DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Eligiendo justito

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

Integrante	LU	Correo electrónico
Buceta, Diego	001/01	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

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre 2018
Trabajo Práctico 1

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

Eligiendo justito

Contexto y motivación

En muchas ocasiones sucede que se tiene un conjunto de ítems con un valor asociado y se quiere elegir un subconjunto de ellos que sume exactamente un valor objetivo. Esto en general está asociado a situaciones en donde se tiene un cierto recurso limitado y se quiere encontrar la manera de elegir ciertos ítems que completen de manera exacta ese recurso. Encontrar el mínimo o el máximo de la suma de los ítems es una tarea muy sencilla. Sin embargo, saber si se puede sumar exactamente un número dado, es una tarea mucho más compleja.

El problema que se menciona se puede encontrar en la literatura como el *Problema de suma de subconjuntos* o *Subset Sum* en inglés. El problema es fundacional en Ciencias de la Computación y se encuentra en la lista de 21 problemas *difíciles* presentada por Karp en [1].

Existen muchos problemas del mundo práctico en donde se debe resolver un *Subset Sum* o una variación del mismo. Supongamos que tenemos un conjunto de procesos que tenemos que asignar a dos procesadores. Cada proceso tiene asociado un valor t_i que es la cantidad de slots de tiempo que requiere para ser ejecutado. Lo que desearíamos en este contexto es dividir los procesos de tal manera que minimicemos el tiempo final T requerido para ejecutarlos todos. Para lograr esto, lo ideal sería dividir de manera equitativa los procesos. Es decir, encontrar un subconjunto de tareas cuya duración sume exactamente $\frac{T}{2}$. Esta situación no es más que un caso particular del problema de *Subset Sum*. En particular a esta especialización del problema (en donde hay que sumar exactamente la mitad del total) se lo conoce como *Partition Problem*.

Así como la mencionada, hay otras situaciones en donde se requiere de este problema, el cual no tiene una resolución algorítmica simple, lo que motiva su estudio en este trabajo.

El problema

Dado un conjunto de n ítems S, cada uno con un *valor* asociado v_i, y un valor objetivo V, decidir si existe un subconjunto de ítems de S que sumen exacto el valor objetivo V. Si existe dicho conjunto, decir cual es la mínima cardinalidad entre todos los subconjuntos posibles. En otras palabras decidir si existe $R \subseteq S$ tal que $\sum_{i \in R} v_i = V$, y si existe, devolver la menor cardinalidad posible de R.

Para este problema, asumiremos que todos los valores mencionados son enteros no negativos.

Enunciado

El objetivo del trabajo práctico es resolver el problema propuesto de diferentes maneras, realizando posteriormente una comparación entre los diferentes algoritmos utilizados.

Se debe:

1. Describir el problema a resolver dando ejemplos del mismo y sus soluciones.

Luego, por cada método de resolución:

2. Explicar de forma clara, sencilla, estructurada y concisa, las ideas desarrolladas para la resolución del problema. Para esto se pide utilizar pseudocódigo y lenguaje coloquial combinando adecuadamente ambas herramientas (**¡sin usar código fuente!**). Se debe también justificar por qué el procedimiento desarrollado resuelve efectivamente el problema.
3. Deducir una cota de complejidad temporal del algoritmo propuesto (en función de los parámetros que se consideren correctos) y justificar por qué el algoritmo desarrollado para la resolución del problema cumple la cota dada.
4. Dar un código fuente claro que implemente la solución propuesta.

   El mismo no sólo debe ser correcto sino que además debe seguir las *buenas prácticas de la programación* (comentarios pertinentes, nombres de variables apropiados, estilo de indentación coherente, modularización adecuada, etc.).

Por último:

5. Realizar una experimentación computacional para medir la performance de los programas implementados, comparando el desempeño entre ellos. Para ello se debe preparar un conjunto de casos de test que permitan observar los tiempos de ejecución en función de los parámetros de entrada, analizando la idoneidad de cada uno de los métodos programados para diferentes tipos de instancias.

A continuación se listan los algoritmos que se deben considerar, junto con sus complejidades esperadas (siendo n la cantidad de ítems a considerar y V el valor objetivo):

- Algoritmo de fuerza bruta. Complejidad temporal perteneciente a $\mathcal{O}(n \times 2^n)$.
- Algoritmo de *Backtracking*. Complejidad temporal perteneciente a $\mathcal{O}(n \times 2^n)$. Se deben implementar dos podas para el árbol de backtracking. Una poda por factibilidad y una poda por optimalidad.
- Algoritmo de Programación Dinámica. Complejidad temporal perteneciente a $\mathcal{O}(n \times V)$.

Solo se permite utilizar $c++$ como lenguaje para resolver el problema. Se pueden utilizar otros lenguajes para presentar resultados.

La entrada y salida de los programas **deberá hacerse por medio de la entrada y salida estándar del sistema**. No se deben considerar los tiempos de lectura/escritura al medir los tiempos de ejecución de los programas implementados.

Deberá entregarse un informe impreso **con a lo sumo 10 paginas** que desarrolle los puntos mencionados.

1. El problema

2. Técnica de Fuerza Bruta

Esta forma de resolver el problema consiste en definir un dominio de posibles soluciones del problema que consideremos y 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:

$$2 \cdot 2 \cdot 2 \cdots 2 = 2^n$$

Por la propia definición, es trivial que si llegará a existir un conjunto R contenido en S y que la sumatoria de sus elementos sea V lo vamos a poder encontrar. Además, almacenando como nueva solución sí y solo sí la encontrada es mejor que la anterior o si no existía,se encontraría en particular la que tenga menos elementos.

2.1. Complejidad

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

3. Backtracking

En esta técnica también es necesario definir el espectro de posibles soluciones del problema. Vamos a considerar también que éstas son el conjunto de partes de S. 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. A esto se lo llama podas y existen de dos tipos: por optimalidad y factibilidad. Dado que en nuestro problema vamos a operar con los elementos de todos los posibles subconjuntos de S, es razonable imaginar que muchas operaciones se realizarán múltiples veces.

Para formar el subconjunto solución se recorrerán los n elementos de S. Sea un subconjunto vacío R, se fijará en la primera posición cada uno de los valores de S y para los siguientes posiciones de R, n, se aplicará recursivamente nuestro algoritmo con

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

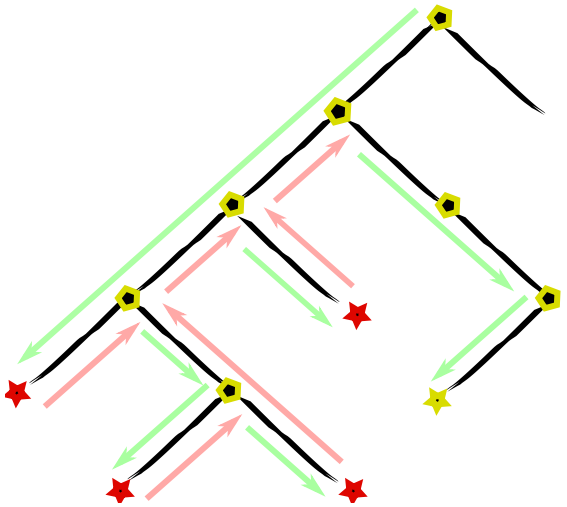
Nuestras podas estarán determinadas por lo siguiente:

3.1. Podas

- Por optimalidad: Si en el proceso de computo de una posible solución, se llegará 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. A nivel algorítmico significará que esa a partir de esa posición no se buscará las posibles combinaciones con los elementos restantes de S'.

- 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, entonces de no buscarán los conjuntos que surjan de combinar los elementos restantes de S con el conjunto actual.

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.



[1]

En las podas de factibilidad, buscamos caracterizar cómo es un subconjunto que no puede ser solución y dejar de computar todos los demás 'subconjuntos padre', es decir, los que contengan a éste. Por otro lado, para una cota optimalidad se identificará las situaciones en las que seguir computando un subconjunto no tenga sentido debido a que en el mejor caso, es decir, si llegara a ser solución, no sería mejor que la que ya encontramos.

3.2. Complejidad

Respecto a la complejidad algorítmica y dado que la establecemos en función de tamaño de la entrada, |S| y V, y para el peor caso, también tendrá complejidad asintótica perteneciente a O(n $\cdot$ 2 ^{n}).
El peor caso sería un valor objetivo mayor que la sumatoria de todos los elementos de S. En ese caso nunca se utilizarán ninguna de las dos podas y se llegarán a formar las 2^n combinaciones de conjuntos, con sus respectivos cómputos de sumas interiores.

Analicemos algunos puntos de diferencia y similitud entre los algoritmos

- En fuerza bruta la complejidad es la teórica en el mejor y peor caso, es decir, $\in \theta$(n $\cdot$ 2 ^{n}) porque siempre se analizará todas las posibles soluciones del dominio definido inicialmente.
- En backtracking la cota superior $\in$ O(2^n $\cdot$ n), pero su cota inferior no es la misma. Por ejemplo, si todos los elementos de S superan a V, entonces la cantidad de pasos realizada sería O(|S|), dado que al poner el primer elemento entraría la poda de factibilidad y se

[1] Imagen tomada del blog de Steve Pemberton

descartarían todos los subconjuntos con el e_1 dentro y se agregaría el siguiente. Como ninguno los elementos de S puede estar en el conjunto solución, esto se repetirá hasta que no queden elementos por agregar, iterando sólo |S| veces.

4. Programación Dinámica

Esta técnica se basa en utilizar la metodología de *divide and conquer* [2] 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 [3]
- Los subproblemas compartan soluciones[4]

AGREGAR EXPLICACION DEL ANALISIS PARA LLEGA Se define PD una función que recibe un conjunto S, de elementos enteros mayores a 0 y un valor V entero mayor a 0, y devuelve el mínimo cardinal del subconjunto s perteneciente a S que todos los elementos suman V; si no existe se devuelve infinito.

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

[2] Metodología que busca simplificar un problema grande o complicado en problemas más chicos y simples de una forma recursiva

[3] Este concepto suele denominarse como: *principio del óptimo*

[4] Se denomina a esto subproblemas superpuestos y un ejemplo muy conocido es la sucesión de Fibonacci