Práctica 2. Optimización y Complejidad

Daniel Fernández Martínez

17 de mayo de 2021

Resumen

El objetivo de esta segunda práctica es estudiar dos algoritmos de ordenación de vectores: el InsertionSort y el QuickSort. Consideraremos en primer lugar un vector de N componentes con valores aleatorios distribuidos según una distribución normal de media nula y varianza unidad, para más tarde comprobar las leyes que rigen la media de los tiempos de ejecución de cada algoritmo.

1. Algoritmo InsertionSort.

Dado que se nos proporcionó el código de ambos algoritmos antes de la realización de la práctica, el valor de la misma reside en el estudio realizado para comprobar las leyes que verifican la media de los tiempos de ejecución. Por ello, se ha hecho uso de algoritmos ya programados por otros usuarios en Python. El siguiente hace referencia al algoritmo InsertionSort, el cual se ha obtenido de la siguiente dirección: `https://www.geeksforgeeks.org/python-program-for-insertion-sort/`. Además, se ha creado también la función `func1()` que se empleará para realizar el ajuste más adelante.

```
# InsertionSort Algorithm
def insertSort(array):

    # Traverse through 1 to len(arr)
    for i in range(1, len(array)):

        key = array[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < array[j] :
                array[j+1] = array[j]
                j -= 1
        array[j+1] = key
    return array

# Polynomial function for insertsort fit
def func1(N, a,b,c):
  return a + b*N + c*N**2
```

1.1. Generar 10000 vectores de N componentes. Sobre estos vectores calcular el tiempo de CPU necesario para el algoritmo anterior. Calcular la media de estos 10000 tiempos. Comprobar (tal como se dedujo en clase) que la media de los tiempos para InsertSort verifica la siguiente ley: $t(N) = c_1 + c_2 N + c_3 N^2$.

Tal y como indica el enunciado, se han generado 10000 vectores de $[10, 20, ..., 130, 140]$ componentes, 14 tamaños distintos en total, todos ellos con media 0 y varianza unidad, haciendo uso de la función `np.random.normal()` del paquete `numpy`. Por cada longitud, se computa la media de los 10000 tiempos obtenidos tras la ordenación de esos vectores, cuyo resultando se almacena en la lista `time_N`. A continuación, se crea una gráfica que representa el último vector ordenado, tal y como se muestra en la Figura 1. Seguidamente, para realizar el ajuste con la ley potencial se hace uso de la función `curve_fit()` del paquete `scipy`, que devuelve los parámetros óptimos del ajuste y la matriz de covarianza de los mismos, que resultan ser: $c_1 = 4{,}35e-06$, $c_2 = 2{,}60e-07$, $c_3 = 7{,}12e-08$. Por último, calculando la raíz cuadrada de las componentes de la diagonal principal de la matriz de covarianza de los parámetros, se obtienen las bandas de confianza con una semianchura de 3 desviaciones típicas respecto de la media. En las Figuras 2 y 3 se muestran los resultados del ajuste sin (y con) las bandas de confianza. Todo ello se ha programado de la siguiente manera:

```
if mode == 0:
    # Study of the InsertionSort Algorithm
    """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
     time needed by the previous two algorithms to order them. Compute the average
     of these times."""
    num_arr = 10000
    time_N = []
    k = 0
    for i in range(10, 150, 10):
        v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors

        totalTime = []
        for j in range(len(v)):
            start = time.time()
            v_insertion = insertSort(v[j])
            end = time.time()
            totalTime.append(end-start)

        time_N.append(np.mean(totalTime))
        print('%s seconds needed for InsertionSort algorithm ' % time_N[k] + '(N = %d)'% i)
        k = k+1

    plt.plot(v_insertion)
    plt.title('Last vector sorted with InsertionSort')
    plt.xlabel('Vector index')
    plt.ylabel('Value')
    plt.tight_layout()
    plt.savefig('insertsorted.png', dpi=300)
    plt.show()

    """ Check that the average time obtained with the InsertSort algorithm
    verifies the following law: t(N) = c1 + c2N + c3N^2 """
    x_data = np.linspace(10, 150, len(time_N))
    plt.plot(x_data, time_N, label='true curve')
    plt.title('InsertSort: $t(N)=c_1+c_2N+c_3N^2$')

    "Fit of time_N"
    # Model fit
    popt, pcov = curve_fit(func1, x_data, time_N)
    a = popt[0]
    b = popt[1]
    c = popt[2]
    print(a, b, c)

    # Plot curve
    fit = func1(x_data,a,b,c)
    plt.plot(x_data, fit, label='fit')
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.legend()
    plt.tight_layout()
    plt.savefig('insertfit.png', dpi=300)
    plt.show()
    "Errors"
    perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
    nstd = 3
    popt_up = popt + nstd*perr
    popt_dw = popt - nstd*perr

    fit_up = func1(x_data, *popt_up)
    fit_dw = func1(x_data, *popt_dw)

    plt.plot(x_data, time_N, 'k', label='true curve')
    plt.plot(x_data, fit, 'r',  label='best fit')
    plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3-$\sigma$ interval')
    plt.title('InsertSort: $t(N)=c_1+c_2N+c_3N^2$')
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.legend(loc='upper left')
    plt.tight_layout()
    plt.savefig('insertfit_interval.png', dpi=300)
    plt.show()
```

Tras la ejecución, se obtienen los siguientes resultados:

```
1.1965131759643555e-05 seconds needed for InsertionSort algorithm (N = 10)
4.019227027893066e-05 seconds needed for InsertionSort algorithm (N = 20)
8.477299213409423e-05 seconds needed for InsertionSort algorithm (N = 30)
0.00014272098541259765 seconds needed for InsertionSort algorithm (N = 40)
0.00021801767349243165 seconds needed for InsertionSort algorithm (N = 50)
0.00031296896934509275 seconds needed for InsertionSort algorithm (N = 60)
0.0004259755373001099 seconds needed for InsertionSort algorithm (N = 70)
0.0005496341943740844 seconds needed for InsertionSort algorithm (N = 80)
0.0006839749574661254 seconds needed for InsertionSort algorithm (N = 90)
0.000842648959159851 seconds needed for InsertionSort algorithm (N = 100)
0.0010178749561309815 seconds needed for InsertionSort algorithm (N = 110)
0.001210760760307312 seconds needed for InsertionSort algorithm (N = 120)
0.0014221953868865968 seconds needed for InsertionSort algorithm (N = 130)
0.001650188636779785 seconds needed for InsertionSort algorithm (N = 140)
```

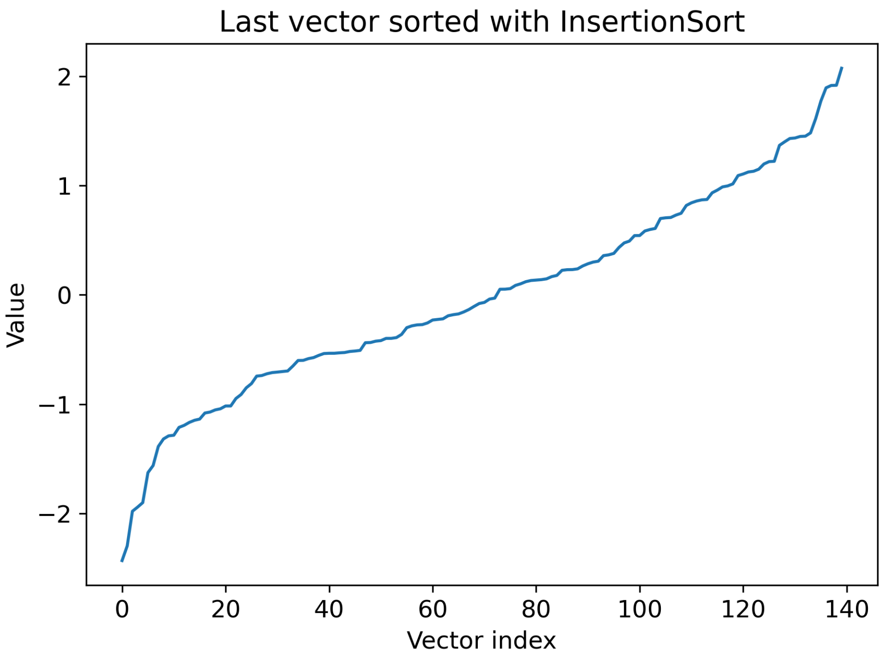

Figura 1

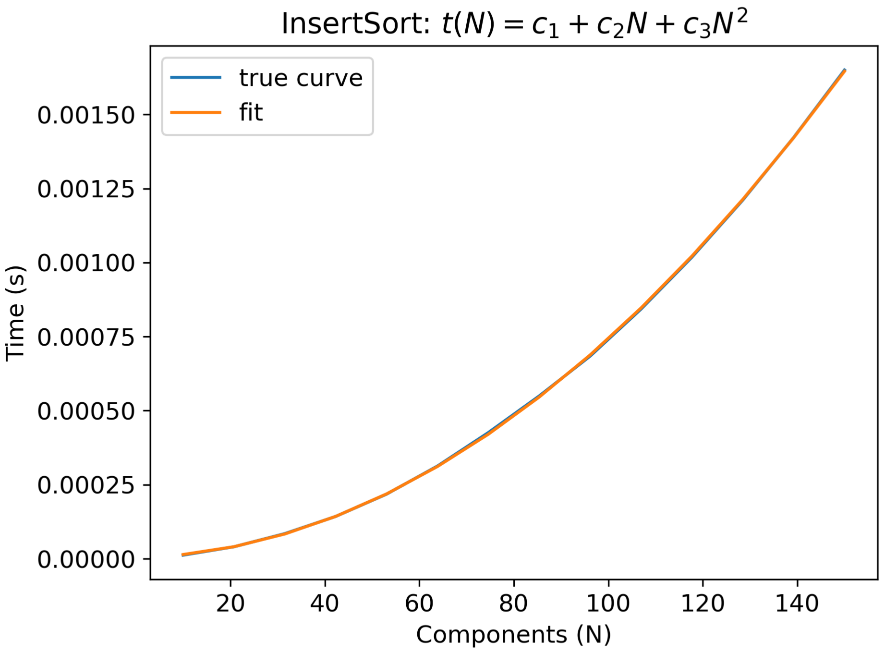

Figura 2

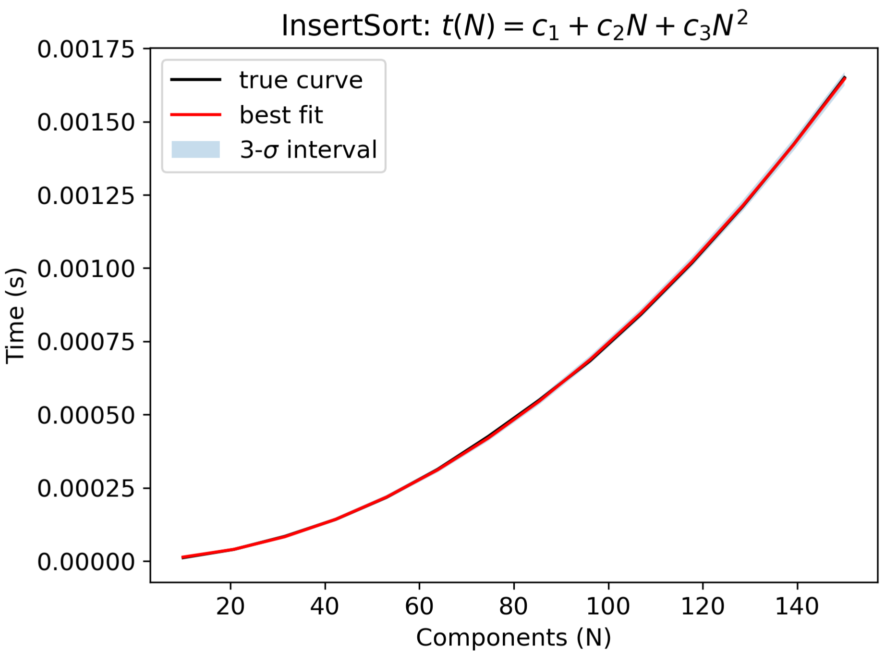

Figura 3

2. Algoritmo QuickSort.

Al igual que antes, se proporciona a continuación la fuente de la que se ha extraído la implementación en Python del algoritmo QuickSort: `https://www.geeksforgeeks.org/python-program-for-quicksort/`. En este caso, se hace uso de una función auxiliar denominada `partition()`, en vez de integrar todo el código en una única rutina. De nuevo, se ha creado en este caso la función `func2()` que se empleará para realizar el ajuste más adelante.

```python
" QuickSort Algorithm"
def partition(arr, low, high):
    i = (low-1)          # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low, high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Function to do Quick sort
def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr, low, high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Logarithmic function for quicksort fit
def func2(N, a,b,c):
  return a + b*N + c*N*np.log(N)
```

2.1. Generar 10000 vectores de N componentes. Sobre estos vectores calcular el tiempo de CPU necesario para el algoritmo anterior. Calcular la media de estos 10000 tiempos. Realizar el mismo análisis para el algoritmo QuickSort, donde se verifica que: $t(N) = a + bN + cNlogN$

Se ha seguido el mismo planteamiento que en el caso del algoritmo anterior. Los resultados obtenidos se encuentran tras el código que se muestra a continuación. Los parámetros óptimos del ajuste en este caso son: $a = 4{,}17e-06$, $b = -1{,}63e-07$ y $c = 5{,}36e-07$.

```python
if mode == 1:
    # Study of the QuickSort Algorithm
    # """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
    #  time needed by the previous two algorithms to order them. Compute the average
    #  of these times."""
    num_arr = 10000
    time_N = []
    k = 0
    for i in range(10, 150, 10):
        v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors

        totalTime = []
```

```python
        for j in range(len(v)):
            start = time.time()
            quickSort(v[j], 0, i-1)
            end = time.time()
            totalTime.append(end-start)

        time_N.append(np.mean(totalTime))
        print('%s seconds needed for QuickSort algorithm ' % time_N[k] + '(N = %d)'% i)
        k = k+1

    plt.plot(v[i])
    plt.title('Last vector sorted with QuickSort')
    plt.xlabel('Vector index')
    plt.ylabel('Value')
    plt.tight_layout()
    plt.savefig('quicksorted.png', dpi=300)
    plt.show()

    """ Check that the average time obtained with the InsertSort algorithm
    verifies the following law: t(N) = a + bN + cNlog(N)"""
    x_data = np.linspace(10, 150, len(time_N))
    plt.plot(x_data, time_N, label='true curve')
    plt.title('QuickSort: $t(N)=a + bN + cN \log{N}$')

    "Fit of time_N"
    # Model fit
    popt, pcov = curve_fit(func2, x_data, time_N)
    a = popt[0]
    b = popt[1]
    c = popt[2]
    print(a, b, c)
    # Plot curve
    fit = func2(x_data,a,b,c)
    plt.plot(x_data, fit, label='fit')
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.legend()
    plt.tight_layout()
    plt.savefig('quickfit.png', dpi=300)
    plt.show()

    "Errors"
    perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
    nstd = 3
    popt_up = popt + nstd*perr
    popt_dw = popt - nstd*perr

    fit_up = func2(x_data, *popt_up)
    fit_dw = func2(x_data, *popt_dw)

    plt.plot(x_data, time_N, 'k', label='true curve')
    plt.plot(x_data, fit, 'r', label='best fit')
    plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3-$\sigma$ interval')
    plt.legend(loc='upper left')
    plt.title('QuickSort: $t(N)=a + bN + cN \log{N}$', )
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.tight_layout()
    plt.savefig('quickfit_interval.png', dpi=300)
    plt.show()
```

```
1.446099281311035e-05 seconds needed for QuickSort algorithm (N = 10)
3.4607791900634764e-05 seconds needed for QuickSort algorithm (N = 20)
5.744564533233643e-05 seconds needed for QuickSort algorithm (N = 30)
8.277754783630372e-05 seconds needed for QuickSort algorithm (N = 40)
0.00010990900993347168 seconds needed for QuickSort algorithm (N = 50)
0.00013573689460754396 seconds needed for QuickSort algorithm (N = 60)
0.00016228137016296388 seconds needed for QuickSort algorithm (N = 70)
0.00019378085136413574 seconds needed for QuickSort algorithm (N = 80)
0.00022270739078521728 seconds needed for QuickSort algorithm (N = 90)
0.0002555163860321045 seconds needed for QuickSort algorithm (N = 100)
```

```
0.0002853411912918091 seconds needed for QuickSort algorithm (N = 110)
0.00031675326824188234 seconds needed for QuickSort algorithm (N = 120)
0.0003530533790588379 seconds needed for QuickSort algorithm (N = 130)
0.00038039159774780275 seconds needed for QuickSort algorithm (N = 140)
4.169973583029417e-06 -1.6298946281122152e-07 5.357823139306962e-07
```

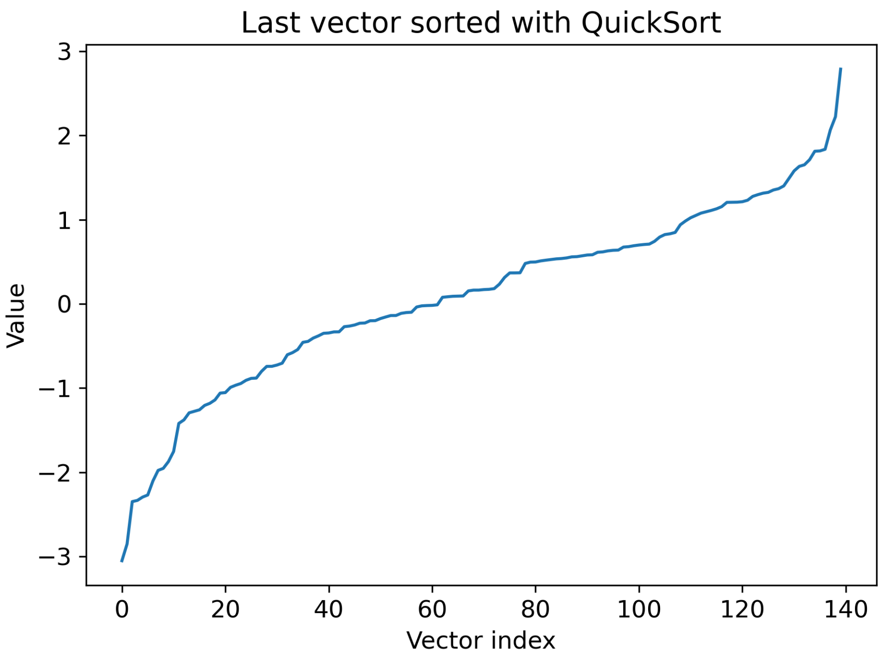

Figura 4

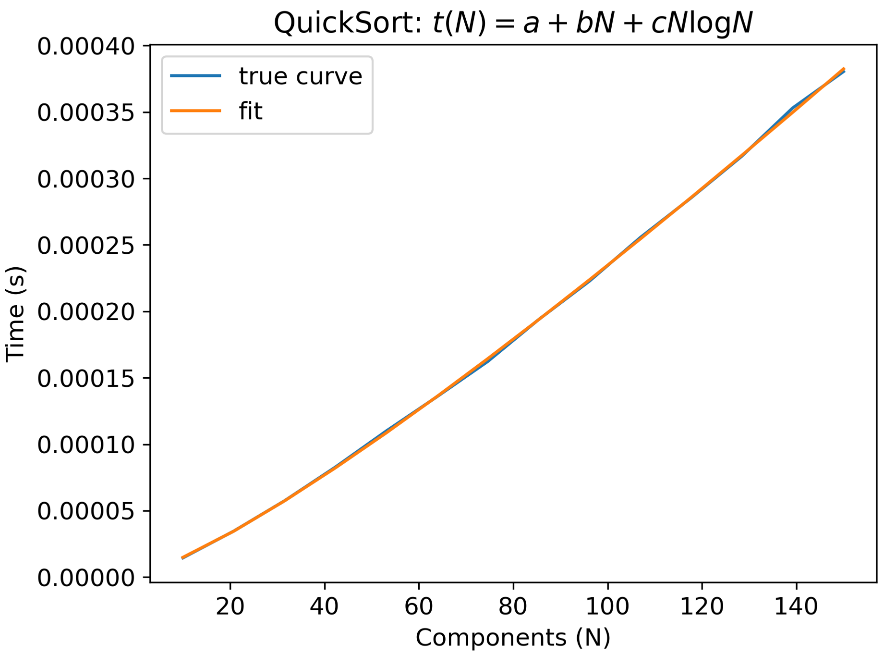

Figura 5

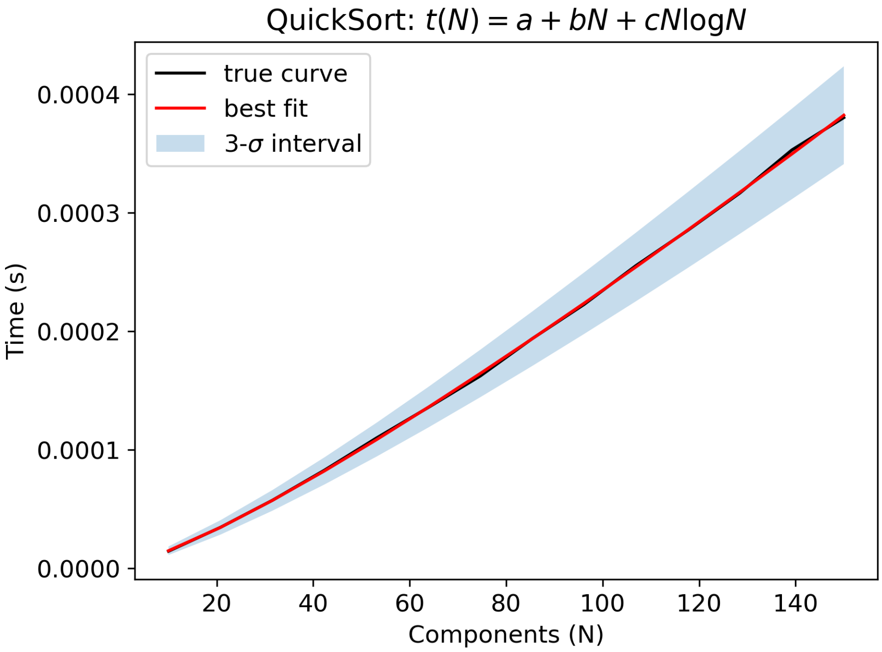

Figura 6

3. Conclusiones

A la vista de los resultados, resulta evidente comprobar que el tiempo de ejecución con el QuickSort es sustancialmente menor. Por ejemplo, para $N = 140$, el algoritmo es 4.34 veces más rápido que el InsertionSort. Respecto al ajuste, todo parece indicar que la desviación del ajuste del algoritmo QuickSort es mucho mayor que la del InsertionSort. Por si resulta de interés, en la siguiente sección se muestra el código completo de la práctica.

4. Código completo

```
"""
Pr ctica 2. Optimizaci n y complejidad.
Daniel Fern ndez Mart nez
"""
import numpy as np
import matplotlib.pyplot as plt
import time
from scipy.optimize import curve_fit

plt.style.use('default')
plt.rcParams.update({'font.size': 12})

" InsertionSort Algorithm"
def insertSort(array):

    # Traverse through 1 to len(arr)
    for i in range(1, len(array)):

        key = array[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < array[j] :
                array[j+1] = array[j]
                j -= 1
        array[j+1] = key
    return array
```

```python

" QuickSort Algorithm"
def partition(arr, low, high):
    i = (low-1)          # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low, high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Function to do Quick sort
def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr, low, high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Polynomial function for insertsort
def func1(N, a,b,c):
  return a + b*N + c*N**2

# Logarithmic function for quicksort
def func2(N, a,b,c):
  return a + b*N + c*N*np.log(N)

mode = 1

if mode == 0:

    # Study of the InsertionSort Algorithm
    """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
     time needed by the previous two algorithms to order them. Compute the average
     of these times."""

    num_arr = 10000
    time_N = []
    k = 0
    for i in range(10, 150, 10):
        v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors

        totalTime = []
        for j in range(len(v)):
            start = time.time()
            v_insertion = insertSort(v[j])
            end = time.time()
            totalTime.append(end-start)

        time_N.append(np.mean(totalTime))
        print('%s seconds needed for InsertionSort algorithm ' % time_N[k] + '(N = %d)'% i)
        k = k+1

    plt.plot(v_insertion)
```

```python
    plt.title('Last vector sorted with InsertionSort')
    plt.xlabel('Vector index')
    plt.ylabel('Value')
    plt.tight_layout()
    plt.savefig('insertsorted.png', dpi=300)
    plt.show()

    """ Check that the average time obtained with the InsertSort algorithm
    verifies the following law: t(N) = c1 + c2N + c3N^2 """
    x_data = np.linspace(10, 150, len(time_N))
    plt.plot(x_data, time_N, label='true curve')
    plt.title('InsertSort: $t(N)=c_1+c_2N+c_3N^2$')

    "Fit of time_N"
    # Model fit
    popt, pcov = curve_fit(func1, x_data, time_N)
    a = popt[0]
    b = popt[1]
    c = popt[2]
    print(a, b, c)

    # Plot curve
    fit = func1(x_data,a,b,c)
    plt.plot(x_data, fit, label='fit')
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.legend()
    plt.tight_layout()
    plt.savefig('insertfit.png', dpi=300)
    plt.show()

    "Errors"
    perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
    nstd = 3
    popt_up = popt + nstd*perr
    popt_dw = popt - nstd*perr

    fit_up = func1(x_data, *popt_up)
    fit_dw = func1(x_data, *popt_dw)

    plt.plot(x_data, time_N, 'k', label='true curve')
    plt.plot(x_data, fit, 'r',  label='best fit')
    plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3-$\sigma$ interval')
    plt.title('InsertSort: $t(N)=c_1+c_2N+c_3N^2$')
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.legend(loc='upper left')
    plt.tight_layout()
    plt.savefig('insertfit_interval.png', dpi=300)
    plt.show()

if mode == 1:
    # Study of the QuickSort Algorithm
    # """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
    #  time needed by the previous two algorithms to order them. Compute the average
    #  of these times."""
    num_arr = 10000
    time_N = []
    k = 0
    for i in range(10, 150, 10):
        v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors

        totalTime = []
        for j in range(len(v)):
            start = time.time()
            quickSort(v[j], 0, i-1)
            end = time.time()
            totalTime.append(end-start)

        time_N.append(np.mean(totalTime))
        print('%s seconds needed for QuickSort algorithm ' % time_N[k] + '(N = %d)'% i)
        k = k+1

    plt.plot(v[i])
```

```
    plt.title('Last vector sorted with QuickSort')
    plt.xlabel('Vector index')
    plt.ylabel('Value')
    plt.tight_layout()
    plt.savefig('quicksorted.png', dpi=300)
    plt.show()

    """ Check that the average time obtained with the InsertSort algorithm
    verifies the following law: t(N) = a + bN + cNlog(N)"""
    x_data = np.linspace(10, 150, len(time_N))
    plt.plot(x_data, time_N, label='true curve')
    plt.title('QuickSort: $t(N)=a + bN + cN \log{N}$')

    "Fit of time_N"
    # Model fit
    popt, pcov = curve_fit(func2, x_data, time_N)
    a = popt[0]
    b = popt[1]
    c = popt[2]
    print(a, b, c)
    # Plot curve
    fit = func2(x_data,a,b,c)
    plt.plot(x_data, fit, label='fit')
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.legend()
    plt.tight_layout()
    plt.savefig('quickfit.png', dpi=300)
    plt.show()

    "Errors"
    perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
    nstd = 3
    popt_up = popt + nstd*perr
    popt_dw = popt - nstd*perr

    fit_up = func2(x_data, *popt_up)
    fit_dw = func2(x_data, *popt_dw)

    plt.plot(x_data, time_N, 'k', label='true curve')
    plt.plot(x_data, fit, 'r', label='best fit')
    plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3-$\sigma$ interval')
    plt.legend(loc='upper left')
    plt.title('QuickSort: $t(N)=a + bN + cN \log{N}$', )
    plt.xlabel('Components (N)')
    plt.ylabel('Time (s)')
    plt.tight_layout()
    plt.savefig('quickfit_interval.png', dpi=300)
    plt.show()
```