Guillermo Rios Gómez: https://colab.research.google.com/drive/1o2hXZStq1Fj_SdbR6cYrcgxXi6buKJXG?usp=sharing Github: https://github.com/GuiRiGo88/03MIAR---Algoritmos-de-Optimizacion---2023.git

```
#Torres de Hanoi - Divide y venceras

def Torres_Hanoi(N, desde, hasta):
  """
  Esta función resuelve el problema de las Torres de Hanoi utilizando recursividad.
  N - Nº de fichas
  desde - torre inicial
  hasta - torre final
  """
  if N==1 :
    print(f"Lleva la ficha desde {desde} hasta {hasta}")
  else:
    Torres_Hanoi(N-1, desde, 6-desde-hasta)
    print(f"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
```

```
#Cambio de monedas - Técnica voraz

SISTEMA = [12, 5 ,2, 1]

def cambio_monedas(CANTIDAD, SISTEMA):
  """
  Esta función resuelve el problema de cambio de monedas utilizando la técnica voraz.
  CANTIDAD - La cantidad de dinero que queremos cambiar.
  SISTEMA - El sistema de monedas disponible.
  """
  SOLUCION = [0]*len(SISTEMA)
  ValorAcumulado = 0

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

```
    valorAcumulado = valorAcumulado + monedas * valor

    if CANTIDAD == ValorAcumulado:
      return SOLUCION

  raise ValueError("No es posible encontrar solucion")

try:
  print(cambio_monedas(15, SISTEMA))
except ValueError as e:
  print(e)
```

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

```
#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]))
    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):
```

```
def reinas(N, solucion=[],etapa=0):
  if len(solucion) == 0:         # [0,0,0...]
    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)
      else:
        reinas(N, solucion, etapa+1)
    else:
      None

  solucion[etapa] = 0

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

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

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

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

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

```
#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")
#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

Dado un conjunto de puntos se trata de encontrar los dos puntos más cercanos

FUERZA BRUTA

```
import random
import math
import time

# Generamos la lista de 1000 números aleatorios
LISTA_1D = [random.randrange(1, 1000) for x in range(1000)]

def dist_cercanos(lista):
    dist_min = math.inf  # Inicializamos con infinito
    punt_cerc = None

    # Comparamos cada par de puntos en la lista
    for i in range(len(lista)):
        for j in range(i + 1, len(lista)):
            distancia1 = abs(lista[i] - lista[j])
            if distancia1 < dist_min:
                dist_min = distancia1
                punt_cerc = (lista[i], lista[j])

    return punt_cerc, dist_min

# Medimos el tiempo de inicio
inicio = time.time()

# Llamamos a la función y mostramos el resultado
puntos, distancia1 = dist_cercanos(LISTA_1D)
```

```
# Medimos el tiempo de finalización
final = time.time()

# Calculamos la duración
duracion = final - inicio

print(f"Los dos puntos más cercanos son: {puntos} con una distancia de {distancia1}")
print(f"Tiempo de ejecución: {duracion} segundos")
```

```
Los dos puntos más cercanos son: (558, 558) con una distancia de 0
Tiempo de ejecución: 0.16610503196716309 segundos
```

- Calcular la complejidad. ¿Se puede mejorar? El algoritmo tiene una complejidad de tiempo de O(n^2). Esto se debe a que hay dos bucles anidados que comparan cada elemento de la lista LISTA_1D con todos los demás elementos. Si se puede mejorar aplicando otras técnicas de optimización

Segundo intento. Aplicar Divide y Vencerás

```
def cercanos(lista):
    # Ordenamos la lista
    ordenada = sorted(lista)

    # Función recursiva para encontrar la distancia mínima
    def dist_min_recur(inicio, fin):
        # Si hay 2 o 3 puntos, se resuelve directamente
        if fin - inicio <= 3:
            return min(abs(ordenada[i] - ordenada[j])
                       for i in range(inicio, fin)
                       for j in range(i + 1, fin + 1))

        # Dividimos la lista en dos mitades
        mitad = (inicio + fin) // 2
        punto_medio = ordenada[mitad]

        # Encontramos la distancia mínima en ambas mitades
        dist_izq = dist_min_recur(inicio, mitad)
        dist_der = dist_min_recur(mitad + 1, fin)

        # Encontramos la distancia mínima entre las dos mitades
        dist_min = min(dist_izq, dist_der)

        # Consideramos los puntos cercanos al punto medio
        puntos_cercanos = [p for p in ordenada[inicio:fin + 1]
                           if abs(p - punto_medio) < dist_min]
```

```
        # Encontramos la distancia mínima entre los puntos cercanos
        dist_min_cercanos = min((abs(puntos_cercanos[i] - puntos_cercanos[j])
                                 for i in range(len(puntos_cercanos))
                                 for j in range(i + 1, len(puntos_cercanos))),
                                default=dist_min)

        return min(dist_min, dist_min_cercanos)

    return dist_min_recur(0, len(ordenada) - 1)

# Medimos el tiempo de inicio
inicio = time.time()

# Llamamos a la función y mostramos el resultado
distancia = cercanos(LISTA_1D)

# Medimos el tiempo de finalización
final = time.time()

# Calculamos la duración
duracion = final - inicio

print(f"La distancia más corta entre dos puntos es: {distancia}")
print(f"Tiempo de ejecución: {duracion} segundos")
```

```
La distancia más corta entre dos puntos es: 0
Tiempo de ejecución: 0.009646892547607422 segundos
```

- Calcular la complejidad. ¿Se puede mejorar? La complejidad de tiempo de este algoritmo es O(n log n), lo que lo hace mucho más eficiente que el enfoque de fuerza bruta para listas grandes

Extender el algoritmo a 2D:

```
# Generamos una lista de 1000 puntos aleatorios en 2D
LISTA_2D = [(random.randrange(1, 1000), random.randrange(1, 1000)) for _ in range(1000)]

def distancia_euclidiana(punto1, punto2):
    return math.sqrt((punto1[0] - punto2[0])**2 + (punto1[1] - punto2[1])**2)

def encontrar_puntos_mas_cercanos_2D(lista):
    # Ordenamos la lista por la coordenada x
    lista_ordenada = sorted(lista, key=lambda punto: punto[0])

    # Función recursiva para encontrar la distancia mínima
    def distancia_minima_recursiva(inicio, fin):
```

```
    def distancia_minima_recursiva(inicio, fin):
        if fin - inicio <= 3:
            return min(distancia_euclidiana(lista_ordenada[i], lista_ordenada[j])
                       for i in range(inicio, fin)
                       for j in range(i + 1, fin + 1))

        mitad = (inicio + fin) // 2
        punto_medio = lista_ordenada[mitad]

        dist_izq = distancia_minima_recursiva(inicio, mitad)
        dist_der = distancia_minima_recursiva(mitad + 1, fin)

        dist_min = min(dist_izq, dist_der)

        # Consideramos los puntos cercanos al punto medio en ambas coordenadas
        franja = [p for p in lista_ordenada[inicio:fin + 1]
                  if abs(p[0] - punto_medio[0]) < dist_min]

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

        # Comparamos los puntos dentro de la franja
        for i in range(len(franja_ordenada)):
            for j in range(i + 1, min(i + 7, len(franja_ordenada))):
                dist_puntos_franja = distancia_euclidiana(franja_ordenada[i], franja_orde
                dist_min = min(dist_min, dist_puntos_franja)

        return dist_min

    return distancia_minima_recursiva(0, len(lista_ordenada) - 1)

# Llamamos a la función y mostramos el resultado
distancia = encontrar_puntos_mas_cercanos_2D(LISTA_2D)
print(f"La distancia más corta entre dos puntos en 2D es: {distancia}")
```

```
La distancia más corta entre dos puntos en 2D es: 1.0
```

Extender el algoritmo a 3D

```
# Generamos una lista de 1000 puntos aleatorios en 3D
LISTA_3D = [(random.randrange(1, 1000), random.randrange(1, 1000), random.randrange(1, 100

def distancia_euclidiana_3D(punto1, punto2):
    return math.sqrt((punto1[0] - punto2[0])**2 + (punto1[1] - punto2[1])**2 + (punto1[2] -

def encontrar_puntos_mas_cercanos_3D(lista):
    # Ordenamos la lista por la coordenada x
    lista_ordenada = sorted(lista, key=lambda punto: punto[0])
```

```
    # Función recursiva para encontrar la distancia mínima
    def distancia_minima_recursiva(inicio, fin):
        if fin - inicio <= 3:
            return min(distancia_euclidiana_3D(lista_ordenada[i], lista_ordenada[j])
                       for i in range(inicio, fin)
                       for j in range(i + 1, fin + 1))

        mitad = (inicio + fin) // 2
        punto_medio = lista_ordenada[mitad]

        dist_izq = distancia_minima_recursiva(inicio, mitad)
        dist_der = distancia_minima_recursiva(mitad + 1, fin)

        dist_min = min(dist_izq, dist_der)

        # Consideramos los puntos cercanos al punto medio en las tres coordenadas
        franja = [p for p in lista_ordenada[inicio:fin + 1]
                  if abs(p[0] - punto_medio[0]) < dist_min]

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

        # Comparamos los puntos dentro de la franja en 3D
        for i in range(len(franja_ordenada)):
            for j in range(i + 1, min(i + 7, len(franja_ordenada))):
                dist_puntos_franja = distancia_euclidiana_3D(franja_ordenada[i], franja_ord
                dist_min = min(dist_min, dist_puntos_franja)

        return dist_min

    return distancia_minima_recursiva(0, len(lista_ordenada) - 1)

# Llamamos a la función y mostramos el resultado
distancia = encontrar_puntos_mas_cercanos_3D(LISTA_3D)
print(f"La distancia más corta entre dos puntos en 3D es: {distancia}")
```

```
La distancia más corta entre dos puntos en 3D es: 8.54400374531753
```