Algoritmos - Actividad Guiada 1

Nombre : Juan Carlos Marin Mejia

URL: https://colab.research.google.com/drive/1h4ZR92lBIYmIoTCQJblhCPvuuj-iI2n0?usp=sharing

GitHub: https://github.com/jcmm518/03MAIR-Algoritmos-de-optimizacion.git

Torres de Hanoi con divide y vencerás

```
def Torres_Hanoi(N,desde,hasta):
  if N==1 :
    print("Lleva la ficha desde :" , desde," Hasta: ", hasta)
  else :
    Torres_Hanoi(N-1,desde,6-desde-hasta)
    print("Lleva la ficha desde :" , desde," Hasta: ", hasta)
    Torres_Hanoi(N-1,6-desde-hasta,hasta)

Torres_Hanoi(3,1,3)
```

```
Lleva la ficha desde : 1  Hasta:  3
Lleva la ficha desde : 1  Hasta:  2
Lleva la ficha desde : 3  Hasta:  2
Lleva la ficha desde : 1  Hasta:  3
Lleva la ficha desde : 2  Hasta:  1
Lleva la ficha desde : 2  Hasta:  3
Lleva la ficha desde : 1  Hasta:  3
```

Devolución de cambio por técnica voraz

```
# Se implementa algoritmo con control de faltante de moneda para completar el total del cambi
def cambio_monedas(N,SM):
  SOLUCION = [0] * len(SM)
  ValorAcumulado = 0

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

      if ValorAcumulado == N :
          print('El numero de monedas necesarias es: ' + str(SOLUCION))
```

```
  if ValorAcumulado < N :
      print("No se tiene una moneda de menor valor que cubra lo faltante, por lo tanto la sol
      print(str(SOLUCION) + ' y ' + str(N-ValorAcumulado) + ' unidades de valor')

cambio_monedas(137,[25,10,5,1])
```

```
    El numero de monedas necesarias es: [5, 1, 0, 2]
```

N- Reinas por técnica de vuelta atrás

```
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="")


def es_prometedora(SOLUCION,etapa):
  #print(SOLUCION)
  #Si la solución tiene dos valores iguales no es valida => Dos reinas en la misma fila

  for i in range(etapa+1):
    #print("El valor " + str(SOLUCION[i]) + " está " +  str(SOLUCION.count(SOLUCION[i])) + "
    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


def reinas(N, solucion=[], etapa=0):
#N: dimension del tablero
  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()
        else :
          reinas(N, solucion, etapa +1)
      else :
        None

      solucion[etapa] = 0


  reinas(6)
```

```
       [2, 4, 6, 1, 3, 5]

        -  -  -  X  -  -
        X  -  -  -  -  -
        -  -  -  -  X  -
        -  X  -  -  -  -
        -  -  -  -  -  X
        -  -  X  -  -  -
       [3, 6, 2, 5, 1, 4]

        -  -  -  -  X  -
        -  -  X  -  -  -
        X  -  -  -  -  -
        -  -  -  -  -  X
        -  -  -  X  -  -
        -  X  -  -  -  -
       [4, 1, 5, 2, 6, 3]

        -  X  -  -  -  -
        -  -  -  X  -  -
        -  -  -  -  -  X
        X  -  -  -  -  -
        -  -  X  -  -  -
        -  -  -  -  X  -
       [5, 3, 1, 6, 4, 2]

        -  -  X  -  -  -
        -  -  -  -  -  X
        -  X  -  -  -  -
        -  -  -  -  X  -
        X  -  -  -  -  -
        -  -  -  X  -  -
```

Encontrar los dos puntos más cercanos

Fuerza bruta

- La forma más sencilla de resolver este problema es utilizando la fuerza bruta. Sin embargo, al aplicar este método lo que estamos haciendo es comparar las distancias entre n(n-1)/2 pares

de puntos, por lo que la complejidad que obtenemos es de O(n^2).

```
import math
import timeit
# funcion para encontrar la distancia entre dos puntos
def dist(p0, p1):
    return math.sqrt(((p0[0] - p1[0])**2)) + ((p0[1] - p1[1])**2)

# Funcion para encontrar los dos puntos mas cercanos por fuerza bruta
def bruteForce(P):
    n=len(P)
    min_val = float('inf')
    for i in range(n):
        for j in range(i + 1, n):
            if dist(P[i], P[j]) < min_val:
                min_val = dist(P[i], P[j])
                puntos_cercanos = (P[i], P[j])
    return puntos_cercanos

P = [(2, 3), (12, 30), (40, 50), (10, 49),  (12, 31), (3, 4),(12, 2),(14, 8),(13, 6)]

hora_inicio = timeit.default_timer ()
print('Los dos puntos mas cercanos son: ')
print(bruteForce(P))
print('\n')
print ("Tiempo que tarda la ejecucion al calcular la distancia entre los puntos es de : %s se
```

```
Los dos puntos mas cercanos son:
((12, 30), (12, 31))


Tiempo que tarda la ejecucion al calcular la distancia entre los puntos es de : 0.00033
```

Divide y venceras

- El problema se puede mejorar y obtener una complejidad de O(n logn^2 n) usando el método recursivo de Divide y Vencerás, pero si se ejecuta el sort desde el inicio y no de una forma secuencial, podemos obetener un O (n logn n), como lo desarrollamos a continuacion

```
import math
import timeit

# función para encontrar la distancia entre dos puntos
def lenght(a,b):
  return (math.sqrt( ((a[0]-b[0])**2)+ ((a[1]-b[1])**2)) )

# función para encontrar los dos puntos mas cercanos por fuerza bruta
def bruteForce(a,p,q,m):
  # Si solo se tienen dos puntos
```

```
    if len(a)==2:
      dist=lenght(a[0],a[1])
      if dist<m:
        p,q=a[0],a[1]
        m=dist
    # Si se tienen mas de dos puntos
    else:
      for x in range(0,len(a)-1):
        for y in range (x+1, len(a)):
          if m>lenght(a[x],a[y]):
            m=lenght(a[x],a[y])
            p,q = a[x],a[y]
    return p,q,m

  # función para buscar los puntos mas cercanos
  def search_pair(a,p,q,m):
    l=len(a)
    if l<=(3):
      return (bruteForce(a,p,q,m)) # Si solo nos quedan tres o menos puntos, utilizamos fuerza
    # Encontrar el medio del array, la posición x de este punto medio y dividir la matriz en do
    midl=l//2
    midx=a[midl][0]
    L=a[:midl]
    R=a[midl:]

    # Busqueda de recurrencia en el lado izquierdo, lado derecho y finalmente en la franja
    p,q,m=search_pair(L,p,q,m)
    p,q,m=search_pair(R,p,q,m)
    p,q,m=get_strip(a,midx,p,q,m)

    # retorna puntos y distancia
    return (p,q,m)


  def get_strip(a,xcoordinate,p,q,m):
    strip = []
    right,left = xcoordinate+int(m), xcoordinate-int(m)
    for x in a:
      if x[0]>right: break
      elif left<=x[0]<=right: strip.append(x)
    # Se organiza de acuerdo a las coordenadas
    strip.sort(key=lambda x:x[1])
    # Comprobando distancias y compárandlas con la más corta. necesitamos verificar solo 6 punt
    for x in range (0,len(strip)):
      for y in range (x+1, len(strip)):
        dist= lenght(strip[x],strip[y])
        if dist<m:
          p,q=strip[x],strip[y]
          m=dist
    return (p,q,m)
```

```
# Programa principal
hora_inicio = timeit.default_timer ()
points = [(2, 3), (12, 30), (40, 50), (10, 49),  (12, 31), (3, 4),(12, 2),(14, 8),(13, 6)]
points.sort() # Organización de los puntos
p,q = points[0],points[1] #Tomamos los puntos iniciales p y q
m=lenght (points[0],points[1]) #Asumimos como menor distancia
pt,pt2,distance=search_pair(points,p,q,m) #Encontramos los puntos más cercanos
#print ('Distancia entre los puntos más cercanos : '+ str(distance) + '\n')
print('Puntos mas cercanos : ' + str(pt) + ' ' + str(pt2))
print ("Tiempo que tarda la ejecucion al calcular la distancia entre los puntos es de : %s se
```

```
Puntos mas cercanos : (12, 30) (12, 31)
Tiempo que tarda la ejecucion al calcular la distancia entre los puntos es de : 0.00033
```