Actividad Guiada 1 de Algoritmos de Optimización

Nombre: Juan Aroca Pérez

https://colab.research.google.com/drive/1g7PcMuMEP3_NMZzD0COvnLLH0FWt6WXQ?usp=sharing

https://github.com/JuanArocaMIAR/03MIAR---Algoritmos-de-Optimizacion---2023/tree/1b144f95cd0925ff70f2559da876bbfbe6a0ad80/AG1

```
#Torres de Hanoi - Divide y venceras
###############################################################################


#####################################################################
def Torres_Hanoi(N, desde, hasta):
  #N - Nº de fichas
  #desde - torre inicial+
  #hasta - torre fina
  if N==1 :
    print("Lleva la ficha desde " + str(desde) + " hasta " + str(hasta)) #Si solo se tiene una ficha el probl

  else:
    Torres_Hanoi(N-1, desde, 6-desde-hasta) #Se divide el problema en dos, primero mover N-1 fichas desde al
    # 6-desde-hasta calcula el pivote (1+2+3=6)
    print("Lleva la ficha desde " + str(desde) + " hasta " + str(hasta))
    Torres_Hanoi(N-1, 6-desde-hasta,  hasta) #Segundo mover N-1 fichas de la pivote al final
    # esto se repite por recursividad llamando a la función hasta que solo quede una ficha y se mueva de una

Torres_Hanoi(3, 1, 3)
#####################################################################


#Cambio de monedas - Técnica voraz
###############################################################################
SISTEMA = [25, 10, 5, 1] #Ordenado de mayor a menor siempre
#####################################################################
def cambio_monedas(CANTIDAD,SISTEMA):
#....
  SOLUCION = [0]*len(SISTEMA)
  ValorAcumulado = 0

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

    if CANTIDAD == ValorAcumulado:
      return SOLUCION

  print("No es posible encontrar solucion")

cambio_monedas(42, SISTEMA)
#####################################################################
```

```
#N Reinas - Vuelta Atrás()
################################################################################

#Verifica que en la solución parcial no hay amenzas entre reinas
#####################################################################
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])) + " veces")
    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

#Traduce la solución al tablero
#####################################################################
def escribe_solucion(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="")

#Proceso principal de N-Reinas
#####################################################################
def reinas(N, solucion=[], etapa=0):
#####################################################################
### ....
  if len(solucion) == 0:         # [0,0,0...] hacemos una lista del mismo tamaño que el numero de reinas
    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)
      else:
        reinas(N, solucion, etapa+1)
    else:
      None

  solucion[etapa] = 0

reinas(8,solucion=[],etapa=0)


escribe_solucion([6, 4, 2, 8, 5, 7, 1, 3])
```

```
#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]
]

#999 se puede sustituir por float("inf"), nos indica que es imposible ir desde el nodo actual a ese punto


#Calculo de la matriz de PRECIOS y RUTAS
#####################################################################
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 que pasar por k
        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'
```

Problema: Encontrar los dos puntos más cercanos

```
import random
import math
```

Para puntos de 1D puede resolverse por "FUERZA BRUTA" ordenando todos los puntos y viendo la distancia que hay entre cada par.

```
# 1D
################################################################################
def par_mas_cercano(puntos):
    # Verificar que hay al menos dos puntos
    if len(puntos) < 2:
        return None

    # En primer lugar se ordenan los puntos de la lista de menor a mayor
    puntos_ordenados = sorted(puntos)

    # Utilizar el primer par de puntos como punto de referencia
    par_cercano = puntos_ordenados[0], puntos_ordenados[1]
    distancia_minima = puntos_ordenados[1] - puntos_ordenados[0]

    # Iterar sobre los puntos para encontrar el par más cercano
    for i in range(2, len(puntos_ordenados)):
        distancia_actual = puntos_ordenados[i] - puntos_ordenados[i - 1]

        # Actualizar el par más cercano si encontramos una distancia menor
        if distancia_actual < distancia_minima:
            distancia_minima = distancia_actual
            par_cercano = (puntos_ordenados[i - 1], puntos_ordenados[i])

    return par_cercano, distancia_minima
################################################################################

lista_1D = [random.randrange(1, 10000) for i in range(10)]
resultado = par_mas_cercano(lista_1D)
print("Par más cercano:" + str(resultado[0]))
print('Se encuentra a una distancia de: ' + str(resultado[1]) + ' unidades')

     Par más cercano:(6943, 6994)
     Se encuentra a una distancia de: 51 unidades
```

Para puntos de 2D puede resolverse mediante la técnica "DIVIDE Y VENCERÁS" ordenando todos los puntos y viendo la distancia que hay entre cada par.

```
# 2D
################################################################################
def par_mas_cercano_2D(lista_2D):
    # Verificar que hay al menos dos puntos
    if len(lista_2D) < 2:
        return None

    # Ordenar los puntos por coordenada x
    puntos_ordenados = sorted(lista_2D)

    # Función auxiliar para calcular la distancia entre dos puntos
    def distancia_2D(p1, p2):
        return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

    # Función principal para encontrar los puntos más cercanos en dos dimensiones
    def par_mas_cercano_recursivo(lista_2D):
        n = len(lista_2D)

        # Caso base: si hay pocos puntos, resuelve directamente, para salir de la recursividad
        if n <= 3:
            return min([(lista_2D[i], lista_2D[j], distancia_2D(lista_2D[i], lista_2D[j])) for i in range(n)

        # Dividir los puntos en dos mitades
        mid = n // 2
        mitad_izquierda = lista_2D[:mid]
        mitad_derecha = lista_2D[mid:]

        # Recursivamente encontrar los puntos más cercanos en cada mitad
        izquierda_cercano = par_mas_cercano_recursivo(mitad_izquierda)
        derecha_cercano = par_mas_cercano_recursivo(mitad_derecha)

        # Encontrar el par más cercano entre las dos mitades
        distancia_minima = min(izquierda_cercano[2], derecha_cercano[2])
        franja_central = [punto for punto in lista_2D if abs(punto[0] - puntos_ordenados[mid][0]) < distanc
        cercano_franja = par_mas_cercano_franja(franja_central, distancia_minima)

        # Devolver el par más cercano entre los tres conjuntos
        return min(filter(None, [izquierda_cercano, derecha_cercano, cercano_franja]), key=lambda x: x[2])

    # Función auxiliar para encontrar el par más cercano en la franja
    def par_mas_cercano_franja(franja, distancia_minima):
        n = len(franja)
        par_minimo = None

        # Ordenar la franja por coordenada y
        franja_ordenada = sorted(franja, key=lambda p: p[1])

        # Iterar sobre los puntos en la franja
        for i in range(n):
            j = i + 1
            while j < n and franja_ordenada[j][1] - franja_ordenada[i][1] < distancia_minima:
                distancia = distancia_2D(franja_ordenada[i], franja_ordenada[j])
                if distancia < distancia_minima:
                    distancia_minima = distancia
                    par_minimo = (franja_ordenada[i], franja_ordenada[j], distancia_minima)
                j += 1

        return par_minimo

    return par_mas_cercano_recursivo(puntos_ordenados)
```

```
################################################################################
# 3D
################################################################################
import math
import random

def par_mas_cercano_3D(puntos):
    # Verificar que hay al menos dos puntos
    if len(puntos) < 2:
        return None

    # Ordenar los puntos en cada dimensión
    puntos_ordenados_x = sorted(puntos, key=lambda x: x[0])
    puntos_ordenados_y = sorted(puntos, key=lambda x: x[1])
    puntos_ordenados_z = sorted(puntos, key=lambda x: x[2])

    # Función para calcular la distancia entre dos puntos 3D
    def distancia(punto1, punto2):
        return math.sqrt((punto1[0] - punto2[0])**2 + (punto1[1] - punto2[1])**2 + (punto1[2] - punto2[2])**2)

    # Función para encontrar los dos puntos más cercanos en una franja
    def par_mas_cercano_en_franja(franja, distancia_minima):
        n = len(franja)
        for i in range(n):
            j = i + 1
            while j < n and (franja[j][1] - franja[i][1]) < distancia_minima:
                distancia_minima = min(distancia_minima, distancia(franja[i], franja[j]))
                j += 1
        return distancia_minima

    # Función principal de divide y vencerás
    def par_mas_cercano_recursivo(px, py, pz):
        n = len(px)

        # Caso base: si hay pocos puntos, usa fuerza bruta
        if n <= 3:
            return min(distancia(px[i], px[j]) for i in range(n) for j in range(i + 1, n))

        # Dividir los puntos en dos mitades
        medio = n // 2
        qx = px[:medio]
        qy = sorted(px[medio:], key=lambda x: x[1])
        qz = sorted(px[medio:], key=lambda x: x[2])

        # Puntos en la franja central
        franja = [punto for punto in py if abs(punto[0] - px[medio][0]) < distancia_minima]

        # Encontrar los dos puntos más cercanos en cada mitad
        delta_izquierdo = par_mas_cercano_recursivo(qx, qy, qz)
        delta_derecho = par_mas_cercano_recursivo(px[medio:], py, pz)
```