



Instituto Politécnico Nacional

Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2021-1, 3CV1, Práctica 5,
07-12-2020

Práctica 5: Algoritmos Greedy.

De la Cruz Sierra Diana Paola

Raya Chávez Samuel Antonio

dianapaodcs@gmail.com, samiraya1323@gmail.com

Resumen Explicación de los algoritmos voraces, su función y algunos ejemplos en práctica como el problema de cambio de moneda, mochila fraccionaria y el algoritmo de Kruskal.

Palabras clave. Greedy, algoritmos voraces, optimización Kruskal.

1. Introducción

Los algoritmos greedy, glotones, codiciosos o simplemente voraces son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras.

Se usan generalmente para resolver problemas de optimización. Construyendo una solución del problema de optimización paso a paso a través de una secuencia de elecciones que son:

- Factibles
- Localmente óptimas
- Irrevocables

Normalmente se aplican a problemas donde no únicamente queremos encontrar la solución, si no que además queremos encontrar la mejor solución.

El propósito de un algoritmo voraz es encontrar una asociación de valores a todas las variables tal que el valor de la función objetivo sea óptimo. Para lograr que dicha solución sea óptima se debe seguir el siguiente proceso secuencial:

Cada paso toma una decisión (decide qué valor del dominio le ha de asignar a la variable actual) aplicando siempre el mismo criterio (función de selección). La decisión es localmente óptima, lo que quiere decir que si se elige cualquier otro valor de los disponibles para esa variable no haría que la función objetivo sea mejor, posteriormente se comprueba si la puede incorporar al conjunto de decisiones que hasta el momento se ha tomado, es decir, comprueba que la nueva decisión junto con todas las anteriormente tomadas no violan las restricciones y es así como se consigue una nueva secuencia de decisiones factible.

En el siguiente paso el algoritmo voraz se encuentra con un problema idéntico, pero estrictamente menor, al que tenía en el paso anterior y vuelve a aplicar la misma función de selección para tomar la siguiente decisión. Esta es, por tanto, una técnica descendente. Nunca se vuelve a reconsiderar ninguna de las decisiones tomadas.

Una vez que a una variable se le ha asignado un valor localmente óptimo y que hace que la secuencia de decisiones sea factible, nunca más se va a intentar asignar un nuevo valor a esa misma variable.

La forma general de la estructura de una función voraz puede verse en el algoritmo 1

Hay que tomar en cuenta también que si la selección es óptima en cada paso no significa que genere una solución óptima global.

Algorithm 1 Pseudocódigo: FuncionVoraz

```
1: procedure FUNCIONVORAZ(Conjunto c)
2:    $S \leftarrow \emptyset$ 
3:   while  $\neg esSolucion$  do
4:      $x = seleccionarMejorCandidato(c)$ 
5:      $c = c - \{x\}$ 
6:     if  $esFactible(S \cup \{x\})$  then
7:        $S = S \cup \{x\}$ 
8:     end if
9:     if  $esSolucion(S)$  then
10:      return S
11:    end if
12:  end while
13: end procedure
```

2. Conceptos básicos

Problema del cambio de monedas: El problema del cambio de monedas dice que dado un conjunto C con M tipos de monedas y con un número inagotable de ejemplares de cada tipo, se tiene que formar la cantidad M empleando el mínimo número de ejemplares de monedas. (Soriano, T, 2008).

Problema de la mochila fraccionaria: En éste problema se dispone de una colección de n objetos donde cada uno de estos dispone de un peso y un valor; es decir, para el objeto i , con $1 \leq i \leq n$, su valor es v_i y su peso es p_i . La mochila es capaz de soportar, como máximo, el peso $PMAX$. Se tiene que determinar que objetos hay que colocar en la mochila de modo que el valor total que transporta sea máximo pero sin que se sobrepase el peso máximo $PMAX$. (Soriano, T, 2008).

Algoritmo de Kruskal: Es un algoritmo que resuelve el problema de encontrar un árbol mínimo de cubrimiento, en cada iteración seleccionará el menor de los arcos todavía no considerados, si el arco seleccionado junto con la solución parcial es viable entonces se incluye en la solución parcial, en caso contrario se descarta. (Fillottrani, P, 2017).

Solución óptima: La decisión tomada es la mejor, es decir, ningún otro valor de los disponibles para esa variable lograría que la función objetivo tuviera una solución mejor. (Soriano, T, 2008).

2.1. Problema del cambio de monedas.

En el algoritmos 2 podemos observar el algoritmo Greedy de devolver-Cambio.

Algorithm 2 Pseudocódigo: devolverCambio

```
1: procedure DEVOLVERCAMBIO(cantidad, monedas)
2:    $n \leftarrow \text{len}(\text{monedas})$ 
3:    $\text{cambio} = \text{Arreglo de tamaño } n \text{ igual a } 0$ 
4:   for  $i \leftarrow 0, i \leq n - 1$  do
5:     while  $\text{monedas}[i] \leq \text{cantidad}$  do
6:        $\text{cantidad} \leftarrow \text{cantidad} - \text{monedas}[i]$ 
7:        $\text{cambio}[i] \leftarrow \text{cambio}[i] + 1$ 
8:     end while
9:   end for
10:   $\text{return cambio}$ 
11: end procedure
```

Una vez visto el algoritmo 2 vamos a mostrar una prueba de escritorio a continuación:

■ Prueba 1 del algoritmo 2:

Supongamos que se tiene el arreglo de monedas dado por:

$\text{monedas} = [10, 9, 1]$ y $\text{cantidad} = 18$

en este caso $n=3$ y $\text{cambio} = [0, 0, 0]$

- para $i = 0$
Se cumple la condición $10 \leq 18$, entonces entra al while 1 vez y ahora:
 $\text{cantidad}=8$ y $\text{cambio}[0] = 1$
- para $i = 1$
Aquí el while No entra debido a que $9 > 8$
- para $i = 2$
Se cumple la condición $1 \leq 8$, entonces entra al while 8 veces teniendo los siguientes cambios desde 1 a 8 respectivamente:
 $\text{cantidad} = 7$, $\text{cambio}[2] = 1$

cantidad = 6, cambio[2] = 2
 cantidad = 5, cambio[2] = 3
 cantidad = 4, cambio[2] = 4
 cantidad = 3, cambio[2] = 5
 cantidad = 2, cambio[2] = 6
 cantidad = 1, cambio[2] = 7
 cantidad = 0, cambio[2] = 8

Regresando la función el siguiente resultado: cambio=[1,0,8]

2.2. Problema de la mochila fraccionaria.

Algorithm 3 Pseudocódigo: mochila fraccionaria

```

1: procedure MOCHILAFRACCIONARIA( $b[0,\dots,n-1]$ ,  $w[0,\dots,n-1]$ ,  $P$ )
2:    $valorPorPeso \leftarrow \text{sortSelection}(b,w)$ 
3:    $n \leftarrow \text{len}(b) = \text{len}(w)$ 
4:    $s \leftarrow 0, i \leftarrow 0$ 
5:    $x =$  Arreglo de tamaño  $n$  igual a cero
6:   while  $s < P$  &  $i < n$  do
7:      $k \leftarrow \text{selecciona}(b, w)$ 
8:      $i++$ 
9:     if  $s + w[k] \leq P$  then
10:       $x[k] \leftarrow 1$ 
11:       $s += w[k]$ 
12:     else
13:       $x[k] \leftarrow (P - s)/w[k]$ 
14:       $s = P$ 
15:     end if
16:   end while
17:   return  $x$ 
18: end procedure

```

En este caso del algoritmo 3 la función *sortSelection* es una función que ordena de manera descendente un arreglo, de acuerdo al que tenga más valor por unidad de peso.

la función *selecciona* va a regresar una posición del objeto el cual ira ocupando en ese instante de acuerdo al arreglo ordenado descendientemente en la función *sortSelection()*. Esta función utiliza el método de ordenamiento

quickSort. El cual en la mayoría de los casos su orden de complejidad es $n \log(n)$, sin embargo en un peor caso, este tendrá complejidad cuadrática.

Ahora vamos a mostrar una prueba de escritorio para este algoritmo:

■ Prueba uno del algoritmo 3:

Supongamos que el valor máximo de la mochila es $p = 100$ y que los valores y pesos son los siguientes:

$b[20,30,65,40,60]$, $w=[10,20,30,40,50]$

entonces al llamar a la función `sortSelection()` se genera el siguiente arreglo $[(2.16,2), (2,0), (1.5,1), (1.2,4), (1,3)]$ en donde el lado izquierdo corresponde al valor por unidad de peso y el lado derecho la posición original en los arreglos b y w , luego:

$n=5$, $s=0$, $i=0$

- $(0 < 100 \ \& \ 0 < 7)$
 $k=2$, $i=1$
 $(30 \leq 100)$, entonces
 $x[2]=1$, $s=30$
- $(30 < 100 \ \& \ 1 < 7)$
 $k=0$, $i=2$
 $(40 \leq 100)$, entonces
 $x[0]=1$, $s=40$
- $(40 < 100 \ \& \ 1 < 7)$
 $k=1$, $i=3$
 $(60 \leq 100)$, entonces
 $x[1]=1$, $s=60$
- $(60 < 100 \ \& \ 1 < 7)$
 $k=4$, $i=4$
 $(120 > 100)$, entonces
 $x[4]=0.8$, $s=100$
 Finalmente regresa $[1, 1, 1, 0, 0.8]$

2.3. Algoritmo de Kruskal

Algorithm 4 Pseudocódigo: Algoritmo de kruskal

```
1: procedure KRUSKAL( $G$ : Grafos,  $n$ : #nodos)
2:   Construcción de fila de prioridad  $cp$  con los arcos del grafo  $G$ 
3:   Inicializar componente conexo
4:    $F = 0$ 
5:   while  $\text{!vacía}(cp) \ \& \ |F| < n - 1$  do
6:      $arista \leftarrow \text{obtenerMin}(cp)$ 
7:      $\text{borrarMin}(cp)$ 
8:      $u \leftarrow \text{componente}(\text{de}(arista))$ 
9:      $v \leftarrow \text{componente}(\text{a}(arista))$ 
10:    if  $\text{numeroComponente}(u) \neq \text{numeroComponente}(v)$  then
11:      añadir arista a  $F$ 
12:       $\text{unirComponentes}(u, v)$ 
13:    end if
14:  end while
15: end procedure
```

Del algoritmo 4, la función *vacía()* corroborará que la fila dada no esta vacía.

obtenerMin se encargará de obtener el arco de menor longitud disponible de la fila *cp*.

borrarMin es la función que borrara el arco que se acaba de obtener de *obtenerMin*.

componente() se encargará de obtener el componente pedido.

numeroComponente() va a obtener el número del correspondiente componente pedido.

Y *unirComponentes()* unirá los nodos que se unan del arco seleccionado creando así una nueva rama.

Haciendo una prueba de escritorio para ver mejor la funcionalidad del algoritmo se obtiene que:

- Prueba 1 del algoritmo 4

Sea el siguiente grafo:

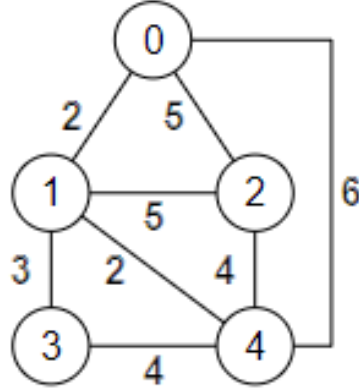


Figura 1: Grafo ejemplo (Wextensible, 2020)

Se tiene entonces que $\#nodos = 5$, y G es el grafo.

Ahora se construye la fila de prioridad cp con los datos del arco G , y se inicializa el componente conexo y $F = 0$

Mientras la fila cp no este vacia y el conjunto F sea menor a $5-1$, es decir, 4, entonces hará lo siguiente hasta que una de las 2 condiciones anteriores no se cumplan.

De *obtenerMin(cp)* se tendrá que $arista=2$, luego *borrarMin(cp)* borrará esta arista de la fila cp .

Con *componente* se obtendrá que $u=0$ y $v=1$. Como $u(0)$ es diferente de $v(1)$, entonces a F se agregará la arista 2 y se unirán los nodos 0 y 1. Ahora $F=1$.

Aún se cumplen las condiciones iniciales, por lo que del siguiente ciclo *obtenerMin(cp)* se tiene que $arista=2$ y *borrarMin(cp)* borrará esta arista de la fila cp .

componente() nos dará que $u=1$ y $v=4$. Como $u(1)$ es diferente de $v(4)$ entonces a F se le agrega la arista 2 y se unirán los nodos 1 y 4. Ahora $F=2$.

Aún se cumplen las condiciones iniciales, por lo que del siguiente ciclo *obtenerMin(cp)* se tiene que $arista=3$ y *borrarMin(cp)* borrará esta arista de la fila cp .

componente() nos dará que $u=1$ y $v=3$. Como $u(1)$ es diferente de $v(3)$ entonces a F se le agrega la arista 3 y se unirán los nodos 1 y 3. Ahora $F=3$.

Aún se cumplen las condiciones iniciales, por lo que del siguiente ciclo *obtenerMin(cp)* se tiene que $arista=4$ y *borrarMin(cp)* borrará esta arista

de la fila cp.

componente() nos dará que $u=2$ y $v=4$. Como $u(3)$ es diferente de $v(4)$ entonces a F se le agrega la arista 4 y se unirán los nodos 2 y 4. Ahora $F=4$.

Como $F=4$ y $n-1=4$, entonces se termina el ciclo y se llegó a una solución con el árbol de cubrimiento mínimo expresado con líneas de color de la siguiente figura:

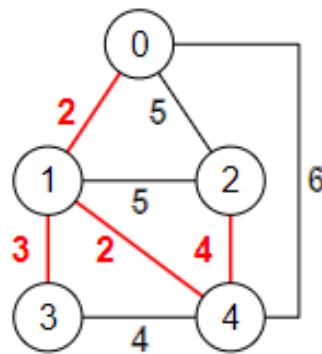


Figura 2: Árbol de cubrimiento mínimo del grafo de la figura 1, (Wextensible, 2020).

3. Experimentación y resultados.

3.1. Problema del cambio de monedas.

En la imagen 3 pueden observarse 2 ejemplos de la entrada y salida del algoritmo 2 ya programado.

```
C:\Windows\System32\cmd.exe
C:\Users\Diana Paola\Documents\Sto Semestre\Algoritmos\Practica5>g++ -o cambio cambioMonedas.cpp
C:\Users\Diana Paola\Documents\Sto Semestre\Algoritmos\Practica5>cambio
Ingrese la cantidad de monedas: 3
Ingrese las n monedas en orden decreciente: 10 9 1
Ingrese cantidad de cambio: 18
Respuesta: 1 0 8
Cambio devuelto en monedas: $10x1 + $1x8 +
C:\Users\Diana Paola\Documents\Sto Semestre\Algoritmos\Practica5>cambio
Ingrese la cantidad de monedas: 4
Ingrese las n monedas en orden decreciente: 20 5 2 1
Ingrese cantidad de cambio: 8
Respuesta: 0 1 1 1
Cambio devuelto en monedas: $5x1 + $2x1 + $1x1 +
C:\Users\Diana Paola\Documents\Sto Semestre\Algoritmos\Practica5>
```

Figura 3: Ejemplos de entrada y salida del programa utilizando el algoritmo 2.

La penúltima línea de cada ejemplo muestra el número de veces que se usa cierta nominación de moneda incluyendo 0 veces. Y la última línea lo desglosa sumando dichas nominaciones ocupadas por la cantidad de cada nominación ocupada.

Ahora para el algoritmo 2 vamos a poner 5 nominaciones fijas de monedas(10, 9, 5, 2 y 1) y variar el valor de *cantidad*, contando de manera muy aproximada el número de pasos que le lleva al algoritmo regresar el valor correspondiente. Esta tabulación se puede ver en la tabla 1

Monedas	Cantidad	No. de pasos
[10,9,5,2,1]	18	15
[10,9,5,2,1]	21	13
[10,9,5,2,1]	30	13
[10,9,5,2,1]	38	19
[10,9,5,2,1]	45	17
[10,9,5,2,1]	62	21
[10,9,5,2,1]	68	25
[10,9,5,2,1]	77	25
[10,9,5,2,1]	83	27
[10,9,5,2,1]	90	25
[10,9,5,2,1]	99	27
[10,9,5,2,1]	133	37
[10,9,5,2,1]	152	39
[10,9,5,2,1]	169	41

Cuadro 1: Mismas nominaciones y distinto cambio

Una vez tabulado varios valores para las mismas nominaciones vamos a graficar los puntos de la tabla 1. Dichos puntos pueden observarse en la imagen 4

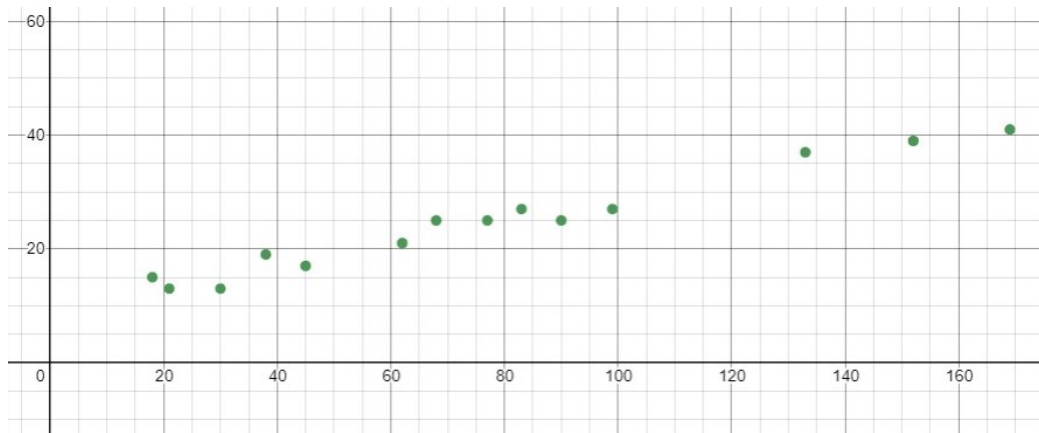


Figura 4: Puntos de la tabla 1.

Ahora vamos a invertir los roles, es decir, vamos a dejar fijo el valor de *cantidad* y variar tanto el tamaño de nominaciones de monedas como su valor. Esta tabulación puede encontrarse en la tabla 2.

n	Monedas	Cantidad	No. de pasos
5	[10,9,5,2,1]	18	15
4	[12,6,3,1]	18	10
6	[13,11,8,7,2,1]	18	16
2	[6,2]	18	10
3	[10,9,1]	18	23
7	[7,6,5,4,3,2,1]	18	15
1	[2]	18	21
10	[10,9,8,7,6,5,4,3,2,1]	18	16
15	[15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]	18	21
9	[15,12,9,7,6,5,4,2,1]	18	17
11	[20,17,16,15,13,11,10,9,7,5,1]	18	17
17	[25,23,22,21,20,19,15,13,12,11,10,8,6,5,3,2,1]	18	23

Cuadro 2: mismo cambio y distintas nominaciones

Ahora vamos a graficar los puntos de la tabla 2 en donde varia n . Dichos puntos pueden observarse en la imagen 5.

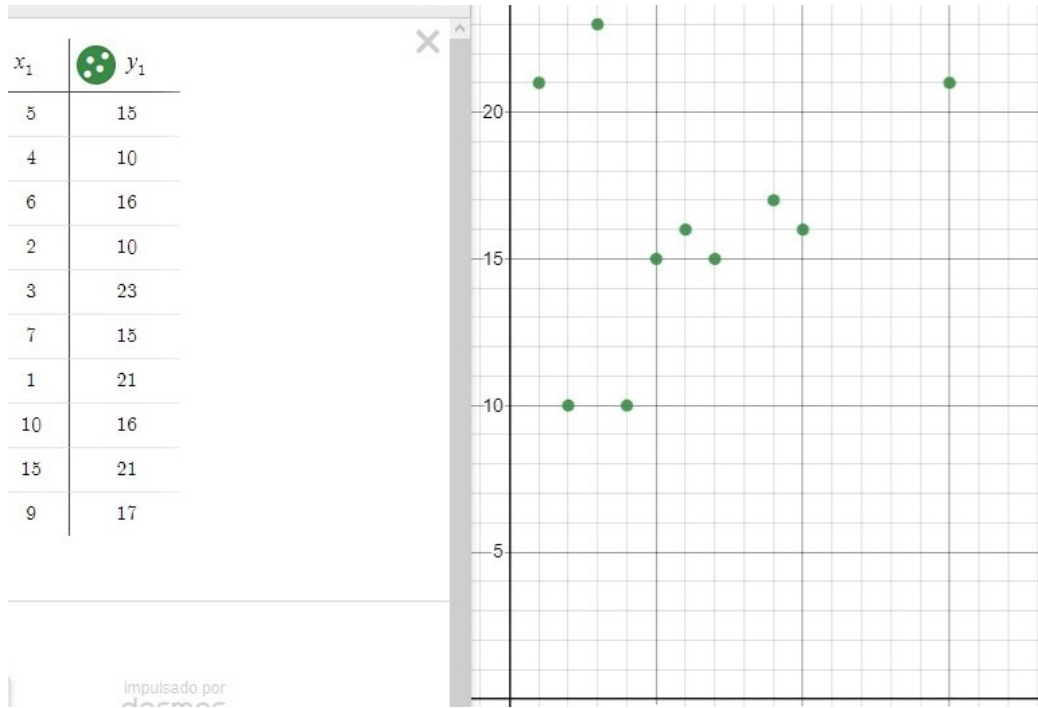


Figura 5: Puntos de la tabla 2.

Notemos las siguientes observaciones del algoritmo 2.

primero que nada hay que notar que para una cantidad c y un arreglo de monedas de tamaño n , el número de pasos no siempre será el mismo, esto porque a pesar de ser el tamaño el mismo para las monedas, su nominación puede cambiar haciendo que el while se ejecute más o menos veces dependiendo del valor. Por ejemplo que ¿Qué pasa si $n = 1$ y la nominación es 1, entonces el while se ejecutará el tamaño de la *cantidad* = c porque irá incrementando de uno en uno.

También recordando que el arreglo monedas está ordenado de manera decreciente. Si la primer nominación (distinta de 1) posicionada en el índice $i = 0$ del arreglo es múltiplo de *cantidad*, el while se ejecutará únicamente en el primer índice del bucle *for*, porque para las otras nominaciones siguientes la condición $monedas[i] \leq cantidad$ ya no se cumplirá y únicamente recorrerá el ciclo *for* completo sin entrar al while, lo que hace que los pasos de ejecución sean menores.

Ahora hablando de las complejidades: En este caso estamos suponiendo que en el algoritmo 2 como parámetro en las monedas ya nos dan un arreglo ordenado. Sin embargo, en caso de no ser así podríamos hacer uso de algún algoritmo de ordenamiento como *quickSort* por citar un ejemplo, el cual tiene complejidad $\Theta(n \log n)$, aún que eso podría hacerse incluso antes de mandar

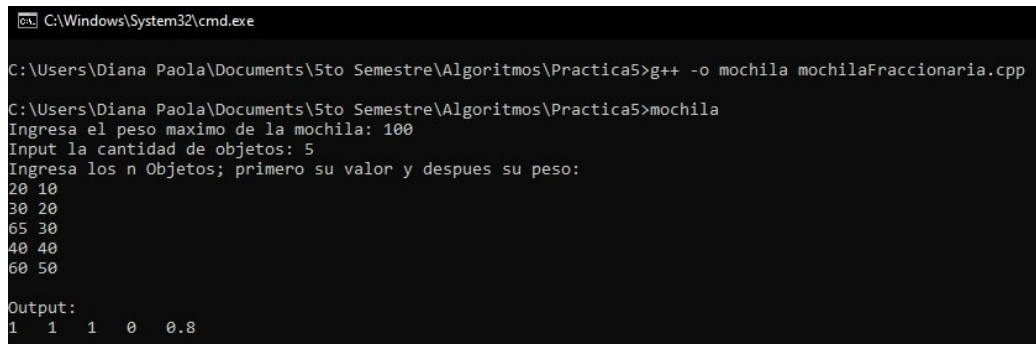
a llamar a la función `devolverCambio`.

Entonces dichas todas las observaciones anteriores y teniendo en cuenta que como parámetro el arreglo *monedas* ya se pasa ordenado de manera decreciente. El algoritmo tiene orden de complejidad $\theta(n)$.

Para concluir y viendo la prueba de escritorio 1 del algoritmo 2 también puede observarse que no siempre el algoritmo es óptimo tal como es en este caso.

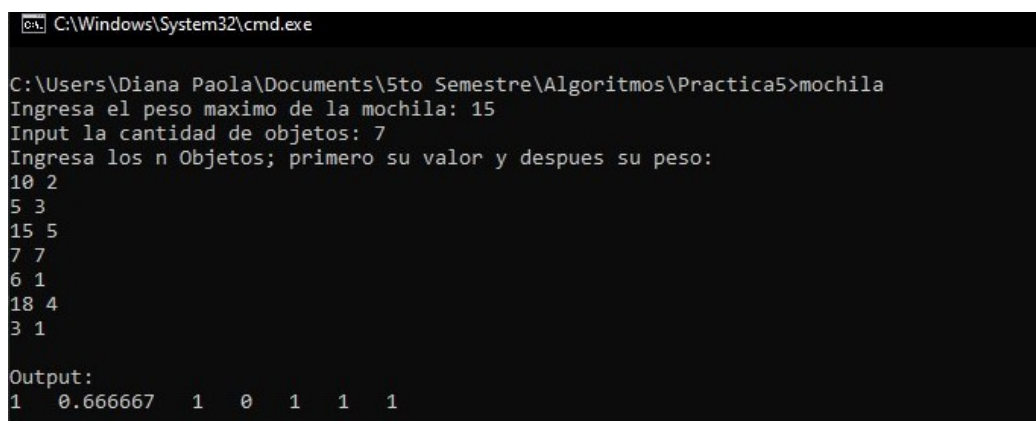
3.2. Problema de la mochila fraccionaria

Ahora vamos a analizar el algoritmo 3 comenzando con 2 ejemplos de entrada y salida, con el algoritmo 3 ya programado. Dichos ejemplos se muestran en las imágenes 6 y 7



```
C:\Windows\System32\cmd.exe
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica5>g++ -o mochila mochilaFraccionaria.cpp
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica5>mochila
Ingresa el peso maximo de la mochila: 100
Input la cantidad de objetos: 5
Ingresa los n Objetos; primero su valor y despues su peso:
20 10
30 20
65 30
40 40
60 50
Output:
1 1 1 0 0.8
```

Figura 6: Ejemplo 1 del programa para el algoritmo 3.



```
C:\Windows\System32\cmd.exe
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica5>mochila
Ingresa el peso maximo de la mochila: 15
Input la cantidad de objetos: 7
Ingresa los n Objetos; primero su valor y despues su peso:
10 2
5 3
15 5
7 7
6 1
18 4
3 1
Output:
1 0.666667 1 0 1 1 1
```

Figura 7: Ejemplo 2 del programa para el algoritmo 3..

El objetivo del algoritmo 3 es maximizar el beneficio de los objetos transportados. Max

$$\sum_{1 \leq i \leq n} X_i b_i$$

sujeo a

$$\sum_{1 \leq i \leq n} X_i W_i \leq P$$

con $0 \leq X_i \leq 1$, $b_i > 0$, $P_i > 0$

Ahora vamos a analizar su complejidad:

Haciendo un análisis A priori, en la tabla 3 se hace calcula la complejidad por medio de bloques

MochilaFraccionaria(b[0,...,n], w[0,...,n], P valorPorPeso = sortSelection(b,w) n = len(b) = len(w) s=0, i=0 x = Arreglo de tamaño n igual a cero while (s<P & & i<n)
--

Cuadro 3: Análisis A priori del algoritmo 3

Una observación es que en un peor caso el bucle *while* recorre n veces, pero de igual manera es más grande $Ologn$.

Asi, analizando lo anterior se tiene que la complejidad de mochilaFraccionaria $\in \theta(nlogn)$

Ahora vamos a hacer un análisis A posteriori:

En la tabla 4 se pueden observar varios valores en pesos, en beneficios y en capacidad de mochila.

n	Beneficio	Peso	Capacidad	No pasos
3	[30,50,15]	[29,48,12]	20	30
4	[10, 20, 60, 23]	[5, 3, 35 11]	50	40
5	[20,30,65,40,60]	[10,20,30,40,50]	100	63
6	[100,25,45,11,60,30]	[45,13,20,5,22,15]	99	61
7	[10,5,15,7,6,18,3]	[5,1,3,7,4,9,1]	12	63
8	[60,30,25,78,64,2,36,21]	[10,20,15,46,30,1,10,18]	120	100
9	[10,20,30,40,50,60,70,80,90]	[5,10,15,20,25,30,35,40,45]	135	154
9	[10,20,30,40,50,60,70,80,90]	[5,10,15,20,25,30,35,40,45]	50	142
10	[5,20,10,25,80,100,65,72,16,7]	[3,6,5,13,45,50,37,43,8,2]	150	128
11	[20,10,60,70,90,80,15,22,12,7,6]	[8,5,60,45,35,10,8,10,6,1,5]	200	147
6	[50,30,45,15,20,19]	[30,20,10,15,13,9]	70	59

Cuadro 4: Análisis A posteriori del algoritmo 3

En la imagen 8 se pueden observar los puntos de la tabla 4

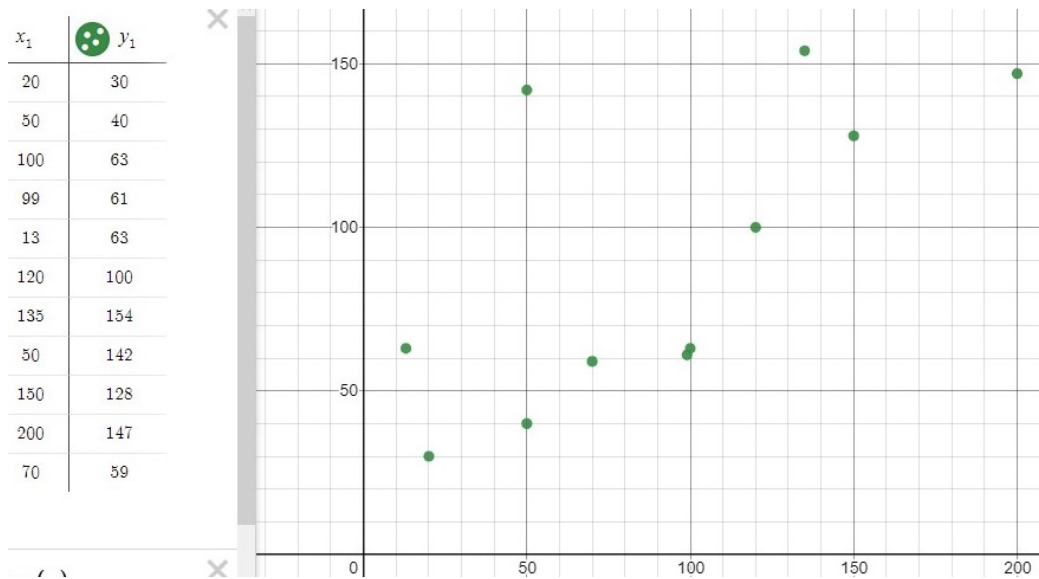


Figura 8: Puntos graficados de la tabla 4.

Y en la imagen 9 se propone como ecuación a $6n \log n$, la cual funge como cota superior de los puntos graficados en la imagen 8.

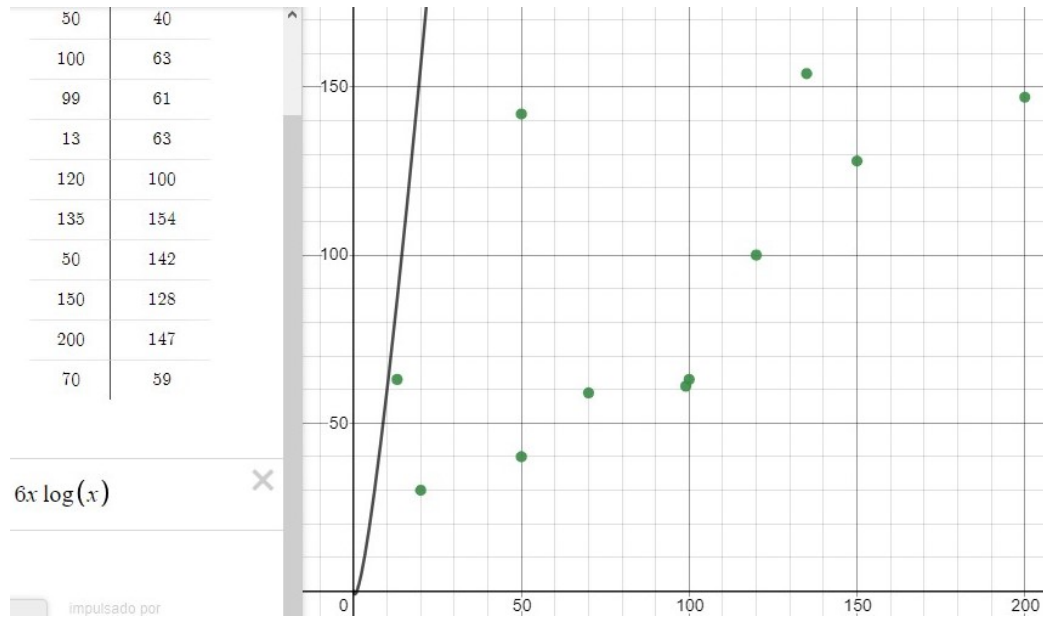


Figura 9: Puntos de la tabla 4 y ecuación propuesta del algoritmo 3.

Es evidente que para una misma capacidad C no siempre sera la misma cantidad de pasos, ya que también depende de los valores utilizados en el arreglo de beneficios b y el arreglo de pesos w , sin embargo viendo el análisis A priori y este A posteriori, se puede concluir que mochilaFraccionaria $\in \theta(n \log n)$

3.3. Algoritmo de Kruskal

```
C:\Users\samir\Desktop>Kruskal
Ingrese el numero de nodos a ocupar
5
Ingrese el numero de arcos a ocupar
8
Ingrese el Nodo de origen: 4
Ingrese el Nodo de destino: 0
Ingrese el arco: 6
Ingrese el Nodo de origen: 0
Ingrese el Nodo de destino: 1
Ingrese el arco: 2
Ingrese el Nodo de origen: 1
Ingrese el Nodo de destino: 2
Ingrese el arco: 5
Ingrese el Nodo de origen: 0
Ingrese el Nodo de destino: 2
Ingrese el arco: 5
Ingrese el Nodo de origen: 1
Ingrese el Nodo de destino: 3
Ingrese el arco: 3
Ingrese el Nodo de origen: 3
Ingrese el Nodo de destino: 4
Ingrese el arco: 4
Ingrese el Nodo de origen: 1
Ingrese el Nodo de destino: 4
Ingrese el arco: 2
Ingrese el Nodo de origen: 2
Ingrese el Nodo de destino: 4
Ingrese el arco: 4
Grafo minimo
Nodo Origen: 0; nodo destino: 1; longitud: 2
Nodo Origen: 1; nodo destino: 4; longitud: 2
Nodo Origen: 1; nodo destino: 3; longitud: 3
Nodo Origen: 2; nodo destino: 4; longitud: 4
```

Figura 10: Ejemplo del algoritmo 4 en funcionamiento.

En la figura 10 se muestra el funcionamiento del código tomando en cuenta el grafo de la figura 1, y como se puede observar, la solución obtenida es la misma que con la figura 2.

Analizando primero a priori se tiene que:

a el número de aristas del grafo y n el número de nodos, el algoritmo de Kruskal muestra una complejidad $O(a \log a)$ o, equivalentemente, $O(a \log n)$. Los tiempos de ejecución son equivalentes porque:

a es a lo sumo n^2 y $\log(n^2) = 2\log(n) \in O(\log n)$. Ignorando los nodos aislados, los cuales forman su propia componente del árbol de expansión mínimo, $n \leq 2a$, así que $\log(n) \in O(\log a)$.

Se puede conseguir esta complejidad de la siguiente manera: primero se ordenan las aristas por su peso usando una ordenación por comparación (comparison sort) con una complejidad del orden de $\Theta(a \log a)$; esto permite que el paso "eliminar una arista de peso mínimo de C " se ejecute en tiempo constante. Lo siguiente es usar una estructura de datos para conjuntos disjuntos para controlar qué nodos están en qué componentes. Es necesario hacer orden de $\Theta(a)$ operaciones ya que por cada arista hay dos operaciones de búsqueda y posiblemente una unión de conjuntos y rangos. Por tanto, la complejidad total es del orden de $\Theta(a \log a) = \Theta(a \log n)$; (LinkFang, 2020).

Ahora haciendo un análisis A posteriori:

A continuación se mostrará una tabla con diferentes datos usados para el código.

n	#Nodos	#Arcos	No pasos
1	5	8	33
2	5	7	33
3	6	10	45
4	8	11	55
5	9	12	60
6	4	5	24
7	8	8	48

Cuadro 5: Análisis A posteriori del algoritmo 4

Tomando en cuenta los números de nodos y el número de pasos ejecutados se tiene algo como la siguiente gráfica, donde se ven los puntos de la tabla 5 y una gráfica propuesta.

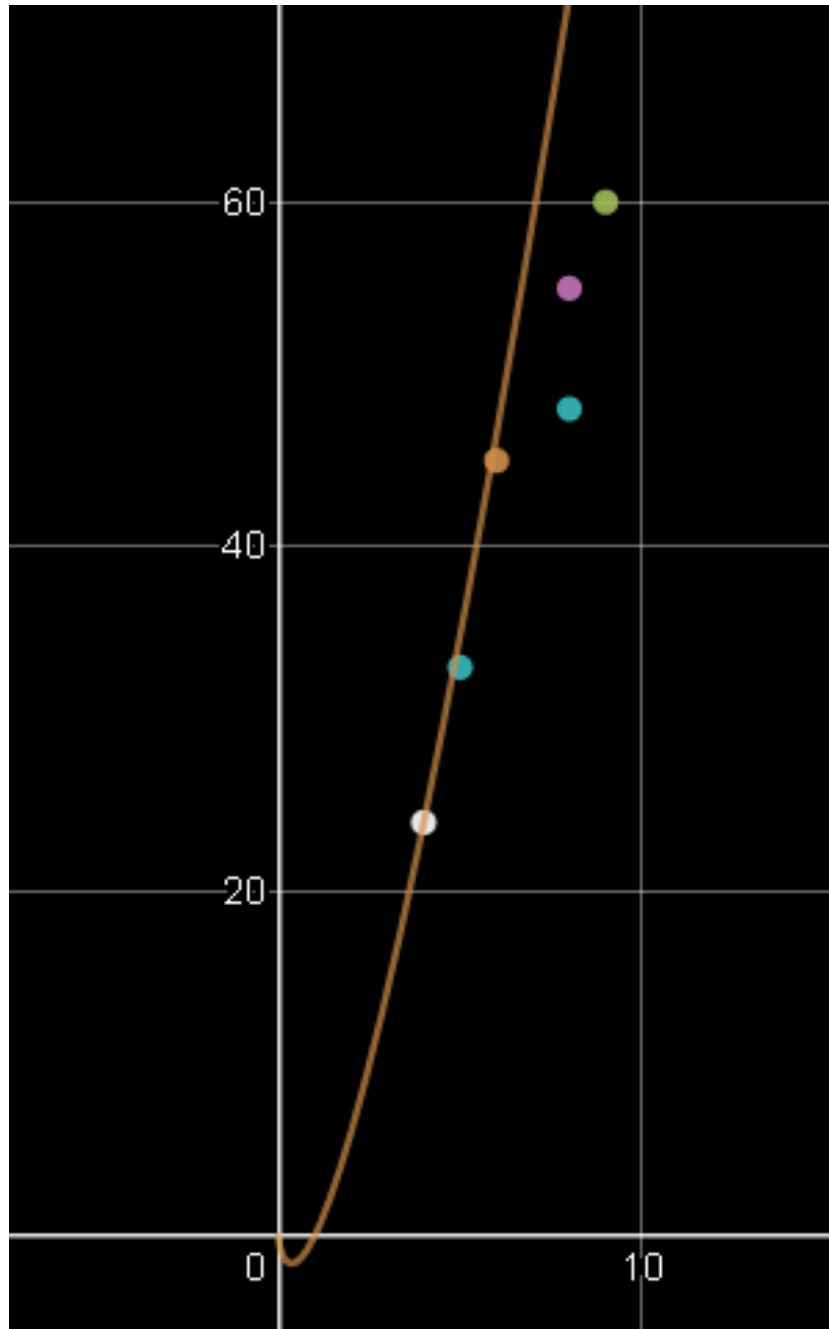


Figura 11: Gráfica de la tabla 5 nodos vs. #pasos

Aquí la gráfica propuesta es $y = 2n \log_2 n$ donde n es el número de nodos del árbol. Por lo tanto, el orden de complejidad es $\Theta(n \log(n))$.

Ahora tomando en cuenta los números de aristas y el número de pasos ejecutados se tiene la siguiente gráfica donde se ven los puntos de la tabla 5

y una gráfica propuesta.

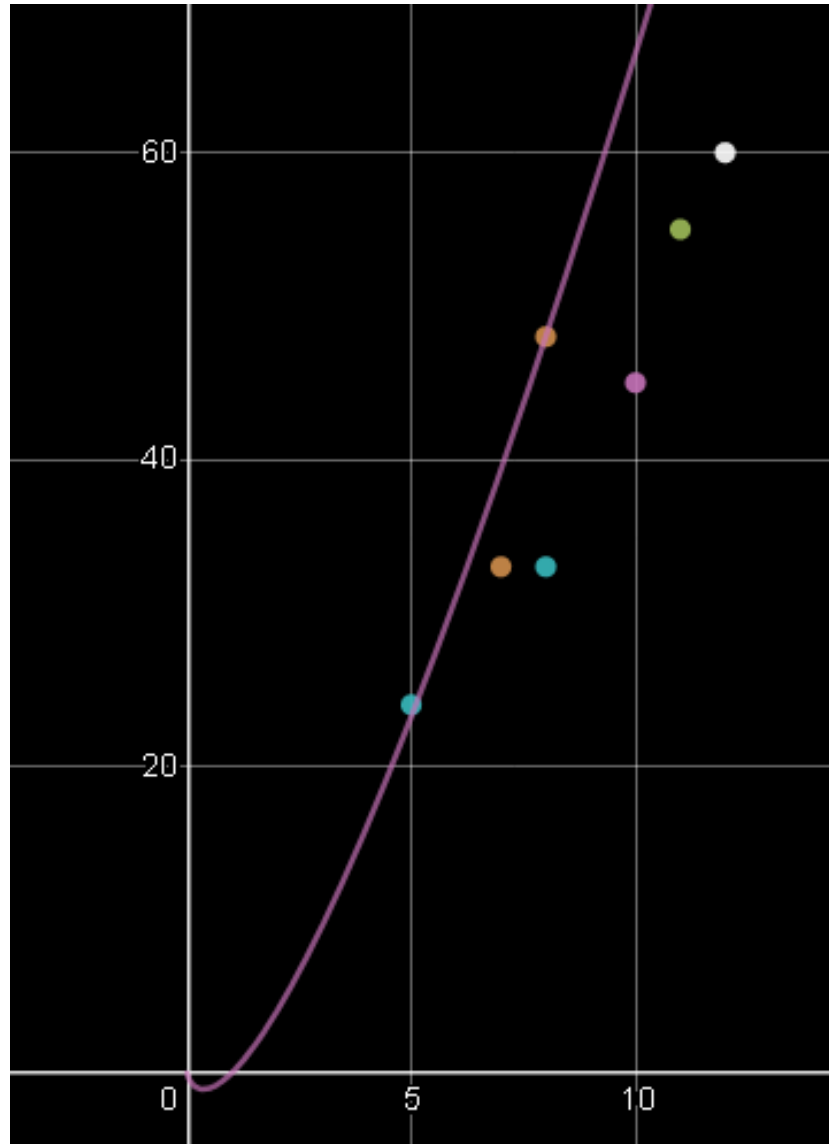


Figura 12: Gráfica de la tabla 5 aristas vs. #pasos

Aquí la gráfica propuesta es $y = 3a \log_2 a$ donde a es el número de aristas del árbol. Por lo tanto, el orden de complejidad es $\Theta(\log(a))$.

Entonces, teniendo para los nodos un orden de complejidad $O(n \log(n))$ y para las aristas un orden de complejidad $\Theta(\log(a))$, se concluye que Kruskal $\in \Theta(n \log(n))$.

Con lo que en cualquier caso que se tomen los nodos o las aristas

4. Conclusiones

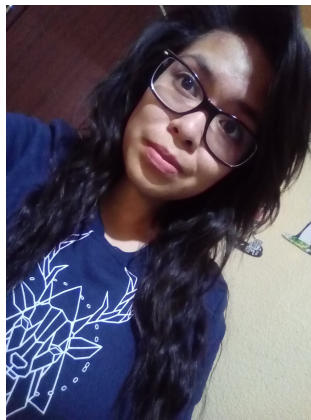
- Conclusiones generales:

En cuanto al algoritmo de moneda se menciona que se supone que la función supone que ya incorpora el arreglo ordenado, sin embargo puede hacerse uso de un algoritmo de ordenación tal y como se utilizó en el algoritmo mochila fraccionaria, aún que solo son pequeños detalles a considerar.

Para el algoritmo de Kruskal, se me hizo de una forma interesante que con árboles medianamente grande el programa empezó a tardarse más hasta no imprimir ningún resultado ya que el tiempo de espera se había agotado.

- Diana Paola De la Cruz Sierra:

Los algoritmos de tipo greedy normalmente son fáciles de implementar y siempre intentarán dar soluciones óptimas, sin embargo no siempre se garantiza que dará tal resultado óptimo, por ejemplo fue el ejemplo que se desarrollo anteriormente de cambio de moneda.



- Raya Chávez Samuel Antonio:

Los algoritmos voraces se me hacen los algoritmos más eficientes ya que estos solo esperan los datos de una manera ya prevista para así poder dar resultados rápidos y evitarse la parte de validar el código.

El único inconveniente que veo es que si después de ingresar los primeros datos hay datos que deberían dar otra solución, el algoritmo tomará los primeros datos como referencia dando así un resultado no deseado.



5. Fuentes de información

- Soriano, T. (2008). Algoritmos voraces. [PDF]. Disponible en: <https://www.cs.upc.edu/~mabad/ADA/curso0708/GREEDY.pdf>
- Rodriguez, E. (2018). Algoritmos voraces (Greedy). [PDF]. Disponible en: <https://www.tamps.cinvestav.mx/~ertello/algorithms/sesion15.pdf>
- Fillottrani, P. (2017). Algoritmos y complejidad (Algoritmos Greedy). [PDF]. Disponible en: <http://www.cs.uns.edu.ar/~prf/teaching/AyC17/downloads/Teoria1x1.pdf>
- Fillottrani, P. (2017). Algoritmos y Complejidad. diciembre 11, 2020, de Depto. Ciencias e Ingeniería de la Computación Universidad Nacional del Sur Sitio web: <http://www.cs.uns.edu.ar/~prf/teaching/AyC17/downloads/Teoria1x1.pdf>
- Wextensible. (2020). Algoritmos de Kruskal y Prim. diciembre 11, 2020, de Wextensible Sitio web: <https://www.wextensible.com/temas/voraces/kruskal-prim.html>
- LinkFang. (2020). Algoritmo de Kruskal. diciembre 11, 2020, de LinkFang Sitio web: https://es.linkfang.org/wiki/Algoritmo_de_Kruskal_Complejidad_del_algoritmo