Optimizaciones de DP

Agustín Santiago Gutiérrez

6 de octubre de 2017

1. Introducción / publicidad

Supongamos que tenemos un algoritmo de programación dinámica con dos parámetros:

$$dp(i, j), \text{ donde } i \text{ toma } N \text{ valores, y } j \text{ toma } M \text{ valores}$$

Y supongamos que la recursión tiene la forma:

$$dp(i, j) = \min_{k \in r(i,j)} f(i, j, k)$$

Donde $f(i, j, k)$ es cualquier cuenta o cómputo que dependerá de la dp, y puede usar otros valores $dp(i_2, j_2)$ de la dp (por algo es justamente una dp recursiva). Además $r(i, j)$ es un cierto rango de opciones que se iteran en busca de la mejor, para computar $dp(i, j)$.

Todo lo anterior es típico tanto de "dp en rangos" como de "dp de particionar en n intervalos", y podría aplicar a otras.

Llamaremos $k(i, j)$ al primer valor de k en el cual se alcanza el mínimo en la expresión de $dp(i, j)$. O sea, $k(i, j)$ es la elección óptima de k para el problema $dp(i, j)$. Llamaremos K a una cota para estos rangos, de forma que siempre sea $0 \leq k(i, j) < K$.

Notar que si quisiéramos tener el valor de $k(i, j)$ explícitamente, se calcula fácilmente al mismo tiempo que el valor $dp(i, j)$, anotando cuál fue el k que obtuvo el óptimo en la iteración del mínimo.

Asumiendo siempre que el cómputo de $f(i, j, k)$ tiene costo $O(1)$ (y asumiendo por supuesto que es $O(1)$ consultar los $dp(i, j)$ ya calculados), el costo total de la dp genérica anterior sería $O(NMK)$. Sino, simplemente todas las complejidades se multiplicarían por el costo de f.

En un contexto completamente general sin más hipótesis, no hay una receta milagrosa para bajar esa complejidad. Sin embargo, existen mecanismos para bajar la complejidad:

1. Si sabemos que la función $k(i, j)$ **es monótona** (ya sea creciente o decreciente) **en ambos parámetros** (por ejemplo, $k(i, j) \leq k(i + 1, j)$ y análogamente para j), y el orden de llenado de la dp lo permite, la **optimización de Knuth** reduce la complejidad a $O(N(M + K))$ u $O(M(N + K))$ (según los detalles del orden de llenado). Si $N = M = K$, pasamos de cúbico a cuadrático.

2. Si sabemos que la función $k(i, j)$ **es monótona en el parámetro** j (ya sea creciente o decreciente), y el orden de llenado de la dp lo permite, la **optimización de Divide and Conquer** reduce la complejidad a $O(N(M + K \lg M))$. Si $N = M = K$, pasamos de cúbico a $N^2 \lg N$.

Se puede observar que la optimización de Divide and Conquer no logra una mejora tan poderosa como la de Knuth, pero requiere de una propiedad más débil para su aplicación. Casualmente, la optimización de Knuth es muchísimo más simple, tanto de entender como de programar, con lo cual es la jugada ideal cuando se puede utilizar.

Como heurística de aplicación de estas optimizaciones, esta propiedad de monotonía es **muy** común en algoritmos de programación dinámica que tengan que elegir cómo "partir" una secuencia, intentando que en

total quede "lo más pareja" posible, con respecto a cierta medida. Algunos ejemplos son "árbol binario de búsqueda óptimo", "cómo asociar un producto de matrices de forma óptima", y este problema de codeforces `http://codeforces.com/contest/321/problem/E`

2. Optimización de Knuth

Las dps más comunes siempre llenan la tabla en algún orden "monótono en i" y "monótono en j". O sea, con for for.

Por ejemplo, en las dps con rangos típicas que hay que ver donde partir, $dp(i,j)$ necesita tener calculados los $dp(i,k)$ y los $dp(k,j)$ con k entre i y j. Se observa que en esta recursión, al llamar a un subproblema, el i crece o queda igual, y el j baja o queda igual. Así que el orden para recorrer sería simplemente:

```
for i := N-1 downto 0
for j := 0 to M-1
   calcular dp(i,j)
```

En casi todos los casos felices de este estilo, gracias a la asumida monotonía de $k(i,j)$ respecto de ambas variables, podremos encontrar alguna cota para $k(i,j)$ en base únicamente a valores ya calculados de la dp. Por ejemplo, $k(i,j-1) \leq k(i,j) \leq k(i+1,j)$, asumiendo que $k(i,j)$ es creciente en cada variable.

La optimización de Knuth entonces es así: Codeamos la dp tal cual, con los mismos for de toda la vida, pero en lugar de iterar el k en todo el rango completo que le correspondería normalmente, lo iteramos entre $k(i,j-1)$ y $k(i+1,j)$ únicamente[1], para ahorrar cómputos.

En código es solamente cambiar los límites del for interno que itera k aprovechando la información de estas cotas, absolutamente nada más (hay que tener un leve cuidado en los bordes, ya que con $i = N-1$ o con $j = 0$ una de estas cotas no existe. Ahí usaríamos simplemente el límite común y corriente, ya sea 0, K o lo que corresponda).

Lo increíble es que esto, que a primera impresión parece una optimización de dudosa efectividad que probablemente mejore la constante nada más, "le saca un N" a la complejidad.

La razón es que la complejidad del cálculo de $dp(i,j)$ es $k(i+1,j) - k(i,j-1) + O(1)$. Al sumar ese costo sobre todos los i,j, queda una complejidad $O(NM)$, más la suma de todos los términos $k(i+1,j)$, menos la suma de todos los términos $k(i,j-1)$. Y esto es una especie de "telescópica 2D": Casi todos los elementos $k(i,j)$ de la tabla aparecen una vez sumando y una restando, así que se cancelan. Solamente sobreviven sumando los del borde $j = M-1$. Al sumar esos al factor $O(NM)$, nos queda una complejidad $O(M(N+K))$

3. Optimización de Divide and Conquer

Esta optimización asume monotonía solamente en una de las variables. Supongamos que asumimos monotonía en j.

Para poder aplicarla, es clave que no haya dependencias dentro de una misma fila. Es decir, los valores $dp(i,j_1)$ y $dp(i,j_2)$ de una misma fila i tienen que poder calcularse en cualquier orden (o al menos, en el orden particular que vamos a usar nosotros, que no guarda una relación prolija con la dp).

Lo anterior es típico de dps del estilo "particionar esto en i pedazos", de manera que la solución para particionar en i pedazos solamente llama a $i-1$ pedazos, o sea otra fila anterior, y nunca dentro de la misma fila.

Aprovechando lo anterior, la clave entonces va a ser computar una fila completa en $O(M + K \lg M)$. De ahí la complejidad final es hacer eso N veces, para ir calculando cada fila.

[1] Más precisamente, queremos iterarlo en la intersección entre el rango normal en que lo hubiéramos iterado, y el rango dado por estas cotas. O sea que si lo hubiéramos iterado entre A y B normalmente, lo estaríamos iterando desde $\max(A, k(i,j-1))$ hasta $\min(B, k(i+1,j))$

La observación milagrosa es que, como dentro de nuestra fila i tenemos $k(i, j_1) \leq k(i, j_2)$ si $j_1 \leq j_2$, eso quiere decir que si empezamos **calculando un elemento del medio**, digamos el $c = \lfloor \frac{M}{2} \rfloor$, la fila nos queda partida en dos partes donde tenemos rangos "prácticamente independientes" para k.

Concretamente, si llamamos $k_c = k(i, c)$, entonces para todos los $0 \leq j < c$ tendremos $0 \leq k(i, j) \leq k_c$, y para los $c < j < M$, tendremos $k_c \leq k(i, j) < K$. O sea, de un lado k se mueve solo en $[0, k_c]$ y del otro se mueve en $[k_c, K)$.

La situación original era que teníamos que calcular todos los valores de la fila, o sea todos los $dp(i, j)$ con j en $[a, b)$ siendo $a = 0, b = M$. Sabíamos para esto que el k solo había que probarlo entre $l_k = 0$ y $h_k = K - 1$. Evaluamos solamente en el elemento del medio c, lo que nos costó K, y como resultado de todo esto, el problema faltante nos quedó partido en dos análogos al original:

Primero, hay que llenar todos los $dp(i, j)$ con j en $[a, b)$, para $a = 0, b = c$. Pero sabemos que en este rango, solo hay que iterar k entre $l_k = 0$ (que era nuestro l_k de partida) y $h_k = k_c$. Podemos recursivamente hacer todo eso de la misma forma (se calcula uno del medio, se divide en 2, etc).

Segundo, hay que llenar todos los $dp(i, j)$ con j en $[a, b)$, para $a = c + 1, b = M$. Pero sabemos que en este rango, solo hay que iterar k entre $l_k = k_c$ y $h_k = K - 1$ (que era nuestro h_k de partida). Podemos recursivamente hacer todo eso de la misma forma (se calcula uno del medio, se divide en 2, etc).

Esto da una recursión de Divide and Conquer que va llenando la fila, y es completamente análoga a la de por ejemplo mergesort: Si imaginamos el árbol de llamadas, cada "piso" siempre cuesta (además de un costo $O(1)$ por nodo del árbol de llamadas) un total de $O(K)$: Ya que el rango original $[0, K)$ se va partiendo en intervalitos que lo componen, pero que en cada piso siempre tienen una longitud total $O(K)$.

El árbol de llamadas tiene un total de $O(M)$ nodos repartidos en $O(\lg M)$ pisos "a lo segment tree", así que la complejidad total del divide and conquer resulta $O(M + K \lg M)$, como prometimos.

Una especie de pseudocódigo (la llamada inicial sería `llenarPedazoDeFila(0,M,0,K-1)`):

```
llenarPedazoDeFila(a,b,lk,hk)
  Si a = b, no hacer nada, rango vacio
  c := (a+b)/2
  calcular dp(i,c) y al mismo tiempo k(i,c)
       pero iterando k solamente en [lk, hk]
  llenarPedazoDeFila(a,c,lk,k(i,c))
  llenarPedazoDeFila(c+1,b,k(i,c), hk)
```