Actividad Guiada 1 del curso: Algoritmos de optimización

Nombre: Carlos Esteban Posada
URL: https://colab.research.google.com/drive/1nLEhlOavPebfAYawy4A_KSPBgvXGXx7V?usp=sharing

URL GIT: https://github.com/cposada8/03MAIR-Algoritmos-de-Optimizacion-CEPM/blob/main/carlos_esteban_posada_AG1.ipynb

Introducción

El presente trabajo corresponde a la primera actividad guiada del curso **algoritmos de optimización**

Este trabajo consiste en la implementación práctica de algunos conceptos vistos en las primeras unidades del curso:

- Algoritmos de ordenamiento
- Técicas de diseño de algoritmos
- Desarrollo de un problema opciónal: par de puntos más cercanos

```
# imports
import numpy as np
import random
import matplotlib.pyplot as plt
from time import time
```

Algoritmos de ordenamiento [Opcional]

En esta sección se realizará la impelmentación de los siguientes algoritmos de ordenamiento:

- Insertion sort
- Bubble sort
- Selection sort
- Merge sort

Insertion sort

Es un algoritmo de ordenamiento muy sencillo, pero a su vez, ineficiente. Logra un orden de complegidad **O(n)** en el mejor de los casos, pero **O(n^2)** en los casos peor y promedio

```
def insertion_sort(lista):
  lista = lista.copy()
  n = len(lista)
  for i in range(1, n):
    valor_actual = lista[i]
    position = i
    while position > 0 and lista[position-1] > valor_actual:
      lista[position] = lista[position-1]
      position -= 1

    lista[position] = valor_actual
  return lista


# prueba del algoritmo
lista = [2, 3, 8, 6, 1]
print("lista original", lista)
print("lista ordenada", insertion_sort(lista))
```

```
 lista original [2, 3, 8, 6, 1]
 lista ordenada [1, 2, 3, 6, 8]
```

Bubble sort

```
def bubble_sort(lista):
  lista = lista.copy()
  n = len(lista)
  for max_index in range(n-1, 0, -1):
    for i in range(max_index):
      if lista[i]>lista[i+1]:
        lista[i], lista[i+1] = lista[i+1], lista[i] # swap the elements
  return lista


# probamos con la misma lista del algoritmo anterior
bubble_sort(lista)
```

```
 [1, 2, 3, 6, 8]
```

Selection sort

Es un algoritmo que en cada iteración busca el elemento más grande (o pequeño) y lo sitúa en la posición que le corresponde. Es orden n^2 en todos los casos

```
def selection_sort(lista):
  lista = lista.copy()
  n = len(lista)
  for max_index in range(n, 0, -1):
    max_elem_index = 0
    max_elem = lista[max_elem_index]
    for i in range(max_index):
      if lista[i]>max_elem:
        max_elem = lista[i]
        max_elem_index = i
    # swap max_elem_index with max_index
    lista[max_elem_index], lista[max_index-1] = lista[max_index-1], lista[max_elem_index]
  return lista


lista = [2, 3, 8, 6, 1]
selection_sort(lista)
```

```
[1, 2, 3, 6, 8]
```

Merge sort

```
# auxiliar function to implement merge sort
def merge_ordered_lists(l1, l2):
  # this function merges the lists l1 and l2
  # so the resulted list is ordered.
  # l1 and l2 must be ordered already
  l_merged = []
  while l1 and l2:
    if l1[0]<l2[0]:
      l_merged.append(l1.pop(0))
    else:
      l_merged.append(l2.pop(0))

  # agregar el resto de la lista que sobrara (si alguna aún tiene elementos)
  if l1:
    l_merged += l1
  else:
    l_merged += l2
  return l_merged


def merge_sort(lista):
  if len(lista) == 1:
    return lista
```

```
    return lista
  else:
    # parto a la mitad la lista
    mid = len(lista)//2
    left_list = lista[:mid]
    right_list = lista[mid:]
    return merge_ordered_lists(merge_sort(left_list), merge_sort(right_list))


merge_sort(lista)
```

```
[1, 2, 3, 6, 8]
```

Comparación de los algoritmos de ordenamiento

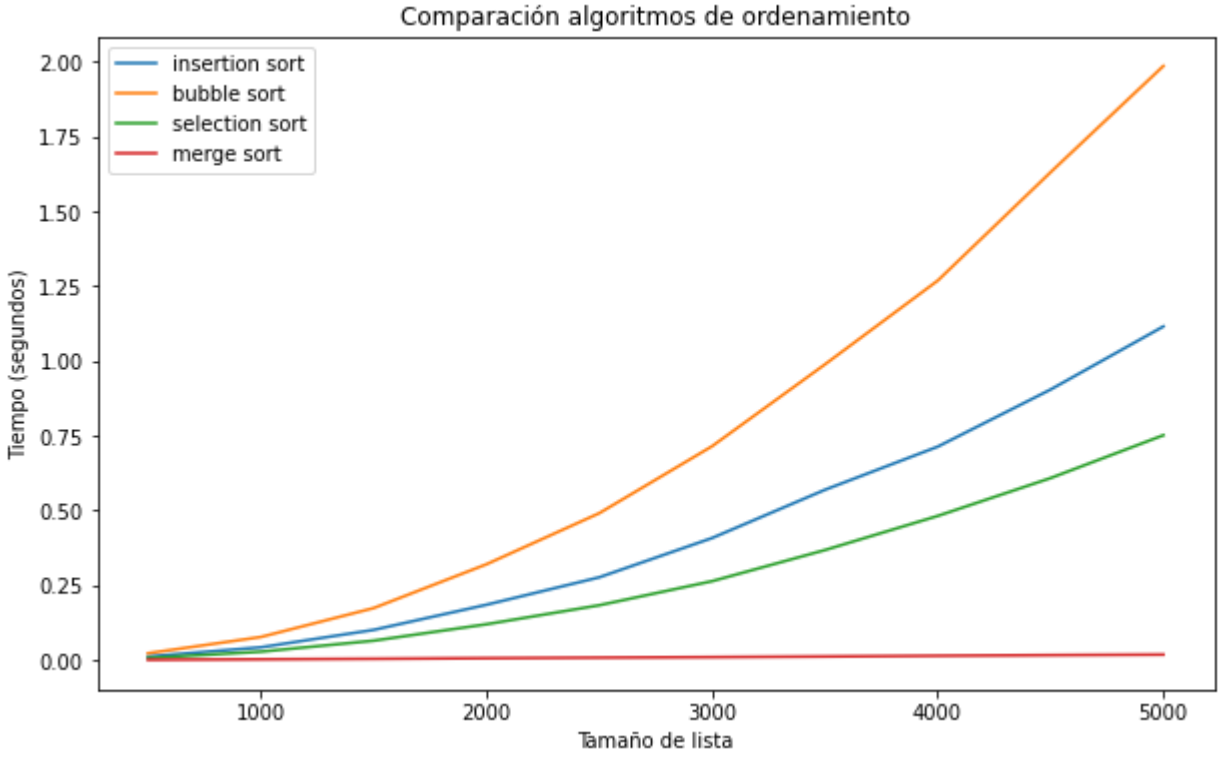
En esta sección se realizará una comparación de los algoritmos de ordenamiento anteriormente implementados. Miraremos el desempeño de cada uno para listas de distintos tamaños

```
def calcular_tiempo_sort_function(f, lista):
  t_ini = time()
  sol_isertion = f(lista.copy())
  t_end = time()
  return t_end-t_ini


tamanos =  range(500, 5500, 500)#[10, 100, 1000, 10_000]


# inicialización de las listas que contendrán los tiempos de desempeño de cada algoritmo
t_insertion = []
t_bubble = []
t_selection = []
t_merge = []

for tam in tamanos:
  print("Calculando listas de tamaño:", tam)
  # Creación de la lista original que cada algoritmo deberá ordenar
  lista = list(np.random.randint(0, 10*tam, tam))

  t_insertion.append(calcular_tiempo_sort_function(insertion_sort, lista))
  t_bubble.append(calcular_tiempo_sort_function(bubble_sort, lista))
  t_selection.append(calcular_tiempo_sort_function(selection_sort, lista))
  t_merge.append(calcular_tiempo_sort_function(merge_sort, lista))
```

```
Calculando listas de tamaño: 500
Calculando listas de tamaño: 1000
Calculando listas de tamaño: 1500
Calculando listas de tamaño: 2000
```

```
Calculando listas de tamaño: 2500
Calculando listas de tamaño: 3000
Calculando listas de tamaño: 3500
Calculando listas de tamaño: 4000
Calculando listas de tamaño: 4500
Calculando listas de tamaño: 5000
```

```
plt.figure(figsize=(10, 6))
plt.plot(tamanos, t_insertion, label="insertion sort")
plt.plot(tamanos, t_bubble, label="bubble sort")
plt.plot(tamanos, t_selection, label="selection sort")
plt.plot(tamanos, t_merge, label="merge sort")
plt.legend()
plt.title("Comparación algoritmos de ordenamiento")
plt.ylabel("Tiempo (segundos)")
plt.xlabel("Tamaño de lista")
plt.show()
```



Se puede observar cómo el peor algoritmo termina siendo el bubble sort. Sigue insertion sort con un desempeño inferior al selection sort. Y finalmente encontramos al merge sort con un desempeño de lejos mucho mejor que los demás algoritmos implementados

Técnicas de diseño de algoritmos

En las siguientes secciones desarrollaremos algunas técnicas de diseño de algoritmos con un ejemplo de problema tipo correspondiente

Divide y Vencerás

Se desarrollará la solución al juego de las Torres de Hanoi por medio de la técnica Divide y Vencerás

```
def torres_hanoi(n, desde, hasta):
  if n==1:
    print("lleva la ficha ", desde, "hasta", hasta)
  else:
    torres_hanoi(n-1, desde, 6-desde-hasta)
    print("lleva la ficha ", desde, "hasta", hasta)
    torres_hanoi(n-1, 6-desde-hasta, hasta)


# solucionar el juego de 3 torres, llevándolas desde la 1 hasta la 3
torres_hanoi(3, 1, 3)
```

```
 lleva la ficha  1 hasta 3
 lleva la ficha  1 hasta 2
 lleva la ficha  3 hasta 2
 lleva la ficha  1 hasta 3
 lleva la ficha  2 hasta 1
 lleva la ficha  2 hasta 3
 lleva la ficha  1 hasta 3
```

Algoritmos voraces - técnica voraz

Solución del problema de cambio monetario por medio de la técnica voraz

```
def cambio_monedas(cantidad, sistema):
  solucion = []
  cantidad_faltante = cantidad
  for moneda in sistema:
    cant_monedas = cantidad_faltante//moneda
    solucion.append(cant_monedas)
    cantidad_faltante -= cant_monedas*moneda
  return solucion


cambio_monedas(37, [25, 10, 5, 1])
```

```
[1, 1, 0, 2]
```

Backtracking y DFS (depth first search)

Problema de las reinas en un tablero de nxn fue resuelto en clase por medio del método de back tracking. En esta sección lo abordaré también por el método de DFS

```
# Funciones para imprimir
def print_solucion_reinas(solucion):
  for sol in solucion:
    linea = ["#" for i in range(len(solucion))]
    linea[sol] = "R"
    print(" ".join(linea))

def print_soluciones_reinas(soluciones):
  for i, solucion in enumerate(soluciones):
    print(f"solución {i+1}")
    print_solucion_reinas(solucion)
    print()
```

Solución por método de back tracking

Esta solución está en base 1. Es decir que las filas se tratarán desde 1 hasta N

```
def es_prometedora(solucion, etapa):
  for i in range(etapa+1):

    # verificar si hay 2 reinas en la misma fila
    if solucion.count(solucion[i]) > 1:
      return False

    # verificar diagonales
    for j in range(i+1, etapa+1):
      if abs(i-j) == abs(solucion[i]- solucion[j]):
        return False

  return True

# solución del problema de N reinas por método de backtracking
def reinas_bt(N, solucion, etapa):
  for i in range(1, N+1):
    solucion[etapa] = i

    if es_prometedora(solucion, etapa):
```

```
    if es_prometedora(solucion, etapa):
      if etapa == N-1:
        print("La solución es:")
        print(solucion)

      else:
        reinas_bt(N, solucion, etapa+1)
    solucion[etapa] = 0



reinas_bt(5, [0 for x in range(5)], 0)
```

```
 La solución es:
 [1, 3, 5, 2, 4]
 La solución es:
 [1, 4, 2, 5, 3]
 La solución es:
 [2, 4, 1, 3, 5]
 La solución es:
 [2, 5, 3, 1, 4]
 La solución es:
 [3, 1, 4, 2, 5]
 La solución es:
 [3, 5, 2, 4, 1]
 La solución es:
 [4, 1, 3, 5, 2]
 La solución es:
 [4, 2, 5, 3, 1]
 La solución es:
 [5, 2, 4, 1, 3]
 La solución es:
 [5, 3, 1, 4, 2]
```

Solución por método DFS

Acá se plantea el problema como un grafo. De tal forma que se recorrerá el árbol por medio de la técnica DFS (Depth first search)

Este método está en base 0, es decir que las filas y columnas se tratarán de 0 hasta N-1

```
def get_hijos_candidato(N, candidato):
  etapa = len(candidato)
  posibles = set(range(N))-set(candidato)
  remover = set()
  for hijo in posibles:
    for i, cand in enumerate(candidato):
      if abs(hijo-cand) == abs(etapa-i):
        remover.add(hijo)
  return list(posibles-remover)
```

```
def reinas(N):
  soluciones = []
  candidatos = [[i] for i in range(N)]
  stack_candidatos = candidatos

  # DFS
  while(stack_candidatos):
    candidato = stack_candidatos.pop(-1)
    if len(candidato) == N:
      soluciones.append(candidato)
    else:
      hijos_candidato = get_hijos_candidato(N, candidato)
      for hijo in hijos_candidato:
        stack_candidatos.append(candidato+[hijo])
  return soluciones



print_soluciones_reinas(reinas(5))
```

```
 solución 1
 # # # # R
 # # R # #
 R # # # #
 # # # R #
 # R # # #

 solución 2
 # # # # R
 # R # # #
 # # # R #
 R # # # #
 # # R # #

 solución 3
 # # # R #
 # R # # #
 # # # # R
 # # R # #
 R # # # #

 solución 4
 # # # R #
 R # # # #
 # # R # #
 # # # # R
 # R # # #

 solución 5
 # # R # #
 # # # # R
 # R # # #
 # # # R #
 R # # # #
```

```
solución 6
# # R # #
R # # # #
# # # R #
# R # # #
# # # # R

solución 7
# R # # #
# # # # R
# # R # #
R # # # #
# # # R #

solución 8
# R # # #
# # # R #
R # # # #
# # R # #
# # # # R

solución 9
R # # # #
# # # R #
```

Problema opcional:

Encontrar los dos puntos más cercados:

- Dado un conjunto de puntos se trata de encontrar la pareja de puntos más cercanos.
- Guía para aprendizaje:
    - Suponer en 1D
    - Primer intento: fuerza bruta
        - Calcular complejidad: ¿Se puede mejorar?
    - Segundo intento: aplicar divide y vencerás
        - Calcular la complejidad: ¿Se puede mejorar?
    - Extender el algoritmo a 2D
    - Extender el algoritmo a 3D

Pareja de puntos más cercanos en 1D

Fuerza bruta

Primero intentaré encontrar la pareja más cercana por fuerza bruta. Esto se hará por medio de buscar todas las posibles combinaciones.

Si tengo N puntos, tendré N*(N-1)/2 combinaciones, lo cual implica un orden algorítmico O(N^2)

```
def par_mas_cercano_1D_fuerza_bruta(puntos):
  # Esta función retornará: los dos puntos más cercanos y la distancia entre estos
  # todo calculado por el método de fuerza bruta

  # se inicializan los puntos cercanos por defecto en los 2 primeros puntos del arreglo
  cercano_a = puntos[0]
  cercano_b = puntos[1]
  menor_dist = abs(cercano_a-cercano_b)
  N = len(puntos)
  # se revisan todas las combinaciones de puntos para encontrar la distancia menor
  for i in range(N-1):
    for j in range(i+1, N):
      actual_dist = abs(puntos[i]-puntos[j])
      if actual_dist<menor_dist:
        menor_dist = actual_dist
        cercano_a = puntos[i]
        cercano_b = puntos[j]

  return cercano_a, cercano_b, menor_dist


N = 100
puntos = np.array(random.sample(range(1000*N), N)) # N puntos aleatorios enteros entre 0 y 10
print(puntos)
print()
print("puntos más cercanos y distancia mínima:")
par_mas_cercano_1D_fuerza_bruta(puntos)
```

```
[67745 62616 77490 30741 40491  3486 95889 42795  7015 13380 23741 23233
 54552 87070 42155 23012 95817 11114 75603 89396 37609  8189 43868 46410
 70229 76364 39265 31056 83153 11072 67347 80637 57653 76826  8787 32439
 76567 90983 35397 17805 46019 55776    52  7983 52397 46091  9350 84033
 92497  7636 89508 40316 70345 60589 89383 32593 83077 52230 59541 84087
 67996 93466 11640 46292 31061 71838 75521 64508 80038 23926 32478 80517
 56234 58054 62408 44534 40669 97811 24978  9462 10211 61345 48686 26348
 83086 39207 61445 19888 59191 60005 71096 69801 56637 55476  6625 30268
  4884  9576 94061 99787]

puntos más cercanos y distancia mínima:
(31056, 31061, 5)
```

Ordenando el arreglo

Dado que estamos tratando con solo una dimensión, el problema se puede resolver de una manera mucho más eficiente.

Suponiendo un arreglo de números ordenado, encontrar el par de distancia mínima es un problema que se puede resolver en orden O(N)

Ahora, lo que haremos será ordenar primero el arreglo, de tal forma que garanticemos encontrar luego rápidamente la pareja de distancia mínima.

Por lo tanto, esta aproximación tendrá un orden **O(N)+O(ordenar)**. Como usualmente los algoritmos de ordenamiento son orden **O(N*log(N))** entonces el orden final de esta aproximación

```
def par_mas_cercano_1D_ordenando(puntos):
  # Esta función retornará: los dos puntos más cercanos y la distancia entre estos
  # todo calculado por el método de ordenar los puntos primero
  ordenados = sorted(puntos)

  # se inicializan los puntos cercanos por defecto en los 2 primeros puntos del arreglo
  cercano_a = ordenados[0]
  cercano_b = ordenados[1]
  menor_dist = abs(cercano_a-cercano_b)
  N = len(puntos)
  for i in range(N-1):
    actual_dist = abs(ordenados[i]-ordenados[i+1])
    if actual_dist < menor_dist:
      menor_dist = actual_dist
      cercano_a = ordenados[i]
      cercano_b = ordenados[i+1]

  return cercano_a, cercano_b, menor_dist


N = 100
puntos = np.array(random.sample(range(1000*N), N)) # N puntos aleatorios enteros entre 0 y 10
print(puntos)
print()
print("puntos más cercanos y distancia mínima:")
par_mas_cercano_1D_ordenando(puntos)
```

```
[44799 14037 12919 36420 60310 31694 19258 58471 29680 51473 60874 11774
 64319 71394 22827 27338 23638 39305 86809 91679 69437 65638 87816  4822
 31978 60134 46083 19891 93988 13647 99656 53147 39996 96489 96175 81769
 61383 23839 50078 29872 29940 72264 91766 86563 56267 40057 44651 61691
 57613 79228 90320 81588 71818 65056 33793 12638   924 42247  5620 94255
 53247 55461 52129 87960 70928 32931 11451 13040 49635 71478 42196 45672
 87920 52130 59188 96621 53490 79797 87359 80024 70407 60748 84745  8851
 68972 50269 96434 86795 25539 81030 59765 34361 59607 99400 74567 83992
  4623 23570 15271 34376]

puntos más cercanos y distancia mínima:
(52129, 52130, 1)
```

Comparación de ambos métodos

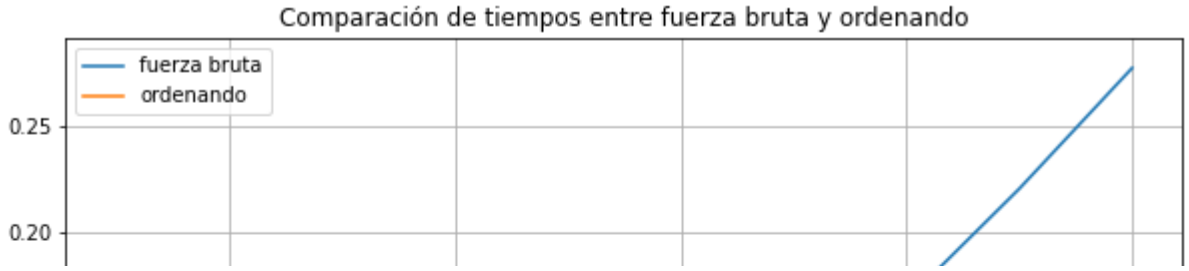
En esta sección se compararán los tiempos de ejecución de ambos métodos para distintos tamaños de entrada

```
tamanos = range(100, 1100, 100) #, 10000, 50000, 100000]


from time import time
tiempos_fb = []
tiempos_sort = []
for tam in tamanos:
  print("#"*8, "tamaño:", tam, "#"*8)

  # se crean los puntos
  puntos = np.array(random.sample(range(1000*tam), tam))
  # print(puntos)
  # cronometrar las funciones
  # print("cronometrando método fuerza bruta")
  t_ini = time()
  sol_fb = par_mas_cercano_1D_fuerza_bruta(puntos)
  t_end = time()
  t_elapsed = t_end-t_ini
  # print("tiempo:", t_elapsed)
  tiempos_fb.append(t_elapsed)

  # print("cronometrando método ordenando")
  t_ini = time()
  sol_sort = par_mas_cercano_1D_ordenando(puntos)
  t_end = time()
  t_elapsed = t_end-t_ini
  # print("tiempo:", t_elapsed)
  tiempos_sort.append(t_elapsed)

  print("solución fuerza bruta: ", sol_fb, "tiempo: ", tiempos_fb[-1])
  print("solución fuerza sort: ", sol_sort, "tiempo: ", tiempos_sort[-1])
  print()
```

```
######## tamaño: 100 ########
solución fuerza bruta:  (7671, 7651, 20) tiempo:  0.003635883331298828
solución fuerza sort:  (7651, 7671, 20) tiempo:  0.00010919570922851562

######## tamaño: 200 ########
solución fuerza bruta:  (136009, 136005, 4) tiempo:  0.01141500473022461
solución fuerza sort:  (136005, 136009, 4) tiempo:  0.00014638900756835938

######## tamaño: 300 ########
solución fuerza bruta:  (77886, 77883, 3) tiempo:  0.029448986053466797
solución fuerza sort:  (69345, 69348, 3) tiempo:  0.0002071857452392578
```

```
######## tamaño: 400 ########
solución fuerza bruta:  (104437, 104431, 6) tiempo:  0.0411992073059082
solución fuerza sort:  (104431, 104437, 6) tiempo:  0.00048232078552246094

######## tamaño: 500 ########
solución fuerza bruta:  (338557, 338552, 5) tiempo:  0.06651806831359863
solución fuerza sort:  (207245, 207250, 5) tiempo:  0.0007925033569335938

######## tamaño: 600 ########
solución fuerza bruta:  (207997, 207996, 1) tiempo:  0.10043740272521973
solución fuerza sort:  (207996, 207997, 1) tiempo:  0.0003948211669921875

######## tamaño: 700 ########
solución fuerza bruta:  (525720, 525718, 2) tiempo:  0.13197708129882812
solución fuerza sort:  (525718, 525720, 2) tiempo:  0.00047969818115234375

######## tamaño: 800 ########
solución fuerza bruta:  (584129, 584140, 11) tiempo:  0.16849255561828613
solución fuerza sort:  (584129, 584140, 11) tiempo:  0.0005462169647216797

######## tamaño: 900 ########
solución fuerza bruta:  (513892, 513891, 1) tiempo:  0.22061371803283691
solución fuerza sort:  (513891, 513892, 1) tiempo:  0.0006105899810791016

######## tamaño: 1000 ########
solución fuerza bruta:  (169351, 169352, 1) tiempo:  0.2768847942352295
solución fuerza sort:  (169351, 169352, 1) tiempo:  0.0011119842529296875
```

```
plt.figure(figsize=(10, 6))
plt.plot(tamanos, tiempos_fb, label="fuerza bruta")
plt.plot(tamanos, tiempos_sort, label="ordenando")
plt.legend()
plt.ylabel("Tiempo (segundos)")
plt.xlabel("Número de puntos")
plt.title("Comparación de tiempos entre fuerza bruta y ordenando")
plt.grid()
plt.show()
```



Nótese cómo el método de fuerza bruta se dispara pues como se mencionó antes es de orden **O(N^2)** mientras que la opción por medio de ordenar la lista antes es orden casi lineal **O(N*log(N))**

Pareja de puntos más cercanos en **2D**


```
import math
import copy


# creación de una clase para representar los puntos 2D
class Point2d():
  def __init__(self, x, y, idf=0):
    self.x = x
    self.y = y
    self.idf = idf # identificador

# función para encontrar la distancia euclídea entre 2 puntos
def dist2d(p1, p2):
  return math.sqrt( (p1.x-p2.x)**2 + (p1.y-p2.y)**2 )
```

Fuerza bruta

Primero intentaré encontrar la pareja más cercana por fuerza bruta. Esto se hará por medio de buscar todas las posibles combinaciones.

Si tengo N puntos, tendré N*(N-1)/2 combinaciones, lo cual implica un orden algorítmico __O(N^2)*__**

```
def fuerza_bruta_2d(puntos):
  cercano_a = puntos[0]
  cercano_b = puntos[1]
  menor_dist = dist2d(cercano_a, cercano_b)
  n = len(puntos)

  for i in range(n-1):
```

```
    for i in range(n-1):
      for j in range(i+1, n):
        actual_dist = dist2d(puntos[i], puntos[j])
        if actual_dist < menor_dist:
          menor_dist = actual_dist
          cercano_a = puntos[i]
          cercano_b = puntos[j]

    return cercano_a, cercano_b, menor_dist
```

```
# probemos el método de fuerza bruta
puntos = [Point2d(2, 3, 1), Point2d(12, 30, 2),
     Point2d(40, 50, 3), Point2d(5, 1, 4),
     Point2d(12, 10, 5), Point2d(3, 4, 6)]


fuerza_bruta_2d(puntos)
```

```
     (<__main__.Point2d at 0x7f22003c34e0>,
      <__main__.Point2d at 0x7f22003c3a58>,
      1.4142135623730951)
```

Divide y vencerás

```
def cercanos_franja(franja, d):
  menor_dist = d


  if len(franja) <= 1:
    return None, None, np.inf


  cercano_a = franja[0]
  cercano_b = franja[1]
  n = len(franja)
  # seleccionar todos los puntos uno por uno
  # intentar el siguiente punto hasta que la diferencia
  # entre las coordenadas y sea menor que d.
  # NOTA: está demostrado que el siguiente loop corre máximo 6 veces
  for i in range(n):
    j = i+1
    while j < n and (franja[j].y-franja[i].y) < menor_dist:
      menor_dist = dist2d(franja[i], franja[j])
      cercano_a = franja[i]
      cercano_b = franja[j]
      j = j+1
```

```
        j += 1
    return cercano_a, cercano_b, menor_dist


# Función recursiva para encontrar
# la pareja de puntos con distancia mínima.
# El arreglo puntos contiene todos los puntos
# ordenados de acuerdo a su coordenada x
def cercanos_rec_2d(puntos, Q):
    n = len(puntos)

    if n<=3:
        return fuerza_bruta_2d(puntos)

    # encontrar el punto del medio
    mid = n//2
    midpoint = puntos[mid]

    # considerar la línea vertical que pasa a través del punto
    # del medio. Calcula la distancia mínima dl a la izquierda del punto mínimo
    # y la distancia mínima dr a la derecha del punto medio
    p1l, p2l, dl = cercanos_rec_2d(puntos[:mid], Q)
    p1r, p2r, dr = cercanos_rec_2d(puntos[mid:], Q)

    # encontrar la menor de ambas distancias:
    if dl < dr:
        p1, p2, d = p1l, p2l, dl
    else:
        p1, p2, d = p1r, p2r, dr

    # construir la franja: un arreglo de puntos que están
    # al rededor del punto medio y contiene puntos cercanos a este
    # a una distancia menor o igual que "d"
    franja = []
    for i in range(n):
        if abs(Q[i].x - midpoint.x) < d:
            franja.append(Q[i])



    # encontrar los puntos más cercanos en la franja y su distancia mínima
    fa, fb, fd = cercanos_franja(franja, d) # franja_a, franja_b son puntos, fd es distancia

    if fd < d:
        return fa, fb, fd
    else:
        return p1, p2, d



# función principal para encontrar los puntos más cercanos en 2d
# usa principalmente la función recursiva cercanos_rec_2d
```

```
def cercanos_2d(puntos):
  puntos.sort(key=lambda punto: punto.x)
  Q = copy.deepcopy(puntos)
  Q.sort(key = lambda point: point.y)

  return cercanos_rec_2d(puntos, Q)


cercanos_2d(puntos)
```

```
     (<__main__.Point2d at 0x7f22003c34e0>,
      <__main__.Point2d at 0x7f22003c3a58>,
      1.4142135623730951)
```

Se puede observar cómo al comparar el método de fuerza bruta y el de divide y vencerás para el mismo conjunto de puntos, arroja el mismo resultado de distancia mínima de **1.4142135...**

Comparación de ambos métodos

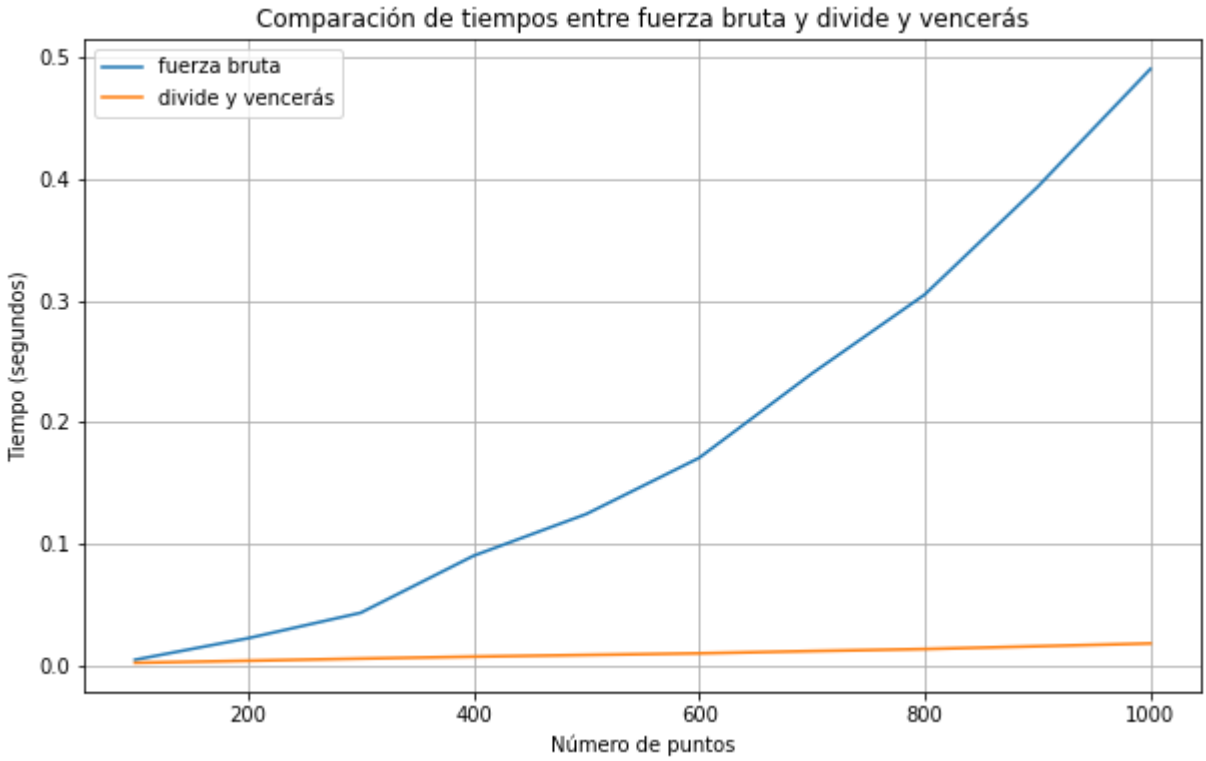
```
tamanos = range(100, 1100, 100) #, 10000, 50000, 100000]


from time import time
tiempos_fb = []
tiempos_div = []
for tam in tamanos:
  print("#"*8, "tamaño:", tam, "#"*8)

  # se crean los puntos
  puntos = [Point2d(random.randint(1, 100*tam), random.randint(-100*tam, 100*tam)) for x in r
  # print(puntos)
  # cronometrar las funciones
  # print("cronometrando método fuerza bruta")
  t_ini = time()
  sol_fb = fuerza_bruta_2d(puntos)
  t_end = time()
  t_elapsed = t_end-t_ini
  # print("tiempo:", t_elapsed)
  tiempos_fb.append(t_elapsed)

  # print("cronometrando método ordenando")
  t_ini = time()
  sol_div = cercanos_2d(puntos)
  t_end = time()
  t_elapsed = t_end-t_ini
  # print("tiempo:", t_elapsed)
  tiempos_div.append(t_elapsed)
```

```
    tiempos_div.append(t_elapsed)

    print("solución fuerza bruta: ", "tiempo: ", tiempos_fb[-1])
    print("solución fuerza div: ", "tiempo: ", tiempos_div[-1])
    print()
```

```
######## tamaño: 100 ########
solución fuerza bruta:  tiempo:  0.004236459732055664
solución fuerza div:  tiempo:  0.0017507076263427734

######## tamaño: 200 ########
solución fuerza bruta:  tiempo:  0.021922588348388672
solución fuerza div:  tiempo:  0.0034389495849609375

######## tamaño: 300 ########
solución fuerza bruta:  tiempo:  0.04288530349731445
solución fuerza div:  tiempo:  0.005209922790527344

######## tamaño: 400 ########
solución fuerza bruta:  tiempo:  0.0897979736328125
solución fuerza div:  tiempo:  0.0068204402923583984

######## tamaño: 500 ########
solución fuerza bruta:  tiempo:  0.12418007850646973
solución fuerza div:  tiempo:  0.008207082748413086

######## tamaño: 600 ########
solución fuerza bruta:  tiempo:  0.1703782081604004
solución fuerza div:  tiempo:  0.009531974792480469

######## tamaño: 700 ########
solución fuerza bruta:  tiempo:  0.23984479904174805
solución fuerza div:  tiempo:  0.0113983154296875

######## tamaño: 800 ########
solución fuerza bruta:  tiempo:  0.30483007431030273
solución fuerza div:  tiempo:  0.013063669204711914

######## tamaño: 900 ########
solución fuerza bruta:  tiempo:  0.3934025764465332
solución fuerza div:  tiempo:  0.015379905700683594

######## tamaño: 1000 ########
solución fuerza bruta:  tiempo:  0.4904484748840332
solución fuerza div:  tiempo:  0.017663002014160156
```

```
plt.figure(figsize=(10, 6))
plt.plot(tamanos, tiempos_fb, label="fuerza bruta")
plt.plot(tamanos, tiempos_div, label="divide y vencerás")
plt.legend()
plt.ylabel("Tiempo (segundos)")
plt.xlabel("Número de puntos")
plt.title("Comparación de tiempos entre fuerza bruta y divide y vencerás")
plt.grid()
```

```
    plt.show()
```


Comparación de tiempos entre fuerza bruta y divide y vencerás
fuerza bruta
divide y vencerás
Tiempo (segundos)
0.0
0.1
0.2
0.3
0.4
0.5
200
400
600
800
1000
Número de puntos