Programación dinámica

Melanie Sclar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

AED III

Soluciones recursivas a problemas

- Muchos algoritmos de utilidad son recursivos: para resolver un problema, se utilizan las soluciones a subproblemas fuertemente relacionados.
- En estos algoritmos, se divide el problema en varios subproblemas que luego se resuelven y se combinan las soluciones obtenidas para resolver el original.
- Un ejemplo de técnica recursiva de diseño de algoritmos es la técnica de Divide and Conquer, vista en algoritmos 2.

¿En qué consiste la programación dinámica?

- La programación dinámica es una técnica de solución de problemas recursiva.
- Al igual que Divide and Conquer, la técnica propone descomponer el problema a resolver en **subproblemas más pequeños de la misma especie**, para resolverlos recursivamente y combinar esas soluciones en una solución al problema original.
- La diferencia esencial que lo contrasta con Divide and Conquer, es que mientras que en esta técnica los subproblemas que se resuelven son independientes entre sí y se resuelven individualmente, la programación dinámica es aplicable cuando los subproblemas **no son independientes**.
- En estos casos, un algoritmo de Divide and Conquer realizaría el mismo trabajo múltiples veces, ya que la solución a un mismo subproblema puede ser **recalculada** muchas veces si se la reutiliza como parte de varios subproblemas más grandes.

¿En qué consiste la programación dinámica? (2)

- La solución que propone la técnica de programación dinámica es **almacenar** las soluciones a subproblemas ya calculadas, de manera de calcularlas una sola vez, y luego leer el valor ya calculado cada vez que se lo vuelve a necesitar.
- Uno de los usos más importantes de esta técnica es en problemas de **optimización**: En estos problemas interesa encontrar la solución que maximiza un cierto puntaje u objetivo, en un espacio de soluciones posibles.
- Un indicador central de la aplicabilidad de la técnica lo constituye el **principio del óptimo.** Este principio afirma que **las partes de una solución óptima** a un problema, deben ser **soluciones óptimas de los correspondientes subproblemas**, y es lo que permite obtener una solución óptima al problema original a partir de soluciones óptimas de los subproblemas.

El esquema general

Los algoritmos de programación dinámica se pueden organizar típicamente en 4 pasos que responden al siguiente esquema general:

1. Caracterizar la estructura de una solución óptima.
2. Definir recursivamente el valor de una solución óptima.
3. Computar el **valor** de una solución óptima. Se calcula de manera *bottom-up*.
4. Construir una solución óptima a partir de la información obtenida en el paso 3

El paso 4 es optativo, ya que si solo nos interesa el valor o puntaje de una solución óptima pero no la solución en sí, este paso de reconstrucción no es necesario.

Problema del recorrido óptimo en una matriz

Sea $M \in \mathbb{N}^{m \times n}$ una matriz de números naturales. Se desea obtener un camino que empiece en la casilla superior izquierda $(1, 1)$, termine en la casilla inferior derecha (m, n) y tal que minimice la suma de los valores de las casillas por las que pasa. En cada casilla (i, j) hay dos movimientos posibles: ir hacia abajo (a la casilla $(i + 1, j)$), o ir hacia la derecha (a la casilla $(i, j + 1)$).

- a Diseñar un algoritmo eficiente basado en programación dinámica que resuelva este problema.
- b Determinar la complejidad del algoritmo propuesto (temporal y espacial).
- c Exhibir el comportamiento del algoritmo sobre la matriz que aparece a continuación.

$$\begin{bmatrix} 2 & 8 & 3 & 4 \\ 5 & 3 & 4 & 5 \\ 1 & 2 & 2 & 1 \\ 3 & 4 & 6 & 5 \end{bmatrix}$$

Fórmula recursiva

La matriz de resultados parciales almacena en $best(i,j)$ la mínima longitud de un camino que empiece en $(1,1)$ y llegue a (i,j), haciendo solo movimientos hacia abajo y hacia la derecha.

- $best(i,j) = M_{i,j} + min(best(i-1,j), best(i,j-1))$
  para $1 < i \leq m$ y $1 < j \leq n$
- $best(i,1) = M_{i,1} + best(i-1,1)$ para $1 < i \leq m$
- $best(1,j) = M_{1,j} + best(1,j-1)$ para $1 < j \leq n$
- $best(1,1) = M_{1,1}$

La longitud del mínimo camino entre esquinas, que constituye la solución al problema, viene dada por $best(m, n)$. Con esto ya podríamos implementar una solución top-down recursiva:

- Si para calcular un $best(i, j)$ necesitamos un resultado ya calculado, lo usamos directamente.
- Sino, lo calculamos recursivamente, almacenamos su valor en la tabla de resultados y luego lo utilizamos.

Algoritmo top-down

```
best(Matriz, i, j):
    if (calculado[i][j] != −1)
        return calculado[i][j]

    if (i = 1 and j = 1)
        calculado[i][j] <− Matriz[1][1]
    else if (i = 1)
        calculado[i][j] <− best(i, j−1) + Matriz[i][j]
    else if (j = 1)
        calculado[i][j] <− best(i−1, j) + Matriz[i][j]
    else
        calculado[i][j] <− min(best(i−1, j), best(i, j−1)) + Matriz[i][j]

    return calculado[i][j]
```

La complejidad del algoritmo resultante es $O(nm)$, tanto espacial como temporal. Puede servir especialmente en problemas donde no se necesitará calcular buena parte de todos los estados para obtener el resultado buscado.

Algoritmo bottom-up

```
longitudCaminoMinimo(Matriz, m, n):
    best[1,1] = Matriz[1,1]
    for i = 2 to m do
        best[i,1] = Matriz[i,1] + best[i−1,1]
    for j = 2 to n do
        best[1,j] = Matriz[1,j] + best[1,j−1]
    for i = 2 to m do
        for j = 2 to n do
            best[i,j] = Matriz[i,j] + min(best[i−1,j], best[i,j−1])
    return best[m,n]
```

La complejidad del algoritmo resultante es $O(nm)$, tanto espacial como temporal. Se puede bajar la complejidad espacial a $O(min(n, m))$ si no interesa reconstruir el camino sino solo su longitud.

Cálculo en el ejemplo

Matriz de entrada:

$$\begin{bmatrix} \mathbf{2} & 8 & 3 & 4 \\ \mathbf{5} & 3 & 4 & 5 \\ \mathbf{1} & \mathbf{2} & \mathbf{2} & \mathbf{1} \\ 3 & 4 & 6 & \mathbf{5} \end{bmatrix}$$

Matriz de best:

$$\begin{bmatrix} \mathbf{2} & 10 & 13 & 17 \\ \mathbf{7} & 10 & 14 & 19 \\ \mathbf{8} & \mathbf{10} & \mathbf{12} & \mathbf{13} \\ 11 & 14 & 18 & \mathbf{18} \end{bmatrix}$$

En ambas matrices, se indica un camino óptimo en negrita.

Para implementar y seguir pensando

- `https://projecteuler.net/problem=81` para implementar el problema que acabamos de ver
- `https://projecteuler.net/problem=67`, es muy parecido al que ya vimos
- `http://www.oma.org.ar/enunciados/omn20reg.htm`, problema 3 nivel 1

De los primeros dos problemas adjuntamos un código que los implementa en sus versiones bottom-up y top-down (pero no los miren antes de intentarlo ustedes!)