Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2021-1, 3CV1, Práctica6 ,
13-12-2020

Práctica 6: Algoritmos Greedy. Programación dinámica.

De la Cruz Sierra Diana Paola
Raya Chávez Samuel Antonio
dianapaodcs@gmail.com, samiraya1323@gmail.com

Resumen. Se manejará el concepto de programación dinámica aplicada a algunos problemas como la sucesión de Fibonacci y el problema de la mochila entera 0-1.
Palabras clave. Programación dinámica, Fibonacci, mochila entera.

1. Introducción

La programación dinámica tiene como finalidad encontrar una solución de un problema de optimización en forma secuencial. La programación dinámica no es un algoritmo de solución única, sino más bien un método para resolver un problema grande y único solventando una secuencia de problemas más pequeños, sin importar el número de ellos.

La solución de problemas mediante la programación dinámica se basa en el llamado principio de optimalidad, el cual, fue enunciado por Bellman (1957) y dice que: "En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima". Sin embargo, se debe evaluar en cada problema presentado que este principio se esté cumpliendo y analizar cómo abordar cada uno de los problemas de acuerdo a sus propias características.

Algo a tomar en cuenta es que si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de la lista de soluciones calculadas y reutilizarla, pero si estamos seguro de que no la volveremos a utilizar se puede descartar para ahorrar espacio.

La programación dinámica puede tomar uno de dos enfoques llamados top-down y bottom-up

- Top-down: el problema se divide en subproblemas, y estos se resuelven recordando las soluciones por si fueran necesarias nuevamente.

- Bottom-up: los problemas que pudieran ser necesarios se resuelven anticipadamente y después se usan para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones.

viendo lo anterior puede notarse que debido a que bottom-up no utiliza llamadas a funciones es más rápido.

Además puede no solo se quiera conocer el óptimo, si no también como llegar a ese valor.Por esto, en cada paso se toman decisiones sobre como dividir al problema en subproblemas cuyas soluciones darán la solución óptima y guardar en cada estado toda información necesaria sobre la decisión óptima.

2. Conceptos básicos

- **Sucesión de Fibonacci:** Se trata de una secuencia infinita de números naturales; a partir del 0 y el 1, se van sumando a pares, de manera que cada número es igual a la suma de sus dos anteriores, de manera que:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55… (Bravo, R., 2018).

- **Problema de la mochila entera:** El problema dice que hay una persona que tiene una mochila con una cierta capacidad de peso y tiene que elegir que elementos se incluirá en la mochila. Cada uno de los elementos tiene un peso y aporta un beneficio. El objetivo de la persona será elegir los elementos que le permitan maximizar el beneficio sin excederse de la capacidad permitida. (López, R., 2018).

- **Memoización:** Ésta es usada para el enfoque top-down y su función es almacenar resultados de funciones, evitando repetir cálculos para entradas previamente procesadas. (Garza, M., 2019)

- **Traslape en algoritmos Greedy:** Los mismos subproblemas pueden ser llegar a crear subproblemas más grandes, de manera que un algoritmo recursivo resolvería los mismos subproblemas una y otra vez haciendo que el algoritmo sea ineficiente. (Garza, M., 2019)

2.1. Sucesión de Fibonacci

Vamos a empezar recordando como es que se calcula el algoritmo para Fibonacci de manera recursiva, sin utilizar programación dinámica. Este algoritmo se puede observar en el algoritmo 1.

Algorithm 1 Pseudocódigo: Fibonacci

```
1: procedure FIBONACCI(INT N)
2:     if n <= 1 then
3:         return n
4:     else
5:         return Fibonacci(n − 1) + Fibonacci(n − 2)
6:     end if
7: end procedure
```

El algoritmo para calcular el primer enfoque $Top-down$ para fibonacci se muestra en el algoritmo 3. Pero antes debemos de aplicarlo debemos realizar los pasos mostrados en el algoritmo 2:

Algorithm 2 Pseudocódigo: PasosPrevios

```
1: procedure PasosPrevios
2:     fibo[TAM]
3:     for i ← 0, i < TAM do
4:         fibo[i] ← −1
5:     end for
6:     fibo[0] ← 0
7:     fibo[1] ← 1
8: end procedure
```

Algorithm 3 Pseudocódigo: FibonacciTopDown

```
1: procedure FibonacciTopDown(int n, Fibo: tabla)
2:     if fibo[n]! = −1 then
3:         returnfibo[n]
4:     end if
5:     fibo[n] ← FibonacciTopDown(n − 2) + FibonacciTopDown(n − 1)
6:     return fibo[n]
7: end procedure
```

Vamos a mostrar un ejemplo haciendo uno del algoritmo 3:

- Vamos a calcular fibo[5]:
  - Haciendo los pasos del algoritmo 2:
    fibo = [0,1,-1,-1,-1]
  - se llama la función FibonacciTopDown(4,tab)
    (fibo[4] = -1), entonces
    fibo[4] = FibonacciTopDown(2) + FibonacciTopDown(3)
    - FibonacciTopDown(2,fibo)
      fibo[2] = -1, entonces
      fibo[2] = FibonacciTopDown(0) + FibonacciTopDown(1)
      - FibonacciTopDown(0)
        fibo[0] != -1, entonces
        return fibo[0] = 0
      - FibonacciTopDown(1)
        fibo[1] != -1, entonces
        return fibo[1] = 1

fibo[2] = 0+1 = 1
return fibo[2] = 1

- FibonacciTopDown(3,fibo)
  fibo[3] = -1, entonces
  fibo[3] = FibonacciTopDown(1) + FibonacciTopDown(2)
  - ◇ FibonacciTopDown(1,fibo)
    fibo[1] != -1, entonces
    return fibo[1] = 1
  - ◇ FibonacciTopDown(2,fibo)
    (fibo[2]=1 != -1), entonces
    return fibo[2] = 1

  fibo[3] = 1+1 = 2
  return fibo[3] = 2

fibo[4] = 2+1 = 3
return fibo[4] = 3

Es asi como Output: 3

Algorithm 4 Pseudocódigo: FibonacciBottomUp

```
1: procedure FibonacciTopDown(int n, Fibo: tabla)
2:     if n ≤ 1 then
3:         return 1
4:     else
5:         fibo[0] ← 0
6:         fibo[1] ← 1
7:         for i ← 2, i ≤ n do
8:             fibo[i] ← fibo[i − 1] + fibo[i − 2]
9:         end for
10:        return fibo[n]
11:    end if
12: end procedure
```

Ahora vamos un ejemplo haciendo uso del algoritmo 4:

- Vamos a calcular para FibonacciBottomUp(6)
  Comenzamos con fibo[0,0,0,0,0,0]
  - FibonacciBottomUp(5, fibo)
    fibo[0]=0

fibo[1]=1

FOR

- i=2
  fibo[2] = fibo[1] + fibo[0] = 1+0 = 1
- i=3
  fibo[3] = fibo[2] + fibo[1] = 1+1 = 2
- i=4
  fibo[4] = fibo[3] + fibo[2] = 2+1 = 3
- i=5
  fibo[5] = fibo[4] + fibo[3] = 3+2 = 5

return 5

Así Output: 5

2.2. Problema de la mochila entera 0-1

A diferencia del problema de mochila fraccionaria, el problema de la mochila entera se trata de llenar la mochila en cuestión obteniendo el máximo beneficio sin fraccionar los objetos.

A continuación se mostrará el pseudocódigo para la resolución del problema de la mochila entera.

Algorithm 5 Pseudocódigo: Mochila entera

```
1: procedure MOCHILA ENTERA(w, b : [1, ..., n], P)
2:     for c ← 0, c ≤ P do
3:         g[0, c] ← 0
4:     end for
5:     for j ← 1, j ≤ n do
6:         g[j, 0] ← 0
7:     end for
8:     for j ← 1, j ≤ n do
9:         for c ← 1, c ≤ P do
10:            if c < w[j] then
11:                g[j, c] ← g[j − 1, c]
12:            else
13:                if g[j − 1, c] ≥ g[j − 1, c − w[j]] + b[j] then
14:                    g[j, c] ← g[j − 1, c]
15:                else
16:                    g[j, c] ← g[j − 1, c − w[j]] + b[j]
17:                end if
18:            end if
19:        end for
20:    end for
21: end procedure
```

Además del algoritmo 5, se tiene que incluir también la siguiente función que nos ayuda a escoger que objetos deben de incluirse en la mochila.

Algorithm 6 Pseudocódigo: Test

```
1: procedure TEST(j, c)
2:     if c > 0 then
3:         if c < w[j] then
4:             test(j − 1, c)
5:         else
6:             if g[j − 1, c − w[j]] + b[j] > g[j − 1, c] then
7:                 test(j − 1, c − w[j]])
8:                 print("guardar objeto", j)
9:             else
10:                test(j − 1, c)
11:            end if
12:        end if
13:    end if
14: end procedure
```

Haciendo un ejemplo del algoritmo 5

Se tiene una mochila de peso=7, y con los objetos: Objeto 1 con peso=2 y beneficio=20,objeto 2 con peso=3 y beneficio=25, y un objeto 3 con peso=6 y beneficio=60.

Entonces, se inicializa una tabla con valores iguales a 0 desde la posición 0 a la 7 y se crearán los espacios correspondientes para los objetos, teniendo algo como esto:

	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
w_1=2	0							
w_2=3	0							
w_3=6	0							

Cuadro 1: Inicialización de los valores

Ahora se comparan los valores y se decide que valor irá en cada cuadro de la tabla

- Se tiene j=1, c=1

  luego, se compara c<w[j], es decir, 1<w[1]=2

  como se cumple la condición, entonces g[1,1]← g[1-1,1] lo cual es igual a 0

- con j=1 y c=2

  la condición 2<w[1]=2 no se cumple, por lo que de aquí en adelante se empezará a comparar con g[j-1,c]>=g[j-1,c-w[j]]+b[j]

  Entonces, se compara 0=g[0,2]>=g[0,(2-2)]+b[1]=0+20=20

  Como no se cumple la condición, entonces g[1,2] va a ser igual a 20

- con j=1 y c=3

  la condición g[j-1,c]>=g[j-1,c-w[j]]+b[j] será

  0=g[0,3]>=g[0,1]+b[1]=0+20=20

  Como no se cumple la condición, entonces g[1,3] va a ser igual a 20

  y como los demás valores a comparar tendrán la misma estructura, entonces se tendrá el mismo valor de 20 para el resto de la fila. Teniendo lo siguiente:

	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
w_1=2	0	0	20	20	20	20	20	20
w_2=3	0							
w_3=6	0							

Cuadro 2: Primera fila completada

- Ahora con j=2,c=1 se tiene

  Se compara c<w[j], es decir, 1<w[2]=3

  como se cumple la condición, entonces g[2,1]← g[2-1,1], lo cual es igual a 0 y se cumple la condición hasta que c=3

- con j=2 y c=3

  la condición 3<w[2]=3 no se cumple, por lo que de aquí en adelante se empezará a comparar con g[j-1,c]>=g[j-1,c-w[j]]+b[j]

  Entonces, se compara 20=g[1,3]>=g[1,(3-3)]+b[2]=0+25=25

  Como no se cumple la condición, entonces g[2,3] va a ser igual a 25

- con j=2 y c=4

  la condición g[j-1,c]>=g[j-1,c-w[j]]+b[j] será

  20=g[1,4]>=g[1,(4-3)]+b[2]=0+25=25

  Como no se cumple la condición, entonces g[2,4] va a ser igual a 25

- con j=2 y c=5

  la condición g[j-1,c]>=g[j-1,c-w[j]]+b[j] será

  20=g[1,5]>=g[1,(5-3)]+b[2]=20+25=45

  No se cumple la condición, entonces g[2,5] va a ser igual a 45 Y de aquí en adelante se cumplirá esta condición para el resto de la fila. Teniendo lo siguiente:

	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
w_1=2	0	0	20	20	20	20	20	20
w_2=3	0	0	20	25	25	45	45	45
w_3=6	0							

Cuadro 3: Segunda fila completada

- Ahora con j=3,c=1 se tiene

  Se compara c<w[j], es decir, 1<w[3]=6

  como se cumple la condición, entonces g[3,1]← g[3-1,1] lo cual es igual a 0 y se cumple la condición hasta que c=3

- con j=3 y c=3

  la condición 3<w[3]=6 se sigue cumpliendo, entonces g[3,3]← g[3-1,3] lo cual es igual a 25 y se cumple la condición hasta que c=5

- con j=3 y c=5

  la condición 5<w[3]=6 se sigue cumpliendo, entonces g[3,5]← g[3-1,5] lo cual es igual a 45

- con j=3 y c=6

  la condición 6<w[3]=6 no se cumple, por lo que de aquí en adelante se empezará a comparar con g[j-1,c]>=g[j-1,c-w[j]]+b[j]

  Entonces, se compara 45=g[2,6]>=g[(3-1),(6-6)]+b[3]=0+60=60

  Como no se cumple la condición, entonces g[3,6] va a ser igual a 60 y se cumple igual cuando c=7. Por lo que queda una tabla como la siguiente:

	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
w_1=2	0	0	20	20	20	20	20	20
w_2=3	0	0	20	25	25	45	45	45
w_3=6	0	0	20	25	25	45	60	60

Cuadro 4: Tabla completa

Ahora, para mostrar cual será la salida del algoritmo 6, se usará el ejemplo anterior para mostrar esto.

- Una vez que se haya acabado de realizar la tabla, se pasarán los últimos valores de j y c

  Se cumple la primera condición que dice que j>0, luego viene la condición c<w[j], es decir, 7<w[3]=6.

  Como no se cumple la condición anterior, entonces ahora se verifica la condición g[j-1, c-w[j]]+b[j])>g[j-1,c], en este caso da 60=0+60=g[3-1,7-6]+b[3]>g[3-1,7]=45.

  Como se cumple la condición, entonces se imprime "Guardar objeto 3ε se vuelve a llamar a la función test(j-1,c-w[j]).

- Ahora j=2 y c=1

  La condición c<w[j], es decir 1<w[2]=3 se cumple, por lo que se vuelve a llamar a la función test(j-1, c).

- Ahora j=1 y c=-2

  La condición c<w[j], es decir -2<w[1]=2 se cumple, entonces se vuelve a llamar a la función test(j-1, c).

  Ahora j=0

  Como ya no se cumple la condición inicial que j>0, entonces se termina la función entregando únicamente como salida "Guardar objeto 3".

Tomando otro ejemplo para mostrar la salida de *test*(), se tiene la siguiente tabla:

	0	1	2	3
	0	0	0	0
w_1=1	0	20	20	20
w_2=1	0	30	50	50
w_3=4	0	30	50	50
w_4=2	0	30	50	80

Cuadro 5: Segundo ejemplo, la capacidad de la mochila de peso=3

Con los beneficios: objeto 1=20, objeto 2=30, objeto 3=10 y objeto 4=50.

- Entonces se pasan los últimos valores de j y c, en este caso j=4 y c=3.

  Se cumple la primera condición que dice que j>0, luego viene la condición c<w[j], es decir, 3<w[4]=2.

  Como no se cumple la condición anterior, entonces ahora se verifica la condición g[j-1,c-w[j]]+b[j])>g[j-1,c]

  En este caso nos da 80=30+50=g[4-1,3-2]+b[4]>g[4-1,3]=50.

  Como se cumple la condición, entonces se imprime "Guardar objeto 4$\imath$ se vuelve a llamar a la función test(j-1,c-w[j]).

- Ahora j=3 y c=1

  La condición c<w[j], es decir 1<w[3]=4 se cumple, entonces se vuelve a llamar a la función test(j-1,c).

- Ahora j=2 y c=1

  La condición c<w[j], es decir 1<w[2]=1 no se cumple, entonces ahora se verifica la condición g[j-1, c-w[j]]+b[j])>g[j-1,c]

  En este caso nos da 30=0+30=g[2-1,1-1]+b[2]>g[2-1,1]=20.

  Como se cumple la condición, entonces se imprime "Guardar objeto 2$\imath$ se vuelve a llamar a la función test(j-1,c-w[j]).

- Ahora j=1 y c=0

  La condición c<w[j], es decir 0<w[1]=1 se cumple, entonces se vuelve a llamar a la función test(j-1,c).

  Ahora j=0

  Como ya no se cumple la condición inicial que j>0, entonces se termina la función entregando únicamente como salida "Guardar objeto 4", "Guardar objeto 2".

3. Experimentación y resultados

3.1. Sucesión de Fibonacci

En la imagen 1 se observan tres ejemplos de como es la entrada y la salida del algoritmo 3 programado.

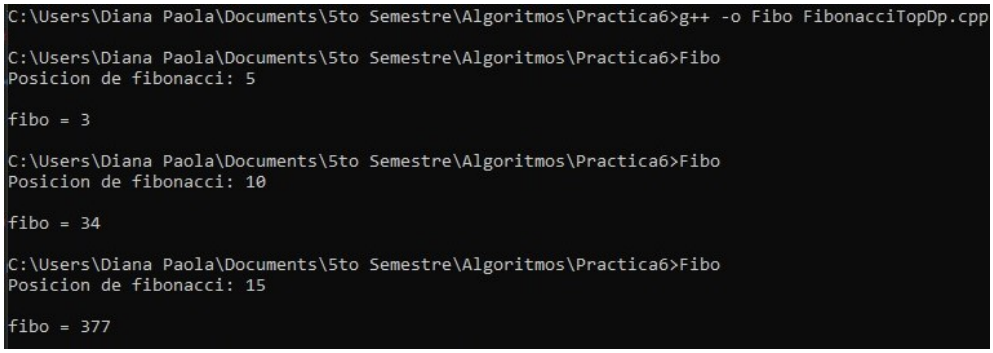


Figura 1: Ejemplo de entrada y salida del algoritmo 3 programado

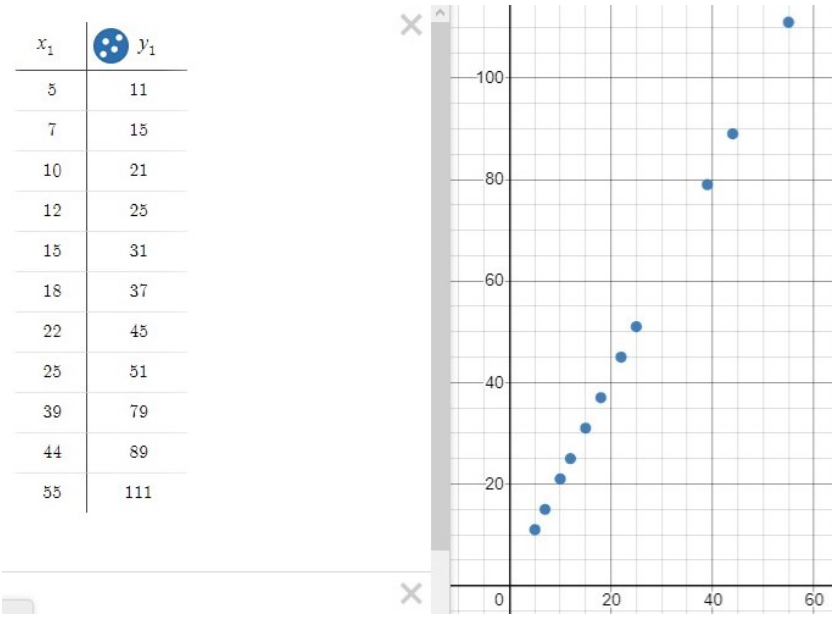
Vamos a hacer un análisis A posteriori del algoritmo 3.

Vamos a hacer un calculo aproximado de cuantos pasos ejecuta el algoritmo 3 para $fibo(n)$, donde n es el número $'n'$ en la sucesión de Fibonacci, dicha aproximación puede encontrarse en la tabla 6. Sin embargo también debemos tener en consideración que para este caso no se contemplaron los pasos para el algoritmo 2 ya que estos son pasos previos y no los estamos considerando tal cual dentro del algoritmo 3.

Fibo(n)	**No. pasos ejecutados**
5	11
7	15
10	21
12	25
15	31
18	37
22	45
25	51
39	79
44	89
55	111

Cuadro 6: Puntos del análisis A posteriori para el algoritmo 3

Para darnos una mejor idea de la distribución, los puntos graficados de la tabla 6 se pueden observar en la imagen 2.



x_1	y_1
5	11
7	15
10	21
12	25
15	31
18	37
22	45
25	51
39	79
44	89
55	111

Figura 2: Puntos del análisis a posteriori del algoritmo 3

Análizando los puntos de la imagen 2 vamos a definir a $3n$ como ecuación propuesta que hará el papel de una cota superior. Esto puede verse de mejor manera en la imagen 3.

7	15
10	21
12	25
15	31
18	37
22	45
25	51
39	79
44	89
55	111

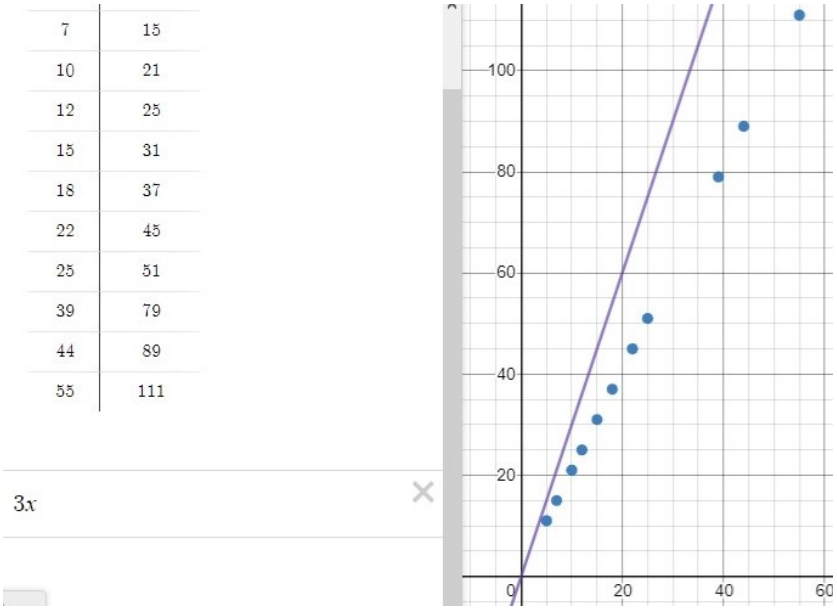


Figura 3: Puntos del análisis a posteriori del algoritmo 3 y gráfica propuesta

Claramente $3n$ está por encima de los puntos en la imagen 2, además, $3n \in \theta(n)$, por lo que podemos decir que el algoritmo 3, FibonacciTopDown $\in \theta(n)$.

Si quisiéramos hacer un análisis A priori del algoritmo 3 pareciera que habría que dar solución a la recurrencia $f(n) - f(n-1) - f(n-2) = 0$, resolviendo esto por el método de recurrencias de segundo orden con coeficientes constantes se tiene que el orden de complejidad es exponencial, pero este caso aplica más bien para el algoritmo 1. Sin embargo en el algoritmo 3 no es asi, ya que al pasar por referencia la tabla $fibo$ se calcula una sola vez fibo(i) para cada i desde i=0 hasta i=n que es el resultado esperado. Es dedir, a pesar de existir traslapes, solo será necesario resolver cada subproblema una sola vez. de manera que fibo(n) = fibo(n-1) + fibo(n-2) ahora es líneal. Es decir $FibonacciTopDown \in \theta(n)$

Ahora en la imagen 4 se muestran tres ejemplos de la entrada y salida pero ahora del algoritmo 4 programado.

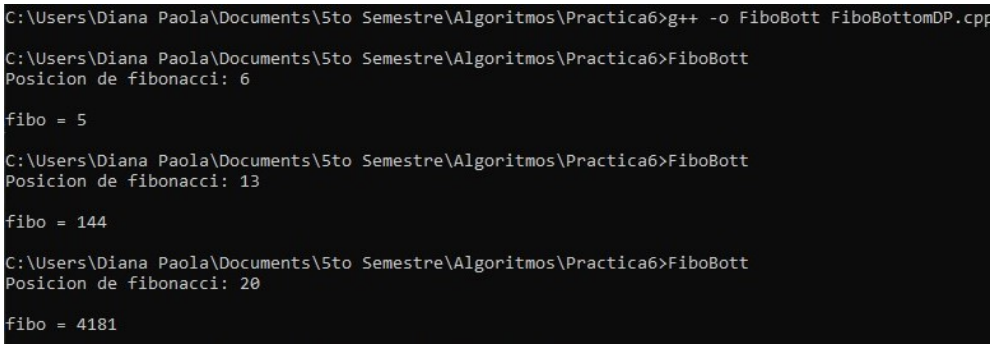


Figura 4: Ejemplo de entrada y salida del algoritmo 4 programado

De igual manera realizando un análisis A posteriori del algoritmo 4, vamos a contar un aproximado el número de pasos que le toma al algoritmo realizar los cálculos para fibo(n), donde $'n'$ es el número en la sucesión fibonacci que queremos encontrar. Esta relación se encuentra en la tabla 7

Fibo(n)	**No. pasos ejecutados**
5	9
9	13
13	17
15	19
18	22
24	28
29	33
33	37
39	43
45	49
50	54

Cuadro 7: Puntos del análisis A posteriori para el algoritmo 4

Y los puntos de la tabla 7 pueden observarse de manera gráfica en la imagen 5.

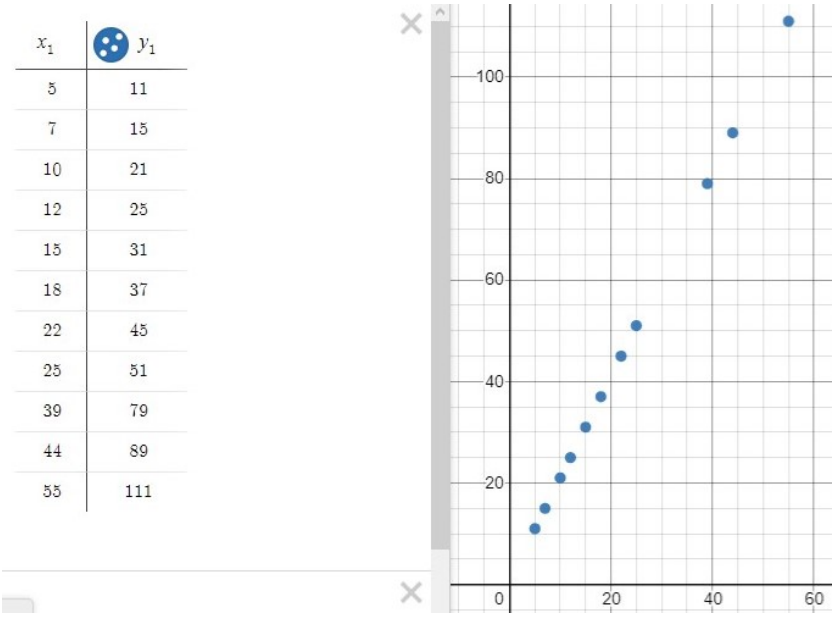


Figura 5: Puntos del análisis a posteriori del algoritmo 4

Vamos a utilizar la función $2n$ para acotar superiormente a los puntos mostrados en la imagen 5. Esto se puede ver de mejor manera en la imagen 6

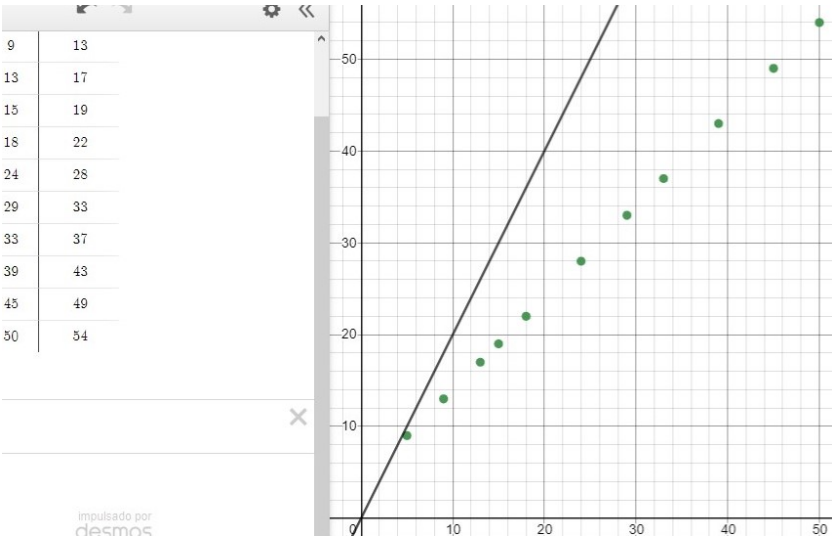


Figura 6: Puntos del análisis a posteriori del algoritmo 4 y gráfica propuesta

Viendo que $2n \in \theta(n)$ y que cubre a todos los puntos por encima, podemos decir en cuanto al algoritmo 4 que $FibonacciBottomUp \in \theta(n)$.

Pero ahora bien, vamos a desarrollar el análisis A priori del algoritmo 4. Calculando la complejidad por medio de bloques podemos ayudarnos de la tabla 8 para observar mejor este análisis.

Sentencia	**Complejidad**		
int FiboBottomUp(integer n, tabla: fibo)			
if n $\leq$ 1		$\theta(1)$	θ(n)
return 1	$\theta(1)$		
else			
fibo[0] $\leftarrow$ 0	$\theta(1)$	θ(n)	
fibo[1] $\leftarrow$ 1			
for i$\leftarrow$2 to i $\leq$ n do	θ(n)		
fibo[i] $\leftarrow$ fibo[i-1]+fibo[i+2]			
return fibo[n]	$\theta(1)$	$\theta(1)$	

Cuadro 8: Análisis A priori del algoritmo 4

Una vez hecho el análisis A priori y A posteriori concluimos para el algoritmo 4: $FibonacciBottomUp \in \theta(n)$, es decir que tiene orden de complejidad lineal.

Finalmente vamos a realizar un análisis A posteriori del algoritmo 1. Haciendo un aproximado de los pasos que ejecuta este algoritmo se observan en la tabla 9.

FibonacciRecursivo(n)	**No. pasos ejecutados**
5	16
6	26
7	42
8	68
9	110
10	178
11	288
12	466

Cuadro 9: Análisis A posteriori del algoritmo 1

Los puntos de la tabla 9 y la gráfica 2^x se encuentran en la imagen 7.

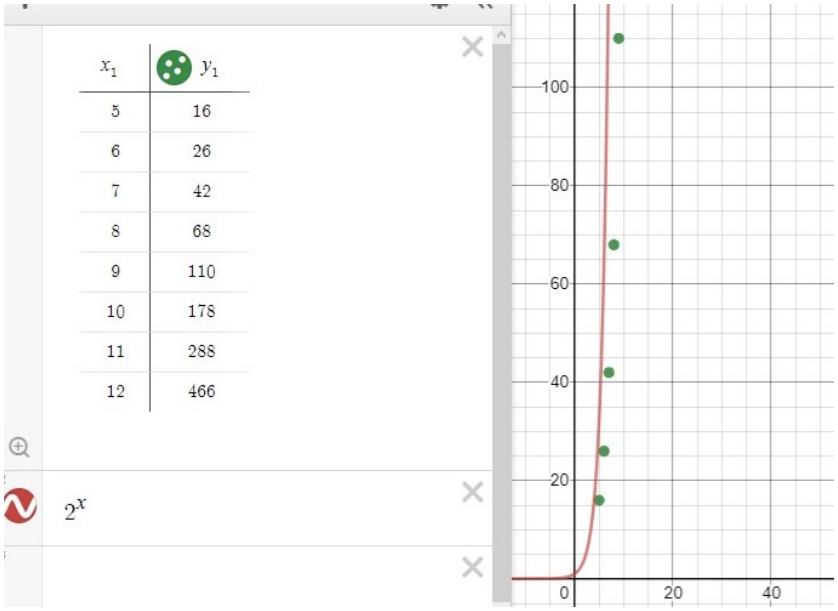


Figura 7: Puntos del análisis a posteriori del algoritmo 1 y gráfica propuesta

Viendo la imagen 7 se observa que FibonacciRecursivo $\in \theta(2^n)$.

El objetivo de hacer este último análisis es para comparar los algoritmos. Observando las imágenes 3, 6 y 7 podemos hacer una mejor comparación del comportamiento de su complejidad. En este caso tanto el algoritmo 3 como el algoritmo 4 son lineales, mientras que el algoritmo 1 es exponencial, claramente el orden de complejidad varía drasticamente, por lo que aplicar programación dinámica en este caso conviene bastante.

3.2. Problema de la mochila entera 0-1

Algunos resultados del conjunto del algoritmo 5 y el algoritmo 6.

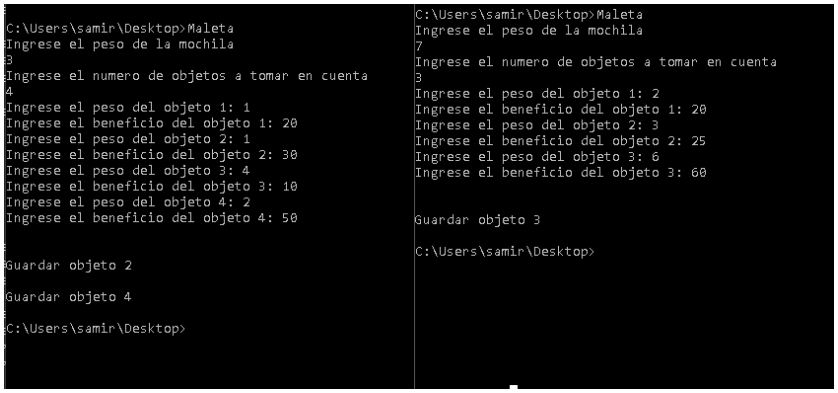


Figura 8: Ejemplos del algoritmo de mochila entera en funcionamiento

Haciendo un análisis a priori del algoritmo 5. Se obtiene la siguiente tabla:

MochilaEntera(b,w:[0,..,n],P	Complejidad		
for $c \leftarrow 0$ to $c \leq P$	O(P)	O(P)	
$g[0,c] \leftarrow 0$	O(1)		
for $j \leftarrow 1$ to $j \leq n$	O(n)	O(n)	
$g[j,0] \leftarrow 0$	O(1)		
for $j \leftarrow 1$ to $j \leq n$	O(n)		
for $c \leftarrow 1$ to $c \leq P$	O(P)		
if $c < w[j]$			O(nP)
$g[j,c] \leftarrow g[j-1,c]$	O(1)		
}else{		O(nP)	
if($g[j-1,c] \geq g[j-1,c-w[j]]+b[j]$)			
$g[j,c] \leftarrow g[j-1,c]$	O(1)		
}else{			
$g[j,c] \leftarrow g[j-1,c-w[j]]+b[j]$	O(1)		

Cuadro 10: Análisis A priori del algoritmo 5

De la tabla 10 se puede observar que el algoritmo 5 tiene un orden de complejidad O(nP) ya que se tienen que hacer ciclos para los n objetos y que se complemente con la cantidad P de la mochila.

Analizando ahora el análisis a Posteriori, se tomarán en cuenta los siguientes datos:

n	Peso de la mochila	Peso de los objetos	Beneficio	No pasos
1	50	[4,25,15,2,67]	[36,95,38,6,17]	252
2	3	[10,3,36,]	[20,20,3]	11
3	27	[18,10,9,8]	[54,9,55,2]	110
4	33	[28,5]	[37, 28]	68
5	43	[7,27,39]	[10,89,21]	131
6	17	[1,5,9,7,3,2]	[9,3,7,3,5,4]	104
7	15	[8,15,7]	[25,75,50]	47
8	7	[10,2,8,9]	[20,8,36,41]	30
9	5	[1,1,1,1,2]	[25,68,20,34,18]	27

Cuadro 11: Análisis a posteriori del algoritmo 5

Haciendo una comparación entre el peso de la mochila y los pasos obtenidos, se obtiene la siguiente gráfica con los puntos ya obtenidos y una recta propuesta.

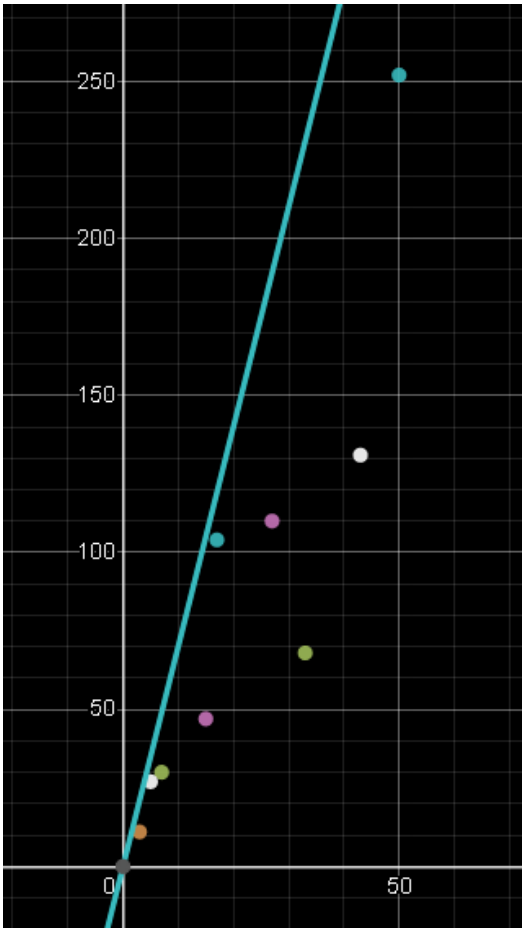


Figura 9: Gráfica de la tabla 11 con una recta propuesta y=nP

De aquí se puede observar que la recta y=nP pasa por encima de todos los puntos, lo que nos da a entender que el algoritmo 5 $\in$ O(nP).

Y comparando los resultados obtenidos en el análisis a priori y posteriori, se concluye que el problema de la mochila entera $\in$ O(nP).

4. Conclusiones

- Conclusiones generales

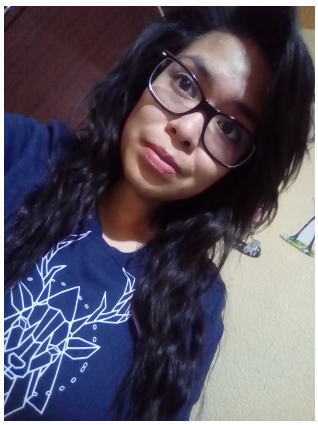
  En cuanto al algoritmo de Fibonacci se sabe que hay muchas maneras de implementarlo, ya sea a fuerza bruta, de manera recursiva o con programación dinámica por citar ejemplos, el que sea de manera recursiva no implica que sea mejor que una fuerza bruta ya que tienen complejidad exponencial y cuadrática respectivamente, sin embargo con

programación dinámica la situación es muy distinta ya que se reduce a complejidad lineal.

Para el algoritmo de la mochila entera, gracias a la recursividad de este algoritmo, el resultado se obtiene de una manera más óptima si se guardasen los datos en una tabla y posteriormente se vuelven a llamar los datos cuando se necesitan.

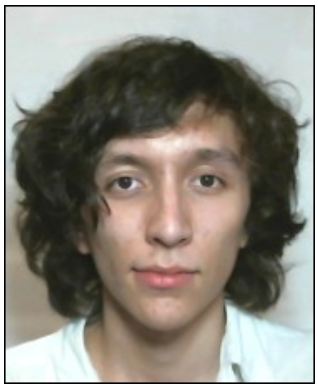
- Diana Paola De la Cruz Sierra

  La programación dinámica es muy importante y útil para problemas de optimización, ya que como se ve puede reducir la complejidad drásticamente. Solo hay que ver que enfoque conviene más implementar, si Top-down o Buttom-up.



- Raya Chávez Samuel Antonio:

  La programación dinámica puede ser muy útil para estos casos en donde los algoritmos de Greedy en lugar de simplificar el problema complican el problema, por lo que es una herramienta que, bien usada, nos pude ayudar a conseguir resultados más rápidos



5. Fuentes de información

- virtuniversidad. (S.F). Programación Dinámica. [PDF]. Disponible en:

  https://www.virtuniversidad.com/greenstone/collect/informatica/archives/HASH0106.dir/doc.pdf

- Facultad de estudios a distancia. (S.F). Capitulo 4: programación dinámica.[PDF]. Disponible en:

  http://virtual.umng.edu.co/distancia/ecosistema/ovas/ingenieria-civil/investigacion-de-operaciones-ii/unidad-4/DM.pdf

- Fochessato, N.(2015). Programación Dinámica. [PDF]. Disponible en:

  https://tc-arg.tk/pdfs/2015/02-DP.pdf

- Bravo, R. (2018). La sucesión de Fibonacci en el diseño. diciembre 18, 2020, de EADE Sitio web:

  https://www.eade.es/blog/186-la-sucesion-de-fibonacci-en-el-diseno: :text=Se %20trata %20

- Garza, M. (2019). Estrategias Algorítmicas - Parte 3. diciembre 18, 2020, de Cinestav Sitio web:

  https://www.tamps.cinvestav.mx/descargables/Cursospropedeuticos/Resoluciondeproblem

- López, R. (2018). Algunos problemas clásicos de Optimización Combinatoria: una propuesta metodológica. diciembre 18, 2020, de Unan Sitio web:

  https://repositorio.unan.edu.ni/8853/1/Artículo-Rudy %20Alberto %20López %20Potosme