2.3 Algoritmos ávidos

También nombrados voraces, glotones cuyo objetivo es obtener una solución global óptima a un problema, es parecida a la programación dinámica, la diferencia con este es que la programación dinámica junta todas las soluciones de cada subproblema y encuentra la solución óptima, en cambio el algoritmo ávido solo toma la solución óptima anterior a la actual con lo que no almacena información o datos anteriores.

El algoritmo ávido comienza con una toma la solución más simple y toma la decisión que resulte más eficiente en ese instante de la ejecución, con lo que se espera llegar a una solución óptima global. Los algoritmos ávidos encuentran las soluciones mucho más rápido que por programación dinámica, y que por cualquier otro método. El problema es que, aunque el enfoque es generalmente fácil de implementar, en la mayoría de las ocasiones falla por no tomar en cuenta todas las combinaciones. Es por esto que no es conveniente utilizarlo a menos de que estemos convencidos de que funciona.

Entre los algoritmos más conocidos que utilizan un enfoque ávido, se encuentran los de Dijkstra, Prim y Kruskal. A veces son utilizados para diseñar algoritmos heurísticos para problemas NP-completo, con lo que se puede obtener una buena aproximación de la solución en poco tiempo. De acuerdo a la complejidad del problema, un enfoque ávido puede dar una aproximación suficientemente buena o, si queremos una respuesta óptima, se emplea para realizar podas en una búsqueda exhaustiva.

2.3.1 Elementos de la estrategia ávida

El problema a resolver ha de ser de optimización $[MIN/MAX]$ y debe existir una función, la función objetivo, que es la que hay que minimizar o maximizar. La siguiente función, que es lineal y multivariable, es una función objetivo típica.

$$f: \mathbb{N}x\mathbb{N}x\mathbb{N}x \dots x\mathbb{N} \rightarrow \mathbb{N}(oR^{+})$$

$$f(x_1, x_2, \dots, x_n) = c_1 \cdot x_1 + c_2 \cdot x_2 + \cdots + c_n \cdot x_n$$

Existe un conjunto de valores posibles para cada una de las variables de la función objetivo, su dominio.

Puede existir un conjunto de restricciones que imponen condiciones a los valores del dominio que pueden tomar las variables de la función objetivo.

La solución al problema debe ser expresable en forma de secuencia de decisiones y debe existir una función que permita determinar cuándo una secuencia de decisiones es solución para el problema (función solución).

Entendemos por decisión la asociación a una variable de un valor de su domino. Debe existir una función que permita determinar si una secuencia de decisiones viola o no las restricciones, la función factible.

2.3.2 El problema de selección de actividades

El propósito de un algoritmo voraz es encontrar una solución, es decir, una asociación de valores a todas las variables tal que el valor de la función objetivo sea óptimo. Para ello sigue un proceso secuencial en el que a cada paso toma una decisión (decide qué valor del dominio le ha de asignar a la variable actual) aplicando siempre el mismo criterio (función de selección).

La decisión es óptima localmente, es decir, ningún otro valor de los disponibles para esa variable lograría que la función objetivo tuviera un valor mejor, y luego comprueba si la puede incorporar a la secuencia de decisiones que ha tomado hasta el momento, comprueba que la nueva decisión junto con todas las tomadas anteriormente no violan las restricciones y así consigue una nueva secuencia de decisiones factibles.

En el siguiente paso el algoritmo voraz se encuentra con un problema idéntico, pero estrictamente menor, al que tenía en el paso anterior y vuelve a aplicar la misma función de selección para tomar la siguiente decisión. Esta es, por tanto, una técnica descendente.
Pero nunca se vuelve a reconsiderar ninguna de las decisiones tomadas. Una vez que a una variable se le ha asignado un valor localmente óptimo y que hace que la secuencia de decisiones sea factible, nunca más se va a intentar asignar un nuevo valor a esa misma variable.

En seudocódigo sería algo así:

función ejecutarAlgoritmoVORAZ(C es conjunto) dev (S es conjunto; v es valor)
{*Pre:* C es el conjunto de valores posibles que hay que asociar a las variables de la función objetivo para obtener la solución. Sin pérdida de generalidad, se supone que todas las variables comparten el mismo dominio de valores que son, precisamente, los que contiene C. En este caso el número de variables que hay que asignar es desconocido y la longitud de la solución indicará cuantas variables han sido asignadas}
S := secuenciaVacía; // **inicialmente la solución está vací**a
{*Inv:* S es una secuencia de decisiones factible y C contiene los valores del dominio que todavía se pueden utilizar}
mientras (¬solución(S) ∩ ¬vacio(C)) hacer
x := selección(C); // **se obtiene el candidato localmente óptimo**
si (factible($(S) \cup \{x\}$) **entonces**
S := añadir(S,{x});
sino
C := C-{x}; // **este candidato no se puede volver a utilizar**
finsi
finmientras
// **la iteración finaliza porque S es una solución o porque el conjunto de candidatos está vacío**
si (solución(S)) entonces
v := valor(objetivo(S))
sino S := secVacia
finsi
{*Post:* (S = secVacia ⇒ no se ha encontrado solución) ∩ (S ≠ secVacia ⇒ S es una secuencia factible de decisiones y es solución ∩ v = valor(S))}
dev (S, v)
finfunción

El coste de este algoritmo depende de dos cosas:

1. El número de iteraciones, que a su vez depende del tamaño de la solución construida y del tamaño del conjunto de candidatos.

2. El coste de las funciones selección y factible ya que el resto de las operaciones del interior del bucle pueden tener coste constante. El coste de la función factible dependerá del problema (número y complejidad de las restricciones). La función de selección ha de explorar el conjunto de valores candidatos y obtener el mejor en ese momento y por eso su coste depende, entre otras cosas, del tamaño del conjunto de candidatos. Para reducir el coste de la función de selección, siempre que sea posible se prepara el conjunto de candidatos antes de entrar en el bucle. Normalmente el preproceso consiste en ordenar el conjunto de valores posibles lo que hace que el coste del algoritmo voraz acabe siendo del orden de $\theta(nlog(n))$ con $n = |C|$. con n el tamaño del conjunto a ordenar.

2.3.3 Códigos de Huffman

La codificación de Huffman es una técnica para la compresión de datos ampliamente usada y efectiva. En el ejemplo siguiente se ilustra esta técnica de codificación. Se dispone de un archivo con 100000 caracteres. Se sabe que aparecen 6 caracteres diferentes y la frecuencia de aparición de cada uno de ellos es:

Caracteres	a	b	c	d	e	F
Frecuencia (en miles)	45	13	12	16	9	5

El problema es cómo codificar los caracteres para reducir el espacio que ocupan y utilizando un código binario. La primera alternativa es utilizar un código de longitud fija. En este caso para distinguir 6 caracteres se necesitan 3 bits. El espacio necesario para almacenar los 100000 caracteres, sabiendo que cada uno de ellos requiere 3 bits, es de 300000 bits.
Otra codificación posible sería:

Caracteres	a	b	c	d	e	F
Longitud Fija	000	001	010	011	100	101

Pero se puede utilizar un código de longitud variable en el que los caracteres más frecuentes tienen el código más corto. Al hacer la codificación se deberá tener en cuenta que ningún código ha de ser prefijo de otro. En caso contrario la descodificación no será única. El espacio necesario es ahora de 224000 bits.

Caracteres	a	b	c	d	e	F
Longitud Variable	0	101	100	111	1101	1100

Esta técnica de codificación se denomina código prefijo. La ventaja es que para codificar basta con concatenar el código de cada uno de los caracteres y la descodificación también es simple y no se produce ninguna ambigüedad ya que ningún código es prefijo de otro código.

Codificación: aabacd ≡ 0·0·101·0·100·111≡ 001010100111
Decodificación: 101011101111011100 ≡ badadcf y es la única posibilidad

La forma habitual de representar el código, tanto el de longitud variable como el de fija, es usando un árbol binario de forma que el proceso de decodificación se simplifica. En este árbol, las hojas son los caracteres y el camino de la raíz hacia las hojas, con la interpretación 0 a la izquierda y 1 a la derecha, esto deja el código de cada hoja.

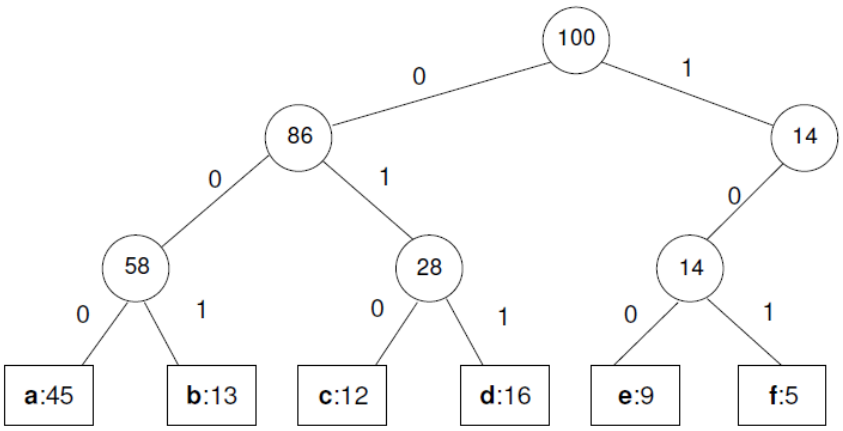

Figura 2.3.3.1 Árbol binario de codificación de longitud fija.

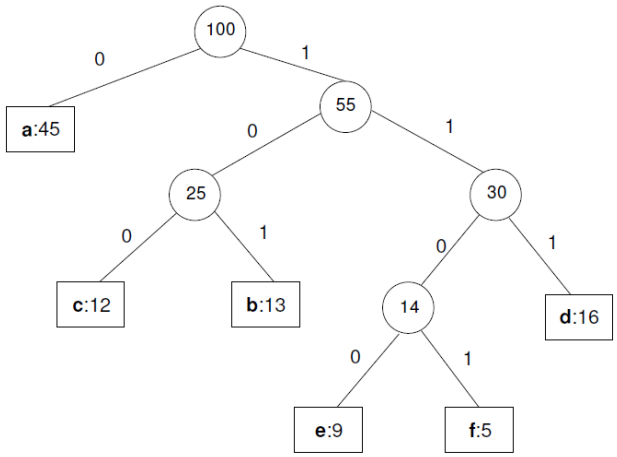

Figura 2.3.3.2 Árbol binario de codificación de longitud variable.

Sea *T* el árbol binario que corresponde a una codificación prefija, entonces el número de bits necesarios para codificar el archivo, *B*(*T*), se calcula de la siguiente forma:

1. Para cada carácter *c* diferente del alfabeto *C* que aparece en el archivo,
   a. Sea *f*(*c*) la frecuencia de *c* en la entrada,
   b. sea $d_T(c)$ la profundidad de la hoja *c* en el árbol *T*, entonces el número de bits requeridos será de:

$$B(T) = \sum_{c \epsilon C} f(c) \cdot d_T(c)$$

Huffman propuso un algoritmo voraz que obtiene una codificación prefijo óptima. Para ello se construye un árbol binario de códigos de longitud variable de manera ascendente de modo que [*MIN*]*B*(*T*). El algoritmo funciona de la siguiente forma:

1. Parte de una secuencia inicial en la que los caracteres a codificar están colocados en orden creciente de frecuencia.
2. Esta secuencia inicial se va transformando, a base de fusiones, hasta llegar a una secuencia con un único elemento que es el árbol de codificación óptimo como se muestra en la figura 2.3.3.3:

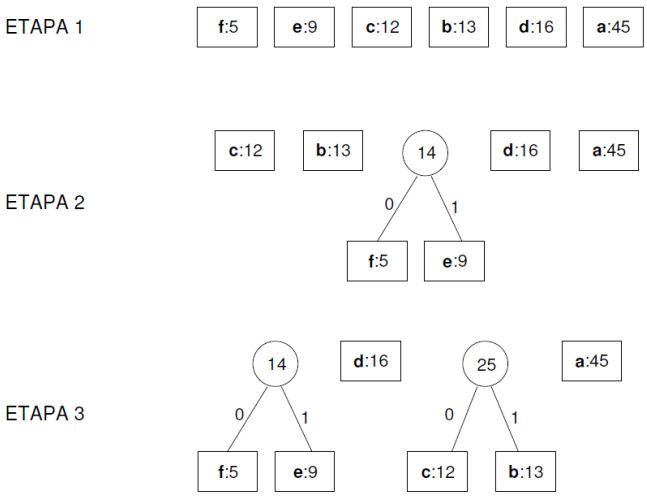

Figura 2.3.3.3 Codificación por etapas de Huffman.

Y así sucesivamente hasta llegar a obtener el árbol de codificación óptimo. Para implementar el algoritmo de los códigos de Huffman, se usa una cola de prioridad. La cola inicialmente contiene los elementos de T" y acaba conteniendo un único elemento que es el árbol de fusión construido. La frecuencia es la prioridad de los elementos de la cola y coincide con la frecuencia de la raíz del árbol. El resultado de fusionar dos elementos/árboles es un árbol cuya frecuencia es la suma de frecuencias de los dos fusionados. Su coste es $\theta(nlog(n))$ con $n = |C|$.

Algoritmo de codificación prefija de Huffman

función ejecutarCodificacionHuffman(C es conj<carácter,frecuenciaAparición>) dev (A es arbin)
{*Pre*: C está bien construido y no es vacío}
var Q es colaDePrioridad;
n := $|C|$;
Q := insertarTodos(C);
// **la cola contiene todos los elementos**
para i=1 hasta n-1 hacer
z:= arbolVacio;
// **elección de los candidatos**
x := primero(Q);
Q := avanzar(Q);
y := primero(Q);
Q := avanzar(Q);
// **construcción del nuevo árbol de fusión**
z.frecuencia := x.frecuencia + y.frecuencia;
z.hijoIzquierdo := x;
z.hijoDerecho := y;
Q := insertar(Q, z);
finpara
{*Post*: Q contiene un único elemento que es un árbol de codificación de prefijo óptimo}
dev (primero(Q))
finfunción

Demostración

Sea *T* un árbol binario de codificación prefija óptimo. Sean *b* y *c* dos hojas hermanas en *T* que se encuentran a profundidad máxima. Sean *x* e *y* dos hojas de *T* tales que son los 2 caracteres del alfabeto *C* con la frecuencia más baja. El caso interesante para la demostración se produce cuando:

$$(b, c) \neq (x, y)$$

Si *T*, que es un árbol óptimo, se puede transformar en otro árbol *T*" también óptimo, en el que los 2 caracteres, *x* e *y*, con la frecuencia más baja serán hojas hermanas que estarán a la máxima profundidad. El árbol que genera el algoritmo voraz cumple exactamente esa condición.

Se puede suponer que $f(b) \leq f(c)$ y que $f(x) \leq f(y)$. También se puede deducir que $f(x) \leq f(b)$ y $f(y) \leq f(c)$. Construyendo un nuevo árbol, *T*', en el que se intercambia la posición que ocupan en *T* las hojas b y x. Esto se muestra en la figura

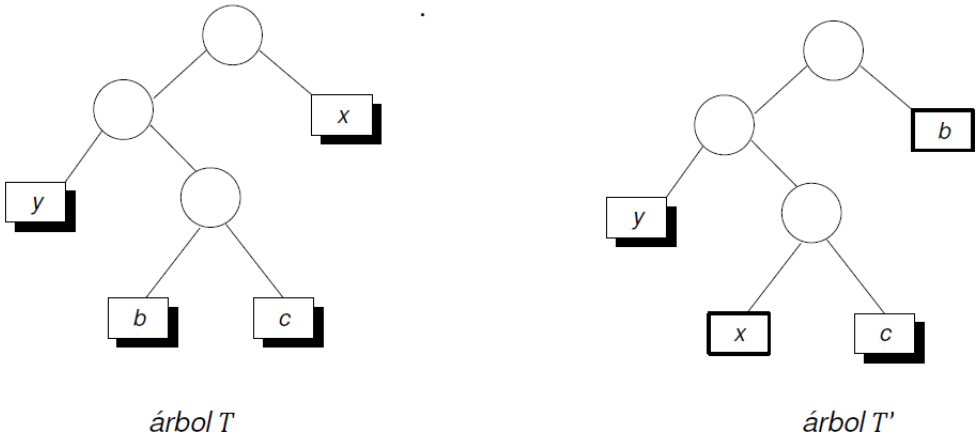

Figura 4.3.3.4 Demostración

El número de bits para el nuevo árbol de codificación *T'* lo denotamos por *B*(*T'*). Se tiene entonces que:

$$B(T) - B(T') = \sum_{c \epsilon C} f(c) \cdot d_T(c) - \sum_{c \epsilon C} f(c) \cdot d_{T'}(c)$$

$$= f(x) \cdot d_T(x) + f(b) \cdot d_T(b) - f(x) \cdot d_{T'}(x) - f(b) \cdot d_{T'}(b)$$
$$= f(x) \cdot d_T(x) + f(b) \cdot d_T(b) - f(x) \cdot d_T(b) - f(b) \cdot d_T(x)$$
$$= \big(f(b) - f(x)\big) \cdot \big(d_T(b) - d_T(x)\big) \geq 0$$

De modo que $B(T) \geq B(T')$ pero dado que *T'* es óptimo, mínimo, *T'* no puede ser menor y $B(T) = B(T')$ por lo que *T'* también es óptimo. De forma similar se construye el árbol *T"* intercambiando c e y como se muestra en la figura 4.3.3.5.

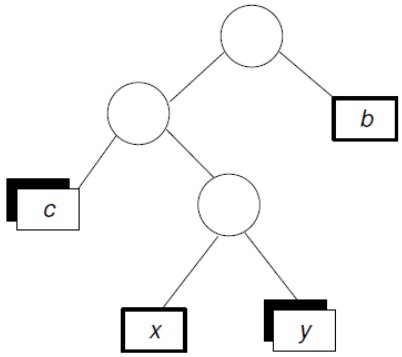

Figura 4.3.3.5 Árbol *T"*

Con este intercambio tampoco se incrementa el coste y $B(T) - B(T'') \geq 0$. Por tanto, $B(T'') = B(T)$ y como *T* es óptimo, entonces *T"* también lo es. Y ya para terminar con la demostración:

Sea *T* un árbol binario que representa un código prefijo óptimo para un alfabeto *C*. Considere dos hojas hermanas, *x* e *y*, de *T* y sea *z* el padre de ambas en *T*. Considere que la frecuencia de *z* es $f(z) = f(x) + f(y)$. Entonces, el árbol $T' = T - \{x, y\}$representa un código prefijo óptimo para el alfabeto $C' = C - \{x, y\} \cup \{z\}$.

Precisamente eso es lo que hace el algoritmo voraz: una vez que ha fusionado los dos caracteres de frecuencia más baja, inserta un nuevo elemento en el alfabeto con frecuencia de la suma de los dos

anteriores y repite el proceso de seleccionar los dos elementos con frecuencia más baja ahora para un alfabeto con un elemento menos.

Para acabar, una posible codificación en C++ como lo muestra el código 2.3.3.1

Código 2.3.3.1 Código de Huffman en C++ (huffman.cpp).

```
#include <queue>
#include <map>
#include <iostream>

using namespace std;

#define null 0
#define foreach(it,c) for (__typeof((c).begin()) it=(c).begin(); it!=(c).end(); ++it)

class Huffman {
private:
   struct node {
      node *fe, *fd, *pare;
      char c;
      double f;
      node(node* fe, node* fd, char c, double f) :
      fe(fe), fd(fd), pare(null), c(c), f(f) {
         if (fe) fe->pare = this;
         if (fd) fd->pare = this;
      }
      ~node() {
         delete fe;
         delete fd;
      }
   };
   node* arrel;
   map<char, node*> fulles;
   struct comparador {
      bool operator()(node* p, node* q) {
         return p->f > q->f;
      }
   };
   string codificar(node* p) {
      if (p->pare == null) {
         return "";
      } else if (p->pare->fe == p) {
         return codificar(p->pare) + '0';
      } else {
         return codificar(p->pare) + '1';
      }
   }
public:
   ~Huffman() {
      delete arrel;
```

```
    }
    Huffman(map<char, double>& F) {
        priority_queue<node*, vector<node*>, comparador> CP;

        foreach(it,F) {
            node* p = new node(null,null,
                        it->first,it->second);
            CP.push(p);
            fulles[it->first] = p;
        }
        while(CP.size() != 1) {
            node* p = CP.top();
            CP.pop();
            node* q = CP.top();
            CP.pop();
            CP.push(new node(p, q, ' ', p->f + q->f));
        }
        arrel = CP.top();
    }
    string decodificar(string s) {
        cout << " LEN(s) " << s.size() << " " << s << endl;
        string r;
        node* p = arrel;
        cout << " " << p->c << endl;
        unsigned i = 0;
        while (i <= s.length()) {
            //cout << " " << i << endl;
            if (p->c!=' ') {
                r += p->c;
                p = arrel;
                }
            else {
                p = s[i++] == '0' ? p->fe : p->fd;
                }
            }
        return r;
    }

    string codificar(string s) {
        string r;
        for (unsigned i = 0; i < s.size(); ++i)
            r += codificar(fulles[s[i]]);
        return r;
    }
};
int main(int argc, char *argv[]) {
    map<char,double> F;

    F['A'] = 0.35;
    F['B'] = 0.1;
```

```
    F['C'] = 0.2;
    F['D'] = 0.2;
    F['E'] = 0.15;

    Huffman h(F);
    string cadena = "ABAC";
    cout << " CADENA      : " << cadena << endl;
    string s = h.codificar(cadena);
    cout << " CODIFICADO  : " << s << endl;
    cout << " DECODIFICADO: " << h.decodificar(s) << endl;
    return 0;
}
```

TAREA

1. Hacer lo mismo que hace huffman.cpp pero en Lenguaje Java, debe ser muy parecido a este código empleando los símiles de C++ en Java.
2. Codificar un archivo de genética de la pagina

https://ftp.ncbi.nlm.nih.gov/ReferenceSamples/giab/data/AshkenazimTrio/HG002_NA24385_son/PacBio_CCS_10kb/

   Recuerde que tendrá que programar el código para abrir el archivo, contar los caracteres y codificar con huffman, guardar en un archivo la codificación.
3. Programar en C++ para que codifique el archivo y guardar lo codificado en otro archivo, comparar ambos archivos en C++ y en Java
4. Reporte sus conclusiones deberá entregar su archivo pdf, archivos codificados, código en C++ y en Java.

2.2.4 El problema de la mochila fraccionaria

El problema de la mochila: hay una persona que tiene una mochila con una cierta capacidad y tiene que elegir qué elementos llevará en ella. Cada uno de los elementos tiene un peso y aporta un beneficio al llevarlo. El objetivo de la persona es elegir los elementos que le permitan maximizar el beneficio sin excederse de la capacidad máxima permitida de carga de la mochila.

También es un problema complejo, si por complejidad nos referimos a la computacional.

“Un problema se cataloga como difícil si su solución requiere de una cantidad significativa de recursos computacionales, sin importar el algoritmo utilizado.”

El problema de la mochila forma parte de una lista de problemas NP−Completos elaborada por Richard Karp en 1972. Para el problema de la mochila, si contamos con 4 productos, para saber cuál es la mejor solución se puede probar las $2^4 = 16$ posibilidades. El 2 depende del hecho de que cada decisión es binaria, incluir o no al producto y el 4 de la cantidad de productos. Las 16 posibilidades es un número manejable, sin embargo, si la cantidad de elementos por ejemplo ascendiera a 20, tendríamos que analizar nada más y nada menos que $2^{20} = 1{,}048{,}576$ posibilidades.

Definición del problema (nuevamente)

Suponga que tenemos *n* distintos tipos de ítems (por ítem se refiere a un objeto), que van del 1 al *n*. De cada tipo de ítem se tienen q_i ítems disponibles, donde q_i es un entero positivo que cumple que q_i está definido entre

$$1 \leq q_i \leq \infty$$

Cada tipo de ítem *i* tiene un **beneficio** asociado dado por b_i y un **peso** (o volumen) p_i. Se asume que el beneficio y el peso no son negativos. Para simplificar la representación, los ítems están listados en orden creciente según el peso (o volumen).

Además, se tiene una mochila, donde se pueden introducir los ítems, que soporta un **peso máximo** (o volumen máximo) *M*.
El problema consiste en introducir en la mochila los ítems de tal forma que se **maximice el valor de los ítems que contiene y siempre que no se supere el peso máximo que puede soportar**. La **solución** al problema está dado por la secuencia de variables x_1, x_2, ..., x_n donde el valor de x_i indica cuantas copias se meterán en la mochila del tipo de ítem i.

El problema se puede expresar matemáticamente por medio del siguiente programa lineal:

Maximizar	$\sum_{i=1}^{n} b_i x_i$
Tal que	$\sum_{i=1}^{n} b_i x_i \leq M$
Y	$1 \leq q_i \leq \infty$

Si $q_i = 1\ para\ i = 1,2,3, \ldots, n$ se dice que se trata del "problema de la mochila 0-1". Si uno o más q_i es infinito entonces se dice que se trata del "problema de la mochila no acotado" también denominado "problema de la mochila entera". En otro caso se dice que se trata del problema de la mochila acotado**.**

Algoritmo

Como se vio con el problema de la mochila entera, ahora vamos a fraccionar el peso de los artículos, es decir según las unidades ahora serán en coma flotante, el siguiente es el seudocódigo.

```
insertarObjetosEnMochilaFracionaria(W, N, Objetos[], solucion[])
        desde i=0 a N
                solucion[i]=0
        peso=0
        // Ciclo voraz
        mientras peso<W
                si peso + pesoObjeto[i] <= W
                        solucion[i] = 1
                        peso = peso + pesoObejto[i]
                sino
                        solucion[i] = (W- peso) / pesoObjeto[i]
                        peso = peso + (solucion[i] * pesoObjeto[i])
        //fin bucle
```

Del seudocódigo, se infiere que en solución está la parte fraccionaria (menor a 1) del ítem que se lleva en la mochila o unitaria (igual a 1) si se lleva todo el peso del ítem. Pero para dar una solución más apegada a la realidad debemos informar al usuario que objetos deberá llevar en la mochila para satisfacer las restricciones, ellas pueden ser:

1. Tomar un conjunto de tuplas de valores (beneficio, peso) para cada objeto sin algún tipo de orden y ver qué resultado arroja el algoritmo sin sobrepasar *M*.
2. Ordenar los elementos por ejemplo según el beneficio de mayor a menor y llevar exactamente *M* peso en la mochila (Valor Máximo).
3. Peso<*M* para maximizar el número de objetos que se llevan en la mochila.
4. Mejor Rentabilidad en donde

$$Rentabilidad = \frac{beneficio}{peso}$$

El código 2.2.4.1 muestra las clases para trabajar el problema, se tiene una clase ObjetoDeMochila que almacena el peso, el beneficio y la parte fraccionaria o unitaria de la solución, otra clase que se llema MochilaVoraz, la que implementa el algoritmo solamente para el punto 1 y una clase principal para llevar a cabo la ejecución del algoritmo.

Código 2.2.4.1 Clases del algoritmo de la Mochila Fraccionaria

```
/**
 * @author sdelaot
 */
public class ObjetoDeMochila {
    private float peso;
    private float beneficio;
    private float solucion;
    public ObjetoDeMochila( float peso, float beneficio ) {
        this.peso = peso;
        this.beneficio = beneficio;
        solucion = 0.0F;
    }
    public float getPeso() {
        return peso;
    }
    public float getBeneficio() {
        return beneficio;
    }
    public void setSolucion(float solucion) {
        this.solucion = solucion;
    }
    @Override
    public String toString() {
        return "ObjetoDeMochila{" + "peso=" + peso + ", beneficio=" + beneficio + ", solucion=" + solucion + '}';
    }
}
import java.util.LinkedList;
/**
 * @author sdelaot
 */
```

```
public class MochilaVoraz {
   LinkedList<ObjetoDeMochila> objetos;
   public MochilaVoraz() {
      this.objetos = new LinkedList<>();
   }
   public void insertarObjetosEnMochilaFraccionaria( int n, int M,
         ObjetoDeMochila [] objetosM, float sol[] ) {
      int i;
      for(i=0;i<n;i++) {
         sol[i]=0;
         }

      float pesoActual=0;
      for(i=0;pesoActual<M && i<n;i++){
         if(pesoActual+objetosM[i].getPeso()<=M) {
            sol[i]=1;
            }
         else {
            sol[i]+=(M-pesoActual)/objetosM[i].getPeso();
            }
         if( sol[i]>0.0 ) {
            objetos.add(objetosM[i]);
            }
         objetosM[i].setSolucion(sol[i]);
         pesoActual += sol[i]*objetosM[i].getPeso();
         }
   }
   public LinkedList<ObjetoDeMochila> getObjetos() {
      return objetos;
   }
}
import java.util.LinkedList;
/**
 * @author sdelaot
 */
public class ProbadorDeMochila {
   public static void main(String[] args) {
      float [] pesos      = { 60, 40, 20, 30, 10, 50 };// = 100
      float [] beneficios = { 50, 40, 30, 66, 20, 60 };// = 155
      ObjetoDeMochila [] objetos =  {
         new ObjetoDeMochila( pesos[0], beneficios[0] ),
         new ObjetoDeMochila( pesos[1], beneficios[1] ),
         new ObjetoDeMochila( pesos[2], beneficios[2] ),
         new ObjetoDeMochila( pesos[3], beneficios[3] ),
         new ObjetoDeMochila( pesos[4], beneficios[4] ),
         new ObjetoDeMochila( pesos[5], beneficios[5] )
         };
      MochilaVoraz mochila = new MochilaVoraz();
      float [] solucion = new float[pesos.length];
      int n = pesos.length;
```

```
        int M = 100;
        mochila.insertarObjetosEnMochilaFraccionaria(n, M, objetos, solucion);
        LinkedList<ObjetoDeMochila> objetosInsertados = mochila.getObjetos();
        for( ObjetoDeMochila odm : objetosInsertados ) {
           System.out.println( odm );
        }
    }
}
```

TAREA

Hay que programar los otros tres puntos que no se programaron (2, 3 y 4), realizar el ordenamiento empleando quicksort para los objetos de mochila (ObjetoDeMochila), cambiar “float solución []” en la llamada al método voraz por “ObjetoDeMochila [] objetos”. Entregar código fuente y reporte de conclusiones (pdf)