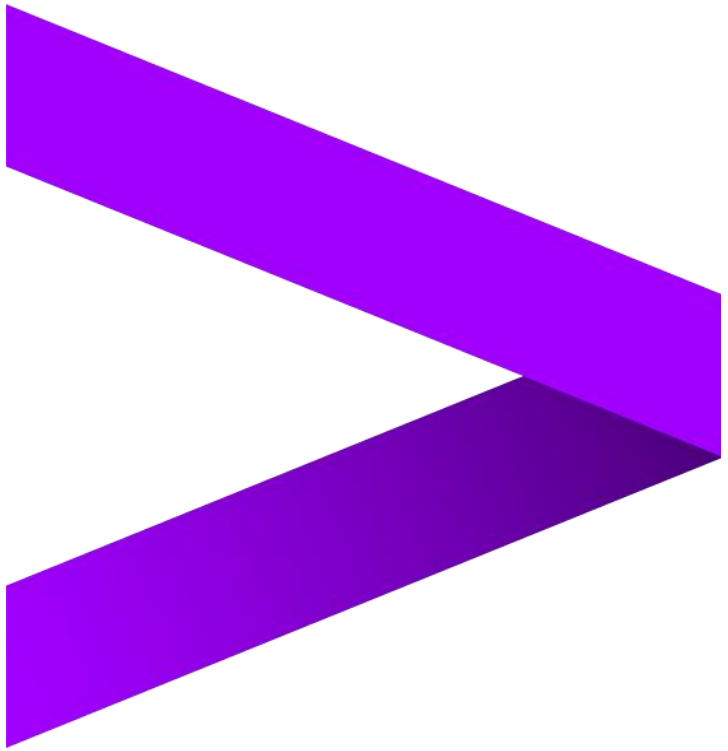
Reto Accenture



Solución por Joaquín Domínguez de Tena

Índice

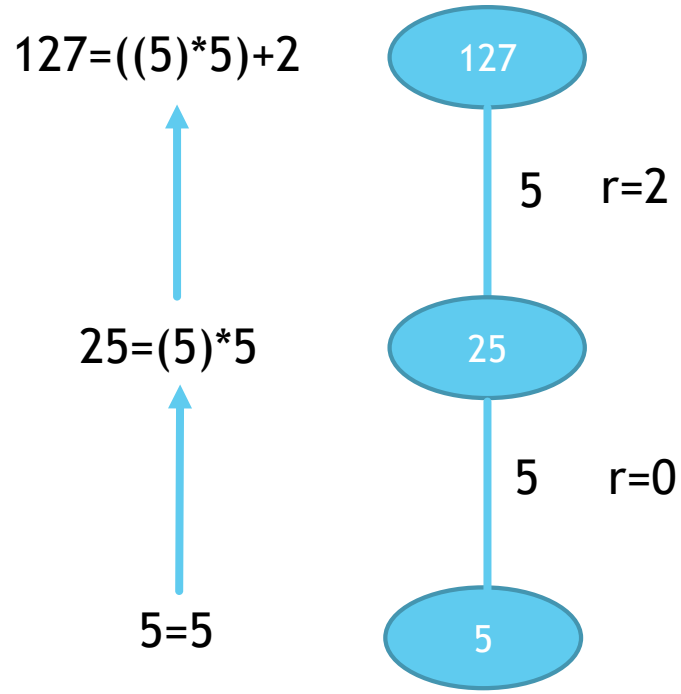
Idea de la solución: El código Binario

- Todo número se puede escribir en código binario en 1 y 0
  - Ej: 27=11001
- Esto se interpreta como suma de potencias de 2:
  - Ej: 27=2^4+2^3+2^0
- Podemos reunir los términos y vemos que esta es una expresión válida como solución
  - Ej: 27=1+2*(1+2*(2*(2+1)))
- Es una solución muy buena, porque depende logarítmicamente del número en cuestión. Para números menores que 2^n utilizará a los sumo 2*n doses o unos (Claro un 2 para la base y 1 o 0 (no contabiliza) para la expresión)
- La idea es que podemos utilizar otras bases de primos, de manera que se reduzca el número de primos usados, por ejemplo 7:
  - Ej: 27=6+7*3 (27=“36 en base 7”)
- Sin embargo aquí usamos números como 6 que no es primo, y tendremos que expresarlos como 2*3! De manera que puede aumentar un poco el coste

Idea de la solución: El código Binario

- Entonces debemos precomputar el coste de cada posible “resto” en términos de primos. Unos pocos cálculos nos daban que cualquiera que fuese el primo a excluir la mayoría de los números se podían escribir como combinación de 3 primos y rara vez, así que no aumenta mucho nuestro coste.
- Ahora planteamos como sacar la expresión de n en base un primo arbitrario:
  - 1) Tomamos n y hacemos a=n%p. Esta será nuestra primera “cifra”
  - 2) Consideramos n’=(n-a)/p
  - 3) Si n’<p entonces esta esta es la última cifra y el algoritmo acaba
  - 4) Si no, entonces repetimos el algoritmo para obtener la siguiente cifra
- A continuación mostramos un gráfico que muestra el algoritmo:

Idea de la solución: El código Binario



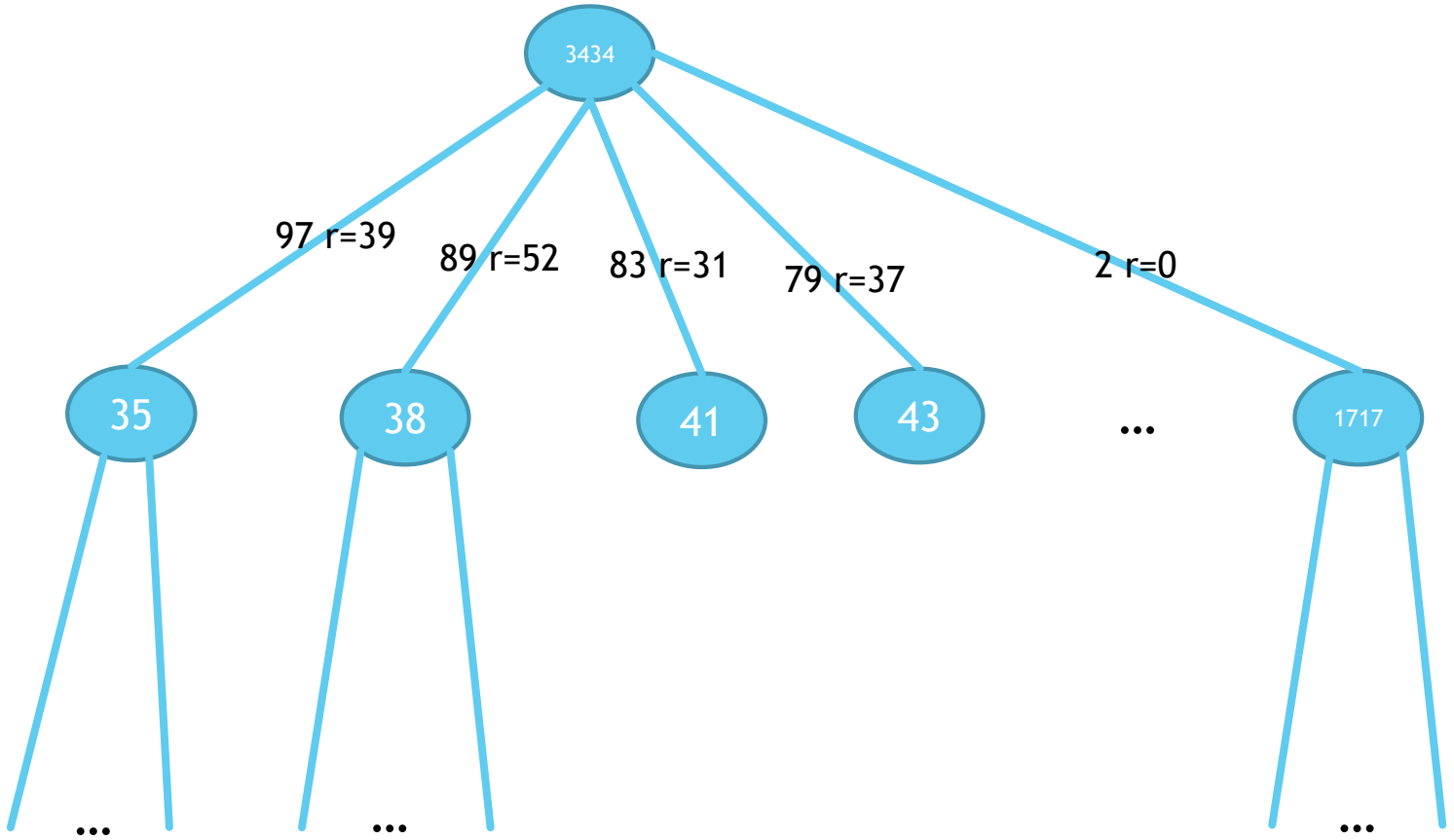
Mejoras del algoritmo

- 1) Aquí se nos ocurrió probar con todos los primos p como “base” y obtener la solución que usaba menos primos de entre todas ellas
- 2) Además, añadimos que si una solución anterior había usado menos primos de los que ya llevaba calculados para una nueva base, entonces el algoritmo debía ignorar esa base, ya que no iba a dar una solución óptima. Esto reduce nuestros tiempos de computación.
- 3) Empezamos calculando en base 97, ya que es el mayor primo, y que reduce rápidamente la magnitud del número en cuestión (Su expresión en base 97 es la que tiene menos dígitos, aunque no necesariamente menor coste, recordemos que 6 puede aparecer como dígito, pero tiene coste 2 (6=3*2))

Mejoras del algoritmo

- 4) La mejora más interesante fue la de “Combinar bases de primos”, es decir considerar soluciones del tipo:
  - Ej: 185675=17+97*(5+83*(2+7*(1+2)))
- Nuestro algoritmo seguirá el mismo método, pero comprobando todos los primos en cada iteración:
  - 1) Para todo primo p:
    - 1.1) Tomamos n y hacemos a=n%p. Esta será nuestra primera “cifra”
    - 1.2) Consideramos n’=(n-a)/p
    - 1.3) Ejecutamos de nuevo el algoritmo con n’. Al coste que nos de, le sumamos los costes de a y p (Para p será 1 claro)
  - 2) De entre todas las soluciones anteriores, elegimos la de menor coste
- A continuación presentamos un gráfico con el algoritmo:

Idea de la solución: El código Binario



Mejoras del algoritmo

- 5) Como hemos visto en el gráfico anterior, el algoritmo se presenta como un árbol. Siguiendo el algoritmo de ramificación y poda, decidimos cortar aquellas ramas que sabemos que no van a mejorar la solución actual.
- 6) Nuestro algoritmo en este punto solo utilizaba las operaciones de suma y multiplicación. La división puede que sea útil en determinados casos, pero para números muy grandes no parece que vaya a ser muy útil. La resta si que puede ser útil así que añadimos una mejora para tenerla también en cuenta:
  - 1) Para todo primo p:
    - 1.1) Tomamos n y hacemos a=n%p. Esta será nuestra primera “cifra”
    - 1.2.1) Consideramos n’=(n-a)/p
    - 1.2.2) Consideramos n’’=(n-a)/p+1
    - 1.3) Ejecutamos de nuevo el algoritmo con n’ y n’’. Al coste que nos de, le sumamos los costes de a y p (Para p será 1 claro)
  - 2) De entre todas las soluciones anteriores, elegimos la de menor coste

Mejoras del algoritmo

- 7) Utilizamos un diccionario para guardar todos los valores que vamos calculando y como para los números pequeños tiene más importancia la suma respecto a la multiplicación que en números grandes también precalculamos las sumas, las restas y las multiplicaciones de dos primos y las sumas de tres números primos. Con esto aseguramos tener todos los restos posibles del 1 al 97.
- 8) Una posible mejora que finalmente no implementamos, fue intentar aumentar nuestra base de números entre los que ir dividiendo y teniéndolos ordenados en función de $\frac{c[p]}{\log p}$ *. Esta idea al final no la utilizamos porque no mejoraba mucho la solución y aumentaba muchísimo el tiempo del algoritmo ya que pasábamos de tener 25 hijos por nodo a más de 100.

* La base del logaritmo es indiferente porque son todas proporcionales por las propiedades de los logaritmos

Mejoras del algoritmo

Las mejoras comentadas a continuación son para optimizar el tiempo sin perder eficacia en el algoritmo:

- 9) Para podar el número de hijos, en función de las cifras del número, utilizamos sublistas de primos, para números muy grandes con los mayores primos nos es suficiente porque nos interesa bajar el número rápidamente. Mientras que si el número es menor utilizamos más primos. Los parámetros que finalmente utilizamos los definimos haciendo pruebas y comparándolas con otras que estaban fuera del tiempo máximo que puede ejecutarse el programa. Los parámetros obtenidos son:
  - Entre 0 y 10^5: todos los primos
  - Entre 10^5 y 10^6: los 20 mayores primos
  - Entre 10^6 y 10^8: los 15 mayores primos
  - Mayores de 10^8: los 5 mayores primos*

* Puede parecer una lista muy pequeña pero tras ejecutar 10 test diferentes de 100 casos cada uno utilizando 5 10 primos, cambiaban las soluciones pero no los costes de estas. En cambio, la solución con 5 primos si era más rápida.

Mejoras del algoritmo

- 10) En principio dejaremos que cada caso tenga como máximo 3 segundos para ejecutarse. Como hay casos que se ejecutan más lento y otros más rápido, si el tiempo medio de los casos que se han ejecutado previamente es menor a 2 segundos por caso, permitimos que el siguiente caso pueda utilizar hasta 5 segundos. Para no hacer esta comprobación constantemente, la hacemos en el nodo raíz, después de desarrollar a cada hijo.