

PRÁCTICA 4: ALGORITMOS DE PROGRAMACIÓN DINÁMICA



UNIVERSIDAD
DE GRANADA

JAVIER BUENAVENTURA MARTÍN JIMÉNEZ
EMILIO GUILLÉN ÁLVAREZ
ALBERTO RODRÍGUEZ FERNÁNDEZ

GUIÓN

- Requisitos de programación dinámica.
- Ecuación recurrente del problema.
- Representación de la ecuación mediante una tabla que almacene las soluciones parciales.
- Comprobar el cumplimiento del Principio de Optimalidad de Bellman.
- Pseudocódigo del algoritmo de Programación Dinámica.
- Implementación del código.
- Eficiencia teórica y práctica del algoritmo.
- Dos instancias del ejemplo concreto.



1. Comprobar si el problema cumple con los requisitos para poder resolverse mediante la técnica de programación dinámica.

La programación dinámica es una técnica algorítmica que es comúnmente utilizada para la resolución de problemas en los cuales se busca optimizar la respuesta y elegir la correcta de entre varias posibles.

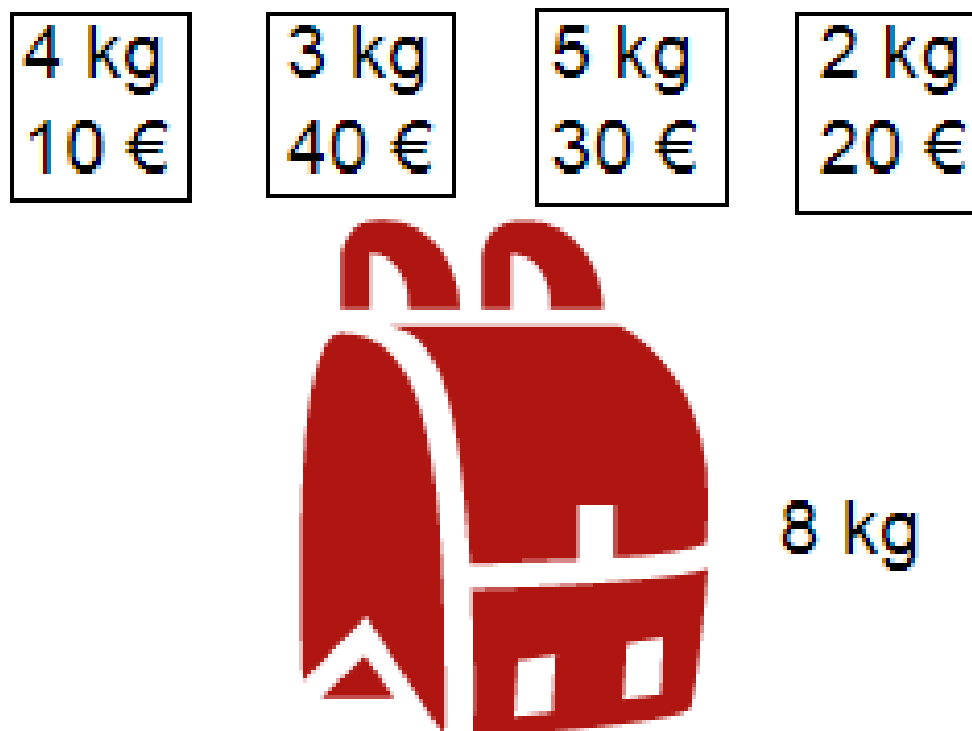
Sabemos que el problema de la mochila consiste a rasgos generales en seleccionar un conjunto de elementos de entre un grupo el cual se nos presenta, cada elemento con un peso y un valor asociados, buscando como solución, maximizar el valor del conjunto de productos sin exceder de la capacidad de nuestra supuesta mochila.

Para determinar si este problema es aplicable a la programación dinámica vemos si cumple con una serie de requisitos:

1. Subestructura óptima: El problema de la mochila puede descomponerse en subproblemas más pequeños, y se plantea una solución como combinación de las distintas soluciones óptimas de esos pequeños y muy útiles subproblemas.
2. Superposición de subproblemas: En este caso los subproblemas en los que se divide nuestro caso se superponen debido a que hay muchas combinaciones distintas e igualmente válidas para llenar la mochila. A través de este método se evita recalcular los resultados previamente almacenados.
3. Subproblemas independientes: Estos subproblemas de los cuales hemos hablado anteriormente en los que es posible dividir el problema podemos considerarlos independientes. Esto se debe a que cuando buscamos la solución de uno de estos subproblemas no depende de la solución de otros. Cada uno puede tratarse de manera aislada para su resolución.

4. Soluciones óptimas almacenadas: Para este tipo de problemas es necesario el almacenamiento de las diversas soluciones óptimas de los subproblemas. Para ello se suele utilizar en programación dinámica y también en este problema una tabla donde encontramos todas estas soluciones.
5. Ordenamiento de subproblemas: Aquí encontramos lo conocido como enfoque bottom-up. Básicamente decimos que se resuelven primero los subproblemas más pequeños, en este caso concreto, por ejemplo, buscar solución a mochilas con pequeñas capacidades, para posteriormente aplicarlas a los problemas más grandes y conseguir eficacia. De esta forma se llega antes a la solución completa que buscamos.

Vemos que por tanto el problema de la mochila cumple con los requisitos y se puede considerar un problema de programación dinámica.



2. Plantear el problema como una ecuación recurrente.

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

El caso general supone que para una capacidad de mochila j considerando echar (o no) los objetos desde el tipo 1 hasta el i , el máximo beneficio se conseguirá de una de las dos formas siguientes:

- No echar el objeto de tipo i y considerar echar sólo desde 1 hasta $i-1$: primera parte $T[i-1][j]$.
- Echar el objeto de tipo i , restando su peso de la capacidad de la mochila $j-w_i$ y aumentando el beneficio en b_i . Seguidamente, viendo si podemos echar (o no) objetos desde el tipo 1 hasta el tipo $i-1$: Segunda parte $b_i + T[i-1][j-w_i]$.

$T[i][j]$ es óptimo si las decisiones tomadas anteriormente para $T[i-1][j]$ y $T[i-1][j-w_i]$ lo son.

Para una capacidad j fija, $T[1][j]$ es óptimo ya que, tanto si $w_1 \leq j$ como si no, se selecciona llevar o no llevar el objeto con beneficio óptimo en la ecuación recurrente. Para $T[2][j]$ también es óptimo, ya que sólo se seleccionará llevar un objeto (o ninguno) si la suma de ambos pesos es superior a j , o ambos si no lo son, por la ecuación recurrente.

Se sigue por inducción hasta el caso $T[i-1][j]$ y suponemos éste óptimo. Por reducción al absurdo, si no lo fuera existiría otra decisión óptima distinta de $T[i-1][j]$ no considerada óptima, pero esto es imposible según la ecuación recurrente dada. Por tanto, $T[i-1][j]$ es óptimo. Demostrado óptimo $T[i-1][j]$, queda demostrado también para cualquier caso $T[i-1][j-p_i]$, como caso particular de $T[i-1][j]$.

Nos hemos dado cuenta de que analizando dicha ecuación recurrente lo que se va haciendo es comprobar el valor con el nuevo peso dentro o continuar con el valor anterior. Por tanto, esto resulta mucho más eficiente que utilizar el método de fuerza bruta.

3. Encontrar una representación de la ecuación para almacenar las soluciones parciales (en forma de tabla) y explicar cómo se rellena.

Para rellenar la tabla de soluciones parciales anteriormente mencionada existe un procedimiento el cual se puede seguir para conseguir nuestro objetivo.

El proceso que se sigue es el siguiente:

- Se crea una tabla con dimensiones $(n+1)(W+1)$, donde n es el número de elementos disponibles y W indica la capacidad máxima de la mochila, ambos datos referidos a ese subproblema.
- La primera columna de esta tabla se rellena toda con 0 (si $W=0$, no hay solución posible)
- Recorrer en forma de bucle desde $i=1$ hasta el valor de n , y desde cada i iniciar otro desde $j=1$ hasta W .
- De esta forma podemos conseguir comparar el peso del elemento i con la capacidad, en este caso j , de la mochila en cada una de las iteraciones.
- Si en ese momento, el valor que abarca i es menor o igual que la capacidad que abarca j , se procede a calcular el máximo que se puede obtener considerando este elemento.
Para ello se usa la fórmula siguiente:
 $\max(\text{valor}(i) + \text{tabla}(i-1)(j - \text{peso}(i)), \text{tabla}(i-1)(j))$.
- Si nos encontramos con el caso de que el peso del elemento i es mayor que la capacidad que abarca j , el máximo posible se toma el que hay en la fila anterior de la tabla, lo cual se traduce como: $\text{tabla}(i-1)(j)$.
- De esta forma se repite el procedimiento anterior para todos y cada uno de los elementos y distintas capacidades de la mochila.
- Cuando finalizan las ejecuciones de los bucles, el valor máximo posible se halla en la posición de la tabla de abajo a la derecha, osea: $\text{tabla}(n)(W)$.

Una vez tenemos la tabla llena al completo podemos rastrear hacia atrás para buscar aquellos datos que se deben seleccionar para la solución máxima.

Se verifica así si el valor de la celda en la que nos encontramos difiere del valor de la celda en la que nos encontrábamos antes en la misma fila. Si conseguimos esto, la solución cuenta con este valor.

Se pide también que estos valores de los que hablamos sean cifras enteras y en caso contrario usar algún criterio de aproximación o redondeo para que consiga serlo.

También contemplamos los casos extremos, en los cuales, por ejemplo el peso o las capacidades sean cifras muy muy grandes o muy muy pequeñas (gramos o toneladas por ejemplo). En este caso se procedería de la siguiente forma: Se aplica un factor de conversión y se pasa todo a una escala que no supere las 50 columnas para evitar así que nos pueda salir una tabla de dimensiones extremas con la cual no se pueda trabajar o sea demasiado complejo. Otra solución planteada es que también la mochila utilizase la media en un rango dado por la diferencia entre el mayor objeto y el menor objeto, para ver los diferentes tamaños utilizará este valor para las columnas.

	0	1	2	3	4	5	6	7	8	9	10	11
b=1 w=1	0	1	1	1	1	1	1	1	1	1	1	1
b=6 w=2	0	1	6	7	7	7	7	7	7	7	7	7
b=18 w=5	0	1	6	7	7	18	19	24	25	25	25	25
b=22 w=6	0	1	6	7	7	18	22	24	28	29	29	40
b=28 w=7	0	1	6	7	7	18	22	28	29	34	35	40

4. Comprobar el cumplimiento del P.O.B.

El principio de optimalidad de Bellman establece que si una solución global óptima a un problema puede ser construida mediante la combinación de soluciones óptimas a subproblemas más pequeños, entonces la solución óptima global también contiene soluciones óptimas a esos subproblemas más pequeños.

En el problema de la mochila 0/1, se nos proporciona un conjunto de elementos, cada uno con un valor y un peso asociados, y una capacidad máxima de peso para la mochila. El objetivo es seleccionar un subconjunto de elementos para incluir en la mochila de manera que se maximice el valor total, sin exceder la capacidad de peso.

Para demostrar el principio de optimalidad de Bellman, utilizaremos una tabla de programación dinámica para almacenar y calcular las soluciones óptimas a subproblemas más pequeños. Cada celda de la tabla representará la solución óptima para un subproblema específico.

Denotemos la capacidad de la mochila como " W " y el número total de elementos como " n ". Creamos una matriz de tamaño $(n+1) \times (W+1)$, donde $matriz[i][j]$ representa el valor máximo que se puede obtener utilizando los primeros " i " elementos y una capacidad de peso máxima de " j ".



Richard Bellman

El algoritmo para llenar la tabla y demostrar el principio de optimalidad de Bellman es el siguiente:

- Inicializar la primera columna de la tabla con ceros, ya que no se puede seleccionar ningún elemento cuando la capacidad de peso es cero.
- Para cada elemento "i" desde 1 hasta "n" y para cada capacidad de peso "j" desde 1 hasta "W", realiza los siguientes pasos:
 - a. Si el peso del elemento "i" es mayor que la capacidad de peso actual "j", entonces $\text{matriz}[i][j] = \text{matriz}[i-1][j]$, es decir, el valor óptimo para este subproblema es igual al valor óptimo para el subproblema anterior con el mismo peso máximo.
 - b. De lo contrario, calcular $\text{matriz}[i][j]$ como el máximo entre $\text{matriz}[i-1][j]$ y el valor del elemento "i" más $\text{matriz}[i-1][j-\text{peso del elemento "i"}]$. Esto significa que podemos considerar incluir el elemento "i" en la solución óptima o excluirlo, y tomamos el máximo de ambas opciones.
- El valor máximo que se puede obtener se encuentra en la celda $\text{matriz}[n][W]$.

Una vez que la tabla esté completa, podemos rastrear la solución óptima retrocediendo desde $\text{matriz}[n][W]$ hacia $\text{matriz}[1][1]$ utilizando las decisiones tomadas en cada subproblema. Si encontramos que la solución óptima global contiene soluciones óptimas a todos los subproblemas más pequeños, entonces se cumple el principio de optimalidad de Bellman para el problema de la mochila 0/1. Por ejemplo la siguiente tabla:

	0	1	2	3	4	5	6	7	8	9	10	11
b=1 w=1	0	1	1	1	1	1	1	1	1	1	1	1
b=6 w=2	0	1	6	7	7	7	7	7	7	7	7	7
b=18 w=5	0	1	6	7	7	18	19	24	25	25	25	25
b=22 w=6	0	1	6	7	7	18	22	24	28	29	29	40
b=28 w=7	0	1	6	7	7	18	22	28	29	34	35	40

5. Diseño del algoritmo de Programación Dinámica de acuerdo a lo establecido anteriormente.

Inicializar la tabla de programación dinámica con la función `RellenarTabla()`:

Crear una matriz de tamaño $(n+1) \times (\text{capacidad}+1)$, donde **n** es la longitud de los **valores** y **pesos**. La matriz almacenará los valores óptimos para cada subproblema.

Inicializar los casos base:

Rellenar la primera columna a 0, ya que no hay capacidad de peso disponible ni elementos disponibles.

Relleno de la tabla mediante un bucle:

Para cada elemento "i" desde 1 hasta "n" y para cada capacidad de peso "j" desde 1 hasta "W", realiza los siguientes pasos:

Si el peso del elemento "i" es mayor que la capacidad de peso actual "j", entonces $\text{matriz}[i][j] = \text{matriz}[i-1][j]$, es decir, el valor óptimo para este subproblema es igual al valor óptimo para el subproblema anterior con el mismo peso máximo.

De lo contrario, calcular $\text{matriz}[i][j]$ como el máximo entre $\text{matriz}[i-1][j]$ y el valor del elemento "i" más $\text{matriz}[i-1][j-\text{peso del elemento "i"}]$. Esto significa que podemos considerar incluir el elemento "i" en la solución óptima o excluirlo, y tomamos el máximo de ambas opciones.

El valor máximo que se puede obtener se encuentra en la celda $\text{matriz}[n][W]$.

Rastrear la solución óptima:

Una vez que la tabla esté completa, podemos rastrear la solución óptima retrocediendo desde $\text{matriz}[n][W]$ hacia $\text{matriz}[1][1]$ utilizando las decisiones tomadas en cada subproblema y quedándonos como solución aquellos elementos que mejor nos convenga.

6. Implementación del algoritmo de Programación Dinámica en C++.

Aportaremos a la memoria el respectivo código en un fichero.cpp el cual podrá compilar y ejecutar correctamente, además de las siguientes explicaciones:

```
int main(int argc, char *argv[]) {

    int *w, *v,**tabla, *sol, *candidatosl, tammochila, cantobj;
    int n, i, j, argumento, k=0;
    chrono::time_point<std::chrono::high_resolution_clock> t0, tf; // Para medir el tiempo de ejecución
    unsigned long tejecucion; // tiempo de ejecución del algoritmo en ms
    ifstream fentrada;

    if (argc <= 1) {
        cerr<<"\nError: El programa se debe ejecutar de la siguiente forma.\n\n";
        cerr<<argv[0]<<" NombreFicheroEntrada \n\n";
        return 0;
    }
}
```

ABRIMOS Y SACAMOS LOS VALORES DEL FICHERO

```
cout << "Abrimos fichero entrada " << endl;
// Abrimos fichero de entrada
fentrada.open( s: argv[1]);
if (!fentrada.is_open()) {
    cerr<<"Error: No se pudo abrir fichero para lectura "<<argv[1]<<"\n\n";
    return 0;
}
// Cogemos el tamaño de la mochila y la cantidad de objetos
int contador=0, valornum;
fentrada >> tammochila;
fentrada >> cantobj;

//Tamaño del vector con los datos y cada uno con un vector con sus conexiones
v = new int [cantobj];
w = new int [cantobj];
tabla = new int* [cantobj];
for (i = 0; i < cantobj ; i++) {
    tabla[i] = new int [tammochila+1];
}
}
```

RELLENAMOS LA TABLA

```
fentrada >> valornum;
while (valornum!= -2 ){
    while( valornum!= -1){
        //GUARDO NUMEROS DEL FICHERO A LOS VECTOR
        v[contador] = valornum;
        fentrada >> valornum;
        w[contador] = valornum;
        contador++;
        fentrada >> valornum;
    }
    fentrada >> valornum;
}

fentrada.close();
```

```
t0 = std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que comienza la ejecución del algoritmo

//Rellenar
tabla = Rellenartabla(v,w,cantobj,tammochila, tabla);
```

```

int** Rellenartabla( int* v,int* w, int cantobj, int tammochila,int ** tabla ){
    if(tammochila==0){
        //CASO BASE
        for(int i = 0; i < cantobj; i++){
            tabla[i][tammochila]=0;
        }
    }else{
        //CASO GENERAL
        int valorarrastre, valornuevo;
        tabla = Rellenartabla( v, w, cantobj, tammochila-1, tabla);
        for(int i = 0; i < cantobj; i++){
            //BUSCAMOS EL VALOR DE ARRASTRE
            if(i!=0){...}else{
                valorarrastre = 0;
            }
            //BUSCAMOS EL VALOR NUEVO
            if(i!=0 && tammochila-w[i] >= 0 ) {
                valornuevo = tabla[i - 1][tammochila-w[i]]+v[i];
            }else{
                valornuevo = 0;
            }
            if (i==0 && tammochila-w[i] >= 0 ) {
                valornuevo = v[i];
            }
        }
    }
}

```

```

        //COMPARAMOS
        if(valorarrastre >= valornuevo){
            tabla[i][tammochila] = valorarrastre;
        }else{
            tabla[i][tammochila] = valornuevo;
        }
    }
}

return tabla;
}

```

BUSCAMOS EL MAYOR VALOR EN LA ÚLTIMA COLUMNA
Y VAMOS MARCHA ATRÁS EN LA TABLA PARA BUSCAR LOS
OBJETOS QUE LO COMPONEN

```

//Buscar solución
cout << "Solución" << endl;
int max=tabla[0][tammochila], pos=0 ;

for (i = 1; i < cantobj ; i++) {
    if( max < tabla[i][tammochila] ) {
        max=tabla[i][tammochila];
        pos=i;
    }
}

cout << "LA SOLUCIÓN ES " << tabla[pos][tammochila] << endl;
cout << "CON LOS OBJETOS " << endl;
cout << "Peso " << w[pos] << " Beneficio " << v[pos] << endl;
int buscar=pos;

tammochila=w[pos];
while(buscar > 0 && tammochila > 0 ){

    while (tabla[pos][tammochila]==tabla[pos-1][tammochila]){
        pos--;
    }

    cout << "Peso " << w[pos] << " Beneficio " << v[pos] << endl;
    tammochila=w[pos];
    pos--;
}

tf= std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que finaliza la ejecución del algoritmo

tejecucion= std::chrono::duration_cast<std::chrono::microseconds>( dt tf - t0).count();

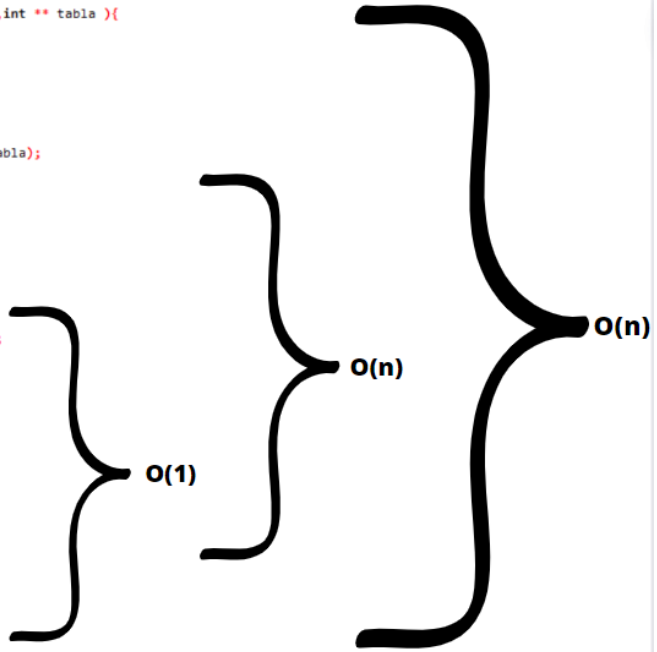
```

7. Cálculo de eficiencia del algoritmo (teórica y práctica).

Tomamos n como cantidad de objetos / elementos y m como el tamaño de la mochila.

EFICIENCIA TEÓRICA

```
int** Rellenartabla( int* v,int* w, int cantobj, int tammochila,int ** tabla ){
    if(tammochila==0){
        //CASO BASE
        for(int i = 0; i < cantobj; i++){
            tabla[i][tammochila]=0;
        }
    }else{
        //CASO GENERAL
        int valorarrastre, valornuevo;
        tabla = Rellenartabla( v, w, cantobj, tammochila-1, tabla);
        for(int i = 0; i < cantobj; i++){
            //BUSCAMOS EL VALOR DE ARRASTRE
            if(i==0) {
                valorarrastre = tabla[i - 1][tammochila];
            }else{
                valorarrastre = 0;
            }
            //BUSCAMOS EL VALOR NUEVO
            if(i==0 && tammochila-w[i] >= 0 ) {
                valornuevo = tabla[i - 1][tammochila-w[i]]+v[i];
            }else{
                valornuevo = 0;
            }
            if (i==0 && tammochila-w[i] >= 0 ) {
                valornuevo = v[i];
            }
            //COMPARAMOS
            if(valorarrastre >= valornuevo){
                tabla[i][tammochila] = valorarrastre;
            }else{
                tabla[i][tammochila] = valornuevo;
            }
        }
    }
    return tabla;
}
```



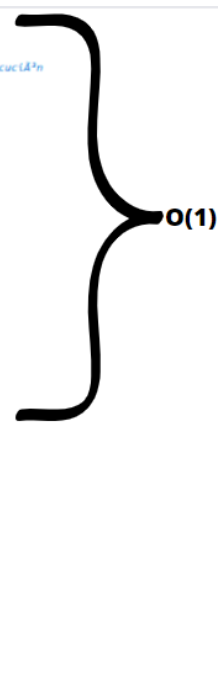
```
int main(int argc, char *argv[]) {
    int *n,*v,**tabla,*sol,*candidatosl, tammochila, cantobj;
    int n, i, j, argumento, k=0;
    chrono::time_point<std::chrono::high_resolution_clock> t0, t1; // Para medir el tiempo de ejecución
    unsigned long tejeucion; // tiempo de ejecución del algoritmo en ms
    ifstream fentrada;

    if (argc <= 1) {
        cerr<<"\nError: El programa se debe ejecutar de la siguiente forma.\n\n";
        cerr<<argv[0]<<" NombreFicheroEntrada \n\n";
        return 0;
    }

    cout << "Abrimos fichero entrada " << endl;
    // Abrimos fichero de entrada
    fentrada.open(argv[1]);
    if (!fentrada.is_open()) {
        cerr<<"Error: No se pudo abrir fichero para lectura "<<argv[1]<<"\n\n";
        return 0;
    }
    // Cogemos el tamaño de la mochila y la cantidad de objetos
    int contador=0, valornum;
    fentrada >> tammochila;
    fentrada >> cantobj;

    //Tamaño del vector con los datos y cada uno con un vector con sus conexiones
    v = new int [cantobj];
    w = new int [cantobj];
    tabla = new int* [cantobj];
    for (i = 0; i < cantobj; i++) {
        tabla[i] = new int [tammochila+1];
    }

    fentrada >> valornum;
    while (valornum != -2){
        while (valornum != -1){
            //GUARDAMOS NUMEROS DEL FICHERO A LOS VECTOR
            v[contador] = valornum;
            fentrada >> valornum;
            w[contador] = valornum;
            contador++;
            fentrada >> valornum;
        }
        fentrada >> valornum;
    }
    fentrada.close();
}
```



```

// BUSQUEDA DEL CAMINO
cout << "Entramos al código" << endl;

t0 = std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que comienza la ejecución del algoritmo

//Rellenar
tabla = Rellenartabla(v,w,cantobj,tamochila, tabla);

//Buscar solución
cout << "Solución" << endl;
int max=tabla[0][tamochila], pos=0;

for (i = 1; i < cantobj; i++) {
    if( max < tabla[i][tamochila] ) {
        max=tabla[i][tamochila];
        pos=i;
    }
}

cout << "LA SOLUCIÓN ES " << tabla[pos][tamochila] << endl;
cout << "CON LOS OBJETOS " << endl;
cout << "Peso " << w[pos] << " Beneficio " << v[pos] << endl;
int buscar=pos;

tamochila-=w[pos];
while(buscar > 0 && tamochila > 0){
    while (tabla[pos][tamochila]==tabla[pos-1][tamochila]){
        pos--;
    }
    cout << "Peso " << w[pos] << " Beneficio " << v[pos] << endl;
    tamochila-=w[pos];
    pos--;
}

tf = std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que finaliza la ejecución del algoritmo
tejecucion = std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();

cerr << "\nTiempo de ejec. (us): " << tejecucion << " para tam. caso " << n << endl;

// Guardamos tam. de caso y t_ejecucion a fichero de salida

delete [] v;
delete [] w;

// Cerramos fichero de salida

return 0;

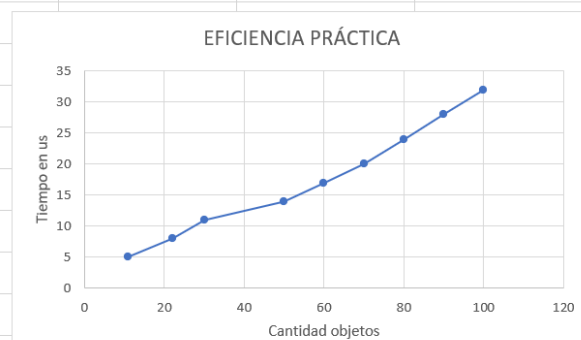
```

Complexity Analysis:

- $O(1)$ for initialization and output.
- $O(n)$ for filling the table.
- $O(1)$ for finding the maximum value in the table.
- $O(n)$ for the main loop.
- $O(1)$ for the while loop.
- $O(n*m)$ for the inner while loop.
- $O(1)$ for final output and cleanup.
- Total Complexity: $O(n*m)$**

EFICIENCIA PRÁCTICA

Cantidad Objetos	Tiempo
11	5
22	8
30	11
50	14
60	17
70	20
80	24
90	28
100	32



TAMAÑO MOCHILA = 11

En este caso, hemos analizado la eficiencia práctica con un tamaño de mochila constante (11) y hemos variado la cantidad de objetos que podemos introducir en una mochila. Al haber ejecutado el código con estos datos hemos obtenido una serie de tiempos y la gráfica superior acerca de los resultados de ejecución.

Observamos que la eficiencia teórica es $O(n \cdot m)$ donde n se interpreta como cantidad de objetos / elementos y m se interpreta como el tamaño de la mochila, mientras que la eficiencia práctica se asemeja más a una recta, esto se debe a que hemos tomado un tamaño de mochila fijo, en este caso de 11, por lo tanto la eficiencia práctica dependerá del número de objetos exclusivamente, por eso la representación se asemeja más a una recta.



8. Aplicación a dos instancias de problemas concretos, que se puedan leer desde un fichero de texto.

1) Probamos con los siguientes valores, 8 2(0,25), 9 10(1,111),4 5(1,25),11 17(1,54), 10 18(1,8), 1 2(2) , 5 10(2), 2 5(2,5), 3 8(2,666) ,7 20(2,86) , 6 18(3).

1. (8, 2) - (0.25)
2. (9, 10) - (1.111)
3. (4, 5) - (1.25)
4. (11, 17) - (1.54)
5. (10, 18) - (1.8)
6. (1, 2) - (2)
7. (5, 10) - (2)
8. (2, 5) - (2.5)
9. (3, 8) - (2.666)
10. (7, 20) - (2.86)
11. (6, 18) - (3)

2) Lo metemos en datos.txt con la siguiente estructura(Capacidad, nº de objetos, objetos (beneficio coste) :

11 11 2 8 -1 10 9 -1 5 4 -1 17 11 -1 18 10 -1 2 1 -1 10 5 -1 5 2 -1 8 3 -1 20 7 -1 18 6 -1 -2

3) El programa construye el caso base, la primera columna la pone a cero, y después va desde la columna 1 hasta la 11 utilizando la fórmula va comprobando los valores y saca la siguiente solución.

0	0	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	2	10	10	10
0	0	0	5	5	5	5	5	10	10	10
0	0	0	5	5	5	5	5	10	10	17
0	0	0	5	5	5	5	5	10	18	18
2	2	2	5	7	7	7	7	10	18	20
2	2	2	5	10	12	12	12	15	18	20
2	5	7	7	10	12	15	17	17	18	20
2	5	8	10	13	15	15	18	20	23	25
2	5	8	10	13	15	20	22	25	28	30
2	5	8	10	13	18	20	23	26	28	31

LA SOLUCIÓN ES 31
CON LOS OBJETOS
Peso 6 Beneficio 18
Peso 3 Beneficio 8
Peso 2 Beneficio 5

4) Probamos con los siguientes valores, 8 2 (0,25), 1 1 (1), 3 3 (1), 9 10 (1,111), 4 5 (1,25), 3 4 (1,333), 2 3 (1,5), 4 6 (1,5), 11 17 (1,54), 5 8 (1,6), 6 10 (1,66), 4 7 (1,75), 10 18 (1,8), 1 2 (2), 2 4 (2), 5 10 (2), 7 14 (2), 5 11 (2,2), 8 16(2), 2 5 (2,5), 7 20 (2,86) , 6 18 (3).

1. (8, 2) - (0.25)	12. (4, 7) - (1.75)
2. (1, 1) - (1)	13. (10, 18) - (1.8)
3. (3, 3) - (1)	14. (1, 2) - (2)
4. (9, 10) - (1.111)	15. (2, 4) - (2)
5. (4, 5) - (1.25)	16. (5, 10) - (2)
6. (3, 4) - (1.333)	17. (7, 14) - (2)
7. (2, 3) - (1.5)	18. (5, 11) - (2.2)
8. (4, 6) - (1.5)	19. (8, 16) - (2)
9. (11, 17) - (1.54)	20. (2, 5) - (2.5)
10. (5, 8) - (1.6)	21. (7, 20) - (2.86)
11. (6, 10) - (1.66)	22. (6, 18) - (3)

5) Lo metemos en datos.txt con la siguiente estructura(Capacidad, nº de objetos, objetos (beneficio coste):

11 22 2 8 -1 1 1 -1 3 3 -1 10 9 -1 5 4 -1 4 3 -1 3 2 -1 6 4 -1 17 11
-1 8 5 -1 10 6 -1 7 4 -1 18 10 -1 2 1 -1 4 2 -1 10 5 -1 14 7 -1 11 5
-1 16 8 -1 5 2 -1 20 7 -1 18 6 -1 -2

6) El programa construye el caso base, la primera columna la pone a cero, y después va desde la columna 1 hasta la 11 utilizando la fórmula va comprobando los valores y saca la siguiente solución.

8 2 0	0	0	0	0	0	0	0	2	2	2	2
1 1 0	1	1	1	1	1	1	1	2	3	3	3
3 3 0	1	1	3	4	4	4	4	4	4	4	5
9 10 0	1	1	3	4	4	4	4	4	10	11	11
4 5 0	1	1	3	5	6	6	8	9	10	11	11
3 4 0	1	1	4	5	6	7	9	10	10	12	13
2 3 0	1	3	4	5	7	8	9	10	12	13	13
4 6 0	1	3	4	6	7	9	10	11	13	14	15
11 17 0	1	3	4	6	7	9	10	11	13	14	17
5 8 0	1	3	4	6	8	9	11	12	14	15	17
6 10 0	1	3	4	6	8	10	11	13	14	16	18
4 7 0	1	3	4	7	8	10	11	13	15	17	18
10 18 0	1	3	4	7	8	10	11	13	15	18	19
1 2 0	2	3	5	7	9	10	12	13	15	18	20
2 4 0	2	4	6	7	9	11	13	14	16	18	20
5 10 0	2	4	6	7	10	12	14	16	17	19	21
7 14 0	2	4	6	7	10	12	14	16	18	20	21
5 11 0	2	4	6	7	11	13	15	17	18	21	23
8 16 0	2	4	6	7	11	13	15	17	18	21	23
2 5 0	2	5	7	9	11	13	16	18	20	22	23
7 20 0	2	5	7	9	11	13	20	22	25	27	29
6 18 0	2	5	7	9	11	18	20	23	25	27	29

LA SOLUCIÓN ES 29
CON LOS OBJETOS
Peso 7 Beneficio 20
Peso 2 Beneficio 5
Peso 2 Beneficio 4