INSTITUTO POLITÉCNICO NACIONAL

Escuela Superior de Cómputo

Análisis y diseño de algoritmos

Ejercicio 8

Gil Juárez Hector David

3CV1

Mayo 13, 2024

Algoritmo recursividad plana

La recursividad plana, también conocida como recursión de cola, implica una técnica en la que una función se llama a sí misma repetidamente hasta que se alcanza una condición de terminación. En este tipo de recursión, cada llamada recursiva se resuelve en el momento en que se realiza, y no hay llamadas pendientes a las que se deba volver cuando se complete la llamada actual.

En el código proporcionado, la función print_ways utiliza un enfoque iterativo utilizando una pila y un bucle while para emular la recursión sin realmente realizar llamadas recursivas. La pila lleva un registro de las combinaciones de pasos posibles junto con el número de escalones restantes para subir. En cada iteración del bucle, se extrae un elemento de la pila, se verifican las condiciones de terminación y se generan las combinaciones de pasos posibles para subir la escalera. El proceso continúa hasta que se agota la pila, lo que significa que se han explorado todas las combinaciones posibles.

```python
def print_ways(n):
    ways = [0] * (n + 1)
    ways[0] = 1
    ways[1] = 1
    ways[2] = 2

    for i in range(3, n + 1):
        ways[i] = ways[i - 1] + ways[i - 2] + ways[i - 3]

    print("Formas posibles de subir la escalera con", n, "peldaños:")
    stack = [(n, [])]
    while stack:
        current_n, path = stack.pop()
        if current_n == 0:
            print(path)
        else:
            if current_n >= 1:
                stack.append((current_n - 1, path + [1]))
            if current_n >= 2:
                stack.append((current_n - 2, path + [2]))
            if current_n >= 3:
                stack.append((current_n - 3, path + [3]))

# Valor fijo de n=3
n = 3
print_ways(n)
```

Algoritmo memorización

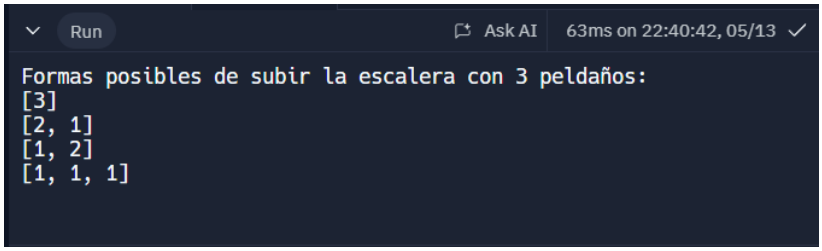
La memorización es una técnica que se utiliza para almacenar resultados de cálculos costosos en memoria, de modo que si se necesita el mismo cálculo nuevamente en el futuro, se pueda recuperar directamente desde la memoria en lugar de volver a calcularlo. Esta técnica es útil para optimizar algoritmos que tienen subproblemas superpuestos, es decir, subproblemas que se repiten en diferentes ramas de la recursión o iteración.

En el código proporcionado, la función count_ways implementa memorización. Utiliza un array memo para almacenar los resultados previamente calculados de las formas posibles de subir la escalera para un número dado de peldaños. Antes de realizar el cálculo, la función verifica si el resultado ya está presente en memo. Si es así, se devuelve directamente el valor almacenado en memo. Si no, se realiza el cálculo normalmente y se almacena el resultado en memo para su uso futuro. Esto reduce la cantidad de cálculos repetidos y mejora la eficiencia del algoritmo, especialmente para escaleras con un gran número de peldaños.

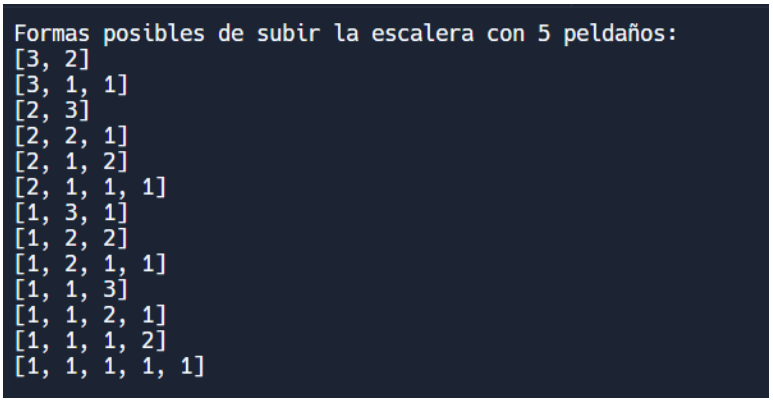
```python
def count_ways(n, memo):
    if n == 0:
        return 1
    if n < 0:
        return 0
    if memo[n] != -1:
        return memo[n]
    memo[n] = count_ways(n - 1, memo) + count_ways(n - 2, memo) + count_ways(n - 3, memo)
    return memo[n]
def print_ways(n):
    memo = [-1] * (n + 1)
    count_ways(n, memo)
    stack = [(n, [])]
    print("Formas posibles de subir la escalera con", n, "peldaños:")
    while stack:
        current_n, path = stack.pop()
        if current_n == 0:
            print(path)
        else:
            if current_n >= 1:
                stack.append((current_n - 1, path + [1]))
            if current_n >= 2:
                stack.append((current_n - 2, path + [2]))
            if current_n >= 3:
                stack.append((current_n - 3, path + [3]))
n = 3
print_ways(n)
```

CASOS PROPUESTOS

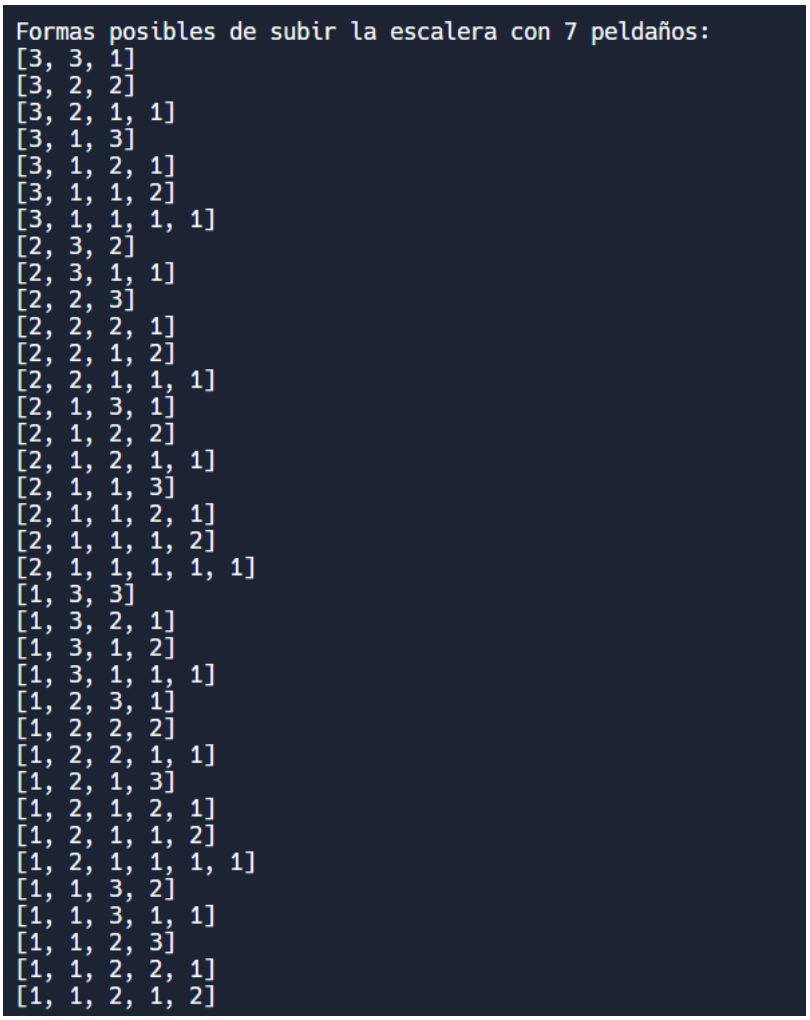
Para N=3



Para N=5



Para N=7



Para N=6

```
Formas posibles de subir la escalera con 6 peldaños:
[3, 3]
[3, 2, 1]
[3, 1, 2]
[3, 1, 1, 1]
[2, 3, 1]
[2, 2, 2]
[2, 2, 1, 1]
[2, 1, 3]
[2, 1, 2, 1]
[2, 1, 1, 2]
[2, 1, 1, 1, 1]
[1, 3, 2]
[1, 3, 1, 1]
[1, 2, 3]
[1, 2, 2, 1]
[1, 2, 1, 2]
[1, 2, 1, 1, 1]
[1, 1, 3, 1]
[1, 1, 2, 2]
[1, 1, 2, 1, 1]
[1, 1, 1, 3]
[1, 1, 1, 2, 1]
[1, 1, 1, 1, 2]
[1, 1, 1, 1, 1, 1]
```