DadosSuma

Enunciado

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. Tomamos dos variantes de este problema.

(A) Consideramos que los dados son **distinguibles**, es decir que si $n = 3$ y $k = 4$, entonces existen 10 posibilidades que suman $s = 6$:
   (1) 4 posibilidades en las que el primer dado vale 1
   (2) 3 posibilidades en las que el primer dado vale 2
   (3) 2 posibilidades en las que el primer dado vale 3
   (4) Una posibilidad en la que el primer dado vale 4

(B) Consideramos que los dados son **distinguibles**, 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, el otro 2, y el otro 1
   (3) Todos los dados valen 2.

a) Definir en forma recursiva la función $f : \mathbb{N}^2 \to \mathbb{N}$, tal que $f(n, s)$ devuelve la respuesta para el escenario (A) (fijado k).
b) Definir en forma recursiva la función $g : \mathbb{N}^3 \to \mathbb{N}$, tal que $f(n, s, k)$ devuelve la respuesta para el escenario (B).

Extra: En cada ejercicio, demostrar que el problema exhibe subproblemas compartidos. Luego dar una implementación usando programación dinámica.

Resolución

a) Sea un $k \in \mathbb{N}$ fijo. Sea $f(n, s)$ el número de formas de tirar n dados, donde cada uno puede valer desde 1 hasta k, y entre todos suman s. Para este ítem, los consideramos distinguibles.

   Esto es lo mismo que contar las tuplas de n elementos, donde el i-ésimo elemento vale desde 1 hasta k, y los valores de la tupla suman s. La tupla representa, en su i-ésimo elemento, cuánto muestra el i-ésimo dado.

   Lo siguiente debe ser cierto para nuestra función f:
   1) $f(0, 0) = 1$. Hay sólo una manera de tirar cero dados y que su suma de cero. Esa manera es no hacer nada.
   2) $f(0, s) = 0$ para todo $s \neq 0$. No hay maneras de tirar cero dados y terminar con una suma distinta a cero.
   3) $f(n, s) = \sum_{i=1}^{k} f(n-1, s-i)$ para todo $n > 0$. Toda forma de tirar n dados, y sumando s, donde los dados son distinguibles, se descompone como tirar el primer dado y que de i, y luego tirar $n-1$ dados y que la suma de estos sea $s-i$, para cada $1 \leq i \leq k$.

   Estas propiedades son suficientes para definir la función pedida.

$$f(n, s) := \begin{cases} 1 \text{ si } n = s = 0 \\ 0 \text{ si } n = 0,\ s \neq 0 \\ \displaystyle\sum_{i=1}^{k} f(n-1, s-i) \text{ si } n > 0 \end{cases}$$

   Una implementación simple de esta función recursiva es:

```
def f(n, s):
  if n == 0 and s == 0: return 1
  if n == 0 and s != 0: return 0
  return sum(f(n - 1, s - i) for i in range(1, k + 1))
```

Vemos que esta función se detiene cuando $n = 0$. Cada llamada se detiene, o hace k llamadas anidadas, donde n baja en 1. Luego, el árbol de recursión tiene k^n vértices. En cada uno hacemos $O(k)$ trabajo de sumar k resultados. Luego el tiempo que toma esta función está en $O(k^{n+1})$.

Vemos que, como cada vértice es un par (i, j), y tenemos $0 \leq i \leq n$, $0 \leq j \leq s$, hay sólo $(n + 1)(s + 1)$ posibles subproblemas distintos, y esto es asintóticamente menor a k^{n+1}, cuando $n \to \infty$. Esto nos sugiere una implementación con programación dinámica.

Una implementación con programación dinámica top-down es meramente poner un cache de valores a la función recursiva:

```
cache = dict()
def f(n, s):
  if n == 0 and s == 0: return 1
  if n == 0 and s != 0: return 0
  if (n, s) not in cache:
    cache[(n, s)] = sum(f(n - 1, s - i) for i in range(1, k + 1))
  return cache[(n, s)]
```

Notar cómo este proceso es puramente mecánico. No requiere pensar. Sin embargo, el comportamiento asintótico de esta función es difícil de analizar, puesto que ya no es una función pura - muta el estado `cache`. En contraste, una implementación con programación dinámica bottom-up es:

```
def f(n, s):
  T = [[0]*(s+1) for _ in range(n+1)]
  for i in range(n + 1):
    for j in range(s + 1):
      if i == 0:
        T[i][j] = 1 if j == 0 else 0
      else:
        T[i][j] = sum(T[i - 1][j - r] for r in range(1, min(j, k) + 1))
  return T[n][s]
```

Notar que usamos `min(j, k) + 1` puesto que, al decir `T[i - 1][j - r]`, necesitamos que `j - r >= 0`. Es decir, que `r <= j`. Luego, en vez de usar `range(1, k + 1)`, usamos `range(1, min(j, k) + 1)`.

Este algoritmo es mucho más fácil de analizar asintóticamente, haciendo $O(nsk)$ operaciones, y usando $O(ns)$ espacio temporario. La transformación de la función recursiva a bottom-up requiere pensar en qué orden podemos llenar la tabla `T`, tal que para cuando necesitemos leer un valor de `T`, ya lo hayamos computado anteriormente. En este caso, vemos que para llenar `T[i][j]`, necesitamos leer `T[i - 1][j - r]` para varios `r`. Luego, mientras llenemos la tabla en orden creciente de `i`, siempre vamos a leer de `T` valores que previamente llenamos.

Notar también cómo podemos mejorar fácilmente la complejidad espacial de esta segunda implementación bottom-up. No necesitamos mantener una tabla `T` de tamaño `n * s`, porque siempre leemos sólo una fila anterior. Luego, podemos usar sólo dos filas, una anterior que leemos, y una actual que escribimos. El invariante que mantenemos es que al terminar la i-ésima iteración del ciclo exterior, `TA[j] = f(i, j)` para todo `j`:

```python
def f(n, s):
  TA = [0] * (s + 1)
  TB = [0] * (s + 1)
  TA[0] = 1
  for i in range(1, n + 1):
    for j in range(s + 1):
      TB[j] = sum(TA[j - r] for r in range(1, min(j, k) + 1))
    TA, TB = TB, TA
  return TA[s]
```

Obteniendo así una complejidad espacial de $O(s)$, y manteniendo la complejidad temporal anterior de $O(nsk)$.

b) Sea $g(n, s, k)$ el número de formas de tirar n dados, donde cada uno puede valer desde 1 hasta k, y entre todos suman s. Para este ítem, los consideramos indistinguibles. Esto es lo mismo que contar las tuplas T de k elementos, donde el i-ésimo elemento de la tupla te dice cuántos dados tienen valor i, y $\sum_{i=1}^{k} iT_i = s$ (es decir, suman s), y $\sum_{i=1}^{k} T_i = n$ (es decir, hay n dados en total). Notar que podemos tener ceros en estas tuplas, si para algún $1 \leq i \leq k$, ningún dado valió i.

Lo siguiente es cierto para nuestra función g:

1) $g(0, 0, k) = 1$ para todo k. Hay exactamente una forma de sumar $s = 0$ con $n = 0$ dados, independientemente de cuánto pueda valer cada uno de los cero dados, y es no hacer nada.
2) $g(n, s, 0) = 0$ para todo $(n, s) \neq (0, 0)$. Como todos los dados tienen valores del 1 hasta 0, no puede haber ningún dado, y luego si n y s no son ambos cero, no hay manera de hacer tales combinaciones.
3) $g(\_, s, \_) = 0$ para todo $s < 0$. Todos los dados tienen valores positivos.
4) $g(n, \_, \_) = 0$ para todo $n < 0$. No puedo tirar un número negativo de dados.
5) $g(n, s, k) = \sum_{i=0}^{n} g(n - i, s - ik, k - 1)$. Esto es poner un i en la k-ésima componente de la tupla, significando que exactamente i dados tuvieron valor exactamente k. Para que en total sumen s, el resto de los dados tienen que sumar $s - ik$, y tiene que haber $n - i$ de ellos. Como dijimos que exactamente i tienen valor k, todos los demás dados tienen que tener valor a lo sumo $k - 1$ cada uno.

Esto es suficiente para definir la función.

$$g(n, s, k) := \begin{cases} 1 \text{ si } n = s = 0 \\ 0 \text{ si } (n, s) \neq (0, 0), \quad k = 0 \\ 0 \text{ si } s < 0 \\ 0 \text{ si } n < 0 \\ \sum_{i=0}^{n} f(n - i, s - ik, k - 1) \text{ en otro caso} \end{cases}$$

Una implementación recursiva de esta función es:

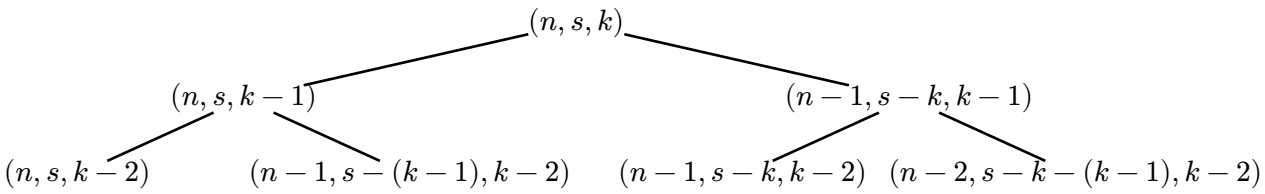
```python
def g(n, s, k):
  if n == 0 and s == 0: return 1
  if k == 0: return 0
  if s < 0: return 0
  if n < 0: return 0
  return sum(g(n - i, s - i * k, k - 1) for i in range(n + 1))
```

El número de llamadas distintas es a lo sumo nsk. Para mostrar que hay subproblemas compartidos, queremos encontrar una cota inferior para el número de llamadas hechas. No en

todos los casos va a haberlos, por ejemplo, si $k = 1$, vamos a hacer n llamadas ($g(n - i, s - ik, 0)$ para $1 \leq i \leq n$), ninguna va a ser igual a ninguna otra, y todas van a terminar inmediatamente porque tienen $k = 0$. Luego, al mostrar que hay subproblemas compartidos queremos ver que en el *peor* caso, una implementación recursiva realiza muchos subproblemas compartidos. Encontremos, entonces, una familia de valores (n, s, k), para los cuales la función recursiva hace mucho más que nsk llamadas.

Para ver cuáles llamadas son hechas, tenemos que analizar el árbol de recursión de esta función. Esta función se detiene cuando $n \leq 0$, $s \leq 0$, o $k = 0$. Queremos que este árbol tenga muchos vértices, entonces queremos que pare lo más tarde posible. En cada llamada, k baja en 1, n baja en i, y s baja en ik. Hagamos que pare en la condición que tarda más en ocurrir, $k = 0$.

Veamos qué pasa, entonces, cuando miramos a los vértices que usaron i pequeños (los que usaron i grandes van a terminar antes, sea por la condición de n o la de s). En particular, miremos sólo $i = 0$ y $i = 1$. En el siguiente árbol, las ramas izquierdas son tomar $i = 0$, las ramas derechas son tomar $i = 1$.



Si hacemos que que este árbol no termine antes de llegar a $k = 0$, tendremos un árbol binario perfecto de altura k, y habremos mostrado que tenemos al menos 2^k vértices en el árbol de recursión. Para evitar que lleguemos a $n = 0$, podemos tomar $n = k + 1$. Como siempre bajamos a n por a lo sumo 1, nunca vamos a terminar con $n = 0$ antes que $k = 0$. Para evitar que lleguemos a $s = 0$, queremos que $s > \sum_{i=0}^{k} k - i = \frac{k(k+1)}{2}$. Luego, tomando $s = 1 + \frac{k(k+1)}{2}$, el árbol de recursión no se va a agotar hasta que $k = 0$, y por tanto encontramos un sub-árbol con 2^k vértices.

En esta familiar, vamos a tener sólo $nsk = (k + 1)\left(1 + \frac{k(k+1)}{2}\right)k$ vértices distintos. Esto es $O(k^4)$ vértices distintos. Sin embargo, tenemos en el árbol de recursión un sub-árbol de 2^k vértices. Luego, cuando $k \to \infty$, el algoritmo recursivo repite un número exponencial de vértices. Esto es tener subproblemas compartidos.

Como encontramos subproblemas compartidos, tiene sentido usar programación dinámica. Un algoritmo top-down de programación dinámica va a ser:

```
cache = dict()
def g(n, s, k):
  if n == 0 and s == 0: return 1
  if k == 0: return 0
  if s < 0: return 0
  if n < 0: return 0
  if (n, s, k) not in cache:
    cache[(n, s, k)] = sum(g(n - i, s - i * k, k - 1) for i in range(n + 1))
  return cache[(n, s, k)]
```

Una versión con programación dinámica bottom-up va a ser:

```
def g(n, s, k):
  TA = [[0]*(s + 1) for _ in range(n + 1)]
  TB = [[0]*(s + 1) for _ in range(n + 1)]
```

```
TA[0][0] = 1
for kk in range(1, k + 1):
  for nn in range(n + 1):
    for ss in range(s + 1):
      TB[nn][ss] = sum(TA[nn - i][ss - i * kk]
                       for i in range(nn + 1)
                       if i <= nn and ss >= i * kk)
  TA, TB = TB, TA
return TA[n][s]
```

Notar cómo usamos sólo $\Theta(ns)$ espacio, y no $\Theta(nsk)$ espacio. Esto es porque cada vez que llamamos a un valor anterior de g, lo necesitamos con un k menor en exactamente 1. Luego, el invariante que mantenemos en este ciclo es que `TA[nn][ss] = g(nn, ss, kk - 1)` para todo `nn, kk`. Usamos a `TB` para escribir los valores de `g(nn, ss, kk)`, y al finalizar esta iteración de `kk`, intercambiamos `TA` por `TB`.

Esta es la ventaja de usar programación dinámica bottom-up, por sobre top-down: Al tener control sobre el órden en el cual llenamos la tabla de valores computados anteriormente, podemos aprovechar esto para reducir la memoria. Asimismo, la versión top-down necesita tiempo para buscar `(n, k, s)` en el cache, que es un diccionario. En este caso podríamos usar una tabla grande, de tamaño `[n][k][s]`, en vez de un diccionario de tuplas. Esto nos ahorraría el tiempo de búsqueda (se convertiría en $O(1)$ en vez de $O(\log(nks))$ con un árbol binario de búsqueda, o $O(1)$ sólo en esperanza si usáramos una tabla de hash), pero seguiría usando k veces más espacio.

La desventaja de usar programación dinámica bottom-up es que:
1) Es más difícil de programar que mecánicamente poner un cache.
2) Si el espacio de subproblemas es grandes, pero exploramos sólo muy pocos de los subproblemas, una solución top-down evita hacer esas llamadas, mientras que en una solución bottom-up no sabemos si un elemento de nuestra tabla va a ser usado, o lo estamos calculando innecesariamente.

Haciendo $O(n)$ trabajo en el ciclo anidado, y teniendo nks iteraciones del ciclo, esta versión usa $O(n^2ks)$ tiempo, y $O(ns)$ espacio.