Chapter 1: From Recursion to Dynamic Programming

```
def countChar(str, char):
  '''
  you can call helper function as countChar_(str[1:], char)
  '''
  if len(str) <= 0:
    return 0
  if str[0] == char:
    return 1 + countChar_(str[1:], char)
  else:
    return countChar_(str[1:], char)
```

Ahora, demos un paso atrás y veamos cuál era el objetivo de este ejercicio.

Hay dos cosas que se hizo en este algoritmo:

1 - Se creo un caso base cuando la cadena está vacía

2 - Se hicieron cálculos para un paso y pasaste el resto del trabajo a la llamada recursiva. Aquí el cálculo es simplemente verificar si el primer carácter de la cadena str es un carácter char o no.

Así es exactamente como se escribe un algoritmo recursivo. Primero, intenta encontrar un patrón que se repita y podría ser un paso recursivo. Luego busca los casos base, que especifican alguna condición de terminación para su algoritmo. Por último, escribe el código para el primer paso y pasa el resto del trabajo a las llamadas recursivas.

Fibonacci

```
def fib(n):
  if n == 0: # base case 1
    return 0
  if n == 1: # base case 2
    return 1
  else: # recursive step
    return fib(n-1) + fib(n-2)

print (fib(6))
```

```
8
```

Un valor cercano a 50 resultaría en un tiempo de espera en nuestra plataforma. Esto se debe a

✓ 0 s completado a las 14:34

```
def permutations(str):
  if str == "": # base case
  #Nuestro algoritmo tiene solo un caso base de cuando la cadena está vacía y devue
    return [""]
  permutes = []
  for char in str:
    #hacemos tantas llamadas recursivas a permutaciones como caracteres hay en nues
    subpermutes = permutations(str.replace(char, "", 1))    # recursive step
    #Para cada carácter, char, encontramos permutaciones de str excluyendo ese cará
    for each in subpermutes:
      permutes.append(char+each)
      #aqui se encuentra el crux del algoritmo
  return permutes

def main():
  print (permutations("abc"))

main()
```

```
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

Place N Queens on an NxN Chessboard

```
count = 0;
def isSafe(i, j, board):
  for c in range(len(board)):
    for r in range(len(board)):
      # check if i,j share row with any queen
      if board[c][r] == 'q' and i==c and j!=r:
        return False
      # check if i,j share column with any queen
      elif board[c][r] == 'q' and j==r and i!=c:
        return False
      # check if i,j share diagonal with any queen
      elif (i+j == c+r or i-j == c-r) and board[c][r] == 'q':
        return False

  return True

def nQueens(r, n, board):
```

```
      # else this is not a suitable box to place queen, and we should check for nex
      board[r][i] = '-'
  return False, board

def placeNQueens(n, board):


  return nQueens(0, n, board)[1]

def main():
  n = 10
  board = [["-" for _ in range(n)] for _ in range(n)]
  qBoard = placeNQueens(n, board)
  qBoard =  "\n".join(["".join(x) for x in qBoard])
  print (qBoard)
main()
```

```
q---------
--q-------
-----q----
-------q--
---------q
----q-----
--------q-
-q--------
---q------
------q---
```

```
count= 0

# Funcion que chequea si es seguro o no
def isSafe(board, r, c):


     # retorna falso si dos reinas se encuentran en la misma diagonal `/` izquierda
    (i, j) = (r, c)
    while i >= 0 and j < len(board):
        if board[i][j] == 'q':
            return False
```

```
    return True


def count_n_queens_solutions(board):
  # se crea un contador para encontrar las soluciones de un tablero N x N
    global count
    for r in board:
        m = (str(r).replace(',', '').replace('\'', ''))

    count= count + 1;


def nQueen(board, r):

    # si `N` reinas se colocan satisfactoriamente, cuenta el numero de soluciones
    if r == len(board):
        count_n_queens_solutions(board)
        return

    # place queen at every square in the current row `r`
    # and recur for each valid movement
    for i in range(len(board)):

        # si es seguro colocar dos reinas
        if isSafe(board, r, i):
            # coloca la reina en el espacio actual
            board[r][i] = 'q'

            # repite para la fila siguiente
            nQueen(board, r + 1)

            # retrocede y quita la reina de la casilla actual
```

> unidades cada vez más grandes.

- Top-down approach

> Esto es igual que recursividad. Se comienza en la parte superior con un problema más grande, haces una llamada recursiva a los subproblemas y esa recursividad desciende hasta llegar a tus casos base. La recursividad es esencialmente el enfoque de arriba hacia abajo. En el enfoque de programación dinámica de arriba hacia abajo, evalúa algo a medida que se necesita y luego puede almacenarlo y reutilizarlo cuando sea necesario nuevamente.

Fibonacci with dynamic programation.

```
import time
import matplotlib.pyplot as plt

calculated = {}

def fib(n):
  if n == 0: # base case 1
    return 0
```

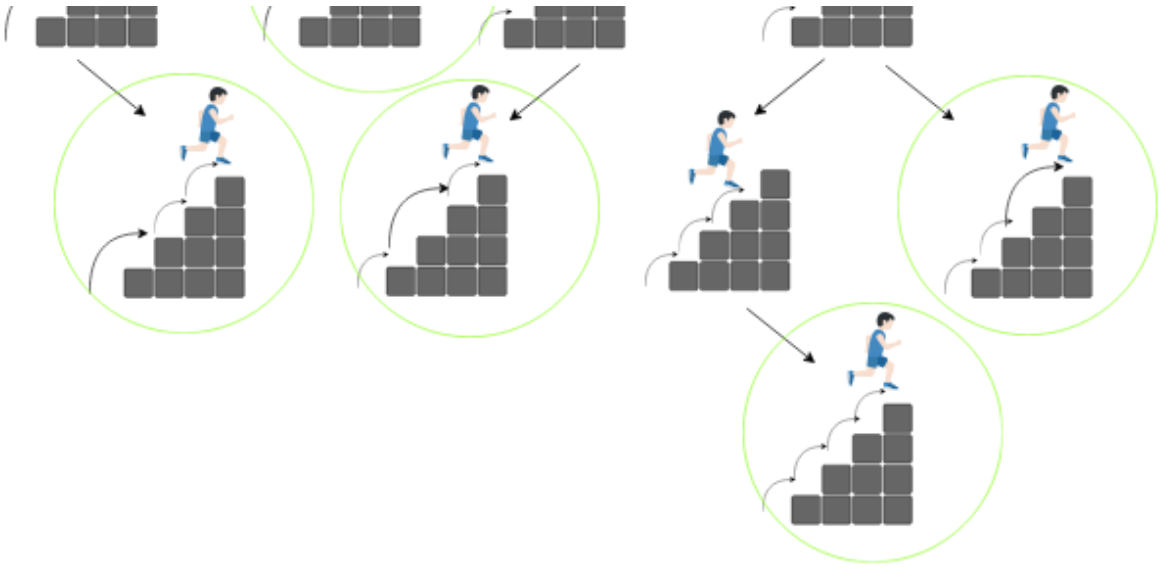
```
# Se crea un diccionario
dictionary = {}

key = 1
value = "abcd"

# Para agregar un valor a una llave

dictionary[key] = value
#or
dictionary[2] = "abc"

# las llaves y los valores pueden ser cualquier cosa,
# desde números enteros hasta cadenas y objetos personalizados

dictionary["hello"] = "hi"
dictionary[1.1] = 1

# Una clase personalizada
```

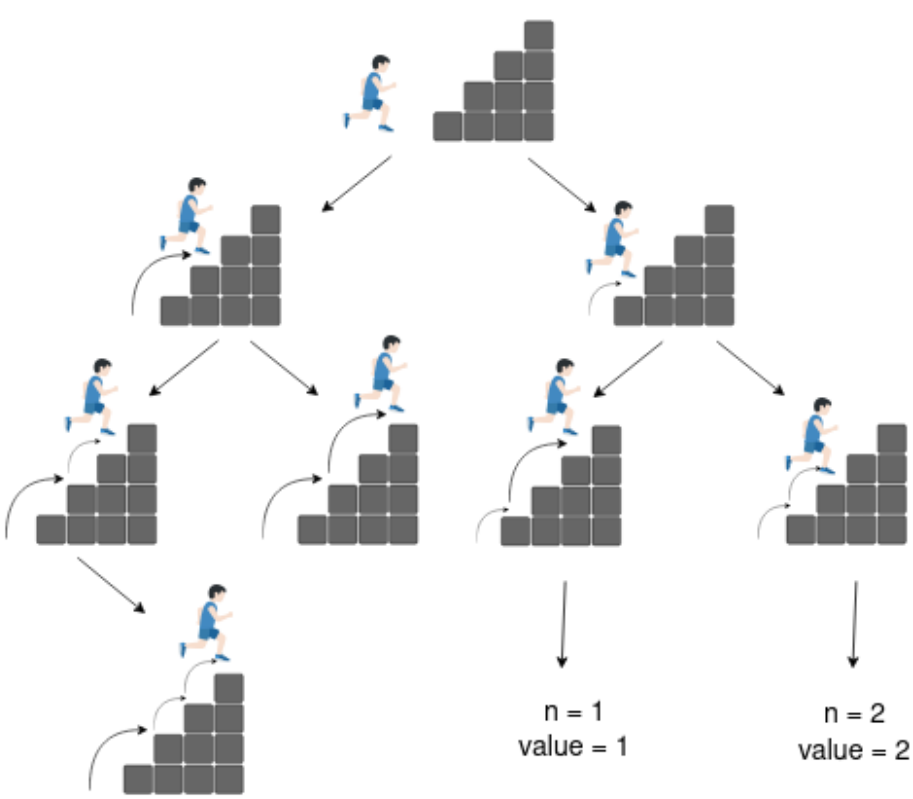
> Esto reduce la complejidad temporal de nuestro algoritmo de O(2n^n) a O(n).

The Staircase Problem

Problem Description

Task: Nick is standing next to a staircase that leads to his apartment. The staircase has n total steps; Nick knows he can climb anywhere between 1 and m steps in one jump. He thinks about





n	value
2	2
1	1

memo table