Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Tema 11. Esquema Voraz

Algorítmica y Complejidad

Conceptos básicos

- Es uno de los esquemas más simples y al mismo tiempo de los más utilizados.
  - Son sencillos de implementar y requieren una mínima cantidad de recursos.
- Típicamente se emplean para resolver problemas de optimización en los que se pretende obtener un subconjunto de elementos que satisfagan ciertas restricciones y que optimicen alguna medida.
  - Aunque no siempre obtienen la mejor solución, son utilizados para obtener aproximaciones a la solución óptima.
  - Tienen una complejidad media aceptable.

*también conocidos como ávidos o avaros (*greedy* en inglés)

Conceptos básicos

- Análisis de complejidad:

  Un algoritmo voraz típicamente realiza n elecciones para un problema de tamaño n. Por tanto, el tiempo esperado de ejecución es:

  – $\Theta(n*\Theta$ (elegir(n)) dónde elegir(n) es una función que permite elegir un elemento entre n objetos.

  – Ordenes habituales son $\Theta(n)$, $\Theta(n \log n)$ y $\Theta(n^2)$.

Conceptos básicos

- La mayoría de los problemas para los que producen buenos resultados tienen dos propiedades:
  - La *propiedad voraz*:
    - Se basa en tomar en cada momento la mejor de las opciones posibles sin pensar en futuras consecuencias.
    - Se confía en que tomando un valor óptimo *local* en cada paso, se llegue a una solución óptima *global*.
    - Para comprobar que una selección voraz a cada subproblema produce una solución óptima global se recurre a demostraciones por inducción (proceso no trivial).
  - Una subestructura óptima:
    - Cuando la solución óptima al problema contiene soluciones óptimas a los subproblemas.

Conceptos básicos

- El esquema voraz es un proceso repetitivo: se basa en ir formando la solución de forma incremental, analizando en cada paso cuál de todos los elementos disponibles es el que más interesa añadir (en función de la *propiedad voraz*).
  - Dado un conjunto finito de entradas C, un algoritmo voraz devuelve un subconjunto S (seleccionados) tal que $S \subset C$ y que además cumple con las restricciones del problema inicial.
  - Cada subconjunto S que satisfaga las restricciones se le suele denominar prometedor, y si éste además logra que la función objetivo se minimice o maximice (según corresponda) diremos que S es una solución óptima.

Conceptos básicos

- Componentes:
  - El conjunto C de candidatos, entradas del problema.
  - El conjunto S de seleccionados, elementos de la solución.
  - Función solución. Comprueba si el subconjunto de seleccionados forma una solución (no importa si es óptima o no lo es).
  - Función de validez o factibilidad. Indica si el candidato elegido (el elemento más prometedor) puede formar parte de la solución en las condiciones actuales del problema.
  - Función de selección. Determina cuál es el elemento más prometedor para completar la solución. Este elemento no puede haber sido rechazado o escogido con anterioridad.
  - Función objetivo. Devuelve la bondad de la solución hallada. Generalmente es la función que tratamos de optimizar. Se utiliza para implementar la función selección.

Conceptos básicos

- Funcionamiento:
  1. Inicialmente el conjunto de candidatos seleccionados está vacío.
  2. En cada paso, se intenta añadir a este conjunto "el mejor" de los candidatos disponibles, utilizando una función de selección.
  3. Tras cada paso, hay que ver si el candidato seleccionado es factible.
     - si el candidato no es factible, se rechaza y no se vuelve a considerar en el futuro;
     - si es factible, se incorpora al conjunto de escogidos y permanece siempre en él;
  4. Tras cada incorporación se comprueba si el conjunto resultante es una solución.
  5. El algoritmo termina cuando se obtiene una solución o cuando no quedan candidatos.

Esquema general

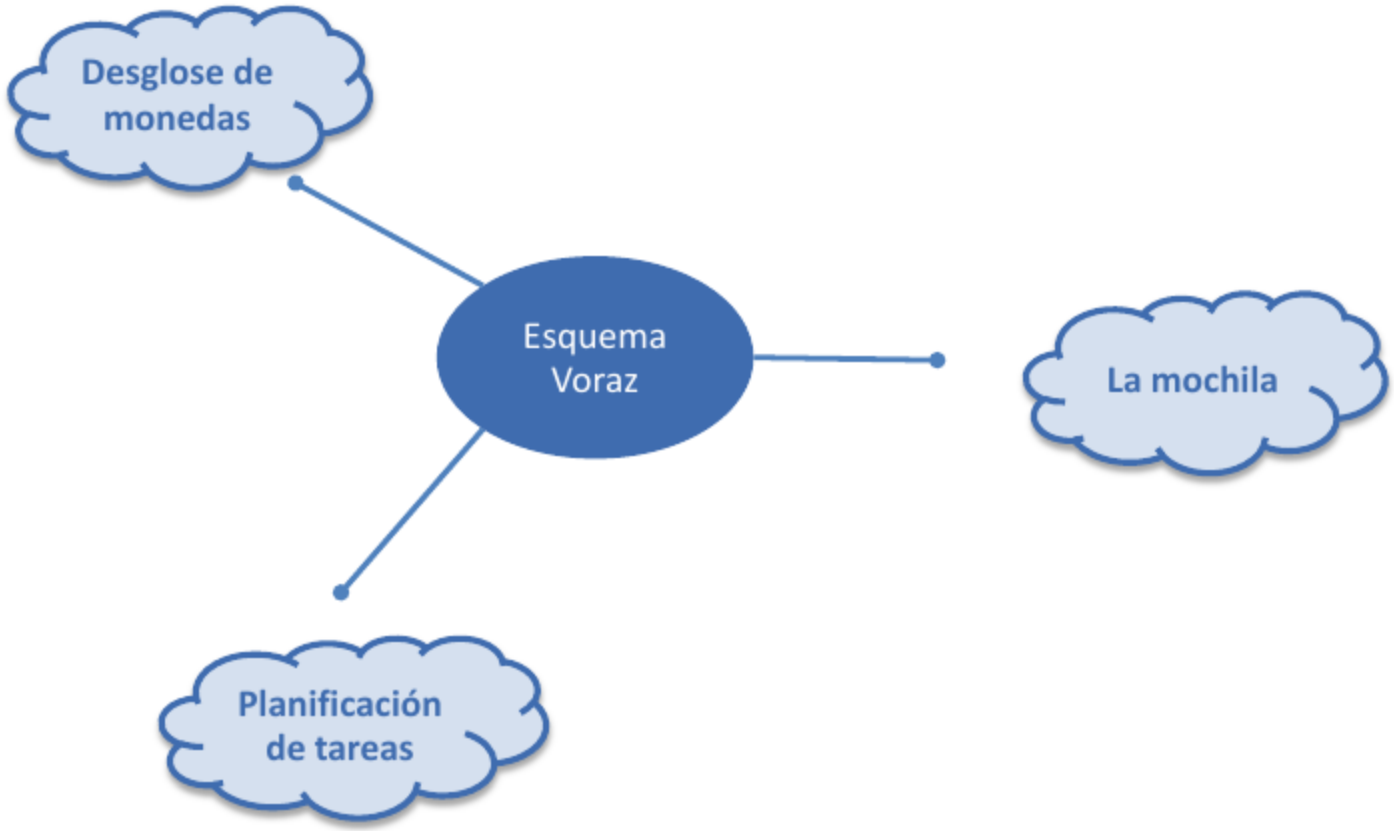
```
public ArrayList<Candidato> voraz(ArrayList<Candidato> candidatos)
{
  inicializarSolucion(solucion);
  while (!esSolucion(solucion) && (quedanCandidatos(candidatos)){
    candidato = seleccionarCandidato(candidatos);
    eliminarCandidato(candidato,candidatos);
    if (esValido(candidato,solucion)) {
       anyadirCandidato(candidato,solucion);
    }
  }
  if (esSolucion(solucion)) {
     return solucion;
  } else {
    return null;
  }
}
```

Esquema general

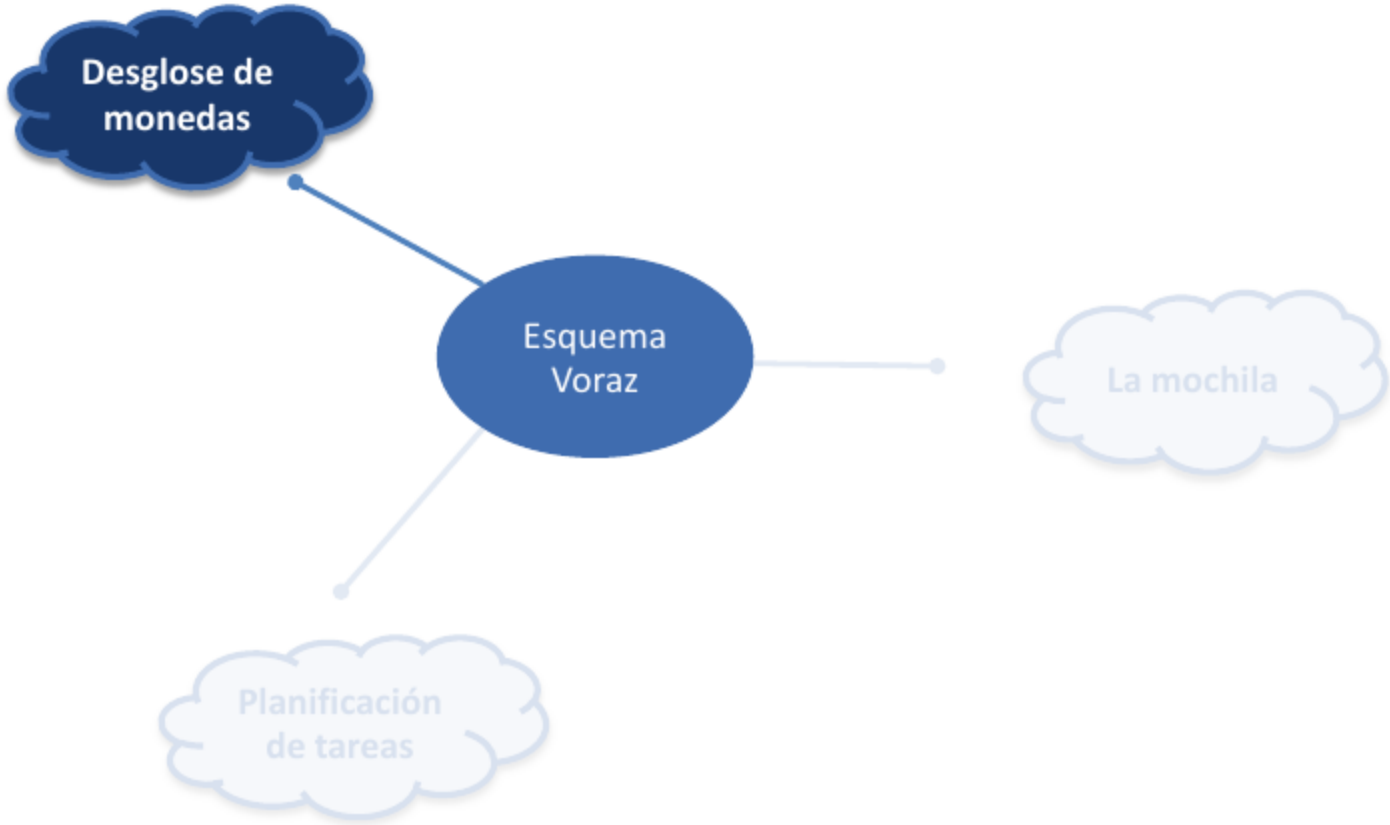
PASOS:

- Paso 1: Determinar las estructuras para representar el *ConjuntoCandidatos* y *ConjuntoSolucion*, fijando cómo se han de introducir/eliminar los candidatos de ellas.
- Paso 2: Estudiar y definir la función *seleccionarCandidato*
- Paso 3: Determinar la función *esValido*.
  - Esta es posiblemente la operación más delicada.
- Paso 4: Determinar *esSolucion* y *quedanCandidatos*.
  - Comprobar las condiciones extremas de finalización del bucle (cuando no hay solución, cuando se acaban los candidatos) de forma que no se produzca un bucle infinito.

Algunos problemas planteados



Algunos problemas planteados



Enunciado

- Se dispone de un sistema monetario con suficientes monedas de cada tipo.
- Y un importe a devolver.
- Se pretende obtener un desglose de monedas para pagar el importe, de forma que el número de monedas empleado sea mínimo.
  - Sea $V=(v_1,v_2,\ldots,v_n)$, sistema monetario
  - Sea C importe a devolver
  - Sea $X=(x_1,x_2,\ldots,x_n)$, desglose de monedas
  - Restricciones: $\Sigma\ x_i \,.\, v_i = C$

Implementación

- Enfoque voraz: Seleccionar la moneda de mayor valor

  Si disponemos del sistema monetario ordenado de mayor a menor valor

```
// candidatos
  int[] valores= {50, 20, 10, 5, 2, 1};
```

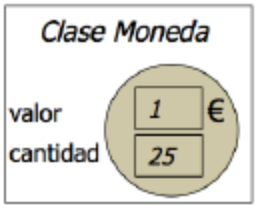
```
int[] numMonedasOrdenadas(int[] valores, int cantidad){
  int[] cant = new int[valores.length];
  int v = cantidad;
  for (int i=0; i<valores.length; i++) {
      cant[i]=v/valores[i];
      v = v % valores[i];
  }
  return cant;
}
```

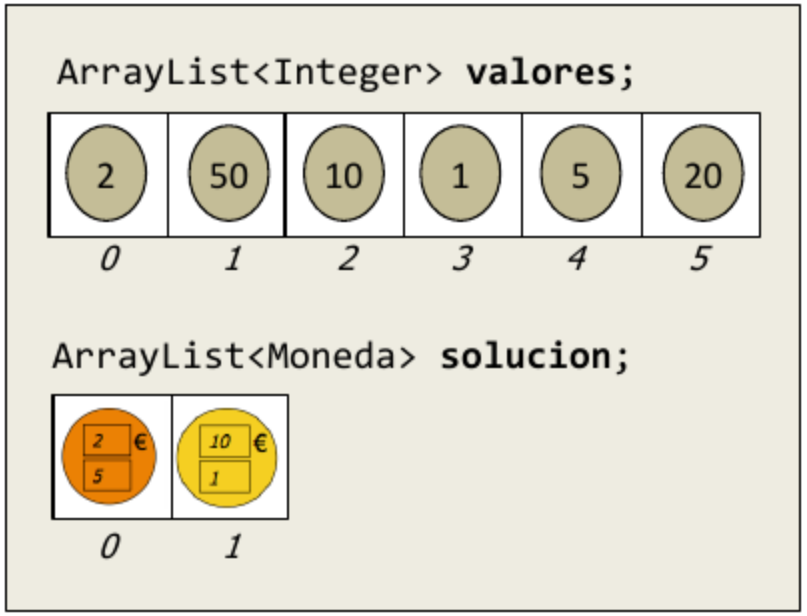
Implementación

- Enfoque voraz: Seleccionar la moneda de mayor valor

  No disponemos del sistema monetario ordenado de mayor a menor valor

```
public class Moneda {
  private int valor;
  private int cantidad;

  Moneda (int valor, int cantidad){
    this.valor  = valor;
    this.cantidad = cantidad;
  }

  /* Conjunto de getters y setters */
}
```

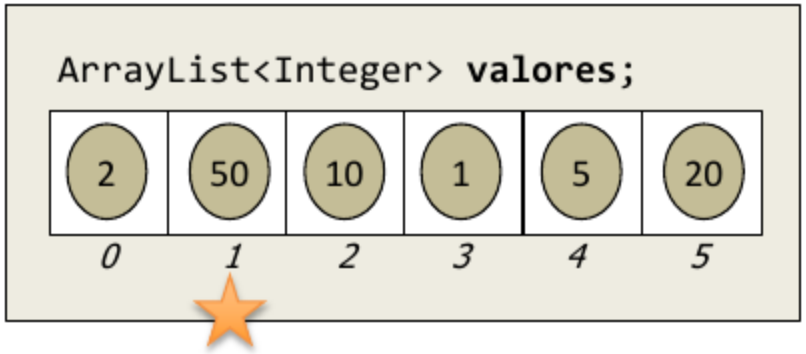


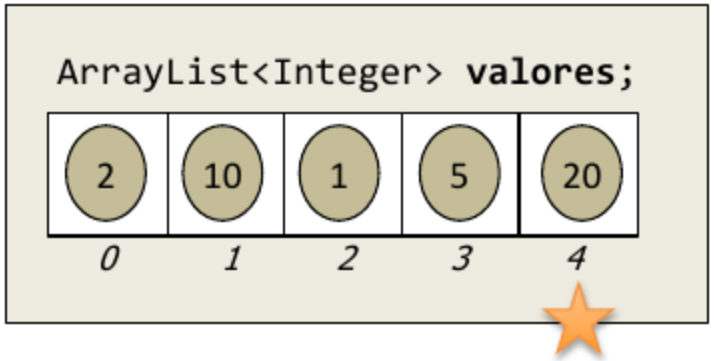


- valores: valores de las monedas del sistema monetario (lista dinámica).
- solucion: monedas que constituyen el desglose.

Implementación

```
int seleccionarCandidatoListaDesordenada(ArrayList<Integer> valores){
    int mayor = Integer.MIN_VALUE;
    for (Integer moneda : valores)
        if ((mayor< moneda)) mayor = moneda;
    return mayor;
}
```





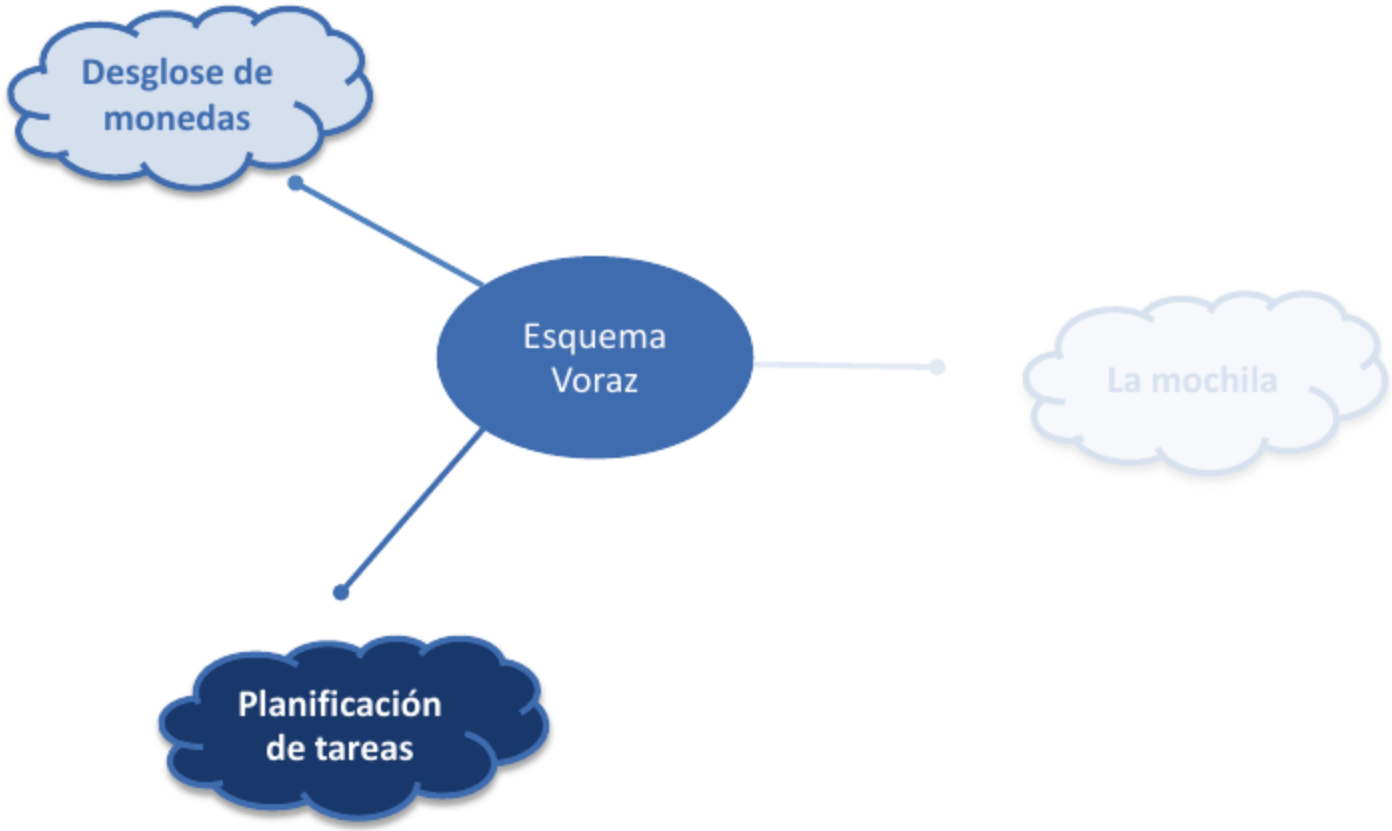
Implementación

```
ArrayList<Moneda> numMonedasDesordenadas(ArrayList<Integer> valores,
                                                    int cantidad){

    ArrayList<Moneda> solucion = new ArrayList<Moneda>();
    int valor ;
    while ((cantidad>0) &&(!valores.isEmpty())) {
        valor = seleccionarCandidatoListaDesordenada(valores);
        valores.remove(new Integer(valor));
        if ((cantidad/valor) >0){
            solucion.add(new Moneda(valor, cantidad/valor));
            cantidad = cantidad % valor;
        }
    }
    if (cantidad==0) { return solucion; }
    else return null;
}
```

Implementación

- Atención: para el *sistema monetario "50, 20, 10, 5, 2, 1"* siempre se encuentra la solución y además es óptima, pero:
  - Un cambio en dicho sistema podría provocar que no se encontrara la solución óptima. Ejemplo:
    - Sistema monetario: 10, 7, 1
    - Cantidad a devolver: 15
    - Solución del algoritmo: 10 + 1 + 1 + 1 + 1 + 1
    - Solución óptima: 7 + 7 + 1
  - E incluso que no se encontrará la solución a un determinado problema. Ejemplo:
    - Sistema monetario: 10, 7, 5
    - Cantidad a devolver: 21
  - Comprobar que la eliminación de alguna de las monedas del sistema monetario propuesto puede provocar las mismas consecuencias.

Algunos problemas planteados



Enunciado

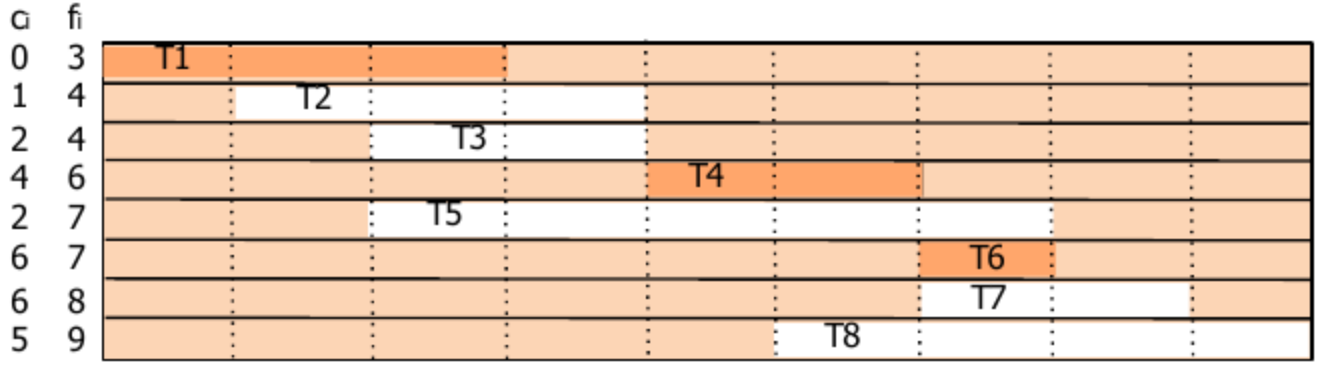
- Se tiene un conjunto de tareas (por ej., procesos en un SO) que deben usar un recurso (por ej., el procesador) que sólo puede ser usado por una actividad en cada instante.

- Cada actividad i tiene asociado:
  - un instante de comienzo c_i y un instante de finalización f_i, tales que $c_i \leq f_i$, de forma que la actividad i, si se realiza, debe hacerse durante $[c_i, f_i)$.
  - Dos actividades i, j se dicen compatibles si los intervalos $[c_i, f_i)$ y $[c_j, f_j)$ no se superponen.

- Se quiere obtener un conjunto de actividades mutuamente compatibles que maximice el número de tareas planificadas (cardinalidad máxima).

Implementación

- Opciones para la propiedad voraz:
  - ✗ Elegir las tareas de mayor tiempo de ejecución (T5).
  - ✗ Elegir las tareas de menor tiempo de ejecución (T6, T3, T4).
  - ✓ Elegir las tareas por orden de terminación (T1, T4, T6).

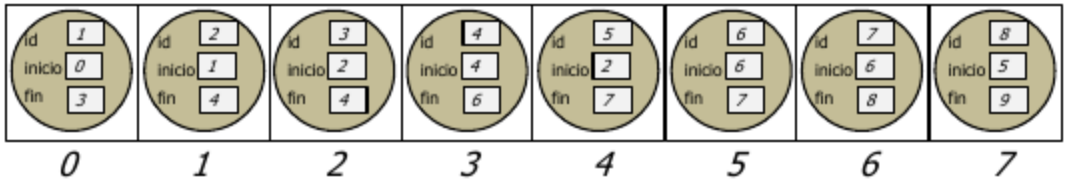


Implementación

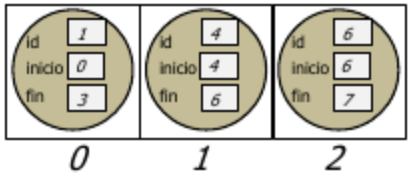
- Clase Actividad contiene la información relativa a una Tarea concreta:

```
public class Actividad{
  private int id;
  private int inicio;
  private int fin;

  Actividad(int id, int inicio, int fin){
    this.id  = id;
    this.inicio  = inicio;
    this.fin = fin;
  }
  /* getters y setters */
}
```

ArrayList<Actividad> **candidatos**;



ArrayList<Actividad> **solucion**;



- Conjunto de candidatos: tareas disponibles.
- Conjunto de seleccionados: las tareas a las que se puede dar servicio sin solaparse.

Implementación

```
private Actividad seleccionarCandidatoListaDesordenada(
                                        ArrayList<Actividad> candidatos) {
    // El mejor candidato será aquel cuyo tiempo de finalización sea
    // el menor que el resto
    Actividad mejor = null;
    for (Actividad actividad : candidatos) {
        if ((mejor == null) || (mejor.getFin() > actividad.getFin()))
            mejor = actividad;
    }
    return mejor;
}

private Actividad seleccionarCandidatoListaOrdenada(
                                        ArrayList<Actividad> candidatos) {
    // tareas ordenadas por finalización
    return candidatos.get(0);
}
```

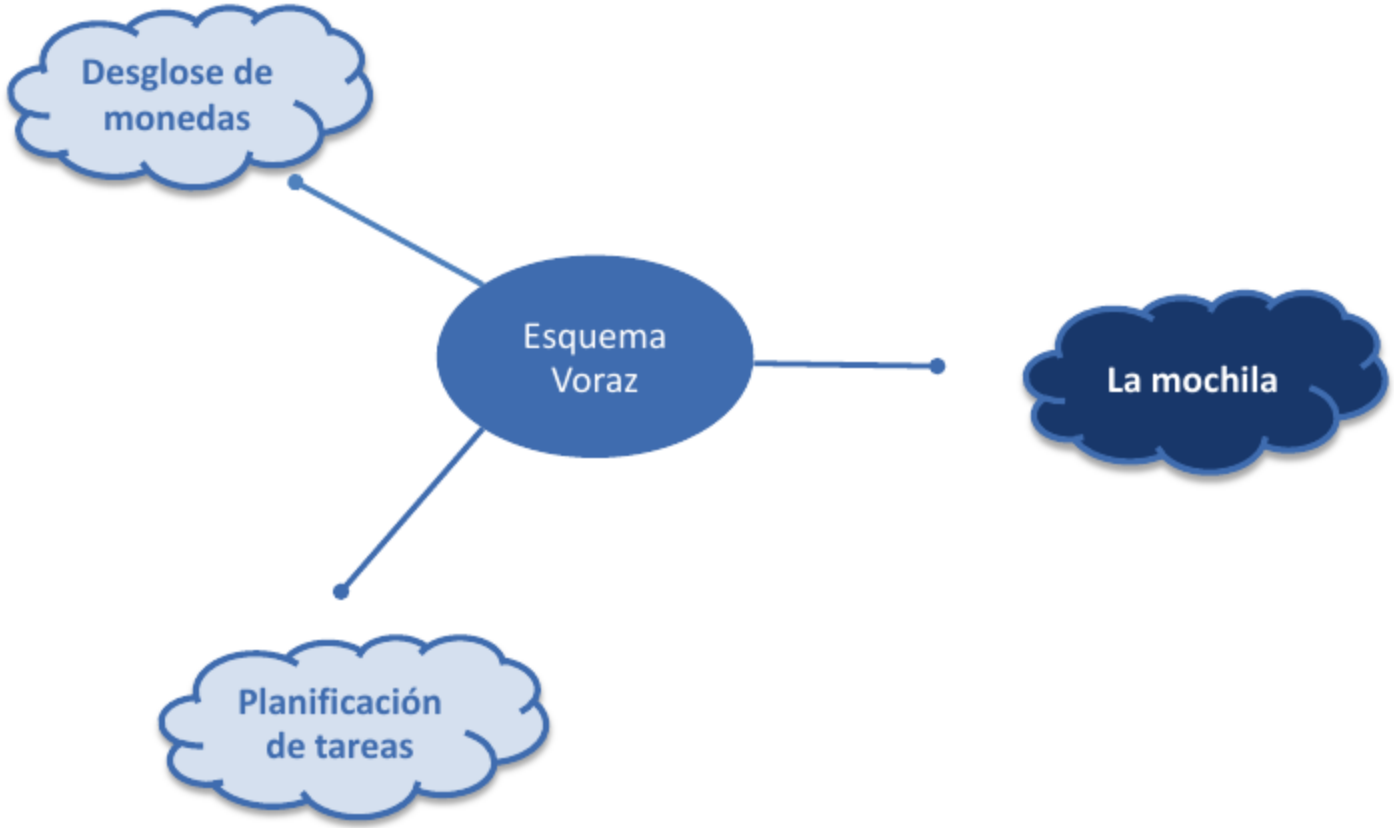
Implementación

```
private boolean esFactibleCandidato(Actividad candidato,
                                    ArrayList<Actividad> solucion) {
    boolean esFactible = true;
    Actividad actividad;
    if (solucion.size()>0)
        if (solucion.get(solucion.size()-1).getFin() >
            candidato.getInicio()) esFactible= false;
    return esFactible;
}
```

Implementación

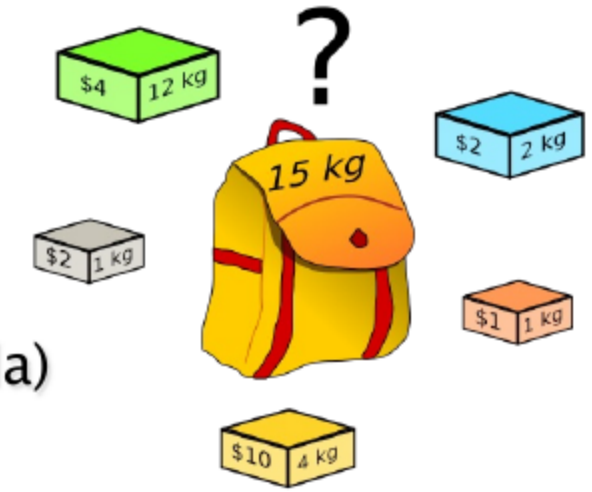
```
public ArrayList<Actividad> algoritmoVoraz(
                                        ArrayList<Actividad> candidatos) {

   ArrayList<Actividad> solucion = new ArrayList<Actividad>();
   Actividad candidato;
   while (!candidatos.isEmpty()) {
       // Seleccionar candidato
       candidato = seleccionarCandidatoListaDesordenada(candidatos);
       candidatos.remove(candidato); // Eliminar candidato
       if (esFactibleCandidato(candidato,solucion))
           // Añadir candidato a la solución
           solucion.add(candidato);
   }
   if (solucion.size()>0) // Ha podido seleccionar al menos una tarea
      return solucion;
   else return null;
}
```

Algunos problemas planteados



Enunciado

- Se tiene una mochila que es capaz de soportar un peso máximo P.
- Se tiene un conjunto de N objetos cada uno de ellos con un peso p_i y un valor v_i predeterminados.
- El objetivo es llenar la mochila de manera que se maximice el beneficio total, respetando la capacidad máxima de la misma.
  - Sea P el peso máximo de la mochila
  - Sea $V=(v_1,v_2,...,v_n)$, valores de los objetos
  - Sea $P'=(p_1,p_2,...,p_n)$, pesos de los objetos
  - $p_i \in \{1,..,P\}$ peso de cada objeto
  - Restricciones: $\Sigma\ p_j \leq P$ ($\forall\ j$ incluido en la mochila)
  - Maximizar: $\Sigma\ v_j$



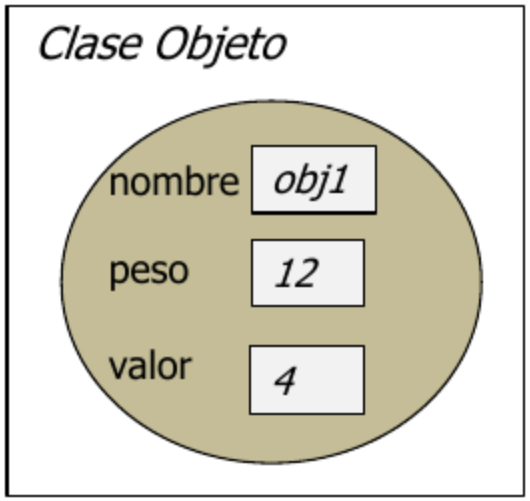
Implementación

- Función Objetivo: maximizar el valor de la mochila.
- Función Factibilidad: sólo se podrá elegir objetos que entren en la mochila en un momento dado.
- Función Solución: cuando hallamos introducido en la mochila el peso máximo o no queden objetos que introducir.
- Función Selección: podemos elegir tres estrategias.
  - × Elegir los objetos con mayor valor primero.
  - × Elegir los objetos de menos peso.
  - √ Elegir siempre el objeto que proporcione mayor beneficio por unidad de peso.

Implementación

- Clase Objeto contiene la información relativa a un objeto concreto:

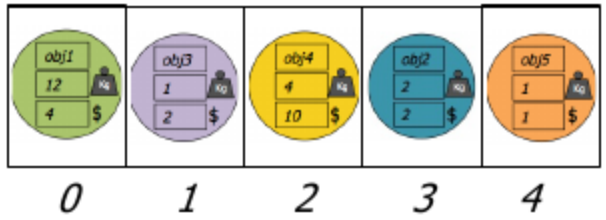
```
public class Objeto{
  private String nombre;
  private int peso;
  private int valor;

  Objeto(String nombre, int peso, int valor){
    this.nombre = nombre;
    this.peso  = peso;
    this.valor = valor;
  }

  public float getBeneficio(){
     return valor * 1.0f/peso;
  }

  /* conjunto de getters y setters */
}
```



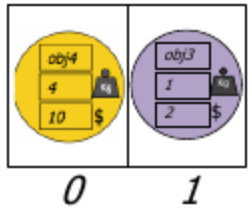
Implementación

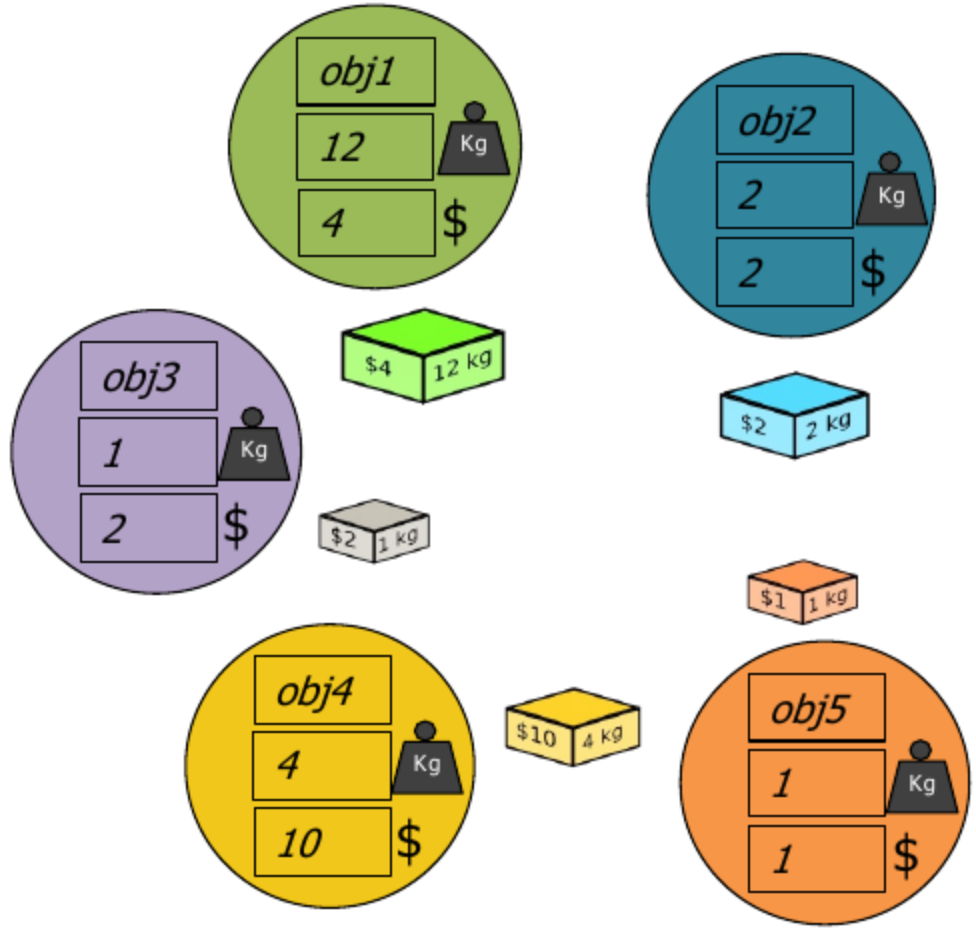
- Conjunto de candidatos: son todos los objetos disponibles.
- Conjunto de seleccionados: son los objetos seleccionados para introducir en la mochila.

```
ArrayList<Objeto> candidatos;
```




```
ArrayList<Objeto> solucion;
```





Implementación

```
private Objeto seleccionarCandidatoListaDesordenada(ArrayList<Objeto> objetos)
{   // Lista de objetos no ordenada por beneficio
    Objeto mayor = null;
    for (Objeto objeto: objetos)
        if ((mayor == null) ||
            (mayor.getBeneficio() < objeto.getBeneficio()))
            mayor = objeto;
    return mayor;
}

private Objeto seleccionarCandidatoListaOrdenada(ArrayList<Objeto> objetos)
{   // Lista de objetos ordenada por beneficio
    return objetos.get(0);
}
```

Implementación

```
public ArrayList<Objeto> algoritmoVoraz (ArrayList<Objeto> candidatos,
                                         int peso) {

    ArrayList<Objeto> solucion = new ArrayList<Objeto>(); // Inicializa solución
    Objeto candidato = null; // Candidato
    while ((peso!=0) && !(candidatos.isEmpty())) {
        // Seleccionar candidato
        candidato = seleccionarCandidatoListaDesordenada(candidatos);
        // Eliminar candidato
        candidatos.remove(candidato);
        if (peso >= candidato.getPeso()) { // Candidato Factible
            solucion.add(candidato); // Añadir candidato a la solución
            peso = peso - candidato.getPeso();
        }
    }
    return solucion;
}
```