

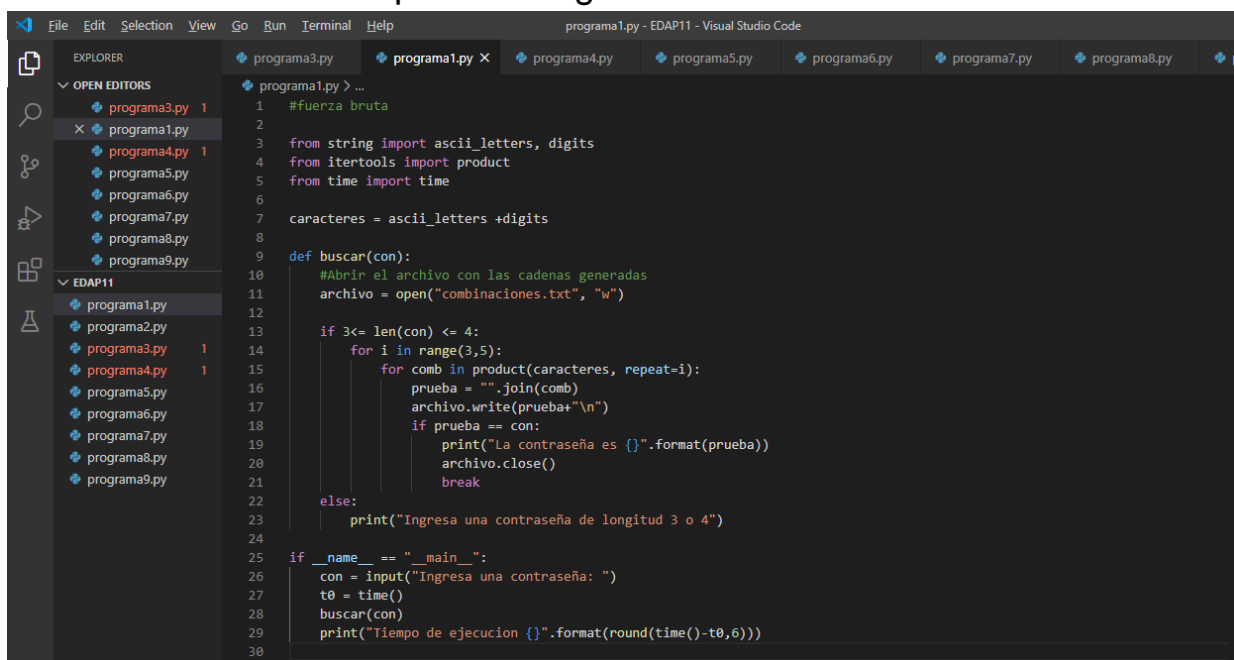
# Práctica 11

## Introducción

En esta práctica se verán algunas estrategias al momento de la creación de algoritmos

### ➤ Desarrollo (con ejercicios)

- En el primer ejercicio se vio la estrategia de fuerza bruta los cuales realizan una búsqueda exhaustiva, el cual busca una contraseña de 4 caracteres ingresados y muestra el tiempo que tarde en ejecutar por completo el programa hasta encontrar la similitud además de que se creaba un archivo de texto en el que se van guardando todas las combinaciones



```
File Edit Selection View Go Run Terminal Help
programa1.py - EDAP11 - Visual Studio Code

EXPLORER
OPEN EDITORS
  programa3.py 1
  programa1.py x
  programa4.py 1
  programa5.py
  programa6.py
  programa7.py
  programa8.py
  programa9.py
EDAP11
  programa1.py
  programa2.py
  programa3.py 1
  programa4.py 1
  programa5.py
  programa6.py
  programa7.py
  programa8.py
  programa9.py

programa1.py > ...
1 #fuerza bruta
2
3 from string import ascii_letters, digits
4 from itertools import product
5 from time import time
6
7 caracteres = ascii_letters + digits
8
9 def buscar(con):
10     #Abrir el archivo con las cadenas generadas
11     archivo = open("combinaciones.txt", "w")
12
13     if 3 <= len(con) <= 4:
14         for i in range(3,5):
15             for comb in product(caracteres, repeat=i):
16                 prueba = "".join(comb)
17                 archivo.write(prueba+"\n")
18                 if prueba == con:
19                     print("La contraseña es {}".format(prueba))
20                     archivo.close()
21                     break
22             else:
23                 print("Ingresa una contraseña de longitud 3 o 4")
24
25 if __name__ == "__main__":
26     con = input("Ingresa una contraseña: ")
27     t0 = time()
28     buscar(con)
29     print("Tiempo de ejecucion {}".format(round(time()-t0,6)))
30
```

- En el segundo caso vimos los algoritmos greedy o también llamados ávidos la ventaja en estos es que una vez considerada una decisión no vuelve a considerar y el programa utilizado en el cual se va regresar cambio de una denominación dada, así como la denominación de las distintas monedas que hay para realizar el cambio

The screenshot shows the Visual Studio Code interface with the file explorer on the left displaying a project named 'EDAP11' containing files from programa1.py to programa9.py. The main editor window shows 'programa2.py' with the following Python code:

```
1 #Algoritmos ávidos-greedy
2 #una vez que se ha ejecutado una decisión, ya no se vuelve a mencionar.
3 def cambio(cantidad, monedas):
4     resultado = []
5     while cantidad > 0:
6         if cantidad >= monedas[0]:
7             num = cantidad // monedas[0]
8             cantidad = cantidad - (num * monedas[0])
9             resultado.append([monedas[0], num])
10            monedas = monedas[1:]
11        return resultado
12
13 if __name__ == "__main__":
14     print(cambio(1000, [20, 10, 5, 2, 1]))
15     print(cambio(20, [20, 10, 5, 2, 1]))
16     print(cambio(30, [20, 10, 5, 2, 1]))
17     print(cambio(98, [5, 20, 1, 50]))
18
```

- Después de eso vimos la estrategia de programación dinámica (bottom-up) en el cual el programa realizado calcula la sucesión de fibonacci en la posición n dada por el usuario

The screenshot shows the Visual Studio Code interface with the file explorer on the left displaying the 'EDAP11' project. The main editor window shows 'programa3.py' with the following Python code:

```
1 #programacion dinamica
2 #Se resuelve un problema a partir de subproblemas que ya han sido resueltos.
3
4 def fibo(numero):
5     a = 1
6     b = 1
7     c = 0
8     for i in range(1, numero-1):
9         c = a + b
10        a = b
11        b = c
12    return c
13
14 def fibo2(numero):
15     a = 1
16     b = 1
17     c = 0
18     for i in range(1, numero-1):
19         a, b = b, a+b
20     return b
21
22 def fibo_bottom_up(numero):
23     fib_parcial = [1, 1, 2]
24     while len(fib_parcial) < numero:
25         fib_parcial.append(fib_parcial[-1] + fib_parcial[-2])
26     print(fib_parcial)
27     return fib_parcial[numero-1]
28
29 f = fibo_bottom_up(5)
30 print(f)
31
```

- En el cuarto es la estrategia descendente-top-down en esto también se calcula serie de fibonacci pero además usando un diccionario que guarda los resultados ya calculados para que no se repitan

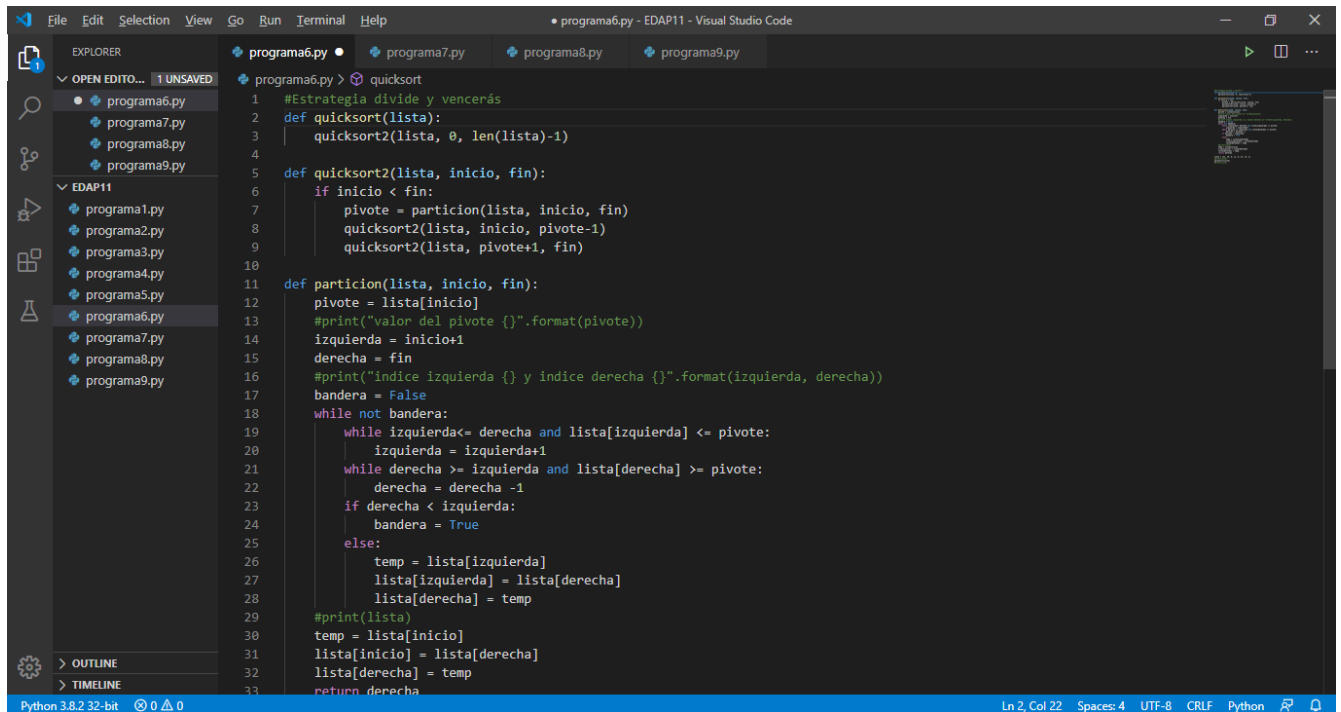
```
1 #Estrategia descendente o top-down
2 #guardar los resultados previamente calculados, haciendo
3 # que no se tengan que volver a hacer las operaciones
4
5 memoria = {1:1, 2:1, 3:2}
6
7 def fibo(numero):
8     a = 1
9     b = 1
10    for i in range(1, numero-1):
11        a, b = b, a+b
12    return b
13
14 def fibo_top_down(numero):
15     if numero in memoria:
16         return memoria[numero]
17     f = fibo(numero-1) + fibo(numero-2)
18     memoria[numero] = f
19     return memoria[numero]
20
21 print(fibo_top_down(5))
22 print(memoria)
23
24 print(fibo_top_down(4))
25 print(memoria)
26
27
```

- En el quinto se usa la estrategia incremental que vimos con una función insertsort que ordena números separándolos en 2 arreglos una parte ordenada y la parte principal partiendo de suponer que el primer elemento está ordenado

```
4 #sublista de números ordenados empezando por las
5 #primeras localidades de la lista.
6 """
7 21 10 12 0 34 15
8 Parte ordenada
9 21                10 12 0 34 15
10 10 21             12 0 34 15
11 10 12 21          0 34 15
12 0 10 12 21        34 15
13 0 10 12 21 34     15
14 0 10 12 15 21 34
15 """
16
17 def insertSort(lista):
18     for index in range(1, len(lista)):
19         actual = lista[index]
20         posicion = index
21         #print("valor a ordenar {}".format(actual))
22         while posicion>0 and lista[posicion-1]>actual:
23             lista[posicion] = lista[posicion-1]
24             posicion = posicion-1
25         lista[posicion] = actual
26         #print(lista)
27         #print()
28     return lista
29
30 lista = [21, 10, 12, 0, 34, 15]
31 #print(lista)
32 insertSort(lista)
33 #print(lista)
34
```

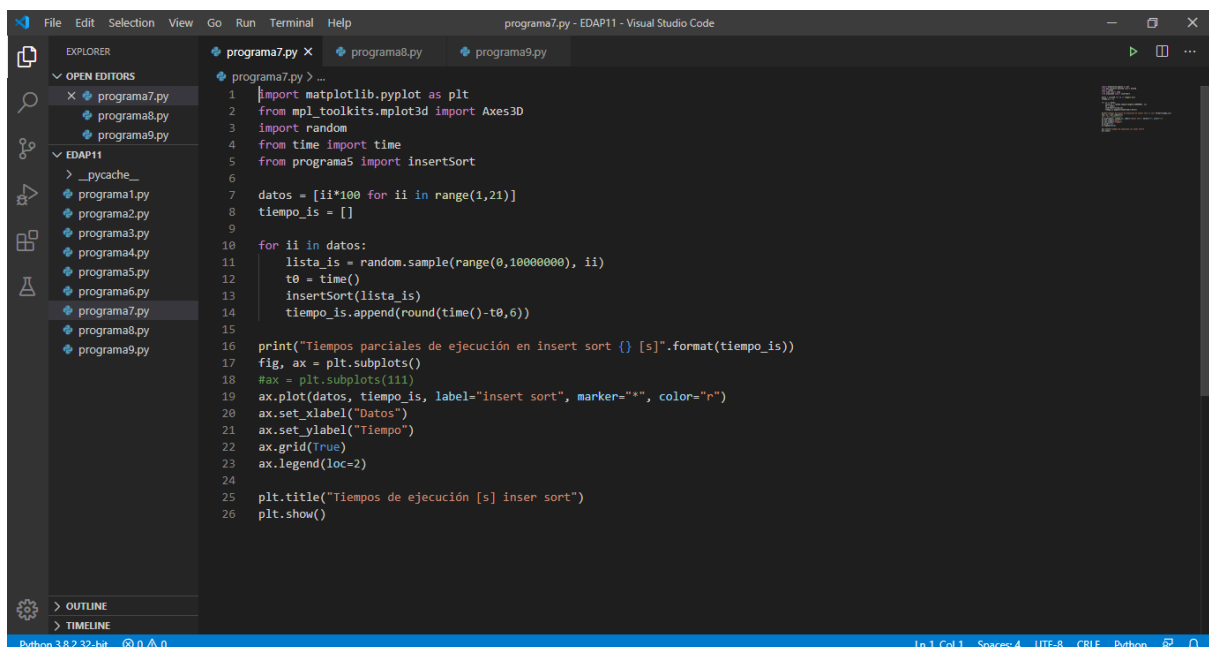
- En el sexto se usa la estrategia de divide y vencerás que ya habíamos observado en anteriores casos de ordenamiento

En este caso usamos el algoritmo de quicksort, el cual tiene como función ordenar números. Se divide, tal como su nombre lo indica, en 2 partes mientras se llama recursivamente para que ordene las partes divididas. Luego selecciona un pivote que servirá para ordenar los demás alrededor de este

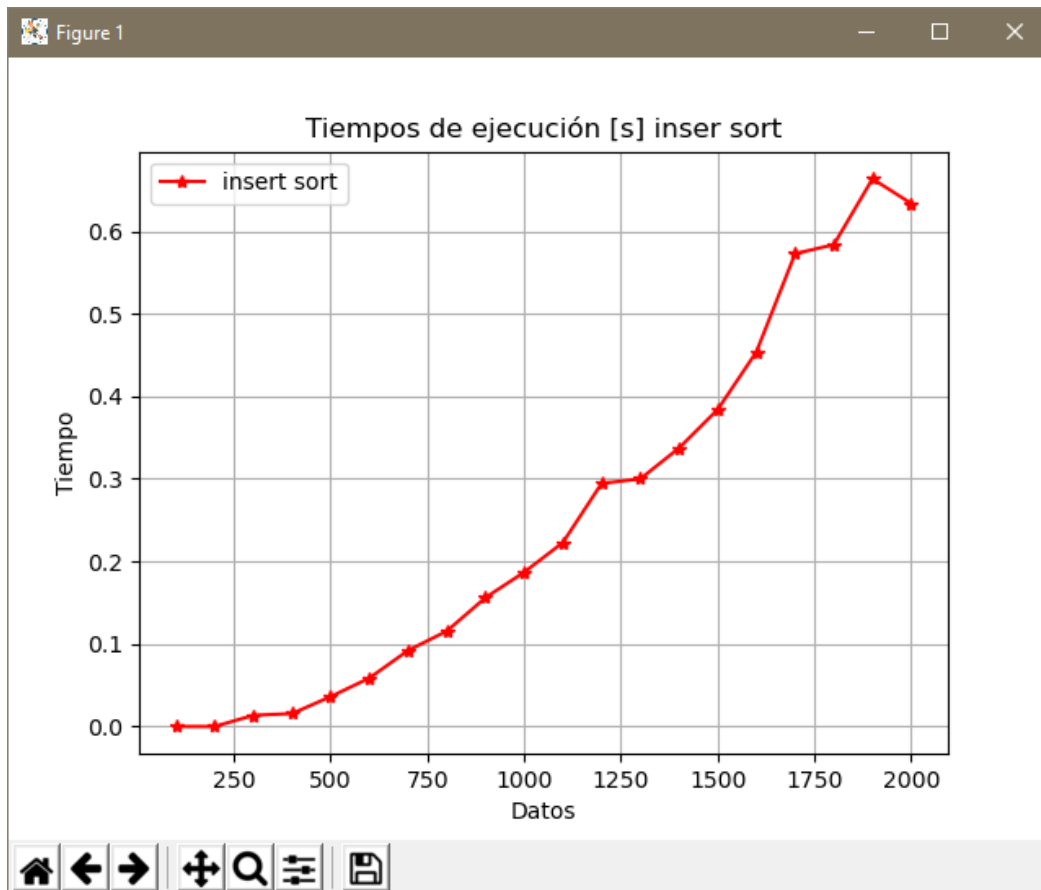


```
programa6.py • quicksort
1 #Estrategia divide y vencerás
2 def quicksort(lista):
3     quicksort2(lista, 0, len(lista)-1)
4
5 def quicksort2(lista, inicio, fin):
6     if inicio < fin:
7         pivote = particion(lista, inicio, fin)
8         quicksort2(lista, inicio, pivote-1)
9         quicksort2(lista, pivote+1, fin)
10
11 def particion(lista, inicio, fin):
12     pivote = lista[inicio]
13     #print("valor del pivote {}".format(pivote))
14     izquierda = inicio+1
15     derecha = fin
16     #print("índice izquierda {} y índice derecha {}".format(izquierda, derecha))
17     bandera = False
18     while not bandera:
19         while izquierda <= derecha and lista[izquierda] <= pivote:
20             izquierda = izquierda+1
21         while derecha >= izquierda and lista[derecha] >= pivote:
22             derecha = derecha -1
23         if derecha < izquierda:
24             bandera = True
25         else:
26             temp = lista[izquierda]
27             lista[izquierda] = lista[derecha]
28             lista[derecha] = temp
29     #print(lista)
30     temp = lista[inicio]
31     lista[inicio] = lista[derecha]
32     lista[derecha] = temp
33     return derecha
```

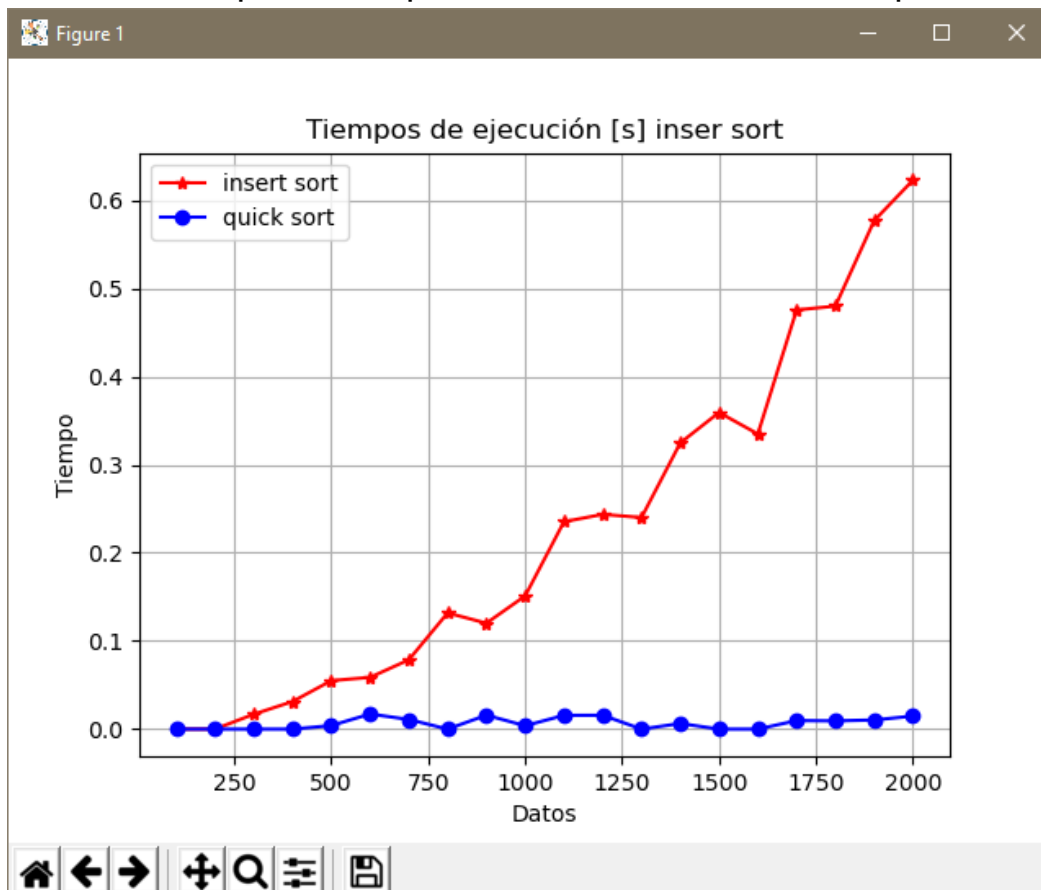
- Ahora se ejecutan algunos programas que grafican y miden el tiempo de ejecución de las funciones de ordenamiento ya vistas: Este es la de insertsort



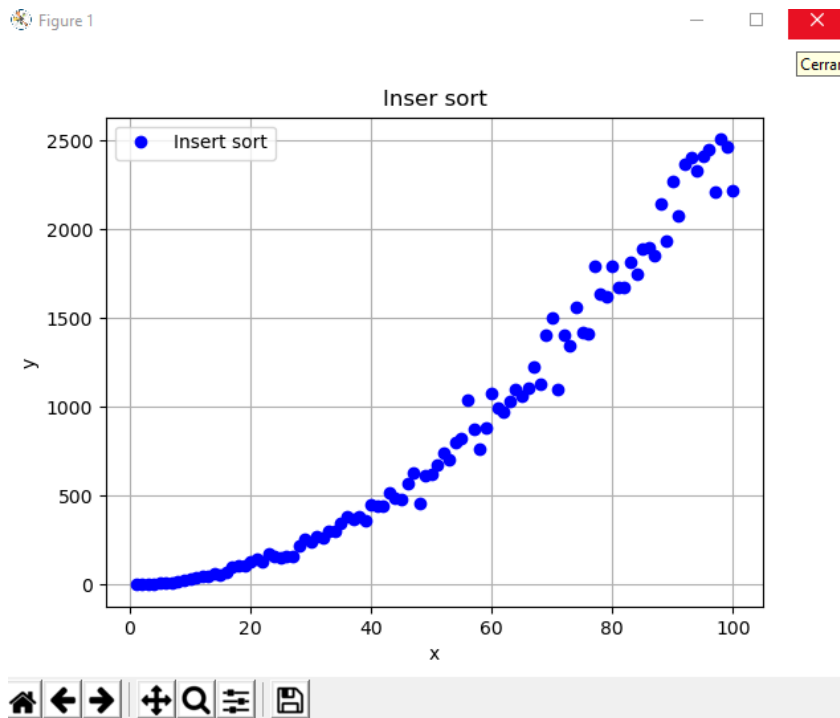
```
programa7.py X • programa8.py • programa9.py
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
3 import random
4 from time import time
5 from programa5 import insertSort
6
7 datos = [ii*100 for ii in range(1,21)]
8 tiempo_is = []
9
10 for ii in datos:
11     lista_is = random.sample(range(0,10000000), ii)
12     t0 = time()
13     insertSort(lista_is)
14     tiempo_is.append(round(time()-t0,6))
15
16 print("Tiempos parciales de ejecución en insert sort {} [s]".format(tiempo_is))
17 fig, ax = plt.subplots()
18 #ax = plt.subplots(111)
19 ax.plot(datos, tiempo_is, label="insert sort", marker="x", color="r")
20 ax.set_xlabel("Datos")
21 ax.set_ylabel("Tiempo")
22 ax.grid(True)
23 ax.legend(loc=2)
24
25 plt.title("Tiempos de ejecución [s] inser sort")
26 plt.show()
```



○ Aquí se comparan tanto insertsort, como quicksort



- El último programa también grafica solo que esta vez contabiliza las veces que se ejecuta una función en insertsort



## ➤ Conclusión

- **Caballero Hernandez Juan Daniel**

Los objetivos de esta práctica se cumplieron bastante bien ya que vimos algunas estrategias de programación además de que vimos cual nos conviene en distintos casos, cual es más eficiente y esto lo pudimos ver reflejado en las gráficas que hicimos utilizando nuevamente matplotlib.