PREDA - UNED

Programación y Estructuras de Datos Avanzadas

Algoritmos voraces y planificación

- Ejemplo 1: Minimización del tiempo en el sistema
  - Hay n clientes o tareas que esperan un servicio de un único agente o servidor, y el tiempo que requerirá dar servicio al cliente i-ésimo es t_i ($1 \leq i \leq n$)
  - El objetivo es minimizar el tiempo medio de estancia de los clientes en el sistema (equivale a minimizar el tiempo total que están en el sistema todos los clientes)
- Ejemplo 2: Planificación con plazos
  - Hay n tareas o trabajos y cada trabajo t_i permite obtener un beneficio b_i, siendo $b_i > 0$, en el caso de que se realice antes de su fecha tope f_i, siendo $f_i > 0$
  - Cada trabajo consume 1 unidad de tiempo
  - El objetivo es seleccionar los trabajos y la secuencia para realizarlos maximizando el beneficio total

Algoritmos voraces y planificación
Minimización del tiempo en el sistema

- Suponiendo que todos llegan a la vez, ¿cómo formar la cola para que se vacíe lo antes posible?

Algoritmos voraces y planificación
Minimización del tiempo en el sistema

- El algoritmo consiste básicamente en ordenar los clientes por orden no decreciente de tiempos de servicio
- ¿Coste?

Algoritmos voraces y planificación
Minimización del tiempo en el sistema

- El algoritmo consiste básicamente en ordenar los clientes por orden no decreciente de tiempos de servicio
- Coste (ordenación): O(n log n)

Algoritmos voraces y planificación
Minimización del tiempo en el sistema

- El algoritmo consiste básicamente en ordenar los clientes por orden no decreciente de tiempos de servicio
- Coste (ordenación): O(n log n)

- ¿Y si hay a agentes para realizar las tareas?


Algoritmos voraces y planificación
Minimización del tiempo en el sistema

- El algoritmo consiste básicamente en ordenar los clientes por orden no decreciente de tiempos de servicio
- Coste (ordenación): O(n log n)

- Generalización para `a` agentes:
    - Reparto “circular” del mismo número de tareas a cada agente por orden no decreciente de tiempos de servicio

Ejercicio de examen

6. Sea un procesador que ha de atender n procesos. Se conoce de antemano el tiempo que necesita cada proceso. Se desea determinar en qué orden se han de atender los procesos para minimizar la suma del tiempo que los procesos permanecen en el sistema. En relación a este problema, ¿cuál de las siguientes afirmaciones es **cierta**?

   (a) El coste del algoritmo voraz que resuelve el problema es, en el mejor de los casos, $O(n^2)$.
   (b) Suponiendo tres clientes con tiempos de servicio t_1=5, t_2=10 y t_3=3, el tiempo mínimo de estancia posible es de 26 segundos.
   (c) Suponiendo tres clientes con tiempos de servicio t_1=5, t_2=10 y t_3=3, el tiempo mínimo de estancia posible es de 31 segundos.
   (d) Todas las afirmaciones anteriores son falsas.

Ejercicio de examen

6. Sea un procesador que ha de atender n procesos. Se conoce de antemano el tiempo que necesita cada proceso. Se desea determinar en qué orden se han de atender los procesos para minimizar la suma del tiempo que los procesos permanecen en el sistema. En relación a este problema, ¿cuál de las siguientes afirmaciones es **cierta**?

   (a) El coste del algoritmo voraz que resuelve el problema es, en el mejor de los casos, $O(n^2)$.
   (b) Suponiendo tres clientes con tiempos de servicio t_1=5, t_2=10 y t_3=3, el tiempo mínimo de estancia posible es de 26 segundos.
   (c) Suponiendo tres clientes con tiempos de servicio t_1=5, t_2=10 y t_3=3, el tiempo mínimo de estancia posible es de 31 segundos.
   (d) Todas las afirmaciones anteriores son falsas.

3 + (3+5) + (3+5+10) = 29

Ejercicio de examen

6. Sea un procesador que ha de atender n procesos. Se conoce de antemano el tiempo que necesita cada proceso. Se desea determinar en qué orden se han de atender los procesos para minimizar la suma del tiempo que los procesos permanecen en el sistema. En relación a este problema, ¿cuál de las siguientes afirmaciones es **cierta**?

   (a) El coste del algoritmo voraz que resuelve el problema es, en el mejor de los casos, $O(n^2)$.
   (b) Suponiendo tres clientes con tiempos de servicio $t_1=5$, $t_2=10$ y $t_3=3$, el tiempo mínimo de estancia posible es de 26 segundos.
   (c) Suponiendo tres clientes con tiempos de servicio $t_1=5$, $t_2=10$ y $t_3=3$, el tiempo mínimo de estancia posible es de 31 segundos.
   (d) Todas las afirmaciones anteriores son falsas.

Ejercicio de examen

1. Un servidor tiene que atender *tres* clientes que llegan todos juntos al sistema. El tiempo que requerirá dar servicio a cada cliente es conocido, siendo t_1=5, t_2=10 y t_3=3. El objetivo es minimizar el tiempo medio de estancia de los clientes en el sistema. Se quiere implementar un algoritmo voraz que construya la secuencia ordenada óptima de servicio a los distintos clientes. Según este algoritmo, el tiempo mínimo de estancia en el sistema del conjunto de clientes es:

   (a) 26
   (b) 29
   (c) 31
   (d) 34

Ejercicio de examen

1. Un servidor tiene que atender *tres* clientes que llegan todos juntos al sistema. El tiempo que requerirá dar servicio a cada cliente es conocido, siendo $t_1=5$, $t_2=10$ y $t_3=3$. El objetivo es minimizar el tiempo medio de estancia de los clientes en el sistema. Se quiere implementar un algoritmo voraz que construya la secuencia ordenada óptima de servicio a los distintos clientes. Según este algoritmo, el tiempo mínimo de estancia en el sistema del conjunto de clientes es:

   (a) 26
   (b) 29
   (c) 31
   (d) 34

3 + (3+5) + (3+5+10) = 29

Ejercicio de examen

1. Un dentista pretende dar servicio a *n* pacientes y conoce el tiempo requerido por cada uno de ellos, siendo t_i , $i= 1,2,\ldots,n$ el tiempo requerido por el paciente *i*. El objetivo es minimizar el tiempo medio de estancia de los pacientes en la consulta. En relación a este problema, ¿cuál de las siguientes afirmaciones es **falsa**?

   (a) El esquema más eficiente para resolver este problema correctamente es el esquema voraz.
   (b) El coste del algoritmo voraz que resuelve el problema es O(n log n).
   (c) Suponiendo tres pacientes con tiempos de servicio t_1=15, t_2=60 y t_3=30, el tiempo mínimo de estancia total posible es de tres horas.
   (d) Suponiendo tres pacientes con tiempos de servicio t_1=15, t_2=20 y t_3=30, el tiempo mínimo de estancia total posible es de 115 minutos.

Ejercicio de examen

1. Un dentista pretende dar servicio a *n* pacientes y conoce el tiempo requerido por cada uno de ellos, siendo t_i , $i= 1,2,\ldots,n$ el tiempo requerido por el paciente *i*. El objetivo es minimizar el tiempo medio de estancia de los pacientes en la consulta. En relación a este problema, ¿cuál de las siguientes afirmaciones es **falsa**?

   (a) El esquema más eficiente para resolver este problema correctamente es el esquema voraz.
   (b) El coste del algoritmo voraz que resuelve el problema es O(n log n).
   (c) Suponiendo tres pacientes con tiempos de servicio t_1=15, t_2=60 y t_3=30, el tiempo mínimo de estancia total posible es de tres horas.
   (d) Suponiendo tres pacientes con tiempos de servicio t_1=15, t_2=20 y t_3=30, el tiempo mínimo de estancia total posible es de 115 minutos.

c) 15 + (15+30) + (15+30+60) = 165
d) 15 + (15+20) + (15+20+30) = 115

Ejercicio de examen

1. Un dentista pretende dar servicio a *n* pacientes y conoce el tiempo requerido por cada uno de ellos, siendo t_i , $i = 1,2,\dots,n$ el tiempo requerido por el paciente *i*. El objetivo es minimizar el tiempo medio de estancia de los pacientes en la consulta. En relación a este problema, ¿cuál de las siguientes afirmaciones es **falsa**?

   (a) El esquema más eficiente para resolver este problema correctamente es el esquema voraz.
   (b) El coste del algoritmo voraz que resuelve el problema es O(n log n).
   (c) Suponiendo tres pacientes con tiempos de servicio $t_1=15$, $t_2=60$ y $t_3=30$, el tiempo mínimo de estancia total posible es de tres horas.
   (d) Suponiendo tres pacientes con tiempos de servicio $t_1=15$, $t_2=20$ y $t_3=30$, el tiempo mínimo de estancia total posible es de 115 minutos.

c) 15 + (15+30) + (15+30+60) = 165
d) 15 + (15+20) + (15+20+30) = 115

Algoritmos voraces y planificación
Planificación con plazos

- Una solución S (conjunto de tareas) es factible si existe al menos una secuencia que permite que todas las tareas del conjunto se puedan completar dentro del plazo, y es óptima si es una solución factible con valor máximo
  - Su valor es la suma de los beneficios de dichas tareas
- Selección: considerar los trabajos en orden decreciente de beneficios siempre que el conjunto sea una solución factible
  - Lema 3.3.1: Si S es un conjunto de trabajos, entonces S es factible si y solo si la secuencia obtenida ordenando los trabajos en orden no decreciente de fechas tope es factible

Algoritmos voraces y planificación
Planificación con plazos

fun PlanificacionPlazos (F[1..*n*], n): Vector[1..n] de natural
    S ← {1}
    para i = 2 hasta n **hacer**
        si los trabajos en S ∪ {i} constituyen una secuencia factible **entonces**
            S ← S ∪ {i}
        fsi
    fpara
    dev S
ffun

El array *F*[] almacena las fechas tope de realización de los *n* trabajos ordenados en orden decreciente de beneficios.

- Lema 3.3.1: Si S es un conjunto de trabajos, entonces S es factible si y solo si la secuencia obtenida ordenando los trabajos en orden no decreciente de fechas tope es factible.

```
tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosDetallado (f: VectorNat, n: natural): VectorNat, natural
    var
        S: VectorNat
    fvar
    S[0] ← 0 {elemento centinela para facilitar la inserción}
    S[1] ← 1 {se incluye el trabajo 1 que es el de máximo beneficio}
    k ← 1 {k: nº de elementos en S}
    para i = 2 hasta n hacer
        r ← k {se busca una posición válida para i}
        mientras (f[S[r]] > f[i]) ∧ (f[S[r]] ≠ r) hacer
            r ← r - 1
        fmientras
        si (f[S[r]] ≤ f[i]) ∧ (f[i]> r) entonces
            para q = k hasta r + 1 incr = -1 hacer
                S[q+1] ← S[q]
            fpara
            S[r+1] ← i
            k ← k +1
        fsi
    fpara
    dev S, k
ffun
```

```
tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosDetallado (f: VectorNat, n: natural): VectorNat, natural
    var
        S: VectorNat
    fvar
    S[0] ← 0 {elemento centinela para facilitar la inserción}
    S[1] ← 1 {se incluye el trabajo 1 que es el de máximo beneficio}
    k ← 1 {k: nº de elementos en S}
    para i = 2 hasta n hacer
        r ← k {se busca una posición válida para i}
        mientras (f[S[r]] > f[i]) ∧ (f[S[r]] ≠ r) hacer
            r ← r - 1
        fmientras
        si (f[S[r]] ≤ f[i]) ∧ (f[i]> r) entonces
            para q = k hasta r + 1 incr = -1 hacer
                S[q+1] ← S[q]
            fpara
            S[r+1] ← i
            k ← k +1
        fsi
    fpara
    dev S, k
ffun
```

Desplazamos los trabajos anteriores hasta que:

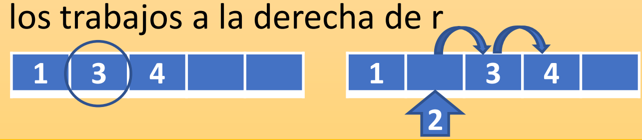
- El plazo es <= al nuestro
- No hay margen para retrasar

```
tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosDetallado (f: VectorNat, n: natural): VectorNat, natural
    var
        S: VectorNat
    fvar
     S[0] ← 0 {elemento centinela para facilitar la inserción}
     S[1] ← 1 {se incluye el trabajo 1 que es el de máximo beneficio}
     k ← 1 {k: nº de elementos en S}
     para i = 2 hasta n hacer
        r ← k {se busca una posición válida para i}
        mientras (f[S[r]] > f[i]) ∧ (f[S[r]] ≠ r) hacer
            r ← r - 1
        fmientras
        si (f[S[r]] ≤ f[i]) ∧ (f[i]> r) entonces
            para q = k hasta r + 1 incr = -1 hacer
                S[q+1] ← S[q]
            fpara
            S[r+1] ← i
            k ← k +1
        fsi
     fpara
     dev S, k
ffun
```

Desplazamos los trabajos anteriores hasta que:

- El plazo es <= al nuestro
- No hay margen para retrasar

Si hay hueco r para insertar el trabajo i (antes de la última posición), desplazamos los trabajos a la derecha de r

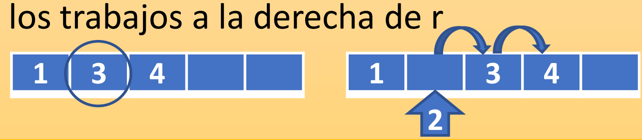


```
tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosDetallado (f: VectorNat, n: natural): VectorNat, natural
    var
        S: VectorNat
    fvar
    S[0] ← 0 {elemento centinela para facilitar la inserción}
    S[1] ← 1 {se incluye el trabajo 1 que es el de máximo beneficio}
    k ← 1 {k: nº de elementos en S}
    para i = 2 hasta n hacer
        r ← k {se busca una posición válida para i}
        mientras (f[S[r]] > f[i]) ∧ (f[S[r]] ≠ r) hacer
            r ← r - 1
        fmientras
        si (f[S[r]] ≤ f[i]) ∧ (f[i]> r) entonces
            para q = k hasta r + 1 incr = -1 hacer
                S[q+1] ← S[q]
            fpara
            S[r+1] ← i
            k ← k +1
        fsi
    fpara
    dev S, k
ffun
```

Desplazamos los trabajos anteriores hasta que:

- El plazo es <= al nuestro
- No hay margen para retrasar

Si hay hueco r para insertar el trabajo i (antes de la última posición), desplazamos los trabajos a la derecha de r



Coste: $O(n^2)$

ficación

```
tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosMejorado (f: VectorNat, n: natural): VectorNat, natural
    var
        S, libre: VectorNat
    fvar
    p ← min(n, max{f[i] |1 ≤ i ≤ n })
    para i = 0 hasta n hacer
        S[i] ← 0
        libre[i] ← i
        Iniciar conjunto i
    fpara
    para i = 1 hasta n hacer
        k ← buscar(min(p,f[i]))
        pos ← libre(k)
        si pos ≠ 0 entonces
            S[pos] ← i
            l ← buscar(pos - 1)
            libre[k] ← libre[l]
            fusionar(k,l) {asignar la etiqueta k o l}
        fsi
    fpara
    k ← 0
    para i = 1 hasta n hacer
        si S[i] > 0 entonces
            k ← k + 1
            S[k] ← S[i]
        fsi
    fpara
    dev S, k
ffun
```

Se añade en la lista de tareas en la posición más tardía posible

- Inicialmente cada posición o instante de tiempo $0, 1, 2, 3, \ldots, p$ está en un conjunto diferente y $libre(\{i\}) = i, 0 \leq i \leq p$
- Si se quiere añadir una tarea con plazo f se busca el conjunto que contenga a f. Sea K dicho conjunto. Si $libre(K) = 0$ se rechaza la tarea; en caso contrario se realizan las siguientes acciones:
  - Se asigna la tarea al instante de tiempo $libre(K)$.
  - Se busca el conjunto que contenga $libre(K) - 1$. Sea L dicho conjunto.
  - Se fusionan K y L. El valor de $libre()$ para este nuevo conjunto es el valor que tenía $libre(L)$.

den
el conjunto

s, entonces S es
rdenando los
s tope es

```
tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosMejorado (f: VectorNat, n: natural): VectorNat, natural
    var
        S, libre: VectorNat
    fvar
    p ← min(n, max{f[i] |1 ≤ i ≤ n })
    para i = 0 hasta n hacer
        S[i] ← 0
        libre[i] ← i
        Iniciar conjunto i
    fpara
    para i = 1 hasta n hacer
        k ← buscar(min(p,f[i]))
        pos ← libre(k)
        si pos ≠ 0 entonces
            S[pos] ← i
            l ← buscar(pos - 1)
            libre[k] ← libre[l]
            fusionar(k,l) {asignar la etiqueta k o l}
        fsi
    fpara
    k ← 0
    para i = 1 hasta n hacer
        si S[i] > 0 entonces
            k ← k + 1
            S[k] ← S[i]
        fsi
    fpara
    dev S, k
ffun
```

Se añade en la lista de tareas en la posición más tardía posible

- Inicialmente cada posición o instante de tiempo $0, 1, 2, 3, \ldots, p$ está en un conjunto diferente y $libre(\{i\}) = i, 0 \leq i \leq p$
- Si se quiere añadir una tarea con plazo f se busca el conjunto que contenga a f. Sea K dicho conjunto. Si $libre(K) = 0$ se rechaza la tarea; en caso contrario se realizan las siguientes acciones:
  - Se asigna la tarea al instante de tiempo $libre(K)$.
  - Se busca el conjunto que contenga $libre(K) - 1$. Sea L dicho conjunto.
  - Se fusionan K y L. El valor de $libre()$ para este nuevo conjunto es el valor que tenía $libre(L)$.

bucle para comprimir S (por si hay huecos que no se han llenado):
k parte de 0 y va de 1 en 1 pero i se salta las posiciones vacías

ficación

den
el conjunto

s, entonces S es
rdenando los
s tope es

```
tipo VectorNat = matriz[0..n] de natural
fun PlanificacionPlazosMejorado (f: VectorNat, n: natural): VectorNat, natural
    var
        S, libre: VectorNat
    fvar
    p ← min(n, max{f[i] |1 ≤ i ≤ n })
    para i = 0 hasta n hacer
        S[i] ← 0
        libre[i] ← i
        Iniciar conjunto i
    fpara
    para i = 1 hasta n hacer
        k ← buscar(min(p,f[i]))
        pos ← libre(k)
        si pos ≠ 0 entonces
            S[pos] ← i
            l ← buscar(pos - 1)
            libre[k] ← libre[l]
            fusionar(k,l) {asignar la etiqueta k o l}
        fsi
    fpara
    k ← 0
    para i = 1 hasta n hacer
        si S[i] > 0 entonces
            k ← k + 1
            S[k] ← S[i]
        fsi
    fpara
    dev S, k
ffun
```

Se añade en la lista de tareas en la posición más tardía posible

- Inicialmente cada posición o instante de tiempo $0, 1, 2, 3, \ldots, p$ está en un conjunto diferente y $libre(\{i\}) = i, 0 \leq i \leq p$
- Si se quiere añadir una tarea con plazo f se busca el conjunto que contenga a f. Sea K dicho conjunto. Si $libre(K) = 0$ se rechaza la tarea; en caso contrario se realizan las siguientes acciones:
  - Se asigna la tarea al instante de tiempo $libre(K)$.
  - Se busca el conjunto que contenga $libre(K) - 1$. Sea L dicho conjunto.
  - Se fusionan K y L. El valor de $libre()$ para este nuevo conjunto es el valor que tenía $libre(L)$.

bucle para comprimir S (por si hay huecos que no se han llenado):
k parte de 0 y va de 1 en 1 pero i se salta las posiciones vacías

ficación

den
el conjunto

s, entonces S es
rdenando los
s tope es

Coste: O(n log n)

Algoritmos voraces y almacenamiento óptimo

- Se dispone de n programas y hay que almacenarlos en un soporte secuencial de longitud L
- Cada programa p_i tiene una longitud l_i, siendo $1 \leq i \leq n$
- Tiempo de acceso:
  - Para poder acceder a p_i asumimos que hay que posicionar el mecanismo de acceso al comienzo del soporte.
  - Si p_i está en la posición x_i , el tiempo de acceso a ese programa será la suma del tiempo para avanzar hasta x_i , más el tiempo de lectura l_i
- Objetivo:
  encontrar una secuencia de almacenamiento de los programas que minimice el tiempo medio de acceso
- ¿Estrategia?

Algoritmos voraces y almacenamiento óptimo

- Se dispone de n programas y hay que almacenarlos en un soporte secuencial de longitud L
- Cada programa p_i tiene una longitud l_i, siendo $1 \leq i \leq n$
- Tiempo de acceso:
  - Para poder acceder a p_i asumimos que hay que posicionar el mecanismo de acceso al comienzo del soporte.
  - Si p_i está en la posición x_i, el tiempo de acceso a ese programa será la suma del tiempo para avanzar hasta x_i, más el tiempo de lectura l_i
- Objetivo:
  encontrar una secuencia de almacenamiento de los programas que minimice el tiempo medio de acceso
- Estrategia:
  seleccionar los p_i en orden no decreciente de longitudes

Algoritmos voraces y almacenamiento óptimo

- Se dispone de n programas y hay que almacenarlos en un soporte secuencial de longitud L

```
fun AlmacenaProgramasSoporteSec (c: conjuntoCandidatos): conjuntoCandidatos
    Ordenar c en orden no decreciente de longitudes
    sol ← ∅
    mientras c ≠ ∅ hacer
        x ← seleccionar(c)
        c ← c - {x}
        sol ← sol ∪ {x}
    fmientras
    devolver sol
ffun
```

La función *seleccionar*() selecciona los programas en el orden establecido.

decreciente de longitudes

Coste: O(n log n)

Algoritmos voraces y almacenamiento óptimo

- ¿Y si tenemos `m` soportes secuenciales?

Algoritmos voraces y almacenamiento óptimo

- Generalización a m soportes secuenciales
  - Como los programas están ordenados de manera que $l_1 \leq l_2 \leq \ldots \leq l_n$, se realiza un reparto "circular" en los m soportes
    (el programa i se almacenará en el soporte $S_{i \bmod m}$)
  - De esta manera, en cada soporte los programas están almacenados en orden no decreciente de sus longitudes

Algoritmos voraces y la mochila con objetos fraccionables

- Se dispone de n objetos con un peso positivo p_i y un valor positivo v_i , y una mochila con un peso máximo M
- Los objetos se pueden fraccionar y una fracción x_i del objeto i, siendo $0 \leq x_i \leq 1$ añadirá a la mochila un peso de $x_i p_i$ y un valor $x_i v_i$
- Objetivo: llenar la mochila de manera que se maximice el valor total considerando el peso máximo:

$$\text{maximizar} \sum_{i=1}^{n} x_i v_i \text{ con la restricción } \sum_{i=1}^{n} x_i p_i \leq M, \text{ y } 0 \leq x_i \leq 1, 1 \leq i \leq n$$

Algoritmos voraces y la mochila con objetos fraccionables

- Una solución óptima debe llenar la mochila
- ¿Funciones de selección posibles?

Algoritmos voraces y la mochila con objetos fraccionables

- Una solución óptima debe llenar la mochila
- Funciones de selección posibles:
  1. Seleccionar el objeto más valioso de los restantes
  2. Seleccionar el de menos peso
  3. Seleccionar el objeto cuyo valor por unidad de peso sea el mayor posible

Algoritmos voraces y la mochila con objetos fraccionables

- Una solución óptima debe llenar la mochila
- Funciones de selección posibles:
  1. Seleccionar el objeto más valioso de los restantes
  2. Seleccionar el de menos peso
  3. Seleccionar el objeto cuyo valor por unidad de peso sea el mayor posible

Algoritmos voraces y la mochila co

- Un
- Fu
  1.
  2.
  3. sea

```
tipo VectorNat = matriz[0..n] de natural
tipo VectorRea = matriz[0..n] de real
fun MochilaObjetosFraccionables (p: VectorNat, v: VectorNat, M: natural): VectorRea
    var
        x: VectorRea
        peso: natural
    fvar
     Ordenar objetos en orden no creciente de vi/pi
     peso ← 0
     para i = 1 hasta n hacer
         x[i] ← 0
     fpara
     mientras peso < M hacer
             i ← mejor objeto de los restantes
             si peso + p[i] ≤ M entonces
                 x[i] ← 1
                 peso ← peso + p[i]
             sino
                 x[i] ← (M - peso)/ p[i]
                 peso ← M
             fsi
     fmientras
     dev x
ffun
```

Algoritmos voraces y la mochila
co

- Un
- Fu
  1.
  2.
  3. sea

```
tipo VectorNat = matriz[0..n] de natural
tipo VectorRea = matriz[0..n] de real
fun MochilaObjetosFraccionables (p: VectorNat, v: VectorNat, M: natural): VectorRea
    var
        x: VectorRea
        peso: natural
    fvar
     Ordenar objetos en orden no creciente de v_i/p_i
     peso ← 0
     para i = 1 hasta n hacer
         x[i] ← 0
     fpara
     mientras peso < M hacer
            i ← mejor objeto de los restantes
            si peso + p[i] ≤ M entonces
                x[i] ← 1
                peso ← peso + p[i]
            sino
                x[i] ← (M - peso)/ p[i]
                peso ← M
            fsi
     fmientras
     dev x
ffun
```

Coste: O(n log n)

Ejercicio de examen

4.- En un problema de mochila con objetos fraccionables tenemos $n = 8$ objetos disponibles. Los pesos de los objetos son $w= (8, 7, 6, 5, 4, 3, 2, 1)$ y los beneficios son $v= (10, 9, 8, 7, 6, 5, 4, 3)$. ¿Cuál es el beneficio óptimo para este ejemplo, suponiendo que la capacidad de la mochila es $M = 9$?

a. 12
b. 15
c. 16.5
d. Ninguna de las otras respuestas es correcta.

Ejercicio de examen

4.- En un problema de mochila con objetos fraccionables tenemos n = 8 objetos disponibles. Los pesos de los objetos son w= (8, 7, 6, 5, 4, 3, 2, 1) y los beneficios son v= (10, 9, 8, 7, 6, 5, 4, 3). ¿Cuál es el beneficio óptimo para este ejemplo, suponiendo que la capacidad de la mochila es M = 9?

a. 12
b. 15
c. 16.5
d. Ninguna de las otras respuestas es correcta.

Dividir valor/peso e introducir hasta llenar la mochila

Ejercicio de examen

4.- En un problema de mochila con objetos fraccionables tenemos n = 8 objetos disponibles. Los pesos de los objetos son w= (8, 7, 6, 5, 4, 3, 2, 1) y los beneficios son v= (10, 9, 8, 7, 6, 5, 4, 3). ¿Cuál es el beneficio óptimo para este ejemplo, suponiendo que la capacidad de la mochila es M = 9?

a. 12
b. 15
c. 16.5
d. Ninguna de las otras respuestas es correcta.

	1	2	3	4	5	6	7	8
w	8	7	6	5	4	3	2	1
v	10	9	8	7	6	5	4	3
v/w	1,25	1,29	1,33	1,40	1,50	1,67	2,00	3,00

Ejercicio de examen

4.- En un problema de mochila con objetos fraccionables tenemos n = 8 objetos disponibles. Los pesos de los objetos son w= (8, 7, 6, 5, 4, 3, 2, 1) y los beneficios son v= (10, 9, 8, 7, 6, 5, 4, 3). ¿Cuál es el beneficio óptimo para este ejemplo, suponiendo que la capacidad de la mochila es M = 9?

a. 12
b. 15
c. 16.5
d. Ninguna de las otras respuestas es correcta.

Introducimos por orden creciente de v/m hasta el objeto 5 del que solo cabe un 75%

	1	2	3	4	5	6	7	8
w	8	7	6	5	4	3	2	1
v	10	9	8	7	6	5	4	3
v/w	1,25	1,29	1,33	1,40	1,50	1,67	2,00	3,00

Beneficio total = 3+4+5+(0,75*6) = 16,5