DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Informe de Trabajo Práctico 1

**Subset Sum Problem**

...

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)
     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 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 subsetSumPDTD(S, V)
    matriz ← matriz de |S| filas y V + 1 columnas        ▷ 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
```

```
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  then
    end if
    if  then
    end if
    if  then
    end if
end procedure
```

### 1.4. Programación Dinámica Bottom Up

Explicación de algo bottom up

## 2. Experimentos

Acá van los experimentos

## 3. Conclusiones

Acá van las conclusiones (si las hay)

## 4. Código

Acá va el code