Greedy: Shop

(https://codeforces.com/problemset/problem/521/D)

Carlos Luis Aguila Fajardo

1 Problema análogo

Se desea maximizar $\prod_{i=0}^{k} a_i$, el producto de los elementos del arreglo $a = \{a_1 \dots a_l\}$ utilizando m de las n posibles transformaciones $b = \{b_1 \dots b_n\}$, de la forma $b_i = (t_i, j_i, v_i)\,,\ t_i \in \{=, +, *\}$. Estas transformaciones pueden ser de tres tipos:

- Asignación $(=)$: $a^*_{j_i} = v_i$
- Adición $(+)$: $a^*_{j_i} = a_{j_i} + v_i$
- Multiplicación $(*)$: $a^*_{j_i} = a_{j_i} \cdot v_i$

de donde a^*_i denota el nuevo valor que tomará el arreglo a en la posición i.

2 Algoritmo greedy

2.1 Reducción a un solo tipo de transformación

Considerando que el objetivo del problema es optimizar una productoria de elementos, sería conveniente que las transformaciones realizadas sobre estos fueran también de forma multiplicativa. Por tanto, por simplicidad para las demostraciones posteriores, reduciremos todas las transformaciones disponibles a transformaciones multiplicativas.

2.1.1 Adición

Consideremos una transformación aditiva en la forma $b_i = (+, j_i, v_i)$, entonces son equivalentes las transformaciones:

$$a^*_{j_i} = a_{j_i} + v_i = a_{j_i} * \frac{a_{j_i} + v_i}{a_{j_i}} = a_{j_i} * v'_i \tag{1}$$

por tanto es equivalente b_i a $b'_i = (*, j_i, v'_i)$. Nótese que lo anterior solo se cumple cuando $a_{j_i} \neq 0$, lo cual ocurre siempre pues $1 \leq a_{j_i} \leq 10^6$ es una condición del problema original.

2.1.2 Adiciones en sucesión

Definición: Se denota $v(b) = v_0$ al valor de la transformación $b = (t, j, v_0)$. Similarmente, se define la operación $v(\{b_1 \dots b_k\}) = \{v(b_1) \dots v(b_k)\}$.

Definición: Se denota $f_j(t)$ al arreglo de los valores de las transformaciones de tipo t correspondientes al elemento a_j, ordenados de forma no creciente. Formalmente:

$$f_j(t) = \left\{ bs_i = (t, j, d_i) \in b \;\middle|\; d_i \leq d_k \;\forall\, k :\; 1 \leq k < i \right\}$$

Sea $f_i(+) = \{bs_1 \dots bs_p\}$, y $v(f_i(+)) = \{d_1 \dots d_p\}$ el arreglo de valores de las transformaciones aditivas correspondientes a a_i, entonces el resultado de aplicar todas las transformaciones de un prefijo de tamaño k de $f_i(+)$ se puede representar como:

$$a^*_i = a_i + d_1 + d_2 + \cdots + d_k = a_i + \sum_{j=1}^{k} d_j = a_i \cdot \frac{a_i + \sum\limits_{j=1}^{k} d_j}{a_i + \sum\limits_{j=1}^{k-1} d_j} \tag{2}$$

de forma que a partir de cada transformación aditiva bs_i es posible diseñar una multiplicativa que le corresponda:

$$b'_i = (*, i, r_k)\,, \quad r_k = \frac{a_i + \sum\limits_{j=1}^{k} d_j}{a_i + \sum\limits_{j=1}^{k-1} d_j} \tag{3}$$

2.1.3 Asignación

Similarmente al caso anterior, sea $b_i = (=, j_i, v_i)$ una transformación de asignación, se puede obtener una transformación multiplicativa de la forma siguiente:

$$a^*_{j_i} = v_i = a_{j_i} \cdot \frac{v_i}{a_{j_i}}$$

$$b'_i = (*,\ j_i,\ \frac{v_i}{a_{j_i}}) \tag{4}$$

Nótese que no es necesario generalizar los casos de asignación, ya que solo tiene sentido realizar, a lo sumo, una asignación por elemento a_i, puesto que cualquier transformación previa a una asignación se pierde luego de realizar esta.

2.2 Precondiciones

Considérese el arreglo de entrada $\{b_1 \dots b_n\}$ de las transformaciones posibles. Fue demostrado en secciones previas que es equivalente un arreglo $\{b'_1 \dots b'_n\}$ donde $\forall i : b'_i$ es multiplicativa. No ha sido profundizada, sin embargo, la condición necesaria para que se cumpla esta equivalencia.

2.2.1 Adición:

Consideremos la expresión (3) de las transformaciones aditivas; como fue mencionado anteriormente, esta expresión es válida para el caso en que se seleccione y aplique un prefijo de tamaño k de las transformaciones disponibles de $f_i(+)$ que, al estar ordenadas de manera no creciente por valor, son las k mayores transformaciones aditivas que se pueden aplicar a a_i.

De ser aplicada, en un paso intermedio, alguna transformación no aditiva sobre a_i esta expresión pasa a ser falsa. Sin embargo, supongamos que se aplica una transformación multiplicativa sobre a_i en dicho contexto, existen entonces $0 \leq p, q \leq k$ tal que:

$$a^*_i = \left(a_i + \sum_{j=1}^{p} d_j\right) \cdot v_q + \left(a_i + \sum_{j=p+1}^{k} d_j\right)$$

Sea $a^{*\prime}_i$ la aplicación de las mismas transformaciones, pero realizadas todas las aditivas primero, es fácil observar que:

$$\left(a_i + \sum_{j=p+1}^{k} d_j\right) \leq \left(a_i + \sum_{j=p+1}^{k} d_j\right) \cdot v_q$$

$$\left(a_i + \sum_{j=1}^{p} d_j\right) \cdot v_q + \left(a_i + \sum_{j=p+1}^{k} d_j\right) \leq \left(a_i + \sum_{j=1}^{k} d_j\right) \cdot v_q$$

$$a^*_i \leq a^{*\prime}_i \tag{5}$$

entonces siempre es correcto aplicar todas las transformaciones aditivas seleccionadas antes que las multiplicativas, por lo cual siempre es correcta la proposición de equivalencia (3).

2.2.2 Asignación:

Supongamos que existen asignaciones asociadas a a_j, sea $b_i = (=, j, v_i)$ la mayor de ellas. Solo tiene sentido aplicar la transformación b_i si se cumple que $b_i > a_j$ y aplicarla antes que cualquier otra transformación. Intuitivamente si no se cumpliera lo primero el resultado sería un valor menor que el previo a aplicar la transformación; en el segundo caso, de no aplicarse la asignación primero se tendrían un conjunto de transformaciones iniciales cuya selección no aportaría nada. Por tanto es válido aplicar la equivalencia (4) siempre y cuando se asigne solamente, si acaso, en la primera transformación.

2.3 Planteamiento del algoritmo

Sea $b' = \{b'_1 \dots b'_n\}$ el resultado de ordenar por valor no creciente las transformaciones $(*)$ correspondientes a las transformaciones de entrada b, independientemente de los a_i. El algoritmo greedy pretende maximizar la función objetivo mediante dos criterios:

1. La selección de las transformaciones que mayor valor multiplicativo aporten. Es decir, el algoritmo selecciona siempre un prefijo tamaño m de b'.

2. Al aplicar las transformaciones $\{b'_1 \dots b'_m\}$, se realiza en orden respecto a sus tipos originales: primero, de existir, la transformación de asignación; luego todas las originalmente aditivas; por último las multiplicativas.

Por lo demostrado en la sección anterior, con el segundo criterio se garantiza que es válida la equivalencia del aporte de valor que se tiene en cuenta en el primero.

Sea $G = \{g_1 \dots g_m\}, G' = \{g'_1 \dots g'_m\}$, respectivamente, el orden de aplicación del algoritmo greedy y sus transformaciones llevadas a forma multiplicativa en orden no creciente.

Sea $O = \{o_1, o_2 \dots o_m\}$ un óptimo del problema, donde los o_j se encuentran ordenados por orden de aplicación. Supongamos que O no mantiene el orden de aplicación del algoritmo greedy, entonces por lo demostrado en (2.2.1) y (2.2.2) es siempre igual o mejor un orden de aplicación como el de G, por lo tanto debe existir al menos un óptimo de esta forma, porque se puede obtener mediante transformaciones equivalentes.

Sea entonces O el óptimo con este orden cuyas transformaciones llevadas a forma multiplicativa O', ordenadas por valor no creciente, tienen el mayor prefijo común con G'. Supongamos que existe j la primera posición donde se cumple que $g'_j \neq o'_j$, de no existir tal j entonces G sería óptimo.

Conocemos que G' es un prefijo de b', y que por tanto siempre toma la transformación de mayor valor disponible. Entonces por construcción se cumple en la posición j que:

$$v(o'_j) \leq v(g'_j)$$

de lo cual existen dos posibles casos:

1. Existe una posición $k > j$ tal que $g'_j = o'_k$, por lo tanto definimos O'' el resultado de intercambiar o'_j y o'_k.

2. El elemento g'_j no se encuentra en ninguna posición de O', por lo tanto como $v(o'_j) \leq v(g'_j)$ definimos O'' el resultado de cambiar la transformación o'_j por g'_j.

Nótese que no puede ocurrir que g'_j se encuentre en una posición de O' menor que j, pues de ocurrir entonces j no sería la primera posición en que G' y O' difieren.

Entonces, como consecuencia de cualquiera de los dos casos, se tiene un óptimo O'' cuya forma ordenada por valores tiene mayor prefijo común con G' que O', lo cual es una contradicción pues por definición O' es el óptimo con mayor prefijo común con G'. Entonces no existe j tal que $g'_j \neq o'_j$, por lo que G es óptimo.

3 Implementación

La implementación del algoritmo greedy planteado se encuentra en el archivo `shop.py`.

3.1 Complejidad temporal

Sean k la cantidad de elementos a optimizar, n, m la cantidad de transformaciones disponibles y a seleccionar respectivamente, la implementación realiza las siguientes operaciones significativas:

1. Poblar una lista $f_j(t)$ de transformaciones agrupadas por cada tipo t y por elemento j-ésimo de a que modifican. Se realizan no más inserciones que la cantidad n de transformaciones.

    - Una inserción utilizando `append` en Python tiene un costo amortizado de $O(1)$; aunque la amortización no implica que no ocurra ocasionalmente una inserción mas lenta, en la mayoría de los casos se comporta de esta manera.
    - Por tanto en el peor de los casos la complejidad temporal amortizada de la poblar la lista de agrupamientos sería $O(n)$.

2. Por cada a_j, hallar, de existir, la máxima transformación de asignación disponible, y añadirla a las transformaciones de suma.

- El total de los tamaños de todas las $f_j(=)$ no superan n.
- En el peor de los casos, todas las transformaciones son de asignación, y aplicables sobre un a_j específico, de forma que la lista tiene el mayor tamaño posible. De ser así, esta lista no puede superar el tamaño n.
- El costo de hallar el máximo elemento de una lista de tamaño n es $O(n)$. De esta forma el costo en el peor caso es $O(n + k)$.

3. Similarmente a la operación anterior, por cada a_j se ordena la lista $f_j(+)$ y se convierten a multiplicativas todas las transformaciones de esta.
   - El total de los tamaños de todas las $f_j(+)$ no superan n.
   - Ordenar por tanto las listas tiene un costo total de $O(k + n \log n)$. Convertirlas a multiplicativa entonces es $O(k + n)$.
4. Por cada a_j, añadir todas las transformaciones de forma $(*)$ a una lista general b'. Naturalmente no hay mas de n transformaciones, por lo que esta operación tiene un costo amortizado de $O(k + n)$, pues se utiliza `append`.
5. Ordenar b' por un costo de $O(n \log n)$ y tomar un conjunto tamaño m de los mayores m elementos de b', en $O(m)$ utilizando la clase `set` de Python.
6. Ordenar b por a_j asociado, luego por tipo, luego por valor.
   - Todos los b_i con mismo a_j asociado pasan a ser ordenados por tipo, luego todos los b_i con mismos a_j asociado e iguales tipos pasan a ser ordenados por valor.
   - El costo de ordenar b es $O(n \log n)$.
7. Respetando el orden del b recientemente ordenado, imprimir los valores seleccionados con complejidad total de $O(n)$.

La complejidad temporal total del algoritmo es entonces:

$$O(k + m + n \log n) = O\big(\max\big\{k, m, n \log n\big\}\big)$$

4 Generación de casos

En el archivo `scripts/tester.py` se encuentra la implementación de un sistema de generación de casos pruebas para el algoritmo greedy planteado. Basta con ejecutar los comandos:

```
cd scripts
python tester.py n
```

donde n es la cantidad de pruebas aleatorias a crear y comprobar. El script imprime el contexto de cada caso ejecutado (los valores de a, b, n) e imprime `"OK."` en caso de estar correcto o por lo contrario imprime `"ERROR."` con contexto adicional de los valores obtenidos de la multiplicatoria.

4.1 Verificación utilizando fuerza bruta

Para verificar la salida del algoritmo greedy es necesario tener una forma de obtener la respuesta correcta sin utilizarlo. Con este objetivo en el script existe la función `solve_shop_with_brute_force`, la cual se encarga de comprobar todos los subconjuntos posibles de b de tamaño $1 \dots m$, almacenando el valor máximo obtenido.

Se asume solamente, de lo demostrado en secciones anteriores, que es correcta la ordenación de las transformaciones planteada, tal que siempre existe un óptimo que aplica las transformaciones en orden $(=, +, *)$; el algoritmo de fuerza bruta encuentra siempre dicho óptimo, o bien alguno de los que son de esa forma.

En cuestiones de complejidad, se itera una vez por cada subconjunto de b, y conocemos que la cantidad de subconjuntos de cualquier conjunto de tamaño n es $O(2^n)$. Por cada iteración además se comprueba el valor que se puede obtener de seleccionar estas transformaciones en orden, lo cual tiene un costo de $O(n \log n)$; por cada una de las a lo sumo m transformaciones, se aplica la misma sobre el a_i correspondiente, lo cual tiene un costo de $O(m)$; por último en cada iteración se obtiene el resultado de la multiplicatoria $\prod_{i=1}^{k} a_i$, lo cual incurre un costo $O(k)$.

El costo total del algoritmo termina siendo entonces $O((n \log n + k) \cdot 2^n)$.

4.2 Medio de verificación de las soluciones

Para verificar que las soluciones de ambos algoritmos son equivalentes, y que en efecto el algoritmo greedy devuelve una respuesta correcta, se utilizan los resultados de la multiplicatoria final.

Considerando que no necesariamente existe un solo óptimo no se tiene en cuenta que el orden de las respuestas sea distinto, o bien incluso que traten las respuestas de ambos algoritmos del mismo subconjunto. Sin embargo, si existe O el óptimo encontrado por el algoritmo de fuerza bruta, y G la solución del greedy, entonces de cumplirse que la productoria de a con las transformaciones de O obtiene el mismo resultado que con las transformaciones de G, el greedy encontró un óptimo para el caso actual.