DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Eligiendo justito

16 de septiembre de 2018 — Algoritmos y Estructura de Datos III

Integrante	LU	Correo electrónico
Buceta, Diego	001/17	diegobuceta35@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

1. Introducción

1.1. El problema y sus análisis

El subset sum problem o problema de la suma de conjunto es un problema muy conocido dentro de las ciencias de la computación y matemática por ser un objeto de estudio dentro de la clase NP-problem [1], se destaca por su facilidad para describirlo y gran dificultad para resolverlo. Los algoritmos más conocidos que pueden resolverlo tienen una complejidad temporal exponencial y esto se traduce en imposibilidad de resolverlo en casos donde la cantidad de elementos sea grande.

Además de las diferentes versiones de este problema [2] existen usos muy diversificados en el ámbito cotidiano y científico. No solo sirve como fuente de nuevas investigaciones en el campo de la teoría de complejidad sino también en ramas como la criptografía, programación genética, etc[3][4].

Definición: Dado C conjunto de enteros no negativos y V entero no negativo se desea saber si existe un s $\subseteq$ S tal que $\sum_{r \in R} r = V$, y si existe que tenga la mínima cardinalidad.

Intuición Una de las primeras aproximaciones para buscar una solución puede ser la de formar todos los conjuntos posibles con los elementos de S y analizar si alguno suma V. Por un lado, este algoritmo asegura que si existe solución se va a encontrar; por el otro lado y como contrapartida de lo anterior, si no existe solución o S es muy grande la cantidad de pruebas que son necesarias superará las capacidades de cómputo actuales.

Otras formas Los análisis previos arrojan suficientes razones como para ahondar en otras soluciones.

Una algoritmo publicado por Horowitz Ellis y Sahni Sartaj [6][7] consiste en dividir los n elementos en dos conjuntos separados, cada uno con $\frac{n}{2}$ elementos. Para cada uno de los conjuntos se almacenan en dos listas la suma de los posibles subconjuntos junto con su cardinal. Posteriormente se utiliza algún algoritmo de comparaciones para ordenarlas, L_1 en orden decreciente y L_2 creciente, lo que suma una complejidad temporal de $2 \cdot O(2^{\frac{n}{2}} \cdot \frac{n}{2})$. Con las dos listas ordenadas, la idea principal es tomar un elemento de cada lista y compararlo con V. Por un lado se toma el elemento más grande (primero)de L_1 y el más chico (primero)de L_2. Entonces:

- Si la suma supera a V: El algoritmo avanza un elemento en L_1
- Si la suma es menor a V: El algoritmo avanza un elementom en L_2
- Si la suma coincide con V: se muestra la solución.

2. Algoritos presentados

En este trabajo presentaremos diferentes algoritmos con los que buscaremos experimentar y encontrar los contextos en los cuales sería beneficioso la aplicación de alguno de ellos.

2.1. Fuerza Bruta

Esta forma de resolver el problema está fuertemente relacionada con las primeras técnicas que surgieron para el problema de la suma de subconjuntos. Lo primero que debe hacerse es definir un dominio de posibles soluciones del problema, y posteriormente describir un algoritmo que permita alcanzarlas a todas.

En nuestro problema, vamos a definir nuestro espectro de posibles soluciones al conjunto de partes del conjunto inicial, S. Este conjunto estará formado por todos los posibles subconjuntos que se puedan tomar con elementos de S. Dado que cada elemento tiene dos opciones, estar o no en un determinado subconjunto, las opciones entonces son: $2 \cdot 2 \cdot 2 \cdots 2 = 2^n$

[1] Problemas de decisión y optimización donde se busca encontrar si existe solución o si existe una mejor que la conocida

[2] Problema de la carga de la mochila; Problema de selección de conjuntos de un conjunto, etc

2.2. Algoritmo

FuerzaBruta($Conjunto(enteros) : C, entero : V$)

```
1  i ← 0
2  P ← generarPartes(C)
3  sol ← −1
4  n ← Tam(C)
5  repeat
6          suma ← sumaElementos(P, i)
7          card ← cantElementos(P, i)
8          if suma == V then
9                  if sol == −1 then
10                         sol ← card
11                 else
12                         if sol > card then
13                                 sol ← card
14 until i <= 2^n
```

Función: generarPartes El código fuente del problema genera el conjunto de partes de la siguiente manera: una variable i se mueve entre 0 y 2^n - 1, y en cada uno de esos valores analiza su representación en bits y además otra variable j se mueve entre 0 y n-1. Es decir, cada vez que i actualiza su valor, j se actualiza n veces. En esas n comparaciones lo que se hace es analizar si el bit jésimo se corresponde un bit marcado en uno de i. Si esto pasa, significa que en el subconjunto iésimo debe estar el elemento j, y así para cada elemento. Esto se hace mediante el corrimiento de bits en O(1) que ofrece $C++$.

Función: sumaElementos y cantElementos Dado un conjunto de conjuntos y un valor, una retorna la suma y la otra el cardinal del subconjunto i-ésimo. A modo de una mejor visualización de las ideas del algoritmo estas funciones cumplen un objetivo simbólico y no tienen una correspondencia simétrica con el código, en caso de plasmar este pseudocódigo en un lenguaje, simplemente cambiará la complejidad espacial debido a que habría 2^n subconjuntos almacenados.

2.2.1. Complejidad

Desde el punto de vista de complejidad, si se define en función del tamaño de entrada, y tomando n como el tamaño de S, nuestro algoritmo será $\theta(n \cdot 2^n)$, porque se recorrerá cada subconjunto del conjunto de partes de S y para cada uno revisará cuáles de los n elementos de S están en ese subconjunto.

2.2.2. Código Fuente

```
int bruteforce(vector<int>& c,  long long int valorObjetivo,int &sol,long long int &pasos){
    int n = c.size();
    long long int total = pow(2,n);
    sol=-1;
    for (long long int i=0; i< total; i++) {
        long card=0;
        long sum = 0;
        for (int j=0; j<n; j++){
            pasos++;
            if (i & (1 << j)) { //chequeo si el elem jesimo tiene el bit en 1 (es decir, debe estar en el subconjunto i), si es 0 no
                esta entonces lo salto.
                card++;
                sum += c[j];
            }

        }
        if(sum == valorObjetivo) {
            if(sol==-1){
                sol = card;
            }else {
                if(sol > card) {
                    sol = card;
                }
            }
        }
    }
    (sol == -1 || sol >= n)? sol = -1:1;
    return sol;
}
```

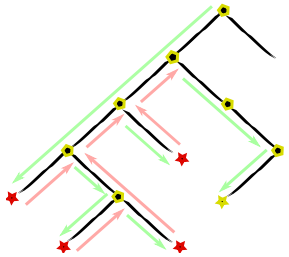
2.3. Backtracking

Lo que caracteriza a esta forma de resolver el problema es que vamos a buscar definir situaciones en las que seguir computando la solución no tenga sentido. Esto quiere decir abandonando posibles candidatos a solución por temás de optimalidad o posibilidad de la solución, a esto se lo denomina podas.

2.3.1. Podas

- Por optimalidad: Si en el proceso de computo de una posible solución, se llegara al caso en que el subconjunto de elementos actual contiene más elementos que la mejor solución encontrada hasta ahora, entonces no se buscará y computará ninguno de los subconjuntos que surjan de tener como los primeros k elementos del subconjunto actual y siendo k el tamaño de éste.
- Por factibilidad: Si en el proceso de cómputo de una posible solución, se llegara al caso en que la sumatoria de los elementos del conjunto formado actual supera a V, significa que no tiene sentido probar con a ninguno de los elementos hacia la derecha, porque están ordenados de forma creciente, y ninguno de los conjuntos que surjan de combinar los elementos restantes de S con el conjunto actual van a funcionar.

La siguiente imagen muestra la idea de las podas. Si tomamos el punto amarillo superior como el inicio de una parte de los posibles caminos del problema total vemos cómo las flechas verdes marcan los caminos que se siguieron por cada nodo y cómo al llegar a puntos problemáticos las flechas rojas marcan el retroceso y recorrido de otro camino, hasta agotarlos a todos. Los puntos problemáticos simbolizan otros inicios de conjuntos de caminos que por algún motivo se decide 'podarlos' y no analizarlos en profundidad.



[3]

2.3.2. Algoritmo

BACKTRACKING($Conjunto(enteros) : C, entero : sol, entero : V, entero : desde, entero : sumaElem, entero : cantElem$)

```
1  if sol = −1 or sol! = −1 and cantElem < sol then
2          if sumaElementos = V then
3                  if sol = −1 or sol > cantElem then
4                          sol ← cantElem
5  i ← desde
6  while i <= |C| and sumaElementos < V
7        do
8          if (sol = −1 or sol < cantElem) then
9                  cantElem ← cantElem + 1
10                 sumaElementos ← sumaElementos + C_i
11                 i ← i + 1
12                 Backtracking(C, sol, V, i, sumaElementos, cantElem)
13                 cantElem ← cantElem − 1
14                 sumaElementos ← sumaElementos − C_i
```

Detalle Como se mencionó anteriormente, suponemos que existe una función auxiliar que llama a esta con los valores de los parámatros bien definidos y C ordenado. La idea del algoritmo es recorrer cada elemento del conjunto de entrada a partir del flag *desde*. Mientras no se necesite podar, el algoritmo acumula el valor de los elementos y la cantidad de ellos entre *desde* y el final de C. Al mismo tiempo, se recursiona con un C' = C − $C_{iteracion_actual}$ y los flags actualizados, o sea, la variable de inicio del subproblema a resolver ahora comenzará en la iteración actual. A *sumaElementos* se le descontará el último elemento sumado y *cantElem* se reducirá en uno; que es igual a C sin el elemento de la iteración actual. Si se encuentra solución, se almacena en *sol*, sin embargo y dado que la diferencia de esta técnica con fuerza bruta es que se trata de apartar candidatos a solución por temas de optimización o factibilidad, el algoritmo termina después de haber intentado analizar todos los posibles subconjuntos del C inicial.
Al comienzo, esta función se llama con los flags *desde* = 0, *sol* = -1, *cantElem* = 0 *sumaElementos* = 0.

2.3.3. Complejidad

Respecto a la complejidad temporal y estableciéndola en función de tamaño de la entrada, |S| = n, analicemos cuál sería el peor caso. El peor caso sería que las podas que definimos no funcionen, y eso pasa si el V que se elige supera a la suma de todos los elementos de C. Con un caso así, nunca sería necesario podar soluciones por factibilidad, porque nunca existirá un subconjunto de C que sume más que V si la suma de C no lo suma y por otro lado como no habrá solución, nunca habrá

[3] Imagen tomada del blog de Steve Pemberton

podas por mejor solución.
En cada paso, el algoritmo realiza a lo sumo n pasos y la complejidad del algoritmo se puede establecer de forma recursiva con la siguiente funcion:

$$T_n = (n-1)T_{n-1} + (n-1), \tag{1}$$

Sea T' lo siguiente y definiendo y T'(0)=0:

$$T'_n = T'_{n-1} + n, n = 1, 2, 3...T'(n) = T'(n-1) + n = T'(n-2) + (n-1) + n \tag{2}$$

$$= T'(n-k) + n - k - 1 + n - k - 2 + ... + n - 1 + n = \sum_{i=1}^{n} i, \tag{3}$$

Por suma de Gauss:

$$= \frac{(n \cdot (n+1))}{2} = \frac{n^2}{2} \in O(n^2) \tag{4}$$

De modo que la complejidad de T' $\in$ O(n^2). Ordenar inicialmente a C no nos modifica la complejidad porque sería simplemente O(T(n)) + O($\log_2 n \cdot n$) que sigue siendo asintóticamente acotado superiormente por nuestra cota.

Veámos que T'$_n$ · (n-1) $\approx$ T$_{n-1}$ · (n-1) + n^2, que por lo visto antes $\in$ O(n) · O(n^2) $\cdots$ ¿$\in$ O($2^n \cdot n$)?
Sea C constante, vemos si existe un valor x_i tal que $\forall\ x_j \geq x_i$ a partir del cual valga:

$$n^3 \leq c \cdot 2^n \cdot n,\ \text{n} > 0, n^2 \leq c \cdot 2^n \tag{5}$$

Aplicamos función creciente a ambos lados sin cambiar la desigualdad

$$\log_n n^2 \leq c \cdot n, 2 \log_n n \leq c \cdot n \tag{6}$$

Y esto es cierto. Ahora falta ver si O(T$_n$) $\in$ O(T$_{n-1}$ · (n-1) + n^2).
Sea K constante, vemos si existe un valor x_i tal que $\forall\ x_j \geq x_i$ a partir del cual valga:

$$T_{n-1} \cdot (n-1) + n < K(\cdot T'_{n-1} \cdot (n-1) + n^2) \tag{7}$$

Si K = 1

$$n < n^2 \tag{8}$$

Entonces el algoritmo de backtracking $\in$ O($2^n \cdot n$)

2.3.4. Código Fuente

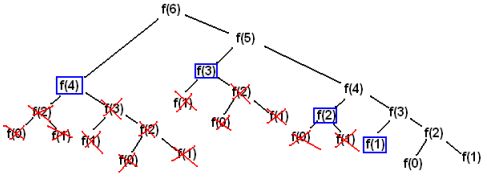
```
int backtracking(vector<int> c,  long long int valorObjetivo, int& sol,long long int &pasos) {
    sol=-1;
    sort(c.begin(),c.end()); //ordenamos de menor a mayor en aprox segun c++ (log |c| * |c|)
    auxBacktracking(c,0,valorObjetivo,sol,0,0,pasos);
    if(sol==-1 || sol >= c.size()) {
        return -1;
    }else{
        return sol;
    }
}

void auxBacktracking(vector<int>& c, int desde,  long long int valorObjetivo, int& sol, long long;  int sumaActual,  long long int
     cantElem,long long int &pasos){
    if(sol==-1 || (sol != -1 && cantElem < sol)){
        if(cantElem > 0) {
            if(sumaActual == valorObjetivo) {
                if(sol==-1){
                    sol = cantElem;
                }
                else {
                    if(sol > cantElem) {
                        sol = cantElem;
                    }
                }
            }
        }
    }
    for (int i = desde; i < c.size() && sumaActual < valorObjetivo; ++i) {
        pasos++;
        if( sol==-1 || sol!=-1 && sol > cantElem) {
            desde++;
            sumaActual +=c[i];
            cantElem++;
            auxBacktracking(c,desde,valorObjetivo,sol,sumaActual, cantElem,pasos);
            cantElem--;
            sumaActual -=  c[i];
        }
    }
}
```

2.4. Programación Dinámica

Esta técnica se basa en utilizar la metodología de *divide and conquer* [4] y almacenar los resultados calculados para no tener que volver a hacerlo cuando se repita un problema. Para poder aplicar usar programación dinámica es necesario que se cumpla entonces:

- El problema pueda ser dividido en subproblemas y que la solución que se encuentre para cada uno de ellos sea óptima, para así construir la solución del problema inicial de forma óptima [5]
- Los subproblemas compartan soluciones[6], si no no tendría sentido almacenar las soluciones.

En este gráfico se puede ver cómo los nodos que ya se calcularon están en azul y los rojos serían nodos que no se calculan porque ya se obtuvo la información que le corresponde.



Dentro de este tipo de programación hay varios algoritmos de implementación. Uno es el de top-down (por el significado en ingles de de arriba para abajo) o memoización y el otro bottom-up (por el significado en ingles de abajo hacia arriba), aunque existen bibliografías que consideran sólo a top-down como programación dinámica.
En este análisis se implementarán de ambas formas de forma que se pueda ahondar más acerca de los beneficios de ambos.

2.4.1. Análisis

La idea de un algoritmo que implemente *divide and conquer* y almacenamiento de subproblemas, sugiere empezar pensando este problema y dividiéndolo en problemas pequeños.

Se puede afirmar que la solución, si existe, puede dividirse en dos casos: la solución no contiene al primer elemento de C y entonces la solución resulte de formar V con la mínima cantidad de elementos de C':

$$C' = C - \{e_1\}$$

Y en particular debe ser óptima. Supongamos por el absurdo que no. La solución al problema sería K $\leq$ |C'|. Supongamos que existe un Q <K que es solución. Pero esto no puede pasar, porque por la propia definición de nuestra solución, se va a devolver la mínima cantidad de elementos, entonces no puede existir un valor menor de elementos que sea solución.

La otra posible solución sería que e_1 sí sea parte de la solución, es decir, habría que encontrar elementos que sumen V' con los elementos del conjunto C' :

$$C' = C - \{e_1\}$$

$$V' = V - e_1$$

O sea que la solución óptima sería 1 + K, K $\leq$ |C'|, K solución. Si existiese un Q <1+K y la suma de los elementos de Q = V', es decir hay una solución mejor para el problema con C' y V'. Pero entonces ahora K = Q y la solución óptima es K + 1, donde el 1 representaría el primer e_1 que por el análisis de este caso formaba parte de la solución óptima.

2.4.2. Formulación recursiva

Se define PD una función que recibe un conjunto S, de elementos enteros no negativos, un valor V entero no negativo y un entero que representa el tamaño de S inicial, devuelve el mínimo cardinal del subconjunto s perteneciente a S que todos los elementos suman V; si no existe se devuelve n, tamaño de S, simbolizando la imposibilidad de encontrar solución.

$$PD(S,V,n) = \begin{cases} n & \text{si } |S| = 0 \wedge V \neq 0 \\ 0 & \text{si V} = 0 \\ PD(\{e_2, e_3, \ldots, e_n\}, V, n) & \text{si } e_1 > V \\ 1 & \text{si } e_1 = V \\ \text{mín}(1 + PD(\{e_2, e_3, \ldots, e_n\}, V - e_1, n), PD(\{e_2, e_3, \ldots, e_n\}, V,)) & \text{otro caso} \end{cases}$$

[4] Metodología que busca simplificar un problema grande o complicado en problemas más chicos y simples de una forma recursiva
[5] Este concepto suele denominarse como: *principio del óptimo*
[6] Se denomina a esto subproblemas superpuestos y un ejemplo muy conocido es la sucesión de Fibonacci

2.4.3. Algoritmo Top Down

TOPDOWN$(C, V, desde, memoria, n)$

1 **if** $NoExiste(C, V, desde, memoria)$ **then**
2 **if** $desde < 0$ **then**
3 **if** $V = 0$ **then**
4 $devolver : 0$
5 **else**
6 $devolver : n$
7 **if** $c_{desde} == V$ **then**
8 $Guardar(1, V, desde, memoria, n)$
9 **else**
10 **if** $C_{desde} > V$ **then**
11 $valor \leftarrow Topdown(C, V, desde - 1, memoria, n)$
12 $Guardar(valor, V, desde, memoria, n)$
13 **else**
14 $valor1 \leftarrow 1 + Topdown(C, V - C_{desde}, desde - 1, memoria, n)$
15 $valor2 \leftarrow Topdown(C, V, desde - 1, memoria, n)$
16 $valor3 \leftarrow min(valor1, valor2)$
17 $Guardar(valor3, V, desde, memoria, n)$
18 $Devolver : -1$ *si* $memoria_{desde,V} \geq n$
19 $Devolver : memoria_{desde,V}$ *si no*

Esto asegura que cada problema que se quiera resolver primero verificará si no esta resuelto. Si lo está, accedemos a la información y la devolvemos. Si no está, se calcula, se guarda y se devuelve. Para tener un acceso eficiente, la implementación de memoria debe ser una matriz de |S| x V, donde $memoria_{desde,V}$ contiene la mínima cantidad de elementos que se necesitan para formar V con elementos de C_1 hasta C_{desde}. A nuestro algoritmo suponemos que llega ya iniciada, es decir, con -1 en todas las posiciones simbolizando que todavía no se guardo solución y en la posición (0,0) se completa con 0 como caso base. Desde es un dato bandera que se utiliza para saber desde dónde comenzar a buscar, porque el algoritmo parte de una problema que en caso de no tener solución pre-calculada se debe computar y es importante saber desde qué problema estamos se está haciendo esto. La primera vez que se llama la función desde = n, ya que hay que formar V con cualquier subconjunto de S. La ventaja de esta implementación con respecto a la próxima es que solo calcula los subproblemas que necesita y nunca calcula un subproblema más de una vez. Una de las desventajas es que al funcionar por recursión se almacena cada llamada recursiva y si la llamada genera un árbol muy grande podría aparecer un problema de memoria.

2.4.4. Código Fuente

```
int buscarValorObjetivoPD_top_down(vector<int> conj, long long int valorObjetivo, int& sol,long long int &pasos) {
    sol=-1;
    if(conj.size() == 0){
        if(valorObjetivo == 0) {
            sol=0;
            return 0;
        }else{
            sol=-1;
            return -1;
        }
    }else if(valorObjetivo == 0) {
        sol=0;
        return 0;
    }
    vector<vector<int> > soluciones(conj.size(),vector<int>((valorObjetivo+1),-1)); //inicializo matriz n x V con -1
    soluciones[0][0]=0;
    sol=aux_top_down(conj,valorObjetivo,conj.size(),soluciones, conj.size()-1,pasos);
    (sol > conj.size() || sol == 0)? sol=-1 : 1;
    return sol;
}

int aux_top_down(vector<int> conjActual, long long int valorObjetivo, int tamInicial, vector<vector<int> > & memoria, int desde,
 long long int &pasos) {
    pasos++;
    if(desde < 0) {
        if(valorObjetivo==0) {
            return 0;
        }else {
            return tamInicial+1;
        }
    }
    if(memoria[desde][valorObjetivo] == -1) { // si no esta calculada
        if(conjActual[desde] == valorObjetivo){
            memoria[desde][valorObjetivo] = 1;
        }else if(conjActual[desde] > valorObjetivo) {  //si el valor actual es mayor que el valor objetivo no puede ser parte de la
             sol
            memoria[desde][valorObjetivo] = aux_top_down(conjActual,valorObjetivo,tamInicial,memoria,desde-1,pasos);
        }else{
            memoria[desde][valorObjetivo] = min(1 + aux_top_down(conjActual,valorObjetivo-conjActual[desde],tamInicial,memoria,desde
                 -1,pasos), aux_top_down(conjActual,valorObjetivo,tamInicial,memoria,desde-1,pasos));
        }
    }
    return memoria[desde][valorObjetivo];
}
```

2.4.5. Algoritmo Bottom Up

BottomUp(C, V, n)

1 $memoria \leftarrow$ *matriz de* $n \cdot V$ *de enteros vacia*
2 $cargarConValores(memoria, n)$
3 $memoria_{0,0} \leftarrow 0$
4 **for** $i \leftarrow 2$ **to** n
5 **for** $j \leftarrow 1$ **to** V
6 **if** $C_i > j$ **then**
7 $memoria_{i,j} \leftarrow memoria_{i-1,j}$
8 **else**
9 $valor1 \leftarrow memoria_{i-1,j}$
10 $valor2 \leftarrow 1 + memoria_{i-1,j-c_j}$
11 $memoria_{i,j} \leftarrow min(valor1, valor2)$
12 *Devolver* : -1 *si* $memoria_{n,V} \geq n$
13 *Devolver* : $memoria_{n,V}$ *si no*

A diferencia del algoritmo anterior donde a partir de un input se buscaba si estaba almacenada su solución y posteriormente si estaba se devolvía y si no se calculaba y devolvía, en este directamente a partir de un input, se computan todos los posibles subproblemas del input en una matriz de n · V y al final de todo se devuelve el problema inicial, es decir, se construyen todas las subsoluciones hasta llegar al problema grande y se devuelve su solución. En el algoritmo se puede ver los movimientos que hace por la matriz son: si el elemento en la posición i, C_i es mayor que el valor objetivo, entonces la solución va a venir de buscar en la fila anterior. Si en cambio esto no pasa, la solución también puede ser ir a la fila anterior o agregar el elemento c_i a la solución y analizar ahora un nuevo problema, donde la busqueda de la solución será de la fila anterior pero con la columna movida hacia la izquierda, dado que cambió el valor objetivo.

La desventaja de esta implementación es que siempre se calculan todos los subproblemas, aunque como ventaja podría reducirse la complejidad espacial solo almacenando las filas que se vayan a utilizar, analizando cuál es el workflow dentro de las celdas de la matriz y eliminando las ya utilizadas.

2.4.6. Código Fuente

```
int buscarValorObjetivoPD_bottom_up( long long int &valorObjetivo, vector<int> &conj, int& sol,long long int &pasos) {
    sol = -1;
    if(conj.size() == 0){
        if(valorObjetivo == 0) {
            sol = 0;
            return 0;
        }else {
            sol = -1;
            return -1;
        }

    }else if(valorObjetivo == 0) {
        sol = 0;
        return 0;
    }
    vector<vector<int> > soluciones;
    soluciones.resize(conj.size()+1);
    for (int i = 0; i < conj.size()+1; ++i) {
        soluciones[i].resize(valorObjetivo+1);
    }
    vector<int> c = conj; //agrego un elemento a C para poder indexar de forma mas clara
    c.push_back(0);
    int aux = c[c.size()-1];
    c[c.size()-1] = c[0];
    c[0] = aux;
    for (int j = 0; j < valorObjetivo+1; ++j) {
        soluciones[0][j]=c.size();
    }
    soluciones[0][0]=0;
    for (int i = 1; i < c.size(); i++) {

        for (int j = 0; j < valorObjetivo+1; ++j) {
            pasos++;
            if(c[i] > j ) {//si el elemento actual supera el valor objetivo no puede ser solucion
                soluciones[i][j] = soluciones[i-1][j];
            }else {
                soluciones[i][j] = min((soluciones[i-1][j]),1+(soluciones[i-1][j-c[i]]));
            }
        }

    }
    sol = soluciones[c.size()-1][valorObjetivo];
    (sol > conj.size() || sol == 0)? sol=-1 : 1;
    return sol;
}
```

2.4.7. Complejidad

Podemos asegurar que la cantidad de operaciones del algoritmo nunca excederá n x V, que es la complejidad a la que estamos ajustados, dado que las operaciones en cada paso reducen al valor objetivo actual o la cantidad de elementos con los que operar. Esto quiere decir que cada paso mueve nuestro *celda subproblema* empezando desde la última posición y moviéndose en cada paso una fila hacia arriba o columnas hacia la izquierda, hasta que el rango de la matriz no lo permite. En el caso del bottom-up la complejidad en cualquier caso $\in O(n \cdot V)$, mientras que en el top-down sólo la cota superior $\in O(n \cdot V)$.

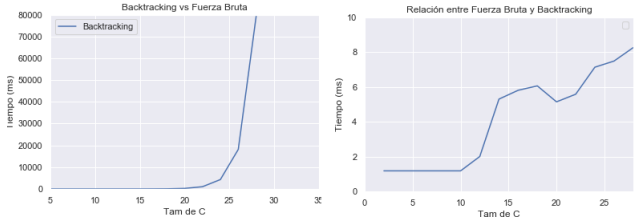
3. Experimentación

Las pruebas que se hagan con valores random se harán utilizando la libreria numpy en python y con diferentes distribuciones. Además, se testearán varias veces cada instancia en cada algoritmo para sacar un promedio de los tiempos de ejecución reduciéndo al mínimo las posibles oscilaciones circunstanciales.

3.1. Experimento 1

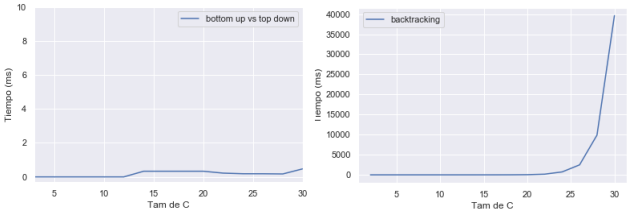
Los primeros análisis sugiere empezarlos comparando los algoritmos más básicos como fuerza bruta y backtracking. Posiblemente el algoritmo de fuerza bruta no permita ser computado para tamaños de C excesivamente grandes y los otros algoritmos sí, entonces sería razonable que se experimenten los algoritmos de forma cruzada y estableciendo relaciones y no todos de forma simultánea porque podría perderse información. Además, la implementación de los algoritmos sugiere que ante contextos de C pequeños, el crecimiento de ambos es similar hasta un determinado valor de n, mientras que para casos de C grandes fuerza bruta no sería opción. Veámoslo.

Fijemos los valores de los elementos de C de forma aleatoria y acotados por n y al valor objetivo $=n^3$, asegurándonos con esto que las podas nunca van a funcionar, dado que nunca C sumará el valor objetivo y como no habrá solución tampoco se aplicará la poda por optimalidad. Esto significa que no conseguiremos exactamente un peor caso si no uno promedio, dado que estamos generando los elementos de forma random bajo determinados parámetros. Evaluemos para instancias de n $= 1,2,\cdots 26$ que parecen ser instancias representativas del problema y dada la complejidad de fuerza bruta es posible con n superiores el tiempo de ejecución no sea razonable, de modo que para evitar la pérdida de información por falta de instancias para n más grandes lo vamos a reemplazar con más instancias.



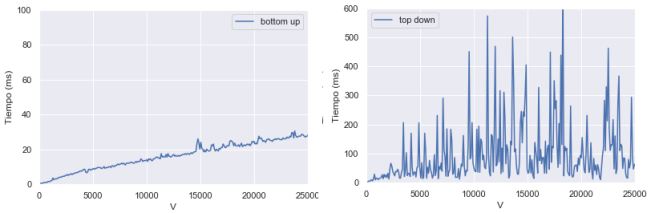
Se puede ver la relación entre ambos algoritmos en función del tiempo. Por un lado, para contextos de tamaños de C chicos (hasta 20 elementos) los dos algoritmos tienen tiempos de ejecución similares, por eso se ve un tramo constante. Después hay crecimiento de fuerza bruta respecto de backtracking, lo cual coincide con el pensamiento inicial de que fuerza bruta es una solución *no tratable* como habíamos visto en la introducción. Veamos por último que efectivamente un gráfico del algoritmo de fuerza bruta demuestra que no sólo tiene un crecimiento exponencial sino que este ya es demasiado grande para n ≥ 25.

Analicemos cuál es el resultados de utilizar algoritmos de programación dinámica en este tipo de instancias:



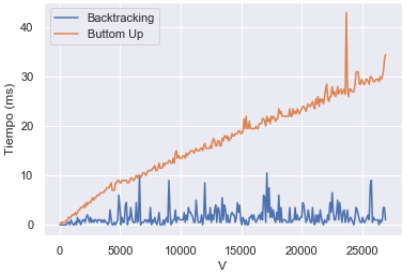
Efectivamente los algoritmos de programación dinámica tienen tiempos similares en este tipo de instancias y el backtracking tuvo un comportamiento de crecimiento exponencial, aunque siendo ampliamente mejor ante un fuerza bruta. Con este gráfico vemos cómo el backtracking en las instancias de peor caso (elementos de C de orden muy chico respecto a V) y analizando una cantidad suficiente de instancias tiene un crecimiento exponencial como indica la complejidad teórica a la que pertenece.

Para terminar este análisis fijemos un n, los valores de C entre 0 y V y al valor objetivo entre n y n^3 para tener una gama amplio respecto de V. Veamos entonces más en profundidad si estas relaciones entre los algoritmos de programación dinámica realmente son por las relaciones entre V y elementos de C.



La razón de visualizar la información obtenida en gráficos separados es que ambos se mueven en diferentes escalas de tiempo, pero sin embargo hay hechos que se pueden resaltar. Ambos crecen a medida que V crece, aunque bottom up lo hace de forma mucho más estable que top down, que es parte de lo que queríamos probar. El motivo de las oscilaciones debe tener varias razones, intentemos explicar algunas: al determinar instancias tan grandes de datos se hace necesario generarlos aleatoriamente bajo determinados parámetros, nada quita que la aleatoriedad genere casos borde que no pertenecerían a la media de instancias que buscamos generar. Por otro lado pero también afectado por esto, cuando existen instancias de tanta densidad de elementos de C acotados ampliamente por V, las llamadas recursivas de top down son cada vez más grandes y además no son gratis, hay que almacenarlas, si en particular una llamada genera un árbol de subproblemas excesivamente grande el problema es enorme.

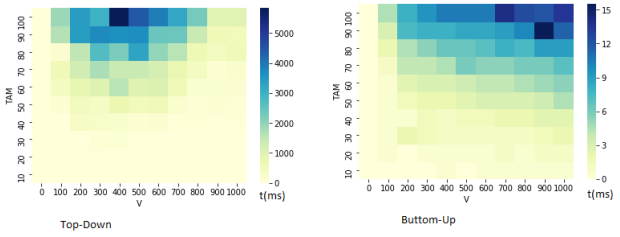
Un hecho curioso sacado de esto es que el backtracking, en este tipo de casos donde hay muchas chances de que las podas tengan una alta actividad, no solo compite sino mejora al bottom up como se ve a continuación:



3.2. Experimento 2

Ahora la cuestión es ver si existen instancias para las cuales convenga utilizar top-down en lugar de bottom up o backtracking. Para sacar ventaja frente a Bottom up necesitamos que $n \cdot V$ sea grande. Para lo siguiente construimos intanicias incrementando n y V simultaneamente, y los elementos de C generamos aleatoriamente de forma uniforme acotados por V.

En el siguiente mapa de calor podemos ver de forma separada como es la relación de los tiempos de ejecución de cada algoritmo en función del tamaño de C y del valor objetivo. Como primer detalle vemos que las escalas de tiempo son diferentes: el top-down tiene gran parte de sus promedios de tiempos en los 5 segundos mientras que para el mismo tipo de instancias bottom up tiene promedios de 15. Por otro lado, vemos que en el bottom up los tiempos están más en función de n y V, es decir, es un algoritmo estable pero que se ve afectado por los tiempos que consume moverse en una matriz cada vez mas grande, a diferencia de top down. Este parece centraliza sus tiempos más grandes en una franja intermedia de los V. Por lo que venimos analizando, esto podría relacionarse con "peores"peores casos que al computarlos genera estas oscilaciones que venimos viendo, de modo que el top down ante casos en donde resuelve pocos subproblemas no se ve afectado por el $n \cdot V$.

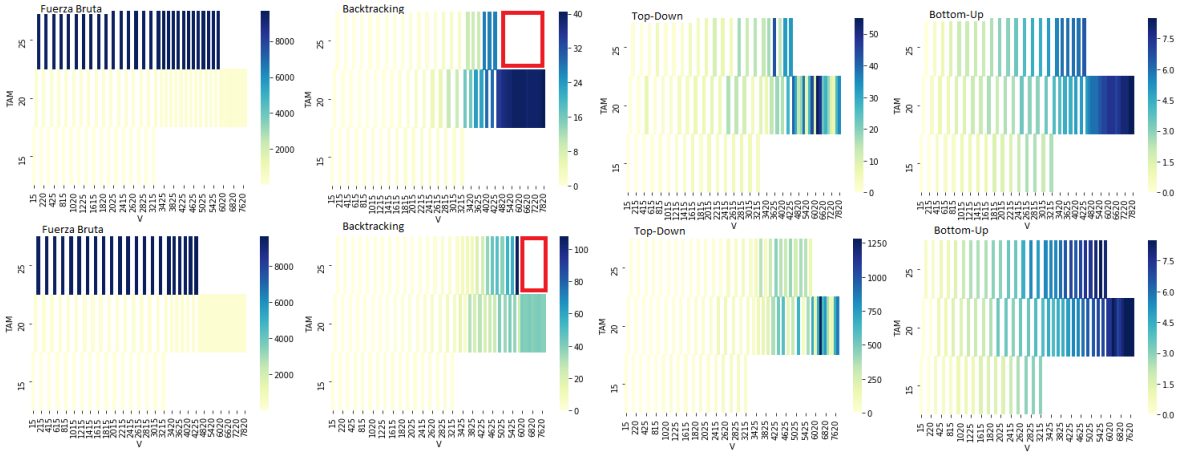


3.3. Experimento 3

En la simulación de diferentes situaciones de la vida cotidiana se pueden utilizar distribuciones normales la generación de determinadas instancias. Sería interesante entonces analizar situaciones en las que los elementos

estén generados de esta forma. Analicemos la siguiente, fijemos n en 15,20,25, que nuestros experimentos arrojaron que eran valores razonables para probar simultáneamente los algoritmos: si los elementos de C siguen una distribución normal de media n, el valor objetivo lo hacemos incrementarse hasta n^3 y la varianza $= \frac{n^2}{V}$ inicialmente y después varianza $= \frac{V}{n}$. En el primer caso es una instancia en donde a medida que crece V, los elementos de C sufren una menor dispersión. Esto puede indicar que, en Backtracking mientras V no sea de un orden mucho mayor a n, como los elementos van a estar cerca de V las podas van a tener mucho trabajo y el algoritmo será eficiente. Por otro lado, si la dispersión se incrementa con V, Backtracking va a empeorar su performance debido a su reducción en el uso de las podas.

Se presentan los mapas de calor en función de V y n. En la parte superior están los gráficos cuando la dispersión disminuye cuando aumenta V.



Finalmente las ideas principales se puede apreciar que para Fuerza Bruta no hay cambios debido a que en cualquier caso realiza todos los subproblemas. Por otro lado Top-Down empeora considerablemente cuando la dispersión aumenta, aunque sorprende su alta eficiencia cuando es baja la dispersión. Lo que pasa es que no habíamos notado que Top-Down almacenará los resultados de los subproblemas y si esos subproblemas son muy parecidos (baja dispersión de los elementos) entonces se beneficiará mucho de esto, y esto evidentemente no se dará con una alta dispersión.

Respecto de Backtracking hubo ciertas diferencias. Posiblemente se deba a que el hecho de que haya o no dispersión lo afecta en tanto y cuanto los elementos no se alejen mucho de V, porque en ese caso casi que actuará con un Fuerza Bruta. Como nota: marcas rojas indican que por temás de tiempo de procesamiento no se llegaron a completar.

4. Conclusiones

Los algoritmos que utilicen fuerza bruta quedan relegados para situaciones muy singulares, con tamaños de C muy pequeños, y su ventaja responde a la facilidad con la que se puede diseñar. Una mucho mejor opción se vio que era el Backtracking, no solo porque también es relativamente sencillo de diseñar sino porque permite ser útil en mayor variedad de situaciones. Vimos que si bien ante casos de C muy grandes padecía lo mismo que Fuerza Bruta, hay casos en lo que esto no pasa. Estos casos son cuando el orden de gran parte de los elementos de C supera al de V, porque vimos que el comportamiento en estas situaciones era casi lineal debido a la cantidad de combinaciones de soluciones que se salteaba.

Desde otro punto de la programación encontramos a Top-Down y Bottom up. La realidad en la experimentación es que Top-Down no fue tan beneficiado por los números, Bottom up demostró ser muy rápido para grandes muestras de tamaños de n y V, incluso los razonablemente grandes, y había que incrementarlos mucho para empezar a notar una deficiencia a base de los movimientos que debe hacer por la matriz. Incluso, en los mejores casos de Backtracking quedó ciertamente relegado para este tipo de casos, dado que al aumentar mucho n y V comenzaba a incrementar los tiempos, mientras que Backtracking al no tener este problema supo sacar las ventajas.

También es necesario mencionar que las facilidades para diseñar Bottom Up son propias de este problema y en otros podría ser realmente difícil, lo cual sumaría ventaja a Top-Down que es muy sencillo.

Como líneas finales queda resaltar la variedad con la que se puede intentar hacer frente a un problema, y lo importante de los análisis y experimentos que pueden trazar una ayuda acerca de los mejor pasos a seguir en situaciones determinadas.

5. Referencias

[3]Dynamic training subset selection for supervised learning in Genetic Programming, Chris GathercolePeter Ross.

[4]Generalizing Cryptosystems Based on the Subset Sum Problem, Aniket Kate and Ian Goldberg
[5]Thomas Cormen, Algorithms and Optimization, Approximation Algorithms, pág 1106
[6]G. J. Woeginger, Exact Algorithms for NP-Hard Problems: A Survey, Lecture Notesin Computer Science 2570, pp. 185-207
[7]Computing partitions with applications to the knapsack problem, Journal of the Association for Computing Machinery
[8]Donald Knuth. *The Art Of Computer Programming : Fundamentals Algorithms*