UNIVERSIDAD INTERNACIONAL DE VALENCIA

Magister Inteligencia Artificial

Trabajo - AG1- Actividad Guiada 2

Docente - Raúl Reyero Díez

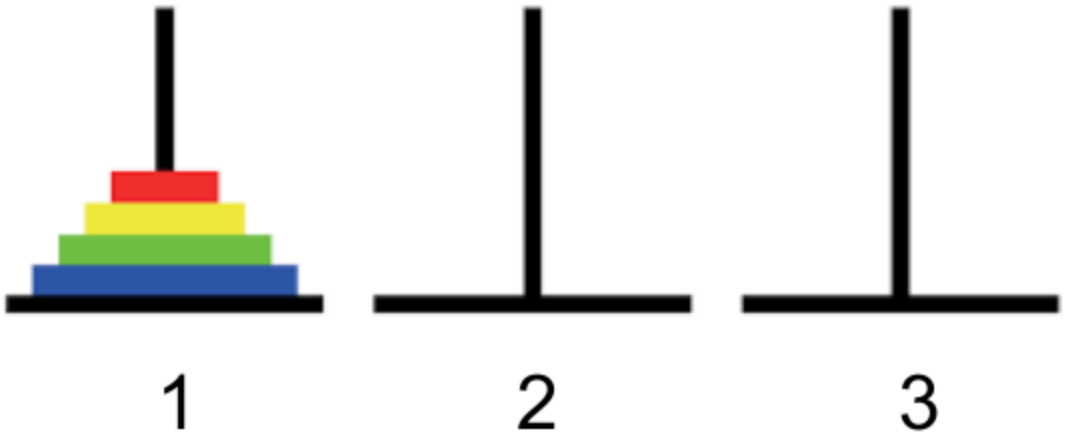
Cátedra - 03MIAR_10_A_2024-25_Algoritmos de Optimización

Alumno: Eduardo Osorio Venegas

- github: https://github.com/eosovngas/VIU_MUIA_AP_102024/tree/main/EOSORIO_AG2_AlgoritmosOptimizacion
- notebook: EOSORIO_AG2_AlgoritmosOptimizacion/EOSORIO_AG2_AlgoritmosOptimizacion.ipynb
- colab: https://colab.research.google.com/github/eosovngas/VIU_MUIA_AP_102024/blob/main/EOSORIO_AG2_AlgoritmosOptimizacion/EOSORIO_AG2_AlgoritmosOptimizacion.ipynb

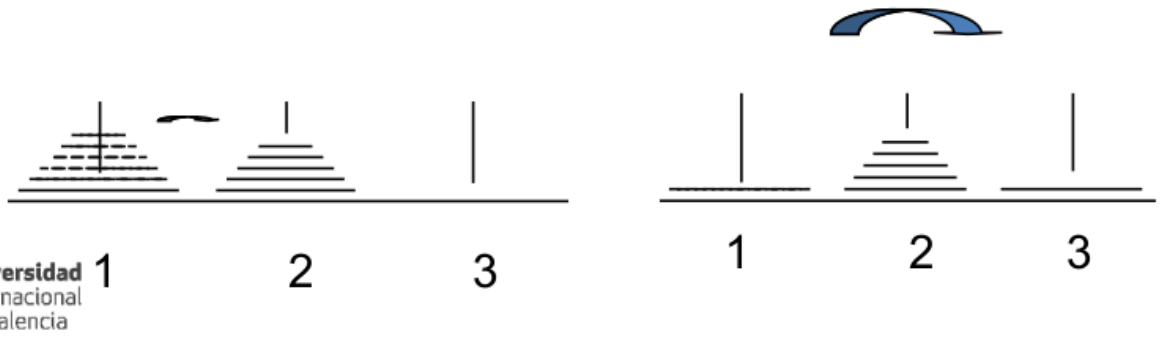
Problema: Torres de Hanoy (divide y venceras)

Torres de Hanoi - Divide y venceras



Resolver(Total_fichas=4, Desde=1 , Hasta=3) es valido con:

- Resolver(Total_fichas=3, Desde=1, Hasta=2)
- Mover(Desde=1, Hasta=3)
- Resolver(Total_fichas=3, Desde=2, Hasta=3)



viu Universidad Internacional de Valencia

```
def mueve_ficha(desde, hasta):
    """
    Mueve una ficha de una torre a otra.
    :param desde: Torre origen
    :param hasta: Torre destino
    """
    print(f"Lleva la ficha desde {desde} hasta {hasta}")


def resuelve_hanoi(n, desde, auxiliar, hasta):
```

```
    """
    Resuelve el problema de las Torres de Hanoi aplicando divide y vencerás con llam
    :param n: Número de fichas
    :param desde: Torre origen
    :param auxiliar: Torre auxiliar
    :param hasta: Torre destino
    """
    if n == 0:
        return  # Caso base: no hay fichas que mover

    # Paso 1: Mover las n-1 fichas al auxiliar
    resuelve_hanoi(n - 1, desde, hasta, auxiliar)

    # Paso 2: Mover la ficha más grande al destino
    mueve_ficha(desde, hasta)

    # Paso 3: Mover las n-1 fichas desde el auxiliar al destino
    resuelve_hanoi(n - 1, auxiliar, desde, hasta)


def torres_hanoi(n):
    """
    Función principal para resolver el problema de las Torres de Hanoi.
    :param n: Número total de fichas
    """
    resuelve_hanoi(n, 1, 2, 3)  # 1: Torre origen, 2: Torre auxiliar, 3: Torre desti


# Ejemplo: Resolver las Torres de Hanoi con 4 discos
torres_hanoi()
```

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

Cómo Funciona

Divide el problema:

- La función **resuelve_hanoi** toma el número de discos (n),
- El poste de origen, el auxiliar y el destino. Divide el problema en tres partes:
    1. Mover n−1 discos al auxiliar.
    2. Mover el disco más grande al destino.
    3. Mover los n−1 discos del auxiliar al destino.
- Caso Base: Si n = 0, no hace nada se asegura que la recursión termina.
- Combina las soluciones:
    1. Llama recursivamente a resuelve_hanoi para resolver los subproblemas.
    2. Usa la función mueve_ficha para imprimir el movimiento de los discos.
- Torre Inicial y Final: la función **torres_hanoi** actúa como una interfaz para configurar la torre origen (1), la auxiliar (2) y la torre destino (3).

```
def inicializar_torres(n):
    """
    Inicializa las torres con las fichas en la primera torre.
    :param n: Número de fichas
    :return: Lista de tres torres (listas)
    """
    return [list(range(n, 0, -1)), [], []]


def mostrar_torres(torres):
    """
    Muestra el estado actual de las torres.
    :param torres: Lista de tres torres (listas)
    """
    print("\nEstado actual:")
    niveles = max(len(torre) for torre in torres)  # Altura máxima
    for i in range(niveles, 0, -1):
        for torre in torres:
            if len(torre) >= i:
                print(f"{torre[i-1]:^5}", end=" ")
            else:
                print("  |  ", end=" ")
        print()
    print("  1     2     3  ")  # Etiquetas de las torres
    print("-" * 20)


def mueve_ficha_visual(torres, desde, hasta):
    """
    Mueve una ficha de una torre a otra y muestra el estado.
    :param torres: Lista de tres torres (listas)
    :param desde: Índice de la torre origen (0, 1, 2)
    :param hasta: Índice de la torre destino (0, 1, 2)
    """
```

```
    ficha = torres[desde].pop()
    torres[hasta].append(ficha)
    print(f"\nMover ficha desde {desde + 1} hasta {hasta + 1}")
    mostrar_torres(torres)


def resuelve_hanoi_visual(n, desde, auxiliar, hasta, torres):
    """
    Resuelve el problema de las Torres de Hanoi con visualización gráfica.
    :param n: Número de fichas
    :param desde: Torre origen
    :param auxiliar: Torre auxiliar
    :param hasta: Torre destino
    :param torres: Lista de tres torres (listas)
    """
    if n == 0:
        return  # Caso base

    # Paso 1: Mover las n-1 fichas al auxiliar
    resuelve_hanoi_visual(n - 1, desde, hasta, auxiliar, torres)

    # Paso 2: Mover la ficha más grande al destino
    mueve_ficha_visual(torres, desde, hasta)

    # Paso 3: Mover las n-1 fichas desde el auxiliar al destino
    resuelve_hanoi_visual(n - 1, auxiliar, desde, hasta, torres)


def torres_hanoi_visual(n):
    """
    Función principal para resolver el problema de las Torres de Hanoi con visualiza
    :param n: Número total de fichas
    """
    torres = inicializar_torres(n)  # Inicializar las torres con las fichas en la pr
    mostrar_torres(torres)  # Mostrar estado inicial
    resuelve_hanoi_visual(n, 0, 1, 2, torres)  # Resolver el problema


# Ejecutar con 4 discos
torres_hanoi_visual(4)
```

```
    Estado actual:
      1     |     |
      2     |     |
      3     |     |
      4     |     |
      1     2     3
    --------------------

    Mover ficha desde 1 hasta 2
```

```
Estado actual:
  2     |     |
  3     |     |
  4     1     |
  1     2     3
--------------------

Mover ficha desde 1 hasta 3

Estado actual:
  3     |     |
  4     1     2
  1     2     3
--------------------

Mover ficha desde 2 hasta 3

Estado actual:
  3     |     1
  4     |     2
  1     2     3
--------------------

Mover ficha desde 1 hasta 2

Estado actual:
  |     |     1
  4     3     2
  1     2     3
--------------------

Mover ficha desde 3 hasta 1

Estado actual:
  1     |     |
  4     3     2
  1     2     3
--------------------

Mover ficha desde 3 hasta 2

Estado actual:
  1     2     |
  4     3     |
  1     2     3
--------------------
```

Como funciona:

- Iniciar: La función **inicializar_torres** crea una lista de tres sublistas. La primera contiene las fichas ordenadas de mayor a menor.

- Visualización: La función **mostrar_torres** imprime el estado actual de las torres, utilizando caracteres ASCII para representar las fichas y las torres.
- Movimiento de fichas: La función **mueve_ficha_visual** realiza el movimiento de una ficha entre torres y actualiza el estado visual.
- Resolución: La función **resuelve_hanoi_visual** aplica el mismo algoritmo recursivo que antes, pero ahora incluye llamadas a **mueve_ficha_visual** para mostrar el movimiento.
- Ejecución: La función principal **torres_hanoi_visual** se encarga de inicializar las torres, mostrar el estado inicial y resolver el problema.

```
#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))

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

Torres_Hanoi(4, 1, 3)
################################################################
```

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

› Cambio de monedas - Técnica voraz

↳ 1 celda oculta

¿Qué ocurre con otros sistemas monetarios?

```
def cambio_monedas(cantidad, sistema):
    """
    Algoritmo voraz para el problema del cambio de monedas.
    """
    solucion = [0] * len(sistema)
    valor_acumulado = 0

    for i, valor in enumerate(sistema):
        monedas = (cantidad - valor_acumulado) // valor
        solucion[i] = monedas
        valor_acumulado += monedas * valor

        if valor_acumulado == cantidad:
            return solucion

    return solucion

def validar_solucion_optima(cantidad, sistema):
    """
    Verifica si el algoritmo voraz da una solución óptima comparando con fuerza brut
    """
    # Solución voraz
    solucion_voraz = cambio_monedas(cantidad, sistema)
    monedas_voraz = sum(solucion_voraz)

    # Fuerza bruta: probar todas las combinaciones
    from itertools import product

    mejor_solucion = None
    min_monedas = float('inf')
    for combinacion in product(range(cantidad // sistema[-1] + 1), repeat=len(sistem
        valor = sum(c * s for c, s in zip(combinacion, sistema))
        if valor == cantidad:
            num_monedas = sum(combinacion)
            if num_monedas < min_monedas:
                min_monedas = num_monedas
                mejor_solucion = combinacion

    return {
        "solucion_voraz": solucion_voraz,
        "monedas_voraz": monedas_voraz,
        "mejor_solucion": mejor_solucion,
        "monedas_optimas": min_monedas,
        "es_optimo": monedas_voraz == min_monedas
    }
```

```
# Ejemplo de validación
sistemas = {
    "Canónico": [25, 10, 5, 1],
    "No Canónico": [10, 7, 1]
}
cantidad = 14

for nombre, sistema in sistemas.items():
    resultado = validar_solucion_optima(cantidad, sistema)
    print(f"\nSistema: {nombre} -> {sistema}")
    print(f"Cantidad: {cantidad}")
    print(f"Solución Voraz: {resultado['solucion_voraz']} ({resultado['monedas_voraz
    print(f"Mejor Solución: {resultado['mejor_solucion']} ({resultado['monedas_optim
    print(f"¿Es Óptimo?: {'Sí' if resultado['es_optimo'] else 'No'}")
```

```
Sistema: Canónico -> [25, 10, 5, 1]
Cantidad: 14
Solución Voraz: [0, 1, 0, 4] (5 monedas)
Mejor Solución: (0, 1, 0, 4) (5 monedas)
¿Es Óptimo?: Sí

Sistema: No Canónico -> [10, 7, 1]
Cantidad: 14
Solución Voraz: [1, 0, 4] (5 monedas)
Mejor Solución: (0, 2, 0) (2 monedas)
¿Es Óptimo?: No
```

- ¿Cuándo funciona bien y cuándo no? Funciona bien cuando el sistema es un sistema canónico: es decir, cuando las monedas mayores son múltiplos de las menores. Ejemplo: [25, 10, 5, 1].
- No funciona bien cuando no es canónico. Ejemplo: [10, 7, 1].

```
#Cambio de monedas - Técnica voraz
################################################################################
SISTEMA = [11, 5 ,1  ]
################################################################
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 = ValorAcumulado + monedas*valor

    if CANTIDAD == ValorAcumulado:
      return SOLUCION
```

```
  print("No es posible encontrar solucion")
cambio_monedas(15,SISTEMA)


#################################################################
```

```
[1, 0, 4]
```

N Reinas - Vuelta Atrás(Backtracking)

```
#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 f
  for i in range(etapa+1):
    #print("El valor " + str(SOLUCION[i]) + " está " +  str(SOLUCION.count(SOLUCION[
    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...]
    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(5,solucion=[],etapa=0)
```

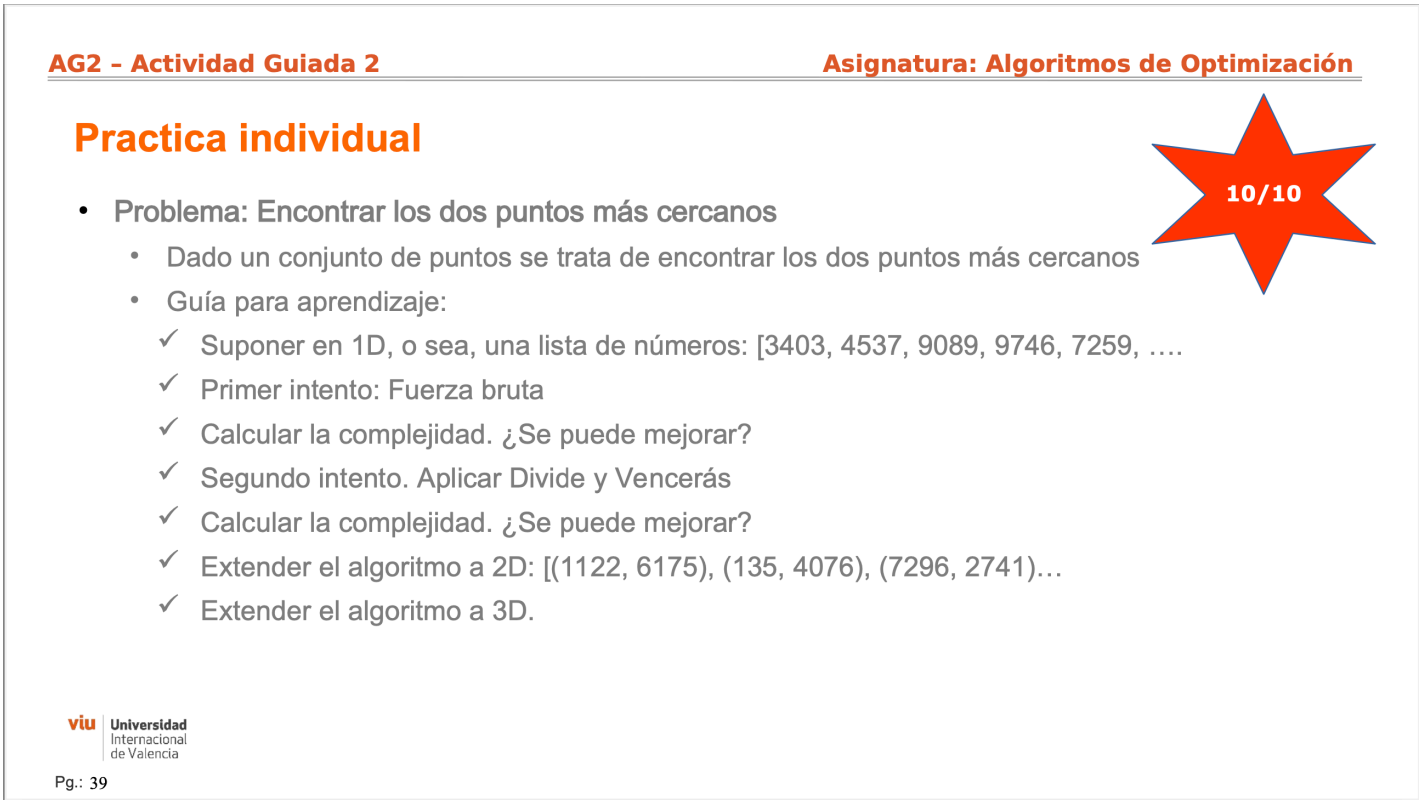
```
[1, 3, 5, 2, 4]
[1, 4, 2, 5, 3]
[2, 4, 1, 3, 5]
[2, 5, 3, 1, 4]
[3, 1, 4, 2, 5]
[3, 5, 2, 4, 1]
[4, 1, 3, 5, 2]
[4, 2, 5, 3, 1]
[5, 2, 4, 1, 3]
[5, 3, 1, 4, 2]
```

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

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

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

Practica individual



Comienza a programar o generar con IA.

Comienza a programar o generar con IA.

Implementación Fuerza Bruta en Python (1D)

- Lista acotada de numeros
- Lista random con numeros duplicados
- Lista random sin numeros duplicados

```
def encontrar_puntos_cercanos_bruto(puntos):
    n = len(puntos)
    min_distancia = float('inf')  # Inicializar con un valor muy grande
```

```
    puntos_cercanos = None

    # Comparar todos los pares posibles
    for i in range(n):
        for j in range(i + 1, n):
            distancia = abs(puntos[i] - puntos[j])  # Distancia en 1D
            if distancia < min_distancia:
                min_distancia = distancia
                puntos_cercanos = (puntos[i], puntos[j])

    return puntos_cercanos, min_distancia

# Lista de puntos en 1D, Lista acotada de numeros
puntos_1d = [3403, 4537, 9089, 9746, 7259]
puntos_cercanos, distancia = encontrar_puntos_cercanos_bruto(puntos_1d)
print(f"Los dos puntos más cercanos en 1D son {puntos_cercanos} con una distancia de
```

```
Los dos puntos más cercanos en 1D son (9089, 9746) con una distancia de 657
```

```
import random

# Generar datos aleatorios en 1D
LISTA_1D = [random.randrange(1, 10000) for _ in range(1000)]

def encontrar_puntos_cercanos_bruto(puntos):
    n = len(puntos)
    min_distancia = float('inf')  # Inicializar con un valor muy grande
    puntos_cercanos = None

    # Comparar todos los pares posibles
    for i in range(n):
        for j in range(i + 1, n):
            distancia = abs(puntos[i] - puntos[j])  # Distancia en 1D
            if distancia < min_distancia:
                min_distancia = distancia
                puntos_cercanos = (puntos[i], puntos[j])

    return puntos_cercanos, min_distancia

# Encontrar los puntos más cercanos, Lista random con numeros duplicados
puntos_cercanos, distancia = encontrar_puntos_cercanos_bruto(LISTA_1D)
print(f"Los dos puntos más cercanos en 1D son {puntos_cercanos} con una distancia de
```

```
Los dos puntos más cercanos en 1D son (6233, 6233) con una distancia de 0
```

```
import random

# Generar datos aleatorios únicos en 1D, Lista random sin numeros duplicados
LISTA_1D = list(set(random.randrange(1, 10000) for _ in range(1000)))
```

```
def encontrar_puntos_cercanos_bruto(puntos):
    n = len(puntos)
    min_distancia = float('inf')  # Inicializar con un valor muy grande
    puntos_cercanos = None

    # Comparar todos los pares posibles
    for i in range(n):
        for j in range(i + 1, n):
            distancia = abs(puntos[i] - puntos[j])  # Distancia en 1D
            if distancia < min_distancia:
                min_distancia = distancia
                puntos_cercanos = (puntos[i], puntos[j])

    return puntos_cercanos, min_distancia

# Encontrar los puntos más cercanos
puntos_cercanos, distancia = encontrar_puntos_cercanos_bruto(LISTA_1D)
print(f"Los dos puntos más cercanos en 1D son {puntos_cercanos} con una distancia de
```

```
Los dos puntos más cercanos en 1D son (8194, 8195) con una distancia de 1
```

Implementación divide y venceras en Python (1D)

```
# Generar datos aleatorios únicos en 1D, Lista random sin numeros duplicados
LISTA_1D = list(set(random.randrange(1, 10000) for _ in range(1000)))

def encontrar_puntos_cercanos_divide(puntos):
    def dividir_y_encontrar(puntos_ordenados):
        # Si solo hay 2 o menos puntos, se hace una comparación directa
        n = len(puntos_ordenados)
        if n <= 3:
            return encontrar_puntos_cercanos_bruto(puntos_ordenados)

        # Dividir el conjunto en dos mitades
        mitad = n // 2
        izquierda = puntos_ordenados[:mitad]
        derecha = puntos_ordenados[mitad:]

        # Recursivamente encontrar el par más cercano en las dos mitades
        (puntos_izq, dist_izq) = dividir_y_encontrar(izquierda)
        (puntos_der, dist_der) = dividir_y_encontrar(derecha)

        # Obtener la distancia mínima entre los dos grupos
        min_distancia = min(dist_izq, dist_der)
        puntos_cercanos = puntos_izq if dist_izq < dist_der else puntos_der

        return puntos_cercanos, min_distancia
```

```
    # Ordenar los puntos para aplicar Divide y Vencerás
    puntos_ordenados = sorted(puntos)
    return dividir_y_encontrar(puntos_ordenados)

# Encontrar los puntos más cercanos en 1D usando Divide y Vencerás
puntos_cercanos, distancia = encontrar_puntos_cercanos_divide(LISTA_1D)
print(f"Los dos puntos más cercanos en 1D (Divide y Vencerás) son {puntos_cercanos}
```

```
Los dos puntos más cercanos en 1D (Divide y Vencerás) son (9493, 9494) con una d
```

Comparación de Complejidad Computacional

Algoritmo	Complejidad	Ventaja
Fuerza Bruta	(O(n^2))	Fácil de implementar, pero ineficiente para grandes conjuntos de datos.
Divide y Vencerás	(O(n \log n))	Más eficiente para grandes conjuntos de datos, aunque más complejo de implementar.

Detalles de la Complejidad

Fuerza Bruta:

- Compara todos los pares posibles de puntos.
- La cantidad de comparaciones es (\binom{n}{2} = \frac{n(n-1)}{2}).
- Esto lleva a una complejidad cuadrática, (O(n^2)).

Divide y Vencerás:

- Divide el conjunto en dos mitades recursivamente.
- Ordena los puntos al inicio, lo cual tiene un costo de (O(n \log n)).
- En cada nivel de recursión combina los resultados con un costo lineal, (O(n)).
- La relación de recurrencia es (T(n) = 2T(n/2) + O(n)), resolviendo a (O(n \log n)).

Conclusión:

- El enfoque de Divide y Vencerás es más eficiente que el de Fuerza Bruta para grandes conjuntos de puntos.

Implementación Fuerza Bruta en Python (2D)

```
# Generar datos aleatorios en 2D
LISTA_2D = [(random.randrange(1, 10000), random.randrange(1, 10000)) for _ in range(

import math

def distancia_euclidiana(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
```

```
def encontrar_puntos_cercanos_bruto_2d(puntos):
    n = len(puntos)
    min_distancia = float('inf')
    puntos_cercanos = None

    # Comparar todos los pares posibles
    for i in range(n):
        for j in range(i + 1, n):
            dist = distancia_euclidiana(puntos[i], puntos[j])
            if dist < min_distancia:
                min_distancia = dist
                puntos_cercanos = (puntos[i], puntos[j])

    return puntos_cercanos, min_distancia

# Encontrar los puntos más cercanos
puntos_cercanos, distancia = encontrar_puntos_cercanos_bruto_2d(LISTA_2D)
print(f"Los dos puntos más cercanos en 2D son {puntos_cercanos} con una distancia de
```

```
Los dos puntos más cercanos en 2D son ((8214, 8982), (8226, 8988)) con una dista
```

Implementación divide y venceras en Python (2D)

```
import random
import math

# Generar datos aleatorios en 2D
LISTA_2D = [(random.randrange(1, 10000), random.randrange(1, 10000)) for _ in range(

def distancia_2d(p1, p2):
    """Calcula la distancia euclidiana entre dos puntos en 2D."""
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def encontrar_puntos_cercanos_divide_2d(puntos):
    def dividir_y_encontrar(puntos_ordenados_x, puntos_ordenados_y):
        n = len(puntos_ordenados_x)

        # Si hay 3 o menos puntos, resolver usando fuerza bruta
        if n <= 3:
            return encontrar_puntos_cercanos_bruto_2d(puntos_ordenados_x)

        # Dividir los puntos en dos mitades
        mitad = n // 2
        izquierda_x = puntos_ordenados_x[:mitad]
        derecha_x = puntos_ordenados_x[mitad:]

        # Crear listas ordenadas por Y para cada mitad
```

```
        punto_mitad = puntos_ordenados_x[mitad][0]
        izquierda_y = [p for p in puntos_ordenados_y if p[0] <= punto_mitad]
        derecha_y = [p for p in puntos_ordenados_y if p[0] > punto_mitad]

        # Resolver recursivamente para las dos mitades
        (puntos_izq, dist_izq) = dividir_y_encontrar(izquierda_x, izquierda_y)
        (puntos_der, dist_der) = dividir_y_encontrar(derecha_x, derecha_y)

        # Obtener la distancia mínima
        min_distancia = min(dist_izq, dist_der)
        puntos_cercanos = puntos_izq if dist_izq < dist_der else puntos_der

        # Considerar puntos cercanos al plano de división
        franja_cercana = [p for p in puntos_ordenados_y if abs(p[0] - punto_mitad) <

        for i in range(len(franja_cercana)):
            for j in range(i + 1, min(i + 7, len(franja_cercana))):  # Revisar hasta
                distancia = distancia_2d(franja_cercana[i], franja_cercana[j])
                if distancia < min_distancia:
                    min_distancia = distancia
                    puntos_cercanos = (franja_cercana[i], franja_cercana[j])

        return puntos_cercanos, min_distancia

    # Ordenar puntos por X y Y
    puntos_ordenados_x = sorted(puntos, key=lambda x: x[0])
    puntos_ordenados_y = sorted(puntos, key=lambda x: x[1])

    return dividir_y_encontrar(puntos_ordenados_x, puntos_ordenados_y)

# Encontrar los puntos más cercanos en 2D usando Divide y Vencerás
puntos_cercanos, distancia = encontrar_puntos_cercanos_divide_2d(LISTA_2D)
print(f"Los dos puntos más cercanos en 2D (Divide y Vencerás) son {puntos_cercanos}
```

```
Los dos puntos más cercanos en 2D (Divide y Vencerás) son ((2287, 9556), (2287,
```

Implementación Fuerza Bruta en Python 3D

```
import random
import math

# Generar datos aleatorios en 3D
LISTA_3D = [(random.randrange(1, 10000), random.randrange(1, 10000), random.randrang

def distancia_3d(p1, p2):
    """Calcula la distancia euclidiana entre dos puntos en 3D."""
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2 + (p1[2] - p2[2])**2)
```

```
def encontrar_puntos_cercanos_bruto_3d(puntos):
    """Encuentra los puntos más cercanos en un conjunto de puntos 3D usando fuerza b
    n = len(puntos)
    min_distancia = float('inf')
    puntos_cercanos = None

    for i in range(n):
        for j in range(i + 1, n):
            distancia = distancia_3d(puntos[i], puntos[j])
            if distancia < min_distancia:
                min_distancia = distancia
                puntos_cercanos = (puntos[i], puntos[j])

    return puntos_cercanos, min_distancia

# Encontrar los puntos más cercanos en 3D
puntos_cercanos, distancia = encontrar_puntos_cercanos_bruto_3d(LISTA_3D)
print(f"Los dos puntos más cercanos en 3D son {puntos_cercanos} con una distancia de
```

```
Los dos puntos más cercanos en 3D son ((962, 8121, 1714), (978, 8121, 1761)) con
```

```
def encontrar_puntos_cercanos_divide_3d(puntos):
    def dividir_y_encontrar(puntos_ordenados_x, puntos_ordenados_y, puntos_ordenados
        n = len(puntos_ordenados_x)
        if n <= 3:
            return encontrar_puntos_cercanos_bruto_3d(puntos_ordenados_x)

        # Dividir los puntos en dos mitades
        mitad = n // 2
        izquierda_x = puntos_ordenados_x[:mitad]
        derecha_x = puntos_ordenados_x[mitad:]

        # Crear listas ordenadas por Y y Z para cada lado
        izquierda_y = [p for p in puntos_ordenados_y if p in izquierda_x]
        derecha_y = [p for p in puntos_ordenados_y if p in derecha_x]
        izquierda_z = [p for p in puntos_ordenados_z if p in izquierda_x]
        derecha_z = [p for p in puntos_ordenados_z if p in derecha_x]

        # Resolver recursivamente para las dos mitades
        (puntos_izq, dist_izq) = dividir_y_encontrar(izquierda_x, izquierda_y, izqui
        (puntos_der, dist_der) = dividir_y_encontrar(derecha_x, derecha_y, derecha_z

        # Combinar resultados
        min_distancia = min(dist_izq, dist_der)
        puntos_cercanos = puntos_izq if dist_izq < dist_der else puntos_der

        # Considerar puntos cercanos al plano de división
        plano_cercano = [p for p in puntos_ordenados_y if abs(p[0] - puntos_ordenado

        for i in range(len(plano_cercano)):
```

```
            for j in range(i + 1, len(plano_cercano)):
                if abs(plano_cercano[j][1] - plano_cercano[i][1]) >= min_distancia:
                    break
                distancia = distancia_3d(plano_cercano[i], plano_cercano[j])
                if distancia < min_distancia:
                    min_distancia = distancia
                    puntos_cercanos = (plano_cercano[i], plano_cercano[j])

        return puntos_cercanos, min_distancia

    # Ordenar los puntos por cada coordenada
    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])

    return dividir_y_encontrar(puntos_ordenados_x, puntos_ordenados_y, puntos_ordena

# Encontrar los puntos más cercanos en 3D usando Divide y Vencerás
puntos_cercanos, distancia = encontrar_puntos_cercanos_divide_3d(LISTA_3D)
print(f"Los dos puntos más cercanos en 3D (Divide y Vencerás) son {puntos_cercanos}
```

```
Los dos puntos más cercanos en 3D (Divide y Vencerás) son ((962, 8121, 1714), (9
```

Análisis de Complejidad Computacional para el Problema de los Puntos Más Cercanos

1D - Fuerza Bruta: (O(n^2))

- El algoritmo compara todos los pares de puntos posibles en una lista de (n) elementos.
- El número de pares posibles es: [\text{Número de comparaciones} = \binom{n}{2} = \frac{n(n-1)}{2}]
- Esto lleva a una complejidad asintótica de (O(n^2)).

1D - Divide y Vencerás: (O(n \log n))

- **División**: Se divide la lista en dos mitades en cada nivel de la recursión. El número de niveles de recursión es (\log n) porque en cada paso se reduce el problema a la mitad.
- **Conquista**:
    - Se resuelve el problema de forma recursiva en ambas mitades, y luego se combina la solución.

    - La combinación (comparar puntos cercanos al plano de separación) es (O(n)) porque se revisan solo los puntos cercanos al corte.
- En total, el tiempo se modela como: [T(n) = 2T\left(\frac{n}{2}\right) + O(n)]
- Resolviendo esta recurrencia, se obtiene (O(n \log n)).

2D - Fuerza Bruta: (O(n^2))

- Similar al caso 1D, se comparan todos los pares de puntos posibles en un conjunto de (n) puntos.
- La distancia euclidiana en 2D requiere calcular: [d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}] pero esto no afecta la complejidad asintótica, que sigue siendo (O(n^2)).

2D - Divide y Vencerás: (O(n \log n))

- **División**: El conjunto de puntos se divide en dos mitades según la coordenada (x).
- **Conquista**:
    - Se resuelve el problema recursivamente en ambas mitades.
    - La combinación considera solo puntos cercanos al plano de división (con una ventana de anchura proporcional a la distancia mínima encontrada).
    - En 2D, hay como máximo 7 puntos que deben compararse por cada punto en la franja (según el teorema geométrico). Esto mantiene la combinación en (O(n)).
- Resolviendo la recurrencia: [T(n) = 2T\left(\frac{n}{2}\right) + O(n)] El tiempo total es (O(n \log n)).

3D - Fuerza Bruta: (O(n^2))

- Igual que en 1D y 2D, se comparan todos los pares posibles.
- La distancia en 3D requiere calcular: [d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}] pero la complejidad sigue siendo (O(n^2)).

3D - Divide y Vencerás: (O(n \log^2 n))

- **División**: Similar al caso 2D, el conjunto se divide en mitades según una coordenada.
- **Conquista**:
    - El problema se resuelve en ambas mitades recursivamente.
    - Para combinar, los puntos cercanos al plano de división (en una franja tridimensional) deben analizarse.

    - En 3D, la cantidad de puntos en la franja y las comparaciones necesarias aumentan ligeramente, dando una complejidad de combinación (O(n \log n)) en lugar de (O(n)).
- La recurrencia se modela como: [T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)] Resolviendo, obtenemos (O(n \log^2 n)).

Resumen de Complejidades

Dimensión	Método	Complejidad
1D	Fuerza Bruta	(O(n^2))
1D	Divide y Vencerás	(O(n \log n))
2D	Fuerza Bruta	(O(n^2))
2D	Divide y Vencerás	(O(n \log n))
3D	Fuerza Bruta	(O(n^2))
3D	Divide y Vencerás	(O(n \log^2 n))

Comienza a programar o generar con IA.