1.5em

Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructuras de Datos y Algoritmos

Ayudantía 5

29 de junio de 2020

Programación Dinámica

Lanzamiento de dados

Suponga que usted tiene n dados de m caras cada uno. Dado un entero cualquiera x, encuentre la cantidad de formas en que, con los n lanzamientos, pueda sumar x.

Matrix Snek

Dada una matriz A de m x n, escribe un algoritmo capaz de encontrar la `serpiente` de largo mayor. Se define una `serpiente` como una secuencia de números adyacentes con a lo más 1 de diferencia entre elementos consecutivos. La serpiente sólo se puede mover hacia abajo y a la derecha.

$$A = \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{1} & 2 \\ 7 & 8 & \mathbf{2} & 5 \\ 6 & 6 & \mathbf{3} & \mathbf{2} \\ 9 & 3 & 2 & \mathbf{3} \end{bmatrix}$$

Ejercicio propuesto: Considerar que la serpiente se puede mover en cualquier dirección.

Solución

Lanzamiento de dados

En primer lugar, podemos notar que un buen acercamiento para este problema es 'condicionar' el resultado de un dado, y resolver un subproblema con un lanzamiento menos. Para esto, llamaremos `Suma(m,n,x)` al algoritmo. Utilizando la idea anterior, podemos definir:

function SUMA(m,n,x):
Asumiendo que lancé un 1, encontrar de cuantas formas sumar $x-1$ con $n-1$ dados
+ Asumiendo que lancé un 2, encontrar de cuantas formas sumar $x-2$ con $n-1$ dados
+ Asumiendo que lancé un 3, encontrar de cuantas formas sumar $x-3$ con $n-1$ dados
...
+ Asumiendo que lancé un m, encontrar de cuantas formas sumar $x-m$ con $n-1$ dados

De esta forma, podemos dividir este problema en **subproblemas mas pequeños**. Si seguimos haciendo este desarrollo, podremos notar que, debido a la estructura del problema, existe un **traslape** de problemas. Estas dos características son esenciales para resolver un problema utilizando programación dinámica.

La forma más fácil de abordar este problema es mediante la definición recursiva que definimos anteriormente. Para la definición de los casos bases, abordaremos 3 casos:

1. **Suma a buscar muy alta:** Si la suma a buscar es mayor que el numero de caras por el número de lanzamientos, quiere decir que aunque obtengamos siempre el número más alto nunca alcanzaremos a obtener el valor, por lo tanto, la cantidad de formas de obtener ese valor es 0.
2. **Suma muy baja:** Si la suma a buscar es menor que la cantidad de lanzamientos, quiere decir que aunque obtengamos siempre el número más bajo (1), la suma obtenida será siempre mayor, por lo tanto, la cantidad de formas de obtener ese valor es 0.
3. **Último lanzamiento :** Si sólo nos queda un lanzamiento, y la suma a obtener es menor que la cantidad de caras, sólo podremos obtener ese valor lanzándolo directamente, por lo tanto, la cantidad de formas de obtener ese valor es 1.

Siguiendo la definición anterior, la implementación del algoritmo en código será:

```
1: function FindWays(m,n,x):
2:     if x > m · n or x < n then
3:         return 0                                  ▷ Primeros dos casos base
4:     else if n = 1 then
5:         return 1                                  ▷ Tercer caso base
6:     else
7:         suma ← 0
8:         for i = {1,...,min(x,m)} do
9:             suma += FindWays(m,n-1,x-i)           ▷ Restamos m, o hasta llegar a 0
10:        end for
11:        return suma
12:    end if
13: end function
```

Esta solución es recursiva solamente, y resolvemos muchas veces cada subproblema. Para solucionar esto, creamos una matriz para almacenar las soluciones de los $x \cdot n$ subproblemas que aparecen.

Algorithm 1 Solución con almacenamiento de resultados

```
1: function InitializeMemo(x, n):
2:     memo ← zeros(n,m)                                   ▷ n Filas, m Columas
3:     return memo
4: end function
5:
6: function FindWaysRecursive(m, n, x, memo):
7:     if memo[n − 1][x − 1] then
8:         return memo[n − 1][x − 1]      ▷ Si ya lo calculamos, solo retornamos el valor
9:     else if x > m · n or x < n then
10:        result ← 0                                      ▷ Primeros dos casos base
11:    else if n = 1 then
12:        result ← 1                                      ▷ Tercer caso base
13:    else
14:        suma ← 0
15:        for i = {1,...,min(x,m)} do
16:            suma += FindWaysRecursive(m,n-1,x-i,memo)
17:        end for
18:        result ←suma
19:        memo[n − 1][x − 1]←result
20:        return result   ▷ Guardo en result, y antes de retornarlo, lo guardo en memo
21:    end if
22: end function
23:
24: function FindWays(m, x, n):
25:    memo = InitializeMemo(x, n)
26:    return FindWaysRecursive(m, x, n,memo)
27: end function
```

Solución Snek

Programación Dinámica

Overlapping Subproblems: Para encontrar una serpiente de largo n, podemos usar una serpiente de largo $n-1$ que sea alcanzable desde otra celda contigua. Como no queremos calcular cada serpiente por sus $n-1$ celdas cada vez que queramos construir una de largo n, intuimos que será conveniente usar programación dinámica para resolver el problema.

Bottom-Up

Como la naturaleza del problema induce a calcular primero serpientes pequeñas y esas ir alargándolas, además del hecho de que podemos ir construyéndolas en un orden pre-definido, se resolverá con enfoque *bottom-up*. Necesitamos una estructura eficiente para guardar las soluciones más pequeñas. Como es un problema sobre *arrays* 2D, será necesario otro *array* 2D para guardar las soluciones parciales hasta cada elemento.

Caso base: Todas las celdas de la matriz pueden verse como una serpiente de largo 1.

Lógica Algoritmo

Vamos a recorrer cada celda de la matriz una sola vez desde fila superior a inferior y columna izquierda a derecha, para siempre haber visitado los predecesores de una celda c antes de evaluarla.

Después de evaluar los largos, usamos un stack para reconstruirla desde el final hasta el inicio.

1. Primero inicializamos las estructuras y varaibles.
2. Como ya se dijo antes, recorremos cada elemento de A solamente una vez.
    a) Para cada elemento se revisan sus precursores. Estas son la celda arria y la celda a la izquierda del elemento.
    b) En estas se revisa que cumpla que las celdas estén conectadas (diferencia nivel ¡=1).
    c) Si eso se cumple se revisa si se actualiza el valor guardado en B. Este representa el largo mayor acumulado hasta el elemento revisado. Si el del antecesor es igual o mayor al de la casilla que se está revisando, el valor de la casilla aumenta y toma como nuevo valor el del antecesor + 1. En caso contrario se queda igual.
    d) Después, en caso de ser mayor, se actualiza el valor del mejor largo con respecto del valor de B de la casilla revisada.
3. Luego, se construye el Snek iterando sobre su largo final, partiendo en la celda final.
    a) Se agrega la celda al stack

b) Se busca si la celda adyacente hacia arriba tenía un largo menor en 1 y estaba conectada,

c) Si no, se hace lo mismo para la de la izquierda

4. Finalmente, se puede retornar el Snek

Notar que este algoritmo con programación dinámica es $\mathcal{O}(n)$, con n el número de celdas en A, ya que para calcular la matriz B se visita una vez cada celda con cantidad de pasos constantes, y la impresión se ve acotada por dicha construcción.

Implementación

```
1: function FindSnek(A,m,n)
2:     B ← Matriz de tamaño m x n llena de 1s
3:     Mejor_Largo ← 0
4:     Columna_Final ← 0
5:     Fila_Final ← 0
6:     Snek ← Stack
7:     ▷ Como la serpiente se mueve sólo hacia abajo y derecha, se parte arriba e izquierda
8:     for i = 0...m do
9:         for j = 0...n do
10:            if i = j = 0 then
11:                Continuar                    ▷ No consideramos la primera celda
12:            end if
13:            if A_{i,j} = A_{i-1,j} ± 1 then
14:                B_{i,j} ← máx(B_{i-1,j} + 1, B_{i,j})
15:            end if
16:            if A_{i,j} = A_{i,j-1} ± 1 then
17:                B_{i,j} ← máx(B_{i,j-1} + 1, B_{i,j})
18:            end if
19:            if Mejor_Largo < B_{i,j} then    ▷ Tenemos una serpiente más larga?
20:                Columna_final ← j
21:                Fila_final ← i
22:                Mejor_Largo ← B_{i,j}
23:            end if
24:        end for
25:    end for
26:    for i = 0 ... Mejor_Largo do             ▷ Reconstrucción de atrás hacia adelante
27:        Snek.push(A_{Fila_F,Col_F})
```

28: **if** $Fila\_F > 0 and A_{Fila\_F,Col\_F} = A_{Fila\_F-1,Col\_F} and B_{Fila\_F-1,Col\_F} = B_{Fila\_F,Col\_F} - 1 \pm 1$ **then**
29: $Fila\_F-$
30: **else if** $A_{Fila\_F,Col\_F} = A_{Fila\_F,Col\_F-1} and B_{Fila\_F,Col\_F-1} = B_{Fila\_F,Col\_F} - 1 \pm 1$ **then**
31: Col_F-
32: **end if**
33: **end for**
34: **RETORNAR** Snek
35: **end function**