Fuerza Bruta 1

Miguel A. Miní

Septiembre 2023

1. Introducción al Paradigma

La búsqueda completa es un enfoque general que se utiliza para abordar una amplia gama de problemas computacionales. En esencia, consiste en **generar todas las posibles soluciones** para un problema dado y luego realizar diversas operaciones con estas soluciones, como contar elementos, encontrar una solución específica, entre otras.

Figura 1: Montessori Puzzle

Existen dos formas principales de encontrar soluciones mediante la búsqueda completa: la iterativa y la recursiva. En la implementación iterativa, se pueden emplear uno o más bucles, como uno solo, dos, tres o incluso más, o técnicas como ***two pointers***, según lo requiera el problema. En cambio, la implementación recursiva puede tomar diversas formas, desde una función recursiva simple hasta una función recursiva que explore diferentes estados a través del método de prueba y error (***backtracking***), con posibles optimizaciones (***pruning***) para evitar explorar estados que se alejen de la solución deseada.

Cuando se aborda un problema mediante la búsqueda completa, es común utilizar una combinación de estas estructuras mencionadas. Sin embargo, es fundamental comprender que cada problema es único y requiere un enfoque particular. **No existe un método universal que funcione para todos los problemas de búsqueda completa**.

Cuando se resuelve un problema de búsqueda completa de manera iterativa, a menudo se le denomina "solución por fuerza bruta". Comenzaremos explorando este paradigma, presentando diversas técnicas para abordar problemas de fuerza bruta y, posteriormente, avanzaremos hacia enfoques más sofisticados para resolver problemas de búsqueda completa.

2. Ideas principales

A pesar de que no existe un algoritmo específico, si podemos encontrar algunas técnicas útiles que se repiten constantemente.

2.1. Fijar variables

La idea en este caso, es localizar aquellas variables que tienen restricciones pequeñas y reducir el problema fijando estas variables.

Problema 2.1.1: Encuentra todas las soluciones enteras (x, y, z) de la ecuación $ax+by+cz = n$. Dados los valores enteros $1 \leq a, b, c, n \leq 5000$.

Solución: Notemos que podemos restringir las variables x, y, z, por $\lfloor \frac{n}{a} \rfloor, \lfloor \frac{n}{b} \rfloor, \lfloor \frac{n}{c} \rfloor$. Así, podemos buscar en las $\frac{n^3}{abc} + O(n^2)$ ternas, con un valor máximo $n^3 + O(n^2)$ cuando $a = b = c = 1$.

Pero, podemos darnos cuenta que dado x y y, z puede ser únicamente determinado, finalizando con un algoritmo $O(n^2)$.

Código:

```
for x in range(n//a + 1):
    for y in range((n - a*x)//b + 1):
        z = (n - a*x - b*y)//c
        if a*x + b*y + c*z == c:
            print(x, y, z)
```

Problema 2.1.2: Halle cuantos valores enteros no negativos x, cumplen que

$$x - d(x) \geq s$$

$$x \leq n$$

Donde $n, s \leq 10^{18}$ y d es la función suma de dígitos.

Solución: Note que $d([0, n]) \subset [0, 9 \times \lfloor \log 10(n) \rfloor] \subset [0, 9 \times 18]$. Por tanto x está necesariamente en el rango $[s, s + 9 \times 18]$ o x siempre cumple.

Complejidad: $O(\log(n) \log(n + s))$.

Código:

```
def sum_of_digits(m):
    res = 0
    while m > 0:
        res += m % 10
        m //= 10
    return res

cnt = max(0, n - s + 9 * 18)
for d in range(0, 9 * 18 + 1):
    x = s + d
    if x > n: break
    if x - sum_of_digits(x) >= s:
        cnt += 1
```

Problema 2.1.3: Responda todas las soluciones enteras a, b, c de la ecuación $\frac{1}{a} + \frac{1}{b} + \frac{1}{c} = \frac{1}{k}$, con $a \geq b \geq c$. Dado $k \leq 50$.

Solución: Primero, acotemos las variables. Obviamente $c > k$, de otra forma, el lado izquierdo sería mayor. Por otro lado, fácilmente podemos corroborar que:

$$\frac{1}{k} = \frac{1}{a} + \frac{1}{b} + \frac{1}{c} \leq \frac{3}{c}$$

Por lo cual

$$c \leq 3k$$

De la misma forma, tenemos:

$$\frac{1}{k} - \frac{1}{c} = \frac{1}{a} + \frac{1}{b} \leq \frac{2}{b}$$

Donde nos importa que $\frac{1}{k} - \frac{1}{c}$ sea mínimo. Tenemos (recuerde $c > k$):

$$\frac{1}{k} - \frac{1}{k+1} \leq \frac{2}{b}$$

$$b \leq 2k(k+1)$$

Así, como en el problema 2.1.1, si fijamos estas dos variables podemos deducir la tercera, si es que existe, en una complejidad $4k^3 + O(k^2)$.

Código:

```
for c in range(k + 1, 3*k + 1):
    for b in range(c, 2*k*(k + 1)):
        p = k*b*c
        q = b*c - k*(b+c)
        if q > 0 && p % q == 0:
            print(p//q, b, c)
```

2.2. Simula

Similarmente como el caso anterior, es buscar restricciones, pero en este caso no es tan sencillo, así que hay un trabajo de *hardcoding* antes de elegir el código, en este caso afrontar el problems suele ser más fácil de lo que parece.

Problema 2.2.1:
Un *humble number* es un número cuyos únicos factores primos son $2, 3, 5, 7$. Recibirás un número n y deberás imprimir el $n-$th *humble number*.

$$n \leq 5842$$

Solución:
Necesitamos calcular $S = \{2^i 3^j 5^k 7^l, 0 \leq i, j, k, l\}$. Digamos que el máximo valor que computamos es L, entonces $i, j, k, l = O(\log L)$.

Así, si fijamos L podemos obtener los elementos de S que son menores a L en $O(\log^4 L)$. Así, podemos probar distintos valores de L hasta que obtengamos más números que el máximo n que

recibimos en nuestra entrada. Ten cuidado con el ***overflow***.

Código:

```
L = int(1e10) #este valor se tiene que descubrir
arr = []
n1 = 1
while n1 <= L:
    n2 = n1
    while n2 <= L:
        n3 = n2
        while n3 <= L:
            n4 = n3
            while n4 <= L
                arr.append(n4)
                n4 *= 7
            n3 *= 5
        n2 *= 3
    n1 *= 2

print(arr[n-1])
```

2.3. Analiza por casos

Este tipo de problemas suelen parecer complicados, pero cuando analizar los casos que pueden ocurrir te das cuenta que el problema se reduce a algo mucho más simple.

Problema 2.3.1:
Te dan un array a de n elementos. Imprime dos números i, j tan que $a_i \neq a_j$, $i < j$ tal que cuando intercambiamos a_i, a_j el array no este ordenado (de forma ascendente ni descendente) o imprime -1 si tal par de números no existe.

$$1 \leq n \leq 10^5$$

Solución:
Una primera idea sería fijar i, j con dos for y verificar si las condiciones se cumplen. Esta solución es $O(n^3)$, por eso así obtendriamos un TLE. Necesitamos algo mejor.
Analicemos el problema por casos de acuerdo a la cantidad de elementos distintos que tiene el array.

- **Todos los elementos son iguales:** La respuesta es -1
- **Hay mas de dos elementos distintos:** Pruebe de que siempre hay solución eligiendo 3 elementos distintos.
- **Hay exactamente dos elementos:** Notemos que el array resultante no puede tener los elementos separados, ahora pensemos que pasaría si hay las siguientes subsecuencias (o lo que es lo mismo, lo comprimimos):
    - x, y, x, y : cambiamos el primero con el segundo (note que ellos se encontrarán consecutivos).
    - x, y, x : En este caso solo hay respuesta si hay suficientes de cada uno (al menos 4 elementos), note que siempre podemos intercambiar elementos consecutivos.

- x, y: Similarmente al caso anterior si hay al menos tres elementos, similarmente sucederá al intercambiar dos elementos consecutivos.

Solución:

```
1
2  sorted_arr = sorted(arr)
3  sorted_arr_2 = list(reversed(sorted_arr))
4  idx = [0]
5  for i in range(1, len(arr)):
6      if arr[i] != arr[i-1]:
7          idx.append(i)
8      if len(idx) == 3:
9          break
10
11 for i in range(len(idx)):
12     for j in range(i+1, len(idx)):
13         x = idx[i]
14         y = idx[j]
15         arr[x], arr[y] = arr[y], arr[x]
16         if arr != sorted_arr and arr != sorted_arr_2:
17             print(x, y)
18             exit(0)
19         arr[x], arr[y] = arr[y], arr[x]
20
21 print(-1)
```