Particiones

Diego,Iker,Raúl,Pablo,Ismael

La función $p(n)$ cuenta, para un número natural n, de cuántas maneras puede escribirse como suma de enteros positivos, sin tener en cuenta el orden de los sumandos.

Por ejemplo, hay $p(5) = 7$ particiones del número 5:

5 $\quad 4+1$ $\quad 3+2$ $\quad 3+1+1$ $\quad 2+2+1$ $\quad 2+1+1+1$ $\quad 1+1+1+1+1$

n	0	1	2	3	4	5	6	7	8	9	10	⋯
$p(n)$	1	1	2	3	5	7	11	15	22	30	42	⋯

http://oeis.org/A000041

El valor $p(0) = 1$ proviene de la toma en consideración de la partición vacía (la que no tiene ningún sumando) para el número $0 = \sum(\varnothing) = \sum_{i\in\varnothing} i$.

¿Cómo calculamos $p(n)$ a partir de n?

La estrategia que proponemos comienza haciendo más difícil el problema, generalizándolo a la evaluación de la función $p_k(n)$, definida como el número de particiones de n que tienen k partes (k sumandos) como mucho.

n \ k	0	1	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1	1	1
1		1	1	1	1	1	1	1	1
2		1	2	2	2	2	2	2	2
3		1	2	3	3	3	3	3	3
4		1	3	4	5	5	5	5	5
5		1	3	5	6	7	7	7	7
6		1	4	7	9	10	11	11	11
7		1	4	8	11	13	14	15	15
8		1	5	10	15	18	20	21	22

Cf. http://oeis.org/A026820

Por ejemplo, de los dibujos anteriores, solo hay $3 = p_2(5)$ (los tres primeros) que quepan en dos filas.

Esta tabla, destacada en un libro del siglo XVIII, se conoce como *tabla de Euler*.[1] La función $p_k(n)$ toma un valor positivo (es decir: hay alguna partición de n con k partes o menos) cuando $n \geq 0$ y $k > 0$, además de en el caso $n = k = 0$. En cualquier otra situación, definimos $p_k(n) = 0$.

1) *Contando con la capacidad para calcular $p_k(n)$ para parámetros cualesquiera, ¿de qué manera se puede obtener la cantidad de particiones de un número?*

La cantidad de particiones de un número se representa con la notación $p(n)$ y sabiendo que $p_k(n)$ es el número de particiones de un número n en a lo sumo k partes, por tanto, podemos calcular las particiones de un número en a lo sumo n partes, de forma que $k = n$ y se cumple que $p(n) = p_n(n)$

[1] L. Euler: *Introductio in Analysin Infinitorum I* (1748), pp. 327-328.

Además, la función de partición $p(n)$ se puede calcular de diversas maneras, y hay varias fórmulas y enfoques matemáticos para hacerlo. Algunas de las técnicas más comunes incluyen:

- **Recursión**: La función de partición se puede calcular de forma recursiva utilizando la relación de recurrencia de Euler. La fórmula recursiva es la siguiente:

$$p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + p(n-12) + p(n-15) - ...$$

  Donde los números $1, 2, 5, 7, 12, 15, ...$ son números pentagonales generados a partir de una fórmula específica.

- **Tabulación**: Se puede utilizar una tabla o matriz para almacenar los valores de $p(n)$ previamente calculados y luego utilizarlos para calcular $p(n)$ de manera eficiente. Este enfoque se llama programación dinámica y permite evitar cálculos redundantes.

- **Fórmulas Analíticas**: Existen fórmulas analíticas que permiten calcular $p(n)$ de manera más eficiente, aunque pueden ser complejas. Una de las fórmulas más conocidas es la fórmula de Hardy-Ramanujan:

$$p(n) \sim \frac{1}{4n\sqrt{3}} \exp(\pi\sqrt{2n/3}$$

- **Algoritmos Informáticos**: También es posible calcular $p(n)$ utilizando algoritmos computacionales, como la programación dinámica o el enfoque de generación de particiones. Estos algoritmos son eficaces para números relativamente grandes.

La complicación del problema tiene como ventaja la posibilidad de recurrir a la siguiente propiedad:

n \ k	0	1	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1	1	1
1		1	1	1	1	1	1	1	1
2		1	2	2	2	2	2	2 →	2
3		1	2	3	3	3	3	3	3
4		1	3	4	5	5	5	5	5
5		1	3	5	6	7	7	7	7
6		1	4	7	9	10	11	11	11
7		1	4	8 →	11	13	14	15	15
8		1	5	10	15	18	20	21	22

$$p_k(n) = p_{k-1}(n) + p_k(n-k).$$

2) *Explica con tus palabras porque se cumple esta igualdad.*

La función $p_k(n)$ representa la cantidad de formas en que el número "n"se puede particionar utilizando números no mayores que k.

La igualdad se desglosa en dos casos:

- $p_{k-1}(n)$ representa la cantidad de particiones de n en las que no se utiliza el número k. En otras palabras, son las particiones de n utilizando solo números de 1 a $k-1$, ya que excluye k.

- $p_k(n-k)$ representa la cantidad de particiones de $n-k$ utilizando números no mayores que k. La razón de restar k es que estamos considerando al menos un k en cada partición. Entonces, estamos contando las particiones de $n-k$ utilizando números de 1 a k, y luego le sumamos k a cada una de esas particiones, lo que da lugar a particiones de n que incluyen k.

La igualdad se cumple porque, al considerar estos dos casos, cubrimos todas las posibles particiones de n en las que se permite el uso de k. En definitiva, estamos desglosando las particiones en aquellas que incluyen k y aquellas que no lo hacen. Sumando estas dos cantidades, obtenemos el total de particiones válidas de n que cumplen con la restricción de usar números no mayores que k.

Esta relación recursiva es fundamental para calcular las particiones de números de manera eficiente y se basa en principios de combinatoria y teoría de números.

Esto conduce a un algoritmo recursivo para la enumeración de las particiones:

Listing 1: Ej3Particiones.py

```python
#!/usr/bin/env python3
def p(n):
    "Calcula el número de particiones de n."
    return p_act(n, n)

def p_act(n, k):
    '''Calcula el número de particiones
    de n con longitud acotada por k.'''
    if n == k == 0:
        return 1
    elif n >= 0 and k >= 1:
        return p_act(n, k-1)\
            + p_act(n-k, k)
    else:
        return 0

if __name__ == '__main__':
    import sys
    if len(sys.argv) > 1:
        n = int(sys.argv[1])
        print('p(%d) = %d' % (n, p(n)))
```

3) *Prueba este algoritmo con distintos valores del parámetro para hacerte una idea de su alcance.*

Para valores como 5 o 10 la ejecución es instantánea. Incrementando a valores de entorno a 50 ya tarda sobre medio segundo, pero enseguida el tiempo se va aumentando exponencialmente. Para $p(80)$, este algoritmo tarda 30 segundos en producir la salida.

4) *Compáralo con el algoritmo que usa Sage:*

```
$ sage ↲

sage: Partitions(5).cardinality() ↲
7
sage: Partitions(1000).cardinality() ↲
24061467864032622473692149727991
sage: Partitions(10 ** 7).cardinality() ↲
920271755026...
```

En este caso, ejecuciones como las probadas en el ejercicio anterior son instantáneas todas. Incluso para $p(1000)$ también se produce la salida de forma instantánea. Se empiezan a notar retardos en valores como $p(10^7)$, necesitando 5 segundos para computar la salida. En resumen, se puede decir que la diferencia de tiempo entre nuestro algoritmo y el utilizado por Sage es notable.
Hay margen entonces para mejorar la solución de la izquierda.

Podríamos afinar las reglas de determinación de la función $p_k(n)$ para ahorrar algunos pasos de la recursión, pero esto tampoco nos conduce demasiado lejos.

Date cuenta de que

$$\forall k \geq n \ : \ p_k(n) = p_n(n).$$

```python
def p_act(n, k):
    if n == k == 0:
        return 1
    elif n >= 0 and k >= 1:
        if n == 0 or k == 1:
            return 1
        elif k > n:
            return p_act(n, n)
        else:
            return p_act(n, k-1)\
                + p_act(n-k, k)
    else:
        return 0
```

5) *Traduce los programas anterior a Scheme y da una cota sobre la complejidad espacial y temporal.*

La complejidad temporal del algoritmo que calcula $p(n)$ sería $\Theta(2^k)$ ya que el algoritmo explora todas las combinaciones posibles de particiones, donde cada parte puede tomar un valor entre 1 y k.

Sin embargo, la complejidad espacial sería $\Theta(k)$ ya que el algoritmo es dominado por la recursión y la pila de llamadas, y por tanto, la recursión puede llegar a una profundidad de k niveles, ya que k representa el numero máximo de partes permitidas en la partición

6) *Siguiendo la estrategia de «memoization», convierte el algoritmo recursivo «ingenuo» en dos algoritmos* (iterativo/«bottom-up» *y* «top-down») *para el cálculo de las particiones de un número y prográmalos en Scheme.*

7) *Responde a las siguientes preguntas en términos de la complejidad espacial y temporal para los nuevos algoritmos.*

   Hasta alcanzar $p(n)$ con la estrategia bottom-up, *¿cuántos términos de la tabla de Euler se calculan? Para cada uno de estos cálculos, ¿a cuántas valores previamente almacenados se recurre? ¿Cuántas operaciones aritméticas se emplean en la obtención de $p(n)$?*

En el enfoque bottom-up, la tabla de Euler se llena desde $p(0)$ hasta $p(n)$ de manera iterativa. Por tanto, se calculan $n+1$ términos de la tabla de Euler.

Para cada uno de estos calculos con estrategia bottom-up se recurre a k valores de las n filas de la tabla y por tanto en el peor caso se recurriría a n^2 valores previamente almacenados ya que es el caso en que $n = k$. De esta forma se deduce, que la complejidad espacial de dicho algoritmo es $\Theta(n^2)$.

En cuanto al número de operaciones aritméticas, se puede observar que se realiza una suma para cada valor en la tabla de Euler basada en la fórmula $p_k(n) = p_{k-1}(n) + p_k(n-k)$. Además, mirando la fórmula deducimos que por cada valor de la tabla se realizan dos operaciones aritméticas: una suma y una resta: Por lo tanto, se emplean 2 operaciones aritméticas por cada valor calculado.

Sabiendo que se calculan $n+1$ términos, el número total de operaciones aritméticas será $2(n+1)$ que es lineal con respecto de n , por lo que, la complejidad temporal en ambas estrategias (tanto bottom-up como top-down) será de $\Theta(n)$ ya que dicha complejidad se refiere al número de operaciones realizadas por el algoritmo.

Observa que no nos preguntamos por la cantidad de «operaciones máquina». Para completar el estudio de la complejidad del algoritmo, sería necesario acotar la talla de los resultados intermedios. Del resultado indicado al pie se deduce que esta talla se mantiene acotada por un polinomio.

8) *¿Permite algún ahorro en este caso la estrategia* top-down*?*

La estrategia top-down con memoización es más eficiente que una implementación puramente recursiva, ya que evita cálculos repetidos almacenando resultados intermedios.

Sin embargo, en el contexto del cálculo de particiones de un número, la estrategia bottom-up suele ser más eficiente en términos de uso de recursos y tiempo, ya que evita la sobrecarga de llamadas recursivas y cálculos repetidos mediante la construcción de una tabla de memoización y el cálculo en un orden específico.

En conclusión, la estrategia top-down es útil en casos específicos, pero para calcular todas las particiones posibles de un número n en a lo sumo k partes, el enfoque bottom-up suele ser preferible debido a su eficiencia.

Para comprender el algoritmo tan eficiente que usa Sage,[2] hace falta sumergirse más en las matemáticas. En concreto, a partir de la identidad asintótica

Hardy y Ramanujan (1918)

$$p(n) \sim \frac{1}{4n\sqrt{3}} \exp(\pi \sqrt{2n/3}),$$

H. Rademacher obtuvo en 1938 una serie que aproxima la función de particiones y dedujo una cota en $O(\sqrt{n})$ para la cantidad de términos que garantiza la obtención del valor exacto.

[2] F. Johansson: «Efficient implementation of the Hardy-Ramanujan-Rademacher formula», LMS J. Comput. Math. **15**, 2012.