Práctica 1: Complejidad Computacional

Erick Jesús Ríos González

September 19, 2024

1 Problema del Producto de Subconjuntos

1.1 Pseodocódigo

Algorithm 1 SubsetProductProblem

```
function GENERATE_SUBSET(A: List of Integers) → List of Integers
    Initialize empty list subset
    for each element elem in A do
        Generate randomly 0 or 1
        if random value is 1 then
            Add elem to subset
        end if
    end for
    return subset
end function

function VERIFY_SUBSET(subset: List of Integers, t: Integer) → Boolean
    Initialize product = 1
    for each element elem in subset do
        product ← product × elem
        if product > t then
            return No
        end if
    end for
    return Yes
end function

function SUBSET_PRODUCT_DECISION(A: List of Integers, t: Integer) →
(Boolean, List of Integers)
    subset ← GENERATE_SUBSET(A)
    is_valid ← VERIFY_SUBSET(subset, t)
    return (is_valid, subset)
end function
```

1.2 Correctitud

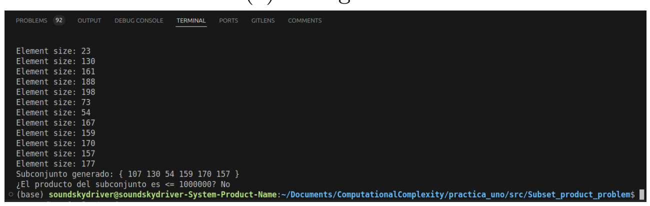
La función `generate_subset` recorre todos los elementos del conjunto A y, para cada elemento, decide aleatoriamente (con probabilidad 50%) si incluirlo en el subconjunto. Como cada elemento es independiente y el proceso es no determinista, cualquier subconjunto posible de A puede ser generado. Esto garantiza que el subconjunto generado es un subconjunto válido de A, ya que solo puede contener elementos de A y no repite elementos. La función `verify_subset` recibe un subconjunto y calcula el producto de sus elementos. A medida que se va calculando el producto, se verifica si este excede el valor t. Si en algún momento el producto excede t, se retorna "No" (False). Si el producto es menor o igual a t para todos los elementos, se retorna "Sí" (True). Esto garantiza que la verificación se realiza correctamente, ya que la multiplicación de elementos y la comparación con t se hace en cada paso, deteniéndose en el momento en que el producto excede t.

1.3 Análisis de Complejidad

La complejidad del algoritmo es polinomial porque la función generar el subconjunto de forma aleatoria es una primitiva, es $O(n)$. Pues no estamos generando todos los subconjuntos posibles, sino que estamos generando un subconjunto arbitrario. La función de verificar el producto de los elementos del subconjunto es $O(n)$ también, puesto que recorremos el subconjunto y multiplicamos los elementos, si el producto excede el valor t, entonces retornamos No, en caso contrario, retornamos Sí. Finalmente la implementación es correcta porque el algoritmo genera un subconjunto de forma aleatoria y verifica si el producto de los elementos del subconjunto es menor o igual a t en tiempo polinomial.

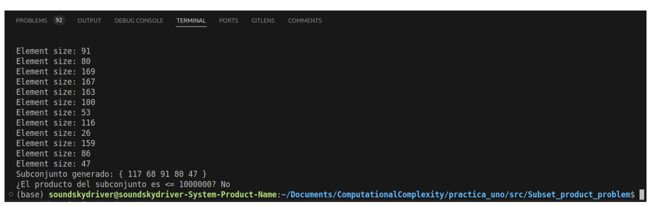
1.4 Ejecución del Algoritmo

(a) Subfigura 1

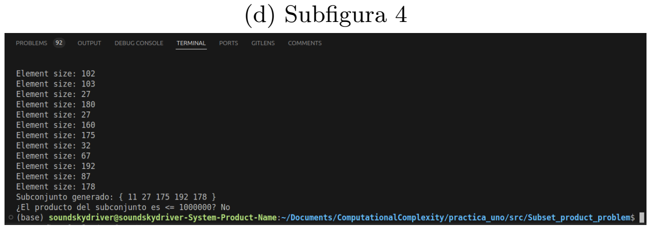


(b) Subfigura 2

(c) Subfigura 3



(d) Subfigura 4



(e) Subfigura 5

Figure 1: Cinco ejecuciones del algoritmo desde terminal

2 Problema de la Mochila

3 Pseodocódigo

Algorithm 2 Knapsack Problem Solver

```
function GENERATE_SUBSET(Item: List of [item_weight, value]) → List of Integers
    Initialize empty list subset
    for each element elem in Item do
        Generate randomly 0 or 1
        if random value is 1 then
            Add elem to subset
        end if
    end for
    return subset
end function
function VERIFY_SUBSET(subset: List of Integers, max_capacity: Integer) → Boolean, Integer
    Initialize total_value ← 0
    for each element elem in subset do
        total_weight ← total_weight + elem.weight
        total_value ← total_value + elem.value
        if total_weight >= max_capacity then
            return (Yes, total_value)
        else
            return (No,0)
        end if-else
    end for
end function
function KNAPSACK_SOLVER(Item: List of [item_weight, value], max_capacity: Integer) → (Boolean, List of Integers)
    subset ← GENERATE_SUBSET(Item)
    (is_valid, total_value) ← VERIFY_SUBSET(subset, max_capacity)
    return (is_valid, total_value)
end function
```

3.1 Correctitud

La función `generate_subset` recorre todos los elementos del conjunto *Item* y, para cada elemento, decide aleatoriamente (con probabilidad 50%) si incluirlo en el subconjunto. Como cada elemento es independiente y el proceso es no determinista, cualquier subconjunto posible de *Item* puede ser generado. Esto garantiza que el subconjunto generado es un subconjunto válido de *Item*, ya que solo puede contener elementos de *Item* y no repite elementos. La función `verify_subset` recibe un subconjunto y calcula el peso total de sus elementos, así como la ganancia total. A medida que se va calculando el peso total, se verifica si este excede el valor *max_capacity*. Si en algún momento el producto excede *max_capacity*, se retorna "No" (False). Si el producto es menor o igual a $\max_c apacity$ para todos los elementos, se retorna "Sí" (True). Esto garantiza que la verificación se realiza correctamente, ya que la suma de los pesos y la comparación con *max_capacity* se hace en cada paso, deteniéndose en el momento en que el *total_weight* excede *max_capacity*.

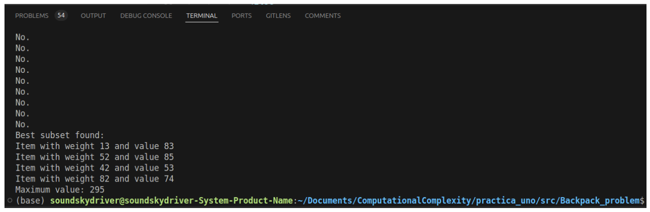
Nota: Para la implementación el programa nos pide encontrar el subconjunto que maximiza la ganancia, por lo que creo un ciclo for con un número de iteraciones definidas desde el inicio, en este caso 1000 iteraciones. Esto sigue siendo correcto, pues durante un numero finito de iteraciones, se generan subconjuntos aleatorios y se verifica si el peso total excede la capacidad máxima. De esta forma obtenemos un maximo local que maximiza la ganancia.

3.2 Análisis de Complejidad

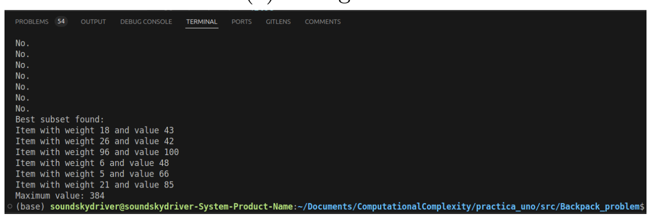
La complejidad del algoritmo es polinomial porque la función generar el subconjunto de forma aleatoria es una primitiva, es $O(n)$. Pues no estamos generando todos los subconjuntos posibles, sino que estamos generando un subconjunto arbitrario. La función de verificar el producto de los elementos del subconjunto es $O(n)$ también, puesto que recorremos el subconjunto y sumamos los pesos de los elementos, si el peso total excede el valor *max_capacity*, entonces retornamos No, en caso contrario, retornamos Sí. Finalmente la implementación es correcta porque el algoritmo genera un subconjunto de forma aleatoria y verifica si el producto de los elementos del subconjunto es menor o igual a *max_capacity* en tiempo polinomial.

Nota: La implementación utilizando un número finito de iteraciones sigue siendo polinomial, pero el algoritmo no garantiza que el subconjunto generado sea el óptimo, sino que es un subconjunto que maximiza la ganancia en un número finito de iteraciones.

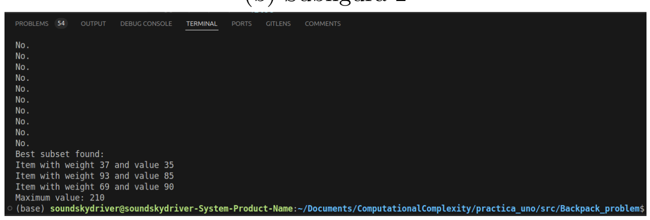
3.3 Ejecución del Algoritmo



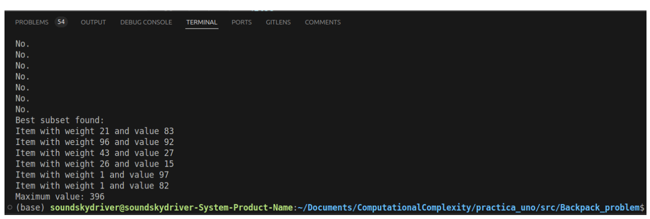
(a) Subfigura 1



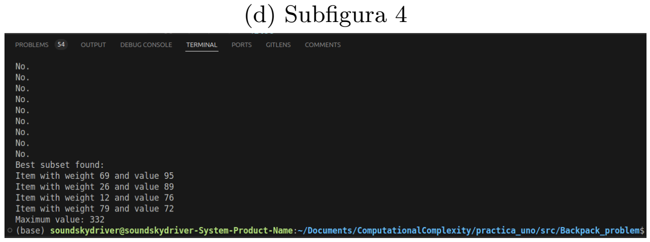
(b) Subfigura 2



(c) Subfigura 3



(d) Subfigura 4



(e) Subfigura 5

Figure 2: Cinco ejecuciones del algoritmo desde terminal