Programación Dinámica

Redigonda Maximiliano
(+ Rom)

Índice

Problemas Introductorios

Maximiliano Redigonda (+ detalles por Román Castellarin)

Dominó

Contar la cantidad de formas de cubrir tableros de 2 x N con piezas de dominó (no deben quedar casillas descubiertas, ni piezas solapadas, y ninguna parte de una ficha puede quedar afuera del tablero).

Casos chicos:

N = 0 | 1 forma
N = 1 | 1 forma
N = 2 | 2 formas
N = 3 | 3 formas
N = 4 | 5 formas

Dominó

Llamemos D(n) a la cantidad de formas de llenar el tablero de 2 x n con fichas de dominó.

Si en un tablero de 2 x n ubicamos una pieza vertical al final, el problema se reduce a llenar un tablero de 2 x (n - 1).

Si en un tablero de 2 x n ubicamos dos piezas horizontales al final, el problema se reduce a llenar un tablero de 2 x (n - 2).

No hay otra forma de llenar las últimas casillas del tablero.

Recursión

Notamos entonces que D(n) = D(n-1) + D(n-2).

Los casos chicos D(0) = 1 y D(1) = 1 resultan fundamentales ahora.

Una función es **recursiva** cuando se llama a sí misma (con otros parámetros).

Las funciones recursivas son buenas candidatas para aplicarles DP (programación dinámica).

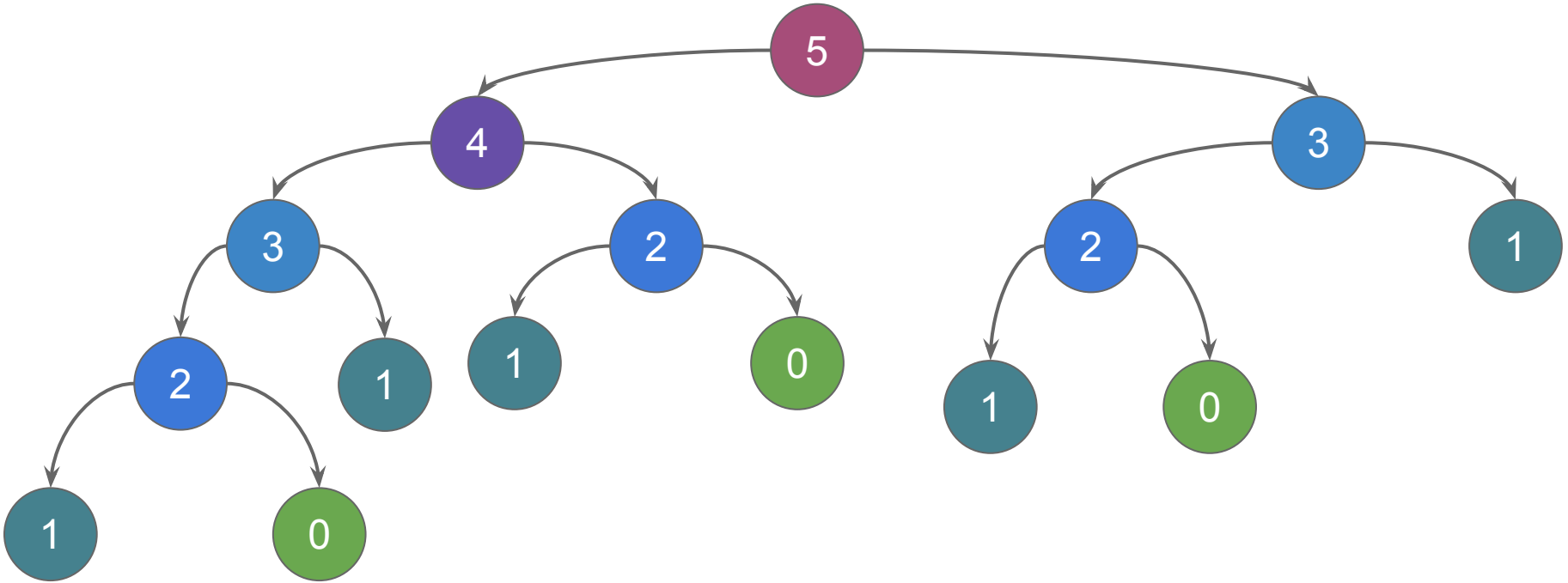
Implementación

Podemos calcular D con la siguiente función recursiva:

```
int D(int n){
  if (n <= 1) return 1;
  return D(n - 1) + D(n - 2);
}
```

Esta implementación tiene un problema importante, veamos el árbol de recursión si llamamos a D(5).

Árbol de llamadas



Pérdida de memoria

En la llamada a D(5), hacemos 5 llamadas a D(1) y 3 llamadas a D(0).

No parece tan malo, pero en la llamada a D(30) tenemos más de un millón de llamadas a D(0) o D(1) (30 es un número bastante chico).

Este problema ocurre porque la recursión **no recuerda** haber resuelto problemas que ya resolvió.

Dominó

Para calcular D(5) nuestra función calcula:

D(4), 1 vez
D(3), 2 veces
D(2), 3 veces
D(1), 5 veces
D(0), 3 veces

Mientras que los humanos sólo necesitamos calcular cada una de esas una vez, y reutilizar ese resultado.

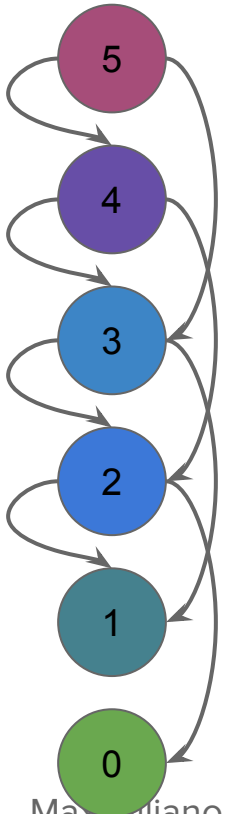
Memoización

Podemos hacer que nuestra computadora haga lo mismo que los humanos, sólo debemos guardar las respuestas de las llamadas a D en una tabla.

```
int dp[105]; // iniciado con -1s

int D(int n){
  if (n <= 1) return 1;
  if (dp[n] != -1) return dp[n];
  return dp[n] = D(n - 1) + D(n - 2);
}
```

Árbol de llamadas



Dominó

Este “parche” a nuestra función recursiva reduce su complejidad de $O(2^n)$ a O(n).

No siempre podremos hacer esto, algunos requerimientos son:

- A parámetros iguales, la función debe retornar el mismo valor *(la función debe ser **pura**, no depender de una variable global).*
- El dominio de nuestra función debe ser suficientemente chico para entrar en memoria. (Cantidad de estados)

Dominó

Pero Maxi, ¿no era más fácil hacer un for y listo?

```
int D[105];
D[0] = D[1] = 1;
for (int n = 2; n < 105; ++n) {
  D[n] = D[n - 1] + D[n - 2];
}
```

En efecto, estos son dos enfoques a la programación dinámica, el primero es conocido como “**TopDown**” y el segundo como “**BottomUp**”.

Dominó

En el caso de este problema, el enfoque BottomUp era el más sencillo (y discutiblemente, el más eficiente).

Sin embargo, cada enfoque tiene sus ventajas y desventajas.

Veremos más ejemplos de programación dinámica y discutiremos las diferencias entre BottomUp y TopDown más adelante.

Técnicas superiores

Algunos problemas de programación dinámica admiten optimizaciones de orden superior.

Por ejemplo, este problema se puede resolver mediante exponenciación logarítmica de matrices, en O(log n):

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

Técnicas superiores

Explicación:

Una potencia A^B puede resolverse en O(log B) porque:

$$A^B = \begin{cases} (A^{\lfloor B/2 \rfloor})^2 & : \text{si B es par} \\ A * (A^{\lfloor B/2 \rfloor})^2 & : \text{si B es impar} \end{cases}$$

Canguro

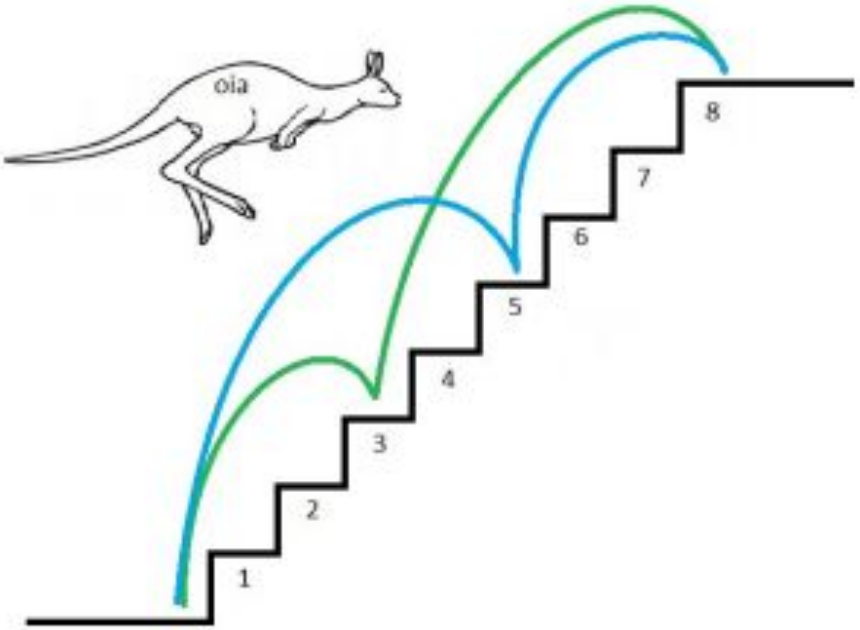
Un canguro está subiendo una escalera, nos dan información sobre los tipos de saltos que puede realizar, y el escalón al que quiere llegar.

La entrada podría ser:

```
11
4 1 3 4 5
```

Significando que quiere llegar al escalón 11, y cuenta con 4 tipos de saltos distintos, con uno avanza 1 escalón, con otro 3, con otro 4 y con otro 5. El canguro comienza en el escalón 0.

Canguro



Maximiliano Redigonda (+ detalles por Román Castellarin)

Canguro

El análisis es muy parecido al problema anterior, si C[n] es la cantidad de formas de llegar al escalón n, tenemos que C[0] = 1, y que:

$C[n] = C[n-1] + C[n-3] + C[n-4] + C[n-5]$

Necesitaríamos 5 casos base, pero podemos considerar que si n es negativo, C[n] = 0.

El código esta vez depende de la entrada al problema.

Canguro - Bottom Up

```
dp[0] = 1;
for (int n = 1; n < 105; ++n) {
  for (int s : saltos) {
    if (n - s >= 0) dp[n] += dp[n - s];
  }
}
```

(En muchos casos, como el número de formas puede crecer muchísimo muy rápido, se nos pide el resto del resultado en la división por algún número primo grande [generalmente 10^9+7])

Canguro - Top Down

```
int dp[105];
int canguro(int n){
  if (n < 0) return 0;
  if (n == 0) return 1;
  if (dp[n] != -1) return dp[n];
  dp[n] = 0;
  for (int s : saltos) dp[n] += canguro(n - s);
  return dp[n];
}
```

Suma Máxima

Nos dan una matriz de N x N con valores positivos o negativos, queremos encontrar el camino desde (0, 0) hasta (N-1, N-1) con mayor suma.

Solo podemos ir hacia la derecha o hacia abajo en un movimiento.

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

Suma Máxima

Definimos `dp[i][j]` como la mayor suma que podemos obtener yendo desde la casilla (i, j) hasta la casilla (N-1, N-1).

Vemos que `dp[N-1][N-1] = M[N-1][N-1]`, esa casilla es inevitable.

En caso de una casilla interior tendremos que sumar el costo asociado a esa casilla, y luego podremos ir hacia la derecha o abajo, elegiremos aquella opción que nos dé la mayor suma:

```
dp[i][j] = M[i][j] + max(dp[i+1][j], dp[i][j+1]);
```

Suma Máxima

Si estamos en un borde corremos el riesgo de salirnos del tablero, podemos evitar esos inconvenientes con condiciones o agregando una fila y columna adicional llena de infinitos negativos (en este caso).

Para poder aplicar programación dinámica, es necesario que el problema tenga **subestructura óptima**, es decir, se tiene que poder alcanzar la solución del problema, utilizando soluciones de subproblemas.

Suma Máxima - Top Down

```
int maxSum(int i, int j){
  if (i > N-1  or  j > N-1) return -INF;
  if (i == N-1 and j == N-1) return M[i][j];
  if (dp[i][j] != sentinel) return dp[i][j];
  dp[i][j] = M[i][j] + max(maxSum(i+1, j), maxSum(i, j+1));
  return dp[i][j];
}
```

Debemos usar un “centinela”, un valor que nunca sea respuesta de nuestro problema naturalmente, para distinguir cuando un problema fue calculado.

Suma Máxima - Bottom Up

```
for (int i = N-1; i >= 0; --i) {
  for (int j = N-1; j >= 0; --j) {
    dp[i][j] = M[i][j] + max(i+1 <= N-1 ? dp[i+1][j] : -INF,
                             j+1 <= N-1 ? dp[i][j+1] : -INF);
  }
}
```

Notemos que tenemos que recorrer los parámetros en un orden particular para que funcione correctamente.

Complejidad y DP Mochila

Maximiliano Redigonda (+ detalles por Román Castellarin)

Complejidad de una DP

Un **estado** es una tupla con los valores de los parámetros, es decir, una instancia particular de un subproblema. Por ejemplo, en el problema de los dominós, un estado podría ser (N = 3), en el problema de máxima suma, un estado podría ser (i = 6, j = 2), etc.

La complejidad de un algoritmo de DP se calcula como la suma de los costos de procesar cada estado.

En el problema de máxima suma, la complejidad es de $O(n^2) \times O(1) = O(n^2)$, y en el problema del canguro la complejidad es de $O(n) \times O(k) = O(nk)$ con k siendo la cantidad de saltos posibles del canguro.

Más definiciones

Si vemos a los estados de nuestra programación dinámica como nodos de un grafo, y a las relaciones entre ellos como aristas (llamadas **transiciones**), obtenemos un grafo dirigido acíclico (conocidos como **DAGs**).

Todo algoritmo de programación dinámica puede verse como programación dinámica en el DAG implícito.

El orden en que debemos recorrer los estados (nodos) para calcularlos está dado por su **orden topológico**.

Como atacar cualquier problema de DP

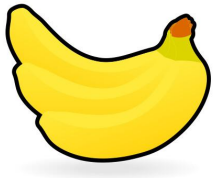
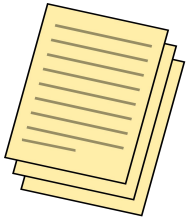
1. Descubrir que es de DP (contar esto, maximizar lo otro, 10^9+7, etc.)
2. Encontrar los estados (generalmente, 90% del trabajo)
3. Encontrar las transiciones
4. Programar

La clave está en encontrar la forma de representar el estado de manera que el problema sea correcto y tenga una complejidad aceptable.

Mochila

Queremos robar algunos objetos de una tienda, hay n objetos, cada uno tiene un valor y un peso, y nuestra bolsa puede cargar un máximo de K kilos, qué objetos robamos para maximizar el valor de la bolsa?



```
peso  = {4,        12,       8,        2};
valor = {14,      800,      16,       5};
```

Mochila

El estado debe representar la cantidad de objetos ya considerados y aquel que vamos a considerar ahora, y también debe representar el peso restante que podemos usar de la mochila.

Hay varias formas de representar la misma información, por ejemplo:

`dp[n][w]` = mayor valor alcanzable luego de evaluar los objetos con índice en [0..n), y con una capacidad restante de w kilos. (`dp[N][K]` es respuesta)

`dp[n][w]` = mayor valor alcanzable luego de evaluar los objetos con índice en [0..n), habiendo utilizado una capacidad de a lo sumo w kilos. (`dp[N][0]` es respuesta)

Mochila

No importa cual decidamos usar, siempre y cuando sea correcta y tenga una complejidad aceptable.

Una vez que tenemos los estados definidos, las transiciones son usualmente la parte más fácil.

En este caso, por cada objeto tenemos dos opciones, lo podemos ignorar y seguir adelante con otros objetos, o (si tenemos capacidad), podemos meterlo en nuestra bolsa, y seguir adelante con la capacidad actualizada.

Mochila - Bottom Up

```
for (int w = 0; w <= K; ++w)
    dp[0][w] = 0;

forn (n, N)
  for (int w = W; w >= peso[n]; --w)
    dp[n+1][k] = max(dp[n][k],
                     precio[n] + dp[n][k - peso[n]]);
```

```
forn(i, N)
    for(int w = W; w >= peso[i]; --w)
        DP[w] = max(DP[w], DP[w-peso[i]] + valor[i]);
```

Mochila - Top Down

Para hacer la versión Top Down, podemos pensar que `dp[n][k]` sea el mayor valor alcanzable luego de considerar todos los primeros n objetos, con una capacidad restante de k.

Aprovechamos que hay muchas formas de definir el significado de nuestra tabla para utilizar aquella que haga el código más sencillo.

Mochila - Top Down

```
int mochila(int n, int k){
  if (n == 0) return 0;
  if (dp[n][k] != -1) return dp[n][k];
  dp[n][k] = mochila(n - 1, k); // ignorar
  if (k >= peso[n])
    dp[n][k] = max(dp[n][k],
                   precio[n] + mochila(n - 1, k - peso[n]);
  return dp[n][k];
}
```

Mochila

El problema que estuvimos viendo hasta ahora es la variante 0-1 de la mochila. Existe la variante no-acotada en la cual tenemos infinitas copias de cada objeto.

Notemos que ambos problemas son interesantes sólo cuando los objetos no pueden ser fraccionarios, ya que de lo contrario un greedy alcanza.

```
forn(i, N)
    for(int w = peso[i]; w <= W; ++w)
        DP[w] = max(DP[w], DP[w-peso[i]] + valor[i]);
```

Top Down vs Bottom Up

Top Down vs Bottom Up

Para la gran mayoría de problemas, hacer una solución top down o una bottom up es una cuestión de gustos, sin embargo, hay casos claros en los que uno de los estilos es el preferido.

Por ejemplo, una solución top down no requiere visitar todos los estados, lo cual resulta necesario en algunos problemas.

Por otro lado, las soluciones bottom up son generalmente “cache-friendly” y suelen ser la solución por defecto de los competidores más experimentados.

Top Down (pros y contras)

Pros:

- Transformación natural desde la definición recursiva.
- No requiere pensar el orden de procesamiento de estados.
- Computa los subproblemas sólo cuando es necesario.

Contras:

- Ligeramente más lenta por el “overhead” de las llamadas a funciones.
- Posibles MLE en caso de que la **profundidad** de la recursión sea grande.

Bottom Up (pros y contras)

Pros:

- Usualmente resulta cache-friendly, lo cual incrementa su velocidad.
- En algunas circunstancias, se puede ahorrar gran cantidad de memoria (reduciendo los estados en una dimensión) (este truco es raramente necesario).

Contras:

- Para programadores más orientados a la recursión, este estilo puede ser anti-intuitivo.
- Requiere visitar todos los estados.
- Debemos pensar en el orden en cual recorrer los estados.

Gracias!