



# PRÁCTICA 4 PROGRAMACIÓN DINÁMICA



Algoritmica

Mercedes León Chaves

Lydia vanDillewijn Soto

Laura Zafra Alarcos



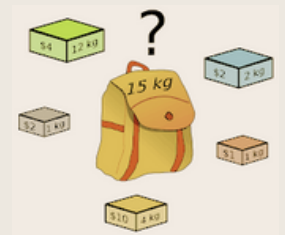
# ÍNDICE

|   |    |
|---|----|
| <b>RESUMEN DEL PROBLEMA</b>             | 03 |
| • Introducción                          |    |
| <b>REQUISITOS PROGRAMACIÓN DINÁMICA</b> | 04 |
| • Naturaleza n-etápica                  |    |
| • Verificación del POB                  |    |
| • Planteamiento de una recurrencia      |    |
| <b>ECUACIÓN RECURRENTE</b>              | 05 |
| • Objetivo                              |    |
| • Resultado directo                     |    |
| • Ecuación                              |    |
| <b>REPRESENTACIÓN DE LA ECUACIÓN</b>    | 06 |
| • Tabla                                 |    |
| <b>P.O.B</b>                            | 07 |
| <b>DISEÑO DEL ALGORITMO</b>             | 08 |
| • Uso de programación dinámica,         |    |
| <b>IMPLEMENTACIÓN</b>                   | 09 |
| • En C++                                |    |
| <b>ANÁLISIS DE EFICIENCIA</b>           | 12 |
| • Eficiencia teórica                    |    |
| • Eficiencia práctica                   |    |
| <b>INSTANCIAS DEL PROBLEMA</b>          | 15 |
| <b>CASOS EXTREMOS</b>                   | 17 |
| <b>COMPILACIÓN Y EJECUCIÓN</b>          | 18 |

# INTRODUCCIÓN

## PROBLEMA

Imagina que eres un ladrón de clase mundial y acabas de entrar a robar en una casa con muchos artículos valiosos. Has traído una mochila, pero en ella solo puedes transportar una cantidad limitada de peso. Tu objetivo es salir con el valor combinado más alto de artículos que quepan en la mochila, pero ¿Cómo eliges estos artículos y cuál es el valor óptimo? Este es el problema de la mochila.



## DESCRIPCIÓN

Tenemos una mochila con una capacidad de peso máximo **M**, y un conjunto de **n** objetos a transportar.

Cada objeto **i** tiene un peso **w<sub>i</sub>** y llevarlo supone un beneficio **b<sub>i</sub>**.

El problema consiste en seleccionar qué objetos incluir en la mochila de modo que se maximice el beneficio, sin superar la capacidad de la misma, sabiendo que los objetos son indivisibles en fracciones y sólo se puede seleccionar llevar el objeto completo o no llevarlo. Si llevamos el objeto **i**, entonces **x<sub>i</sub>=1**. En caso contrario, **x<sub>i</sub>=0**.

$$\text{Maximizar } \sum_{i=1}^n x_i b_i \quad \text{sujeto a} \quad \sum_{i=1}^n x_i w_i \leq M$$

|  |   |
|--|---|
| Capacidad Máxima de la Mochila   | M   |
| Número de Objetos  | n=1,2,...,n   |
| Beneficio de llevar cada objeto  | b <sub>i</sub>  |
| Peso de cada objeto  | w <sub>i</sub>  |
| Objetos ordenados de menor a mayor   | b <sub>i</sub> /w <sub>i</sub>  |
| Función objetivo   | Maximizar $\sum_{i=1}^n x_i b_i$ sujeto a $\sum_{i=1}^n x_i w_i \leq M$ |
| Las variables para llevar 1 o 0 en el objeto i   | X <sub>i</sub>  |
| Beneficio de considerar llevar los objetos desde el 1 al i, sabiendo que la capacidad de la mochila es j | T(i,j)  |

# REQUISITOS PD



## NATURALEZA N-ETÁPICA

Este problema se puede resolver por etapas: En cada etapa  $i$  seleccionaremos llevar (o no) el objeto  $i$ , considerando la capacidad restante de la mochila. Si lo llevamos, restaremos su peso a  $j$ .



## VERIFICACIÓN DEL POB

POB (Principio de Optimalidad de Bellman): Una política óptima tiene la propiedad de que sean cuales sea el estado inicial y la decisión inicial, las decisiones restantes deben constituir una solución óptima con respecto al estado resultante de la primera decisión.

Cumple este requisito, esta desarrollado en el apartado 4.



## PLANTEAMIENTO DE UNA RECURRENCIA

En este se puede plantear una ecuación de recurrencia que represente la forma de ir logrando etapa por etapa la solución óptima. Esto quiere decir que se resuelve mediante recurrencia, para que la eficiencia del problema sea la más óptima posible.

Cumple este requisito, esta desarrollado en el apartado 2.



## CASOS BASE

Como vemos más tarde, existen casos base (cuando la capacidad de la mochila es 0 y cuando hay un solo objeto disponible).



## PROBLEMA DE OPTIMIZACIÓN

Nuestro objetivo será llevar la combinación de objetos que maximice el beneficio sujeto a la restricción de no superar la capacidad de la mochila.



# ECUACIÓN EN RECURRENCIAS

Necesitamos obtener una ecuación que nos proporcione el máximo beneficio de llevar o no llevar un conjunto de  $n$  objetos (ordenados de 1 a  $n$  como se ha planteado anteriormente, de menor a mayor razón valor/peso) teniendo en cuenta la capacidad máxima de la mochila. Es decir, es necesario encontrar una ecuación  $T(n,M)$ , donde  $n$  es el subconjunto de objetos y  $M$  la capacidad máxima de la mochila, que será constante.

Vemos a continuación los componentes de esta ecuación que, naturalmente, al tratarse de un problema resoluble mediante programación dinámica, será recurrente.

## Caso base (solución directa):

- $T(i,0)$ : la mochila no puede soportar nada de peso, no podemos llevar ningún objeto y el beneficio por tanto es 0.
- $T(1,M)$ : solo consideramos el primer objeto. Podremos llevarlo si su peso es menor o igual a  $M$ . En caso de llevarlo, el beneficio es  $b_1$  ( $T(1,M) = b_1$ ). En caso contrario, no hay beneficio ( $T(1,M) = 0$ ).
- Consideramos no válida cualquier solución en la que la capacidad de la mochila sea menor que 0. Supondremos que la capacidad de la mochila será positiva en los casos planteados.

## Caso general (el resultado se calcula a partir de resultados anteriores):

Considerando un conjunto de objetos del 1 al  $i$ , y una capacidad de la mochila  $j$ , el máximo beneficio se conseguirá de alguna de las dos siguientes formas:

- No llevar el objeto  $i$  si no cabe en la mochila, robar los objetos numerados de 1 a  $i-1$ , es decir,  $T[i-1][j]$ . No se modifica la capacidad restante de la mochila ni el beneficio acumulado.
- Llevar el objeto  $i$  si este cabe en la mochila. Se debe restar su peso  $w$  de la capacidad restante de la mochila  $j$  ( $j-w_i$ ). Además, debemos sumar al beneficio que ya llevamos el beneficio que supone el objeto,  $b_i$ . Posteriormente, vemos si podemos llevar los objetos del 1 a  $i-1$ . Esta parte de la ecuación queda:  $b_i + T[i-1][j-w_i]$

Por lo tanto, la ecuación en recurrencias buscada es:

$$T[i][j] = \max\{T[i-1][j], b_i + T[i-1][j-w_i]\}$$

# REPRESENTACIÓN

## DE LA ECUACIÓN DE RECURRENCIAS

Utilizando la ecuación de recurrencia  $T[i][j] = \max\{T[i-1][j], b_i + T[i-1][j-w_i]\}$ , se representa a continuación una tabla que **cuantifica el valor máximo (óptimo)** que se puede llegar a obtener cogiendo artículos dependiendo de su peso y valor. Los valores que aparecen son acumulativos y son **óptimos locales**.

### Datos que aparecen en la tabla

- **n** = El **número** de artículos
- **b** = El **valor** de cada artículo
- **w** = El **peso** de cada artículo
- **M** = El **peso total** que puede soportar la mochila

La **posición** de cada artículo en la tabla se debe a su **relación valor/peso**, estando ordenados de menor a mayor.

Si por ejemplo se toman los valores:

- **n** = 5
- **b** = {1, 6, 18, 22, 28}
- **w** = {1, 2, 5, 6, 7}
- **M** = 10

| n \ M                | 0 | 1 | 2 | 3 | 4 | 5  | 6  | 7  | 8  | 9  | 10 |
|----------------------|---|---|---|---|---|----|----|----|----|----|----|
| 0                    | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  |
| 1<br>$b=1$<br>$w=1$  | 0 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  | 1  |
| 2<br>$b=6$<br>$w=2$  | 0 | 1 | 6 | 7 | 7 | 7  | 7  | 7  | 7  | 7  | 7  |
| 3<br>$b=18$<br>$w=5$ | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 |
| 4<br>$b=22$<br>$w=6$ | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 |
| 5<br>$b=28$<br>$w=7$ | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 35 |

### Para rellenar la tabla

Si se visualiza la tabla como una matriz donde se cuantifica el beneficio ( $T[n][M]$ ):

Desde la posición  $[n][M]$  en la que se está (exceptuando inicialización), se calcula si se toma o no el objeto considerado ( $n$ ).

Para ello, se calculan por separado la parte derecha ( $b[n] + T[n-1][j-w_n]$ ) e izquierda ( $T[n-1][j]$ ) de la ecuación de recurrencias:

- Si  $(b[n] + T[n-1][j-w_n]) > T[n-1][M] \rightarrow$  Se coge el artículo  $T[n][M] = (b_n + T[n-1][M-w_n])$
- Si  $(b[n] + T[n-1][j-w_n]) < T[n-1][M] \rightarrow$  No se coge el artículo ( $T[n-1][M]$ ) (se arrastra el valor que se tenía en la mochila)

Por eso, el valor con el que se va rellenando es  $T[i][j] = \max\{T[i-1][j], b_i + T[i-1][j-w_i]\}$ , porque se va tomando el valor que da más beneficio.

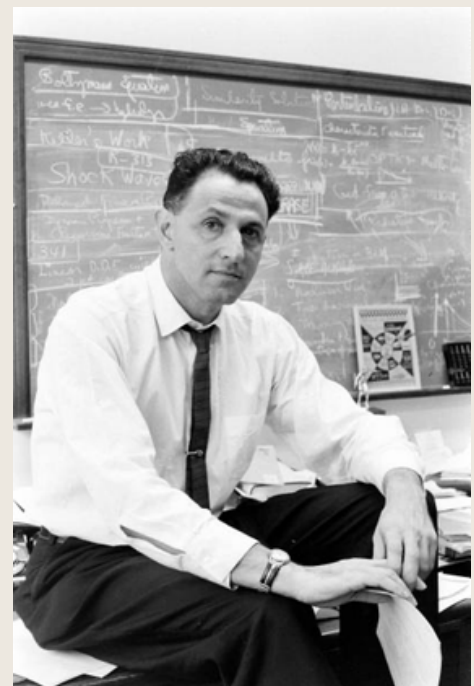
# PRINCIPIO DE OPTIMALIDAD DE BELLMAN

Este principio dicta que si una secuencia de pasos para resolver un problema es óptima, entonces cualquier subsecuencia de estos mismos pasos también es óptima. Aplicado a nuestro problema,  $T[i][j]$  será óptimo si las decisiones tomadas anteriormente para resolver  $T[i-1][j]$  y  $T[i-1][j-w_i]$  también lo son.

Para demostrar su cumplimiento, fijamos como caso base  $T[1][j]$ , donde  $j$  es una capacidad fija de la mochila. Se escoja o no este elemento (quepa o no en la mochila), el resultado (beneficio) obtenido es un óptimo. Solo hay un objeto que será escogido o no en función de que quepa en la mochila. En el caso  $T[2][j]$  nos encontramos con otro óptimo. Elegiremos llevar los dos objetos si la suma de sus respectivos pesos es menor a la capacidad de la mochila  $j$ , o el objeto uno o ninguno si este límite se supera.

Haciendo uso de inducción, esto se cumple hasta llegar al caso  $T[i-1][j]$ . Además, no es posible llegar a una solución no óptima, puesto que la ecuación recurrente busca maximizar el beneficio en cada paso. Sabiendo que  $T[i-1][j]$  es un óptimo, se demuestra de la misma forma que  $T[i-1][j-p_i]$  es también un óptimo.

"Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve"



# DISEÑO DEL ALGORITMO

## SOLUCIÓN AL PROBLEMA

### NO HAY OBJETOS / CAPACIDAD

Si **no hay elementos** o **no hay capacidad**, no hay beneficio posible, no son necesarios más cálculos:

- Recorrer todas las capacidades (columnas) de la tabla: si no hay elementos ( $i=0$ , para toda la primera fila), sea cual sea la capacidad el beneficio es 0.
- Recorrer todos los objetos (filas) de la tabla: si la capacidad es 0, ( $w_i=0$ , para toda la primera columna), sea cual sea el conjunto de objetos, el beneficio es 0.

### A PARTIR DE CAPACIDAD=1, Y N\_OBJETOS=1

Recorrer cada columna, desde 1 hasta M (capacidad límite total de la mochila)

Recorrer cada fila, desde 1 hasta n (objetos totales a considerar, que recordemos, están ordenados)

- Si la capacidad disponible es menor que el peso del objeto que se considera
  - No se coge el objeto considerado, se toma el beneficio para el objeto en la fila anterior en la tabla para la capacidad establecida
- Si la capacidad disponible es mayor o igual que el peso del objeto considerado
  - Calcular la parte izquierda de la ecuación de recurrencias ( $T[i-1][j]$ )
  - Calcular la parte derecha de la ecuación de recurrencias ( $b_i + T[i-1][j-w_i]$ )
  - Si el resultado de la parte izquierda es mayor que el de la parte derecha
    - El beneficio máximo que se obtiene es el mismo que si no se coge el elemento, es decir,  $T[i][j]=T[i-1][j]$
  - Si el resultado de la parte derecha es mayor que el de la parte izquierda
    - El beneficio máximo que se obtiene es el obtenido en el cálculo de la segunda parte de la ecuación, es decir,  $T[i][j]=b_i + T[i-1][j-w_i]$



# IMPLEMENTACIÓN I

```
1  #include <iostream>
2
3  #include <fstream>
4
5  using namespace std;
6
7  //Capacidad de la mochila
8  const int M = 12;
9  //Número de objetos
10 const int n = 6;
11
12 void lecturaObjetos(const char fichero[], int b[], int w[], int n) {
13     ifstream fi;
14     //Abro fichero
15     fi.open(fichero);
16
17     if (fi) { //si no hay errores
18         cout << endl << "Leyendo el fichero " << fichero << endl;
19         int beneficio, peso;
20
21         b[0] = 0;
22         w[0] = 0;
23
24         //Leo cada pareja de valores
25         for (int i = 1; i < n; i++) {
26             fi >> beneficio >> peso;
27             b[i] = beneficio;
28             w[i] = peso;
29         }
30     } else cout << "Error en la lectura del fichero" << endl;
31
32     //Cierro fichero
33     fi.close();
34 }
```

# IMPLEMENTACIÓN II

```
36- int Mochila(int n, int M, int b[], int w[]) {
37     int T[n][M];
38
39     //Inicialización
40     //Si no hay elementos
41-   for (int j = 0; j < M; j++) {
42       T[0][j] = 0;
43   }
44   //Si la capacidad es 0
45-   for (int i = 0; i < n; i++) {
46       T[i][0] = 0;
47   }
48
49-   for (int j = 1; j < M; j++) {
50-       for (int i = 1; i < n; i++) {
51           //Si la capacidad disponible es menor que el peso del objeto que se considera
52-           if (j < w[i]) {
53               T[i][j] = T[i - 1][j];
54-           } else {
55               int izda = T[i - 1][j];
56               int dcha = b[i] + T[i - 1][j - w[i]];
57
58-               if (izda > dcha) {
59                   //No se coge el objeto considerado
60                   T[i][j] = izda;
61-               } else {
62                   //Se coge el objeto considerado
63                   T[i][j] = dcha;
64               }
65           }
66       }
67   }
68
69   return T[n - 1][M - 1];
70 }
72- int main() {
73
74-   if (n <= 1 || M <= 1) { //hay solo un elemento que es el objeto/capacidad nulos
75       cout << "No hay beneficio." << endl;
76       return 0;
77   }
78
79   //Atributos de los objetos
80   int b[n]; //valor
81   int w[n]; //peso
82
83   //Función de lectura
84   lecturaObjetos("ejemplocorto.txt", b, w, n);
85
86   //Algoritmo de la mochila
87   cout << "El beneficio máximo que se puede obtener es: ";
88   cout << Mochila(n, M, b, w) << endl;
89 }
```

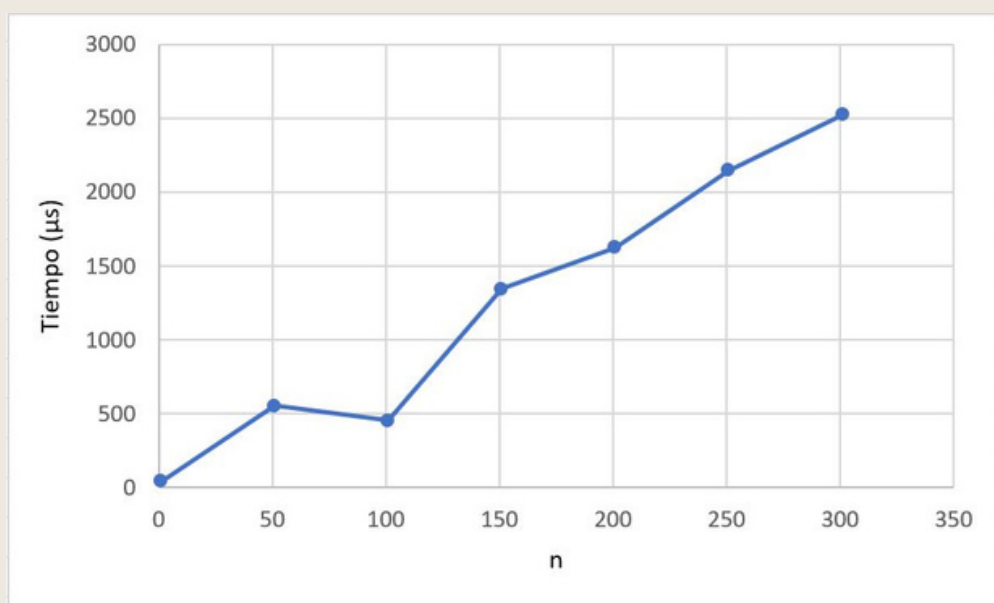
# ANÁLISIS DE EFICIENCIA

## EFICIENCIA PRÁCTICA

Para representar la eficiencia práctica, se ha calculado el tiempo que tarda el algoritmo para diferentes tamaños de  $n$ . Estos tiempos se han calculado gracias a las funciones que aporta la librería `chrono`.

Como se puede comprobar, a medida que aumenta el tamaño de  $n$ , también lo hace el tiempo de ejecución del algoritmo casi de manera lineal.

| $n$ | Tiempo ( $\mu s$ ) |
|-----|--------------------|
| 1   | 44                 |
| 51  | 554                |
| 101 | 453                |
| 151 | 1345               |
| 201 | 1625               |
| 251 | 2148               |
| 301 | 2525               |



# ANÁLISIS DE EFICIENCIA

## EFICIENCIA TEÓRICA

```
1 #include <iostream>
2
3 #include <fstream>
4
5 using namespace std;
6
7 //Capacidad de la mochila
8 const int M = 12; | o(1)
9 //Número de objetos
10 const int n = 6; | o(1)
11
12 void lecturaObjetos(const char fichero[], int b[], int w[], int n) {
13     ifstream fi; | o(1)
14     //Abro fichero
15     fi.open(fichero); | o(1)
16
17     if (fi) { //si no hay errores
18         cout << endl << "Leyendo el fichero " << fichero << endl; | o(1)
19         int beneficio, peso; | o(1)
20
21         b[0] = 0; | o(1)
22         w[0] = 0; | o(1)
23
24         //Leo cada pareja de valores
25         for (int i = 1; i < n; i++) {
26             fi >> beneficio >> peso; | o(1)
27             b[i] = beneficio; | o(1)
28             w[i] = peso; | o(1)
29         }
30     } else cout << "Error en la lectura del fichero" << endl; | o(1)
31
32     //Cierro fichero
33     fi.close(); | o(n)
34 }
```

$O(n)$

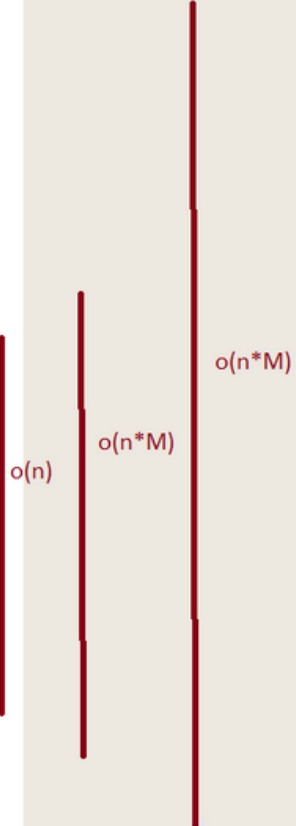
$O(n)$

$O(n)$

Como podemos ver en esta parte del código, en la función para leer desde fichero los datos para poder ejecutar el programa, tendríamos una eficiencia de  $O(n)$ , ya que está compuesto de líneas donde se producen asignaciones cuya eficiencia es de  $O(1)$ , además cuenta con un 'if' en el que su eficiencia estaría determinada por la parte de mayor peso, por un lado en el 'else' tendríamos una eficiencia de  $O(1)$ , pero en el if sería de  $O(n)$  ya que esta compuesto por un 'for' que sus ejecuciones dependen de la variable 'n'. Por tanto esta parte tiene eficiencia  $O(n)$ .

# ANÁLISIS DE EFICIENCIA

```
36- int Mochila(int n, int M, int b[], int w[]) {
37-     int T[n][M]; o(1)
38-
39-     //Inicialización
40-     //Si no hay elementos
41-     for (int j = 0; j < M; j++) {
42-         T[0][j] = 0; o(1)
43-     } o(M)
44-     //Si la capacidad es 0
45-     for (int i = 0; i < n; i++) {
46-         T[i][0] = 0; o(1)
47-     } o(n)
48-
49-     for (int j = 1; j < M; j++) {
50-         for (int i = 1; i < n; i++) {
51-             //Si la capacidad disponible es menor que el peso del objeto que se considera
52-             if (j < w[i]) {
53-                 T[i][j] = T[i - 1][j]; o(1)
54-             } else {
55-                 int izda = T[i - 1][j]; o(1)
56-                 int dcha = b[i] + T[i - 1][j - w[i]]; o(1)
57-
58-                 if (izda > dcha) {
59-                     //No se coge el objeto considerado
60-                     T[i][j] = izda; o(1)
61-                 } else {
62-                     //Se coge el objeto considerado
63-                     T[i][j] = dcha; o(1)
64-                 }
65-             }
66-         }
67-     }
68-
69-     return T[n - 1][M - 1]; o(1)
70- }
```



El código utiliza dos bucles anidados para recorrer todos los elementos y todas las capacidades posibles de la mochila. La variable 'n' representa el número de elementos y la variable 'M' representa la capacidad máxima de la mochila. Por lo tanto, el número total de iteraciones será proporcional a  $n * M$ .

Dentro de los bucles, hay operaciones básicas como asignaciones y comparaciones. Estas operaciones tienen una complejidad constante, por lo que no afectan significativamente la complejidad total del algoritmo.

En términos de complejidad temporal, el código tiene una complejidad de  $O(n * M)$ , lo que significa que su tiempo de ejecución aumenta proporcionalmente al producto de  $n$  y  $M$ . Si  $n$  y  $M$  son grandes, el algoritmo puede volverse lento.

Cabe destacar que la eficiencia de este código también puede depender de los valores específicos en los arreglos 'b' y 'w'. Si estos arreglos son grandes o contienen valores extremadamente grandes, podría haber un impacto en el rendimiento.

En resumen, la eficiencia del código de la mochila que has proporcionado es de  $O(n * M)$  en términos de complejidad temporal.

# ANÁLISIS DE EFICIENCIA

```
72- int main() {  
73-  
74-   if (n <= 1 || M <= 1) { //hay solo un elemento que es el objeto/capacidad nulos  
75-       cout << "No hay beneficio." << endl; o(1)  
76-       return 0; o(1)  
77-   }  
78-  
79-   //Atributos de los objetos  
80-   int b[n]; //valor o(1)  
81-   int w[n]; //peso o(1)  
82-  
83-   //Función de lectura  
84-   lecturaObjetos("ejemplo corto.txt", b, w, n); o(n)  
85-  
86-   //Algoritmo de la mochila  
87-   cout << "El beneficio máximo que se puede obtener es: "; o(1)  
88-   cout << Mochila(n, M, b, w) << endl; o(n*M)  
89- }
```

$o(1)$

$o(n*M)$

El main esta formado como podemos ver, por el 'if' que va a tener una eficiencia constante porque esta formado comprobaciones y una salida de texto.

Tiene un par de asignaciones que tambien tienen una eficiencia constante.

Y se llama a la función de lectura de eficiencia  $o(n)$  y el algoritmo de la mochila que tiene una eficiencia de  $o(n*M)$ , por lo que podemos decir que el programa tiene una eficiencia de  $o(n*M)$

## Eficiencia Final:

# $O(n*M)$

# INSTANCIAS I

## EJEMPLO CORTO (5 OBJETOS)

Hemos implementado este ejemplo para comprobar el funcionamiento del algoritmo. En él tenemos 5 objetos con los pesos y los precios marcados en la imagen. Debemos optimizar el beneficio sin sobrepasar el límite de peso de la mochila, es decir 11kg. El resultado que obtenemos es que el beneficio máximo es de 40€. En este caso se habrían escogido el teléfono y la cámara y se habrían omitido el resto de objetos.

Leyendo el fichero `ejemplo corto.txt`  
El beneficio máximo que se puede obtener es: 40



Además, antes de la ejecución del algoritmo principal Mochila, se comprueba que tanto la capacidad de la mochila como el número de elementos no sean nulos o negativos. En tal caso, se obtiene el mensaje siguiente y finaliza la ejecución.

“No hay beneficio.”

## INSTANCIAS DE MAYOR TAMAÑO

A continuación, presentamos dos ejemplos de mayor tamaño sobre los que hemos aplicado el algoritmo. En ambos casos, los valores y pesos de los objetos se han generado de forma aleatoria. Como se requería incorporar los datos ya ordenados al algoritmo, con ayuda de una hoja de cálculo (en nuestro caso Excel) se han obtenido los números requeridos en cada caso. Se ha calculado la razón beneficio/coste para cada objeto y se han proporcionado ordenados de menor a mayor por esta relación de optimalidad al algoritmo. A continuación vemos los dos ejemplos que se han elaborado y los respectivos resultados de sus ejecuciones.

# INSTANCIAS II

## COMPOSICIÓN DE LAS INSTANCIAS

- El primer ejemplo consta de 50 elementos, con un tamaño de mochila de límite 100. En este el valor de los elementos se mueve en el rango 1-50 y el peso en el rango 1-10. En la primera imagen se puede apreciar parte de la hoja de cálculo donde se ha realizado el ordenamiento de los objetos. La primera y la segunda columna corresponden al valor y el beneficio del objeto, respectivamente, generados aleatoriamente en el rango estimado. La tercera columna es el resultado de dividir el valor entre el peso de cada objeto. Posteriormente, las dos primeras columnas se han copiado a un archivo de texto para su procesamiento.
- El segundo ejemplo consta de 300 elementos, con un tamaño de mochila de límite 1000. En este el valor de los elementos se mueve en el rango 1-50 y el peso en el rango 1-10. Se ha procedido del mismo modo que con el segundo ejemplo. La segunda imagen muestra dos fragmentos del archivo txt que contiene la información.

|      |    |            |
|------|----|------------|
| 7    | 45 | 6.42857143 |
| 5    | 33 | 6.6        |
| 3    | 20 | 6.66666667 |
| 5    | 34 | 6.8        |
| 4    | 28 | 7          |
| 6    | 45 | 7.5        |
| 3    | 23 | 7.66666667 |
| 5    | 43 | 8.6        |
| 4    | 35 | 8.75       |
| 5    | 48 | 9.6        |
| 5    | 50 | 10         |
| 3    | 42 | 14         |
| 1    | 17 | 17         |
| 1    | 25 | 25         |
| 1    | 27 | 27         |
| 1    | 30 | 30         |
| 1    | 32 | 32         |
| 1    | 35 | 35         |
| peso |    | beneficio  |
|      |    | b/w        |

|   |    |   |    |
|---|----|---|----|
| 1 | 10 | 2 | 6  |
| 1 | 8  | 1 | 3  |
| 1 | 7  | 4 | 8  |
| 1 | 6  | 5 | 9  |
| 2 | 10 | 4 | 7  |
| 1 | 5  | 6 | 10 |
| 2 | 10 | 6 | 10 |

## RESULTADOS DE LA EJECUCIÓN

Leyendo el fichero M101-n51.txt  
El beneficio máximo que se puede obtener es: 51

ejemplo 1

Leyendo el fichero M1001-n301.txt  
El beneficio máximo que se puede obtener es: 6602

ejemplo 2



# CASOS EXTREMOS

En el caso en el cual la capacidad de la mochila sea muy grande (o muy pequeña), por ejemplo 1.000.000, a la hora de hacer el algoritmo tendríamos un problema de ocupación de la memoria a la hora de hacer la tabla para la representación de este, por lo que tendríamos que realizar un cambio de escala en estos caso. De este modo se puede construir una tabla más asequible a la hora de ocupar espacio en memoria.

Para ello podríamos implementar una parte en el código para resaltar que los pesos de los objetos impuestos y la capacidad de la mochila deben tener la misma medida y escala. Además de que deben tener tamaños asequibles para la memoria. Tomamos como límite que la capacidad de la mochila debe ser de como máximo 100 unidades. De este modo evitamos el problema de los límites o capacidades disponibles en el computador.

Por lo que podemos poner un 'if' con límites en las columnas, para que cuando se sobresalga el límite establecido, para el programa para que el usuario meta estos datos de forma correcta, teniendo en cuenta lo anterior.

Con ello podemos resolver el único problema de la programación dinámica, que es la ocupación de memoria según la tabla preestablecida.

| <div>÷ ← LONGITUD → ×</div>  |            |           |       |           |            |           |
|---|------------|-----------|-------|-----------|------------|-----------|
| KM  | HM         | DAM       | m     | dm        | cm         | mm        |
| Kilómetro   | hectómetro | decámetro | metro | decímetro | centímetro | milímetro |
|   |            |           |       |           |            |           |

# COMPILACIÓN Y EJECUCIÓN

En primer lugar, hemos analizado el problema y diseñado el algoritmos para resolverlo. Posteriormente lo hemos implementado en lenguaje C++, en nuestro caso en el IDE CLion, con el que también hemos compilado y ejecutado el código.

Para nuestro análisis práctico, hemos necesitado medir los tiempos de ejecución del algoritmo. Con esta finalidad hemos empleado la biblioteca 'chrono', que nos permite cuantificar con precisión de microsegundos la hora del computador.

Con el objetivo de obtener una representación gráfica de la eficiencia práctica, hemos realizado varias mediciones para el algoritmos, cambiando los tamaños de  $n$  (número de objetos). Hemos partido desde un número de objetos 1, yendo probando de 50 en 50, hasta llegar al último número de objetos comprobado, que ha sido 301.

## FICHEROS

Para el algoritmos hemos obtenido el valor y el peso de los distintos objetos del problema a través de dos ficheros "M101-51.txt" y "M1001-n301.txt". Como hemos podido ver antes en la implementación, la lectura e inicialización del conjunto de valores se ha realizado a través de la función lecturaObjeto, que usaba el fichero como flujo de entrada.



El primer archivo esta formado por dos columnas, en donde la primera representa el peso de cada objeto , y la segunda representa el valor de cada objeto. Los valores escogidos han sido formados aleatoriamente , en el peso[1-10] y el valor [1-50]. Además el primero estaría formado por 50 objetos y el segundo por 300 objetos.

El ejecutable y los ficheros aportados tienen que estar en el mismo directorio en el momento de ejecución, o bien dejarse indicada la ruta en la que se encuentra el archivo con los puntos desde el código.



# COMPILACIÓN Y EJECUCIÓN



## MEDICIÓN DE TIEMPOS

```
/*
Como hemos mencionado antes, hemos usado la biblioteca chrono para medir el tiempo del
algoritmo trabajado en la práctica. Lo hemos hecho de la siguiente manera:
*/
//Hemos incluido la biblioteca necesaria para medir con precisión de microsegundos

#include <chrono>

//Hemos declarado las variables t0 y tf
chrono::time_point<std::chrono::high_resolution_clock> t0, tf;

/*
...
Lectura del fichero de entrada
...
*/

//t0 marca y almacena la hora de inicio de la ejecución del algoritmo
t0 = std::chrono::high_resolution_clock::now();

/*
...
Ejecución del algoritmo: llamada Mochila
...
*/

//tf marca y almacena la hora del final de la ejecución del algoritmo
tf = std::chrono::high_resolution_clock::now();

//Cálculo del tiempo total que se ha necesitado para completar el proceso
unsigned long duration = std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();

//Imprimimos la duración por terminal para posteriormente procesar los datos y construir la...
//gráfica que muestra el umbral

cout << "Tiempo de ejecución: " << duration << " microsegundos." << endl;
```