ISIS 1105 – 2020-2

Problema B – Orden de cortes más barato

0. Identificación

- Julián Oliveros – je.oliverosf – 201821595
- Camilo Rozo – ce.rozob – 201820147

1. Algoritmo de Solución

Explicación de elección del algoritmo

Al principio era notorio que había una cierta similitud entre este problema y el Cutting-rod problem que aparece en el libro de Cormen, que también es un problema de optimización y tienen un contexto vagamente similar, pero es de maximización y no minimización como este. Razones por las cuales la idea que permaneció en todo momento fue la de implementar una solución utilizando programación dinámica. Esto tuvo más sentido cuando en el ejemplo del enunciado se vio un poco la recursividad del problema, porque al cortar una vez, el problema restante son dos versiones (izquierda y derecha del corte) más pequeñas del mismo problema de optimización. Creemos que esta elección fue la más viable porque usando las técnicas en clase para implementar algoritmos de programación dinámica es posible conseguir soluciones que siempre hallen la respuesta óptima en un tiempo corto considerando los límites que nos permitieron asumir del enunciado del problema (tamaño de la varilla entre 1 y 100, puntos de corte ordenados ascendentemente y costo del corte basado en la longitud y no arbitrario).

Documentación formal del algoritmo

Algoritmo principal, ProblemaB

E/S	Nombre	Tipo	Descripción
E	n	int	Tamaño de la varilla a cortar
E	p	Array [0,m) of int	Lista de puntos de corte
S	a	int	Costo mínimo de cortar la varilla de largo n en los puntos dados por p.

$$Precondición: \{(\forall i | 0 \leq i < m : p[i] < p[i+1]) \wedge n \geq 0\}$$

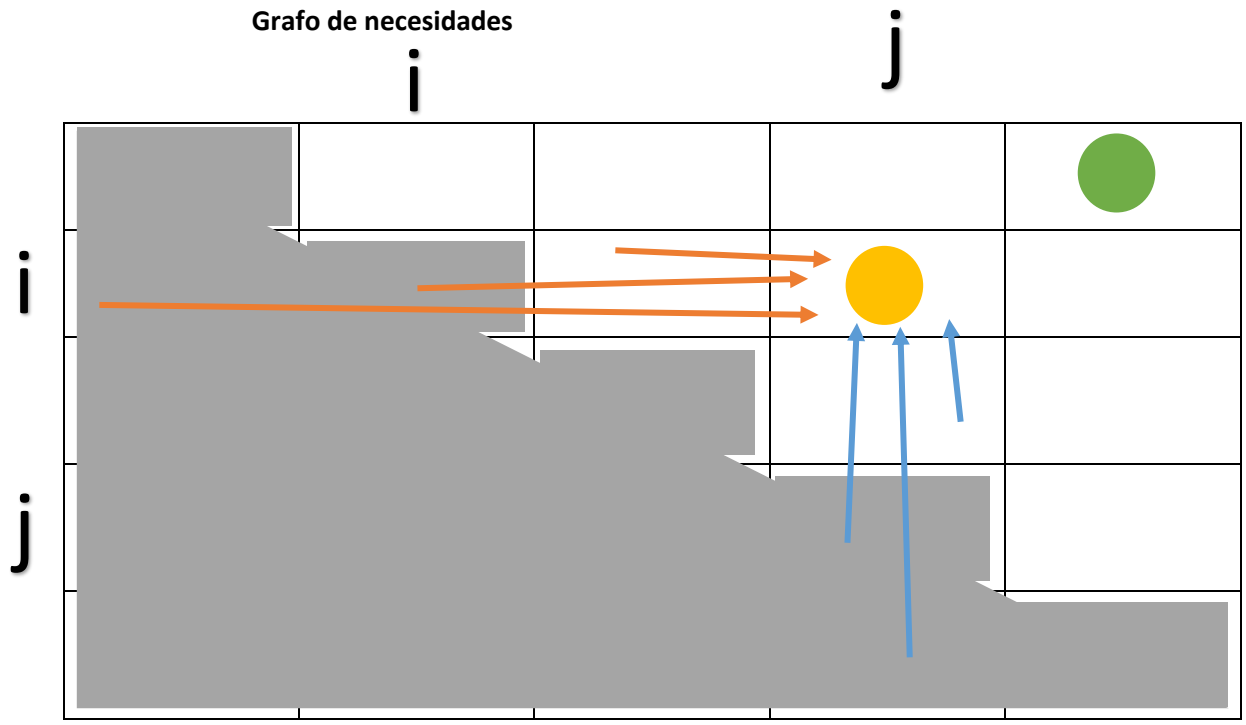
Función por minimizar y ecuación de recurrencia

Definimos para $0 \leq i \leq n$ y $0 \leq j < n$, que $T(i,j)$ sea la función que calcula el costo mínimo de cortar la "sub-varilla" que inicia en i y termina en j. Así, la solución al problema sería T (0, n). Así, se puede plantear la siguiente ecuación de recurrencia

$$T(i,j) = 0 \; si \; i = j \; || j \leq i$$

$$T(i,j) = (\min k | i < p[k] < j : T(i,p[k]) + (i-j) + T(p[k],j)) \; si \; i \neq j$$

$$Postcondición: \{a = T(0,n)\}$$



Subrutina, método minPositive

Contexto:

Para una parte de la recurrencia hay que poder diferenciar entre una posición de la matriz t[i][j] que estoy visitando pero aún no he calculado, y una posición de la matriz t[i][j] que ya calculé, debido a que es un problema de minimización, es posible que el valor 0 sea posible de obtener, por ejemplo, para un caso donde no tenga sentido preguntarse el costo de cortar desde i hasta j, ejemplo concreto: cortar entre 3 y 7 en los puntos 2, 9 y 15 no tiene sentido porque no corto nada y me cuesta 0, entonces, para diferenciar esos casos decidimos temporalmente marcar la casilla con -1, por lo cual la función min(int a, int b) de java nos daría un cálculo incorrecto por el número inicial -1, de modo que el método minPositive calcula el mínimo entre 2 enteros a y b, con la excepción de que alguno de los 2 sea negativo, en cuyo caso retorna el opuesto. Ejemplo: Math.min(-1,5)=-1, minPositive(-1,5)=5.

E/S	Nombre	Tipo	Descripción
E	a	int	entero
E	b	int	entero
S	m	int	Mínimo entre a y b, o el opuesto al que sea negativo.

2. Análisis de complejidades espacial y temporal

- Cálculo de complejidades y explicación de estas.

Algoritmo principal, problemaB

Instrucción	Símbolo	Constante
Asignación	$=$	k_1
Suma	$+$	k_2
new	new	k_3
Menor	$<$	k_4
Mayor igual	$\geq$	k_5
.length	$.length$	k_6
Mayor	$>$	k_7
AND lógico	$\&\&$	k_8
minPositive	N/A	k_9
Resta	$-$	k_{10}
Equivalencia	$==$	k_{11}
Incremento	$++$	k_{12}
Decremento	$--$	k_{13}

$$T(n,m) = k_1 + k_2 + k_2 + k_3 + k_1 + k_1 + k_1 + (n+1)(k_4 + k_6) + (3n)(k_5) + (n)(k_1) + (2n)(k_1 + k_1 + (m+1)(k_4 + k_6) + (m)(k_{12} + k_1 + k_7 + k_8 + k_4 + k_1 + k_9 + k_{10} + k_2 + k_2) + k_{11} + k_1) + (3n)(k_{11} + k_6 + k_{10}) + (n)(k_1 + k_{12} + k_1) + (2n)(k_{10} + k_{10})$$

$$T(n,m) = k_1 + k_2 + k_2 + k_3 + k_1 + k_1 + k_1 + nk_4 + k_4 + nk_6 + k_6 + 3nk_5 + nk_1 + 2nk_1 + 2nk_1 + 2nmk_4 + 2nk_4 + 2nmk_6 + 2nk_6 + 2nmk_{12} + 2nmk_1 + 2nmk_7 + 2nmk_8 + 2nmk_4 + 2nmk_1 + 2nmk_9 + 2nmk_{10} + 2nmk_2 + 2nmk_2 + 2nk_{11} + 2nk_1 + 3nk_{11} + 3nk_6 + 3nk_{10} + nk_1 + nk_{12} + nk_1 + 2nk_{10} + 2nk_{10}$$

$$T(n,m) = (4k_1 + 2k_2 + k_3 + k_4 + k_6) + n\big(9k_1 + 3k_4 + 3k_5 + 6k_6 + 7k_{10} + 5k_{11} + k_{12} + m(4k_1 + 4k_2 + 4k_4 + 2k_6 + 2k_{12} + 2k_7 + 2k_8 + 2k_9 + 2k_{10})\big)$$

$$\boldsymbol{Orden\ de\ complejidad\ estimado \approx O(n * m)}$$

$$Espacio(n,m) = O(n^2); matriz\ cuadrada\ de\ tamaño\ n$$

Subrutina, método minPositive

Instrucción	Símbolo	Constante	Cantidad ejecuciones
Menor	$<$	k_1	3

$$T(n) = 3 * k_1 \rightarrow O(1)$$

$$Espacio(n) = O(1)$$

3. Comentarios finales

- Observamos que el desempeño es aceptable dados los límites del enunciado

- En las casillas marcadas con -1 se consideró usar Integer.MAX_VALUE para hallar el mínimo, pero el algoritmo deja de dar la respuesta correcta por un motivo que desconocemos, además, en tareas/actividades pasadas nos han ocurrido problemas de Overflow relacionados a esta alternativa, por lo que decidimos no tenerlo en cuenta.
- Vemos que la matriz para implementar programación dinámica es triangular superior, por lo que creemos que es posible aprovecharse de esa propiedad para ahorrar memoria usando otra estructura de datos. Una idea posible es usar explícitamente un grafo de necesidades.
- Encontramos que las limitaciones establecidas en el enunciado reducen considerablemente la complejidad del problema, por lo que no tenerlas en cuenta e intentar hacer una solución que funcione en un caso más general puede resultar demasiado complicado.