Actividad Guiada 1 de Algoritmos de Optimizacion

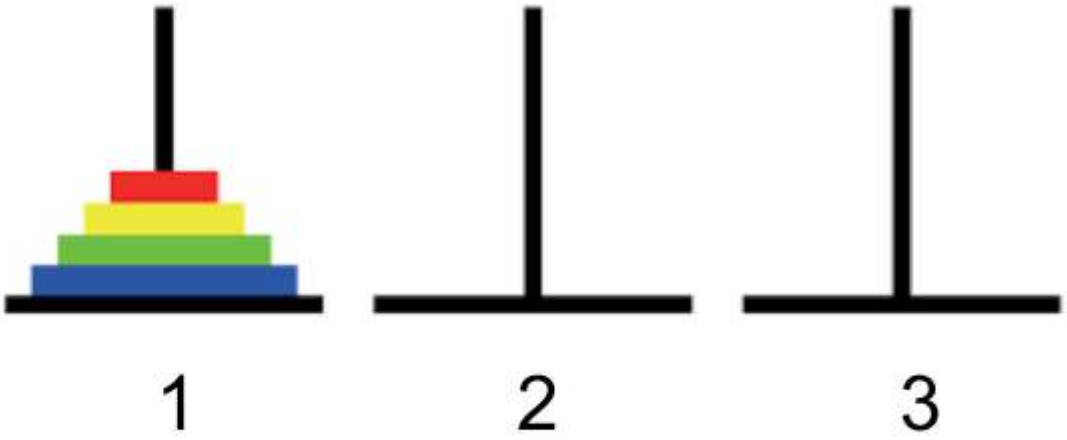
Nombre: Victor Callejas Fuentes

vcallejasf@student.universidadviu.com

https://colab.research.google.com/drive/1E89DuMmunVpRMWo_x-inUnNnQJzCGfbT?usp=sharing

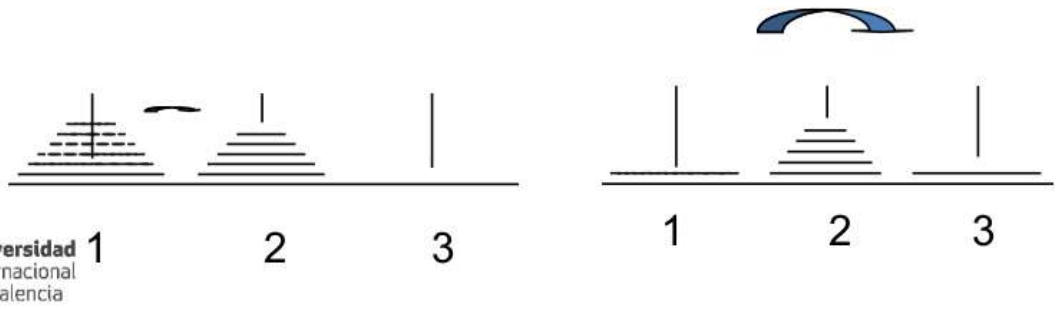
https://github.com/VictorCallejas/03miar/tree/main

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

```
# 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(5, 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
Lleva la ficha desde 1 hasta 2
Lleva la ficha desde 3 hasta 2
Lleva la ficha desde 3 hasta 1
Lleva la ficha desde 2 hasta 1
Lleva la ficha desde 3 hasta 2
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
Lleva la ficha desde 2 hasta 1
Lleva la ficha desde 3 hasta 2
Lleva la ficha desde 3 hasta 1
Lleva la ficha desde 2 hasta 1
Lleva la ficha desde 2 hasta 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
```

Análisis del Algoritmo - Torres de Hanoi

Técnica algorítmica: Divide y Vencerás

Descripción del problema: El problema de las Torres de Hanoi consiste en mover n discos de una torre origen a una torre destino, usando una torre auxiliar, siguiendo las reglas:

1. Solo se puede mover un disco a la vez
2. Un disco solo puede colocarse sobre otro más grande o en una torre vacía

Estrategia Divide y Vencerás:

1. **Dividir:** Separar el problema en subproblemas más pequeños
2. **Conquistar:** Resolver los subproblemas recursivamente
3. **Combinar:** Juntar las soluciones

Para mover n discos de A a C:

- Mover n-1 discos de A a B (usando C como auxiliar)
- Mover el disco n de A a C
- Mover n-1 discos de B a C (usando A como auxiliar)

Análisis de Complejidad:

- **Recurrencia:** T(n) = 2T(n-1) + 1
- **Complejidad temporal:** O(2^n) - exponencial
- **Complejidad espacial:** O(n) - por la profundidad de la recursión
- **Número de movimientos:** 2^n - 1

Cambio de monedas - Técnica voraz

↳ 2 cells hidden

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

Análisis del Algoritmo - N-Reinas

Técnica algorítmica: Vuelta Atrás (Backtracking)

Descripción del problema: Colocar N reinas en un tablero de NxN de manera que ninguna reina amenace a otra. Una reina amenaza a otra si están en la misma fila, columna o diagonal.

Estrategia Backtracking:

1. **Exploración sistemática:** Probar todas las posiciones posibles
2. **Poda:** Descartar ramas del árbol de búsqueda que no pueden llevar a solución
3. **Retroceso:** Si una configuración no es válida, volver atrás y probar otra opción

Funcionamiento del algoritmo:

- Se representa la solución como un vector donde `SOLUCION[i]` indica la fila donde está la reina de la columna i
- Para cada columna, se prueban todas las filas posibles
- `es_prometedora()` verifica si la configuración parcial es válida:
  - No hay dos reinas en la misma fila
  - No hay dos reinas en la misma diagonal

Análisis de Complejidad:

- **Complejidad temporal (peor caso):** O(N!) - se exploran todas las permutaciones
- **Complejidad temporal (con poda):** Significativamente menor en la práctica
- **Complejidad espacial:** O(N) - para almacenar la solución y la pila de recursión

Número de soluciones para diferentes N:

N	Soluciones
4	2
5	10
8	92

Problema de los Dos Puntos Más Cercanos (Entrega Extra)

Descripción del Problema

Dado un conjunto de n puntos en el espacio (1D, 2D o 3D), encontrar el par de puntos con la menor distancia euclidiana entre ellos.

Enfoques de Solución

1. **Fuerza Bruta:** Comparar todos los pares posibles - $O(n^2)$
2. **Divide y Vencerás:** Dividir el espacio y combinar resultados - O(n log n)

```
# Importaciones
import math
import random
import time
from functools import wraps
from typing import List, Tuple

# Tipo para representar puntos en cualquier dimension
Punto = Tuple[float, ...]
```

```
# Decorador simple para medir tiempo de ejecucion
def medir_tiempo(func):
    """Decorador que mide e imprime el tiempo de ejecucion de una funcion."""
    @wraps(func)
    def wrapper(*args, **kwargs):
        inicio = time.perf_counter()
        resultado = func(*args, **kwargs)
        tiempo = time.perf_counter() - inicio
        print(f"{func.__name__}: {tiempo:.6f}s")
        return resultado
    return wrapper
```

1. Solución por Fuerza Bruta

El enfoque más simple: comparar todos los pares de puntos posibles.

Complejidad: $O(n^2)$ ya que hay n(n-1)/2 pares posibles.

```
def puntos_mas_cercanos_fuerza_bruta(puntos: List[Punto]) -> Tuple[Punto, Puntc
    """
    Encuentra los dos puntos más cercanos usando fuerza bruta.
    Funciona para puntos en cualquier dimensión (1D, 2D, 3D, ...).

    Complejidad temporal: O(n²)
    Complejidad espacial: O(1)
```

```
        Args:
            puntos: Lista de puntos (tuplas de coordenadas)

        Returns:
            Tupla con (punto1, punto2, distancia_minima)
        """
        n = len(puntos)
        if n < 2:
            raise ValueError("Se necesitan al menos 2 puntos")

        min_dist = float('inf')
        punto1, punto2 = None, None

        # Comparar todos los pares posibles: O(n²)
        for i in range(n):
            for j in range(i + 1, n):
                dist = math.dist(puntos[i], puntos[j])
                if dist < min_dist:
                    min_dist = dist
                    punto1, punto2 = puntos[i], puntos[j]

        return punto1, punto2, min_dist
```

2. Solución por Divide y Vencerás

Estrategia:

1. **Dividir:** Ordenar puntos por coordenada x y dividir en dos mitades
2. **Conquistar:** Resolver recursivamente para cada mitad
3. **Combinar:** Verificar puntos en la franja central (la parte crítica)

Complejidad: O(n log n) con optimizaciones adecuadas

```
    def puntos_mas_cercanos_dyv(puntos: List[Punto]) -> Tuple[Punto, Punto, float]:
        """
        Encuentra los dos puntos más cercanos usando Divide y Vencerás.
        Funciona para puntos en cualquier dimensión (1D, 2D, 3D, ...).

        Complejidad temporal: O(n log n)
        Complejidad espacial: O(n)

        Args:
            puntos: Lista de puntos (tuplas de coordenadas)

        Returns:
            Tupla con (punto1, punto2, distancia_minima)
        """
        n = len(puntos)
```

```
        if n < 2:
            raise ValueError("Se necesitan al menos 2 puntos")

        # Caso base: usar fuerza bruta para pocos puntos
        if n <= 3:
            return puntos_mas_cercanos_fuerza_bruta(puntos)

        # Ordenar por coordenada x (primera coordenada)
        puntos_ordenados = sorted(puntos, key=lambda p: p[0])

        return _dyv_recursivo(puntos_ordenados)


    def _dyv_recursivo(puntos_x: List[Punto]) -> Tuple[Punto, Punto, float]:
        """
        Función recursiva auxiliar para divide y vencerás.

        Args:
            puntos_x: Puntos ordenados por coordenada x

        Returns:
            Tupla con (punto1, punto2, distancia_minima)
        """
        n = len(puntos_x)

        # Caso base: fuerza bruta para 3 o menos puntos
        if n <= 3:
            return puntos_mas_cercanos_fuerza_bruta(puntos_x)

        # DIVIDIR: Separar en dos mitades
        medio = n // 2
        punto_medio = puntos_x[medio]

        izquierda = puntos_x[:medio]
        derecha = puntos_x[medio:]

        # CONQUISTAR: Resolver recursivamente
        p1_izq, p2_izq, dist_izq = _dyv_recursivo(izquierda)
        p1_der, p2_der, dist_der = _dyv_recursivo(derecha)

        # Encontrar la menor distancia de las dos mitades
        if dist_izq < dist_der:
            mejor_p1, mejor_p2, delta = p1_izq, p2_izq, dist_izq
        else:
            mejor_p1, mejor_p2, delta = p1_der, p2_der, dist_der

        # COMBINAR: Verificar la franja central
        # Puntos dentro de la franja de ancho 2*delta alrededor de la línea divisor
        franja = [p for p in puntos_x if abs(p[0] - punto_medio[0]) < delta]

        # Ordenar la franja por coordenada y (segunda coordenada si existe)
```

```
        if len(puntos_x[0]) > 1:
            franja.sort(key=lambda p: p[1])

        # Verificar puntos cercanos en la franja
        # Por la geometria del problema, solo necesitamos comparar con los siguient
        for i in range(len(franja)):
            # Solo comparar con los siguientes puntos (maximo 7 por la geometria)
            j = i + 1
            while j < len(franja) and (len(franja[0]) == 1 or franja[j][1] - franja
                dist = math.dist(franja[i], franja[j])
                if dist < delta:
                    delta = dist
                    mejor_p1, mejor_p2 = franja[i], franja[j]
                j += 1
                # Limitar a 7 comparaciones máximo (propiedad geométrica)
                if j - i > 7:
                    break

        return mejor_p1, mejor_p2, delta
```

3. Pruebas en 1D, 2D y 3D

3.1 Prueba en 1D (puntos en una línea)

```
# Prueba en 1D
print("=" * 60)
print("PRUEBA EN 1D (puntos en una línea)")
print("=" * 60)

puntos_1d = [(1,), (5,), (3,), (8,), (2,), (10,), (7,)]
print(f"Puntos 1D: {puntos_1d}")

# Fuerza bruta
p1_bf, p2_bf, dist_bf = puntos_mas_cercanos_fuerza_bruta(puntos_1d)
print(f"\nFuerza Bruta:")
print(f"  Puntos más cercanos: {p1_bf} y {p2_bf}")
print(f"  Distancia: {dist_bf}")

# Divide y vencerás
p1_dv, p2_dv, dist_dv = puntos_mas_cercanos_dyv(puntos_1d)
print(f"\nDivide y Vencerás:")
print(f"  Puntos más cercanos: {p1_dv} y {p2_dv}")
print(f"  Distancia: {dist_dv}")

print(f"\n¿Resultados coinciden? {dist_bf == dist_dv}")
```

```
============================================================
PRUEBA EN 1D (puntos en una línea)
```

```
============================================================
Puntos 1D: [(1,), (5,), (3,), (8,), (2,), (10,), (7,)]

Fuerza Bruta:
  Puntos más cercanos: (1,) y (2,)
  Distancia: 1.0

Divide y Vencerás:
  Puntos más cercanos: (7,) y (8,)
  Distancia: 1.0

¿Resultados coinciden? True
```

3.2 Prueba en 2D (puntos en un plano)

```
# Prueba en 2D
print("=" * 60)
print("PRUEBA EN 2D (puntos en un plano)")
print("=" * 60)

puntos_2d = [(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3, 4)]
print(f"Puntos 2D: {puntos_2d}")

# Fuerza bruta
p1_bf, p2_bf, dist_bf = puntos_mas_cercanos_fuerza_bruta(puntos_2d)
print(f"\nFuerza Bruta:")
print(f"  Puntos más cercanos: {p1_bf} y {p2_bf}")
print(f"  Distancia: {dist_bf:.4f}")

# Divide y vencerás
p1_dv, p2_dv, dist_dv = puntos_mas_cercanos_dyv(puntos_2d)
print(f"\nDivide y Vencerás:")
print(f"  Puntos más cercanos: {p1_dv} y {p2_dv}")
print(f"  Distancia: {dist_dv:.4f}")

print(f"\n¿Resultados coinciden? {abs(dist_bf - dist_dv) < 0.0001}")
```

```
============================================================
PRUEBA EN 2D (puntos en un plano)
============================================================
Puntos 2D: [(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3, 4)]

Fuerza Bruta:
  Puntos más cercanos: (2, 3) y (3, 4)
  Distancia: 1.4142

Divide y Vencerás:
  Puntos más cercanos: (2, 3) y (3, 4)
  Distancia: 1.4142

¿Resultados coinciden? True
```

3.3 Prueba en 3D (puntos en el espacio)

```
# Prueba en 3D
print("=" * 60)
print("PRUEBA EN 3D (puntos en el espacio)")
print("=" * 60)

puntos_3d = [
    (1, 2, 3), (4, 5, 6), (7, 8, 9),
    (1, 2, 4), (10, 11, 12), (0, 0, 0),
    (5, 5, 5), (5, 5, 6)
]
print(f"Puntos 3D: {puntos_3d}")

# Fuerza bruta
p1_bf, p2_bf, dist_bf = puntos_mas_cercanos_fuerza_bruta(puntos_3d)
print(f"\nFuerza Bruta:")
print(f"  Puntos más cercanos: {p1_bf} y {p2_bf}")
print(f"  Distancia: {dist_bf:.4f}")

# Divide y vencerás
p1_dv, p2_dv, dist_dv = puntos_mas_cercanos_dyv(puntos_3d)
print(f"\nDivide y Vencerás:")
print(f"  Puntos más cercanos: {p1_dv} y {p2_dv}")
print(f"  Distancia: {dist_dv:.4f}")

print(f"\n¿Resultados coinciden? {abs(dist_bf - dist_dv) < 0.0001}")
```

```
============================================================
PRUEBA EN 3D (puntos en el espacio)
============================================================
Puntos 3D: [(1, 2, 3), (4, 5, 6), (7, 8, 9), (1, 2, 4), (10, 11, 12), (0, 0, 0),

Fuerza Bruta:
  Puntos más cercanos: (1, 2, 3) y (1, 2, 4)
  Distancia: 1.0000

Divide y Vencerás:
  Puntos más cercanos: (5, 5, 5) y (5, 5, 6)
  Distancia: 1.0000

¿Resultados coinciden? True
```

4. Comparación de Rendimiento

Comparamos el tiempo de ejecución de ambos algoritmos con conjuntos de datos más grandes.

```
import time

def generar_puntos_aleatorios(n: int, dimension: int, rango: float = 1000.0) ->
    """Genera n puntos aleatorios en la dimensión especificada."""
    return [tuple(random.uniform(0, rango) for _ in range(dimension)) for _ in

def comparar_rendimiento(tamaños: List[int], dimension: int = 2):
    """Compara el rendimiento de ambos algoritmos para diferentes tamaños."""
    print(f"\nComparación de rendimiento en {dimension}D:")
    print("-" * 70)
    print(f"{'N puntos':>10} | {'Fuerza Bruta':>15} | {'Divide y Vencerás':>18}
    print("-" * 70)

    for n in tamaños:
        puntos = generar_puntos_aleatorios(n, dimension)

        # Medir fuerza bruta
        inicio = time.time()
        puntos_mas_cercanos_fuerza_bruta(puntos)
        tiempo_bf = time.time() - inicio

        # Medir divide y vencerás
        inicio = time.time()
        puntos_mas_cercanos_dyv(puntos)
        tiempo_dv = time.time() - inicio

        speedup = tiempo_bf / tiempo_dv if tiempo_dv > 0 else float('inf')

        print(f"{n:>10} | {tiempo_bf:>13.6f}s | {tiempo_dv:>16.6f}s | {speedup:

# Ejecutar comparación
print("=" * 70)
print("COMPARACIÓN DE RENDIMIENTO")
print("=" * 70)

tamaños = [100, 500, 1000, 2000]
comparar_rendimiento(tamaños, dimension=2)
```

```
======================================================================
COMPARACIÓN DE RENDIMIENTO
======================================================================

Comparación de rendimiento en 2D:
----------------------------------------------------------------------
  N puntos |    Fuerza Bruta |  Divide y Vencerás |    Speedup
----------------------------------------------------------------------
       100 |      0.001426s |         0.000428s |      3.33x
       500 |      0.025845s |         0.001411s |     18.32x
      1000 |      0.105644s |         0.004951s |     21.34x
      2000 |      0.522120s |         0.006092s |     85.71x
```

```
# Comparacion de rendimiento
def generar_puntos_aleatorios(n, dimension, rango=1000.0):
    return [tuple(random.uniform(0, rango) for _ in range(dimension)) for _ in

@medir_tiempo
def buscar_fuerza_bruta(puntos):
    return puntos_mas_cercanos_fuerza_bruta(puntos)

@medir_tiempo
def buscar_dyv(puntos):
    return puntos_mas_cercanos_dyv(puntos)

# Prueba con 1000 puntos 2D
puntos_prueba = generar_puntos_aleatorios(1000, 2)
print("Comparando algoritmos con 1000 puntos 2D:")
buscar_fuerza_bruta(puntos_prueba)
buscar_dyv(puntos_prueba)
```

```
Comparando algoritmos con 1000 puntos 2D:
buscar_fuerza_bruta: 0.076097s
buscar_dyv: 0.002714s
((537.7834568037346, 560.2836261183858),
 (538.9097073784413, 561.1760375939712),
 1.436954626557092)
```

5. Análisis de Complejidad y Mejoras

Análisis de Complejidad

Algoritmo	Complejidad Temporal	Complejidad Espacial
Fuerza Bruta	$O(n^2)$	O(1)
Divide y Vencerás	O(n log n)	O(n)

Justificación del Divide y Vencerás

Recurrencia: T(n) = 2T(n/2) + O(n)

- **División:** O(1) para dividir el array
- **Conquista:** 2T(n/2) para resolver los subproblemas
- **Combinación:** O(n) para verificar la franja central

Por el Teorema Maestro, esto resulta en **O(n log n)**.

¿Por qué solo 7 comparaciones en la franja?

Esta es la parte más elegante del algoritmo. Debido a la geometría del problema:

- En la franja de ancho 2δ, los puntos están organizados de forma que solo pueden haber máximo 8 puntos en cualquier rectángulo de δ × 2δ
- Esto limita las comparaciones necesarias a 7 por punto
- Esta propiedad se mantiene en 2D y se puede extender a dimensiones superiores

Posibles Mejoras

1. **Pre-ordenamiento:** Ordenar por Y una sola vez al inicio
2. **Estructura de datos:** Usar KD-Tree para búsquedas más eficientes
3. **Paralelización:** Los subproblemas se pueden resolver en paralelo
4. **Aleatorización:** Algoritmos randomizados pueden lograr O(n) esperado

Double-click (or enter) to edit