Algoritmos - Actividad Guiada 1

Nombre: Domingo Jiménez Liébana

URL:
https://colab.research.google.com/drive/10BsXYROPvWrcBs1TONGW0fVHvEDQsdJO?usp=sharing

https://github.com/domiTEN/03miar-algoritmos-optimizacion/blob/main/actividad-guiada-1/Domingo_Jimenez_Liebana_Algoritmos_AG1.ipynb

Torres de Hanoi con Divide y vencerás

```
def Torres_Hanoi(N, desde, hasta):
  if N ==1 :
    print("Lleva la ficha " ,desde , " hasta ", hasta )

  else:
    #Torres_Hanoi(N-1, desde, 6-desde-hasta )
    Torres_Hanoi(N-1, desde, 6-desde-hasta )
    print("Lleva la ficha " ,desde , " hasta ", hasta )
    #Torres_Hanoi(N-1,6-desde-hasta, hasta )
    Torres_Hanoi(N-1, 6-desde-hasta  , hasta )


Torres_Hanoi(3, 1 , 3)
```

```
Lleva la ficha  1  hasta  3
Lleva la ficha  1  hasta  2
Lleva la ficha  3  hasta  2
Lleva la ficha  1  hasta  3
Lleva la ficha  2  hasta  1
Lleva la ficha  2  hasta  3
Lleva la ficha  1  hasta  3
```

```
#Sucesión_de_Fibonacci
#https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci
#Calculo del termino n-simo de la suscesión de Fibonacci
def Fibonacci(N:int):
  if N < 2:
    return 1
  else:
    return Fibonacci(N-1)+Fibonacci(N-2)

Fibonacci(5)
```

```
8
```

Devolución de cambio por técnica voraz

```
def cambio_monedas(N, SM):
  SOLUCION = [0]*len(SM)   #SOLUCION = [0,0,0,0,..]
  ValorAcumulado = 0

  for i,valor in enumerate(SM):
    monedas =  (N-ValorAcumulado)//valor
    SOLUCION[i] = monedas
    ValorAcumulado = ValorAcumulado + monedas*valor

    if ValorAcumulado == N:
      return SOLUCION


cambio_monedas(15,[25,10,5,1])
```

```
[0, 1, 1, 0]
```

N-Reinas por técnica de vueta atrás

Este es el algoritmo al que he decidido aplicar la mejora.

Principalmente, lo que he hecho es reducir el número de llamadas a *es_prometedora* que tiene O(n^2) ya que tiene un bucle que recorre, en el peor de los casos, todas las columnas (*for i in range(etapa+1)*) dentro de otro que también las recorre todas (*for j in range(i+1, etapa +1)*).

En general, el orden de complejidad es **O(n! * n^2)** ya que la vuelta atrás es O(n!) y

el método **es_prometedora** al que llama cada vez que coloca una reina, es O(n^2).

La vuelta atrás es O(n!) porque es un bucle de O(n) y se llama a sí misma de forma recursiva, recorriendo todas las filas -1 en el pero de los casos (misma fila). Por lo que es $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n!$.

La mejora la hago en 2 fases:

- **Mejora 1** en la que obtengo las posiciones inválidas de la siguiente columna con una función de O(n) y evito llamadas a **es_prometedora**.
- **Mejora 2** en la que quito directamente las llamadas a **es_prometedora** y sólo llamo a la nueva función de O(n), por lo que la complejidad pasa de O(n! * n) a **O(n! * n)**.

Sólo aclarar que esto no es una complejidad deseada puesto que ya sabemos que si es exponencial o factorial tendremos problema en casos con valor de n muy grandes. De hecho, con 16 reinas, me ha llegado a tardar 721 minutos.

Los algoritmos se pueden ver más abajo, pero dejo una tabla comparativa de los tiempos aquí para consultarlos fácilmente.

Comparativa de algoritmos – Problema de las N Reinas

Los tiempos que muestro aquí, son en mi máquina en local. En Google Colab el tiempo es mayor.

Número de Reinas	Algoritmo	CPU	Tiempo	Llamadas a 'es_prometedora'
4	Original	46 µs	46 µs	60
	Mejora 1	60 µs	133 µs	16
	Mejora 2	38 µs	38 µs	0
8	Original	17,8 ms	18,4 ms	15.720
	Mejora 1	6,3 ms	6,46 ms	2.056
	Mejora 2	4,04 ms	4,61 ms	0
12	Original	21 s	21,1 s	10.103.868
	Mejora 1	4,06 s	4,02 s	856.188
	Mejora 2	1,33 s	1,34 s	0

Algoritmo original

En primer lugar, saco a una celda las funciones "escribe" y "es_prometedora" que no cambiarán. Tan sólo le he añadido una variable "total" para contar el número de veces que "es_prometedora" es llamada.

```
def escribe(S):
  n = len(S)
  for x in range(n):
    print("")
    for i in range(n):
      if S[i] == x+1:
        print(" X " , end="")
      else:
        print(" - ", end="")

total = 0
def es_prometedora(SOLUCION,etapa):
  global total
  total += 1
  #print(SOLUCION)
  #Si la solución tiene dos valores iguales no es valida => Dos rei
  for i in range(etapa+1):
    #print("El valor " + str(SOLUCION[i]) + " está " +  str(SOLUCIC
    if SOLUCION.count(SOLUCION[i]) > 1:
      return False

    #Verifica las diagonales
    for j in range(i+1, etapa +1 ):
      #print("Comprobando diagonal de " + str(i) + " y " + str(j))
      if abs(i-j) == abs(SOLUCION[i]-SOLUCION[j]) : return False
  return True
```

Esta es la solución original sin modificar. Tan sólo hago que imprima el tiempo que tarda en hacerlo para 12 reinas, el total de llamadas a "es_prometedora" y el total de soluciones encontradas.

Ejecutando este caso en mi máquina, ha tardado un total de **21,2 segundos** y ha hecho **10.103.868 llamadas a "es_prometedora"**.

```
def reinas(N, solucion=[], etapa=0):
  if len(solucion) == 0:
      solucion=[0 for i in range(N)]

  for i in range(1, N+1):
    solucion[etapa] = i

    if es_prometedora(solucion, etapa):
      if etapa == N-1:
        #print(solucion)
        #escribe(solucion)
        #print()
        global total_soluciones
        total_soluciones += 1
      else:
        reinas(N, solucion, etapa+1)
    else:
      None

    solucion[etapa] = 0

total_soluciones = 0
total = 0
%time reinas(12)
print("Total de llamadas a es_prometedora: ", total)
print("Total de soluciones encontradas: ", total_soluciones)
```

```
CPU times: user 47.1 s, sys: 30.4 ms, total: 47.1 s
Wall time: 48.2 s
Total de llamadas a es_prometedora:  10103868
Total de soluciones encontradas:  14200
```

Mejora 1: calcular posiciones inválidas de la columna para evitar llamadas a *es_prometedora*

Esta es la primera mejora que he hecho. Para ello, he añadido la función *get_column_invalid_positions* que calcula las posiciones inválidas para la siguiente columna basándose en la posición de las reinas ya colocadas. De esta manera, evitamos probar posiciones que ya sabemos que no funcionarán, reduciendo así el número de llamadas a "es_prometedora".

Ejecutando este caso en mi máquina, ha tardado un total de **4,02 segundos (16,08 segundos menos que el original)** y ha hecho **856.188 llamadas a "es_prometedora" (9.247.680 menos que el original)**.

```
def get_column_invalid_positions(SOLUCION, etapa):
  invalid_positions = set()
  solucion_length = len(SOLUCION)

  for i in range(etapa + 1):
    row = SOLUCION[i]
    invalid_positions.add(row)

    diff = etapa - i
    diag_up = row - diff
    diag_down = row + diff

    if 1 <= diag_up <= solucion_length:
      invalid_positions.add(diag_up)
    if 1 <= diag_down <= solucion_length:
      invalid_positions.add(diag_down)

  return invalid_positions
```

```
def reinas(N, solucion=[], etapa=0, invalid_positions=set()):
  if len(solucion) == 0:
      solucion=[0 for i in range(N)]

  for i in range(1, N+1):
    if i in invalid_positions:
      continue
    solucion[etapa] = i

    if es_prometedora(solucion, etapa):
      if etapa == N-1:
        #print("Solución encontrada:")
        #print(solucion)
        #escribe(solucion)
        #print()
        global total_soluciones
        total_soluciones += 1
      else:
        invalid_positions_next = get_column_invalid_positions(soluc
        reinas(N, solucion, etapa+1, invalid_positions_next)
    else:
      None

    solucion[etapa] = 0

total_soluciones = 0
total = 0
%time reinas(12)
print("Total de llamadas a es_prometedora: ", total)
print("Total de soluciones encontradas: ", total_soluciones)
```

```
CPU times: user 7.77 s, sys: 4.03 ms, total: 7.78 s
Wall time: 7.79 s
Total de llamadas a es_prometedora:  856188
Total de soluciones encontradas:  14200
```

Mejora 2: eliminación de la función *es_prometedora*

En este caso, tan sólo he modificado la función "reinas" para que directamente, no se realice ninguna llamada a "es_prometedora" ya que cada vez que avanzamos una columna, sólo probamos a poner las reinas en posiciones que son correctas. Por tanto, al posicionar una reina en la última columna, quiere decir que hemos encontrado una solución correcta.

Ejecutando este caso en mi máquina, ha tardado un total de **1,33 segundos (19,87 segundos menos que el original)** y ha hecho **0 llamadas a "es_prometedora"**.

```
def reinas(N, solucion=[], etapa=0, invalid_positions=set()):
  if len(solucion) == 0:
      solucion=[0 for i in range(N)]

  for i in range(1, N+1):
    if i in invalid_positions:
      continue
    solucion[etapa] = i

    if etapa == N-1:
        global total_soluciones
        total_soluciones += 1
        #print("Solución encontrada:")
        #print(solucion)
        #escribe(solucion)
        #print()
    else:
        invalid_positions_next = get_column_invalid_positions(soluc
        reinas(N, solucion, etapa+1, invalid_positions_next)

    solucion[etapa] = 0

total_soluciones = 0
total = 0
%time reinas(12)
print("Total de llamadas a es_prometedora: ", total)
print("Total de soluciones encontradas: ", total_soluciones)
```

```
CPU times: user 2.21 s, sys: 1.93 ms, total: 2.21 s
Wall time: 2.22 s
Total de llamadas a es_prometedora:  0
Total de soluciones encontradas:  14200
```

Viaje por el rio. Programación dinámica

```
TARIFAS = [
[0,5,4,3,999,999,999],
[999,0,999,2,3,999,11],
[999,999, 0,1,999,4,10],
[999,999,999, 0,5,6,9],
[999,999, 999,999,0,999,4],
[999,999, 999,999,999,0,3],
[999,999,999,999,999,999,0]
]



##################################################################
def Precios(TARIFAS):
##################################################################
  #Total de Nodos
  N = len(TARIFAS[0])

  #Inicialización de la tabla de precios
  PRECIOS = [ [9999]*N for i in [9999]*N]
  RUTA = [ [""]*N for i in [""]*N]

  for i in range(0,N-1):
    RUTA[i][i] = i             #Para ir de i a i se "pasa por i"
    PRECIOS[i][i] = 0          #Para ir de i a i se se paga 0
    for j in range(i+1, N):
      MIN = TARIFAS[i][j]
      RUTA[i][j] = i

      for k in range(i, j):
        if PRECIOS[i][k] + TARIFAS[k][j] < MIN:
            MIN = min(MIN, PRECIOS[i][k] + TARIFAS[k][j] )
            RUTA[i][j] = k          #Anota que para ir de i a j hay
        PRECIOS[i][j] = MIN

  return PRECIOS,RUTA
##################################################################



PRECIOS,RUTA = Precios(TARIFAS)
#print(PRECIOS[0][6])

print("PRECIOS")
for i in range(len(TARIFAS)):
```

```
  print(PRECIOS[i])

print("\nRUTA")
for i in range(len(TARIFAS)):
  print(RUTA[i])

#Determinar la ruta con Recursividad
def calcular_ruta(RUTA, desde, hasta):
  if desde == hasta:
    #print("Ir a :" + str(desde))
    return ""
  else:
    return str(calcular_ruta( RUTA, desde, RUTA[desde][hasta])) +  \
                ',' + \
                str(RUTA[desde][hasta] \
              )

print("\nLa ruta es:")
calcular_ruta(RUTA, 0,6)
```

```
PRECIOS
[0, 5, 4, 3, 8, 8, 11]
[9999, 0, 999, 2, 3, 8, 7]
[9999, 9999, 0, 1, 6, 4, 7]
[9999, 9999, 9999, 0, 5, 6, 9]
[9999, 9999, 9999, 9999, 0, 999, 4]
[9999, 9999, 9999, 9999, 9999, 0, 3]
[9999, 9999, 9999, 9999, 9999, 9999, 9999]

RUTA
[0, 0, 0, 0, 1, 2, 5]
['', 1, 1, 1, 1, 3, 4]
['', '', 2, 2, 3, 2, 5]
['', '', '', 3, 3, 3, 3]
['', '', '', '', 4, 4, 4]
['', '', '', '', '', 5, 5]
['', '', '', '', '', '', '']

La ruta es:
',0,2,5'
```