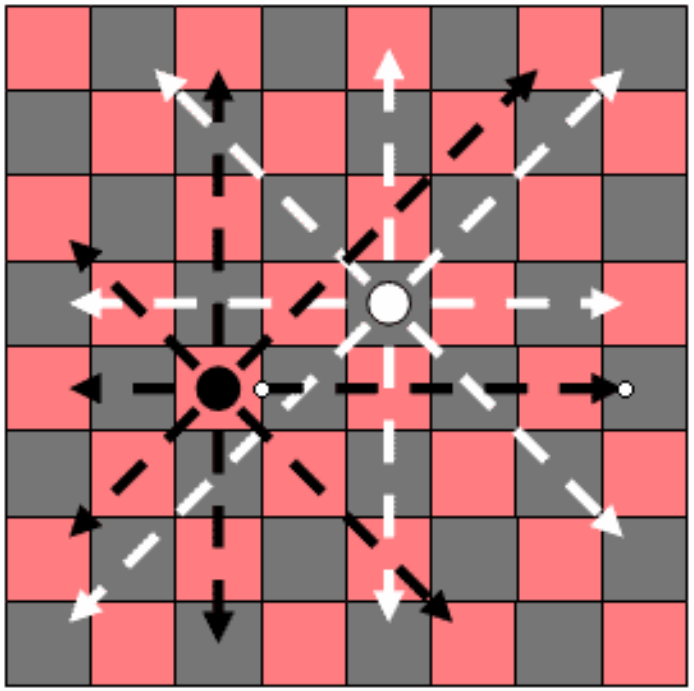
nombre valor peso num_disponibles q = v/p	lista de objetos valor total peso total	Solución parcial Capacidad restante Objetos por consultar	Objetos disponibles Capacidad total Solución
Métodos para el tratamiento del **OBJETO** de la mochila	Métodos para el tratamiento de la **SOLUCIÓN** del problema de la mochila	esFinal getAlternativas getMejorAlternativa avanza retrocede	ejecutaBacktracking bt ejecutaVoraz vorazR actualizaMejorSolución



Ejercicio 8 (1/2)

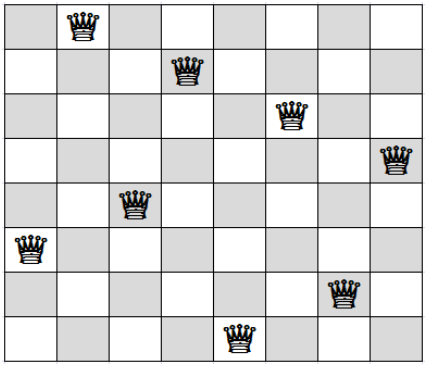
Problema de las *n*-reinas: encontrar todas las soluciones para la ubicación de n reinas en un tablero de ajedrez de tamaño $n * n$ sin que las reinas entren en conflicto entre sí.





Ejercicio 8 (2/2)

fila columna diagonal principal diagonal secundaria	lista de reinas diagonalesP usadas diagonalesS usadas columnas usadas	Solución parcial Número de reinas restantes Número total de reinas	Lista de soluciones Número total de reinas Número de soluciones
Métodos para el tratamiento de la **REINA** del tablero	Métodos para el tratamiento de la **SOLUCIÓN** del problema de las reinas	esFinal getAlternativas getMejorAlternativa avanza retrocede	ejecutaBacktracking bt ejecutaVoraz vorazR actualizaMejorSolución



Ejercicio 9

Problema del cambio

valor moneda cantidad disponible	lista de monedas valor total número de monedas	Solución parcial Número de monedas restantes Valor restante	Monedas disponibles Valor a devolver Mejor solución
Métodos para el tratamiento de la **MONEDA** del cambio	Métodos para el tratamiento de la **SOLUCIÓN** del problema del cambio	esFinal getAlternativas getMejorAlternativa avanza retrocede	ejecutaBacktracking bt ejecutaVoraz vorazR actualizaMejorSolución

Referencias

Recursos electrónicos

- cppreference. (2025). Referencia C++: https://en.cppreference.com/w/.
- Goalkicker.com (2025). C++: Note for Professionals.
- Goalkicker.com (2025). Algorithms: Notes for Professionals.
- Joyanes, L. (2006). Programación en C++: Algoritmos, Estructuras de datos y Objetos.

Referencias

Libros:

- Brassard, G., Bratley, P. (1997). Fundamentos de Algoritmia.
- Guerequeta, R., Vallecillo, A. (1998). Técnicas de diseño de algoritmos.
- Joyanes, L. (2008). Fundamentos de programación: algoritmos, estructura de datos y objetos.
- Knuth, D. (2011). The Art of Computer Programming, Volumes 1-4. Third Edition.
- Martí, N., Ortega, Y., Verdejo, J.A. (2013). Estructuras de datos y métodos algorítmicos: Ejercicios resueltos.
- Parker, A. (1993). Algorithms and Data Structures in C++. CRC Press.

¿Preguntas?