Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2012

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1
Grupo 5

Autor	Correo electrónico	LU
Emilio Ferro	reclarinette2002@yahoo.com.ar	441/00
Marina A. Niikado	mariniik@yahoo.com.ar	711/07
Franco Pistasoli	fpistasoli@gmail.com	194/08
Ezequiel Schwab	ezequiel.schwab@gmail.com	132/09

Índice

1. Desarrollo

Problema 1

Introducción

En este problema se pedía devolver cual era la mayor ganancia posible dada una serie de precios de un producto para distintos dias. Es decir, si lo quisieramos formalizar matematicamente, dada una secuencia de n precios $p_1, ..., p_n$, se desea obtener j', i' con $j' > i'$ tal que $p_{j'} - p_{i'} \geq p_j - p_i \; \forall \; 1 \leq i < j \leq n$. Notar que si la resta diera un numero negativo para todos los casos (una secuencia de precios estrictamente decreciente), el resultado que se querria devolver es 0 (no haber comprado, y por lo tanto vendido, ningun dia). Para incluir este caso en la expresion matematica anterior, basta con decir ademas que j' puede ser igual a i' (como si pudieramos comprar y vender en el mismo dia, y por lo tanto la ganancia maxima va a ser siempre al menos 0).

Algoritmo propuesto

El algoritmo propuesto es el siguiente:

- Creamos 2 variables locales, ganancia, inicializada en 0, y precioDeCompra, inicializada en ∞
- Recorremos el arreglo de precios:
  - Si el precio que estamos mirando en cada iteracion es menor al precio que teniamos guardado en precioDeCompra, entonces reemplazamos el valor de precioDeCompra con ese nuevo valor.
  - En cambio, si es mayor (o igual aunque este ultimo caso no nos interesa) nos fijamos cual es la ganancia obtenida vendiendo ese dia, y habiendo comprado al precio que teniamos guardado en precioDeCompra. Si la ganancia es mayor a la que teniamos guardada, se modifica la variable ganancia.
- Una vez recorrido toda la secuencia de precios, en la variable ganancia vamos a tener el resultado que buscabamos.

Complejidad

La complejidad del algoritmo es claramente θ(n), donde n es la cantidad de precios (tamano de la secuencia de precios). Esto se puede ver facil en el pseudocodigo mostrado en el desarrollo del algoritmo: Se recorre toda la secuencia de precios siempre (θ(n)), y en cada iteracion lo unico que se hace es una cantidad constante de operaciones basicas (comparaciones y asignaciones) por lo que la complejidad final no cambia.

Problema 2

Introducción

En este problema se presenta una red de amistades entre investigadores, que generan nuevas ideas y transmiten su entusiasmo a los demás. Al ocurrírsele una idea a un investigador, un 50 % de su entusiasmo se transfiere a su amistad más directa. De esta manera, un 50 % del entusiasmo recibido por ésta se transmite a su amistad más directa, y así sucesivamente.

El problema en particular consiste en, dados dos investigadores p y q, determinar cuántas veces el entusiasmo de p se fraccionará a la mitad hasta llegar a q.

Algoritmo propuesto

El problema descripto anteriormente puede traducirse así: dados dos nodos p y q de un grafo, hallar la longitud del camino mínimo entre ellos. Si l es esta longitud, entonces significa que el investigador q recibirá $\frac{1}{2^l}$ del entusiasmo generado por p.

Contamos con una estructura del tipo diccionario *investigadores*, en la cual tenemos como clave a cada uno de los investigadores y como significado, sus respectivos amigos directos. El grafo está implementado con una lista de adyacencias, la cual relaciona a cada nodo (el cual representa a un investigador) con los nodos adyacentes a él (amigos del investigador). Además, tenemos otra estructura del tipo diccionario $pasosMinimos$, en la cual guardamos la cantidad mínima de pasos del investigador p hacia los demás investigadores.

- *investigadores*:Dicc(nodo r, Lista(nodos ady a r))
- Claves(*investigadores*): nodos del grafo
- $pasosMinimos$:Dicc(nodo r, cant mín aristas desde p a r)
- Claves($pasosMinimos$): nodos del grafo

La técnica elegida para la resolución del problema fue *backtracking*.
La idea consiste en ir recorriendo todos los caminos posibles del árbol, pero ignorando los que superen cierta cantidad de pasos.
Se tiene como raíz a p y las ramas corresponden a cada uno de los amigos de este investigador.
El criterio de poda del árbol está dado según si al pasar por un nodo r, y al obtener la cantidad de pasos mínimos de este nodo en el diccionario $pasosMinimos$, se supera ese valor (o es igual). Si esto ocurre, no se continúa por ese camino. Este diccionario se actualiza en el momento en que para el caso de un nodo r, encontramos un camino en el que la cantidad de pasos tomados desde p hasta r es menor al valor obtenido hasta el momento.

El algoritmo propuesto termina en el momento en que todas las posibles ramas del árbol fueron recorridas. Como tenemos los valores de los pasos mínimos correspondientes a todos los investigadores, evitamos de esta manera, caer en ciclos infinitos. En cada iteración se hace un llamado recursivo, y en el caso de que la cantidad de pasos se

supere o sea igual con respecto a caminos anteriores, no se continúa por ese camino y se sigue por otro. Por lo que nos aseguramos siempre estar recorriendo el camino más óptimo. Finalmente, en el diccionario $pasosMinimos$ tendremos la cantidad mínima de pasos que toma llegar del nodo p a q. Este dato es la solución buscada para saber la cantidad de veces que el entusiasmo original fue fraccionado.

Para llegar a la solución final que presentamos en esta sección, hemos pasado por algunos caminos alternativos. Uno de ellos fue considerar un algoritmo similar al propuesto anteriormente, en lo que se refiere a estructuras utilizadas para la implementación del grafo, pero algo diferente al momento de recorrer el árbol para llegar a la solución.
En este caso, se cuenta con una variable res en la que se va guardando la cantidad mínima de pasos hasta llegar al nodo q desde p. Además, en lugar de un diccionario con los pasos mínimos de todos los nodos del grafo, sólo tenemos la información de si ese nodo fue recorrido o no durante el camino realizado por una de las ramas del árbol (con raíz p). La cantidad de ramas equivale a la cantidad de nodos adyacentes a p.

- $investigadores$: Dicc(nodo r, Lista(nodos ady a r))
- Claves($investigadores$): nodos del grafo
- $siRecorrio$: Dicc(nodo r, bool)
- Claves($siRecorrio$): nodos del grafo
- res : mín cant pasos desde p a q

La técnica utilizada también fue *backtracking* pero al momento de descartar caminos, sólo ocurría esto si se superaba la cantidad de pasos almacenados en la variable res o si el siguiente nodo a recorrer, ya había aparecido en la rama actual. Esto último evitaba que se produjeran ciclos infinitos en el árbol.
Luego de terminar de recorrer todas las ramas del árbol se llegaba a la solución buscada. Sin embargo, en muchos de los casos, no se podían descartar caminos que con el anterior algoritmo sí era posible. La clave estaba en que en el primer algoritmo, se tenía la cantidad de pasos mínimos de todos los nodos, no sólo de q. De esta forma, una vez llegado a un nodo r que no fuera q y todavía no habiendo superado el valor de res, pero sí la cantidad de pasos mínima de p a r hasta el momento era menor, ya no se continuaba por ese camino (cuando en este algoritmo se hubiera seguido igual por ese camino). En el primer algoritmo se recorren de manera óptima todas las ramas y subramas del árbol, no siendo así en esta forma alternativa.

Complejidad

En esta sección hablaremos de la complejidad temporal del algoritmo `obtenerEntusiasmo`. Hallaremos una cota y justificaremos por qué la alcanza.

Si nos referimos al código Java de `obtenerEntusiasmo`, veremos que su complejidad depende básicamente de las complejidades de tres métodos auxiliares: `toAmistades`,

`generarAmistades` y `obtenerEntusiasmoAux`. Todas las demás operaciones que se realizan, con excepción de las llamadas a estos tres métodos, consisten en asignaciones y dos estructuras de decisión. Las asignaciones se hacen en tiempo constante. El primer `if` invoca al método `containsKey` de la clase HashMap, la cual es lineal en la cantidad de investigadores[1], mientras que el segundo `if` toma tiempo constante.

Sean T_{tA}, T_{gA} y T_{oEA} a los tiempos en peor caso de estos tres algoritmos respectivamente ($tA = toAmistades, gA = generarAmistades, oEA = obtenerEntusiasmoAux$). Sea T_{oE} el tiempo en peor caso del algoritmo `obtenerEntusiasmo`. Entonces,

$$T_{oE}(a) = |investigadores| + T_{tA}(a) + T_{gA}\left(\frac{a}{2}\right) + T_{oEA}(a)$$

siendo $a = 2 * |amistades|$, o sea el doble de la cantidad de amistades en la comunidad de investigadores. A continuación desarrollamos cada uno de los tiempos.

- T_{tA} El método `toAmistades` consiste de una inicialización de un arreglo redimensionable (estructura de datos ArrayList), la cual es $\mathcal{O}(1)$, y un ciclo con operaciones de tiempo constante (asignaciones, métodos `get` y `add` de ArrayList[1], y la creación de un objeto Amistad, el cual se hace en tiempo constante). Este ciclo se ejecuta $\frac{a}{2}$ veces, que equivale a $|amistades|$ veces. Luego, $T_{tA}(a)$ es $\mathcal{O}(\frac{a}{2})$.

- T_{gA} El método `generarAmistades` consiste de un ciclo que recorre un arreglo redimensionable de objetos Amistad. Dentro del ciclo, se ejecutan primero dos asignaciones y llamadas a los métodos `dameAmigo` (de la clase Entusiasmo) y `GetPrimero` y `GetSegundo` (de la clase Amistad). Los dos últimos métodos se ejecutan en tiempo constante.

  En `dameAmigo` se invoca a los métodos `containsKey` y `put`, ambos de la clase HashMap. `containsKey` debe recorrer todas las claves del HashMap, luego toma tiempo lineal en la cantidad de claves[1], que equivale a la cantidad de investigadores, que llamamos $|investigadores|$. Por otro lado, `put` toma tiempo constante, asumiendo que la función de hash usada distribuye los elementos apropiadamente a lo largo de la tabla[1]. Luego, `dameAmigo` es $\mathcal{O}(|investigadores|)$.

  En las dos últimas líneas se llama dos veces al método `add`, de la clase ArrayList, que toma tiempo constante (amortizado)[1].

  Por lo tanto, cada iteración del ciclo en `generarAmistades` toma tiempo $\mathcal{O}(|investigadores|)$. Como el ciclo se ejecuta $|amistades|$ veces, entonces T_{gA} es $\mathcal{O}(|investigadores| * |amistades|)$.

- T_{oEA} El código Java del método `obtenerEntusiasmoAux` es el siguiente:

```
private int obtenerEntusiasmoAux(String p, String q, int cantPasosRealizados)
1: ArrayList < String > amigos = investigadores.get(p).GetAmigos();
2: if (cantPasosRealizados == 1) then                          ▷ O(|investigadores|)
3:     pasosMinimos = newHashMap < String, Integer > ();       ▷ O(1)
4:     for (String investigador : investigadores.keySet()) do  ▷ O(|investigadores|)
5:         pasosMinimos.put(investigador, Integer.MAX_VALUE);  ▷ O(1)
6:     end for
7:     pasosMinimos.put(p, 0);                                 ▷ O(1)
8: end if
```

```
 9: for (String amigo : amigos) do                          ▷ Ver más adelante
10:     if (amigo.equals(q)) then                           ▷ guarda O(1)
11:         cantPasosMin = cantPasosRealizados;             ▷ O(1)
12:         break;                                          ▷ O(1)
13:     else
14:         if (cantPasosMin > cantPasosRealizados + 1) then   ▷ guarda O(1)
15:             if (pasosMinimos.get(amigo) > cantPasosRealizados) then ▷ guarda
    O(1)
16:                 pasosMinimos.put(amigo, cantPasosRealizados);   ▷ O(1)
17:                 cantPasosMin = obtenerEntusiasmoAux(amigo, q, cantPasosRealizados+
    1);                                                     ▷ Ver más adelante
18:             end if
19:         else
20:             break;                                      ▷ O(1)
21:         end if
22:     end if
23: end for
24: return cantPasosMin;
```

Este código consiste básicamente de un `if` y un `for`. El `if` (líneas 2 a 8) toma tiempo $\mathcal{O}(|investigadores|)$, ya que se llama al método `containsKey` y se ejecuta un ciclo que recorre las claves de *investigadores*, una tabla de hash que contiene a los investigadores como claves. Las operaciones que se usan de HashMap (`put` y constructor)[1] son de tiempo constante.

En las líneas 9 a 23 se ejecuta un ciclo tantas veces como amigos tiene el investigador p. En principio se puede decir que esta cantidad está acotada superiormente por $|investigadores| - 1$, ya que un investigador puede tener como amigos a todos menos él. En peor caso se entra a la rama `else` del `if` (líneas 13 a 22), dentro del ciclo. Allí se llama a los métodos `get` y `put` de HashMap, que ya vimos que toman tiempo constante, y en la línea 17 se hace la llamada recursiva a `obtenerEntusiasmoAux`.

Para armar la ecuación de recurrencia para este algoritmo, necesitamos que el tamaño de alguno de los parámetros de entrada de la función vaya disminuyendo en cada iteración. En este caso, podríamos decir que se hace recursión sobre la profundidad actual del subárbol en el árbol de backtracking. Este tamaño no siempre disminuye en iteraciones consecutivas (aunque sabemos que tiende a 0, que es cuando finaliza la ejecución del algoritmo). Entonces, si pr es la profundidad en el árbol de backtracking en la iteración i, entonces llamemos pr' a la profundidad en el árbol en la iteración $i + 1$. Luego,

$$T_{oEA}(pr) = |investigadores| + |amigos| * T_{oEA}(pr') \leq$$

$$|investigadores| + |investigadores| * T_{oEA}(pr')$$

ya que $|amigos| \leq |investigadores| - 1$, por lo comentado anteriormente.

Abstrayéndonos de esta ecuación, pensemos el peor caso para este algoritmo. Por un lado, sabemos que el algoritmo encuentra solución (es decir, devuelve un valor mayor o igual a 1) o no encuentra una solución (devuelve 0). Analizaremos

entonces la complejidad de este algoritmo según cada caso.

Caso en que se encuentra una solución

En este caso, sabemos que existe un camino de p a q en el grafo que relaciona a los investigadores del problema. Supongamos entonces que tenemos la mayor cantidad posible de amistades entre los investigadores (para tener muchas formas de ir de p a q en el grafo). Además, el orden en que vienen los investigadores del parámetro `amistadesLista` de `obtenerAmistades` es tal que q aparece siempre último en la lista de amigos de cada investigador (campo `amigos` de la clase INVESTIGADOR). Si tenemos i investigadores, entonces la mayor cantidad de amistades entre ellos es $\binom{i}{2}$, ya que corresponde a la cantidad de maneras de agrupar a los investigadores de a pares. Dadas estas hipótesis, que se cumplen para el peor caso que puede darse en este algoritmo, entonces comenzamos nuestro razonamiento acerca de su complejidad.

Empecemos por ver cuántos movimientos hay que hacer en el árbol de backtracking (o simplemente, cuántos pasos toma el algoritmo `obtenerEntusiasmoAux`) para una cantidad i de investigadores, donde se tiene $\binom{i}{2}$ amistades (la mayor cantidad posible). Se quiere ir de p a q, y q aparece siempre al final de la lista de amigos de cada investigador. Por simplicidad, denotaremos a los i investigadores con las primeras i letras del abecedario, y p y q son, respectivamente, la primera e i-ésima letra del abecedario. Luego, separemos en casos:

- $i = 2, p = A, q = B$; investigadores: A, B; amistades: (A,B)
- $i = 3, p = A, q = C$; investigadores: A, B, C; amistades: (A,B); (A,C); (B,C)
- $i = 4, p = A, q = D$; investigadores: A, B, C, D; amistades: (A,B); (A,C); (A,D); (B,C); (B,D); (C,D)
- $i = 5, p = A, q = E$; investigadores: A, B, C, D, E; amistades: (A,B); (A,C); (A,D); (A,E); (B,C); (B,D); (B,E); (C,D); (C,E); (D,E)
- $i = 6, p = A, q = F$; investigadores: A, B, C, D, E, F; amistades: (A,B); (A,C); (A,D); (A,E); (A,F); (B,C); (B,D); (B,E); (B,F); (C,D); (C,E); (C,F); (D,E); (D,F); (E,F)

Podemos ver que para $i = 2$ se requieren 2 pasos para salir del algoritmo de backtracking; para $i = 3$, se requieren 8 pasos; para $i = 4$, 18 pasos; para $i = 5$, 32 pasos; y para $i = 6$, 50 pasos. En general, para i investigadores y $\binom{i}{2}$ amistades, tenemos un árbol de backtracking con i niveles, donde en cada nivel $i \geq 2$, hay $i - 1$ nodos. Además, notamos que de un caso donde hay j investigadores a otro donde hay $j + 1$, se agregan $(j - 1) + j$ nodos. Debido a que los nodos que se agregan en el caso $i = j+1$ son hojas, sumado a que el algoritmo de backtracking retrocede en el árbol cuando llega en particular a una hoja, entonces por cada hoja que se agrega se realizan 2 pasos. Por lo tanto, si para el caso j se tardó k pasos, entonces en el caso $j + 1$ se tardarán $k + 2((j - 1) + j)$ pasos.

Para confirmar esto, veamos por ejemplo los casos $i = 2$ e $i = 3$. Para el primer caso se tardan 2 pasos. Para el caso de 3 investigadores, se tarda $2+2((2-1)+2) = 8$ pasos. Para $i = 4$ se tarda entonces $8 + 2((3 - 1) + 3) = 18$ pasos. Siguiendo así, conseguimos una fórmula recursiva p_i que nos dice la cantidad de pasos que

toma el algoritmo de backtracking para i investigadores e $\binom{i}{2}$ amistades (el peor caso). Ésta es:

$$p_1 = 0;\ p_i = 2(i-2+i-1) + p_{i-1} = 4i - 6 + p_{i-1}\ \forall i \geq 2$$

Para encontrar una fórmula cerrada a esta sucesión, la expandimos:

$$p_i = 4i - 6 + 4(i-1) - 6 + 4(i-2) - 6 + \ldots + 4(i-k) - 6 + \ldots + 0 =$$

$$4\sum_{j=1}^{i-1}(i-j+1) - 6(i-1) = 4\left((i-1)i + (i-1) - \frac{(i-1)i}{2}\right) - 6i + 6 =$$

$$4\left(\frac{(i-1)i}{2} + (i-1)\right) - 6i + 6 = 2i^2 - 4i + 2$$

Luego, $p_i = 2i^2 - 4i + 2$. Pero veamos que esto se puede mejorar todavía si descomponemos p_i como producto de dos factores:

$$p_i = 2i^2 - 4i + 2 = 2(i-1)(i-1) = \frac{i(i-1)}{2}\frac{4(i-1)}{i}$$

Resolviendo esto, nos queda:

$$p_i = \frac{i(i-1)}{2}\frac{4(i-1)}{i} = \frac{i(i-1)(i-2)!}{2(i-2)!}\frac{4(i-1)}{i} = \frac{i!}{2!(i-2)!}\frac{4(i-1)}{i} =$$

$$\binom{i}{2}\frac{4(i-1)}{i} \leq \binom{i}{2}i$$

ya que $\frac{4(i-1)}{i} \leq i$ para todo $i \geq 0$.

Por lo tanto, como p_i era la cantidad de pasos del algoritmo de backtracking (`obtenerEntusiasmoAux`), y probamos que $p_i \leq \binom{i}{2}i = |amistades||investigadores| \in \mathcal{O}(|amistades|*|investigadores|)$, entonces la complejidad de `obtenerEntusiasmoAux` es $\mathcal{O}(|amistades| * |investigadores|) + \mathcal{O}(|investigadores|) = \mathcal{O}(|amistades| * |investigadores|)$ (el segundo sumando se debe a la complejidad del primer `if`, que sólo se ejecuta una vez en todo el algoritmo).

Juntando todo, se tiene que $T_{oE}(a) = |investigadores|+|amistades|+|investigadores|* |amistades| + |investigadores| * |amistades|$

Entonces `obtenerEntusiasmo` es $\mathcal{O}(|investigadores| * |amistades|)$.

Caso en que no se encuentra una solución

Si el algoritmo no encuentra un camino de p a q en el grafo, significa que, o bien q no es amigo (directo o indirecto) de p pero sí de algún otro investigador, o q no es amigo de nadie. Si ocurre lo primero, naturalmente la complejidad será mejor que en el segundo caso, ya que para una cantidad x de investigadores (suponiendo que queremos tener la mayor cantidad posible de amistades entre los investigadores), en el primer caso p tendrá menos amigos (directos o indirectos) que en el segundo, donde en este último hay $\binom{i-1}{2}$ amistades (siempre pensando en

el peor caso del algoritmo de backtracking). Hecho este razonamiento, analizamos entonces el segundo caso, es decir, cuando q está "aislado" (no es amigo de nadie).

Nuevamente, supongamos entonces que tenemos la mayor cantidad posible de amistades entre todos los investigadores salvo q. Si tenemos i investigadores, entonces esta cantidad es $\binom{i-1}{2}$, ya que corresponde a la cantidad de maneras de agrupar a los $i-1$ investigadores salvo q de a pares.

Como hicimos en el caso en que el algoritmo encuentra una solución, empecemos por ver cuántos movimientos hay que hacer en el árbol de backtracking para una cantidad i de investigadores, donde se tiene $\binom{i-1}{2}$ amistades. Se quiere ir de p a q (que no va a ser posible). Por simplicidad, denotaremos a los i investigadores con las primeras i letras del abecedario, y p y q son, respectivamente, la primera e i-ésima letra del abecedario. Separamos de nuevo en casos:

- $i = 3, p = A, q = C$; investigadores: A, B, C; amistades: (A,B)
- $i = 4, p = A, q = D$; investigadores: A, B, C, D; amistades: (A,B); (A,C); (B,C)
- $i = 5, p = A, q = E$; investigadores: A, B, C, D, E; amistades: (A,B); (A,C); (A,D); (B,C); (B,D); (C,D)
- $i = 6, p = A, q = F$; investigadores: A, B, C, D, E, F; amistades: (A,B); (A,C); (A,D); (A,E); (B,C); (B,D); (B,E); (C,D); (C,E); (D,E)

Notemos que el caso $i = 2$ no lo contemplamos, ya que la lista de amistades sería vacía (y ni siquiera llega a ejecutarse este algoritmo). Para $i = 3$, se requieren 2 pasos; para $i = 4$, 12 pasos; para $i = 5$, 24 pasos; y para $i = 6$, 40 pasos. En general, para i investigadores y $\binom{i-1}{2}$ amistades, vemos que de un caso donde hay $j \geq 4$ investigadores a otro donde hay $j+1$, se agregan $(j-1)+(j-1)$ nodos. Además, por cada nodo que se agrega se realizan 2 pasos. Por lo tanto, si para el caso j se tardó k pasos, entonces en el caso $j+1$ se tardarán $k+2((j-1)+(j-1))$ pasos.

La fórmula recursiva p_i que nos dice la cantidad de pasos que toma el algoritmo de backtracking para i investigadores e $\binom{i-1}{2}$ amistades (el peor caso) es:

$$p_3 = 2; p_4 = 12 \; p_i = 2(i-2+i-2) + p_{i-1} = 4i - 8 + p_{i-1} \; \forall i \geq 5$$

Para encontrar una fórmula cerrada a esta sucesión, la expandimos:

$$p_i = 4i - 8 + 4(i-1) - 8 + 4(i-2) - 8 + \ldots + 4(i-k) - 8 + \ldots =$$

$$4\sum_{j=1}^{i-1}(i-j+1) - 8(i-1) = 4\left((i-1)i + (i-1) - \frac{(i-1)i}{2}\right) - 8i + 8 =$$

$$4\left(\frac{(i-1)i}{2} + (i-1)\right) - 8i + 8 = 2i^2 - 6i + 4$$

Luego, $p_i = 2i^2 - 6i + 4$. Descomponemos p_i como producto de dos factores:

$$p_i = 2(i-1)(i-2) = 4\frac{(i-1)(i-2)}{2} = 4\frac{(i-1)(i-2)(i-3)!}{2!(i-3)!} = 4\frac{(i-1)!}{2!(i-3)!} =$$

$$4\binom{i-1}{2} \in \mathcal{O}\left(\binom{i-1}{2}\right) = \mathcal{O}\left(|amistades|\right)$$

Entonces, sabemos que el algoritmo `obtenerEntusiasmoAux` toma tiempo $\mathcal{O}(|amistades|)+\mathcal{O}(|investigadores|)$, siendo el segundo sumando la complejidad del primer `if` que sólo se ejecuta una vez en todo el algoritmo. Y además sabemos que la cantidad de pasos, en peor caso, que se hace en el árbol de backtracking es exactamente el cuádruple de la cantidad de amistades máxima entre todos los investigadores excepto q. O sea es lineal en la cantidad de amistades.

Por lo tanto, se tiene que $T_{oE}(a) = |investigadores|+|amistades|+|investigadores|*|amistades| + |amistades|$

Entonces `obtenerEntusiasmo` es $\mathcal{O}(|investigadores| * |amistades|)$.

Concluimos, luego de haber analizado los 2 casos para el `obtenerEntusiasmoAux` que la complejidad del método principal `obtenerEntusiasmo` es $\mathcal{O}(|investigadores| * |amistades|)$

Problema 3

Introducción

En este problema se pretende, dado un listón de una determinada longitud y una serie de cortes a realizar a lo largo de éste, minimizar el costo final del trabajo, sabiendo que el costo de cada corte es proporcional al largo del listón a cortar en cada momento. El algoritmo que resuelva este problema debe devolver la secuencia óptima, es decir, el orden en el cual deben hacerse los cortes para lograr lo pedido.

Algoritmo propuesto

Para el algoritmo del ejercicio 3 tuvimos varios inconvenientes hasta que llegamos a la solucion deseada. Las clases de "soporte' para este ejercicio fueron:

- `Cortes`: La cual representa una secuencia de cortes y el costo asociado a esa secuencia de cortes.
- `Liston`: La cual representa un liston con sus cortes realizados. Para representar los cortes realizados, tenemos una lista de enteros donde cada entero (salvo el ultimo que es la longitud) representa un corte realizado al liston. Las funciones que presenta son las siguientes:
  - `CortarListon`: Dada una posicion para cortar, se realiza el corte correspondiente en el liston, y se devuelve cual fue el costo.
  - `deshacerCorte`: Dada una posicion, se deshace el corte que se habia realizado en el liston (la funcion requiere que la posicion de corte haya sido un corte realizado anteriormente).

Primero, cumpliendo con la frase"Make it work. Make it right. Make it fast." empezamos creando un algoritmo similar al del ejercicio 2, realizando backtracking (este se encuentra en el paquete versionInicial). Una vez que lo hicimos y vimos que funcionaba, quisimos "podar" ese backtracking haciendo que si se paso del minimo que teniamos hasta el momento, que no siga recorriendo la rama. Al querer agregar esto, nos dimos cuenta que, de la forma que lo habiamos implementado, no podiamos "cortar" el backtracking (o no se nos ocurrio como), por lo que lo tuvimos que re-pensar de nuevo para sí esta vez poder cortar si ya se habia superado el minimo encontrado hasta el momento. En la version inicial, el pseudocodigo seria lo siguiente:

- Creamos variable local "costoMinimo" inicializada en ∞
- Para cada corte a realizar
  - Nos guardamos el costo
  - Llamamos recursivamente para obtener el minimo costo, pero ahora con ese corte menos en la secuencia de cortes a realizar
  - Sumamos el costo de la llamada recursiva anterior para ver cuanto nos dio el costo total.

- Si el costo que nos dio era menor al costo que teniamos guardado en costoMinimo, entonces lo reemplazamos con el nuevo valor.

De esta forma, al ser costoMinimo una variable local de la funcion, nunca podabamos el arbol porque las llamadas recursivas no sabian si ya se habia alcanzado el minimo o no. Si poniamos al minimo en una variable global, en cada llamada recursiva se iba a estar pisando el valor ya que siempre el costo de un solo corte x ejemplo va a ser menor (o igual) al de toda la secuencia de corte. De esta forma, pensamos como solucionar este problema, llegando a la solucion siguiente para el ej3 con backtracking: Agregamos 2 parametros mas a la funcion, uno con los cortes realizados para llevar cuenta de ellos, y el otro con el costo parcial hasta el momento, asi cada vez que se hace una llamada recursiva, se sabe cuanto costo se venia llevando hasta el momento, y si es necesario parar o no porque se supero el minimo. A continuacion, una explicacion breve del pseudocodigo:

- Creamos variable local "costoMinimo" inicializada en ∞
- Para cada corte a realizar
  - Realizo el corte, y le sumo el costo parcial que tenia como parametro. Ademas agrego el corte a los cortes reliazados que tenia como parametro de la funcion y saco del vector de cortes a realizar el corte en cuestion.
  - Si el costo resultante es mayor que costoMinimo, paro.
  - Sino, si ya no quedan cortes a realizar, me guardo el nuevo costoMinimo con su repectivo cortes realizados
  - Si en vez de eso, todavia quedan cortes a realizar, llamo recursivamente a la funcion.

Ademas del algoritmo, implementamos (a pedido de un ayudante) validaciones sobre la entrada y tests a cada una de las funciones que habiamos hecho, lo cual nos llevo bastante tiempo (ademas que ninguno del grupo era programador de Java) y nos desenfocamos del problema principal.

Este algoritmo igualmente seguia siendo muy lento ya que, a pesar de que ahora si estabamos podando, igualmente repetiamos calculos. Es decir, si teniamos la secuencia de cortes [2,4,7] y un liston de longitud 10, si realizabamos el corte en 4, era lo mismo si haciamos despues el corte en 2 o 7, pero en nuestro algoritmo lo calculabamos 2 veces. Una vez que vimos programacion dinamica en clase, intentamos realizar una version del algoritmo con programacion dinamica sin exito, hasta que llegamos a la clase practica de programacion dinamica en la ultima semana justo antes de entregar el tp. En ese momento (y a las corridas porque todavia nos faltaba gran parte del informe y tests pensando que todavia teniamos una semana) llegamos a la solucion de programacion dinamica. La idea del algoritmo es la siguiente:

Primero agregamos a la secuencia de cortes a realizar (llamemosla "cortes") el 0 al comienzo, y la longitud al final. Tenemos una matriz de n x n (donde n es la longitud

de “cortes”) llamada“costos”, donde la semantica de costos[i][j] = “costo minimo entre los cortes consecutivos que estan entre los indices i y j de la secuencia de cortes a realizar”. Matematicamente, $costos[i][j] = cortes(j) - cortes(i) + min_{i<k<j}\{costos[i][k] + costos[k][j]\}$ donde cortes es la secuencia de cortes a realizar. De esta forma, el resultado que queremos lo vamos a tener en costos[0][n-1]. Ademas, inicializamos a costos[i][j] = $0 \; \forall i < j / i + 1 = j$

Mostremos, para un mayor entendimiento, como funciona el algoritmo con un ejemplo: cortes = 0,4,7,20,48 (longitud seria 48 en este caso)

El algoritmo empieza agarrando secuencias de corte de tamaõ 3:

costos[0][2] = cortes(2) - cortes(0) + 0 = 7 (el minimo, que es costos[0][1] + costos[1][2], da 0 porque los habiamos asignado asi anteriormente..los siguientes 2 pasos son analogos)
costos[1][3] = cortes(3) - cortes(1) + 0 = 16
costos[2][4] = cortes(4) - cortes(2) + 0 = 41

Ahora, seguimos con las secuencias de corte de tamaño 4:

costos[0][3] = cortes(3) - cortes(0) + $min_{0<k<3}$ {costos[0][k] + costos[k][3]} = 20 + $min_{0<k<3}\{costos[0][k] + costos[k][3]\}$

Miremos ahora que nos da el minimo:

costos[0][1] + costos[1][3] = 0 + 16 (ambos ya calculados anteriormente..la magia de la programacion dinamica!)
costos[0][2] + costos[2][3] = 7 + 0 (ambos ya calculados anteriormente de nuevo)

Mirando cual es el minimo nos da que es 7, por lo tanto costos[0][3] = 20 + 7 = 27

costos[1][4] = cortes(4) - cortes(1) + $min_{1<k<4}\{costos[1][k] + costos[k][4]\}$ = 44 + $min_{1<k<4}\{costos[1][k] + costos[k][4]\}$

Haciendo de la misma forma que antes, hallamos que el minimo nos da 16, por lo tanto costos[1][4] = 60

Por ultimo, llegamos a la ultima longitud, de tamaño 5, que es la solucion que queriamos:

costos[0][4] = cortes(4) - cortes(0) + $min_{0<k<4}\{costos[0][K] + costos[k][4]\}$ = 48 + $min_{0<k<4}\{costos[0][K] + costos[k][4]\}$

Miremos que nos da el minimo:

costos[0][1] + costos[1][4] = 0 + 60 = 60
costos[0][2] + costos[2][4] = 7 + 41 = 48
costos[0][3] + costos[3][4] = 27 + 0 = 27

Miramos cual es el minimo, el minimo de los 3 es 27, por lo tanto nos queda que costos[0][4] = 48 + 27 = 75, que es efectivamente el costo minimo posible del problema.

Para cumplir con lo pedido en realidad, como no nos pedian el costo minimo sino que la secuencia que lograba ese costo minimo, en la matriz nos vamos guardando no solo el costo minimo, sino que la secuencia de cortes que efectivamente lo logra.

Por ultimo, para asegurarnos que los resultados obtenidos con este algoritmo eran los deseados, creamos el test `testPDContraBacktrack`, que generaba numeros al azar, y chequeaba que los costos minimos obtenidos con este algoritmo de programacion dinamica eran los mismos que los de backtracking.

Complejidad

En esta sección hablaremos de la complejidad temporal del algoritmo `hallarCortesMin`. Hallaremos una cota y justificaremos por qué la alcanza.

Si nos referimos al código Java de `hallarCortesMin`, observamos que en la primera línea se crea un vector de enteros (operación de tiempo constante) y luego se llama a otro método `hallarCortesMin` que devuelve un objeto de tipo CORTES, del cual se obtiene el orden en que hay que hacer los cortes para asegurar que el costo sea mínimo.

Es esta última función la que hace backtracking en un árbol, cuyas ramas son todas las formas de ordenar la secuencia de cortes pasada como parámetro (`cortes_a_realizar`) al método principal `hallarCortesMin` (el que devuelve el vector correspondiente a la secuencia óptima). Si probamos con distintas secuencias de cortes, veremos que el peor caso (aquel que hace menos cantidad de podas) es cuando los cortes se distribuyen proporcionalmente en el listón. Por ejemplo, si tomamos un listón de longitud 10, entonces el peor caso para $n = 4$ será $[2, 4, 6, 8]$, o para $n = 5$ será $[1, 3, 5, 7, 9]$, etc. Podríamos afirmar que éste es el peor caso, ya que cuando tenemos cortes desproporcionados (por ejemplo una secuencia $[1, 2, 3]$ para un listón de longitud 10), se logra podar varias ramas (en general hojas) del árbol de backtracking. Esto sucede ya que, con una configuración de cortes desproporcionados, se llega varias veces a nodos cuyo costo acumulado ya supera el costo mínimo, y por lo tanto ya no se debe continuar por esa rama. Un ejemplo de esto sería un listón también de longitud 10 y cortes $[4, 5, 7, 8]$, donde vemos que los intervalos que forman $([0, 4]; [4, 5]; [5, 7]; [7, 8]; [8, 10])$ son de longitudes distintas (cosa que no ocurre cuando los cortes son proporcionales), y llega a haber 6 podas en total (a diferencia de un caso donde los cortes son $[2, 4, 6, 8]$, proporcionales, y llega a haber 4 podas solamente).

Vemos, a partir del código Java, que la complejidad total del método `hallarCortesMin` depende de la complejidad de tres métodos (ya que el resto son operaciones de tiempo constante). Estos tres métodos son `cortarListon`, `deshacerCorte` y `remove` (operación de VECTOR), las cuales son todas de tiempo lineal en la cantidad de cortes (n). Además, haciendo los árboles de backtracking para cada cantidad de cortes, se observa la siguiente cantidad de pasos para cada valor de n (y para el peor caso, es decir tomando cortes proporcionales[1]):

- $n = 2$, 8 pasos

[1] En este ejemplo tomamos un listón de longitud 10, y vamos dividiendo el listón en partes iguales para determinar cuáles serán los cortes

- $n = 3$, 30 pasos
- $n = 4$, 120 pasos
- $n = 5$, 544 pasos
- $n = 6$, 3320 pasos

  ...

Pero a esto habrá que sumarle la complejidad de las tres operaciones que comentamos recién. Entonces, la complejidad total en peor caso de `hallarCortesMin` será:

- $n = 2$, 8*2*3 pasos
- $n = 3$, 30*3*3 pasos
- $n = 4$, 120*4*3 pasos
- $n = 5$, 544*5*3 pasos
- $n = 6$, 3320*6*3 pasos

  ...

O sea que la complejidad de `hallarCortesMin` es claramente exponencial.

Veamos ahora la complejidad de la version con `Programacion Dinamica`. La podemos obtener razonando de la descripcion del algoritmo:

- Primero tenemos un ciclo en donde vamos aumentando en 1 la longitud de los cortes consecutivos que vamos agarrando (comenzando desde 2, y hasta n-1).
  - Una vez seleccionada la longitud de la secuencia de cortes, miramos cada una de las secuencias de cortes que podemos agarrar.
    - Por cada secuencia de cortes, tenemos que analizar buscar el minimo en nuestra matriz "costos". Mientras mas grande sea la longitud de cortes, mas grande es la cantidad de datos sobre la cual tenemos que buscar el minimo.

A simple vista tenemos 3 for's anidados, lo que nos da una idea de un orden cubico. Pero miremos un poco mas en detalle:

En el ejemplo mostrado en el desarrollo, vamos haciendo la siguiente cantidad de operaciones en cada ciclo:

Por cada longitud: costosARellenar * iteracionesParaEncontrarMinimo

Cuando la longitud era 3: (5-2) * 1 = 3 * 1 (correspondientes a: costos[0][2], costos[1][3], costos[2][4]..y por cada uno teniamos un solo minimo para iterar)
Cuando la longitud era 4: (5-3) * 2 = 2 * 2
Cuando la longitud era 5: (5-4) * 3 = 1 * 3

Viendo el ejemplo, es facil de ver que la cantidad de operaciones es del orden de: $\sum_{i=2}^{n-1}(n-i)(i-1)$. Simplifiquemos un poco la ecuacion:

$$\sum_{i=2}^{n-1}(n-i)(i-1) = \sum_{i=1}^{n}((n-i)(i-1)) - n = \sum_{i=1}^{n}(ni - i^2 + i - n) - n = \sum_{i=1}^{n} ni - \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i - \sum_{i=1}^{n} n - n = n\frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} - n^2 - n = \frac{n(n+1)(n+1)}{6} + \frac{n(n+1)}{2} - n^2 - n = \frac{n(n+1)(n+1)}{6} + \frac{3(n-n^2)}{6} - n = \frac{n^3 - n^2 + 4n}{6} - n = \frac{n^3 - n^2 - 2n}{6} = \mathcal{O}(n^3)$$

2. Resultados

Problema 1

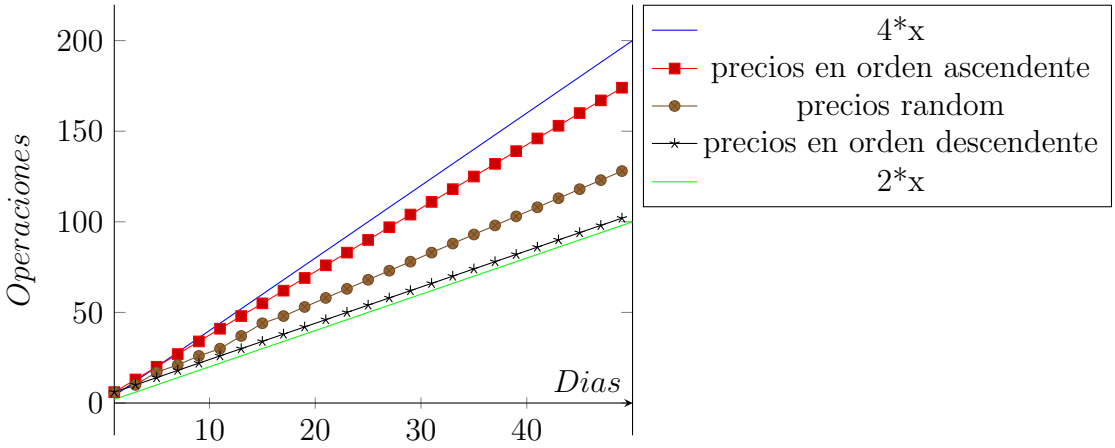

Figura 1: Variacion de la cantidad de operaciones variando la cantidad de dias.

Para los tests de velocidad tomamos en cuenta lo siguiente: La minima cantidad de operaciones se realiza cuando la sucesion de precios esta ordenada en forma descendente. Esto es ya que al estar ordenado en forma descendente, el algoritmo solo asigna el nuevo valor en cada iteracion a la variable "precioDeCompra". Por el otro lado, la maxima cantidad de operaciones se realiza cuando la sucesion de precios esta ordenada en forma creciente. Esto sucede porque el algoritmo va a realizar algunas operaciones más que en el caso anterior, ya que ademas va a tener que preguntar si la ganancia obtenida es mayor, y hacer la asignacion correspondiente al efectivamente ser mayor. Por otro lado, cualquier sucesion de precios no ordenada va a caer entre medio de estas 2 sucesiones de puntos. De esta forma, podemos concluir como vemos en la figura, que podemos acotar con una funcion lineal tanto por debajo como por arriba a nuestros resultados, lo cual se corresponde con la complejidad vista en el punto anterior (θ(n)).

Problema 2

Como se mencionó anteriormente, puede existir o no una solución para el problema que intenta resolver el algoritmo propuesto. Los test elegidos para evaluar la función principal *obtenerEntusiasmo*, representan ambos casos. Si hay solución, entonces lo que se evalúa es poder llegar de q a p de la manera óptima, y si no la hay, se devolverá un 0.

En cuento a los tiempos de corrida del algoritmo, en los siguientes gráficos se comparan con la complejidad calculada en la sección anterior.

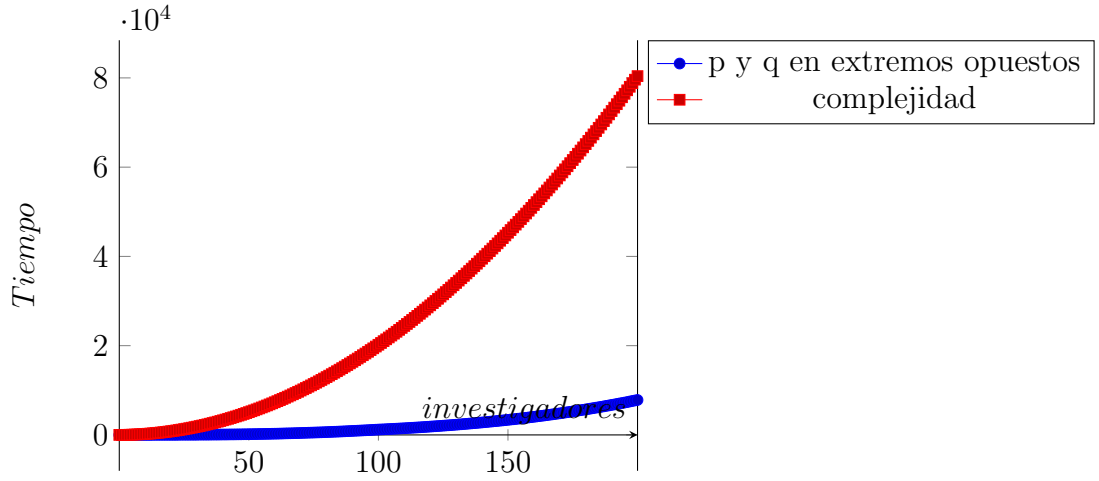

Figura 2: q en extremo opuesto a p

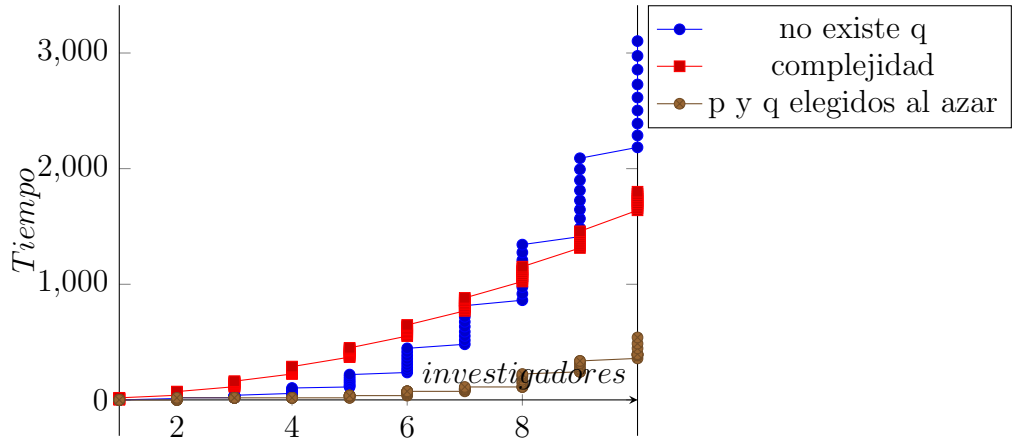

Figura 3: no existe q entre los investigadores

Los tiempos medidos fueron el promedio tomado de sucesivas ejecuciones del algoritmo. Éstos fueron tomados a medida que se fue incrementando la cantidad de investigadores. La primer figura corresponde al caso en que se tienen al investigador p y q en extremos opuestos. Se lo considera como un peor caso porque para llegar a la solución se tienen que recorrer todos los nodos del árbol, que a los sumo son la cantidad total de investigadores. Se puede observar que la complejidad medida está siempre por encima de la curva del tiempo medido de corrida. En la segunda figura se comparan con la complejidad calculada, un caso al azar y un peor caso. Para calcular el tiempo de ejecución nuevamente se vuelve a incrementar la cantidad de investigadores, pero esta vez, también se aumenta la cantidad de amistades. Es decir que no tenemos como en la figura anterior, una sóla rama para recorrer, si no que por cada investigador, existen $|investigadores| - 1$ posibles ramas. El peor caso ocurre cuando no existe q porque se van a estar recorriendo la mayor cantidad de nodos posibles. Como no se va a encontrar q el algoritmo termina cuando no existen más caminos por tomar. Para el caso al azar, se tomaron los nodos p y q de manera aleatoria. Tendrámos los tres casos:

- p no existe, entonces la ejecución es rápida porque no existen amigos de p que recorrer.

- q no existe, corresponde a peor caso.
- p y q existen.

Se puede ver en el gráfico que hasta un cierto punto, la complejidad calculada se mantiene por encima de los tiempos medidos tal como ocurría en el caso anterior. Al aumentar la cantidad de investigadores y amistades entre ellos (se incrementan la cantidad de ramas del árbol), la complejidad queda por debajo de los tiempos calculados pero con valores muy cercanos entre si, por lo que intuimos que si lo multiplicamos por una constante a la complejidad efectivamente nos quedaria por arriba de los resultados, como desarrollamos en la seccion de complejidad.

Problema 3

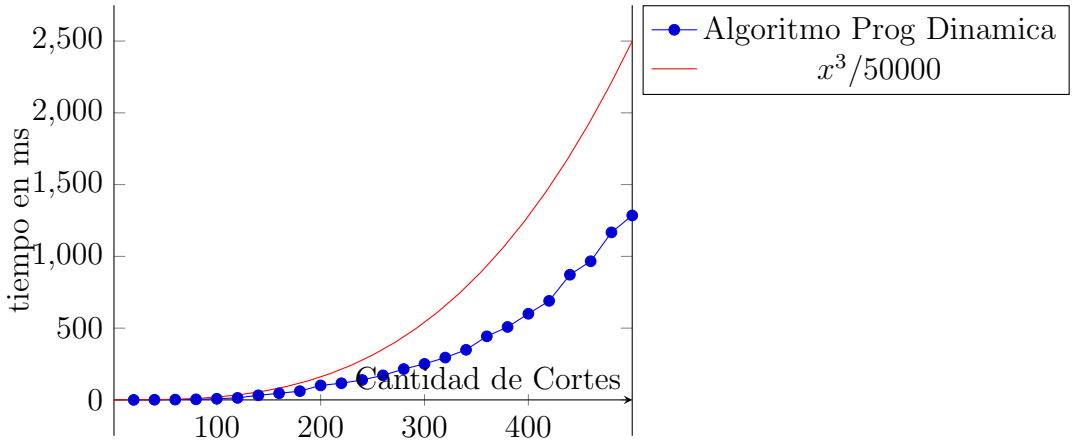

Figura 4: Crecimiento del tiempo con respecto al tamaño de la entrada (cantidad de cortes a realizar) para la vesion con Programacion Dinamica

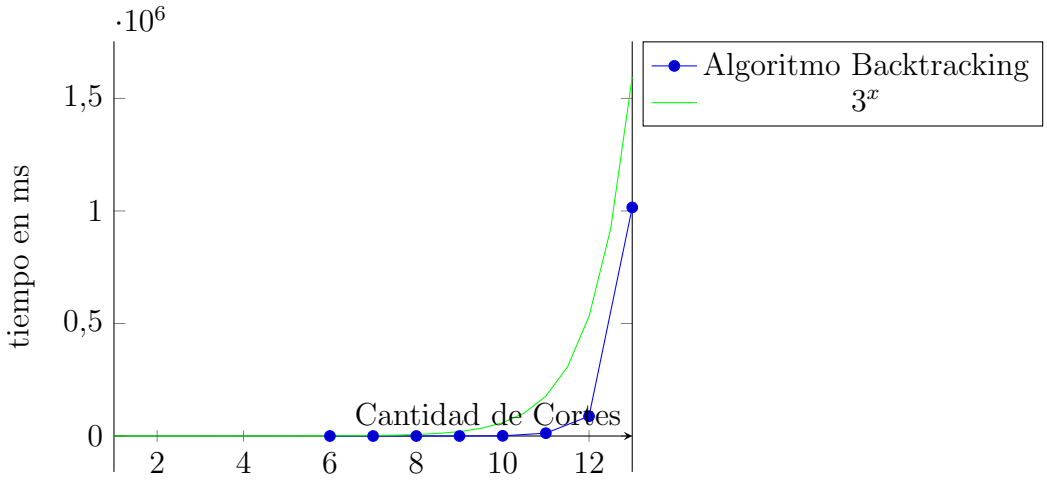

Figura 5: Crecimiento del tiempo con respecto al tamaño de la entrada (cantidad de cortes a realizar) para la version con Backtracking

Los graficos confirman nuestra sospecha en el analisis de complejidad de que la version de backtracking era del orden exponencial (graficamente, pareceria ser $\mathcal{O}(3^n)$

) y confirman que nuestra cota hallada para el caso de programacion dinamica puede ser correcta ($\mathcal{O}(n^3)$)

3. Conclusiones

Problema 1

La solucion de este problema resutlo ser mucho mas elegante de lo que pensabamos en un principio. Ademas, destacamos que con este problema, pudimos hacer un analisis de la complejidad contando la cantidad de operaciones en vez de contando el tiempo, lo cual resulta mucho mas preciso.

Problema 2

Este ejercicio nos ayudo a ver como podemos modelar un grafo y realizar backtracking sobre el. Ademas, realizando una modificacion y ayudandonos de una estructura de soporte, vimos como pudimos reducir la complejidad de backtracking de exponencial a polinomial.

Problema 3

Este problema nos sirvio, mas que nada, para poder razonar como realizar una implementacion de programacion dinamica, partiendo de un problema que a priori, no veiamos que se repetian muchos calculos. Una vez que testeamos el algoritmo de backtracking y vimos que con una secuencia de cortes de mas de 10 se volvia muy lento, intentamos hacerlo mas eficiente encontrando una solucion en la programacion dinamica. Y que solucion! Pasamos de una complejidad exponencial, a una complejidad polinomial (y los tests nos ayudaron a ver algo que se nos dijo siempre en clase pero que nunca lo habiamos comparado nosotros mismos en una pc...el porque se dice que un problema esta bien resuelto si es polinomial, y no lo esta si es exponencial). Y no solo eso, sino que hasta el codigo quedo mucho mas conciso y "lindo" a la vista. Concluimos entonces que con este problema, logramos ver las grandes virtudes de la programacion dinamica.

4. Referencias

A lo largo de la confección del trabajo práctico, se consultó la siguiente bibliografía:

[1] JavaTM 2 Platform, Standard Edition, v 1.4.2 API Specification (http://docs.oracle.com/javase/1.4.2/docs/api/)