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 realiza para eliminar ramas que si bien llegan a una solución, se sabe que no es la optima. 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, tamMejorConj)
    if inicio == |S| then
        if sumActual == V then
            tamMejorConj ← minPos(tamMejorConj, longActual)
            return longActual
        else
            return −1
        end if
    else
        if not(tamMejorConj == −1) ∧ longActual >= tamMejorConjunto then
            return −1
        end if
        if sumActual == V then
            tamMejorConj ← minPos(tamMejorConj, longActual)
            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. También supone que $tamMejorConj$ se pasa por referencia.

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. El algoritmo posee dos **podas de optimalidad**, una 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. La segunda poda es la siguiente, al encontrar una solución al problema, se guarda la longitud del subconjunto solución como la mejor solución hasta el momento, si en otra rama de ejecución veo que la longitud del candidato a solución es más larga o de igual tamaño que la guardada, entonces se que por esa rama nunca voy a encontrar una solución más optima, entonces corto esa rama.

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 así:

```
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] ← 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] ← 0
            else
                if S[i] == j then
                    matriz[i][j] ← 1
                else
                    matriz[i][j] ← −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 ← subsetSumPDTDRec(S, i − 1, j, matriz)             ▷ Calculo el problema sin contar el elemento actual
        m2 ← subsetSumPDTDRec(S, i − 1, j − s[i], matriz)      ▷ Calculo el problema teniéndolo en cuenta
        if m1 == −1 and m2 == −1 then
            matriz[i][j] ← −1
        else
            if minPos(m1, m2) == m1 then
                matriz[i][j] ← m1
            else
                matriz[i][j] ← 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.

Vamos a ver que la complejidad del algoritmo es la pedida, primero se carga la matriz donde guardamos los resultados para no calcularlos de nuevo, eso es $O(N * V)$, ahora veamos que el resto da también $O(N * V)$. La función recursiva del algortimo se mueve por la matriz utilizando dos parámetros, i y j, i está acotado por $|S| = N$ y j esta acotado por V, como el valor mínimo que puede tomar cada uno es 0 (cuando alguna de las dos es $<= 0$ se llega a un caso base), entonces por las combinaciones posibles que pueden tomar los parametros, la funcion se llama a lo sumo $N * V$ veces. Ahora veamos los pesos de las llamadas recursivas, si la solución que busca la llamada no fue calculada anteriormente, entonces se van a realizar llamadas que eventualmente llegan a los casos base, que son $O(1)$, luego se realizan operaciones $O(1)$ para calcular el máximo valor devuelto por las llamadas recursivas y guardarlo en la matriz. Si la solución que busca ya había sido calculada entonces se obtiene la solución en $O(1)$ accediendo a la matriz. Entonces las llamadas recursivas son $O(1)$ y como a lo sumo se realizan $O(N * V)$ llamadas por lo dicho anteriormente. Si juntamos las complejidades entonces tenemos $O(N * V) + O(N * V) = O(2 * N * V) = O(N * V)$. Y entonces la complejidad del algoritmo es $O(N * V)$.

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 $O(|S|*V) = O(n*V)$.

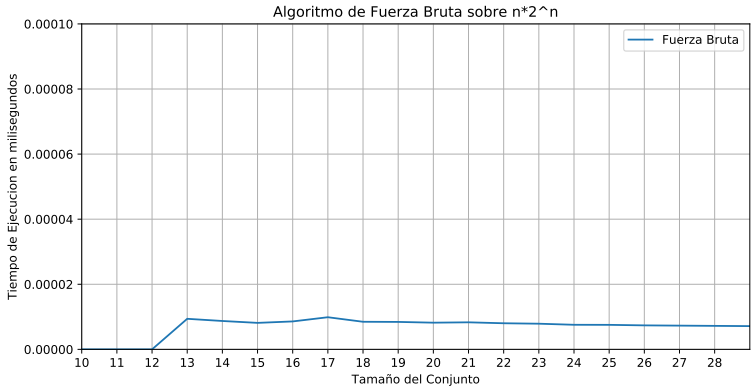
2. Experimentos

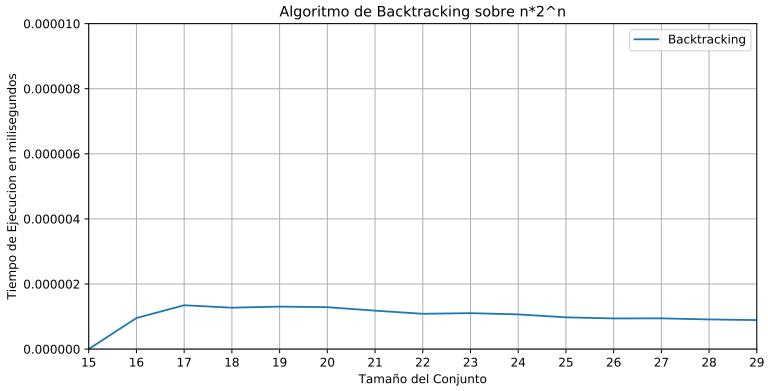
Como fue mencionado anteriormente nos interesa saber cuál de los algoritmos es más eficiente para resolver el *Subset Sum Problem*, por los cálculos de complejidad anteriormente realizados podemos intuir que los algoritmos de programación dinámica son los que van a tener ventaja sobre la fuerza bruta y el backtracking. Los test realizados en el experimento fueron generados de tal forma que trataran de acercarse al peor caso de los algortimos, Fuerza Bruta y Backtracking no tienen peor caso, ya que sin importar la entrada siempre calculan exactamente su complejidad. El algoritmo de Backtracking tiene un peor caso, si la entrada es tal que se ignoran las podas, entonces hace todos los cálculos posibles, para evitar las podas basta con que el valor objetivo sea mayor que la suma de los todos los elementos del conjunto, y que el conjunto no tenga 0. Debido a la manera en la que se implemento el algoritmo Top Down, tampoco tiene peor caso ya que como siempre tiene que generar la matriz donde guarda los datos, la complejidad es $O(N * V)$ siempre.

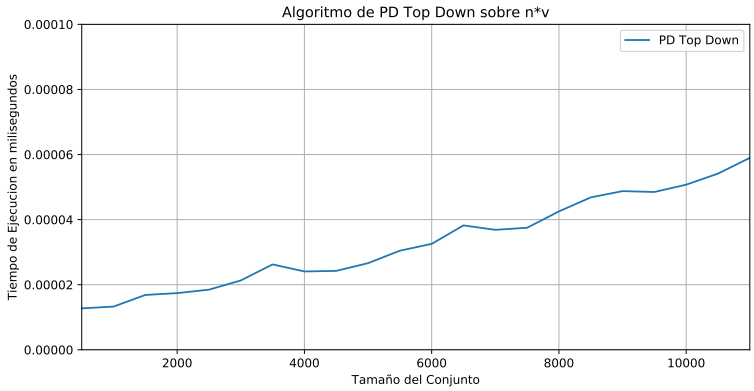
2.1. Experimentos sobre complejidad

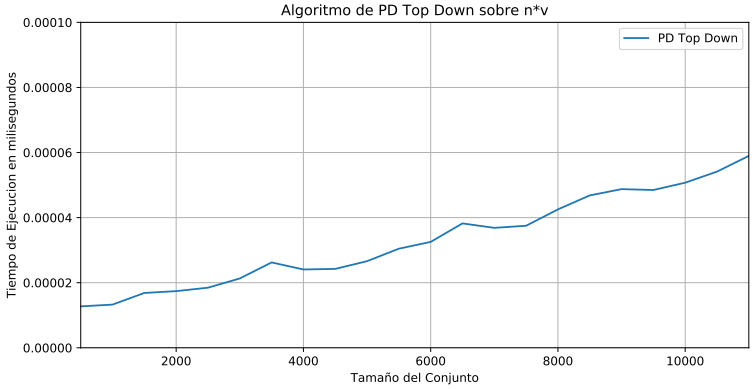
Antes de comparar los algoritmos vamos a ver que las complejidades calculadas para cada uno sean las correctas, si tomamos los resultados en tiempo que nos va dando cada algoritmo y los dividimos por la función que representa la complejidad de cada uno, el resultado debería dar acercase a 1, o ir decreciendo si el algoritmo es de menor complejidad. Si graficamos esto entonces deberíamos obtener una linea parecida a una función constante o una función lineal decreciente.

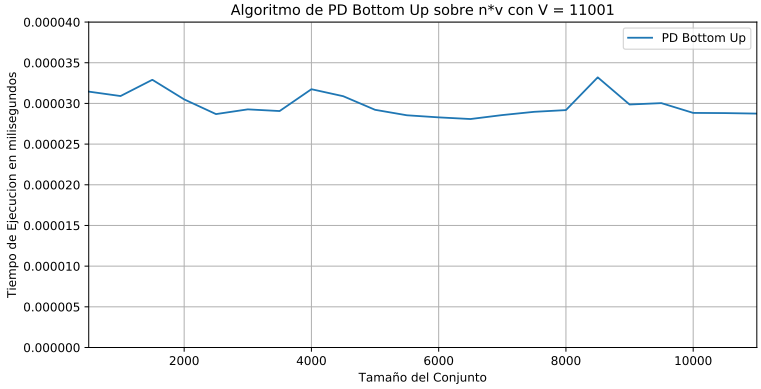
Los graficos resultantes son los siguientes:

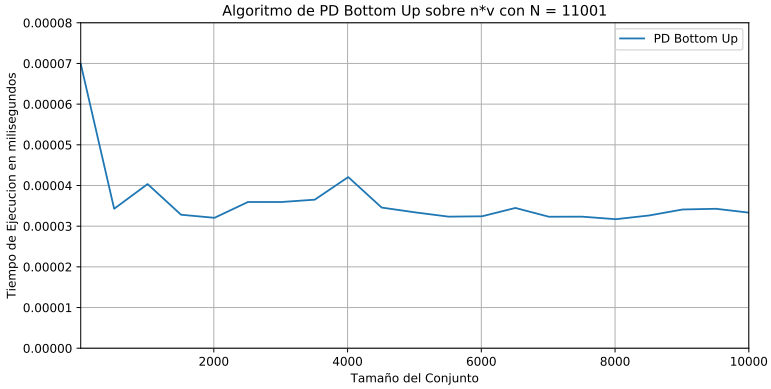








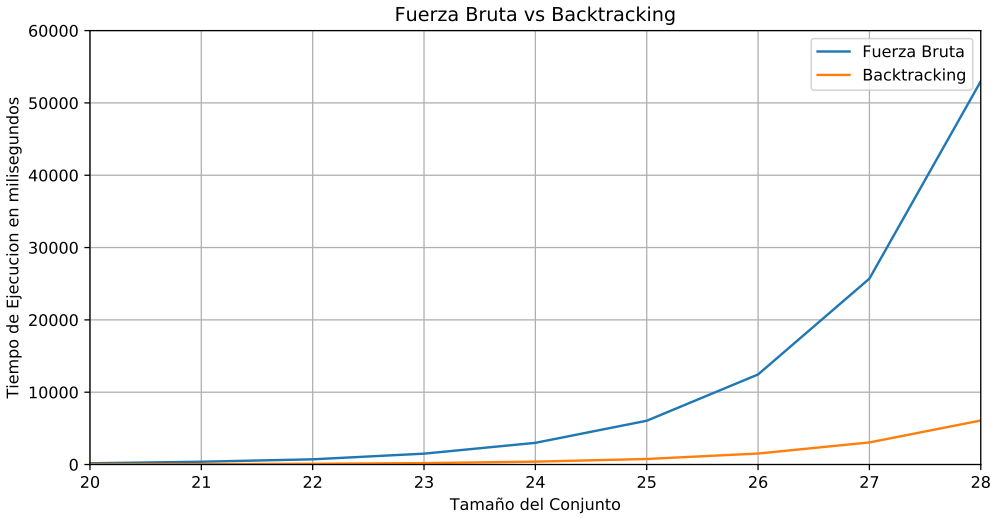




Entonces debido a que los gráficos son casi constantes o parecidos a funciones linealmente decrecientes podemos comprobar que la complejidad de los algoritmos es la calculada anteriormente.

2.2. Backtracking vs Fuerza Bruta

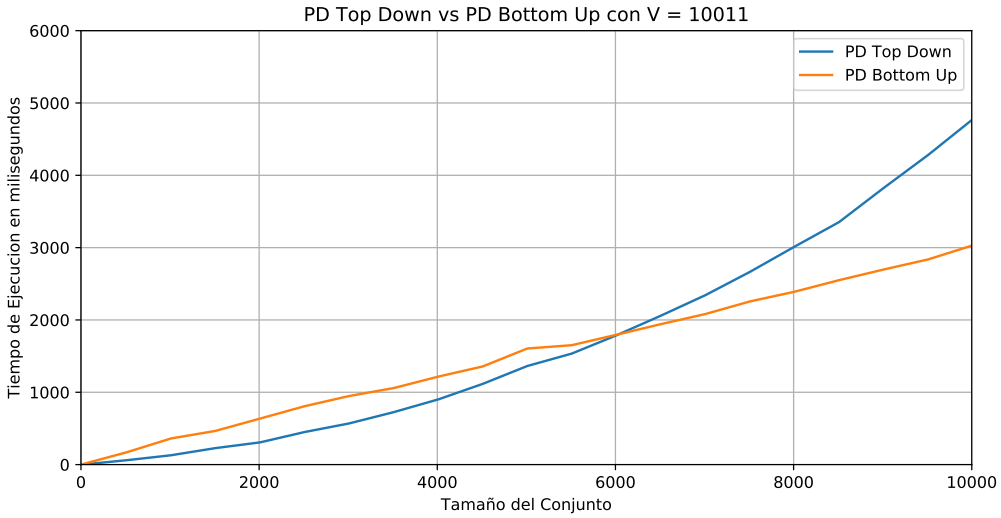
Los algoritmos de Backtracking y Fuerza Bruta tienen una complejidad exponencial, sin embargo como el Backtracking realiza podas para poder ahorrar cálculos innecesarios, seria interesante ver que las podas permitan resolver el problema antes de tiempo. Para eso se graficaron los tiempos que las funciones tardaron en resolver el Subset Sum Problem con valores de N cada vez más grandes. El grafico fue el siguiente:



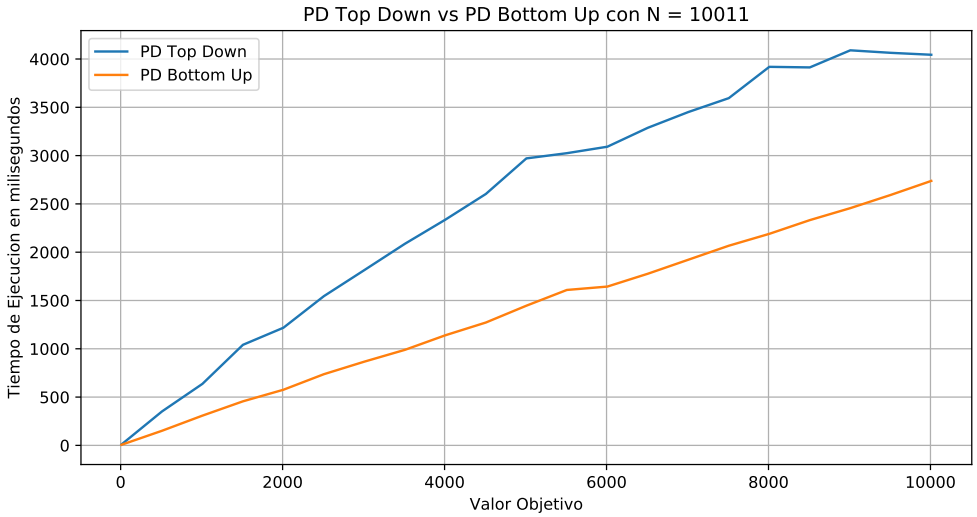
Se puede ver que aunque tomen los mismos valores de entrada, el algoritmo de Backtracking es mejor que el de Fuerza Bruta en todo momento. Esto se debe a que, aunque se eviten las podas, el algoritmo de Backtracking va calculando el mínimo a medida que se ejecutan las recurciones, mientras que el algoritmo de Fuerza Bruta tiene que calcular la suma de cada subconjunto candidato a solución en cada iteración, por lo que tarda más.

2.3. Comparación entre algoritmos de programación dinámica

Ya que los algoritmos Top Down y Bottom Up parecen tener una complejidad mejor que exponencial, 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.

2.4. Backtracking vs Programación Dinámica

3. Cambios