

# 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.

```
1 # InsertionSort Algorithm
2 def insertSort(array):
3
4     # Traverse through 1 to len(arr)
5     for i in range(1, len(array)):
6
7         key = array[i]
8
9         # Move elements of arr[0..i-1], that are
10        # greater than key, to one position ahead
11        # of their current position
12        j = i-1
13        while j >= 0 and key < array[j] :
14            array[j+1] = array[j]
15            j -= 1
16        array[j+1] = key
17    return array
18
19 # Polynomial function for insertsort fit
20 def func1(N, a,b,c):
21     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: $t(N) = c_1 + c_2N + c_3N^2$ .

Tal y como indica el enunciado, se han generado 10000 vectores de  $[10, 20, \dots, 130, 140]$  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:

```

22 if mode == 0:
23     # Study of the InsertionSort Algorithm
24     """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
25         time needed by the previous two algorithms to order them. Compute the average
26         of these times."""
27     num_arr = 10000
28     time_N = []
29     k = 0
30     for i in range(10, 150, 10):
31         v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors
32
33         totalTime = []
34         for j in range(len(v)):
35             start = time.time()
36             v_insertion = insertionSort(v[j])
37             end = time.time()
38             totalTime.append(end-start)
39
40         time_N.append(np.mean(totalTime))
41         print('%s seconds needed for InsertionSort algorithm ' % time_N[k] + '(N = %d)' % i)
42         k = k+1
43
44     plt.plot(v_insertion)
45     plt.title('Last vector sorted with InsertionSort')
46     plt.xlabel('Vector index')
47     plt.ylabel('Value')
48     plt.tight_layout()
49     plt.savefig('insertsorted.png', dpi=300)
50     plt.show()
51
52     """ Check that the average time obtained with the InsertSort algorithm
53     verifies the following law:  $t(N) = c_1 + c_2N + c_3N^2$  """
54     x_data = np.linspace(10, 150, len(time_N))
55     plt.plot(x_data, time_N, label='true curve')
56     plt.title('InsertSort:  $t(N)=c_1+c_2N+c_3N^2$ ')
57
58     "Fit of time_N"
59     # Model fit
60     popt, pcov = curve_fit(func1, x_data, time_N)
61     a = popt[0]
62     b = popt[1]
63     c = popt[2]
64     print(a, b, c)
65
66     # Plot curve
67     fit = func1(x_data,a,b,c)
68     plt.plot(x_data, fit, label='fit')
69     plt.xlabel('Components (N)')
70     plt.ylabel('Time (s)')
71     plt.legend()
72     plt.tight_layout()
73     plt.savefig('insertfit.png', dpi=300)
74     plt.show()
75     "Errors"
76     perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
77     nstd = 3
78     popt_up = popt + nstd*perr
79     popt_dw = popt - nstd*perr
80
81     fit_up = func1(x_data, *popt_up)
82     fit_dw = func1(x_data, *popt_dw)
83
84     plt.plot(x_data, time_N, 'k', label='true curve')
85     plt.plot(x_data, fit, 'r', label='best fit')
86     plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3- $\sigma$  interval')
87     plt.title('InsertSort:  $t(N)=c_1+c_2N+c_3N^2$ ')
88     plt.xlabel('Components (N)')
89     plt.ylabel('Time (s)')
90     plt.legend(loc='upper left')
91     plt.tight_layout()
92     plt.savefig('insertfit_interval.png', dpi=300)
93     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)
```

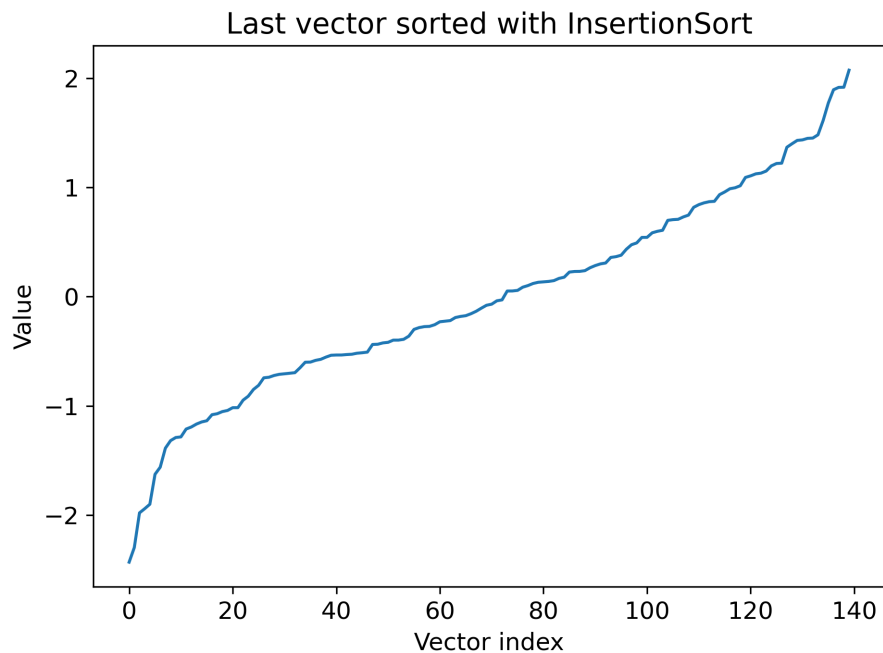


Figura 1

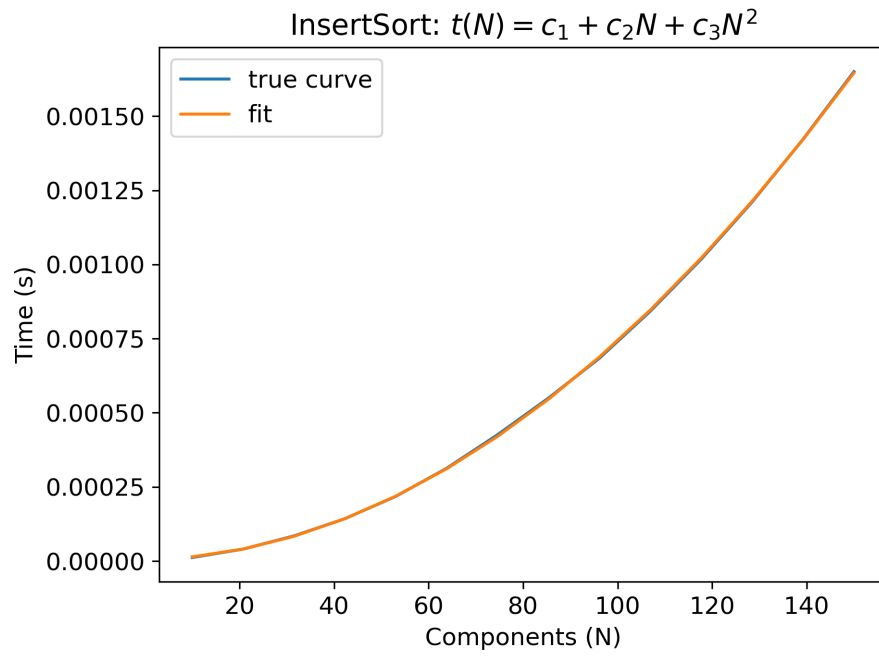


Figura 2

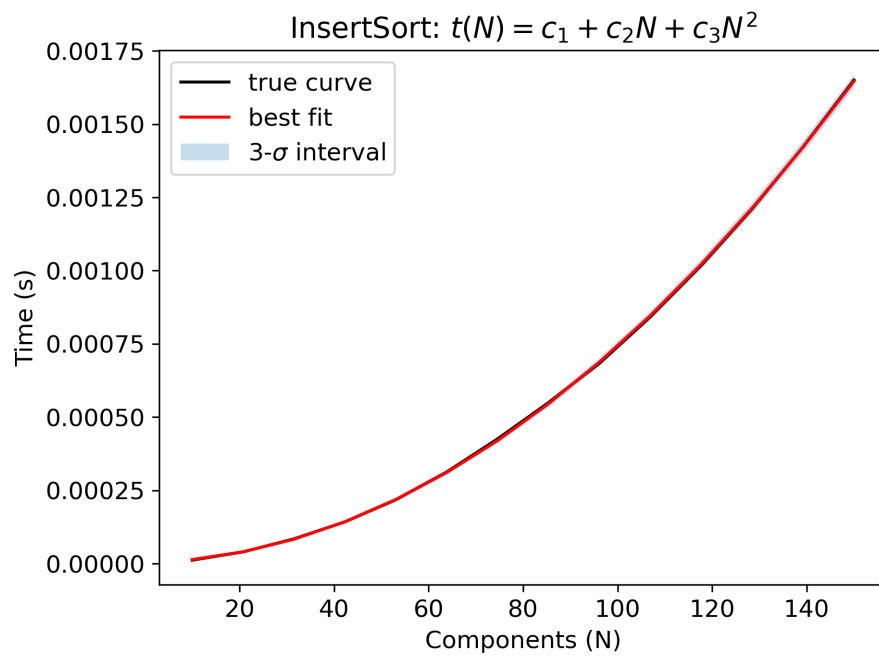


Figura 3

## 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.

```
94 " QuickSort Algorithm"
95 def partition(arr, low, high):
96     i = (low-1)          # index of smaller element
97     pivot = arr[high]    # pivot
98
99     for j in range(low, high):
100
101         # If current element is smaller than or
102         # equal to pivot
103         if arr[j] <= pivot:
104
105             # increment index of smaller element
106             i = i+1
107             arr[i], arr[j] = arr[j], arr[i]
108
109     arr[i+1], arr[high] = arr[high], arr[i+1]
110     return (i+1)
111
112 # The main function that implements QuickSort
113 # arr[] --> Array to be sorted,
114 # low  --> Starting index,
115 # high --> Ending index
116
117 # Function to do Quick sort
118 def quickSort(arr, low, high):
119     if len(arr) == 1:
120         return arr
121     if low < high:
122
123         # pi is partitioning index, arr[p] is now
124         # at right place
125         pi = partition(arr, low, high)
126
127         # Separately sort elements before
128         # partition and after partition
129         quickSort(arr, low, pi-1)
130         quickSort(arr, pi+1, high)
131
132 # Logarithmic function for quicksort fit
133 def func2(N, a,b,c):
134     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: $t(N) = a + bN + cN\log N$

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  $c = 5,36e - 07$ .

```
135 if mode == 1:
136     # Study of the QuickSort Algorithm
137     # """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
138     # time needed by the previous two algorithms to order them. Compute the average
139     # of these times."""
140     num_arr = 10000
141     time_N = []
142     k = 0
143     for i in range(10, 150, 10):
144         v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors
145
146         totalTime = []
```

```

147     for j in range(len(v)):
148         start = time.time()
149         quickSort(v[j], 0, i-1)
150         end = time.time()
151         totalTime.append(end-start)
152
153     time_N.append(np.mean(totalTime))
154     print('%s seconds needed for QuickSort algorithm ' % time_N[k] + '(N = %d)' % i)
155     k = k+1
156
157     plt.plot(v[i])
158     plt.title('Last vector sorted with QuickSort')
159     plt.xlabel('Vector index')
160     plt.ylabel('Value')
161     plt.tight_layout()
162     plt.savefig('quicksorted.png', dpi=300)
163     plt.show()
164
165     """ Check that the average time obtained with the InsertSort algorithm
166     verifies the following law:  $t(N) = a + bN + cN\log(N)$  """
167     x_data = np.linspace(10, 150, len(time_N))
168     plt.plot(x_data, time_N, label='true curve')
169     plt.title('QuickSort:  $t(N)=a + bN + cN \log\{N\}$ ')
170
171     "Fit of time_N"
172     # Model fit
173     popt, pcov = curve_fit(func2, x_data, time_N)
174     a = popt[0]
175     b = popt[1]
176     c = popt[2]
177     print(a, b, c)
178     # Plot curve
179     fit = func2(x_data, a, b, c)
180     plt.plot(x_data, fit, label='fit')
181     plt.xlabel('Components (N)')
182     plt.ylabel('Time (s)')
183     plt.legend()
184     plt.tight_layout()
185     plt.savefig('quickfit.png', dpi=300)
186     plt.show()
187
188     "Errors"
189     perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
190     nstd = 3
191     popt_up = popt + nstd*perr
192     popt_dw = popt - nstd*perr
193
194     fit_up = func2(x_data, *popt_up)
195     fit_dw = func2(x_data, *popt_dw)
196
197     plt.plot(x_data, time_N, 'k', label='true curve')
198     plt.plot(x_data, fit, 'r', label='best fit')
199     plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3- $\sigma$  interval')
200     plt.legend(loc='upper left')
201     plt.title('QuickSort:  $t(N)=a + bN + cN \log\{N\}$ , )
202     plt.xlabel('Components (N)')
203     plt.ylabel('Time (s)')
204     plt.tight_layout()
205     plt.savefig('quickfit_interval.png', dpi=300)
206     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

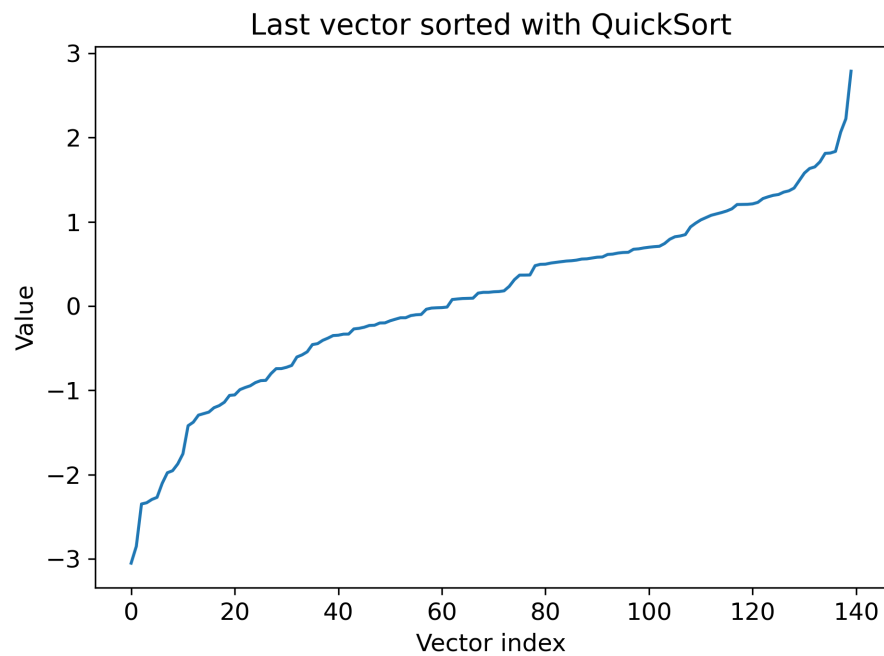


Figura 4

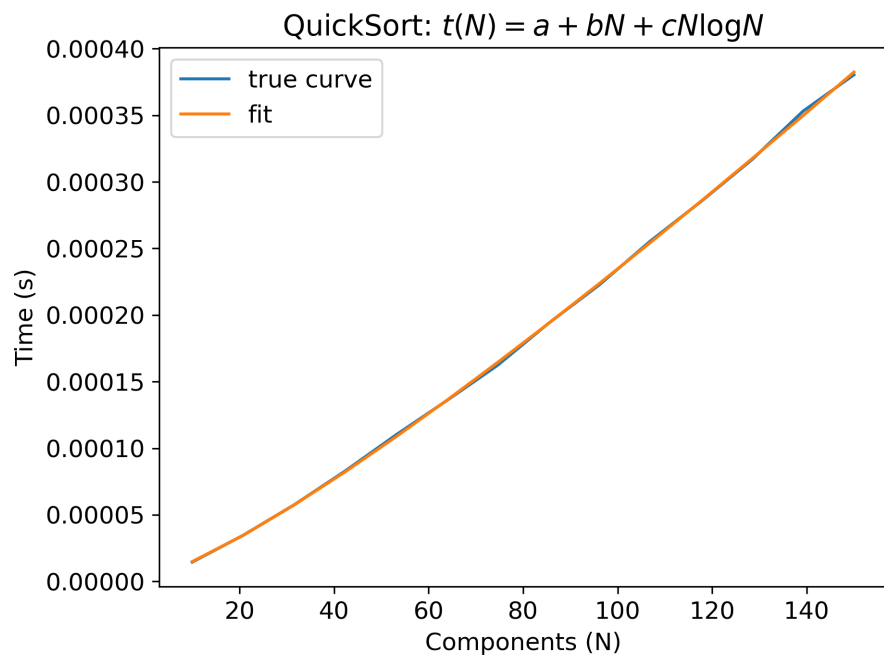


Figura 5

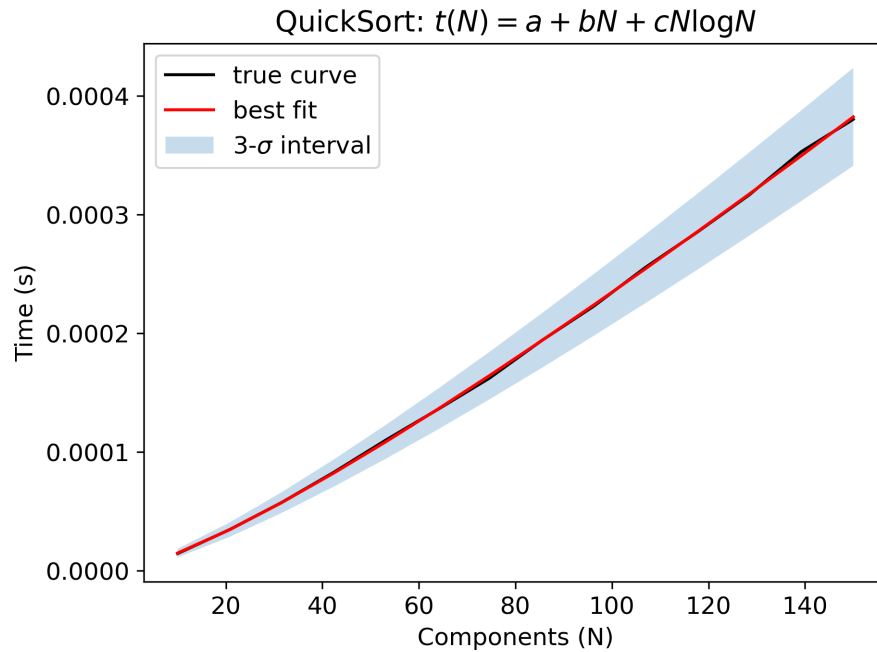


Figura 6

### 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

```

207 """
208 Pr ctica 2. Optimizaci n y complejidad.
209 Daniel Fern ndez Mart nez
210 """
211 import numpy as np
212 import matplotlib.pyplot as plt
213 import time
214 from scipy.optimize import curve_fit
215
216 plt.style.use('default')
217 plt.rcParams.update({'font.size': 12})
218
219 " InsertionSort Algorithm"
220 def insertSort(array):
221
222     # Traverse through 1 to len(arr)
223     for i in range(1, len(array)):
224
225         key = array[i]
226
227         # Move elements of arr[0..i-1], that are
228         # greater than key, to one position ahead
229         # of their current position
230         j = i-1
231         while j >=0 and key < array[j] :
232             array[j+1] = array[j]
233             j -= 1
234         array[j+1] = key
235     return array

```



```

236
237 " QuickSort Algorithm"
238 def partition(arr, low, high):
239     i = (low-1)          # index of smaller element
240     pivot = arr[high]    # pivot
241
242     for j in range(low, high):
243
244         # If current element is smaller than or
245         # equal to pivot
246         if arr[j] <= pivot:
247
248             # increment index of smaller element
249             i = i+1
250             arr[i], arr[j] = arr[j], arr[i]
251
252     arr[i+1], arr[high] = arr[high], arr[i+1]
253     return (i+1)
254
255 # The main function that implements QuickSort
256 # arr[] --> Array to be sorted,
257 # low --> Starting index,
258 # high --> Ending index
259
260 # Function to do Quick sort
261 def quickSort(arr, low, high):
262     if len(arr) == 1:
263         return arr
264     if low < high:
265
266         # pi is partitioning index, arr[p] is now
267         # at right place
268         pi = partition(arr, low, high)
269
270         # Separately sort elements before
271         # partition and after partition
272         quickSort(arr, low, pi-1)
273         quickSort(arr, pi+1, high)
274
275 # Polynomial function for insertsort
276 def func1(N, a,b,c):
277     return a + b*N + c*N**2
278
279 # Logarithmic function for quicksort
280 def func2(N, a,b,c):
281     return a + b*N + c*N*np.log(N)
282
283 mode = 1
284
285 if mode == 0:
286
287     # Study of the InsertionSort Algorithm
288     """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
289         time needed by the previous two algorithms to order them. Compute the average
290         of these times."""
291
292     num_arr = 10000
293     time_N = []
294     k = 0
295     for i in range(10, 150, 10):
296         v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors
297
298         totalTime = []
299         for j in range(len(v)):
300             start = time.time()
301             v_insertion = insertionSort(v[j])
302             end = time.time()
303             totalTime.append(end-start)
304
305         time_N.append(np.mean(totalTime))
306         print('%s seconds needed for InsertionSort algorithm ' % time_N[k] + '(N = %d)' % i)
307         k = k+1
308
309     plt.plot(v_insertion)

```

```

310 plt.title('Last vector sorted with InsertionSort')
311 plt.xlabel('Vector index')
312 plt.ylabel('Value')
313 plt.tight_layout()
314 plt.savefig('insertsorted.png', dpi=300)
315 plt.show()
316
317 """ Check that the average time obtained with the InsertSort algorithm
318 verifies the following law:  $t(N) = c_1 + c_2N + c_3N^2$  """
319 x_data = np.linspace(10, 150, len(time_N))
320 plt.plot(x_data, time_N, label='true curve')
321 plt.title('InsertSort:  $t(N)=c_1+c_2N+c_3N^2$ ')
322
323 "Fit of time_N"
324 # Model fit
325 popt, pcov = curve_fit(func1, x_data, time_N)
326 a = popt[0]
327 b = popt[1]
328 c = popt[2]
329 print(a, b, c)
330
331 # Plot curve
332 fit = func1(x_data, a, b, c)
333 plt.plot(x_data, fit, label='fit')
334 plt.xlabel('Components (N)')
335 plt.ylabel('Time (s)')
336 plt.legend()
337 plt.tight_layout()
338 plt.savefig('insertfit.png', dpi=300)
339 plt.show()
340
341 "Errors"
342 perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
343 nstd = 3
344 popt_up = popt + nstd*perr
345 popt_dw = popt - nstd*perr
346
347 fit_up = func1(x_data, *popt_up)
348 fit_dw = func1(x_data, *popt_dw)
349
350 plt.plot(x_data, time_N, 'k', label='true curve')
351 plt.plot(x_data, fit, 'r', label='best fit')
352 plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3- $\sigma$  interval')
353 plt.title('InsertSort:  $t(N)=c_1+c_2N+c_3N^2$ ')
354 plt.xlabel('Components (N)')
355 plt.ylabel('Time (s)')
356 plt.legend(loc='upper left')
357 plt.tight_layout()
358 plt.savefig('insertfit_interval.png', dpi=300)
359 plt.show()
360
361 if mode == 1:
362     # Study of the QuickSort Algorithm
363     # """ Generate 1000 N-component vectors. Over these vectors, compute the CPU
364     # time needed by the previous two algorithms to order them. Compute the average
365     # of these times. """
366     num_arr = 10000
367     time_N = []
368     k = 0
369     for i in range(10, 150, 10):
370         v = np.random.normal(0, 1, size=(num_arr, i)) # num_arr i-component vectors
371
372         totalTime = []
373         for j in range(len(v)):
374             start = time.time()
375             quickSort(v[j], 0, i-1)
376             end = time.time()
377             totalTime.append(end-start)
378
379         time_N.append(np.mean(totalTime))
380         print('%s seconds needed for QuickSort algorithm ' % time_N[k] + '(N = %d)' % i)
381         k = k+1
382
383 plt.plot(v[i])

```

```

384 plt.title('Last vector sorted with QuickSort')
385 plt.xlabel('Vector index')
386 plt.ylabel('Value')
387 plt.tight_layout()
388 plt.savefig('quicksorted.png', dpi=300)
389 plt.show()
390
391 """ Check that the average time obtained with the InsertSort algorithm
392 verifies the following law:  $t(N) = a + bN + cN\log(N)$  """
393 x_data = np.linspace(10, 150, len(time_N))
394 plt.plot(x_data, time_N, label='true curve')
395 plt.title('QuickSort:  $t(N)=a + bN + cN \log\{N\}$ ')
396
397 "Fit of time_N"
398 # Model fit
399 popt, pcov = curve_fit(func2, x_data, time_N)
400 a = popt[0]
401 b = popt[1]
402 c = popt[2]
403 print(a, b, c)
404 # Plot curve
405 fit = func2(x_data, a, b, c)
406 plt.plot(x_data, fit, label='fit')
407 plt.xlabel('Components (N)')
408 plt.ylabel('Time (s)')
409 plt.legend()
410 plt.tight_layout()
411 plt.savefig('quickfit.png', dpi=300)
412 plt.show()
413
414 "Errors"
415 perr = np.sqrt(np.diag(pcov))/np.sqrt(len(x_data))
416 nstd = 3
417 popt_up = popt + nstd*perr
418 popt_dw = popt - nstd*perr
419
420 fit_up = func2(x_data, *popt_up)
421 fit_dw = func2(x_data, *popt_dw)
422
423 plt.plot(x_data, time_N, 'k', label='true curve')
424 plt.plot(x_data, fit, 'r', label='best fit')
425 plt.fill_between(x_data, fit_up, fit_dw, alpha=.25, label='3- $\sigma$  interval')
426 plt.legend(loc='upper left')
427 plt.title('QuickSort:  $t(N)=a + bN + cN \log\{N\}$ , )
428 plt.xlabel('Components (N)')
429 plt.ylabel('Time (s)')
430 plt.tight_layout()
431 plt.savefig('quickfit_interval.png', dpi=300)
432 plt.show()

```