

PRÁCTICA 6: PROGRAMACIÓN DINÁMICA.

Mendoza Martínez Eduardo, Aguilar Gonzalez Daniel.

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
edoomm8@gmail.com, daquilar glz97@gmail.com

Resumen: La siguiente práctica pretende mostrar el comportamiento de 3 algoritmos Algoritmo de fibonacci con el enfoque top-down y bottom-up, Problema de la mochila entera y Problema de líneas de producción estos implementados mediante el método de Programación dinámica, comparando sus tiempos de ejecución, así como la complejidad de estos.

Palabras Clave: Solución óptima, Enfoque top-down y bottom-up, Algoritmo de Fibonacci, Problema de la mochila entera y Algoritmo de líneas de producción.

1. Introducción

A lo largo del curso hemos desarrollado diferentes algoritmos los cuales pueden ser atacados con más de un método de solución de algoritmos como lo puede ser el algoritmo de fibonacci el cual se trató en una práctica anterior dándole solución con dos algoritmos diferentes uno iterativo y otro recursivo. En este caso trataremos este algoritmo mediante el método de programación dinámica. Al resolver un algoritmo con métodos diferentes no debe haber algún cambio al momento de mostrar su resultado ya que este debe ser el mismo debido a que el algoritmo da solución a un problema específico como observaremos en el desarrollo de esta práctica. En esta práctica mostraremos el cómo funciona la programación dinámica al momento de implementar un algoritmo como los 3 antes mencionados y observaremos a continuación.

2. Conceptos Básicos

Se requiere conocer algunos conceptos relacionados a la implementación de los algoritmos estudiados en esta práctica y los métodos a través de los cuales se desarrollaron. Se presenta a continuación la información

2.1. Programación dinámica

La programación dinámica es una técnica matemática que se utiliza para la solución de problemas matemáticos seleccionados, en los cuales se toma un serie de decisiones en forma secuencial. Proporciona un procedimiento sistemático para encontrar la combinación de decisiones que maximice la efectividad total, al descomponer el problema en etapas, las que pueden ser completadas por una o más formas (estados), y enlazando cada etapa a través de cálculos recursivos. La programación dinámica es básicamente un algoritmo de optimización. Significa que podemos resolver cualquier problema sin usar programación dinámica, pero podemos resolverlo de una mejor manera u optimizarlo usando programación dinámica. La idea básica de la programación dinámica es almacenar el resultado de un problema después de resolverlo. Entonces, cuando tenemos la necesidad de usar la solución del problema, entonces no tenemos que resolver el problema nuevamente y solo usar la solución almacenada.

2.2. Enfoques de programación Dinamica (top-down y bottom-up)

Top down.

El enfoque top-down enfatiza la planificación y conocimiento completo del sistema. Se entiende que la codificación no puede comenzar hasta que no se haya alcanzado un nivel de detalle suficiente, al menos en alguna parte del sistema. Esto retrasa las pruebas de las unidades funcionales del sistema hasta que gran parte del diseño se ha completado. También conocida como de arriba-abajo y consiste en establecer una serie de niveles de mayor a menor complejidad (arriba-abajo) que den solución al problema. Consiste en efectuar una relación entre las etapas de la estructuración de forma que una etapa jerárquica y su inmediato inferior se relacionen mediante entradas y salidas de información. Este diseño consiste en una serie de descomposiciones sucesivas del problema inicial, que recibe el refinamiento progresivo del repertorio de instrucciones que van a formar parte del programa.

Bottom-up.

Bottom-up hace énfasis en la programación y pruebas tempranas, que pueden comenzar tan pronto se ha especificado el primer módulo. Este enfoque tiene el riesgo de programar cosas sin saber como se van a conectar al resto del sistema, y esta conexión puede no ser tan fácil como se creyó al comienzo. La reutilización del código es uno de los mayores beneficios del enfoque bottom-up. El diseño ascendente se refiere a la identificación de aquellos procesos que necesitan computarizarse con forme vayan apareciendo, su análisis como sistema y su codificación, o bien, la adquisición de paquetes de software para satisfacer el problema inmediato.

2.3. Serie de Fibonacci

La serie Fibonacci, es una sucesión que comienza con los números 0 y 1, y a partir de estos, cada termino siguiente es la suma de los dos anteriores.

Ejemplo:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

2.4. Problema de la mochila entera

El problema de la mochila (KP) puede ser definido con un conjunto de n artículos donde cada artículo es identificado por n_x , con un valor entero p_x , y un peso w_x . El problema consiste en elegir un subconjunto de n artículos maximizando el beneficio obtenido considerando el peso total de los artículos seleccionados, sin exceder la capacidad c de la mochila. Se dispone de una mochila de capacidad C y de un conjunto de N objetos, donde los objetos son indivisibles. Describen a un objeto k que tiene un beneficio b_k y un peso p_k , para $k = 1, 2, \dots, N$. Para los autores, el problema consiste en averiguar qué objetos se pueden insertar en la mochila sin exceder la capacidad total de la misma, obteniendo el máximo beneficio.

3. Experimentación y Resultados

En esta sección se presentan la experimentación de cada algoritmo y los resultados que nos arrojan cada uno de ellos así como sus gráficas y explicaciones breves.

3.1. Algoritmo de Fibonacci

A continuación mostraremos la implementación del algoritmo de Fibonacci a través de dos enfoques que pertenecen a la programación dinámica

3.1.1. Pseudocódigo Algoritmo fibonacci Enfoque top-down

```

1-- fibo = [100]
2-- for i = 0 to i<100
3--     fibo[i]=-1
4-- fibo[0]=0
5-- fibo[1]=1
6-- fibonacci_desc(n[0,...,n], fibo:tabla)
7--     if (fibo[n] !=-1)
8--         return fibo[n]
9--     fibo[n]= fibonacci_desc(n-2) + fibonacci_desc(n-1)
```

```
10--      return fibo[n]
```

3.1.2. Algoritmo de Fibonacci Enfoque top-down

Consiste simplemente en comenzar resolviendo el problema de manera natural y almacenando las soluciones de los subproblemas en el camino.

Buscamos encontrar el n-esimo termino de la sucesión es decir si $n=5$ buscaremos que numero es el que se encuentra en la posición numero 5 de la serie de fibonacci.

Mencionado lo anterior comenzamos haciendo pruebas del algoritmo con este enfoque, nuestra primer prueba a realizar es con el valor de $n= 50$ es decir buscar el valor que toma la posición 50. En la figura 1 se muestra el tiempo que tardo en realizar esta instrucción y en la figura 2 la gráfica que nos arroja n contra el tiempo.

```
edoomm@edoomm-Satellite-L735:~/Documents/Análisis-de-algoritmos/P6$ python3 Top\ down.py
1 numero de la sucesion de fibonacci
---Calculado en 1.1444091796875e-05 segundos
4 numero de la sucesion de fibonacci
---Calculado en 5.245208740234375e-06 segundos
16 numero de la sucesion de fibonacci
---Calculado en 1.1444091796875e-05 segundos
64 numero de la sucesion de fibonacci
---Calculado en 3.0279159545898438e-05 segundos
256 numero de la sucesion de fibonacci
---Calculado en 0.00012636184692382812 segundos
1024 numero de la sucesion de fibonacci
---Calculado en 0.0005166530609130859 segundos
4096 numero de la sucesion de fibonacci
---Calculado en 0.026207923889160156 segundos
```

Figura 1 - Tiempo de ejecución para $n=50$ enfoque top down.

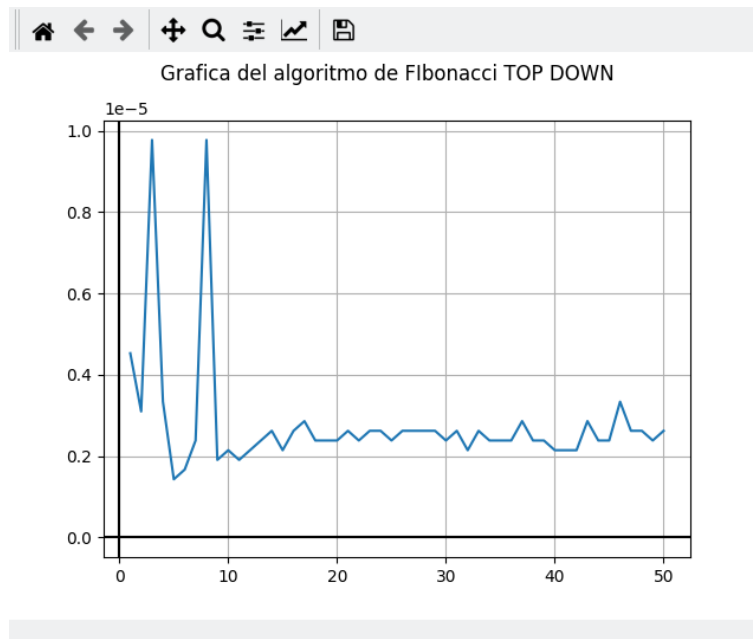


Figura 2 - Gráfica del algoritmo para $n=50$ enfoque top down.

Al observar la figura 2 nos damos cuenta que hay variaciones entre las primeras 10 posiciones, a partir de esa posición las variaciones son más pequeñas pero

esta informacion no nos basta para poder buscar una función que acote de manera correcta nuestra grafica, así que seguiremos aumentando el valor de n buscando otro comportamiento de la grafica. Nuestra siguiente prueba es aumentar el valor a $n=1000$ como se muestra a continuación. En la figura 3 podemos observar el tiempo que tarda en buscar el 1000 esimo termino de la serie y en la figura 4 la grafica obtenida.

```

991 numero de la sucesion de fibonacci
---Calculado en 1.1920928955078125e-06 segundos
992 numero de la sucesion de fibonacci
---Calculado en 1.430511474609375e-06 segundos
993 numero de la sucesion de fibonacci
---Calculado en 1.430511474609375e-06 segundos
994 numero de la sucesion de fibonacci
---Calculado en 9.5367431640625e-07 segundos
995 numero de la sucesion de fibonacci
---Calculado en 1.1920928955078125e-06 segundos
996 numero de la sucesion de fibonacci
---Calculado en 1.430511474609375e-06 segundos
997 numero de la sucesion de fibonacci
---Calculado en 1.1920928955078125e-06 segundos
998 numero de la sucesion de fibonacci
---Calculado en 1.1920928955078125e-06 segundos
999 numero de la sucesion de fibonacci
---Calculado en 1.6689300537109375e-06 segundos
1000 numero de la sucesion de fibonacci
---Calculado en 1.1920928955078125e-06 segundos

```

Figura 3 - Tiempo de ejecución para $n=1000$ enfoque top down.

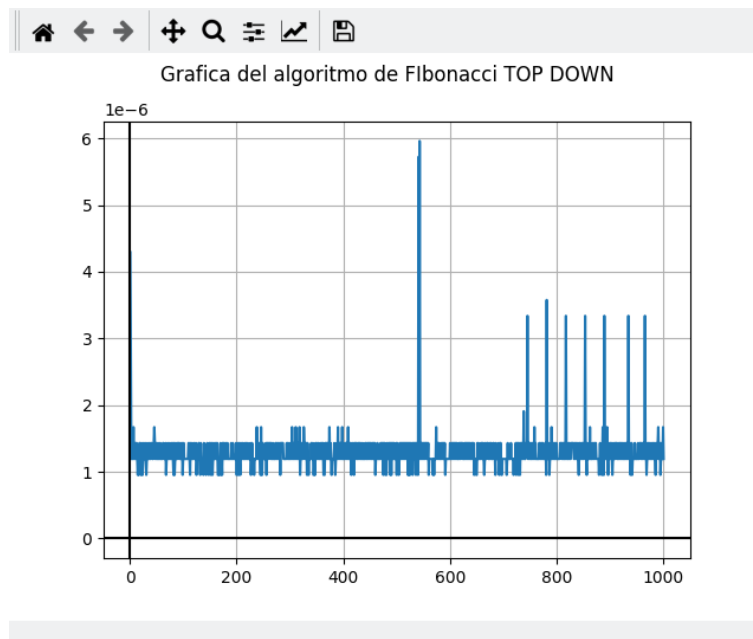


Figura 4 - Gráfica del algoritmo para $n=1000$ enfoque top down.

Aumentando el valor a 1000 seguimos viendo que nuestra grafica tiene variaciones en determinar el tiempo de cada posición sin aún poder observar la forma que tomara nuestra gráfica, así que como lo venimos haciendo seguiremos incrementando nuestro valor hasta poder observar de mejor manera el comportamiento que buscamos. Nuestro siguiente experimento es aumentar el valor a $n = 3125$ y analizar su grafica que produce la cual se muestra en la figura 6 y en la figura 5 se muestra el tiempo que tarda el programa en ejecutar esa cantidad.

```
edoomm@edoomm-Satellite-L735:~/Documents/Análisis-de-algoritmos/P6$ python3 Top\ down.py
1 numero de la sucesion de fibonacci
---Calculado en 1.0013580322265625e-05 segundos
5 numero de la sucesion de fibonacci
---Calculado en 5.9604644775390625e-06 segundos
25 numero de la sucesion de fibonacci
---Calculado en 1.4543533325195312e-05 segundos
125 numero de la sucesion de fibonacci
---Calculado en 5.626678466796875e-05 segundos
625 numero de la sucesion de fibonacci
---Calculado en 0.000339508056640625 segundos
3125 numero de la sucesion de fibonacci
---Calculado en 0.024589061737060547 segundos
```

Figura 5 - Tiempo de ejecución para $n=3125$ enfoque top down.

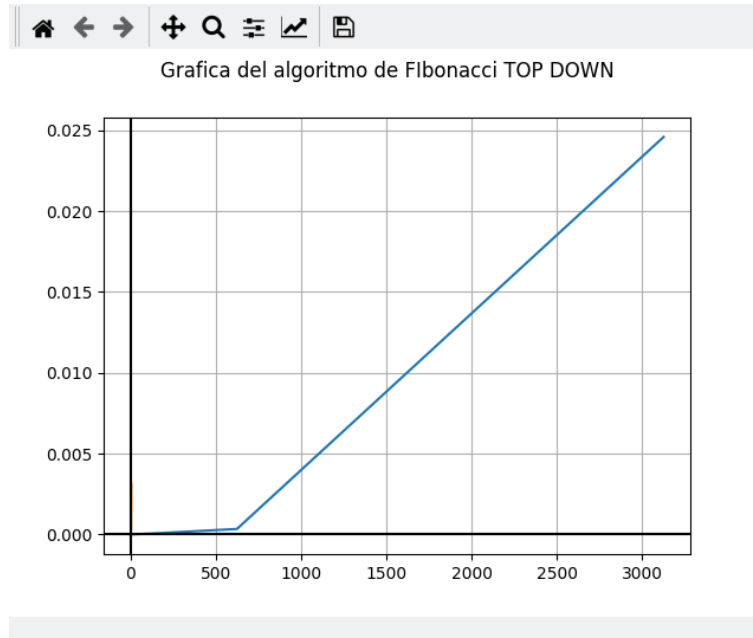


Figura 6 - Gráfica del algoritmo para $n=3125$ enfoque top down.

En la figura 6 podemos observar que la gráfica comienza a tomar una forma lineal después de una cantidad mayor a 700 dándonos así más información del comportamiento que tomara para poder darnos una idea de que forma podría ser la función que la acote de manera adecuada nuestra gráfica que en esta ocasión parece ser de forma lineal. Haremos una última prueba incrementando

el valor a $n = 4096$ para poder cerciorarnos que nuestra grafica continua tomando esta forma como se muestra en la figura 8 y en la figura 9 observamos su tiempo de ejecución.

```
edoomm@edoomm-Satellite-L735:~/Documents/Análisis-de-algoritmos/P6$ python3 Top\ down.py
1 numero de la sucesion de fibonacci
---Calculado en 1.1444091796875e-05 segundos
4 numero de la sucesion de fibonacci
---Calculado en 5.245208740234375e-06 segundos
16 numero de la sucesion de fibonacci
---Calculado en 1.1444091796875e-05 segundos
64 numero de la sucesion de fibonacci
---Calculado en 3.0279159545898438e-05 segundos
256 numero de la sucesion de fibonacci
---Calculado en 0.00012636184692382812 segundos
1024 numero de la sucesion de fibonacci
---Calculado en 0.0005166530609130859 segundos
4096 numero de la sucesion de fibonacci
---Calculado en 0.026207923889160156 segundos
```

Figura 7 - Tiempo de ejecución para $n=4096$ enfoque top down.

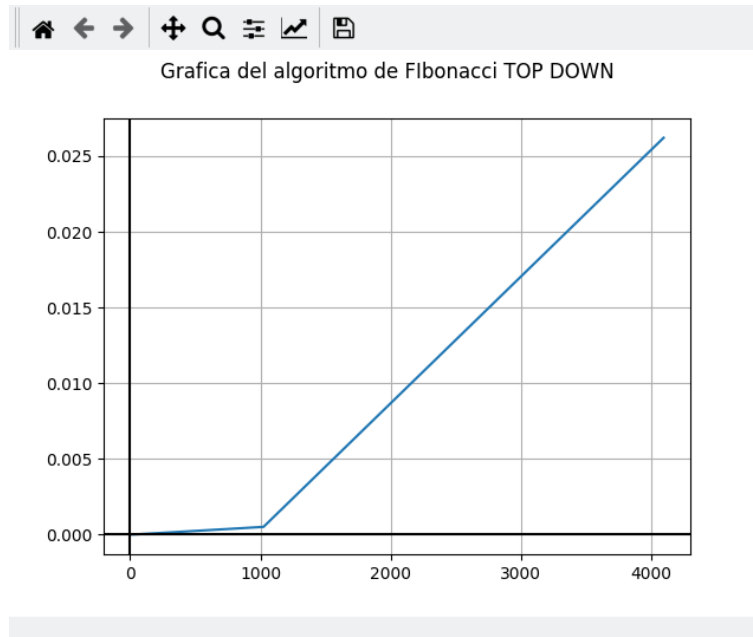


Figura 8 - Gráfica del algoritmo para $n=4096$ enfoque top down.

Como podemos observar efectivamente la grafica sigue tomando el mismo comportamiento que mencionamos anteriormente asi que ahora procederemos a calcular una función que acote nuestra grafica. Como se menciono anteriormente nuestra grafica va tomando una forma lineal por lo que podemos buscar otra de la misma índole.

La función obtenida nos queda de la siguiente manera:

$$f(n) = \frac{1}{10000}n$$

En la figura 10 podemos observar una grafica con valor de 10,000 acotada de manera correcta por nuestra función propuesta y en la figura 9 el tiempo que tardo en ejecutarse.

```

---Calculado en 0.006108522415161133 segundos
edoomm@edoomm-Satellite-L735:~/Documents/Analisis-de-algoritmos/P6$ python3 Top\ down.py

1 numero de la sucesion de fibonacci
---Calculado en 1.049041748046875e-05 segundos
10 numero de la sucesion de fibonacci
---Calculado en 1.0251998901367188e-05 segundos
100 numero de la sucesion de fibonacci
---Calculado en 7.510185241699219e-05 segundos
1000 numero de la sucesion de fibonacci
---Calculado en 0.0005965232849121094 segundos
10000 numero de la sucesion de fibonacci
---Calculado en 0.03348517417907715 segundos

```

Figura 9 - Tiempo de ejecución para $n=10000$ enfoque top down acotada por $f(n)$.

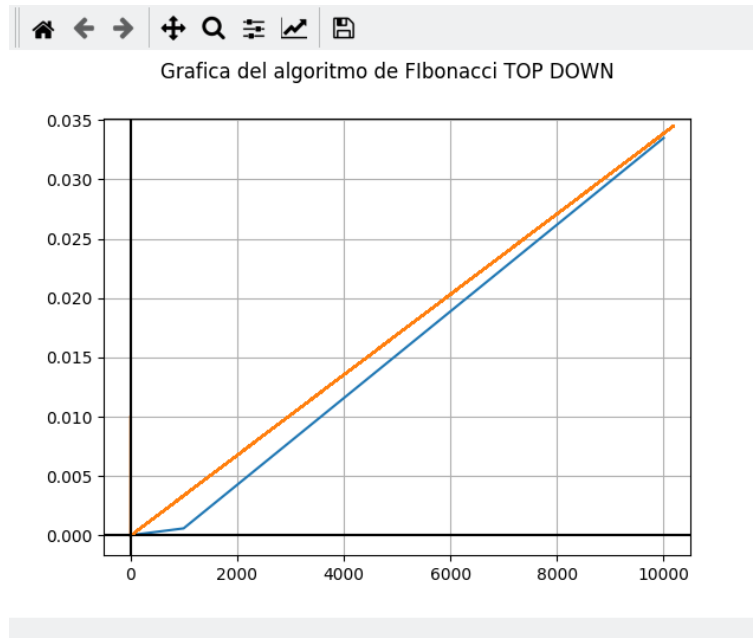


Figura 10 - Gráfica del algoritmo para $n=10000$ enfoque top down acotada por $f(n)$.

Gracias a todas las pruebas anteriores y a sus graficas que estas arrojaron pudimos encontrar la función que las acota llegando a la conclusión que el algoritmo Fibonacci con Enfoque top down de Programación dinámica tiene complejidad de orden Lineal $\theta(n)$.

3.1.3. Pseudocódigo Algoritmo fibonacci Enfoque bottom-up

```

1-- fibo = [100]
2-- fibonacci_asc(n[0,...,n], fibo:tabla)
3--     if n <= 1
4--         return 1
5--     else
6--         fibo[0] = 0

```



```

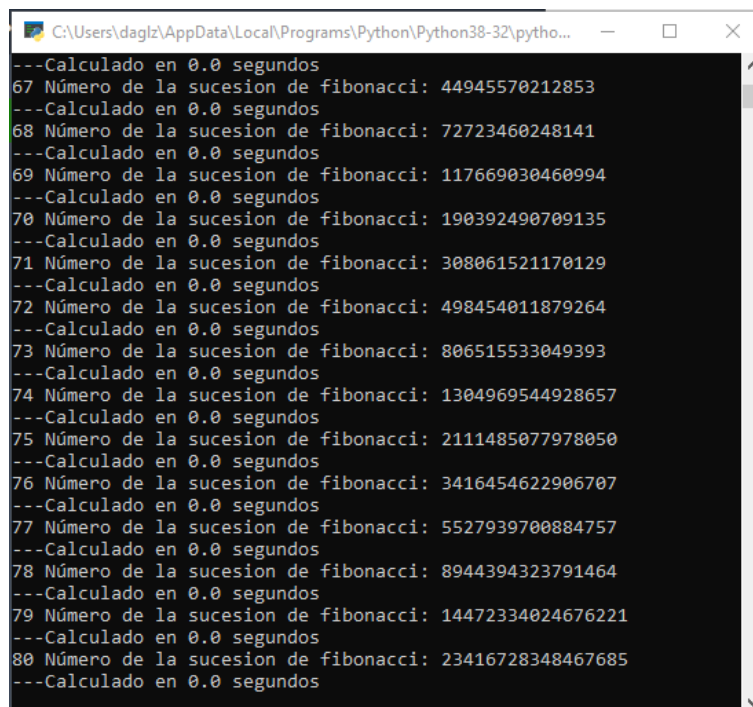
7--      fibo[1] = 1
8--      for i=2 to i<=n do
9--          fibo[i] = fibo[i-1] + fibo[i-2]
10--     return fibo[n]

```

3.1.4. Algoritmo de Fibonacci Enfoque bottom-up

Este enfoque consiste en crear una tabla de abajo hacia arriba y devuelve la última entrada de la tabla.

Comenzamos haciendo pruebas del algoritmo con este enfoque, iniciando con $n = 80$. En la figura 11 mostramos el tiempo que tardo en buscar el valor de esta posición y el valor del numero que esta en esa posición y en la figura 12 la grafica que resulta haciendo la comparacion de la posición buscada contra el tiempo que tardó .



```

C:\Users\daglz\AppData\Local\Programs\Python\Python38-32\pytho...
---Calculado en 0.0 segundos
67 Número de la sucesion de fibonacci: 44945570212853
---Calculado en 0.0 segundos
68 Número de la sucesion de fibonacci: 72723460248141
---Calculado en 0.0 segundos
69 Número de la sucesion de fibonacci: 117669030460994
---Calculado en 0.0 segundos
70 Número de la sucesion de fibonacci: 190392490709135
---Calculado en 0.0 segundos
71 Número de la sucesion de fibonacci: 308061521170129
---Calculado en 0.0 segundos
72 Número de la sucesion de fibonacci: 498454011879264
---Calculado en 0.0 segundos
73 Número de la sucesion de fibonacci: 806515533049393
---Calculado en 0.0 segundos
74 Número de la sucesion de fibonacci: 1304969544928657
---Calculado en 0.0 segundos
75 Número de la sucesion de fibonacci: 2111485077978050
---Calculado en 0.0 segundos
76 Número de la sucesion de fibonacci: 3416454622906707
---Calculado en 0.0 segundos
77 Número de la sucesion de fibonacci: 5527939700884757
---Calculado en 0.0 segundos
78 Número de la sucesion de fibonacci: 8944394323791464
---Calculado en 0.0 segundos
79 Número de la sucesion de fibonacci: 14472334024676221
---Calculado en 0.0 segundos
80 Número de la sucesion de fibonacci: 23416728348467685
---Calculado en 0.0 segundos

```

Figura 11 - Tiempo de ejecución para $n=80$.

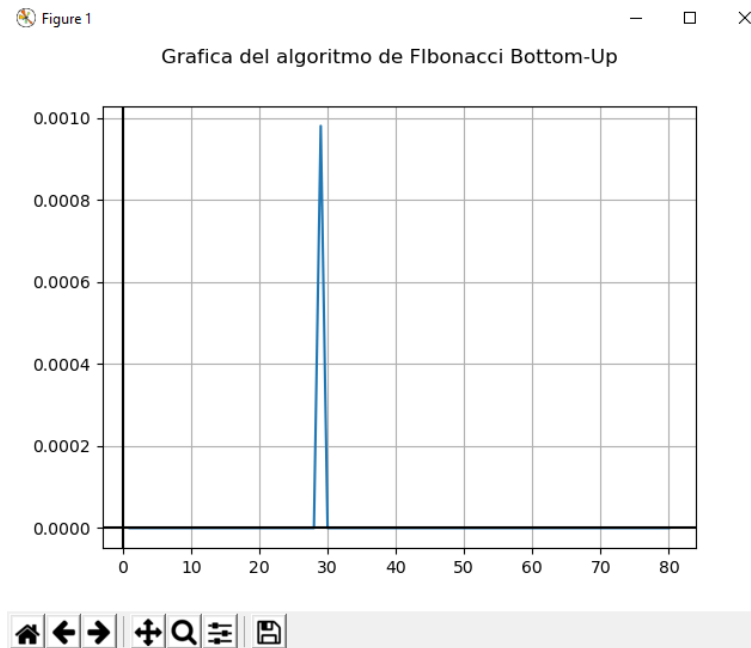


Figura 12 - Gráfica del algoritmo para $n=80$.

En la grafica anterior que no toma una forma en la cual podamos observar el comportamiento de esta asi que lo siguiente que haremos es aumentar el valor de n a 800 y analizaremos la forma que toma la gráfica con este valor y determinaremos si es la adecuada o seguimos aumentando el valor de n . En la figura 13 observamos el tiempo que tomo en encontrar el valor de la posición 800 y en la figura 14 su gráfica.

```

C:\Users\daglz\AppData\Local\Programs\Python\Python38-32\python.exe
---Calculado en 0.00099945068359375 segundos
794 Número de la sucesion de fibonacci: 386101382166516474039396022520588
1290642618588143535442629650892849321138424808721667330305914891352398829
018164037784860890097725341978213033877853138921866971476017
---Calculado en 0.0009987354278564453 segundos
795 Número de la sucesion de fibonacci: 624725159448736167929003456941238
8977523253930560892938227223266680603736894292437992563804744759239172503
220823069261664877091466883515638367183321660143831291627330
---Calculado en 0.0009996891021728516 segundos
796 Número de la sucesion de fibonacci: 101082654161525264196839947946182
7026816587251870442838085687415952992487531910115965989411065965059157133
2238987107046525767189192225493851401061174799065698263103347
---Calculado en 0.0009992122650146484 segundos
797 Número de la sucesion de fibonacci: 163555170106398880989740293640306
5924568912644926532131908409742621052861221339359765245791540440983074383
5459810176308190644280659109009489768244496459209529554730677
---Calculado en 0.0 segundos
798 Número de la sucesion de fibonacci: 264637824267924145186580241586489
2951385499896796974969994097158574045348753249475731235202606406042231516
7698797283354716411469851334503341169305671258275227817834024
---Calculado en 0.0009989738464355469 segundos
799 Número de la sucesion de fibonacci: 428192994374323026176320535226795
8875954412541723507101902506901195098209974588835496480994146847025305900
3158607459662907055750510443512830937550167717484757372564701
---Calculado en 0.00099945068359375 segundos
800 Número de la sucesion de fibonacci: 692830818642247171362900776813285
1827339912438520482071896604059769143558727838311227716196753253067537417
0857404743017623467220361778016172106855838975759985190398725
---Calculado en 0.0009992122650146484 segundos

```

Figura 13 - Tiempo de ejecución para $n=800$.

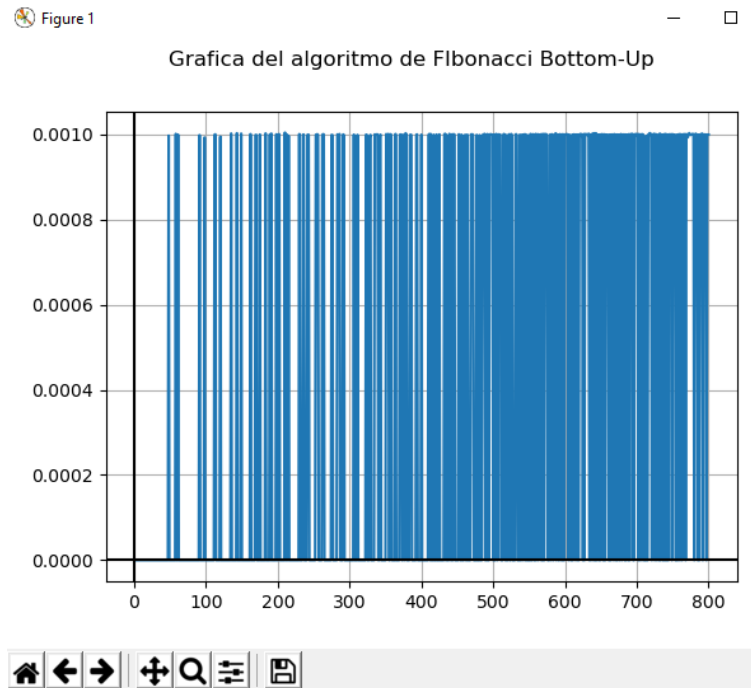


Figura 14 - Gráfica del algoritmo para $n=800$.

Al aumentar el valor vemos que nuestra grafica sigue sin tomar un comportamiento que nos brinde información para poder encontrar una funcion y acotarla, así que nuevamente aumentaremos el valor de n a 4 cifras y seguiremos analizando.

```

C:\Users\dagiz\AppData\Local\Programs\Python\Python38-32\python.exe
331903276104376830042845525894939078924885825165898842637199062697185409611280
558946421695902926675663788546816137265052036213153392211337203130401417226119
117094991855307858954228708472899199866050666777801239417868352233513822368306
141157044835000896152168842810324773808497839712786397061263761872144244202658
333612037810794710623679903189287356630503809180684634399853992727387799831351
609147665705890558156950720149864263447267695560506698946448809280333526165497
978763920301340208376742378393773619372542975636021954269713162656025523121874
739826350034755752132967929103960839986564640903144044676485651863607106971600
423651632770812196556752832808268743830591602226425732080214529575588859822116
015731046357317232128858640309124101232115359784993563849885327921672156806082
430038866399963299300201018080326352707844916607962388308368728099922127179155
888462869
---Calculado en 0.00699305534362793 segundos
5070 Número de la sucesion de fibonacci: 1651395379846466447842264238609216759
610352065888759235679963252124705114139891029861977143423613192697019403215493
643880127766501264956973882603602788962016572796829977540887299315123699430773
721409400762130312535402348388876492835864540793685858989784201339177026610359
224127567576498431295675491532334613550041974651791712283566095418096113562807
585083832500915392657100316830021807693757823458274647865199957659515148625531
749519445729191398474718352804482250302700369399650216642608023285249415974698
294476946643368246933743910441567705692200952235077296410518784397542775560239
955205425250116277141076578648271496938927048308245027144078109628489295962766
73418130708528693753893133641180590917873334206618908904203809022103589502955
071898032036011534234937575407601775421689652625740621595052411509336833352842
771634293889987949970395729694838659947051221903451818163597139998377668227855
532796165082078914614447885030889494118183317957575001783899013965393071003947
966091144838798011970450398597176940500170974688364567467110398749813348739180
015814760
---Calculado en 0.007998943328857422 segundos

```

Figura 15 - Tiempo de ejecución para $n=5070$.

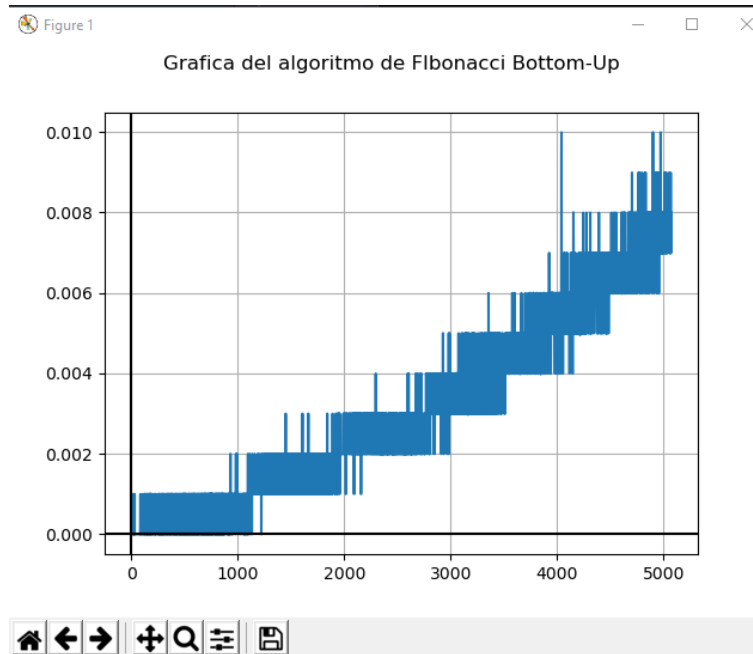


Figura 16 - Gráfica del algoritmo para n=5070.

En la figura 16 observamos que la gráfica comienza a tomar forma aunque aún no se ve claramente como es, así que continuaremos aumentando el valor de n a 8000 como se muestra en las siguientes figuras.

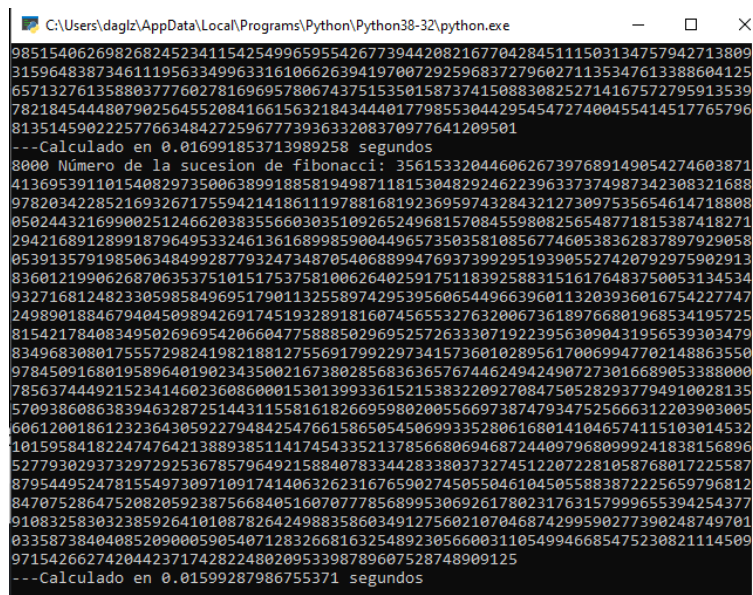


Figura 17 - Tiempo de ejecución para n=8000.

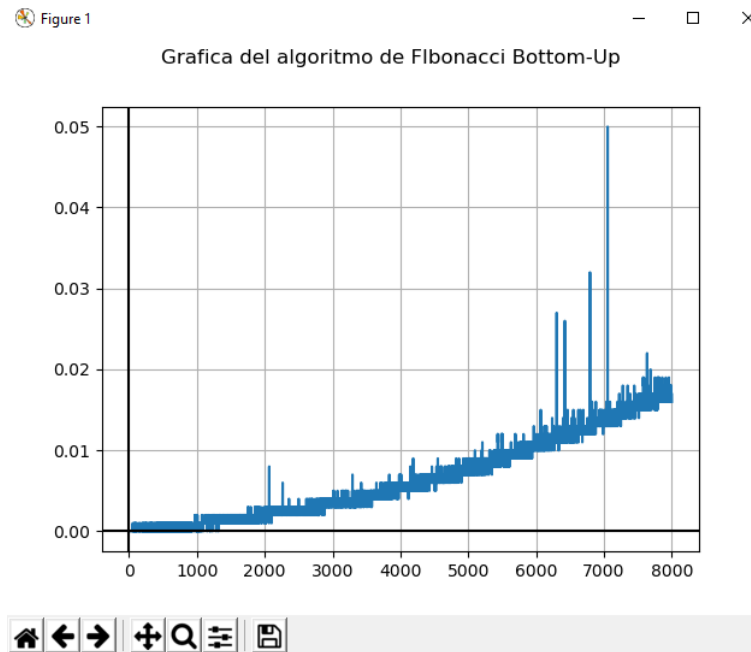


Figura 18 - Gráfica del algoritmo para n=8000.

En la figura 18 nos damos cuenta que con el valor de 8000 la gráfica ya muestra una forma más clara con la que el siguiente paso es buscar una función que pueda acotar de manera correcta esta gráfica.

Como hemos visto anteriormente entre más incrementamos el valor de n la gráfica se va viendo más clara cada vez así que aunque con el valor de 8000 ya tenemos una mejor grafica, aumentaremos el valor y buscaremos una funcion.

Lo siguiente es encontrar una función que acote nuestra grafica y nos determine el orden de complejidad de este algoritmo todo basado en graficas que hemos mostrado. La función obtenida nos queda de la siguiente manera:

$$f(n) = \frac{1}{60000}n$$

En la figura 19 mostramos como nuestra $f(n)$ obtenida acota de manera correcta la grafica del n-esimo de la serie de Fibonacci el cual tiene el valor de 12300 y en la figura 20 el tiempo que tardó en realizarse.

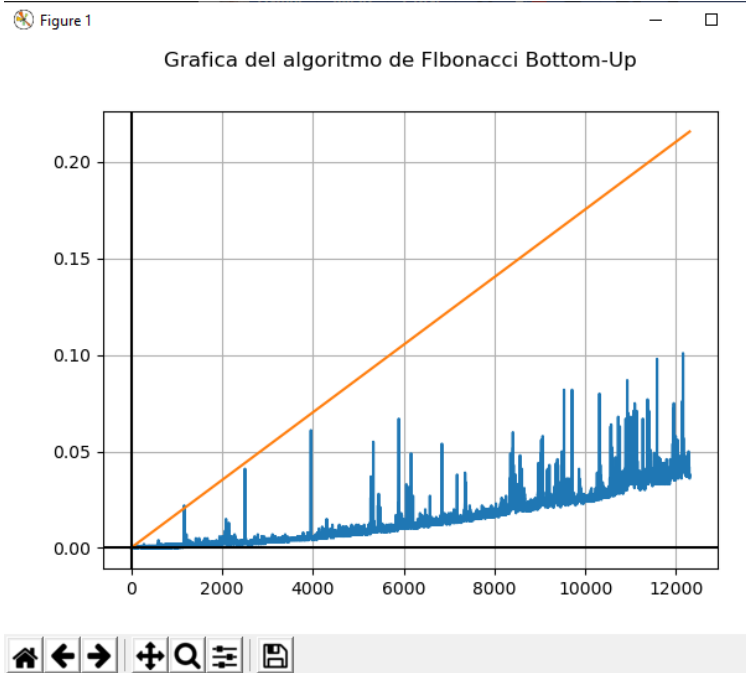


Figura 19 - Gráfica con valor 12300 acotada por $f(n)$.

```

C:\Users\dagiz\AppData\Local\Programs\Python\Python38-32\python.exe
12300 Número de la sucesion de fibonacci: 15793925043853427170865246317928727240
43150743369853750392828158686671607410335885255585347343701422685503517407997246
336149009539095202849066263324862953335848774791027274915057190322503574820473
99179983203467017775142475014675680183183273486317488293246028722806032586113374
21262293260022891291556141114779206087223802952061576271028224920456787591962789
32733373941025021232445364362667099957844307956074675116179556559268808567414805
54515836419968174423510322820210177006598282924326461387710193460554025072295209
47394565642207256693210064907048643904081917626980736647649017248560111090640125
18665765382648303751786291366362073799950913529703834343414921933896836649665700
0900785460505359311200267761934651096039797804278135776874006640332429335543315
84188745344307612587881732736123702066125329123658848164785036617855134548646879
83804926631476153407745478566597596225833779180494572933194074866386878525369465
64669018986726304202343973475542522194523215369019219983909642954377479533679564
76312036361451918584194787206208214366822886924201160030296300483016379235632846
84876810078008703939943387543822638207386901679775259414774480160679910563068936
66910276911076608508512377086474778936566910080811769499793203530709866088173169
87061174476034614738406327611856934515623647086518478367129605024762291574481806
1214918626439707592973510468434577669799126905281671908587425159777960352949043
43073171683076835985373786751773472976162194081608411256598984557444707716686317
92429864134794008305636012564471057741258584417189130281041799052010148621424088
34259812769885017127783258863789863456024206542511037883796085475838262248174454
70948680859058157187177199952078500940188901291720244048731353591028817423034329
91479319104125925391240365095908636250836556624203201815465272211612517780205766
75650645575876537342600792111364956840266171887487936815836456668635965968137051
79996162969177217542630948077469086232707530613790897439420100180857005755525569
48079070612035962766547332624301282665770199797017053762837548472348712297957727
7748072673727832735948387340943399423322849637175317290204513522533260834960531
42385228556608108392311935368181635584749401427742552411331963790653061500725660
90661287415536454136571567147986256609896886910608409543890263608788279476960359
96864139999830253040258888424269125839417852120267723003884195063908471830430214
60347994962433953912208323027021032747184800606660832668535966241772314891062071
62924970144190425147870933391736182198655040888195017798931011862231528654606314
12773409615502479391793709948690640425823686230163600
---Calculado en 0.03797793388366699 segundos

```

Figura 20 - Tiempo de ejecución para $n=12300$.

Como prueba extra incrementamos el valor de n manteniendo los mismos valores de nuestra función para observar que la función acota correctamente nuestra gráfica como se muestra en la figura 21.

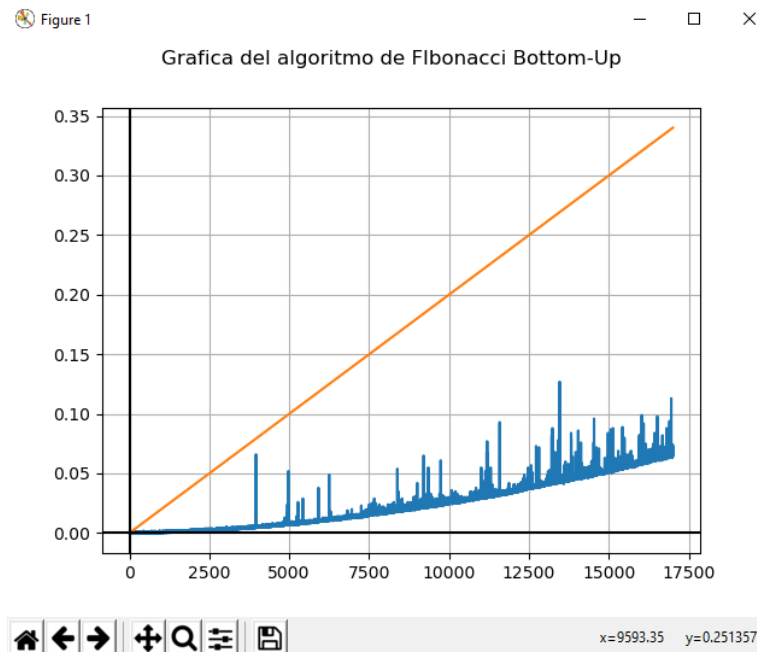


Figura 21 - Gráfica con valor 17300 acotada por $f(n)$.

Concluyendo así que gráficamente el algoritmo Fibonacci con el Enfoque bottom-up de Programación dinámica tiene complejidad de orden Lineal $\theta(n)$.

3.1.5. Top-down vs Bottom-up

En esta sección mostramos que algoritmo de fibonacci se puede resolver de varias formas, aquí lo resolvimos mediante los dos enfoques que tiene el método programación dinámica y gracias a los resultados experimentales mostrados anteriormente pudimos darnos cuenta que aunque los dos enfoques se componen y actúan de manera diferente ambos tienen un Orden de complejidad Lineal es decir $\theta(n)$, nos dimos cuenta de esto gracias a las gráficas aquí mostradas las cuales se obtenían gracias al tiempo que tardaba el programa en encontrar el valor de una posición.

Ambos tienen complejidad lineal pero cada uno de ellos actúa de manera distinta aunque no lo podamos ver de manera directa, ejemplo de esto es el tiempo en que cada uno de ellos calcula cada número.

El enfoque que mejor resultados nos dio según las gráficas fue el Enfoque Top-down ya que este enfoque comenzó a tener variaciones tiempo más notables a partir del valor 1000 fue en este punto donde comenzó a incrementarse de tal manera que nos daba una aparente forma lineal a demás las variaciones eran muy leves esto quiere decir que el tiempo en calcular una posición es muy corto esto debido a que es más amable con el rendimiento ya que nos ayuda a evitar cálculos innecesarios debido a que cada que calcula un número nuevo lo almacena esto con motivo que si más adelante requiere calcularlo no lo hará

de nuevo, solo lo mandará a llamar debido a que usa llamadas a funciones de recursividad. Esto puede poner en desventaja este enfoque ya que las llamadas a funciones recursivas requieren memoria de una fuente la cual tiene límites fijos y si la profundidad de recursión es demasiado profunda el programa se puede bloquear.

El principal beneficio que nos brinda el enfoque top-down es ejecutar de manera más rápida nuestro programa y ser más amable con su rendimiento al evitar realizar cálculos innecesarios es por este motivo que con el Enfoque Top-down se obtienen mejores resultados.

3.2. Algoritmo de la Mochila Entera

En prácticas previas estudiamos el algoritmo de la mochila fraccionaria, donde se maximizaba el beneficio que una mochila puede transportar con ciertos objetos dados, en esta ocasión veremos el algoritmo de la mochila entera. La principal diferencia entre estos 2 es, como su nombre lo indica, que en una se pueden ingresar fracciones de ciertas cosas, mientras que la mochila entera no podemos ingresar fracciones de los objetos que estarán dentro de la mochila, en otras palabras, podríamos decir que los objetos a ingresar son indivisibles.

3.2.1. Pseudocódigos del algoritmo de la mochila entera

Para resolver la problemática de la mochila entera es necesario implementar dos funciones. La primera de ella la denominamos como *generarTabla* y su pseudocódigo se muestra a continuación.

```

generarTabla(w, b:[1,...,n], P:int)
1-  for c = 0 to c <= P do
2-      g[0,c] = 0
3-  for j = 1 to j <= n do
4-      g[j,0] = 0
5-  for j = 1 to j <= n do
6-      for c = 1 to c <= P to
7-          if c < w[j]
8-              g[j,c] = g[j-1,c]
9-          else
10-              if g[j-1,c] >= g[j-1,c-w[j]] + b[j]
11-                  g[j,c] = g[j-1,c]
12-              else
13-                  g[j,c] = g[j-1, c-w[j]] + b[j]
14-  return g

```

La anterior función generará una tabla y con esta salida, podremos resolver el problema de la mochila entera a través del pseudocódigo siguiente.

```

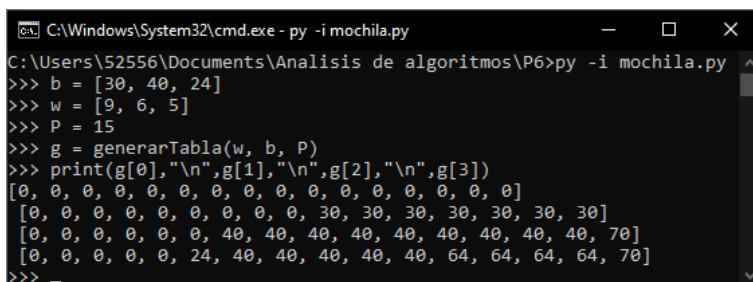
test(w, b:[1,...,n], P:int, g:[0,...,n][0,...,P])
1-  if j > 0 then
2-      if c < w[j] then
3-          test(j-1,c)
3-      else
4-          if g[j-1, c-w[j]] + b[j] > g[j-1,c] then
5-              test(j-1, c-w[j])
6-              print("guardar objeto", j)
7-          else
8-              test(j-1, c)

```

3.2.2. Ejemplo de la diapositiva

Posterior a implementar los pseudocódigos en Python sin ningún tipo de error al ejecutarlo, analizamos que estuviese trabajando correctamente y nos basamos en el ejemplo de la diapositiva, por lo que debíamos obtener los mismos resultados.

Primeramente, declaramos los arreglos de los beneficios y pesos y el peso P de la mochila para después generar la tabla con la que posteriormente nos apoyaremos. Todo esto se muestra en la figura 22, y vemos que nuestra tabla g se genera tal cual fue presentado en las diapositivas.



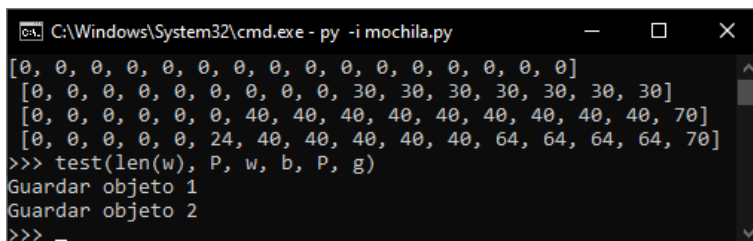
```

C:\Windows\System32\cmd.exe - py -i mochila.py
C:\Users\52556\Documents\Análisis de algoritmos\P6>py -i mochila.py
>>> b = [30, 40, 24]
>>> w = [9, 6, 5]
>>> P = 15
>>> g = generarTabla(w, b, P)
>>> print(g[0], "\n", g[1], "\n", g[2], "\n", g[3])
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 30, 30, 30, 30, 30, 30]
[0, 0, 0, 0, 0, 0, 0, 40, 40, 40, 40, 40, 40, 40, 40, 70]
[0, 0, 0, 0, 0, 0, 24, 40, 40, 40, 40, 40, 64, 64, 64, 70]
>>>

```

Figura 22 - Generación de tabla auxiliar del ejemplo de la diapositiva

Seguidamente, llamamos a nuestra función *test* que nos permitirá ver que objetos guardar. Obtuvimos los mismos resultados que se obtuvieron en la diapositiva, así como se muestra en la figura 23, por lo que nuestra implementación fue correcta.



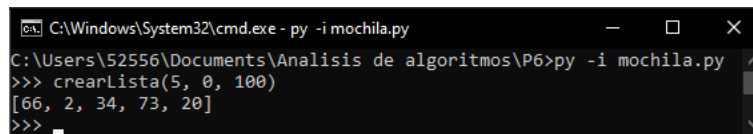
```

C:\Windows\System32\cmd.exe - py -i mochila.py
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 30, 30, 30, 30, 30, 30]
[0, 0, 0, 0, 0, 0, 0, 40, 40, 40, 40, 40, 40, 40, 40, 70]
[0, 0, 0, 0, 0, 0, 24, 40, 40, 40, 40, 40, 64, 64, 64, 70]
>>> test(len(w), P, w, b, P, g)
Guardar objeto 1
Guardar objeto 2
>>>

```

Figura 23 - Resultado en consola del algoritmo de la mochila entera con el ejemplo de la diapositiva.

Ya teniendo nuestro algoritmo funcionando correctamente y antes de continuar con la siguiente sección, donde veremos 10 ejemplos distintos usando este algoritmo, necesitaremos crear algunas funciones que nos permitan crear datos aleatoriamente, tales como arreglos. La función será *crearLista*, donde le pasaremos el tamaño de la lista y el rango que querramos que tengan los elementos. En la figura 24 vemos efectivamente que nos regresa arreglos completamente aleatorios.



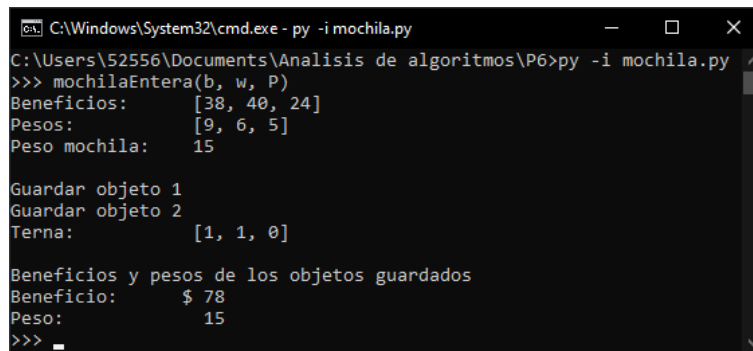
```

C:\Windows\System32\cmd.exe - py -i mochila.py
C:\Users\52556\Documents\Análisis de algoritmos\P6>py -i mochila.py
>>> crearLista(5, 0, 100)
[66, 2, 34, 73, 20]
>>>

```

Figura 24 - Resultado en consola de la función *crearLista*

Después, creamos una función denominada *mochilaEntera* que nos servirá como apoyo para poder mostrar mejor la información que nos regresan las dos funciones anteriores. A su vez, modificamos la función *test* de modo que aceptase un nuevo parametro denominado *terna*, un arreglo en el cual se irán almacenando los índices de los objetos a guardar para así obtener la terna de interés y poder calcular sus pesos y beneficios de estos objetos. En la figura 25 vemos su funcionamiento con el ejemplo de la diapositiva.



```

C:\Windows\System32\cmd.exe - py -i mochila.py
C:\Users\52556\Documents\Análisis de algoritmos\P6>py -i mochila.py
>>> mochilaEntera(b, w, P)
Beneficios: [38, 40, 24]
Pesos: [9, 6, 5]
Peso mochila: 15

Guardar objeto 1
Guardar objeto 2
Terna: [1, 1, 0]

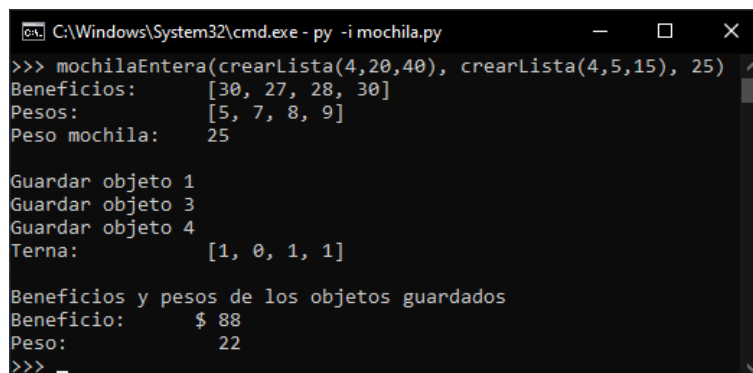
Beneficios y pesos de los objetos guardados
Beneficio: $ 78
Peso: 15
>>>

```

Figura 25 - Resultado en consola de la función *mochilaEntera*

3.2.3. 10 ejemplos del algoritmo de la mochila entera

Para el primer ejemplo decidimos aumentar un poco el tamaño de los arreglos de las dos listas a 4 elementos, mientras que para el peso de la mochila solo se aumento un poco. La figura 26 nos muestra que el algoritmo funciona acorde a se planteo con un nuevo ejemplo completamente distinto y si lo analizamos observándolo nosotros, vemos que efectivamente dió con la solución óptima donde se obtiene el máximo beneficio posible.



```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(4,20,40), crearLista(4,5,15), 25)
Beneficios: [30, 27, 28, 30]
Pesos: [5, 7, 8, 9]
Peso mochila: 25

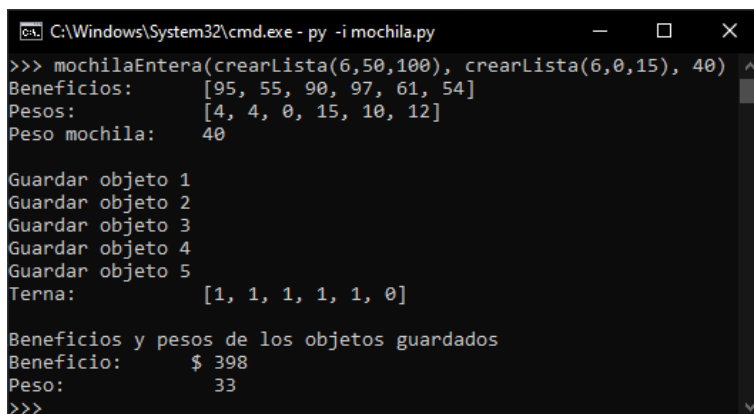
Guardar objeto 1
Guardar objeto 3
Guardar objeto 4
Terna: [1, 0, 1, 1]

Beneficios y pesos de los objetos guardados
Beneficio: $ 88
Peso: 22
>>>

```

Figura 26 - Primer ejemplo de la mochila entera

Para el segundo ejemplo, aumentamos más nuestros valores, en lo que respecta al tamaño de los arreglos, los beneficios y la capacidad de la mochila, mientras que en los pesos de los objetos utilizamos valores relativamente pequeños y nuevamente en la figura 27 se muestra la solución óptima de la mochila entera.



```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(6,50,100), crearLista(6,0,15), 40)
Beneficios: [95, 55, 90, 97, 61, 54]
Pesos: [4, 4, 0, 15, 10, 12]
Peso mochila: 40

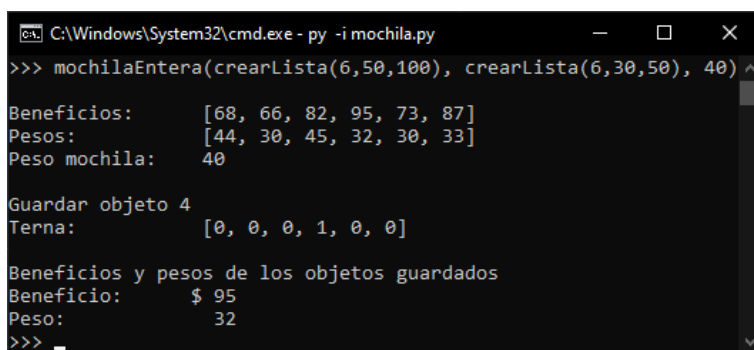
Guardar objeto 1
Guardar objeto 2
Guardar objeto 3
Guardar objeto 4
Guardar objeto 5
Terna: [1, 1, 1, 1, 1, 0]

Beneficios y pesos de los objetos guardados
Beneficio: $ 398
Peso: 33
>>>

```

Figura 27 - Segundo ejemplo de la mochila entera

En el tercer ejemplo decidimos utilizar valores similares, y lo único que variamos fue el tamaño de los pesos de los objetos, donde utilizamos un rango que pudiese pasar la capacidad de la mochila, y como se ve en la figura 28, el algoritmo toma el objeto con mayor beneficio, por lo que continua obteniendo la solución óptima.



```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(6,50,100), crearLista(6,30,50), 40)
Beneficios: [68, 66, 82, 95, 73, 87]
Pesos: [44, 30, 45, 32, 30, 33]
Peso mochila: 40

Guardar objeto 4
Terna: [0, 0, 0, 1, 0, 0]

Beneficios y pesos de los objetos guardados
Beneficio: $ 95
Peso: 32
>>>

```

Figura 28 - Tercer ejemplo de la mochila entera

Para el cuarto ejemplo decidimos bajar la cantidad de elementos de nuestras listas para observar el funcionamiento del algoritmo más a fondo, es decir, para poder observar como se crea la tabla con un ejemplo distinto al de la diapositiva. Primeramente creamos y asignamos listas aleatorias así como es mostrado en la figura 29


```

C:\Windows\System32\cmd.exe - p...
>>> mochilaEntera(b, w, P)
Beneficios: [64, 65, 37, 50, 62]
Pesos: [11, 19, 13, 12, 19]
Peso mochila: 30

Guardar objeto 1
Guardar objeto 2
Terna: [1, 1, 0, 0, 0]

Beneficios y pesos de los objetos guardados
Beneficio: $ 129
Peso: 30
>>>

```

Figura 32 - Cuarto ejemplo de la mochila entera: Función *mochilaEntera*

En nuestro quinto ejemplo continuamos observando las soluciones óptimas que nuestro algoritmo da, establecimos 10 elementos en nuestros arreglos y variamos significativamente los valores de los beneficios de los objetos, mientras que los pesos los mantuvimos más cerrados unos con otros y el peso de la mochila lo establecimos en 60. En la figura 33 vemos el resultado de este ejemplo.

```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(10,10,200), crearLista(10,15,25), 60)
Beneficios: [179, 171, 193, 157, 138, 90, 43, 121, 192, 164]
Pesos: [24, 17, 22, 20, 16, 21, 25, 24, 22, 16]
Peso mochila: 60

Guardar objeto 3
Guardar objeto 9
Guardar objeto 10
Terna: [0, 0, 1, 0, 0, 0, 0, 0, 1, 1]

Beneficios y pesos de los objetos guardados
Beneficio: $ 549
Peso: 60
>>>

```

Figura 33 - Quinto ejemplo de la mochila entera

Posteriormente duplicamos nuestros valores, para el tamaño de arreglos, mantuvimos un rango considerablemente amplio para los beneficios de las mochilas y de los pesos. Obtuvimos el peso exacto y el beneficio máximo en nuestra solución, esto es mostrado en la figura 34.

```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(20,200,300), crearLista(20,30,60), 100)
Beneficios: [204, 210, 277, 228, 245, 243, 259, 269, 203, 272, 255, 214, 235, 202, 244, 202, 210, 233, 292, 296]
Pesos: [35, 43, 47, 47, 47, 35, 41, 41, 43, 57, 56, 58, 40, 44, 50, 44, 38, 30, 48, 59]
Peso mochila: 100

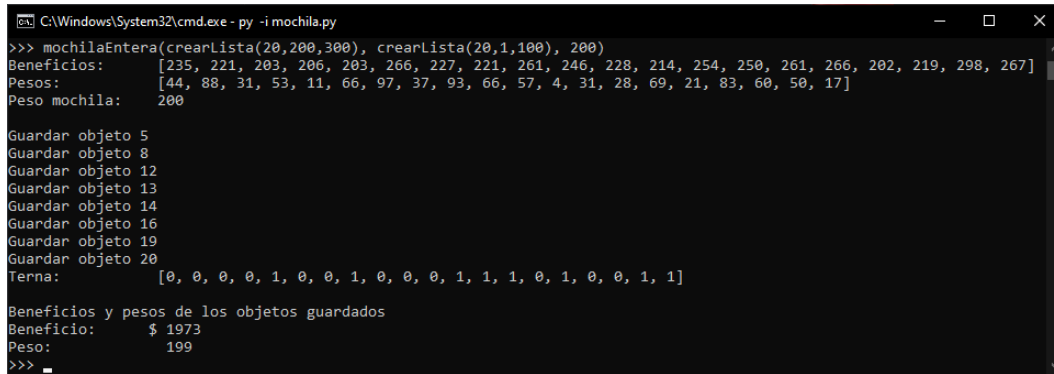
Guardar objeto 1
Guardar objeto 6
Guardar objeto 18
Terna: [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]

Beneficios y pesos de los objetos guardados
Beneficio: $ 680
Peso: 100
>>>

```

Figura 34 - Sexto ejemplo de la mochila entera

Ya que tuvimos un número pequeño de objetos, comparado con el número de elementos, decidimos variar más el rango de nuestros pesos y duplicar la capacidad de la mochila y obtuvimos efectivamente más objetos y por ende más beneficios, como se muestra en la figura 35.



```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(20,200,300), crearLista(20,1,100), 200)
Beneficios: [235, 221, 203, 206, 203, 266, 227, 221, 261, 246, 228, 214, 254, 250, 261, 266, 202, 219, 298, 267]
Pesos: [44, 88, 31, 53, 11, 66, 97, 37, 93, 66, 57, 4, 31, 28, 69, 21, 83, 60, 50, 17]
Peso mochila: 200

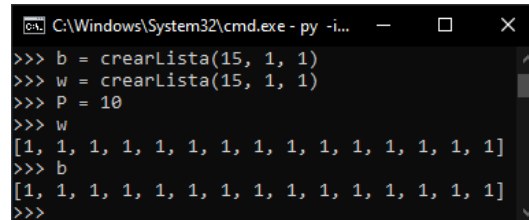
Guardar objeto 5
Guardar objeto 8
Guardar objeto 12
Guardar objeto 13
Guardar objeto 14
Guardar objeto 16
Guardar objeto 19
Guardar objeto 20
Terna: [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1]

Beneficios y pesos de los objetos guardados
Beneficio: $ 1973
Peso: 199
>>>

```

Figura 35 - Séptimo ejemplo de la mochila entera

En nuestro octavo ejemplo decidimos retomar una pregunta cuestionada en la práctica 5, lo que ocurre cuando los beneficios y pesos tienen el mismo valor. Primero, creamos nuestras listas con valores iguales y al peso le asignamos un 10 así como es mostrado en la figura 36.



```

C:\Windows\System32\cmd.exe - py -i...
>>> b = crearLista(15, 1, 1)
>>> w = crearLista(15, 1, 1)
>>> P = 10
>>> w
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> b
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>

```

Figura 36 - Octavo ejemplo de la mochila entera: Asignación de listas con elementos iguales

Y luego pasamos a observar como es que se genera la tabla auxiliar con mismos beneficios y pesos, en donde podemos observar algo bastante peculiar, la fila 0 como en cualquier otro caso esta llena de ceros, luego en la fila vemos que se llena la tabla del 0 al 1 hasta la columna 10, empezando por el 1 en la columna 1, para la fila 2 tenemos del 0 al 2 hasta la columna 2 y las demás columnas se llenan con 2, para la fila 3 se tiene del 0 al 3 hasta la columna 3 y las demás columnas se llenan con 3 y así sucesivamente hasta llegar a la fila 11, donde de ahí hasta la fila 15 (que es igual al peso) se llena del 0 al 10, si tuviéramos más peso, por ejemplo 12, este fenómeno ocurriría hasta la fila 12 y así para distintos pesos o números de elementos, lo cual resulta bastante distintivo a las demás tablas que habíamos estado generando, donde solíamos ver muchos ceros. Todo esto se muestra en la figura 37.

```

C:\Windows\System32...
>>> g = generarTabla(w, b, P)
>>> for i in range(len(g)):
...     print(g[i])
...
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 1, 2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
[0, 1, 2, 3, 4, 4, 4, 4, 4, 4]
[0, 1, 2, 3, 4, 5, 5, 5, 5, 5]
[0, 1, 2, 3, 4, 5, 6, 6, 6, 6]
[0, 1, 2, 3, 4, 5, 6, 7, 7, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>

```

Figura 37 - Octavo ejemplo de la mochila entera: Generación de la tabla g

En seguida, vemos que el algoritmo sigue realizando lo que se le pide, encontrar la solución óptima, aún teniendo mismos valores en los elementos de los arreglos de pesos y beneficios, y toma los primeros de ellos, tal como se muestra en la figura 38.

```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(b, w, P)
Beneficios: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Pesos: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Peso mochila: 10

Guardar objeto 1
Guardar objeto 2
Guardar objeto 3
Guardar objeto 4
Guardar objeto 5
Guardar objeto 6
Guardar objeto 7
Guardar objeto 8
Guardar objeto 9
Guardar objeto 10
Terna: [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]

Beneficios y pesos de los objetos guardados
Beneficio: $ 10
Peso: 10
>>>

```

Figura 38 - Octavo ejemplo de la mochila entera: Función *mochilaEntera*

En los 2 últimos ejemplos aumentaremos el tamaño del número de elementos de nuestras listas así como los valores que tenemos dentro de ellos y de igual forma la capacidad de la mochila para observar como se comporta nuestro algoritmo. Para el ejemplo número 9 tenemos un arreglo de beneficios que varía de los 200-300 en su valor, mientras que para los pesos varían de los 50-150 con 25 de tamaño en cada arreglo y la capacidad de la mochila está dada por 700, los resultados se muestran en la figura 39


```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(25,100,300), crearLista(25,50,150), 700)
Beneficios: [155, 143, 183, 179, 164, 173, 187, 291, 119, 238, 267, 163, 223, 165, 100, 191, 102, 166, 176, 198, 231, 108, 274, 149, 199]
Pesos: [91, 104, 54, 55, 76, 147, 69, 73, 116, 121, 58, 138, 90, 75, 108, 80, 80, 93, 108, 67, 133, 102, 138, 58, 117]
Peso mochila: 700

Guardar objeto 3
Guardar objeto 4
Guardar objeto 5
Guardar objeto 7
Guardar objeto 8
Guardar objeto 11
Guardar objeto 13
Guardar objeto 14
Guardar objeto 16
Guardar objeto 20
Terna: [0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]

Beneficios y pesos de los objetos guardados
Beneficio: $ 2048
Peso: 697
>>>

```

Figura 39 - Noveno ejemplo de la mochila entera

Por último incrementamos a 30 el tamaño de nuestros arreglos y el rango de los valores de los beneficios van del 100-900, para los pesos van del 25-250, la capacidad de la mochila fue de 1000. Con todos estos ejemplos vimos que el beneficio no importa mucho que tanto varíe, pero los pesos son donde se determina que tantos objetos puede llevar la mochila y es donde entra la utilidad de este algoritmo, que en todos los ejemplos dados, siempre nos arroja la solución óptima. La figura 40 nos muestra el último ejemplo de este algoritmo de la mochila

```

C:\Windows\System32\cmd.exe - py -i mochila.py
>>> mochilaEntera(crearLista(25,100,300), crearLista(25,50,150), 700)
Beneficios: [155, 143, 183, 179, 164, 173, 187, 291, 119, 238, 267, 163, 223, 165, 100, 191, 102, 166, 176, 198, 231, 108, 274, 149, 199]
Pesos: [91, 104, 54, 55, 76, 147, 69, 73, 116, 121, 58, 138, 90, 75, 108, 80, 80, 93, 108, 67, 133, 102, 138, 58, 117]
Peso mochila: 700

Guardar objeto 3
Guardar objeto 4
Guardar objeto 5
Guardar objeto 7
Guardar objeto 8
Guardar objeto 11
Guardar objeto 13
Guardar objeto 14
Guardar objeto 16
Guardar objeto 20
Terna: [0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]

Beneficios y pesos de los objetos guardados
Beneficio: $ 2048
Peso: 697
>>>

```

Figura 40 - Décimo ejemplo de la mochila entera

3.3. Algoritmo de Líneas de Producción

Por último, en esta práctica, estudiaremos el algoritmo que nos permite resolver el problema de obtener el mejor tiempo en una línea de producción. Se mostrarán algunos ejemplos y observaremos como se resuelven estas problemáticas mediante programación dinámica

3.3.1. Pseudocódigos del algoritmo de líneas de producción

Primero que todo, mostraremos el pseudocódigo en el que nos basaremos para posteriormente implementar el código en Python.

El primer pseudocódigo nos permite generar 2 tablas, en una de ellas podemos saber que recorrido es el mejor acorde a los tiempos que se tienen en todas las estaciones de trabajo y en aquellas que también nos permiten cambiarnos de una línea a la otra; mientras que en la otra tabla se nos dan los índices de las estaciones de donde provino para obtener esa solución óptima. La denominamos *generarTablas* y se muestra a continuación.

```

generarTablas(n, a[1,2][1..,n], t[1,2][1..,n], e[1,2], x[1,2])
1-  f[1][1] = e[1] + a[1][1]
2-  f[2][1] = e[2] + a[2][1]
3-  for j = 2 to n do
4-      if f[1][j-1]+a[1][j]<=f[2][j-1]+t[2][j-1]+a[1][j]
5-          f[1][j] = f[1][j-1] + a[1][j]
6-          I[1][j] = 1
7-      else
8-          f[1][j] = f[2][j-1] + t[2][j-1] + a[1][j]
9-          I[1][j] = 2
10-     if f[2][j-1]+a[2][j]<=f[1][j-1]+t[2][j-1]+a[2][j]
11-         f[2][j] = f[2][j-1] + a[2][j]
12-         I[2][j] = 2
13-     else
14-         f[2][j] = f[1][j-1] + t[1][j-1] + a[2][j]
15-         I[2][j] = 1
16-     if f[1][n] + x[1] <= f[2][n] + x[2]
17-         f* = f[1][n] + x[1]
18-         I* = 1
19-     else
20-         f* = f[2][n] + x[2]
21-         I* = 2
23-     return f, f*, I, I*

```

Por otro lado, el otro algoritmo es para imprimir el recorrido que se debe

hacer para tener nuestra solución óptima. A esta función la denominamos *imprimirLineaProduccion*.

```
imprimirLineaProduccion(n, I, I*)
1- i = I*
2- print "l nea", i, "estaci n", n
3- for j = n downto 2
4-     i = I[i][j]
5-     print "l nea", i, "estaci n", j-1
```

3.3.2. Implementación y ejemplo de la diapositiva

A continuación implementamos el pseudocódigo antes mencionado y verificamos que estuviese dando la solución óptima con apoyo del ejemplo dado en las diapositivas.

Primeramente empezamos por implementar la función de *generarTablas*, donde solo tuvimos que acomodar los índices de nuestro pseudocódigo para que pudiese trabajar correctamente este algoritmo. Declaramos las variables para las estaciones, que en este caso fue una matriz de 2×6 y del mismo modo la variable para el tiempo de traslado de línea a línea que en este caso fue una matriz de 2×5 , también se declararon los arreglos de tamaño 2 para los tiempos de inicio y los tiempos finales, todas estas variables sacadas del ejemplo de la diapositiva. En la figura 41 mostramos que en efecto, esta función nos genera las tablas que son mostradas en las diapositivas, por lo que nuestro código fue implementado correctamente.

```

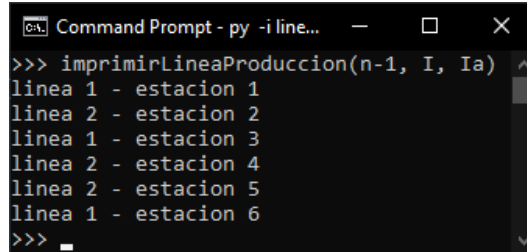
C:\Users\52556\aa>py -i lineaPro...
C:\Users\52556\aa>py -i lineaProduccion.py
>>> f,fa,I,Ia = generarTablas(n,a,t,e,x)
>>> for i in range(len(f)):
...     print(f[i])
...
[9, 18, 20, 24, 32, 35]
[12, 16, 22, 25, 30, 37]
>>> fa
38
>>> for i in range(len(I)):
...     print(I[i])
...
[1, 2, 1, 1, 2]
[1, 2, 1, 2, 2]
>>> Ia
1
>>>

```

Figura 41 - Resultado en consola de la función *generarTablas*

En seguida, implementamos la función para imprimir las líneas de producción, pero hicimos un pequeño cambio a lo propuesto en el pseudocódigo, en lugar de usar un ciclo *for*, hicimos la función recursiva, de modo que al momento

de imprimir el recorrido que se tiene que hacer lo imprima de inicio a fin y no de fin a inicio como se muestra en la diapositiva. La figura 42 nos muestra el funcionamiento de *imprimirLineaProduccion*



```

C:\> Command Prompt - py -i line...
>>> imprimirLineaProduccion(n-1, I, Ia)
linea 1 - estacion 1
linea 2 - estacion 2
linea 1 - estacion 3
linea 2 - estacion 4
linea 2 - estacion 5
linea 1 - estacion 6
>>>

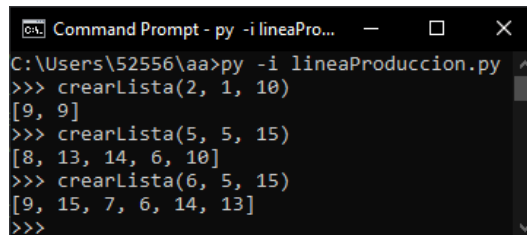
```

Figura 42 - Resultado en consola de la función *imprimirLineaProduccion*

3.3.3. Ejemplos

Para realizar ejemplos lo más arbitrarios que se puedan y a su vez mostrar el completo funcionamiento de este algoritmo, tendremos que implementar algunas funciones que nos permitan crear listas y matrices $2 \times n$ aleatorias y así como darles un rango a los elementos que aparezcan dentro de estas.

La primera de ellas ya la hemos implementado en multiples ocasiones, la denominamos *crearLista* y recibe como parametros: el tamaño de la lista, límite inferior del rango, límite superior del rango. Su funcionamiento se muestra en la figura 43.



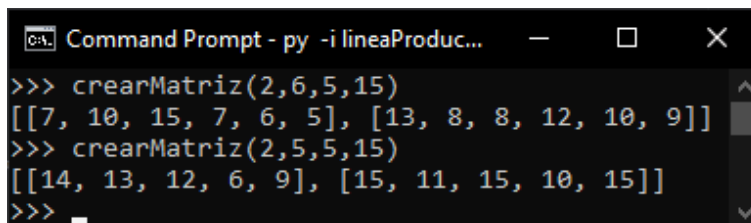
```

C:\Users\52556\aa>py -i lineaProduccion.py
>>> crearLista(2, 1, 10)
[9, 9]
>>> crearLista(5, 5, 15)
[8, 13, 14, 6, 10]
>>> crearLista(6, 5, 15)
[9, 15, 7, 6, 14, 13]
>>>

```

Figura 43 - Resultado en consola de la función *crearLista*

Seguidamente, creamos la función *crearMatriz* y se recibe los parametros del número de filas y el número de columnas y el rango de los números que llevará dentro. En la figura 44 vemos que nos regresa arreglos dentro de otro arreglo, que es otra manera en la que se puede interpretar una matriz.



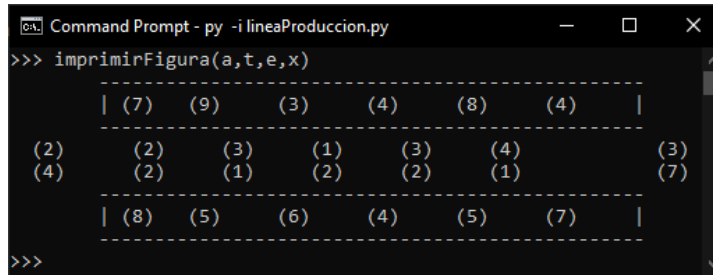
```

C:\> Command Prompt - py -i lineaProduc...
>>> crearMatriz(2,6,5,15)
[[7, 10, 15, 7, 6, 5], [13, 8, 8, 12, 10, 9]]
>>> crearMatriz(2,5,5,15)
[[14, 13, 12, 6, 9], [15, 11, 15, 10, 15]]
>>>

```

Figura 44 - Resultado en consola de la función *crearMatriz*

Posterior a esto, para visualizar mejor nuestros ejemplos, desarrollamos una función que solamente se encarga de mostrar los valores de nuestras matrices y listas de manera parecida a lo que vimos en las diapositivas, la denominamos *imprimirFigura* y la figura 45 nos muestra como funciona con el ejemplo de la diapositiva.



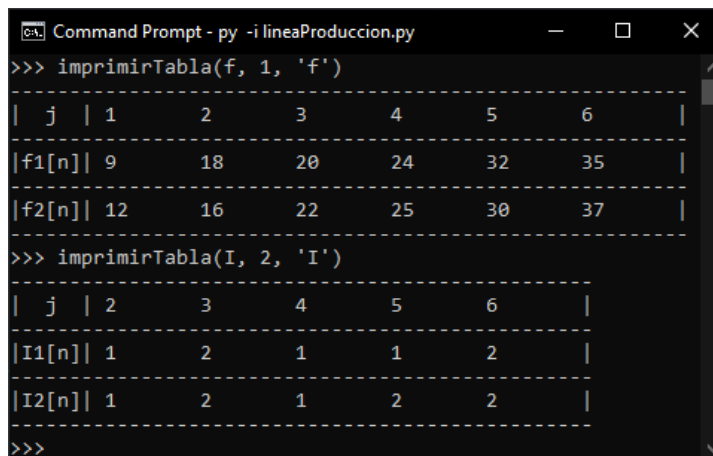
```

>>> imprimirFigura(a,t,e,x)
  | (7) (9) (3) (4) (8) (4) |
  |-----|
(2) | (2) (3) (1) (3) (4) (3) |
(4) | (2) (1) (2) (2) (1) (7) |
  |-----|
  | (8) (5) (6) (4) (5) (7) |
  |-----|
>>>

```

Figura 45 - Resultado en consola de la función *imprimirFigura*

Continuando con funciones que permitan una mejor visualización, implementamos *imprimirTabla* que nos imprime de una mejor manera las matrices, esta función únicamente recibe la matriz, el incremento de los índices de las cabeceras de las columnas y el caracter de las cabeceras de las filas, en la figura 46 vemos como funciona más claramente.



```

>>> imprimirTabla(f, 1, 'f')
  | j | 1 2 3 4 5 6 |
  |-----|
|f1[n]| 9 18 20 24 32 35 |
  |-----|
|f2[n]| 12 16 22 25 30 37 |
  |-----|

>>> imprimirTabla(I, 2, 'I')
  | j | 2 3 4 5 6 |
  |-----|
|I1[n]| 1 2 1 1 2 |
  |-----|
|I2[n]| 1 2 1 2 2 |
  |-----|
>>>

```

Figura 46 - Resultado en consola de la función *imprimirTabla*

Todas estas funciones que se describieron las juntamos en una última función denominada *lineaprod* que recibe un número n que representa el número de estaciones, un número x para el límite inferior de los tiempos de estación y un número y para el límite superior de los tiempos de estación. Esta función será la que nos permita crear ejemplos lo más arbitrarios posibles y visualizar la información de la salida de nuestros algoritmos de una mejor manera. De igual manera implementamos una función *lineaprod2* que permite hacer lo mismo que *lineaprod* pero recibiendo las matrices y listas. En la figura 47 vemos el funcionamiento de *lineaprod2* con el ejemplo de la diapositiva

```

Command Prompt - py -i lineaProduccion.py
C:\Users\52556\aa>py -i lineaProduccion.py
>>> lineaprod2(n, a, t, e, x)

  (7) (9) (3) (4) (8) (4)
  -----
(2) (2) (3) (1) (3) (4) (3)
(4) (2) (1) (2) (2) (1) (7)
  -----
  (8) (5) (6) (4) (5) (7)
  -----
| j | 1 | 2 | 3 | 4 | 5 | 6 |
|f1[n]| 9 | 18 | 20 | 24 | 32 | 35 |
|f2[n]| 12 | 16 | 22 | 25 | 30 | 37 |
  -----
f* = 38

  -----
| j | 2 | 3 | 4 | 5 | 6 |
|I1[n]| 1 | 2 | 1 | 1 | 2 |
|I2[n]| 1 | 2 | 1 | 2 | 2 |
  -----
I* = 1

linea 1 - estacion 1
linea 2 - estacion 2
linea 1 - estacion 3
linea 2 - estacion 4
linea 2 - estacion 5
linea 1 - estacion 6
>>>

```

Figura 47 - Resultado en consola de la función *lineaprod2*

Por lo que para el primer ejemplo, decidimos usar una n no tan grande y los tiempos de igual manera fueron con un rango no tan amplio, esto para observar que efectivamente este calculando la solución óptima. El ejemplo que se generó, mostrado en la figura 48, nos muestra que se debe recorrer toda la línea 2 y si analizamos un poco, en efecto, esta es la solución óptima, porque al parecer la línea 1 también tiene tiempos cortos, pero la línea 2 es la que tiene el tiempo mínimo.

```

Command Prompt - py -i l...
>>> lineaprod(3, 1, 5)

  (2) (1) (2)
  -----
(2) (5) (5) (4)
(1) (1) (3) (2)
  -----
  (2) (1) (3)
  -----
| j | 1 | 2 | 3 |
|f1[n]| 4 | 5 | 7 |
|f2[n]| 3 | 4 | 7 |
  -----
f* = 9

  -----
| j | 2 | 3 |
|I1[n]| 1 | 1 |
|I2[n]| 2 | 2 |
  -----
I* = 2

linea 2 - estacion 1
linea 2 - estacion 2
linea 2 - estacion 3
>>>

```

Figura 48 - Primer ejemplo del algoritmo de las líneas de producción

A partir de lo obtenido en consola, facilmente podemos realizar una figura más gráfica que nos muestre la ruta óptima, tal como lo muestra la figura 49.

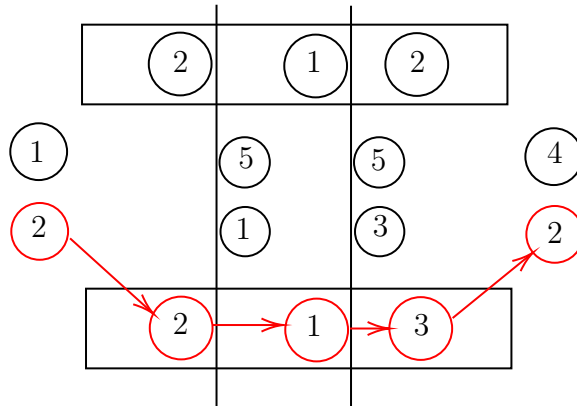


Figura 49 - Ruta óptima del primer ejemplo

En el segundo ejemplo aumentamos un poco el número de estaciones y el rango de tiempo de estas mismas. Coincidio que en este ejemplo la ruta a seguir se da por solo una línea, pero en este caso es por la línea 1 donde obtenemos el mejor tiempo y nuevamente si observamos a detalle esta solución es la óptima. Este ejemplo es mostrado en la figura 50.

```

Command Prompt - py -i lineaProduccion.py
>>> lineaprod(5, 2, 8)
-----
| (6)  (7)  (2)  (4)  (8)  |
-----
(3)    (7)  (3)  (2)  (4)    (4)
(6)    (7)  (8)  (5)  (2)    (6)
-----
| (4)  (2)  (6)  (7)  (6)  |
-----
-----
| j | 1  2  3  4  5  |
-----
|f1[n]| 9 16 18 22 30 |
-----
|f2[n]| 10 12 18 25 31 |
-----
f* = 34
-----
| j | 2  3  4  5  |
-----
|I1[n]| 1  1  1  1  |
-----
|I2[n]| 2  2  2  2  |
-----
I* = 1
linea 1 - estacion 1
linea 1 - estacion 2
linea 1 - estacion 3
linea 1 - estacion 4
linea 1 - estacion 5
>>>

```

Figura 50 - Segundo ejemplo del algoritmo de las líneas de producción

Con lo anterior, podemos concluir lo que se muestra en la figura 51.

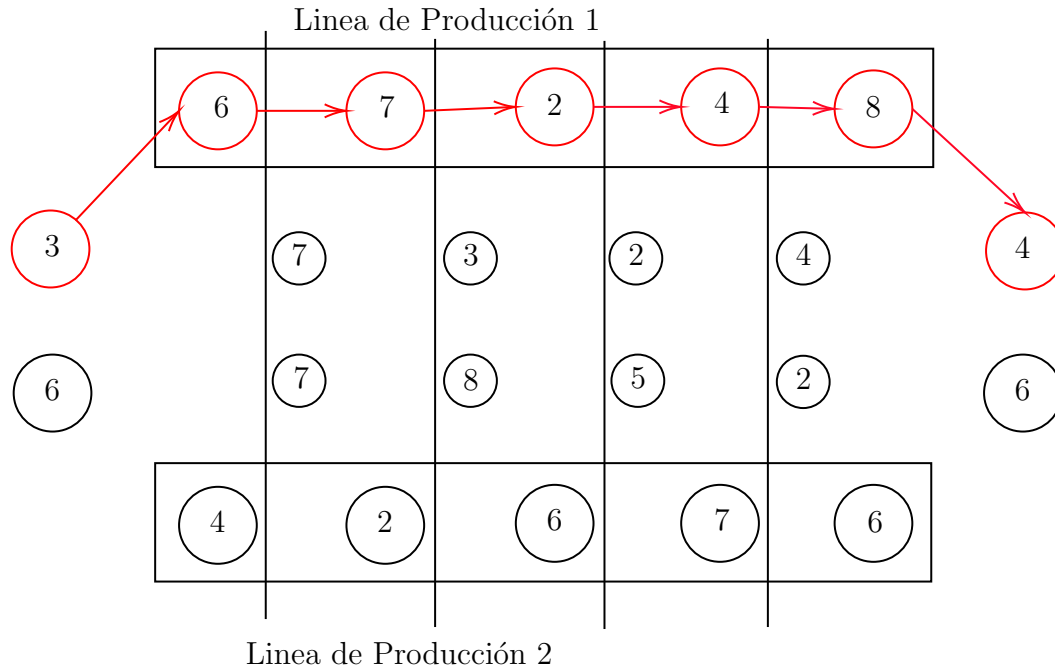


Figura 51 - Ruta óptima del segundo ejemplo

El tercer ejemplo decidimos hacerlo algo parecido al mostrado en la diapositiva, pero este tomará valores aleatorios en sus listas y matrices. En la figura 52 observamos que ahora sí hubo un intercambio de línea ocurrido en la estación 3, pero vemos que prácticamente la mitad del recorrido se mantuvieron en una misma línea. Aún así seguimos obteniendo las soluciones óptimas a los problemas que se han estado generando.

```

Command Prompt - py -i lineaProduccion.py
>>> lineaprod(6,2,9)

  (5) (4) (7) (8) (5) (3)
-----
(2) (8) (2) (4) (3) (8) (9)
(8) (5) (6) (3) (6) (6) (4)
-----
  (6) (9) (5) (4) (7) (2)

j | 1 | 2 | 3 | 4 | 5 | 6 |
f1[n] | 7 | 11 | 18 | 26 | 31 | 34 |
f2[n] | 14 | 23 | 18 | 22 | 29 | 31 |
f* = 35

j | 2 | 3 | 4 | 5 | 6 |
l1[n] | 1 | 1 | 1 | 1 | 1 |
l2[n] | 2 | 1 | 2 | 2 | 2 |
l* = 2

Linea 1 - estacion 1
Linea 1 - estacion 2
Linea 2 - estacion 3
Linea 2 - estacion 4
Linea 2 - estacion 5
Linea 2 - estacion 6
>>>

```

Figura 52 - Tercer ejemplo del algoritmo de las líneas de producción

La figura 53 nos muestra esta solución óptima obtenida del tercer ejemplo.

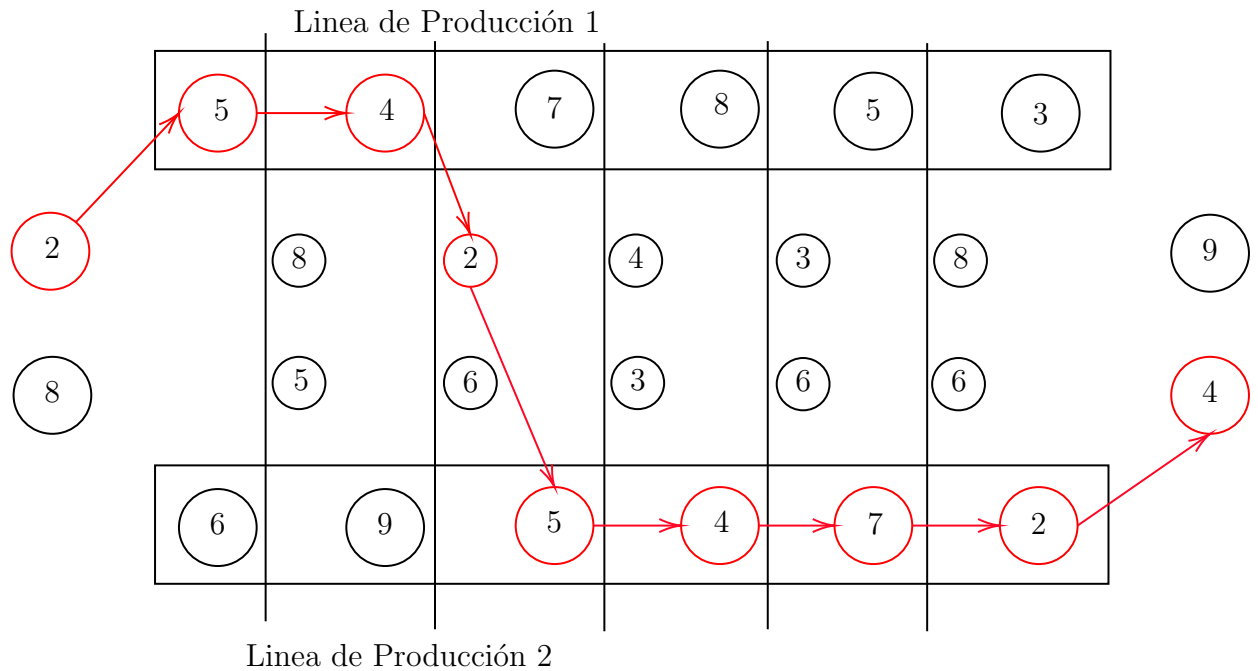


Figura 53 - Ruta óptima del tercer ejemplo

En nuestro penúltimo ejemplo aumentamos todo de manera significativa, el número de estaciones y el rango lo variamos más. Observamos que con un rango más variado ocurren un poco más de transiciones de línea a línea, pero sigue tendiendo un poco a mantenerse en una misma línea. La figura 54 nos muestra todo esto.

```

Command Prompt - py -i:linesProduction.py
>>> lineaprod(15, 5, 75)
| (71) (63) (72) (69) (16) (65) (24) (57) (26) (54) (49) (56) (42) (15) (9) |
(10) (37) (18) (11) (34) (5) (72) (74) (71) (44) (52) (64) (21) (32) (41) (38)
(49) (28) (39) (13) (10) (74) (13) (7) (22) (36) (67) (52) (69) (78) (5)
| (5) (7) (37) (48) (44) (33) (7) (19) (27) (18) (39) (25) (72) (43) (59) |
+-----+
| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
+-----+
| f1[n] | 81 | 137 | 172 | 188 | 172 | 237 | 245 | 281 | 284 | 338 | 387 | 428 | 439 | 454 | 463 |
+-----+
| f2[n] | 54 | 61 | 98 | 146 | 190 | 218 | 217 | 236 | 263 | 281 | 328 | 345 | 417 | 460 | 519 |
+-----+
f* = 493
+-----+
| j | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
+-----+
| l1[n] | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 1 | 1 | 2 | 2 | 1 | 1 |
+-----+
| l2[n] | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
+-----+
f* = 1
línea 2 - estación 1
línea 2 - estación 2
línea 2 - estación 3
línea 2 - estación 4
línea 1 - estación 5
línea 2 - estación 6
línea 2 - estación 7
línea 2 - estación 8
línea 2 - estación 9
línea 2 - estación 10
línea 2 - estación 11
línea 2 - estación 12
línea 1 - estación 13
línea 1 - estación 14
línea 1 - estación 15
>>>

```

Figura 54 - Cuarto ejemplo del algoritmo de las líneas de producción

Y por último para este ejemplo generamos su ruta óptima mostrada en la figura 55.

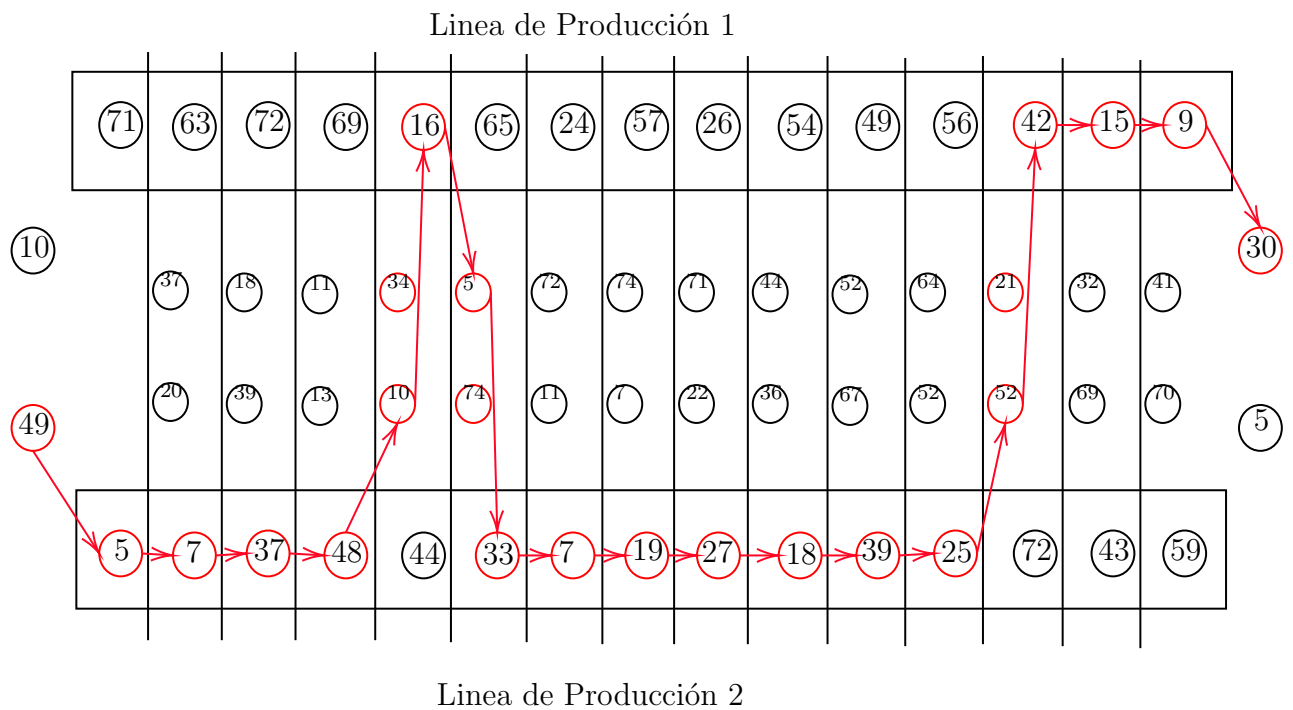


Figura 55 - Ruta óptima del cuarto ejemplo

3.3.4. Ejercicio de la diapositiva

El ejercicio presentado en al final de las diapositivas es el que se muestra en la figura 56. La resolución de este problema a detalle se muestra en el anexo, donde se tendrá que llegar a la misma solución que el problema nos arroje.

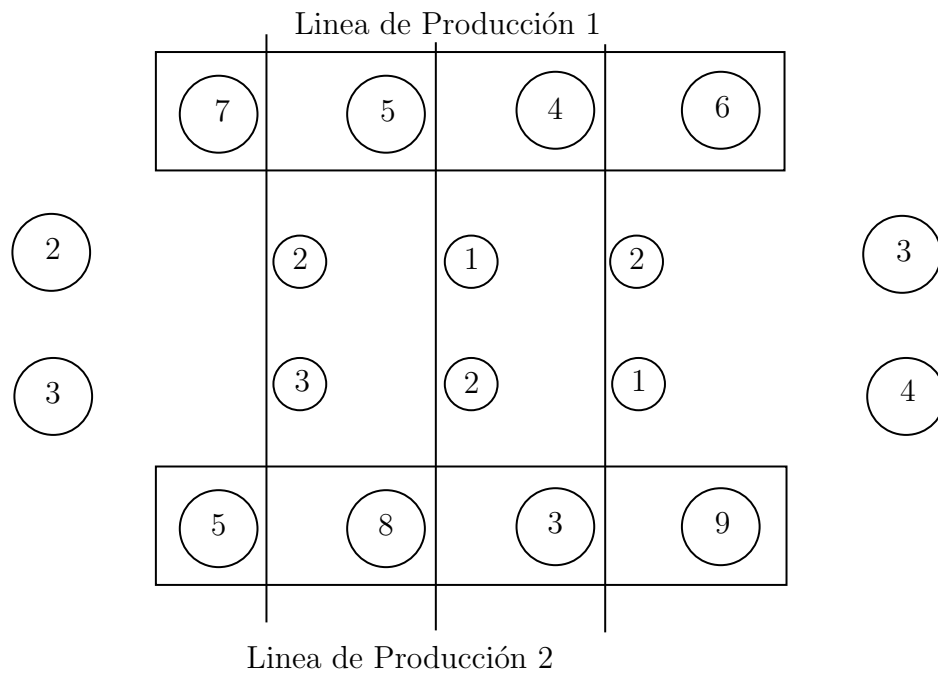


Figura 56 - Diagrama ejercicio Lineas de producción

Por lo que empezamos declarando las variables a utilizar en nuestro código, tal como se muestra en la figura 57.

```

C:\Users\52556\aa>py -i lineaPro...
>>> a = []
>>> a.append([7,5,4,6])
>>> a.append([5,8,3,9])
>>> n = len(a[0])
>>>
>>> t = []
>>> t.append([2,1,2])
>>> t.append([3,2,1])
>>>
>>> e = [2,3]
>>> x = [3,4]
>>>

```

Figura 57 - Declaracion de variables del ejercicio

Posterior a esto, simplemente usamos nuestra función *lineaprod2* y le mandamos las variables que recién declaramos. La solución óptima a este problema se muestra en la figura 58.

```
Command Prompt - py -i lineaProduc...
>>> lineaprod2(n, a, t, e, x)
-----
| (7)  (5)  (4)  (6)  |
-----
(2)      (2)      (1)      (2)      (3)
(3)      (3)      (2)      (1)      (4)
-----
| (5)  (8)  (3)  (9)  |
-----

| j | 1 | 2 | 3 | 4 |
-----
| f1[n] | 9 | 14 | 18 | 24 |
-----
| f2[n] | 8 | 16 | 18 | 27 |
-----
f* = 27

| j | 2 | 3 | 4 |
-----
| I1[n] | 1 | 1 | 1 |
-----
| I2[n] | 2 | 1 | 2 |
-----
I* = 1

Linea 1 - estacion 1
Linea 1 - estacion 2
Linea 1 - estacion 3
Linea 1 - estacion 4
>>>
```

Figura 58 - Declaracion de variables del ejercicio

Y finalmente de la figura 59 nos podemos apoyar para ver que realmente es la solución óptima a este problema, que ciertamente coincide con la solución también obtenida en el anexo.

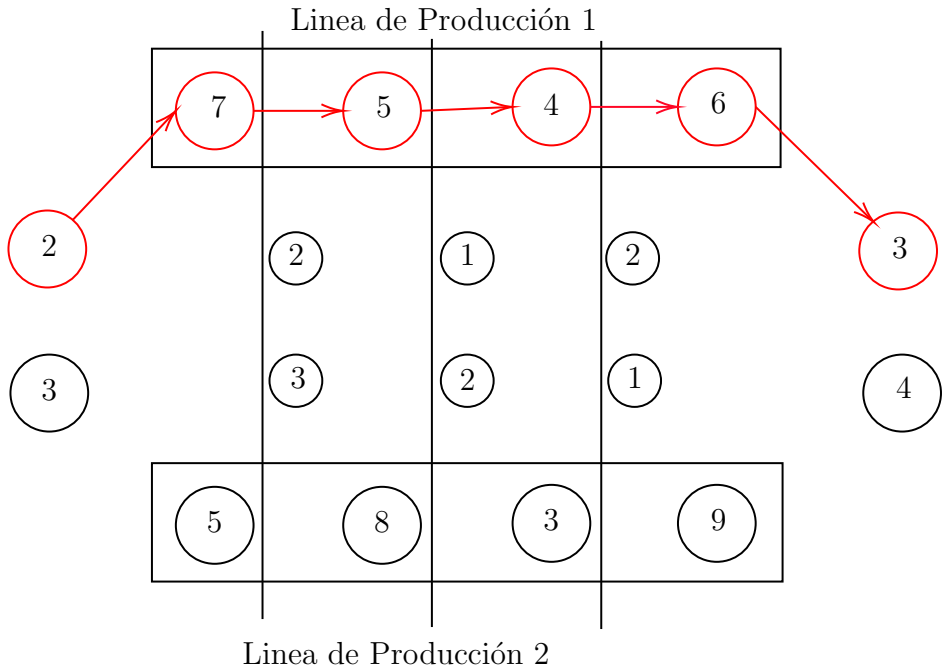


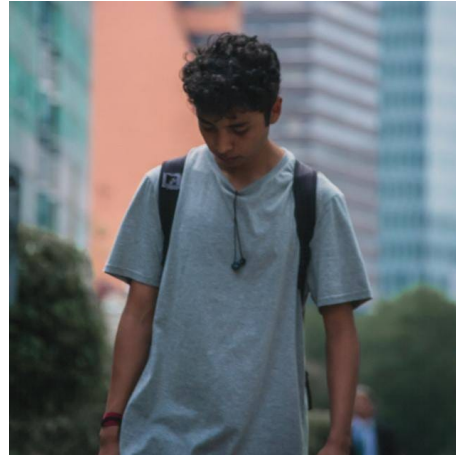
Figura 59 - Ruta optima

4. Conclusiones

Eduardo Mendoza Martínez

Al término de la presente práctica fui capaz de comprender el funcionamiento de la programación dinámica y como esta puede mejorar algunos algoritmos que ya han resuelto diversos problemas, pero con programación dinámica se resuelven de una manera más rápida y eficiente, de igual modo, con este concepto se pueden resolver nuevas problemáticas tales como las que vimos en esta práctica con las *Líneas de producción*. Le encontré cierto parecido a lo que se maneja en el paradigma orientado a objetos, ya que en él, se maneja una idea de que "si ya existe, úsalo" y es

lo mismo que podemos observar en el algoritmo de *Fibonacci*, en el cual observamos que gran parte de los números de esta sucesión son almacenados y con ellos posteriormente podemos encontrar aún más, o de manera más rápida encontrar alguno que ya se había encontrado directa o indirectamente. En los demás algoritmos hicimos mucho uso de tablas (o matrices), no fueron complicadas de implementar, sin embargo personalmente no había trabajado tanto con arreglos bidimensionales y fue un poco complicado a la hora de obtener índices, desplegar la tabla, pero al final pudimos presentar nuestros datos y las resoluciones a los problemas sin ningún percance. Conocer nuevas formas de resolver problemas es esencial para nosotros como próximos ingenieros, ya que con esto podremos encontrar solución a nuevos problemas o mejorar lo que ya existe, tal como nos lo muestra la programación dinámica.



Daniel Aguilar Gonzalez

El desarrollo de esta práctica me ayudo a entender un poco más a fondo como es que funciona la programación dinámica y como gracias a esta podemos dar solución a algoritmos que ya habíamos desarrollado en prácticas anteriores pero con otro método como fue el caso de fibonacci.

En esta práctica vimos principalmente programación dinámica y como hacer más eficiente la implementación de algunos algoritmos como lo fue fibonacci el cual se implemento con los dos enfoques que maneja la programación dinámica top-down arriba hacia abajo y bottom-up abajo hacia arriba, me ayudo a identificar cual de los dos es más eficiente gracias a un análisis realizado a través de pruebas plasmadas en graficas llegando a la conclusión de que el más eficiente es top-down. La programación dinámica es un método el cual puede tambien utilizar algún otro es decir recursividad, como se vio en los algoritmos anteriores y esto lo hace más eficiente ya que aplica la forma de que si ya calculamos algo una vez no es necesario hacerlo dos veces "Si ya lo tenemos, usemoslo" y eso nos ayuda a no hacer calculos innecesarios, repetir un proceso que ya nos dio un resultado ahorrando asi tiempo y recursos. Implementamos el algoritmo de lineas de producción el cual en lo personal me parecio de mucha importancia ya que se puede aplicar en la vida real porque nos ayuda a encontrar un camino óptimo por el cual debe pasar un producto para producirse de manera más rápida. Se presentó un ejercicio para el cual hicimos su proceso a mano primeramente y despues en el programa implementado coincidiendo en resultados mostrando la misma ruta óptima de producción. Es de vital importancia saber diferentes maneras de atacar un algoritmo para poder determinar cual de todas ellas nos conviene mas implementar para ahorrar tiempo y recursos y hacer más eficiente.



5. Anexo

En esta seccion veremos algunos problemas anexos complementarios de los algoritmos vistos en la practica.

5.1. Lineas de Produccion

Considerando 2 lineas de producci3n donde cada nodo representa una estaci3n de trabajo y cada numero dentro el nodo es el tiempo que tarda en operar esa estaci3n como se muestra en la siguiente figura:

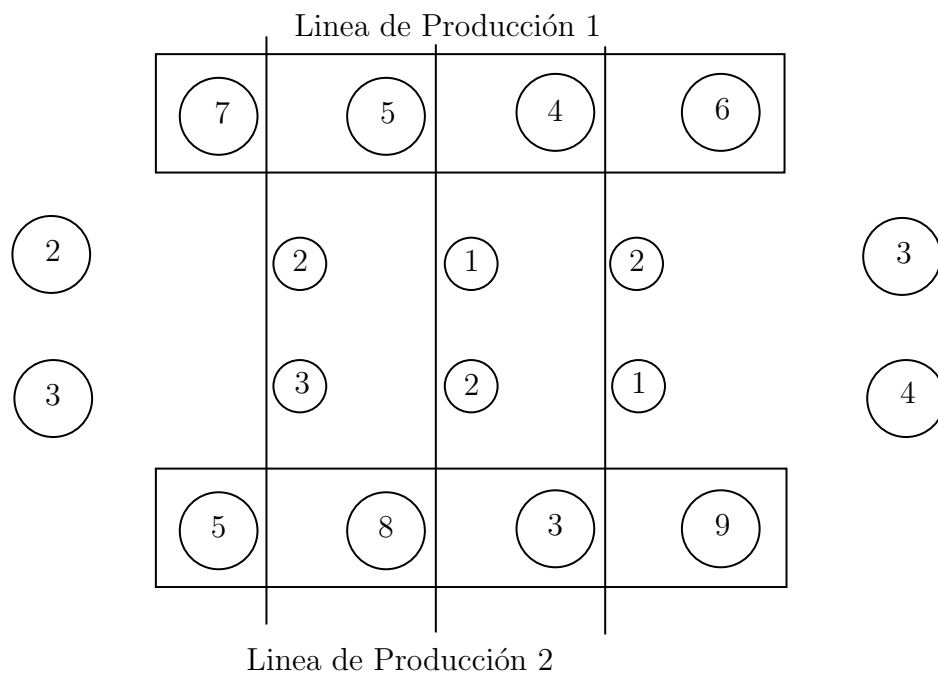


Figura 60 - Diagrama ejercicio Lineas de producci3n

Los nodos que se encuentran al centro de ambas estaciones son los tiempos que tarda en pasar de una estaci3n a otra. El nodo m1s cercano a la linea de producci3n es el tiempo que tarda en pasar de esa linea a la otra.

Los nodos que se encuentran a la izquierda de las lineas son los tiempos que tarda el producto en entrar en ellas y los de la derecha son los tiempos que tarda en salir. Las lineas plasmadas verticalmente marcan las estaciones de trabajo. Mencionado lo anterior procederemos a resolver un ejercicio.

Problema.

Encontrar el camino m1s 3ptimo por el cual debe pasar un producto para su elaboraci3n. Determinando cual es la suma total f^* , cual es la linea m1s 3ptima I^* y el camino que debe seguir el producto dentro de las lineas.

Desarrollo.

Para dar solución a este ejercicio debemos llenar dos tablas una para f^* que nos permite saber la suma óptima que se obtiene de pasar por las estaciones de trabajo y la de I^* que nos permite saber la línea de producción final que se obtuvo.

f^* . Para obtener f^* construiremos una tabla en la cual iremos registrando la suma de los tiempos que se obtienen cada que el producto pasa por una estación de trabajo para ambas líneas.

I^* . Para obtener I^* haremos una tabla en la cual registraremos el número de la línea que viene la suma, es decir al llegar a una estación el producto puede venir de la misma línea o bien de otra.

Comenzaremos con el desarrollo del ejercicio:

Estación 1

- $f1[1]$.

Comenzaremos por analizar la línea 1 donde como podemos observar nuestro primer nodo tiene un valor de 2 y el siguiente un valor de 7 así que procedemos a realizar la suma y el resultado será agregado a la tabla.

- $f2[1]$

Para comenzar en la línea 2 observamos que tenemos el valor de 2 seguido de 5 así que los sumaremos y el resultado será agregado a la tabla.

$$\text{Línea 1. } 2 + 7 = 9$$

$$\text{Línea 2. } 3 + 5 = 8$$

j	1	2	3	4
$f1[n]$	9			
$f2[n]$	8			

Cuadro 1: Estación 1

Estación 2

Para la estación 2 haremos una suma del tiempo obtenido de la estación 1 en la misma línea más el tiempo del nodo de la estación 2 y una suma del tiempo obtenido de la estación 1 de la otra línea más el tiempo de cambio de línea

más el tiempo del nodo de la estación 2, haremos una comparación entre ellos y colocaremos en la tabla el tiempo menor.

Observaremos de qué línea es donde viene el número menor si de la actual o viene de un cambio y se colocará dentro del recuadro el número de la línea que viene.

- $f1[2]$.

Tiempo sobre línea 1 + tiempo de nodo y tiempo sobre línea 2 + tiempo de cambio + tiempo de nodo

$$\begin{aligned} \text{L1. } 9 + 5 &= 14 \\ \text{L2. } 8 + 3 + 5 &= 16 \\ 14 &< 16 = 14 \\ 14 &\text{ proviene de L1} \end{aligned}$$

- $f2[2]$

Tiempo sobre línea 1 + tiempo de cambio + tiempo de nodo y tiempo sobre línea 2 + tiempo de nodo.

$$\begin{aligned} \text{L1. } 9 + 2 + 8 &= 19 \\ \text{L2. } 8 + 8 &= 16 \\ 16 &< 19 = 16 \\ 16 &\text{ proviene de L2} \end{aligned}$$

j	1	2	3	4
$f1[n]$	9	14 1		
$f2[n]$	8	16 2		

Cuadro 2: Estación 2

Estación 3

Para la estación 3 realizaremos el mismo procedimiento que la estación anterior

- $f1[3]$.

Tiempo sobre linea 1 + tiempo de nodo y tiempo sobre linea 2 + tiempo de cambio + tiempo de nodo

$$\text{L1. } 14 + 4 = 18$$

$$\text{L2. } 16 + 2 + 4 = 22$$

$$18 < 22 = 18$$

18 proviene de L1

- f2[3]

Tiempo sobre linea 1 + tiempo de cambio + tiempo de nodo y tiempo sobre linea 2 + tiempo de nodo.

$$\text{L1. } 14 + 1 + 3 = 18$$

$$\text{L2. } 16 + 3 = 19$$

$$18 < 19 = 18$$

18 proviene de L1

j	1	2	3	4
f1[n]	9	14 1	18 1	
f2[n]	8	16 2	18 1	

Cuadro 3: Estacion 3

Estación 4

Para la estación 4 realizaremos el mismo procedimiento que la estación anterior

- f1[4].

Tiempo sobre linea 1 + tiempo de nodo y tiempo sobre linea 2 + tiempo de cambio + tiempo de nodo

$$\text{L1. } 18 + 6 = 24$$

$$\text{L2. } 18 + 1 + 6 = 25$$

$$24 < 25 = 24$$

24 proviene de L1

- $f2[4]$

Tiempo sobre linea 1 + tiempo de cambio + tiempo de nodo y tiempo sobre linea 2 + tiempo de nodo.

$$L1. 18 + 2 + 9 = 29$$

$$L2. 18 + 9 = 27$$

$$27 < 29 = 27$$

27 proviene de L2

j	1	2	3	4
$f1[n]$	9	14 1	18 1	24 1
$f2[n]$	8	16 2	18 1	27 2

Cuadro 4: Estacion 4

Ya finalizamos nuestra tabla, lo siguiente es al resultado de la ultima estacion sumarle los tiempos de salida los cuales se encuentran en los nodos de la derecha del diagrama es decir:

$$f1[n] = 24 + 3 = 27$$

$$f2[n] = 27 + 4 = 31$$

Ahora bien al tener ambos resultados elegiremos el resultado menor y ese sera nuestro valor de f^* , es decir

$$f^* = 27$$

Lo siguiente que debemos hacer es hacer la tabla para poder determinar el valor de I^* . Esta tabla se forma con los numeros que fuimos poniendo dentro de los recuadros pequeños es decir de las lineas que provenian las sumas menores asi que nos queda de la siguiente manera:

j	2	3	4
f1[j]	1	1	1
f2[j]	2	1	2

Cuadro 5: Tabla para I^*

Teniendo nuestra tabla completa analizaremos la columna de la ultima estación es decir la que contiene la estación numero 4 y nuestra tabla anterior (cuadro 4). Como podemos observar en nuestra Tabla 4 el valor menor se encuentra en la linea 1 y en nuestra tabla 5 vemos que proviene de la linea 1 asi que este es el valor que determina la I^* .

$$I^* = 1$$

Obtenidos los valores de I^* y f^* vamos a formar la ruta que permite el tiempo optimo . Analizaremos cada columna de la tabla 4 iniciando de derecha a izquierda comparando los valores de ambas lineas y determinando el menor, al saber cual es el menor iremos a la tabla 5 y encerraremos de que linea proviene ejemplo para la columna 4 el menor es el 24 que se encuentra en linea 1, vamos a tabla 5 y encerramos el numero 1 de la columna 4 y asi sucesivamente. Finalmente nuestra tabla nos queda de esta manera:

j	2	3	4
f1[j]	1	1	1
f2[j]	2	1	2

Cuadro 6: Tabla para I*

Ya identificado de donde proceden los valores menores usaremos el diagrama principal. Nuestro tiempo de salida fue 3 esto quiere decir que proviene de la estacion 4 de la linea 1:

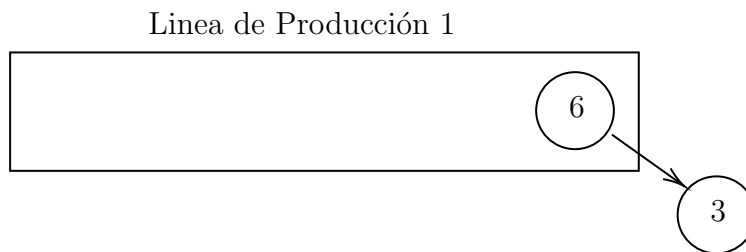


Figura 61 - Union Salida con Estacion 4

El haber encerrado nuestras procedencias nos ayuda para conectar nuestra estacion actual con una anterior. Posicionados en la columna 4 buscamos el numero encerrado y nos damos cuenta que su procedencia viene por Linea 1 como se muestra a continuación.

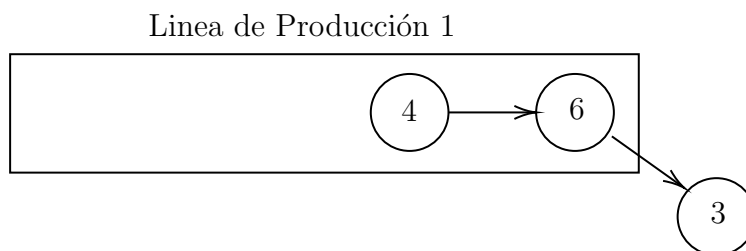


Figura 62 - Procedencia Estacion 4

Haremos el mismo procedimiento pero ahora analizando la columna de la Estacion numero 3 donde de igual manera vemos que su procedencia viene por línea 1.

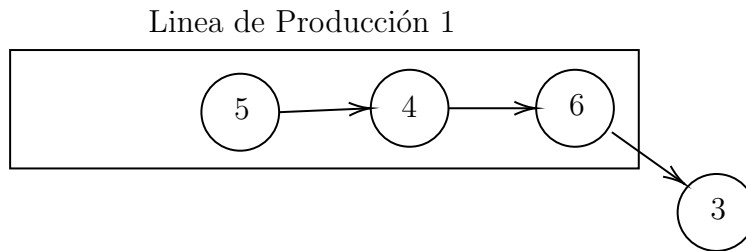


Figura 63 - Procedencia Estación 3

Ya nos encontramos en la Estación 2 así que de igual forma observamos nuestra tabla 6 y vemos que nuestra estación 2 tiene procedencia por línea 1 quedando de la siguiente manera.

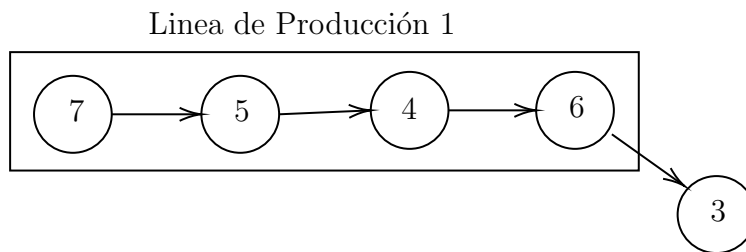


Figura 64 - Procedencia Estación 2

Por último ya posicionados en la estación 1 solo tenemos una opción de procedencia que es la estación de entrada quedando de la siguiente manera.

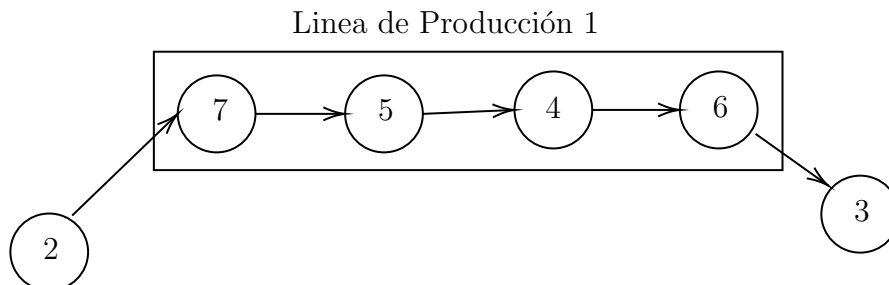


Figura 65 - Procedencia Estacion 1

En la siguiente figura se muestra finalmente como quedó nuestra ruta óptima marcada de color rojo la cual tarda menos tiempo en producir un producto que en este caso solo recorre la Línea 1 y es esta la que nos da el tiempo menor.

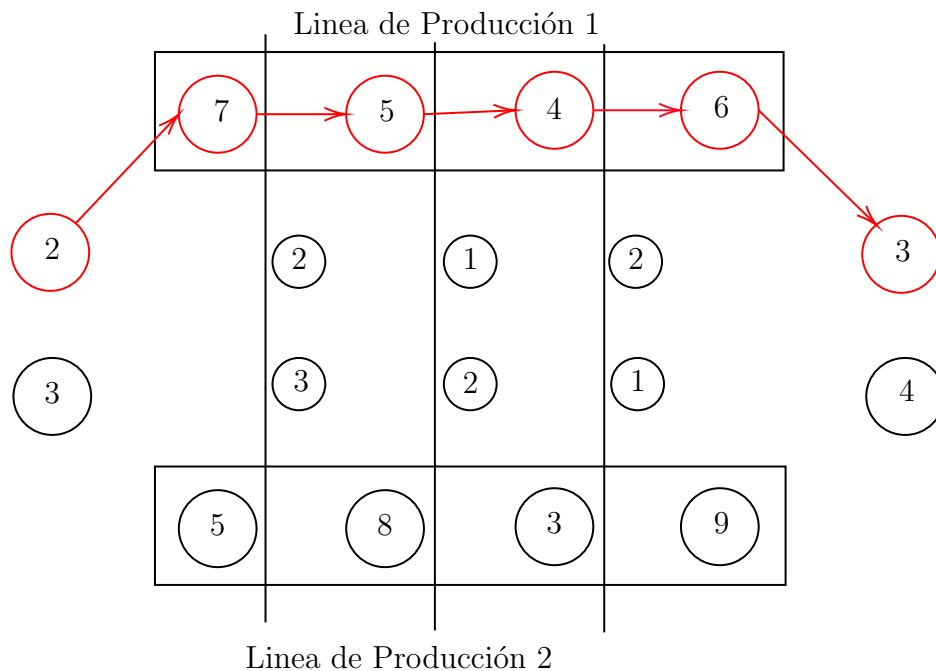


Figura 66 - Ruta optima

A continuación se muestra la solución final del problema:

Linea 1, Estación 4
 Linea 1, Estación 3
 Linea 1, Estación 2
 Linea 1, Estación 1

5.2. Algoritmo de Prim

Este algoritmo se usa normalmente para ahorrar recursos, su aplicación más común es la implementación de cables de redes, de servidores, de postes de luz entre otros.

Sirve para poder hallar el “árbol recubridor mínimo”, en un grafo conexo no dirigido. Es capaz de encontrar un subconjunto de las aristas que formen un árbol que incluya todos los vértices del grafo inicial, donde el peso total de las aristas del árbol es el mínimo posible.

5.2.1. Definición.

Sea V el conjunto de nodos de un grafo pesado no dirigido. El algoritmo de Prim comienza cuando se asigna a un conjunto U de nodos un nodo inicial Perteneciente a V , en el cual “crece” un árbol de expansión, arista por arista.

En cada paso se localiza la arista más corta (u, v) que conecta a U con $V-U$, y después se agrega v , el vértice en $V-U$, a U . Este paso se repite hasta que $V=U$. El algoritmo de Prim es de $O(N^2)$, donde $|V| = N$.

5.2.2. Funcionamiento.

- Se marca un vértice cualquiera. Será el vértice de partida.
- Se selecciona la arista de menor peso incidente en el vértice seleccionado anteriormente y se selecciona el otro vértice en el que incide dicha arista.
- Repetir el paso 2 siempre que la arista elegida enlace un vértice seleccionado y otro que no lo esté. Es decir, siempre que la arista elegida no cree ningún ciclo.
- El árbol de expansión mínima será encontrado cuando hayan sido seleccionados todos los vértices del grafo.

5.2.3. Pseudocódigo Algoritmo de Prim

El pseudocódigo para el algoritmo de prim muestra cómo creamos dos conjuntos de vértices U y VU . U contiene la lista de vértices que se han visitado y VU la lista de vértices que no. Uno por uno, movemos los vértices del conjunto VU al conjunto U conectando el borde de menor peso.

```

1-- T = Conjunto vacío;
2-- U = { 1 };
3-- while (U != V)
4--     let (u, v) Borde de menor costo;
5--     T = T union {(u, v)}
6--     U = U union {v}

```


5.2.4. Ejemplo Algoritmo de Prim

Encuentre un árbol de expansión de costo mínimo para el siguiente gráfico.

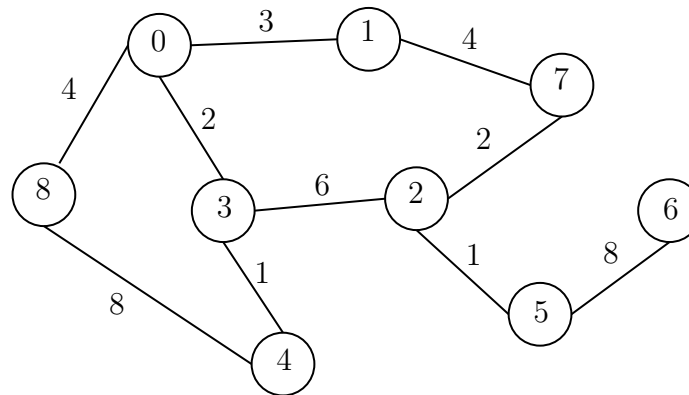


Figura 67 - Grafico 1

Desarrollo. A continuacion procederemos a resolver el problema presentado.

- Paso 1

Escogemos un nodo inicial: nodo 0 y lo marcamos como alcanzado (nodo verde)

:

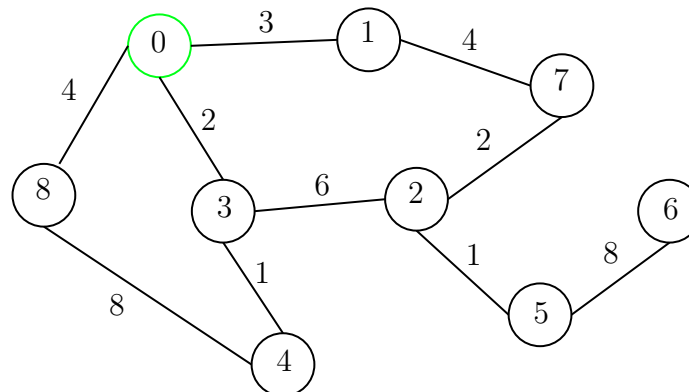


Figura 68 - Grafico nodo inicial

Todos los demás nodos están marcados como no alcanzados.

- Paso 2

Encuentre una ventaja con un costo mínimo que conecte un nodo alcanzado a un nodo no alcanzado. Esta ventaja es: $(0, 3)$.

Agregue el borde $(0,3)$ al MST y marque el nodo 3 como alcanzado

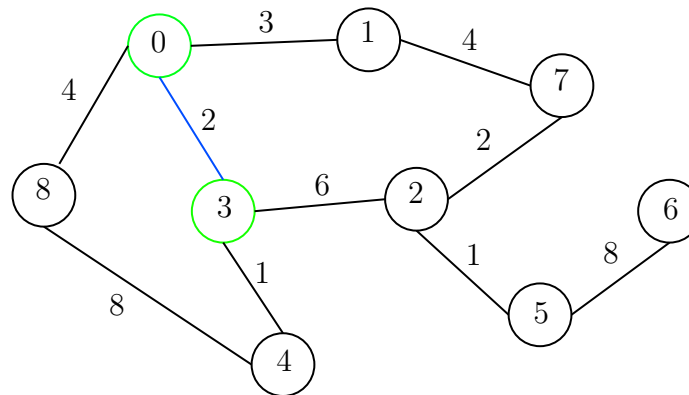


Figura 69 - Grafico unión nodos (0,3)

- Paso 3

Encuentre una ventaja con un costo mínimo que conecte un nodo alcanzado a un nodo no alcanzado. Esta ventaja es: (3, 4).

Agregue el borde (3,4) al MST y marque el nodo 4 como alcanzado :

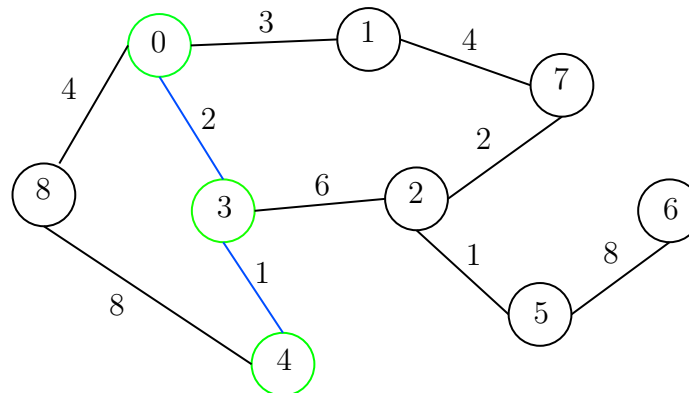


Figura 70 - Grafico unión nodo (3,4)

- Paso 4

Encuentre una ventaja con un costo mínimo que conecte un nodo alcanzado a un nodo no alcanzado. Esta ventaja es: (0, 1).

Agregue el borde (0,1) al MST y marque el nodo 1 como alcanzado.

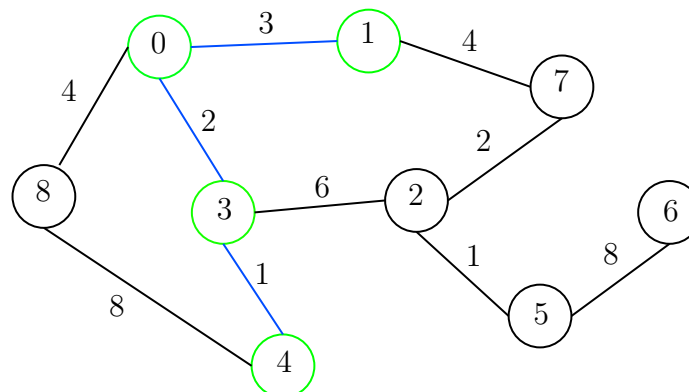


Figura 71 - Grafico unión nodos (0,1)

- Paso 5

Aceleraremos un poco el proceso así que. El siguiente borde agregado es: (1,7).

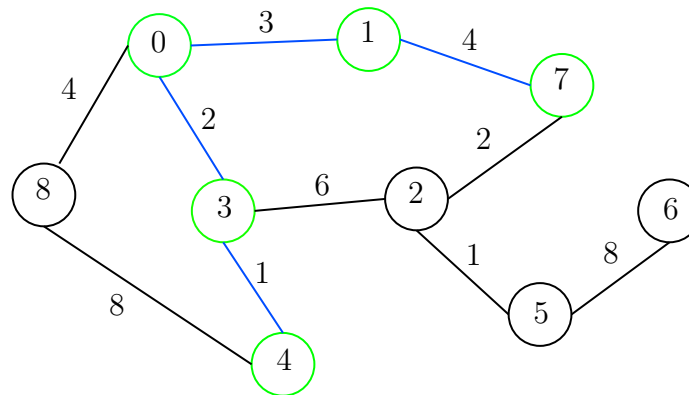


Figura 72 - Grafico unión nodos (1,7)

- Paso 6

El siguiente borde agregado es: (7,2).

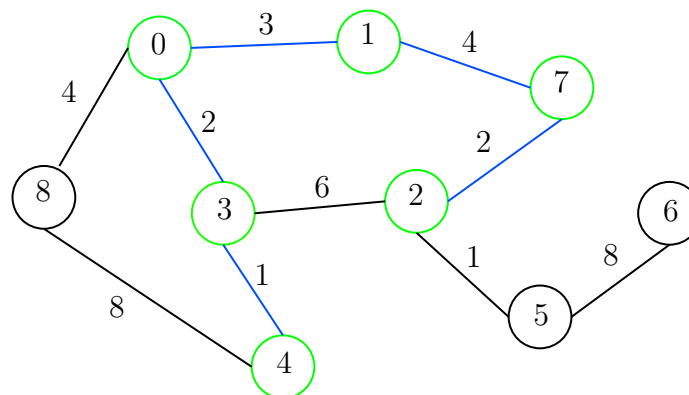


Figura 73 - Grafico unión nodos (7,2)

- Paso 7

El siguiente borde agregado es: (2,5)

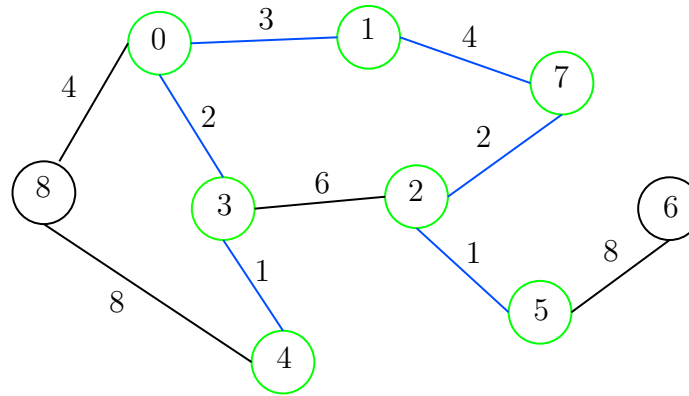


Figura 74 - Grafico unión nodos (2,5)

- Paso 8

El siguiente borde agregado es: (0,8).

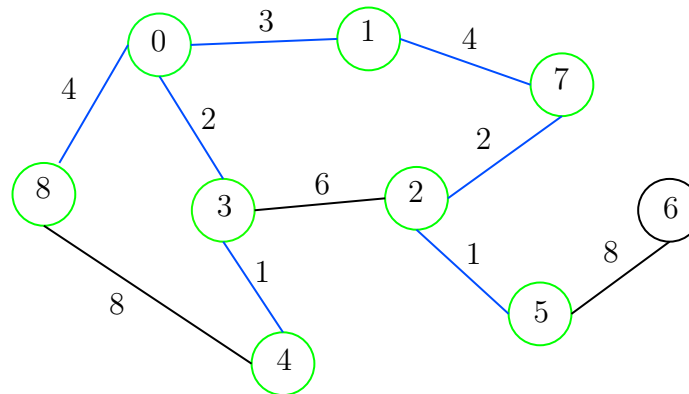


Figura 75 - Grafico unión nodos (0,8)

- Paso 9

El siguiente borde agregado es: (5,6).

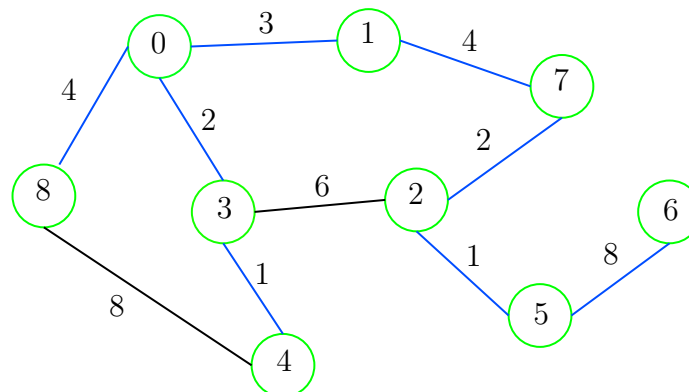


Figura 76 - Grafico unión nodos (5,6)

Como podemos observar ahora se alcanzan todos los nodos del grafico cumpliendo el objetivo del algoritmo de Prim.
El costo mínimo del árbol de expansión es:

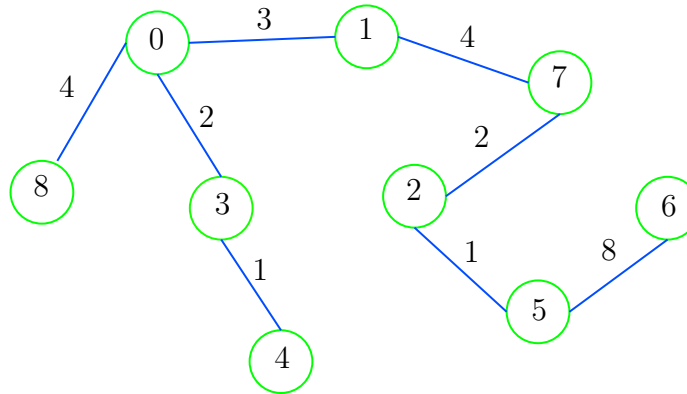


Figura 77 - Árbol de Expansión Mínimo.

5.3. Algoritmo de Kruskal.

Es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor total de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).

El algoritmo de Kruskal, dado un grafo conexo, no dirigido y ponderado, encuentra un árbol de expansión mínima. Es decir, es capaz de encontrar un subconjunto de las aristas que formen un árbol que incluya todos los vértices del grafo inicial, donde el peso total de las aristas del árbol es el mínimo posible.

5.4. Funcionamiento

- Se crea un bosque B (un conjunto de árboles), donde cada vértice del grafo es un árbol separado
- Se crea un conjunto C que contenga a todas las aristas del grafo
- Mientras C es no vacío
- Eliminar una arista de peso mínimo de C
- Si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol

- En caso contrario, se desecha la arista. Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo.

5.5. Pseudocódigo algoritmo kruskal

```

1- kruskal (G: grafo , n: #nodos)
2- Construir una cola de prioridad cp con los arcos del grafo G
3- Inicializar componente conexas
4- F = Conjunto vacío
5- while !vacía(cp) && |F| < n-1
6-     arista = obtenerMin(cp)
7-     borrarMin(cp)
8-     u = componente(de(arista))
9-     v = componente(a(arista))
10-    if numeroComponente(u) != numeroComponente(v)
11-        añadir arista a F
12-    unirComponentes(u,v)

```

5.6. Orden de Complejidad Kruskal

El algoritmo de kruskal se ejecuta sobre estructuras de datos simples.

Al momento de ordenar las aristas del grafo por su peso se usa una ordenación de comparación llamada (Comparison sort) la cual tiene una complejidad de orden $O(n \log n)$ lo cual permite que el paso "eliminar una arista de peso mínimo de C" se ejecute en tiempo constante.

Se usa una estructura de datos sobre conjuntos disjuntos al momento de controlar qué vertices están en qué componentes lo cual tiene una complejidad de orden $O(n)$ ya que por cada arista hay dos operaciones de búsqueda y posiblemente una unión de conjuntos.

Incluso una estructura de datos sobre conjuntos disjuntos simple con uniones por rangos puede ejecutar las operaciones mencionadas en $O(m \log n)$.

Sabiendo esta información llegamos a la conclusión que la complejidad total es del orden de

$$Kruskal \in \Theta(n \log n)$$

6. Bibliografía

[1] D. Programming, "Dynamic Programming — Top-Down and Bottom-Up approach — Tabulation V/S Memoization", CodesDope, 2020. [Online]. Available: <https://www.codesdope.com/course/algorithms-dynamic-programming/>.

[2] Ingenieria.unam.mx, 2020. [Online]. Available: https://www.ingenieria.unam.mx/sistemas/PDF/Avisos/Seminarios/SeminarioV/Sesion6_daliaFlores20abr15.pdf.

[3] "Técnicas de diseño", Desarrolloweb.com, 2020. [Online]. Available: <https://desarrolloweb.com/articulos/2183.php>.

[4] "Números de Fibonacci Quantdare", Quantdare, 2020. [Online]. Available: <https://quantdare.com/numeros-de-fibonacci/>.

[5] Pisinger, D. (2003). Where are the hard knapsack problems?. Technical Report 2003/8, DIKU, University of Copenhagen, Denmark.

[6] "Algoritmo de Prim", Estructura de Datos II, 2020. [Online]. Available: <https://estructurasite.wordpress.com/algoritmo-de-prim/>.

[7] "Algoritmo de Prim - Complejidad Algorítmica", Sites.google.com, 2020. [Online]. Available: <https://sites.google.com/site/complejidadalgoritmicaes/prim>.

[8] Mathcs.emory.edu, 2020. [Online]. Available: <http://www.mathcs.emory.edu/cheung/Courses/171/Syllabus/11-Graph/prim2.html>.

[9] "Algoritmo de Kruskal - EcuRed", Ecured.cu, 2020. [Online]. Available: https://www.ecured.cu/Algoritmo_de_Kruskal.

[10] "Algoritmo de Kruskal - Complejidad Algorítmica", Sites.google.com, 2020. [Online]. Available: <https://sites.google.com/site/complejidadalgoritmicaes/kruskal>.