Universidad del Valle

Análisis y Diseño de Algoritmos II

Facultad de Ingeniería

Escuela de Ingeniería de Sistemas y Computación

PROYECTO DEL CURSO I

Subasta Pública de Acciones

Integrantes

Edinson Orlando Dorado Dorado

edinson.dorado@correounivalle.edu.co

19419966

Alessandro Díaz García

alessandro.diaz@correounivalle.edu.co

1940983

Abril de 2023

Tabla de contenido

Proyecto

2.3. ¿Entendimos el problema?

A = 1000; B = 100; n = 2;
[(500; 600; 100); (450; 800; 400); (100; 1000; 0)]

Describa cinco (5) asignaciones de acciones válidas y diferentes para esa entrada y calcule el valor recibido por cada una de ellas. Guarde los resultados en una tabla.

600	400	0	480000
200	800	0	460000
500	500	0	475000
400	600	0	470000
300	700	0	465000

Describa una asignación de acciones X tal que vr(X) sea máximo.

- vr(X)=[600,400,0] = $500 * 600 + $450 * 400 = 480000

3.1 Usando fuerza bruta

3.1.1 Entendimos el algoritmo

Enumere todas las asignaciones de acciones de la forma descrita para la entrada

A = 1000; B = 100; n = 4;
[(500; 600; 400); (450; 400; 100); (400; 400; 100); (200; 200; 50); (100; 1000; 0)]

y calcule el valor recibido por cada una de ellas. Guarde los resultados en una tabla.

Enum	600...0	400...0	400...0	200...0	1000...0	Resultado
1	600	400	0	0	0	480000
2	600	0	400	0	0	460000
3	600	0	0	200	200	360000
4	0	400	400	200	0	380000
5	0	0	0	0	1000	100000
6	0	400	0	200	400	260000
7	0	0	400	200	400	240000
8	0	0	0	200	800	120000
9	0	0	400	0	600	220000
10	0	400	0	0	600	240000
11	0	400	400	0	200	360000
12	600	0	0	0	400	340000

3.1.2 Complejidad

El algoritmo de fuerza bruta usa una función auxiliar llamada hallar_combinaciones, que retorna un array con todas las posibles combinaciones válidas. Este algoritmo, tiene una complejidad de $O(2^n)$ donde n es el número de compradores diferentes al gobierno.

La función principal accionesFB, la cual usa la función auxiliar (por lo que ya de por si sería $O(2^n)$) lee cada una de esas posibilidades, y guarda cuál es la temporalmente mejor.

Así se concluye con que la complejidad del algoritmo de fuerza bruta es $O(2^n) + O(n) = O(2^n)$.

La complejidad espacial, debido a que cada vez que hace una posibilidad de las $O(2^n)$ con un comprador, guarda un array con esa posibilidad, seria $O(2^n * n)$

3.1.3 Corrección

El algoritmo no da siempre la respuesta correcta (las veces que la da es casi nula en la mayoría de problemas), debido a que solo revisa si las combinaciones en las que el

número de compradores ofrece comprar es o cero, o todas las acciones (c) que están dispuestos a comprar. Y las acciones sobrantes, son vendidas al comprador que ofrece menor precio, al gobierno.

3.2 Usando un algoritmo voraz

3.2.1 Describiendo el algoritmo

Toma primero la mayor cantidad de acciones que el primer comprador puede comprar, luego toma la mayor cantidad que el segundo comprador puede comprar, así hasta asignar las acciones restantes al gobierno.

3.2.2. Entendimos el algoritmo

Calcule la salida de su algoritmo para la entrada:
A = 1000, B = 100, n = 4,
[(500, 600, 400),(450, 400, 100),(400, 400, 100), (200, 200, 50),(100, 1000, 0)]
Calcule el valor recibido por la solución. ¿Es la solución óptima?

- X = [600,400,0,0,0] = 600 * $500 + 400 * $450 + 0 + 0 + 0 = 480000

Sí es la solución óptima.

Describa al menos 4 entradas más, calcule la solución entregada por el algoritmo y verifique si es o no la óptima. Guarde los resultados en una tabla.

Problema	**Solución**	**Dinero recibido**
A=1000 [(500, 300, 200), (450, 300, 100), (400, 400, 100), (200, 200, 50), (100, 1000, 0)]	X = [300,300,400,0,0]	445000 Sí es la óptima
A=1000 [(500, 1000, 400), (450, 400, 100), (400, 400, 100), (200, 200, 50),	X = [1000,0,0,0,0]	500000 Sí es la óptima

(100, 1000, 0)]		
A=1000 [(500, 50, 10), (490, 700, 600), (450, 900, 500), (150, 1000, 0)]	X = [50,700,0,250]	405500 No es la óptima La óptima es: X=[50, 0, 900, 50], 437500
A=1000 [(500, 300, 200), (475, 1000, 900), (400, 500, 50), (100, 1000, 0)]	X = [300,0,500,200]	370000 No es la óptima La óptima es: X=[0, 1000, 0, 0], 475000

3.2.3 Complejidad

La complejidad es O(n), debido a que solamente recorre una vez cada comprador. La complejidad espacial también es O(n) ya que por cada comprador, almacena su número de acciones solo una vez en el array.

3.2.4 Corrección

Este algoritmo, al solo tomar en cuenta la mejor opción en cada paso, no garantiza correctitud ya que no analiza todas las posibles opciones, lo que impide que sea óptimo, pues puede que la solución esté en esas posibles opciones que no son revisadas.

Este algoritmo solo da la respuesta correcta cuando la solución requiere que cada comprador, empezando por el comprador que ofrece mayor beneficio, compre la mayor cantidad de acciones posibles.

3.3 Usando programación dinámica

3.3.1. Caracterizando la estructura de una solución óptima

Si existe una solución óptima para un máximo A de acciones y un número n de compradores, entonces también podemos descomponer esta solución óptima en subproblemas que consisten en un número de acciones menores a A con los mismos n compradores que tienen una solución óptima.

Existe solapamiento de subproblemas: los distintos subproblemas comparten subsubproblemas.

Esto significa que podemos obtener una solución óptima global a partir de la combinación de soluciones óptimas a subproblemas más pequeños.

El número de subproblemas que hay es el número de compradores multiplicado el número de acciones a la venta: n*A, y estos subproblemas comparten subsubproblemas.

3.3.2. Definiendo recursivamente el valor de una solución óptima

Primero se empieza por la venta de un número de acciones pequeño, en todos los casos, se empieza por 1; después de haber calculado cual es la mejor compra que cada comprador puede hacer con lo que puede comprar, para un número de acciones = 1, se sigue con un número mayor de acciones, acciones = 2, que usa acciones = 1, luego, con los valores temporales óptimos para acciones = 1 y acciones = 2, se procede a hallar los valores temporales óptimos para acciones = 3, y así hasta encontrar el valor de la solución óptima. Esto se repite hasta que todas las posibilidades han sido tomadas en cuenta (acciones = 2A-1).

3.3.3. Describiendo el algoritmo para calcular el costo de una solución óptima

Usando los valores óptimos en la columnas (A - x) (siendo x un número entero no mayor a A acciones), se tiene una función que arma una solución para cada comprador con la mayor cantidad de dinero dada para una cantidad A de acciones por parte de un conjunto de compradores incluido el mismo. De esta forma, como los valores usados para hallar el costo mínimo en la compra de un determinado número de acciones son óptimos, este número también va a ser óptimo. Siendo el costo de la solución, óptimo, o lo que es lo mismo, el precio más alto que se puede dar por un número de acciones A.

3.3.4. Describiendo el algoritmo para calcular una solución óptima

Para las implementaciones de programación dinámica nos basamos en el problema de la mochila, e hicimos una analogía en la que la “capacidad” de la mochila en este caso es el número de acciones a la venta, el “peso” es el número de acciones que un comprador compra y el “beneficio” el dinero que paga cada comprador por un número de acciones.

En el desarrollo del proyecto realizamos dos algoritmos dinámicos, ambos dando siempre la solución óptima, pero nuestra implementación inicial presentó un problema.

Implementación inicial

Nuestra primer implementación del algoritmo usando programación dinámica consiste en tomar las ofertas de cada comprador y por cada oferta posible que pudiera dar en el rango entre r y c se crea una nueva oferta o subproblema en la matriz que llamamos “matriz de beneficios” y ponemos los compradores incluidos en esa oferta en una matriz de “caminos”.

Entrada:

A	6
B	5
n	3
	(15,3,1)
	(10,4,3)
	(5,6,1)

Matriz de beneficios:
Cada fila corresponde a un comprador con un peso (número de acciones) y beneficio (dinero). Cada columna indica la capacidad (cantidad de acciones).

	Peso	Beneficio	**1**	**2**	**3**	**4**	**5**	**6**
Comprador 1	**1**	**15**	15	20	25	15+30	15+40	15+25
	2	**30**	0	30	35	30+10	30+30	30+40
	3	**45**	0	0	45	45+5	45+10	45+30
Comprador 2	**3**	**30**	0	0	30	30+15	30+30	30+45
	4	**40**	0	0	0	40	40+15	40+30
Gobierno	**1**	**5**	5	20	35	5+30	5+15+30	5+60
	2	**10**	0	10	25	10+30	10+45	10+45
	3	**15**	0	0	15	15+15	15+30	15+45
	4	**20**	0	0	0	20	20+15	20+30
	5	**25**	0	0	0	0	25	25+15
	6	**30**	0	0	0	0	0	30

Matriz de caminos:
Cada vez que se añade un nuevo valor en la matriz de beneficios se añade al comprador o compradores a la matriz de caminos, esto para evitar que durante las combinaciones se creen ofertas en las que haya un mismo comprador dos veces.

	Peso	Beneficio	**1**	**2**	**3**	**4**	**5**	**6**
Comprador 1	**1**	**15**	1	1,G	1,G	1,2	1,2	1,G
	2	**30**	0	1	1,G	1,G	1,2	1,2
	3	**45**	0	0	1	1,G	1,G	1,2
Comprador 2	**3**	**30**	0	0	2	2,1	2,1	2,1
	4	**40**	0	0	0	2	2,1	2,1
Gobierno	**1**	**5**	G	G,1	G,1	G,1	G,1,2	G,1,2
	2	**10**	0	G	G,1	G,1	G,1	G,1,2
	3	**15**	0	0	G	G,1	G,1	G,1
	4	**20**	0	0	0	G	G,1	G,1
	5	**25**	0	0	0	0	G	G,1
	6	**30**	0	0	0	0	0	G

Nuestro algoritmo inicia en la columna o número de acción 1 y recorre la matriz de izquierda a derecha hasta la acción A. Por cada columna que lee recorre cada comprador:

- Si el comprador puede dar el número de acciones en su totalidad se agrega a la matriz de beneficios el dinero que aporta.
- Si el número de acciones que ofrece este comprador es menor al número de acciones actuales, las acciones que hacen falta se buscan en la columna de la capacidad que hace falta para completar las A acciones.

Este proceso lo repite hasta llegar a la capacidad A, en esta columna se obtiene el valor más alto, los compradores incluidos y el número de acciones de cada uno de estos se obtienen de la matriz de caminos en la misma posición que obtuvimos la solución óptima en la matriz de beneficios.

El algoritmo retorna la solución óptima para todas las pruebas realizadas, el problema que surgió fue cuando ejecutamos las pruebas dadas por los profesores, en las cuales habían muchos compradores y/o valores de A muy grandes por lo que con la mitad de las pruebas dadas este algoritmo necesitaría más de veinticuatro horas o incluso más de un mes para llegar a la solución óptima.

Analizando el algoritmo nos dimos cuenta que la complejidad de este era demasiado alta ya que a cada oferente se le crea un número de subproblemas dependiendo de la diferencia entre el número de acciones máximo que está dispuesto a comprar (c) y el número de acciones mínimo que está dispuesto a comprar (r). A su vez esto se debe hacer por cada acción hasta A y cada uno de estos debe revisar toda una columna de la matriz, por lo que llegamos a esta complejidad:

$$O((\sum_{i=1}^{n} c_i - r_i)^2 * A)$$

En espacio, (tomando en cuenta que la matriz de caminos tiene un array con hasta n compradores, a diferencia de la matriz de beneficios):

$$O((\sum_{i=1}^{n} c_i - r_i) * A * n)$$

Tomando como ejemplo la entrada "bsp_10000_7_5.sub":

10000
7
6
52,7842,3241
50,4555,2659
35,9923,2990
23,5062,66
17,3708,1763
7,10000,0

$$(4601 + 1896 + 6933 + 4996 + 5471 + 10000)^2 * 10000 = 1.1490066e + 13$$

Esto significa que nuestro programa debía hacer aproximadamente 11,490,066,000,000 de operaciones para esta prueba y en nuestros computadores la ejecución tomó 4 días en terminar.

Considerando que habían pruebas con valores mucho más altos, analizando el algoritmo nos dimos que la gran cantidad de operaciones que el programa usaba muchas eran innecesarias ya que la mitad de la matriz está llena de ceros que no se necesitan para obtener la solución óptima, por lo que estábamos gastando mucho tiempo y espacio que no era necesario.

Este programa lo incluímos en la carpeta anexa "Dinámica no factible".

A pesar de que esta implementación garantiza correctitud, pero no era útil en la práctica, decidimos realizar una nueva versión con programación dinámica del problema de la subasta.

Implementación final

Después de analizar nuestro programa anterior buscamos la manera de ahorrar al máximo en tiempo y espacio. Debido a que nuestros programas están hechos en Python la complejidad de la búsqueda e inserción en una lista es O(n), mientras que un diccionario de Python (que utiliza tablas hash) en promedio tiene una complejidad de O(1), por lo que decidimos escoger este último para nuestra nueva implementación.

Nos seguimos basando en el problema de la mochila, para nuestra implementación final ya no tomamos cada posible número de acciones que puede dar un comprador como un subproblema en una fila nueva en la matriz, ya que esto nos eleva demasiado la complejidad. Ahora solo tomamos a cada comprador y creamos una fila por cada uno y creamos una columna por cada número de acciones desde 1 hasta A. Debido a que estamos usando diccionarios, solo vamos creando los valores que necesitamos almacenar para usar después, para no consumir espacio innecesario, por lo que ahora no tenemos filas y columnas, tenemos un diccionario de nombre "matriz" que tiene el número de acciones como llave, como valor un id del comprador y estos un valor con el dinero de la oferta, convertimos la matriz en un diccionario que la representa solo con valores que se necesiten. De la misma manera que la implementación inicial, tenemos un diccionario de "caminos" para saber qué compradores están incluidos en esa oferta y no incluir dos veces a un mismo comprador en una misma solución de un subproblema.

La primera parte del algoritmo consiste en recorrer inicialmente cada acción desde 1 hasta A, revisa cada comprador y si está en el rango de acciones $r \leq acciones \leq c$ que está dispuesto a comprar, ponemos su beneficio en el diccionario. Cuando el número de acciones es mayor a c suponemos que el comprador siempre va a dar su máximo de acciones y buscamos un subproblema si existe, que permita completar las A acciones.

Esto lo hace hasta terminar de leer las A acciones, cuando esto sucede ya tenemos una "matriz" en la que cada comprador da siempre el máximo de acciones (c) que está dispuesto a dar para cada número de acciones que se está leyendo.

No en todas las subastas va a existir un comprador que en la solución óptima incluya su máximo de acciones (c). Por lo que necesitamos saber si haciendo combinatorias con valores menores a c para cada comprador obtenemos la solución óptima. Para esto, el algoritmo cuando termina de leer las A acciones empieza a recorrer las acciones

menores a *c* en reversa hasta llegar al número de acciones mínimo (r) que está dispuesto un comprador a ofrecer.

Aunque creamos diccionarios podemos seguir visualizando el problema como una matriz:

A	**8**
B	**20**
n	**3**
	50,3,2
	40,4,3
	20,8,1

	r (mín)	c (máx)	p	1	2	3	4	5	6	7	8
Comp 1	2	3	50	X	100	150	4-3=1 150+20 170	5-3=2 150+40 190	6-3=3 150+120 270	7-3=4 150+160 310	8-3=5 150+100 250
Comp 2	3	4	40	X	X	120	160	5-4=1 160+20 180	6-4=2 160+100 260	7-4=3 160+150 310	8-4=4 160+170 330
Comp 3	1	8	20	20	40	60	80	100	120	140	160

Hacemos combinaciones usando los cálculos a los subproblemas realizados previamente desde c-1 hasta r (de mayor a menor) para cada comprador:

9	10	11	12	13	14	15
8-2=6 100+120 220						
8-3=5 120+190 310						
8-7=1 140	8-6=2 120+100	8-5=3 100+150 250	8-4=4 80+160 240	8-3=5 60	8-2=6 40+270 310	8-1=7 20+310 330

Diccionario de “caminos”:

	r (mín)	c (máx)	p	1	2	3	4	5	6	7	8
Comprador 1	6	8	52	X	1	1	1,3	1,3	1,2	1,2	1,3
Comprador 2	3	6	50	X	X	2	2	2,3	1,2	1,2	1,2,3
Comprador 3	0	10	25	3	3	3	3	3	3	3	3

9	10	11	12	13	14	15
1,3						
1,2,3						
3	1,3	1,3	2,3	3	1,2,3	1,2,3

La razón para hacer esto después de leer inicialmente las A acciones es que cuando estamos leyendo la primera mitad de acciones no podemos hacer cálculos con subproblemas con mayor número de acciones ya que aún no se han calculado, debido a la estrategia bottom-up que estamos utilizando.

3.3.5. Complejidad

Complejidad en tiempo

La complejidad de este algoritmo sería el número de compradores (n) al cuadrado por dos veces el número de acciones menos 1 (debido a las combinaciones hechas en reversa, después de leer las A acciones):

$$n^2 * (A * 2 - 1)$$

$$O(n^2 * A)$$

Por lo que tomando nuestro ejemplo anterior “bsp_10000_7_5.sub”:

10000
7
6
52,7842,3241
50,4555,2659
35,9923,2990

23,5062,66
17,3708,1763
7,10000,0

El programa haría aproximadamente 719,964 operaciones, lo cual es ínfimo comparado con nuestra implementación inicial de 11,490,066,000,000 operaciones. Nuestros tiempos de ejecución pasaron a menos de 1 segundo para esta entrada, y máximo 10 segundos para cualquier prueba de las dadas.

Complejidad en espacio

La complejidad en espacio es $O(n^2 * A)$ debido a que el diccionario de "caminos" tiene un diccionario de compradores dentro de él donde en el peor de los casos incluye a los n compradores, a comparación del diccionario "matriz".

¿Es útil en la práctica?

Suponiendo que tenemos un equipo que procesa $3*10^8$ operaciones por minuto y que utiliza 4 bytes de almacenamiento por celda. Si $A=32*10^6$ $n=2^{10}$

Tiempo que tomará el equipo en resolver el problema:
$O(n^2 * A)$ = 3.3554432e+13/$3*10^8$ = 111848 minutos = 1864 horas = 77.67 días

Espacio:
$O(n^2 * A)$ = 3.3554432e+13 * 4 bytes = 1.3421773e+14 / 1,000,000 = 134,217,730 megabytes = 131072 GB

Evidentemente tanto en espacio como en tiempo se tienen valores y una complejidad muy alta para el caso real.

¿Qué hacer en la práctica?

Se propuso ya no por cualquier cantidad de acciones, si no por paquetes de 5000 acciones.

Con nuestra implementación la complejidad tanto en tiempo como en espacio para la programación modificada sería:

$$O(n^2 * A \ / \ tamaño\ paquetes)$$

Ya que en vez de iterar por cada acción desde 1 hasta A, el algoritmo hace iteraciones por cada tamaño de los paquetes.

Tiempo:
$O(n^2 * A / tamaño\ paquetes)$ = 3.3554432e+13/5000=6,710,886,400/3*10^8
= 22.36 minutos

Espacio:
$O(n^2 * A / tamaño\ paquetes)$ = 3.3554432e+13/5000= 6,710,886,400 * 4 bytes = 26843545600 bytes = 26,8 GB

Usando la programación modificada con paquetes la complejidad en tiempo es mucho más razonable, usable y útil en la práctica para el caso real a comparación de la dinámica no modificada, pero la complejidad en espacio puede que no sea lo suficientemente útil para el caso real en ese entonces, pero esto es inherente a la programación dinámica que sacrifica espacio por tiempo.

3.4. Implementación

Link repositorio GitHub

https://github.com/alessandrodiaz/Proyecto-Subasta-Algoritmos-II
Implementado en Python. Clonar el repositorio y correr el archivo main.py en una terminal

Comparación entre las 4 soluciones

Link resultado de las pruebas

https://docs.google.com/spreadsheets/d/1yTc9dtQ5ui52swSBd_DZFvVblmcQWVbyEc4_EJWsgh4/edit?usp=sharing

El de fuerza bruta debido a su complejidad tiene tiempos de ejecución muy altos, lo que lo hace muy lento, es muy ineficiente y no obtiene la solución óptima en la mayoría de los casos. Solo obtuvo la solución óptima en 3 de los 52 casos de pruebas y en muy pocas de nuestras pruebas.

El algoritmo voraz a pesar de ser muy eficiente y rápido debido a su complejidad, no garantiza correctitud, ni es el adecuado para problemas que dependen de soluciones

previas como es el caso de nuestro problema. Solo llega a la solución óptima en 27 de los 52 casos de pruebas y en algunas de nuestras pruebas no logra la óptima.

La programación dinámica nos garantiza correctitud. Requiere de más memoria y espacio y puede ser más difícil de entender y aplicar a comparación de las técnicas anteriores. Todas las pruebas con programación dinámica y dinámica modificada obtuvieron la solución óptima en máximo 10 segundos de ejecución.

Conclusiones

- La fuerza bruta puede ser fácil de entender, pero es ineficiente por su complejidad en tiempo y en espacio.
- La programación voraz posee la menor complejidad en tiempo y espacio, pero no garantiza correctitud.
- La programación dinámica penaliza espacio, pero nos garantiza correctitud y nos puede dar mejor complejidad en tiempo a comparación de la fuerza bruta.
- Es importante entender y desarrollar adecuadamente un problema para construir una solución óptima con programación dinámica.