Enunciado: AED III - Practica 1: Técnicas algorítmicas - Ejercicio 12 B

Se arrojan simultáneamente n dados, cada uno con k caras numeradas de 1 a k. Queremos calcular todas las maneras posibles de conseguir la suma total $s \in \mathbb{N}$ con una sola tirada. Consideramos que los dados son indistinguibles, es decir que si $n = 3$ y $k = 4$, entonces existen 3 posibilidades que suman $s = 6$:

1. Un dado vale 4, los otros dos valen 1
2. Un dado vale 3, otro 2 y otro 1
3. Todos los dados valen 2

a) Definir en forma recursiva la función $G : \mathbb{N} \to \mathbb{N}$ tal que $G(n, s, k)$ es igual a la cantidad de maneras de sumar s con n dados indistinguibles de k caras.

b) Demostrar que G tiene la propiedad de superposición de subproblemas

c) Definir un algoritmo top-down para calcular $G(n, s, k)$ indicando claramente las estructuras de datos utilizadas y la complejidad resultante.

d) Escribir el (pseudo-)código del algoritmo top-down resultante.

Contexto y conocimientos previos esperados

Este ejercicio se encuentra en la guía practica de técnicas algorítmicas, más precisamente, es el último de la sección de programación dinámica, por lo que si fuese presentado en una clase expositiva, probablemente sea dado como un ejercicio de nivel avanzado del tema, o de parcial.

Igualmente podemos notar que el ejercicio es guiado por sus incisos, hasta llegar al código del algoritmo resultante, lo que permitiría detallar paso por paso las técnicas y herramientas usadas para el desarrollo del mismo.

Se asume que las y los estudiantes dominan la técnica de backtracking, y poseen una idea general de la técnica de programación dinámica (PD), es decir, poseen la noción de función recursiva, comprenden el concepto de memoización, y logran descifrar la complejidad de distintas operaciones básicas, y de problemas que utilicen PD, ya que el objetivo del ejercicio es cerrar el tema y fijar ideas del mismo.

Resolución

Ahora mostraría dos o tres ejemplos más con su correspondiente salida, para garantizar que el alumnado comprenda el enunciado (en especial la parte de dados indistinguibles)

Para la resolución tendremos esencialmente 4 pasos, uno por cada inciso, primero plantear la función recursiva, luego probar que hay superposición de problemas, describir el algoritmo a usar, y finalmente plantear el pseudocódigo del mismo.

Función recursiva

¿Cuál sería la semántica de la función recursiva?

Dadas los siguientes resultados de tiradas:

5 2 2 y **2 5 2**

¿Cómo hacemos para no contarlas como tiradas diferentes?

Para ello pensaremos que ordenamos los dados, y así consideraremos estas tiradas como iguales (**2 2 5**), por lo que podremos definir así la semántica de nuestra función recursiva:

$G(n, s, k)$: Cantidad de maneras ordenadas de obtener s tirando n dados de k caras.

¿Cuál sería el paso recursivo de la función?

Considerando a los dados ordenados, nos centraremos en pensar que pasa con el último dado, donde tendremos dos posibilidades, es k, y entonces la cantidad de formas ordenadas que tengo para los otros dados es $G(n-1, s-k, k)$, o no es k , por lo que es menor o igual a $k-1$, y como está ordenado, los demás también lo son. Podemos deducir entonces que la cantidad formas ordenadas es igual a $G(s, n, k-1)$.

¿Cuáles serán los casos base de nuestra recursión?

Para comenzar notemos que si $n = 0$, la única configuración de dados posibles es el conjunto de dados el conjunto vacío, puesto que significaría que no tiramos dados, y por lo tanto es 1 si $s = 0$ y 0 en otros casos. Por otro lado, si $k \leq 0$ no podemos armar ninguna configuración, ya que las caras de los dados van de 1 a k. Finalmente podemos agregar que si $s > nk$, entonces no tenemos ninguna configuración posible, ya que lo máximo que podemos obtener con n dados es nk.

Ya planteado el paso recursivo y los casos base, estamos en condiciones de definir G

$$G(n,s,k) = \begin{cases} 1 & \text{si } n = 0 \text{ y } s = 0 \\ 0 & \text{si } n = 0 \text{ y } s \neq 0 \\ 0 & \text{si } k = 0 \\ 0 & \text{si } nk > s \\ G(n-1, s-k, k) + G(n, s, k-1) & \text{caso contrario} \end{cases}$$

En este paso preguntaría por las posibles dudas que surjan, y trataría de explicarlas, pues la definición de la función recursiva es la base de los incisos siguientes.

Superposición de problemas

¿Cuándo hay superposición de problemas?

Recordando lo que ya habrán visto, tenemos que plantear la cantidad de llamadas recursivas de nuestra función contra la cantidad de subproblemas, o formas de llamar a la función que existen.

Notemos que cuando cuando hacemos las llamadas recursivas, o bien el valor de n o el de k disminuye en uno, así consideramos el caso cuando n, k y s son suficientemente grandes, como en caso recursivo hacemos 2 llamadas recursivas, el número de llamadas recursivas es $\Omega(2^{\min(n,k)})$.

Mientras que la cantidad de subproblemas en cambio es igual a $nk \times \min(nk, s) \subseteq \mathcal{O}(n^2k^2)$.

Así cuando se cumpla $2^{\min(n,k)} >> n^2k^2$, tenemos superposición de problemas.

Definimos la memoización y el algoritmo Top-Down

¿Cómo guardamos llamados repetidos?¿Por qué los guardamos?

Tenemos que determinar la estructura de memoria, la cual nos permitirá utilizar la superposición de problemas para reducir la complejidad (usando aquí la técnica de programación dinámica). Para ello, recordando las distintas formas posibles de llamar a nuestra función, en este caso, tenemos 3 parámetros, y para cada uno de ellos conocemos su rango de valores posibles, por lo que utilizaremos una matriz M de tamaño $n \times s \times \min(nk, s)$ para guardar los valores ya calculados, y la inicializamos con un valor de indefinido ($\perp$). Una vez inicializada la memoria, procederemos a plasmar en nuestro algoritmo lo pensado con la función recursiva:

- Si $n = 0$ retornamos 1 si $s = 0$, si no 0
- Si $k = 0$ o $nk > s$ devolvemos 0, ya que vimos que no hay configuraciones posibles.
- Si $M[s, n, k] = \perp$, llamamos recursivamente y guardamos $M[s, n, k] \leftarrow G(s-k, n-1, k) + G(s, n, k-1)$
- Si ya tenemos $M[s, n, k]$ lo devolvemos.

Antes de presentar el pseudocódigo preguntaría si hay alguna duda sobre la memoización, pues es el paso clave de la programación dinámica.

Damos el pseudocódigo y complejidad del algoritmo

¿Cambia mucho la explicación del algoritmo al pseudocódigo?

En este caso no, es simplemente escribir lo plasmado anteriormente en pseudocódigo, con las aclaraciones correspondientes a la complejidad del algoritmo en un costado.

Algoritmo 1 $G(n, s, k)$ calcula la cantidad de maneras ordenadas de sumar s con n dados de k caras.

```
1: M ← matriz de dimensión n × s × min(nk, s) inicializada en ⊥        ▷ O(nk min(nk, s))
2: function G(n, s, k)
3:     if n = 0 and s = 0 then
4:         return 1                                                    ▷ O(1)
5:     end if
6:     if n = 0 or k = 0 or nk > s then
7:         return 0                                                    ▷ O(1)
8:     end if
9:     if M[s][n][k] =⊥ then
10:        M[s][n][k] ← G(n − 1, s − k, k) + G(n, s, k − 1)            ▷ O(1)
11:    end if
12:    return M[s][n][k]                                               ▷ O(1)
13: end function
```

Complejidad Temporal: $\mathcal{O}(nk \min(nk, s))$

¿Cómo calculamos la complejidad del algoritmo?

Recordemos que la complejidad de un algoritmo de programación dinámica, como el que realizamos, es equivalente a:

$$\textit{Cantidad de estados} \times \textit{Costo de calcular cada estado}$$

En nuestro caso, tenemos $nk \min(nk, s)$ estados, mientras que cada estado se calcula operaciones básicas, llamando recursivamente a la función, y luego sumando los resultados ($\mathcal{O}(1)$), por lo que la complejidad final del algoritmo es $\mathcal{O}(nk \min(nk, s))$, notada en la inicialización de la estructura de memoización (matriz).

Aquí es donde se nota la importancia de agregar el último caso base de nuestra recursión pues pudimos reducir aún más la complejidad, tomando el mínimo entre s y nk.

Por último mostraría dos ejemplos cortos detallando que realiza el algoritmo en cada paso, y daría pie a que los alumnos resuelvan el problema de tratar con dados distinguibles, en lugar de indistinguibles, agregando la sugerencia de que ahora se puede fijar el k, y trabajar únicamente con una función $F(n, s)$, con el objetivo que apliquen estrategias similares a las desarrolladas.

Conclusión

Con este ejercicio tratamos sobre las ideas generales de la programación dinámica, y resolvimos en clase, mediante una explicación detallada de cada paso un problema con dificultad similar a un parcial. Mostramos cómo pensar una función recursiva, desde el paso recursivo a los casos base, cómo probar la superposición de problemas. Además reforzamos el concepto de memoización, y el procedimiento a seguir para calcular la complejidad de un algoritmo con programación dinámica.