Algoritmos y estructuras de datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico I

Integrante	LU	Correo electrónico
Suárez, Romina Julieta	182/14	romi_de_munro@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

1.1. Problema a resolver: Eligiendo Justito

Este trabajo práctico se basa en la resolución del siguiente problema y el análisis de los algoritmos propuestos para el mismo. Se tiene un conjunto de números enteros y positivos (incluido el cero) de los cuales se busca obtener el cardinal del subconjunto que logre sumar un número T, el cual es un valor entero y positivo también. En caso de no poder llegar a esta suma, se debe devolver -1. Para resolver este ejercicio se ha pedido realizar 3 distintos algoritmos: La técnicas de Fuerza Bruta, Backtracking, y programación dinámica:
El primero se basa en la búsqueda de la respuesta de forma directa y analizando todos los posibles casos. La cual tiene como desventaja que es muy poco eficiente, y no suele llevar un registro ordenado de casos a analizar. La segunda técnica buscará todas las posibles combinaciones de subconjuntos creando un árbol de respuestas, de las cuales se tomará la mínima, de existir. Este proceso puede ser acotado, ya que al conocer el árbol, se pueden recortar ramas para agilizar la búsqueda. Por último, programación dinámica construirá las soluciones en base a las ya calculadas y sin repetir las que analizó previamente. Esto mejorará muchísimo la eficiencia, comparado con las anteriores.

Un par de ejemplos:
Ej.1
3 5
1
2
3
El resultado obtenido es 2.
Ej. 2
1 1
1
El resultado obtenido es 1.
Ej.3
1 0
1
El resultado obtenido es 0.

2. Desarrollo

Se utilizó el lenguaje C++ para el desarrollo del código y Python para los casos de test. Los vectores resultado y valores contienen: los valores que comprenden el resultado y los valores tomados por entrada respectivamente. Y resMin es una variable global inicializada en -1, para devolverla intacta si no se encuentra solución.

2.1. Algoritmo de Fuerza Bruta

Algorithm 1 void fuerzaBruta(int índice,int n, int v, vector <int>&valores, vector <int>& resultado)

```
1: if índice == n then
2:     Fijarse si se logro llegar a v y modificar el valor de resMin;
3: else
4:     if índice <n then                          ▷ Para no pasarme de los elementos disponibles
5:         incluir el elemento valores[índice] a resultado
6:         hacer recursión con el elemento
7:         quitar el elemento valores[índice] de resultado
8:         hacer recursión sin el elemento
```

Algorithm 2 int sumaDeVec(vector <int>& entry)

```
1: while i <entry.size() do
2:     res = res + entry[i]
3:     i++
4: return res
```

2.2. Explicación

Este algoritmo va directamente a crear a todos los subconjuntos posibles del conjunto original, y sumará todos los valores de cada uno. Si se logra obtener el número buscado, se actualizará la variable global resMin. La función sumaDeVec tomará el vector resultado actual y sumará sus valores, para comprobar si dicho vector suma V.

2.3. Algoritmo Backtracking

Algorithm 3 void backtracking(int índice,int n, int v, vector <int>valores, vector <int>resultado)

```
1: if resultado.size() <resMin then                                          ▷ poda1
2:     if llegue al resultado then
3:         modificar el valor de resMin;
4:     else
5:         if índice <n then                          ▷ Para no pasarme de los elementos disponibles
6:             if sumaDeVec(resultado) + valores[índice] <= v then           ▷ poda 2
7:                 incluir el elemento valores[índice] a resultado
8:                 hacer recursión con el elemento
9:                 quitar el elemento valores[índice] de resultado
10:            hacer recursión sin el elemento          ▷ El else se debe analizar siempre
```

2.4. Explicación

Desde el main cargamos los valores en el vector llamado 'valores', el valor n, y v, que serán el numero de valores a analizar y el valor al que se quiere llegar, respectivamente. Llamamos a backtracking con el valor 0, que es el primer valor del vector de valores que vamos a analizar. La función backtracking se encarga de realizar la recursión que agrega los valores al conjunto resultado. Así genera todo el árbol de posibilidades. La función sumaDeVec tomará el vector resultado actual y sumará sus elementos, para ver si es igual a v. Cuando se llega al final de la rama, se compara con el resMin antiguo y el tamaño del conjunto de resultado hasta ahora. Si este último es más pequeño, se actualiza el resultado. Al final de todas las ramas, el algoritmo toma resMin como respuesta de esa instancia. Se modificó el algoritmo inicial para que se pudieran tomar varias instancias seguidas para la experimentación. Eso se logró incluyendo a la entrada un doble cero al final del grupo total de entradas: Ej. 3 5

1
2
3
1 1
1
2 3
1
2
0 0

El algoritmo tomará cada entrada y llamará a la función backtracking hasta que lea un doble cero. Esto es gracias a un while antes de cargar la entrada al vector de valores.

2.5. Podas

La poda nº1 es "la poda de optimalidad". La misma corrobora que al llamado de dicha recursión, el vector resultado actual no tenga más elementos que el resMin. Si es así, backtracking solo seguirá agregando valores a

ese vector, lo cual nos llevara a una solución, si es que existe, aún más grande de la que ya teníamos. Haciéndonos perder tiempo, ya que dicha solución no nos servirá.

La poda nº2 es la de factibilidad, y es la que se encarga de podar el árbol en caso de que se vaya a agregar al conjunto de respuesta, un valor que ya supere V.Si esto ocurriese, ese valor no comprendería parte de la solución, y por lo tanto, no deberíamos seguir con dicha rama.

2.6. Correctitud

El algoritmo de fuerza bruta genera todos los posibles subconjuntos y analiza el valor de la suma de cada uno de ellos. Esto significa, que de existir la solución, se encontrará en uno de todos los subconjuntos analizados.Y Termina, pues, el grupo de partes de un conjunto es acotado.
El algoritmo de backtracking genera todas las posibles ramas de un árbol completo. Cada nodo del mismo es una solución parcial del conjunto de valores que se requiere y cada hoja es una posible solución al problema, de la cual nosotros queremos la que contenga el conjunto mínimo de valores que sume v y la misma se encuentra en una de estas hojas. Dado el ejercicio, este algoritmo descarta las soluciones que son invalidas al tomar la decisión de agregar a un valor o no. Por esto, no recorre el árbol completo, pues descarta las que al nodo actual, son invalidas. Las podas se encargan de cortar las ramas que no serán una respuesta al problema, pues o esa rama eran peores que la ya encontrada como respuesta, o ese conjunto es invalido. Por lo tanto, solo se descartan hojas que no deberían ser tomadas como solución o invalidas. Si se quitaran las podas, se generaría el árbol completo. Y de existir la solución, se encontraría en tiempo finito, pues el árbol generado es el conjunto de partes de el conjunto original de valores y el mismo es acotado.

2.7. Calculo de Complejidad

Siendo 'n' la cantidad de valores y 'v' el valor buscado, el código posee la siguiente complejidad, en peor caso, y sin podas:

- En el main, se cargan las encuestas, lo cual tarda O(n).
- Por cada elemento, la función backtracing (o fuerza bruta) realizará recursiones. Esta posibilidad se traduce en $O(2^n)$.
- Además, la función "$sumaDeVec$" toma $O(n)$ y se realiza cada vez que se llega al final de una rama.
- Por lo tanto, la cota final seria $O(2^n.n)$.
- Si agregamos las podas: La complejidad de la $poda_1$ es Lineal, es decir O(1) y la complejidad de la $poda_2$ es $O(1)$, también.

2.8. Algoritmo de Programación Dinámica

Dentro del main, tenemos:

```
1:  int f[n+1][v+1];
2:  f[0][0] = 0;
3: for  (int i = 1; i <= v; i++) do
4:      f[0][i] = INF;
5: for int i = 1; i <= n; i++) do
6:     for (int t = 0; t <= v; t++) do
7:         if  (valores[i-1] >t) then
8:             f[i][t] = f[i-1][t];
9:         else
10:            f[i][t] = min(f[i-1][t], 1 + f[i-1][t - valores[i-1]]);
11: if  (f[n][v] == INF) then
12:      return -1;
13: else
14:      return f[n][v];
```

2.9. Explicación

Para este algoritmo, realizaremos la versión bottom up. Esto significa que recorreremos la matriz de resultados desde el comienzo hasta el final, y no hacia atrás. Calculando los valores desde el primero hasta el último. Para ello se crea una matriz de n+1 por v+1. Es así, con una fila y columna de más, para poder utilizar la primera fila como la inicial del análisis de valores, y la primera columna, que nos servirá para analizar desde el 0 a v, incluido.
Luego rellenaremos la primera fila , donde cada casillero tendrá un numero infinito o INF, exceptuando la primera que tendrá el número 0.
Entonces comenzaremos a realizar el siguiente análisis. Si el valor actual del vector de valores es mayor que el t actual, entonces colocaremos el valor de la matriz que esté directamente sobre el nuestro.
Si no es mayor, y entonces, puede servir como respuesta, se buscará el mínimo valor entre el que se encuentra sobre nosotros, o 1 +, la respuesta que cumplió en la matriz para el t actual menos el valor[i] actual. En otras palabras, esto último significa, que incluiremos nuestro valor, y buscaremos dentro de la matriz, como se llego a cumplir el t, menos el valor que ya estamos agregando.
Así se completará toda la matriz. Dando como resultado el valor que buscamos en f[n][v], si es que existe. Por eso, al final se realiza la comprobación de la matriz y se responde.
Recordar que también se modifico el algoritmo para que tome varias instancias del ejercicio, y leyendo un doble cero al final, para saber que terminó de levantar entradas.
A continuación un ejemplo:
La entrada es:
3 5
1
2
3
Esta seria la matriz ya recorrida:

	0	1	2	3	4	5
0	0	INF	INF	INF	INF	INF
1	0	1	INF	INF	INF	INF
2	0	1	1	2	INF	INF
3	0	1	1	1	2	2

Tomando los índices del algoritmo, y siendo i las filas, y t las columnas, Véase que en f[1][1] se aplicó el mínimo entre tomar el finito inmediato de arriba, o 1 + el f[0][0] que es 0, dando nos como resultado, 1.

2.10. Correctitud

El algoritmo de programación dinámica termina, pues recorre toda la matriz, y la matriz es de celdas finitas. Es correcto pues siempre toma el mínimo entre tomar o no tomar ese valor dentro del vector solución. Entonces cada celda contiene un valor, y este es óptimo para dicha sub respuesta. Esto quiere decir, que si buscásemos la respuesta para v/2, la misma será la mínima y estará correcta dentro de la matriz, para ese numero parcial.

2.11. Calculo de Complejidad

Siendo 'n' la cantidad de valores y 'v' el valor buscado, el código posee la siguiente complejidad, en peor caso:

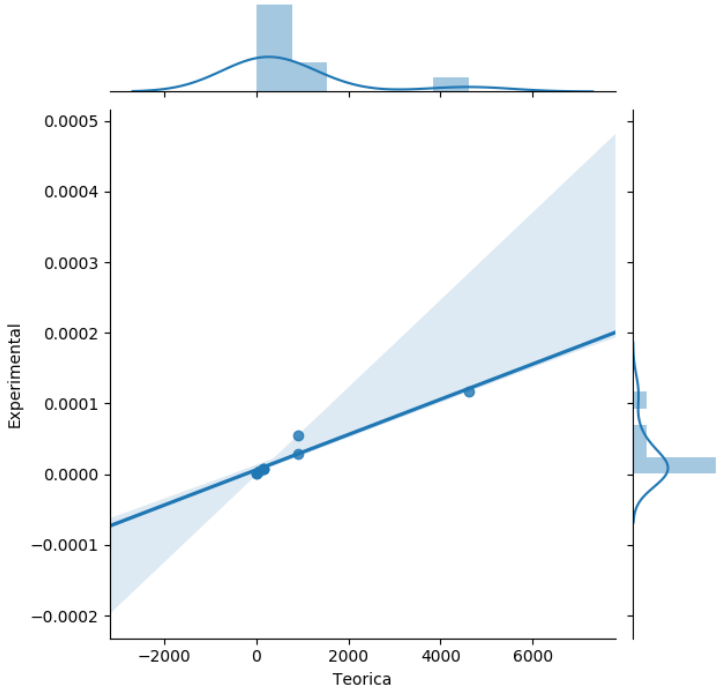
- En el main, se cargan las encuestas, lo cual tarda O(n).
- Se crea la matriz f con n+1 filas y v+1 columnas.
- Al ser bottom up, se recorrerá con los 2 fors todos los valores de la matriz, tomando lo que tarda en recorrese la matriz, y despreciando el +1 de n y v, tenemos una complejidad de $O(n.v)$.

3. Experimentación

Para los experimentos se utilizó Python para generar las muestras, y las librerías Matplotlib, panda y seaborn para generar los gráficos. Se realizaron dos pruebas de complejidad, la comparación de los 3 algoritmos, la comparación de fuerza bruta y backtracking con todas sus podas, y la comparación de backtracking con programación dinámica, modificando los valores n y v.

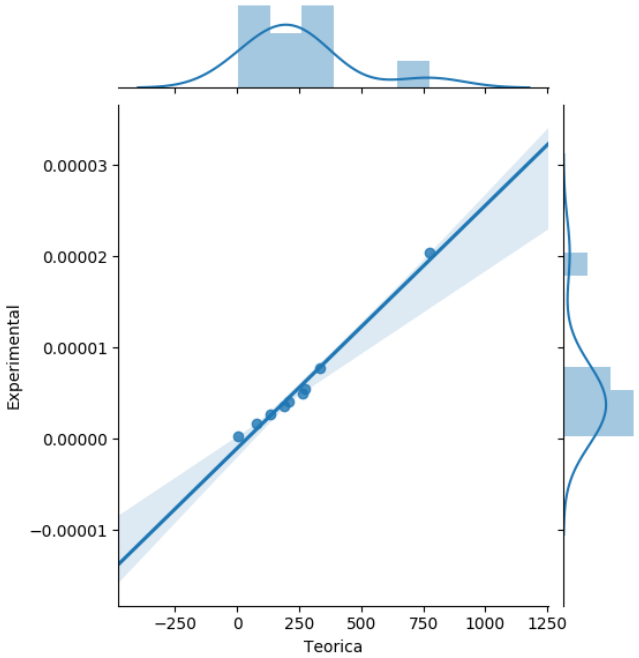
3.1. Complejidad Teórica vs Práctica

3.1.1. Fuerza bruta o backtracking sin podas



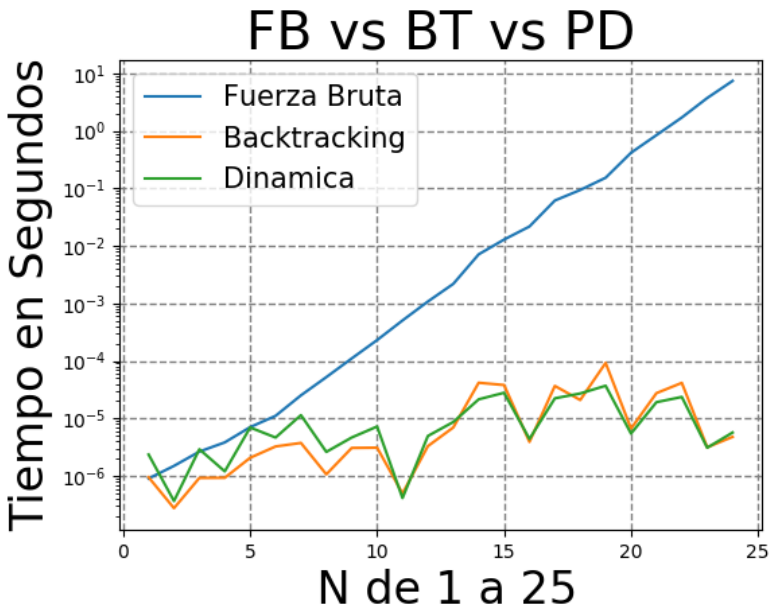
Este gráfico se generó para mostrar la correlación de fuerza bruta o backtracking sin podas, que es $O(2^n.n)$. Dado que ambas complejidades, teórica y práctica viven en mundos de tiempo distintos, solo podemos ver de que manera se relacionan, para comparar semejanzas.
Se tomaron 10 instancias al azar y se corrieron con el algoritmo. Luego con otro algoritmo tomando solo la entrada de n y multiplicándola por la complejidad $O(2^n.n)$.
Y se corrió el gráfico que muestra la correlación de pearson. El mismo nos dió un valor de 0.964642200255409, de correlación . Esto quiere decir, que si este valor es 1 o -1, o se aproxima mucho a ellos, se puede concluir que la complejidad teórica y experimental se relacionan directamente: Cuando una aumenta, la otra también lo hace y viceversa.
EL gráfico en si muestra como ambas funciones de complejidad se asemejan a la curva x=y, exceptuando casos donde discrepan y por cuanto, los cuales se ven como puntos sobre la recta. Los cuales en este caso son descartables ya que no se alejan demasiado de la curva.

3.1.2. Programación Dinámica



Para programación dinámica, se buscó crear el mismo gráfico que compruebe que la complejidad propuesta de $O(n.v)$ se relaciona directamente con la complejidad experimental.
Con 10 entradas al azar, este gráfico nos dió un valor de 0.9916114069917589, llegando a que la complejidad propuesta y alcanzada, son las correctas.

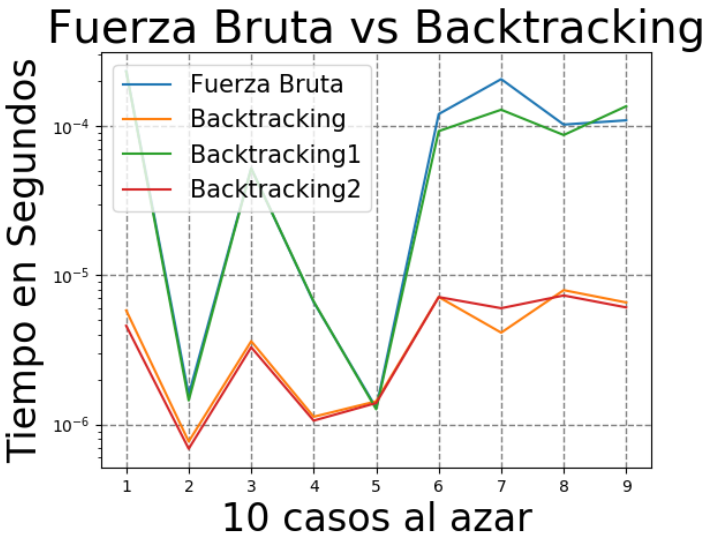
3.2. Los 3 métodos combinados



Hipótesis: Las complejidades nos van a mostrar que fuerza bruta es menos eficiente que backtracking, y este último lo es menos que Programación dinámica.

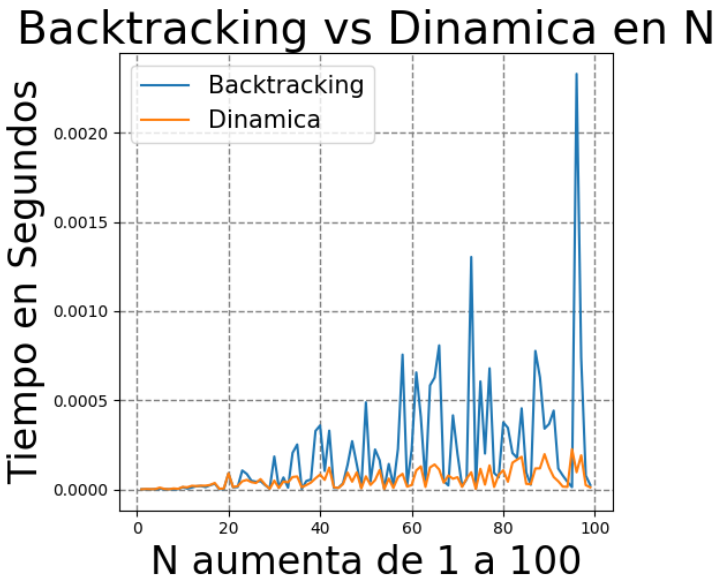
Para realizar este gráfico, n fue aumentando de 1 a 25 para ver como influye en las 3 complejidades, recordando que backtracking tiene sus podas, y sera más eficiente que fuerza bruta. Pero a bajos valores de n, backtracking y dinámica se comportaran de manera similar, pues ambas complejidades tardaran poco tiempo en correr. Para ver esta diferencia, se corrió el gráfico de backtracking vs dinámica en n, que está más adelante.

3.3. Fuerza Bruta vs Backtracking



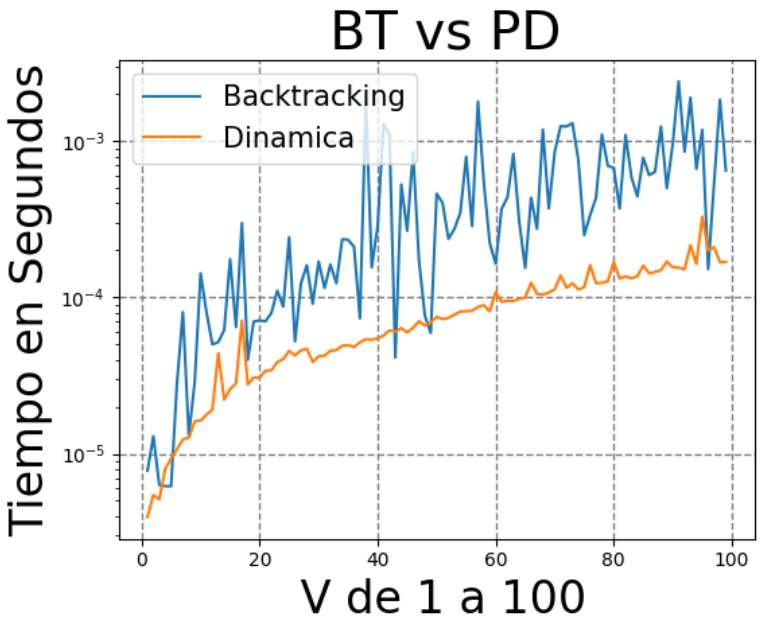
Hipótesis: En este caso, se quiso analizar cuanto influyen las podas, y cual de las 2 es mejor.
Como podemos ver, la poda 1 es la menos efectiva, pues puede podar solo en caso de que ya se haya logrado encontrar una solución. Incluso si se lograra esto, puede que hayan varias soluciones del mismo cardinal, pero de distintos valores, y no se podrían podar esas ramas.
Vemos también que la más eficiente es la poda 2, pues es la que corta, si o si, una rama si el valor a agregar al conjunto es mas grande, cortando muchos posibles caminos dentro del árbol.

3.4. Backtraking vs Dinámica con N en aumento



Hipótesis: A valores mayor de N, programación dinámica será mucho más eficiente que backtracking.
Se creo este gráfico para demostrar que a medida que n aumenta, la complejidad de backtracking empeora, pero la de dinámica es mas eficiente. Pues la complejidad de 2*n comienza a pesar mucho mas que v si comparamos ambas complejidades, $O(2^n.n)$ y $O(v.n)$. Para ello, Se mantuvo v, se realizaron 100 instancias, y n fue aumentando de 1 a 100.

3.5. Backtraking vs Dinámica con V en aumento



Hipótesis: A valores mayores de v, la complejidad de programación dinámica aumentará, pero seguirá siendo más eficiente que backtracking.
En este gráfico podemos ver como dinámica depende de v, que va aumentando de 1 a 100 (n se mantiene constante) y vemos como esto no afecta a backtraking que oscila por sus resultados.
Esto ocurre, pues dinámica depende del valor de v para crear la matriz y recorrer sus celdas. Por lo tanto, tardará más en terminar, pero seguirá teniendo mejor complejidad que backtracking.

4. Conclusiones

En este trabajo se pudo estudiar el uso de la técnica Backtracking y Programacion Dinamica y su forma de analizar un problema y resolverlo. En mi opinión, es muy útil cuando se tiene un problema con muchas posibles soluciones. Además, Se pudieron confirmar visualmente las hipótesis previamente generadas.