DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Informe de Trabajo Práctico 1

Subset Sum Problem

Domingo 16 de Septiembre de 2018 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Springhart, Gonzalo	308/17	glspringhart@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Introducción

En este informe vamos a comparar a eficiencia de distintos algoritmos utilizados para resolver un problema conocido como *Subset Sum Problem* (o Problema de suma de subconjuntos). El mismo consiste en lo siguiente, dado un conjunto S de n elementos, cada uno con un valor asociado v_i y un valor objetivo V, se quiere saber si existe un subconjunto de ítems de S que sumen exactamente el valor objetivo, y si existe dicho subconjunto, se quiere saber cuál es la mínima cardinalidad entre todos los subconjuntos posibles, en otras palabras, hay que decidir si existe $R \subseteq S$ tal que $\sum_{i \in R} vi = V$. Se asumen también que los valores de S son enteros no negativos (aunque el problema se puede resolver también sin necesidad de esta restricción).
El objetivo es ver cuál de los algoritmos es más eficiente al resolver el problema, se van a presentar 4 algoritmos que resuelven el problema, indicando como funcionan, justificando sus complejidades y comprobando a travéz de experimentos que estas complejidades son ciertas.

1. Algoritmos y justificación de complejidades

Se va a usar la siguiente notación:

- S es el conjunto, que tiene n elementos, cada uno con un valor asociado v_i con $i \in \{1, ..., n\}$
- V es el valor objetivo

1.1. Fuerza Bruta

El primer algoritmo presentado para resolver el problema es uno de *Fuerza Bruta*, básicamente el algoritmo genera todos los conjuntos posibles con los elementos de S y se fija cuál de ellos tiene elementos tales que su suma de exactamente V, mientras los calcula se va quedando con el que tiene menor cardinalidad.
El algoritmo implementado en este trabajo práctico es el siguiente:

```
procedure subsetSumFuerzaBruta(S, V)
    longMinima ← −1
    for contador de 0 a 2^|S| − 1 do                    ▷ Se ejecuta 2^|S| veces
        longActual ← 0
        sumaTotal ← 0
        for i de 0 a |S| − 1 do                         ▷ Se ejecuta |S| veces
            if El i-esimo bit de contador es 1 then
                sumaTotal ← sumaTotal + S[i]
                longActual ← longActual + 1
            end if
        end for
        if sumaTotal == V then                          ▷ Toda esta parte es O(1)
            if longMinima == −1 then
                longMinima ← longActual
            else
                longMinima ← min(longMinima, longActual)
            end if
        end if
    end for
     return longMinima
end procedure
```

Este algoritmo genera todos los subconjuntos del conjunto partes de S de la siguiente forma, podemos interpretar cada subconjunto de S como un conjunto donde algunos elementos de S están y otros no. Entonces si pensamos a cada elemento como un bit donde 0 indica que el elemento no esta y 1 que si, podemos ver que con $|S|$ bits nos alcanza para generar todos los subconjuntos del conjunto partes de S.
El algoritmo ejecuta un ciclo que incrementa un contador (que usamos para saber en que subconjunto está) $2^{|S|}$ veces, dentro de cada ciclo se fija el valor de cada bit de contador (que tiene $|S|$ bits) y suma los elementos de S que correspondan a la posición del bit que valga 1, entonces como mucho se suman $|S|$ elementos, las operaciones para calcular el mínimo después de hacer la suma son $O(1)$.
Entonces la complejidad del algoritmo es $O(|S| * 2^|S|) = O(n * 2^n)$.

1.2. Backtracking

El Backtracking es un algoritmo que se utiliza para encontrar todas o algunas de las soluciones de algún problema, se basa en ir armando la solución correcta al problema desechando las que no pueden ser correctas a medida que se ejecuta, lo que se conoce como poda, existen podas de **factibilidad**, que son las que se realizan cuando se sabe que la rama de soluciones no puede llegar al resultado y podas de **optimalidad** que se realizan para optimizar la solución en ciertos casos. Esa característica de desechar soluciones que no pueden ser correctas lo hace superior a probar todas las posibles.
El algoritmo de backtracking implementado en este trabajo es el siguiente:

```
procedure subsetSumBacktracking(S, V)
    ordenar(S)                                   ▷ Uso un ordenamiento que sea O(n * log(n))
    return subsetSumBacktrackingAUX(S, V, 0, 0, 0)
end procedure
```

```
procedure subsetSumBacktrackingAUX(S, V, inicio, longActual, sumActual)
    if inicio == |S| then
        if sumActual == V then
            return longActual
        else
            return −1
        end if
    else
        if sumActual == V then
            return minPos(longActual, subsetSumBacktrackingAUX(S, V, inicio, longActual − 1, sumActual − S[inicio − 1]))
        else if sumActual > V then
            return −1
        else if sumActual == sumActual + S[inicio] then
            return subsetSumBacktrackingAUX(S, V, inicio + 1, longActual, sumActual)
        else
            return minPos(subsetSumBacktrackingAUX(S, V, inicio + 1, longActual + 1, sumActual + S[inicio]), subsetSumBacktrackingAUX(S, V, inicio + 1, longActual, sumActual))
        end if
    end if
end procedure
```

El algoritmo implementado de forma recursiva aque resuelve el problema funciona de la siguiente forma, cada elemento de S puede estar o no en la solución, entonces en cada recursión pido el minimo entre tomar al elemento como parte de la solucion y no tomarlo. El caso base es cuando ya no quedan elementos para tomar, en ese caso se ve si la suma da o no y se devuelve la longitud correspondiente (-1 si no da y la longitud de la solucion si da). Si se alcanza al valor objetivo después de agregar a un elemento, compara la longitud de esa solución con la que no incluyo ese elemento haciendo recursión y se queda con la menor. $minPos(a, b)$ devuelve el menor entre a y b, si uno de los dos es negativo, devuelve el otro y si los dos son negativos devuelve a uno de los dos, de esa forma si no hay solución se puede devolver -1.
Este algoritmo supone que S esta ordenado de forma creciente, esto es importante para poder justificar una de las podas hechas.

La **poda de factibilidad** del algoritmo consiste en que, si la suma al agregar un elemento se pasa de V, entonces esa rama de solución nunca va a encontrar un subconjunto que lo sume, ya que al estar ordenado los elementos que siguen son mayores. La **poda de optimalidad** se realiza cuando el elemento que se va a agregar conjunto de solucion es el 0, como un 0 no aporta nada a la suma, se saltea y se ve directamente el elemento siguiente.

Para calcular la complejidad hay que ver como se van abriendo las llamadas recursivas, si ignoramos las podas (que no afectan este cálculo) podemos ver que en cada recursión se van realizando dos llamadas a la función, una si se incluye al elemento actual en la solución y una si no. De esa forma la cantidad de llamadas va aumentando de forma exponencial a medida que sigue la función, hasta que se terminen los elementos del conjunto. Si graficamos como se va ejecutando la función se va a formar un árbol completo (aunque varias hojas tendrían lo mismo). Por lo que la función se ejecuta $2^{|S|} = 2^n$ veces. Por lo tanto la complejidad de este algoritmo es $O(|S| * log(|S|) + 2^|S|) = O(n * log(n) + 2^n) \in O(n * 2^n)$.

1.3. Programación Dinámica Top Down

La programación dinámica es un método de programación que se basa en resolver un problema dividiéndolo en subproblemas de forma recursiva y guardando resultados ya calculados para no volverlos a calcular. Para evitar resolver los

mismos subproblemas existen dos estilos de programación dinámica, en esta sección se muestra un algoritmo en el estilo **Top Down**. Este tipo de programación dinámica se parece mucho a como uno lo resolveria con recursión, se va partiendo el problema principal en subproblemas más pequeños, con la diferencia de que de realiza un proceso de **memoización**, o en otras palabras, se guardan los resultados en una tabla para buscarlos en lugar de calcularlos de nuevo.
Para este algoritmo la tabla va a representar lo siguiente, las columas de la tabla representan un posible valor objetivo, mientras que las filas representan la cantidad de elementos usados de S (empezando desde el primero), y en cada celda se guarda el cardinal del conjunto más chico que forma el valor objetivo que corresponde a la columna, así por ejemplo la posición [3][6] tiene guardado el cardinal de los subconjuntos más chico que usa los primeros 3 elementos de S y suma 6, en caso de no haber subconjunto que sume el valor de la columna, en la celda habrá un -1.
Entonces el algoritmo queda asi:

---

procedure SUBSETSUMPDTDREC(S, i, j, $matriz$) ▷ $matriz$ se pasa por referencia
  if $i < 0$ **then**
    return -1
  end if
  if $j < 0$ **then**
    return -1
  end if
  if $j == 0$ **then**
    if $matriz[i][j] != -10$ **then**
      return $matriz[i][j]$
    else
      $matriz[i][j] \leftarrow 0$
      return $matriz[i][j]$
    end if
  end if
  if $i == 0$ **then**
    if $matriz[i][j] != -10$ **then**
      return $matriz[i][j]$
    else
      if $j == 0$ **then**
        $matriz[i][j] \leftarrow 0$
      else
        if $S[i] == j$ **then**
          $matriz[i][j] \leftarrow 1$
        else
          $matriz[i][j] \leftarrow -1$
        end if
      end if
      return $matriz[i][j]$
    end if
  end if
  if $matriz[i][j] != -10$ **then**
    return $matriz[i][j]$
  else
    $m1 \leftarrow subsetSumPDTDRec(S, i-1, j, matriz)$ ▷ Calculo el problema sin contar el elemento actual
    $m2 \leftarrow subsetSumPDTDRec(S, i-1, j-s[i], matriz)$ ▷ Calculo el problema teniéndolo en cuenta
    if $m1 == -1$ and $m2 == -1$ **then**
      $matriz[i][j] \leftarrow -1$
    else
      if $minPos(m1, m2) == m1$ **then**
        $matriz[i][j] \leftarrow m1$
      else
        $matriz[i][j] \leftarrow m2 + 1$ ▷ El mínimo vino de incluir el elemento, entonces la longitud es uno más de lo que devolvió el subproblema
      end if
    end if
    return $matriz[i][j]$
  end if
end procedure

```
procedure subsetSumPDTD(S, V)
    matriz ← matriz de |S| filas y V + 1 columnas con −10 en cada valor      ▷ Esto es O(|S| ∗ V) = O(n ∗ V)
    subsetSumPDTDRec(S, |S| − 1, V, matriz)                                  ▷ La matriz se pasa por referencia
    return matriz[|S| − 1][V]
end procedure
```

Este algoritmo llega a la solución usando la misma idea del de que el de backtracking, tomando el minimo entre la solución que incluye el elemento actual y el que no, así se van partiendo los subproblemas hasta llegar a los casos base, que son cuando se pasen las dimensiones de la matriz que resulta en un -1 y cuando se llega a la columna 0 donde se devuelve 0. También se va ejecutando de forma parecida al de backtracking, se van haciendo dos llamados a funciones en cada recursión, pero estas recursiones sólo se realizan una vez, si en la matriz hay un -10 entonces ese valor no fue computado y se calcula. Pero una vez calculado, se guarda y se se pide entonces se accede directamente al guardado. También la recursión se realiza sobre el tamaño de la matriz, por lo que esta acotada por $O(|S| * V) = O(n * V)$.

1.4. Programación Dinámica Bottom Up

El otro estilo de Programación Dinámica se conoce como **Bottom Up**, en lugar de resolver el problema partiendolo en subproblemas, va resolviendo los subproblemas primero y llega a la solución del problema principal.
En lugar de utilizar la recursión directamente, deducimos de ésta una formula que nos permita ir llenando la misma matriz que se uso en el algoritmo Top Down de forma directa. Y al final de devuelve la celda correspondiente de la matriz.
El algoritmo es el siguiente:

```
procedure subsetSumPDBU(S, V)
    matriz ← crear matriz de |S| filas y V + 1 columnas
    for i de 0 a |S| − 1 do
        matriz[i][0] ← 0
    end for
    for j de 0 a V do
        if S[0] == j then
            matriz[0][j] ← 1
        else
            matriz[0][j] ← −1
        end if
    end for
    for i de 1 a |S| − 1 do
        for j de 1 a V do
            m1 ← matriz[i − 1][j]
            if j − S[i] < 0 then
                m2 ← −1
            else
                m2 ← matriz[i − 1][j − S[i]]
            end if
            if minPos(m1, m2) == m1 then
                matriz[i][j] ← m1                                  ▷ S[i] no es parte de la solución
            else
                matriz[i][j] ← m2 + 1                              ▷ S[i] es parte de la solución, hay que sumar 1
            end if
        end for
    end for
    return matriz[|S| − 1][V]
end procedure
```

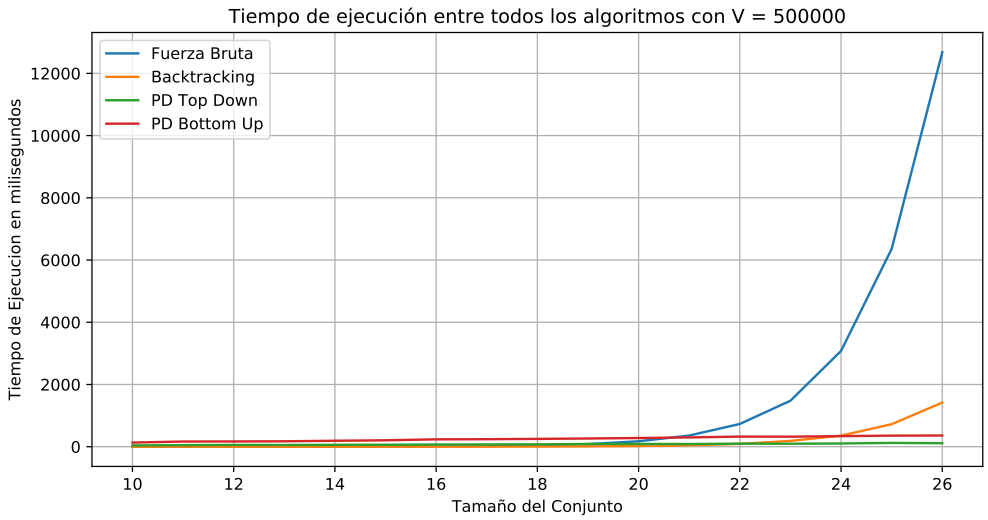
En este algoritmo la matriz se interpreta igual que en el anterior, las columnas son posibles valores objetivos y las filas son la cantidad de elementos (tomados desde el primero) del conjunto, el valor de toda la columna 0 va a ser 0, ya que dado cualquier conjunto S, el conjunto vacío es un subconjunto de S y la suma de sus elementos es 0. En la primera fila todos las columnas van a tener -1 cuando el número de la columna no sea igual al primer elemento de S y 1 cuando sea. Basándonos en la recursión podemos ver que tomando un elemento, la menor cardinal de un subconjunto que suma V es la mínima cardinal entre el conjunto que suma V y no incluya al elemento actual y la cardinal del conjunto que suma V e incluye el elemento actual de S. Entonces para una celda $(i,\ j)$ la cardinal va a ser el mínimo entre $(i-1,\ j)$ y $(i-1,\ j-S[i])$, si ambos son negativos entonces el cardinal es -1. Si el cardinal mínimo es el de $(i-1,\ j-S[i])$, entonces hay que sumarle 1 para que tome en cuenta la inclusión del elemento $S[i]$ en la solución. Entonces la solución al problema se encontraría en la celda $(|S|-1,\ V)$.

La complejidad de este algoritmo es simple de calcular, podemos ignorar la complejidad de los dos primeros ciclos ya que la que realmente pesa es la del tercero, que se ejecuta $|S| - 1 * V$ veces, lo que significa que es $O(|S| * V)$.
Entonces el algoritmo es $O(|S| * V) = O(n * V)$.

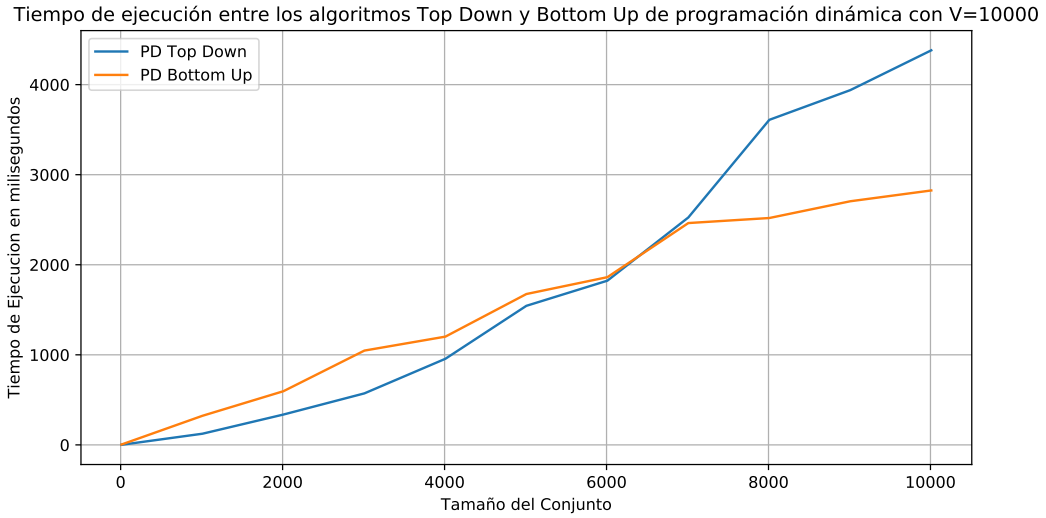
2. Experimentos

Como fue mencionado anteriormente nos interesa saber cuál de los algoritmos es más eficiente para resolver el *Subset Sum Problem*, por los cálculos de complejidad anteriormente realizados podemos intuir que los algoritmos de programación dinámica son los que van a tener ventaja sobre la fuerza bruta y el backtracking.
Para poder apreciar las diferencias entre ellos, se diseño un caso de test que prueba el tiempo que tardan los algoritmos en su peor caso, para la Fuerza Bruta no existe realmente un peor caso ya que siempre se fija en todos los subconjuntos posibles, al algoritmo Bottom Up le sucede algo parecido, como siempre genera toda la tabla, no hay un valor de entrada que le haga hacer más o menos pasos, sin embargo es afectado junto con el Top Down por el valor de V, entonces el peor caso se basó en el de Backtracking y el Top Down.
El peor caso de Backtracking consiste en evitar las podas que lo optimizan, por ende las entradas que van a dar el peor caso son: un conjunto tal que ningun elemento sea 0 y un valor objetivo tal que sea mayor que la suma de todos los elementos del conjunto. De esa forma se evitan las podas y el algoritmo tiene que calcular todas las ramas.
El peor caso del algoritmo Top Down seria un conjunto cuyos elementos sean todos 1, ya que provocaría que la recursión tome más pasos para llegar a los casos base, y por ende tome más tiempo.

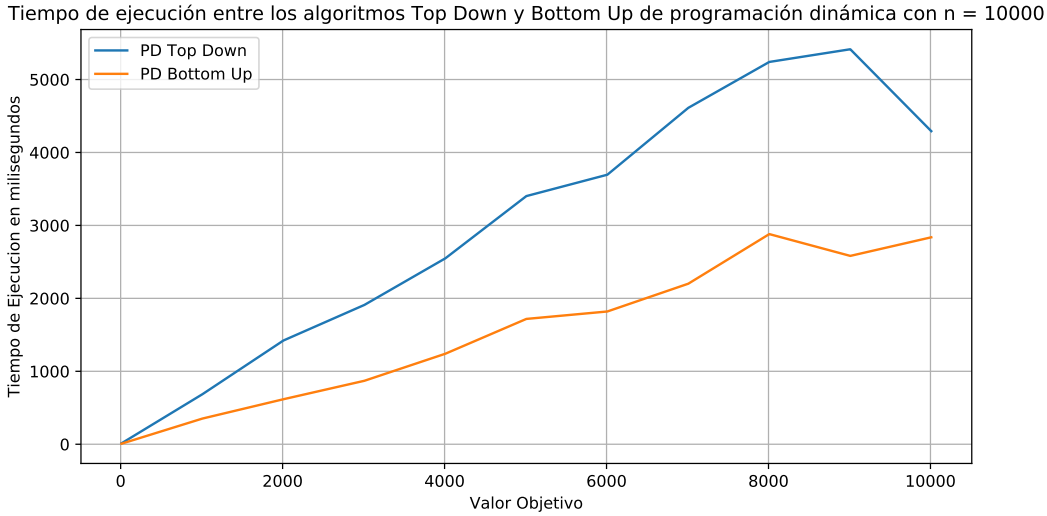
Utilizando un script hecho en python se fueron creando inputs que cumplen esta característica, que después procesó una versión modificada del programa del TP para generar outputs, que se leen desde jupyter notebooks para poder generar los gráficos.
Despúes de realizar varios experimentos surgió un problema, aunque se construyo un peor caso para podes comparar a los algoritmos, los de programación dinámica resultaron ser mucho más eficientes que lo esperado. En el siguiente gráfico se aumento el tamaño del conjunto dejando a V fija en un número arbitrariamente alto:



Si bien este gráfico no nos dice mucho de como son los dos algoritmos de programación dinámica, nos ayuda a ver que aún tomando datos de V grandes son mejores que la Fuerza Bruta y Backtracking. El tamaño del conjunto no llega a crecer mucho ya que como la Fuerza Bruta pasa por todos los subconjuntos de partes de S la cantidad de pasos que tiene que realizar crece demasiado rápido, al punto de que no alcanza la memoria del sistema para poder calcularlos, esto no pasa de la misma forma con los otros algoritmos.
Ya que los algoritmos Top Down y Bottom Up resultaron mejores, nos interesaría saber cuál de ellos es mejor, para eso se hicieron dos experimentos, uno donde se altera la cantidad de elementos del conjunto y otro donde se altera el valor objetivo del problema.
El primer experimento produjo el suguiente gráfico:



Y el segundo produjo este:



De estos gráficos podemos ver que a la larga el algoritmo Bottom Up es el que termina resolviendo el problema en menos tiempo, pero lo que es más interesante es que los dos algoritmos tienden a tardar más a media que aumenta el valór V que si aumenta la cantidad de elementos en el conjunto. En el caso del algoritmo Top Down, en el primér gráfico se ve que llega a tardar como mucho unos 4,5 segundos mientras que en el segundo se ve que se pasa de los 5 segundos en una ocasión. Esto puede estar sucediendo porque el algoritmo Top Down depende del valor V para ir llegando a los casos base, mientras que el Bottom Up llena su matriz siempre al mismo paso, aunque esto le saque ventaja cuando V no es muy grande.

Luego de la experimentación podemos concluir que el algoritmo más óptimo para resolver el problema de Subset Sum en general es el Bottom Up, aunque los otros métodos podrían ser utilizados si se tiene conocimiento sobre cuales van a ser los datos de entrada, por ejemplo si se supiera que el conjunto va a tener muchos 0 el algoritmo de Backtracking podría ser utilizado y aún se obtendrian resultados en tiempos rápidos.

3. Código

```
#include <iostream>
#include <vector>
#include <math.h>
#include <algorithm>
#include <fstream>
#include <chrono>
#include <cstring>

//Todo el código se encuentra en el mismo archivo
//para facilitar lectura/localizacion del código

//Código de Input/Output (IO)
std::pair<int, std::vector<int> > handleInput(char* ar){
```

```
14      int cantElem, valorObjetivo;
15      std::vector<int> conjunto;
16      std::ifstream archivo(ar);
17
18      archivo >> cantElem;
19      archivo >> valorObjetivo;
20
21      for(int i = 0; i < cantElem; i++){
22          int elem;
23          archivo >> elem;
24          conjunto.push_back(elem);
25      }
26
27      archivo.close();
28
29      return std::make_pair(valorObjetivo, conjunto);
30  }
31
32  //Fin código IO
33
34  //Código de Fuerza Bruta (FB)
35
36  //Resuelvo el problema pasando por todos los conjuntos del conjunto partes
37  int subsetSumFuerzaBruta(std::vector<int> conjuntoInicial, int valorObjetivo){//O(n*(2^n))
38      //std::vector<std::vector<int> > conjuntoPartes;
39      unsigned long contador;
40      unsigned long tamFinal = pow(2, conjuntoInicial.size());
41
42      //bool existe = false;
43      int longMin = -1;
44
45      //Acó pasamos por todos los conjuntos del conjunto partes
46      for(contador = 0; contador <= tamFinal; contador++){//O(2^n)
47          int sumaTotal = 0;
48          int longActual = 0;
49
50          //Me fijo el valor binario de contador, donde los 1 equivalen
51          //a incluir el elemento en el conjunto solucion
52          for(int elem = 0; elem < conjuntoInicial.size(); elem++){//O(n)
53              if(contador & (1 << elem)){
54                  //El valor va en la solucion
55                  sumaTotal += conjuntoInicial[elem];
56                  longActual++;
57              }
58          }
59
60          //Ya tengo la suma de un intento de solucion
61          if(sumaTotal == valorObjetivo){
62              if(longMin < 0){
63                  longMin = longActual;
64              }
65              else if(longMin > longActual){
66                  longMin = longActual;
67              }
68          }
69
70      }
71
72      //Ahora longMin es la cardinal más chica los conjuntos que suman valorObjetivo
73      return longMin;
74  }
75
76  //Fin código FB
77
78  ////////////////////////////////////////////////////////////////////////////////
79
80  //Código de Backtracking (BT)
81
82  //Devuelve el menor entre a y b, pero si uno es negativo devuelve el otro
83  int minimoPositivo(int a, int b){ //O(1)
84      if(a < 0)
85          return b;
86      if(b < 0)
87          return a;
88
89      return (a < b)? a : b;
90  }
91
92  //Función recursiva que resuelve el backtracking
93  int subsetSumBTRec(std::vector<int>& conjuntoInicial, int valorObjetivo, int inicio, int longActual, int
        sumActual){
94      if(inicio == conjuntoInicial.size()){
95          if(sumActual == valorObjetivo){
96              return longActual;
97          }
98          else{
99              return -1;
100         }
101     }
102     else{
103         if(sumActual == valorObjetivo){
104             return minimoPositivo(longActual, subsetSumBTRec(conjuntoInicial, valorObjetivo, inicio,
```

```
                longActual - 1, sumActual - conjuntoInicial[inicio - 1]));
105         }
106         else if(sumActual > valorObjetivo){
107             //Corte De Factibilidad, como esta ordenado el conjunto seguir por esta rama nunca puede dar
                    la solucion
108             return -1;
109         }
110         else if(sumActual == sumActual + conjuntoInicial[inicio]){
111             //El elemento que sigue es un cero, conviene no incluirlo en el conjunto asi no aumenta la
                    cardinalidad
112             return subsetSumBTRec(conjuntoInicial, valorObjetivo, inicio + 1, longActual, sumActual);
113         }
114         else{
115             return minimoPositivo(subsetSumBTRec(conjuntoInicial, valorObjetivo, inicio + 1, longActual +
                     1, sumActual + conjuntoInicial[inicio]), subsetSumBTRec(conjuntoInicial, valorObjetivo,
                    inicio + 1, longActual, sumActual));
116         }
117     }
118 }
119
120 int subsetSumBacktracking(std::vector<int>& conjuntoInicial, int valorObjetivo){//O(2^n)
121     //Ordeno el conjunto
122     std::sort(conjuntoInicial.begin(), conjuntoInicial.end()); //n*log(n)
123
124     return subsetSumBTRec(conjuntoInicial, valorObjetivo, 0, 0, 0); //O(2^n)
125 }
126
127 //Fin código BT
128
129 ////////////////////////////////////////////////////////////////////////////////
130
131 //Código de Programación Dinámica (PD)
132
133 //Version Top Down
134
135 int subsetSumPDTDRec(std::vector<int>& conjuntoInicial, int i, int j, std::vector<std::vector<int> > &
        matriz){
136     //Casos Base
137
138     if(i < 0){
139         return -1;
140     }
141
142     if(j < 0){
143         return -1;
144     }
145
146     if(j == 0){
147         if(matriz[i][j] != -10){
148             return matriz[i][j];
149         }
150         else{
151             matriz[i][j] = 0;
152             return matriz[i][j];
153         }
154     }
155
156     if(i == 0){
157         if(matriz[i][j] != -10){
158             return matriz[i][j];
159         }
160         else{
161             if(j == 0){
162                 matriz[i][j] = 0;
163             }
164             else{
165                 matriz[i][j] = (conjuntoInicial[i] == j) ? 1 : -1;
166             }
167             return matriz[i][j];
168         }
169     }
170
171     //Inducción
172     if(matriz[i][j] != -10){
173         return matriz[i][j];
174     }
175     else{
176         int m1 = subsetSumPDTDRec(conjuntoInicial, i-1, j, matriz);
177         int m2 = subsetSumPDTDRec(conjuntoInicial, i-1, j - conjuntoInicial[i], matriz);
178
179         if(m1 == -1 && m2 == -1){
180             matriz[i][j] = -1;
181         }
182         else{
183             if(minimoPositivo(m1, m2) == m1){
184                 matriz[i][j] = m1;
185             }
186             else{
187                 matriz[i][j] = m2 + 1;
188             }
189         }
```

```
190
191         return matriz[i][j];
192     }
193 }
194
195 int subsetSumPDTD(std::vector<int> &conjuntoInicial, int valorObjetivo){
196     std::vector<std::vector<int> > matriz (conjuntoInicial.size(), std::vector<int>(valorObjetivo + 1));
197
198     for(int i = 0; i < conjuntoInicial.size(); i++){
199         for(int j = 0; j < valorObjetivo + 1; j++){
200             matriz[i][j] = -10; //Valor arbitrario que indica que no hay nada guardado en [i][j]
201         }
202     }
203
204     return subsetSumPDTDRec(conjuntoInicial, conjuntoInicial.size() - 1, valorObjetivo, matriz);
205 }
206
207 //Version Bottom Up
208
209 int subsetSumPDBU(std::vector<int>& conjuntoInicial, int valorObjetivo){
210     std::vector<std::vector<int> > matriz (conjuntoInicial.size(), std::vector<int>(valorObjetivo + 1));
211
212     for(int i = 0; i < conjuntoInicial.size(); i++){
213         matriz[i][0] = 0;
214     }
215
216     for(int j = 1; j < valorObjetivo + 1; j++){
217         matriz[0][j] = (conjuntoInicial[0] == j)? 1 : -1;
218     }
219
220     for(int i = 1 ; i < conjuntoInicial.size(); i++){
221         for(int j = 1; j < valorObjetivo + 1; j++){
222             int m1 = matriz[i - 1][j];
223             int m2 = ((j - conjuntoInicial[i]) < 0)? -1 : matriz[i - 1][j - conjuntoInicial[i]];
224
225             if(minimoPositivo(m1, m2) == m1){
226                 matriz[i][j] = m1;
227             }
228             else{
229                 matriz[i][j] = m2 + 1;
230             }
231         }
232     }
233
234     return matriz[conjuntoInicial.size() - 1][valorObjetivo];
235 }
236
237 //Fin código PD
238
239 int main(int argc, char** argv)
240 {
241     std::cout << "------------------------AED3TP1-------------------------" << std::endl;
242     std::pair<int, std::vector<int> > entrada;
243     if(strcmp(argv[1], "") == 0){
244         std::cout << "Error al leer archivo!" << std::endl;
245         return 0;
246     }
247
248     entrada = handleInput(argv[1]);
249
250     std::cout << "Valor Objetivo: " << entrada.first << std::endl;
251     std::cout << "Tam. Conjunto: " << entrada.second.size() << std::endl;
252
253     std::cout << "-----------------------------------------------------" << std::endl;
254
255     int resultadoFB = -1, resultadoBT = -1, resultadoPDTD = -1, resultadoPDBU = -1;
256
257     std::cout << "Resolviendo con Fuerza Bruta" << std::endl;
258
259     auto ahora = std::chrono::_V2::steady_clock::now();
260     resultadoFB = subsetSumFuerzaBruta(entrada.second, entrada.first);
261     auto fin = std::chrono::_V2::steady_clock::now();
262
263     std::cout << "Tam. de conjunto minimo que suma " << entrada.first << ": " << resultadoFB << std::endl
        ;
264     auto duracion1 = std::chrono::duration_cast<std::chrono::milliseconds>(fin - ahora);
265     std::cout << "El Algoritmo de Fuerza Bruta tardo: " << duracion1.count() << " milisegundos" << std::
        endl;
266
267     std::cout << "-----------------------------------------------------" << std::endl;
268     std::cout << "Resolviendo con Backtracking" << std::endl;
269
270     ahora = std::chrono::_V2::steady_clock::now();
271     resultadoBT = subsetSumBacktracking(entrada.second, entrada.first);
272     fin = std::chrono::_V2::steady_clock::now();
273
274     std::cout << "Tam. de conjunto minimo que suma " << entrada.first << ": " << resultadoBT << std::endl
        ;
275     auto duracion2 = std::chrono::duration_cast<std::chrono::milliseconds>(fin - ahora);
276     std::cout << "El Algoritmo de Backtracking tardo: " << duracion2.count() << " milisegundos" << std::
        endl;
277
278     std::cout << "-----------------------------------------------------" << std::endl;
```

```
279     std::cout << "Resolviendo con Programancion Dinamica (Algoritmo Top Down)" << std::endl;
280
281     ahora = std::chrono::_V2::steady_clock::now();
282     resultadoPDTD = subsetSumPDTD(entrada.second, entrada.first);
283     fin = std::chrono::_V2::steady_clock::now();
284
285     std::cout << "Tam. de conjunto minimo que suma " << entrada.first << ": " << resultadoPDTD << std::
            endl;
286     auto duracion3 = std::chrono::duration_cast<std::chrono::milliseconds>(fin - ahora);
287     std::cout << "El Algoritmo de Prog. Dinamica (Top Down) tardo: " << duracion3.count() << "
            milisegundos" << std::endl;
288
289     std::cout << "-------------------------------------------------------" << std::endl;
290     std::cout << "Resolviendo con Programancion Dinamica (Algoritmo Bottom Up)" << std::endl;
291
292     ahora = std::chrono::_V2::steady_clock::now();
293     resultadoPDBU = subsetSumPDBU(entrada.second, entrada.first);
294     fin = std::chrono::_V2::steady_clock::now();
295
296     std::cout << "Tam. de conjunto minimo que suma " << entrada.first << ": " << resultadoPDBU << std::
            endl;
297     auto duracion4 = std::chrono::duration_cast<std::chrono::milliseconds>(fin - ahora);
298     std::cout << "El Algoritmo de Prog. Dinamica (Bottom Up) tardo: " << duracion4.count() << "
            milisegundos" << std::endl;
299     std::cout << "-------------------------------------------------------" << std::endl;
300
301     //Escritura de output en archivo
302     std::ofstream salida("output.csv", std::ios::app);
303
304     salida << entrada.second.size() << "," << entrada.first << ",";
305     salida << duracion1.count() << ",";
306     salida << duracion2.count() << ",";
307     salida << duracion3.count() << ",";
308     salida << duracion4.count() << ",";
309     salida << resultadoFB << ",";
310     salida << resultadoBT << ",";
311     salida << resultadoPDTD << ",";
312     salida << resultadoPDTD << std::endl;
313
314     salida.close();
315     return 0;
316 }
```