Algorítmica

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Algorítmica

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

Índice general

1. Programación Dinámica

- Para problemas en los que necesitamos estados anteriores (en fibonacci, para calcular fibonacci(n) necesitamos tener fibonacci($n-1$) y fibonacci($n-2$), y para fibonacci($n-1$) necesitamos, ...).
- En el camino aparecen requisitos que se repiten (necesitamos calcular varias veces fibonacci(k)). En vez de calcularlo todas las veces, calcularlo una sola vez. Para calcular fibonacci(6) necesitamos 5 veces fibonacci(2).
- Antes de calcular el subproblema, mira si lo tienes ya resuelto (si lo tiene, lo usa y si no lo tiene, lo calcula).
- Es necesaria una estructura para almacenar las soluciones a los subproblemas, con la finalidad de ahorrar llamadas recursivas.

Ejemplos de dónde usar programación dinámica son:

- Fibonacci.
- Calcular números combinatorios.
- Calcular potencias naturales.
- Cualquier problema con solapamiento de subproblemas (encontramos subproblemas que se repiten).

Una cosa es la programación dinámica y otra es la memorización:

Memorización. Almacenamos en una estructura (como un diccionario) los resultados.

Programación dinámica. Una vez que los hemos almacenado, buscamos un patrón para ver cómo se completan las soluciones de alguna forma más eficiente (quitando sobrecarga por la recursividad). Buscamos una forma de rellenar la estructura de datos.

Cuando aplicar programación dinámica

Normalmente para problemas de optimización (minimizar o maximizar). La solución al problema la tenemos que ver como un proceso de selección de varias etapas.

- Se aplica a problemas que pueden suponer un alto coste computacional que dispone de subestructuras optimales que se solapan (se repiten a lo largo del cálculo de la solución).

La eficiencia del algoritmo suele ser polinomial. Normalmente suele ser $O(n \cdot m)$ donde n es el tamaño de la estructura de datos y m el tiempo para cada casilla.

Comparación con Divide y Vencerás

Divide y vencerás.

- Se aplica a subproblemas independientes.
- La técnica suele ser descendente.

Programación dinámica.

- Se aplica a subproblemas que se solapan (que se resuelven más de una vez).
- La técnica suele ser ascendente.
- Asegura optimilidad pero puede llevar a un algoritmo que no sea eficiente.

Teorema 1.1 (Principio de Optimalidad de Bellman)**.** *Una solución óptima está compuesta de subsoluciones óptimas.*

Cuando se cumpla el principio, se podrá utilizar la programción dinámica.

Ejemplo. Un ejemplo que no cumple el Principio de Optimalidad de Bellman es el problema del camino más largo entre dos nodos en un grafo.

Por tanto, no podemos aplicar programación dinámica para resolverlo.

1.1. Pasos para desarrollar un algoritmo

1. Plantear la solución a un problema como una secuencia de decisiones.
2. Ver que se verifica el principio de optimalidad **??**.
3. Plantear la solución como una función recursiva y ver la tipología de los subprobelmas.
4. Ver cómo un problema grande se puede calcular a partir de los problemas más pequeños.
5. Tratar de buscar un enfoque ascendente (resolver problemas pequeños y resolver problemas mayores).

Ante un problema del estilo buscar un camino óptimo, es capaz de decir el costo del camino pero no de decir el camino. Para ello:

- O se puede deducir el camino a partir del costo.
- O apuntar en una tabla auxiliar las decisiones tomadas.

1.2. Ejemplos de Programación Dinámica

Ejemplo. Problema de devolver cambio: ¿cómo devolver el cambio con el menor número de monedas?

Podemos considerarlo como un problema de resolución multietápica. Cumple el Principio de Optimalidad de Bellman, luego podemos aplicar programación dinámica para resolver el problema.

Ejemplo. Producto encadenado de matrices. Su solución es similar al árbol binario de búsqueda óptima (por ejemplo, en un árbol de palabras, poner las más frecuentes arriba y las menos abajo).

- Ante una solución que es multiplicar al tun tún, es mejor calcular primero el número de operaciones a realizar y quedarse con la multiplicación que dé el menor.

Se cumple el Principio de Optimalidad de Bellman, luego puede hacerse con programación dinámica.

Buscamos hacer la definición recursiva de la solución. Para ello vemos dónde colocar el primer paréntesis:

$$N[i,j] = 0 \qquad \text{si } i = j$$
$$N[i,j] = \min_{k \in [i,j]} \{N[i,k] + N[k+1,j] + p_{i-1}p_k p_j\} \text{ si } i < j$$

Siendo $N[i,j]$ el costo de multiplicar desde la matriz i-ésima a la j-ésima.

- Tenemos $O(n^2)$ subproblemas.
- Los subproblemas triviales son aquellos en los que $i = j$.

Representamos los $N[i,j]$ en una tabla. Formas de rellenar la tabla:

- Por diagonales.
- De abajo a arriba y de izquierda a derecha.

Una vez rellenada la tabla triangular superior, tenemos el número mínimo de operaciones para calcular el producto.

Tenemos un algoritmo $O(n^3)$. Sabemos el costo mínimo pero no tenemos cuál es la parantización que nos lo da. Para ello, o volvemos sobre nuestros pasos o almacenamos el menor en cada momento (el k de cada paso).

Ejemplo. Subsecuencia común de mayor longitud (LCS): Dadas dos secuencias de símbolos X e Y, buscamos la mayor subsecuencia común a ambas de mayor longitud (pudiendo estar tanto en X como en Y caraceteres intermedios):

$$X = A\ B\ C\ B\ D\ A\ B$$
$$Y = B\ D\ C\ A\ B\ A$$

La mayor subsecuencia común es $B\ D\ A\ B$, aunque también puede ser $B\ C\ B\ A$. Tiene aplicaciones en genética, diferencias de archivos y detección de plagio.

- Un algoritmo de fuerza bruta compararía cualquier subsecuencia de X con símbolos de Y.

  Si $|X| = m$ y $|Y| = n$, hay que contrastar 2^m subsecuencias de X contra los n elementos de Y, o al revés. Siendo:

  $$k = \text{mín}\{n, m\} \qquad t = \text{máx}\{n, m\}$$

  tenemos un orden de $O(t \cdot 2^k)$, en el mejor caso.

Cumple el Principio de Optimalidad de Bellman. Buscamos una secuencia de decisiones a desarrollar: Si meto o no un carácter.

Definimos:

$$X_i = (x_1, \ldots, x_i) \qquad Y_j = (y_1, \ldots, y_j)$$

Considerando que $X = (x_1, \ldots, x_m)$ con $m \geqslant i$ y $Y = (y1, \ldots, y_n)$ con $n \geqslant j$.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{si } x[i] = y[j] \\ \text{máx}\{c[i, j-1], c[i-1, j]\} & \text{si } x[i] \neq y[j] \end{cases}$$

siendo $c[i, j]$ la longitud de la solución óptima al problema de X_i y Y_j.

Por tanto, la solución es $c[n, m]$. Rellenaríamos la matriz por filas o por columnas (ver qué necesitamos para calcular $c[i, j]$, en cada caso). Algoritmo de orden $O(n \cdot m)$.

Para encontrar la solución miramos la solución recursiva que hicimos:

- Si son iguales la columna y la fila, disminuimos en 1 la fila y la columna.
- Si no son iguales, vuelvo al máximo.

Ejemplo. Dado un grafo y dos nodos, buscar el camino mínimo entre ambos.

Secuencia de decisiones a tomar: si consideramos un vértice o no para ser vértice intermedio del camino.

Notación. Notaremos por $D[i, j]$ al camino mínimo entre i y j.

Además, notaremos por $D_k[i, j]$ al camino mínimo entre i y j en un problema de los primeros k nodos del grafo.

De esta forma, $D_0[i, j] < \infty \iff$ hay un camino directo entre i y j.

Definición recursiva:

- Si el camino de i a j por k vértices no pasa por k:

  $$D_k[i, j] = D_{k-1}[i, j]$$

- Si pasar por k:

  $$D_k[i, j] = D_{k-1}[i, k] + D_{k-1}[k, j]$$

De donde:

$$D_k[i,j] = \text{mín}\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$$

Tenemos varias matrices, desde 0 hasta $n-1$ (siendo n el tamaño del grafo). Notemos que a la hora de construir la matriz D_k, no alteramos ni la fila k ni la columna k.

Obtenemos el **algoritmo de Floyd**, de eficiencia $O(n^3)$.

Ejemplo. Distancia de edición. Dadas dos cadenas, buscar la transformación de menor coste para llegar de una palabra a otra.

Podemos reemplazar, insertar y borrar caracteres.

Notación. Notamos por $d(i,j)$ a la distancia de edición de transformar la cadena desde el carácter 1 hasta el i hasta la que tenemos por transformar desde el 1 hasta el j.

Definición recursiva:

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{si} \quad s[i] = t[j] \\ 1 + \text{mín}\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & \text{si} \quad s[i] \neq t[j] \end{cases}$$

Por ejemplo:

$$d(\text{casa}, \text{costa}) = d(\text{cas}, \text{cost})$$

$$d(\text{casas}, \text{cosa}) = \text{mín} \left\{ \begin{array}{l} 1 + d(\text{casa}, \text{cosa}), \\ 1 + d(\text{casas}, \text{cos}), \\ 1 + d(\text{casa}, \text{cos}) \end{array} \right\}$$

Ejemplo. Problema de caminos mínimos con peso negativo. Estudiamos el algoritmos de Bellman-Ford.

Notación. Notamos por $D_i(v)$ al camino de costo mínimo entre el nodo v (origen) y el t (destino) usando como mucho i arcos.

2. Relaciones de Problemas