



Programación Dinámica

Técnicas y Diseños de Algoritmos - 2024 1c

Alfredo Umfurer



Recursión

¿Para qué sirve? ¿Por qué está buena?

La recursividad es una técnica que nos permite resolver el problema al reducirlo a operaciones más sencillas que emplean la solución de versiones más pequeñas del mismo.

- “Si es caso chiquito es fácil resolverlo” (Caso base)
- “Si tuviera resueltas estas partes, podría usarlas para resolver mi caso actual” (Caso recursivo)



Recursión

En backtracking usamos la recursión como un mecanismo para generar todo el espacio de soluciones.

En muchos casos podremos saber si las soluciones que generamos no serán factibles u óptimas y evitando resolver el problema para algunas de las instancias recursivas (podas de factibilidad ú optimalidad)



Programación Dinámica

Idea:

Si llamo a una función **más de una vez** con los mismo parámetros, me conviene computarla una sola vez y guardarme el resultado





Solución Recursiva

- + Memorización**
- + Superposición de Problemas**

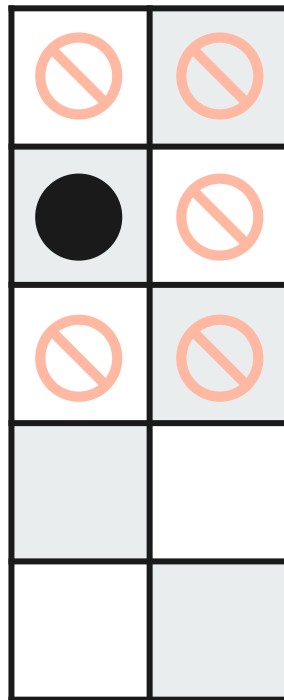
Programación Dinámica




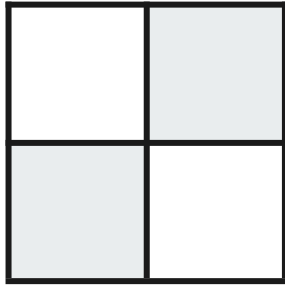
Poniendo las fichas en el tablero

Tenemos un tablero de $2 \times n$ y queremos poner fichitas de manera que no hayan dos que sean vecinas (es decir que no compartan lados ni esquinas).

¿De cuántas maneras podemos hacerlo?




$$n = 2$$




$$n = 1$$

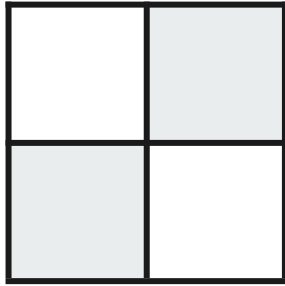


$$n = 0?$$



“Si es caso chiquito es fácil resolverlo” (Caso base)


$$n = 2$$



$$n = 1$$



$$n = 0?$$



“Si es caso chiquito es fácil resolverlo” (Caso base)

$$f(0) = 1$$

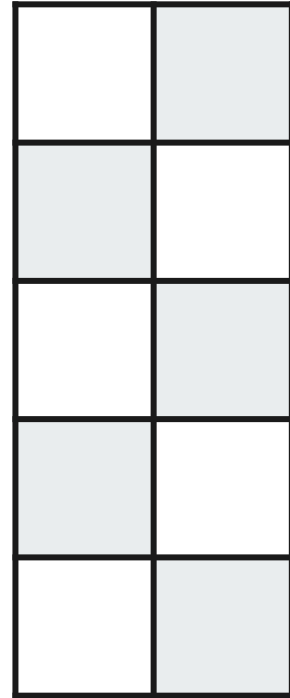
$$f(1) = 3$$

$$f(2) = 5$$



Poniendo las fichas en el tablero

“Si tuviera resueltas estas partes, podría usarlas para resolver mi caso actual”

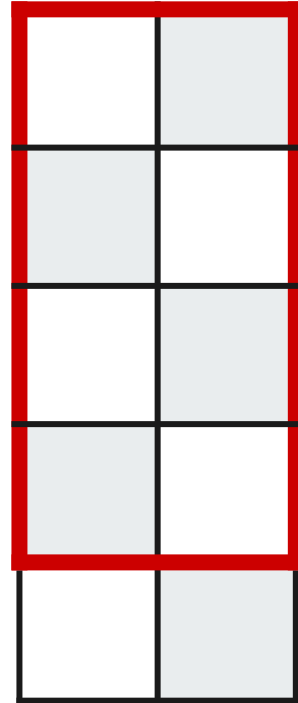


Poniendo las fichas en el tablero

Si no pongo fichas en la última fila, ¿De cuántas maneras puedo rellenar el resto del tablero?

$$f(n - 1)$$

Y si pongo una?



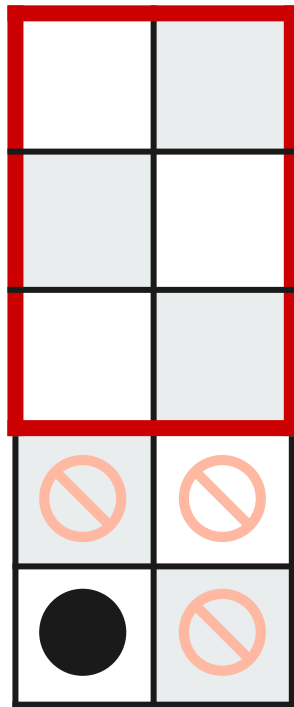
Poniendo las fichas en el tablero

Si no pongo fichas en la última fila, ¿De cuántas maneras puedo rellenar el resto del tablero?

$$f(n - 1)$$

Y si pongo una?

$$f(n - 2)$$





Definimos la función

$f(n)$: Cantidad de formas de poner fichas en un tablero de $2 \times n$ sin que haya vecinos

$$f(n) = \begin{cases} 1 & n = 0 \\ 3 & n = 1 \\ f(n-1) + 2f(n-2) & n > 1 \end{cases}$$



Lo codeamos?



Superposición de problemas

Decimos que tenemos superposición de problemas si la cantidad de llamadas recursivas de la función sin memorizar es *mucho mayor* que la cantidad de estados posibles.

$$\Omega(\#llamadasRekursiva) \gg O(\#estados)$$



Superposición de problemas

Cantidad de llamadas recursivas:

$$R(n) = R(n-1) + R(n-2) + O(1)$$

$$> 2R(n-2)$$

$$R(n) > 2^{n/2}$$

$$R(n) \subseteq \Omega(\sqrt{2}^n)$$

Cantidad de estados posibles: $E(n) \subseteq O(n)$

Superposición: $\Omega(\sqrt{2}^n) \gg O(n)$ ✓



Complejidad de algoritmo:

Para calcular la complejidad del algoritmo, podemos emplear la misma estrategia que en backtracking: Sumar las complejidades de cada uno de los estados por los que pasamos (ignorando el costo de las llamadas recursivas)

Pasamos por n estados posibles (porque con la memorización evitamos repetirlos) y en cada estado hacemos $O(1)$ operaciones

Además tenemos que considerar el costo de inicializar la memoria: $O(n)$

$$n \times O(1) + O(n) = O(n)$$



Complejidad de algoritmo:

Para calcular la complejidad del algoritmo, podemos emplear la misma estrategia que en backtracking: Sumar las complejidades de cada uno de los estados por los que pasamos (ignorando el costo de las llamadas recursivas)

Pasamos por n estados posibles (porque con la memorización evitamos repetirlos) y en cada estado hacemos $O(1)$ operaciones

Además tenemos que considerar el costo de inicializar la memoria: $O(n)$

$$n \times O(1) + O(n) = O(n)$$

*Cuando calculamos la complejidad, solemos expresarla en función del *tamaño de la entrada*, para este caso como el tamaño de la entrada es $O(\log(n))$, nuestro algoritmo sería exponencial. Estos algoritmos que son polinomiales respecto al valor de la entrada, se suelen llamar *pseudo-polinomiales*



Se puede hacer mejor?



Se puede hacer mejor?

Se puede ver que:

$$f(2n+1) = f(n)^2 + 2f(n-1)^2$$

y que:

$$f(2n) = f(n-1)(f(n) + 2f(n-2))$$



Vamos de Paseo...

Estamos organizando el viaje del fin de semana y durante el recorrido vamos a pasar por n ciudades hasta llegar a nuestro recorrido. Tenemos una promo con la cual en la i -ésima ciudad podemos cargar el tanque lleno por c_i (No importa cuánto tengamos en el tanque). Si sabemos que la distancia entre la ciudades i e $i + 1$ es d_i y que con un tanque lleno podemos viajar D kilómetros. ¿Cual es el menor costo que puede tener el viaje? ¿En cuales ciudades tenemos que cargar el tanque?

Decidimos resolver usando la función " $f(x)$ = costo mínimo para llegar a la ciudad x "

Sabemos que $f(0) = 0$, porque no nos cuesta nada llegar a la ciudad de la que partimos

Para la función recursiva, notemos que existe una ciudad x donde, en nuestro recorrido óptimo, habremos cargado combustible por última vez antes de llegar a n .

Llegar a esta ciudad nos cuesta $f(x)$ y cargar combustible en la misma c_x

Así tenemos que $f(n) = f(x) + c_x$.

No sabemos cual es x , pero podemos probar entre todas las candidatas (las que están a distancia menor que D)

Luego tenemos que

$$f(n) = \min_{d(n, i) < D} (f(i) + c_i) = \min(f(k) + c_k, f(k+1) + c_{k+1}, \dots, f(n-1) + c_{n-1}),$$
 donde k es la primera ciudad a distancia menor que D a n

La complejidad del algoritmo es $O(n^2)$, ya que tenemos n posibles estados, y cada uno tiene complejidad $O(n)$, ya que en el peor caso tenemos que calcular el mínimo de entre n valores.

Para obtener las ciudades en las que tenemos que cargar combustible, podemos buscar entre las ciudades que están a distancia menor a D de n , aquella/s x tales que $f(n) - c_x = f(x)$. Cuando la encontremos, habremos encontrado la última en la que cargamos combustible antes de llegar a n . Para encontrar la anterior en la que cargamos, buscaremos un y , tal que $f(y) + c_y = f(x)$. Y así sucesivamente, hasta llegar a la ciudad 0, la ciudad de partida.

El análisis de la complejidad es similar al de obtener el costo. Calcular $f(n)$ para la primera iteración es $O(n^2)$ y todas las siguientes llamadas a f , serán $O(1)$, ya que estarán memorizadas. Como en cada paso chequeamos entre $O(n)$ ciudades y esto lo hacemos $O(n)$ veces, esto también nos cuesta $O(n^2)$, con lo que la complejidad es $O(n^2)$.

Si les interesa volver a pensar el problema cuando sepan bottom-up, pueden intentar resolverlo en $O(n)$ usando una minqueue:

https://cp-algorithms.com/data_structures/stack_queue_modification.html

Como comentario, este problema también se puede resolver con otras funciones recursivas, como por ejemplo:

$g(x)$ = "costo mínimo para llegar a n si parto desde x "

en este caso, $g(n) = 0$

y $g(x) = c_x + \min(g(i))$ con $i > x$ ^ $d(x, i) < D$