MA934_Class2_2260253

October 19, 2022

1 MA934 - class 2

1.1 Deadline: 12:00 noon on Thursday 20 October

You should make at least one commit to your repository per computational task below - usually more.

For this assignment, you must create a new Jupyter notebook called MA934_Class2_UniID.ipynb to contain the implementations that you write. You can separate out individual tasks if you prefer, but the full submission should be made as a single .zip via our website. The platform will not allow you to upload more than one file.

A few tips: - please make sure to debug intermediate outputs as you code along. You are welcome to design smaller test cases and toy problems to verify your work (even if they are not part of the final submission). - consider possible forms of input or arguments and make sure your solution can cope with *interesting* cases. - do not forget to comment your code and use Markdown cells to explain what you are doing. A perfectly functional solution with no information about the thought process will not receive more than a subset of the points (~70% depending on the difficulty of the problem and how transparent the algorithm flow is). - generally getting used to writing tidy solution is good practice. Feel free to use online resources for editing guidance.

1.2 Task 1 - insertion sort

[1]:
```
# Import libraries
import time
import numpy as np
from numpy import linalg as LA
import random
import matplotlib.pyplot as plt
plt.rcParams['text.usetex'] = True
```

Insertion sort algorithm: This function is designed to only work for arrays and lists and will return an error for alternative datatypes. The idea is to iterate through elements of a list/array of size n and move the element of interest leftwards until it meets an element of smaller size. Eventually, we arrive at a list/array of integers in ascending order.

[2]:
```
#function to sort arrays using the insertion method
def SortINSR(a):
    if type(a) == np.ndarray or type(a) == list:
        n = len(a)
        if n == 1:
            return a #re-return the array if it contains only one element
        else:
            for j in range(1,n): #for each iteration we focus on moving the␣
↪a[j] element of the original array
                k = j
                while a[k] < a[k-1] and k > 0: #iterates until a[k] is in␣
↪suitable position or at the beginning
                    hold = a[k-1] #placeholder variable to store a[k-1] before␣
↪replacement
                    a[k-1] = a[k] #a[k] is now smaller value
                    a[k] = hold #a[k] is now larger value
                    k -= 1 #move index down the array to compare with next␣
↪smaller element

            return a #array has been sorted
    else:
        return 'Error: argument must be a list or array'
```

Testing:

[3]:
```
#test case
#visible by eye - arrays up to 100 in size
test1 = np.array(random.sample(range(1,5000),100))
print(SortINSR(test1))

#for larger arrays
test1 = np.array(random.sample(range(1,5000),1000))
for l in range(2,len(test1)):
    if test1[l] < SortINSR(test1)[:l-1].max() or test1[l] < test1[l-1]:
        print('whoops') #should have no output if correct
```

```
[  12  162  178  213  263  291  336  396  427  510  548  568  665  691
  713  773  778  780  830  890  923  936 1000 1079 1162 1166 1202 1373
 1414 1442 1518 1552 1569 1623 1626 1718 1800 1839 1861 1889 1915 1946
 2058 2070 2148 2156 2160 2201 2243 2273 2283 2311 2347 2414 2417 2491
 2512 2536 2584 2679 2767 2828 2891 2990 3052 3079 3185 3274 3358 3389
 3397 3453 3456 3517 3586 3645 3678 3780 3947 3987 3996 4158 4194 4235
 4243 4313 4333 4373 4398 4655 4742 4797 4802 4871 4879 4898 4910 4919
 4934 4957]
```

1.3 Task 2 - mergesort

(Given) interlace function: This function is designed to **only** work for lists and will return an error for alternative datatypes. Since the "+" operation differs in meaning from lists to arrays, implementing the algorithm for arrays will return the sum of entries in the arrays, rather than appending elements to `alist`. The objective is to interlace two lists, `list1` and `list2` of size n and m respectively, and create a new list, called `alist`, of size $n + m$. In each iteration we append alist with the smaller of `list1` and `list2` elements. When we make a comparison between the i-th element of `list1` against the j-th element of `list2`, our next iteration will either compare the $i+1$-th element of `list1` against the j-th element of `list2` if `list1[i]` is smaller, or compare the i-th element of `list1` against the $j+1$-th element of `list2` if `list2[j]` is smaller. This is the recursive part of the algorithm. For our next comparison we can call the function again to interlace `list1[i:]` and `list2[j:]`, as the previous elements in `list1` and `list2` have already been considered for interlacing and lie in `alist`.

[4]:
```
#Interlacing function
def interlace(list1, list2):
    if type(list1) == list and type(list2) == list:
        alist = [] #new empty list to hold the interlaced elements
        if (len(list1) == 0):
            return list2
        elif (len(list2) == 0):
            return list1
        elif list1[0] < list2[0]:
            alist.append(list1[0]) #add list1[0] to alist if smaller than␣
 ↪list2[0]
            return alist + interlace(list1[1:], list2) #recursive part - will␣
 ↪compare next element of list1 with list2[0] etc
        else:
            alist.append(list2[0]) #add list2[0] to alist if smaller than␣
 ↪list1[0]
            return alist + interlace(list1, list2[1:]) #recursive part - will␣
 ↪compare next element of list1 with list1[0] etc

    else:
        return 'Error: arguments must be lists'
```

Testing:

[5]:
```
#test cases (interlacing)
test2 = random.sample(range(1,2000),10)
test3 = random.sample(range(1,2000),10)
print('test2 = ',test2)
print('test2 = ',test3)
print(interlace(test2,test3))
```

```
test2 =  [27, 938, 1116, 438, 941, 1437, 822, 46, 335, 948]
test2 =  [314, 330, 878, 1995, 1412, 1991, 768, 569, 1463, 969]
[27, 314, 330, 878, 938, 1116, 438, 941, 1437, 822, 46, 335, 948, 1995, 1412,
1991, 768, 569, 1463, 969]
```

Mergesort function: This function is designed to work for lists and will return an error for alternative datatypes. If the given argument is an array, it will convert it to a list using `array.tolist()`. The idea is to apply the recursive interlace function above to interlace the first half of the list with the second. If our given list is odd, our separation point is $m = \lfloor n/2 \rfloor$ so the second list is one element larger. Mergesort is a recursive algorithm - the two arrays are already sorted and the final sort should occur in $\mathcal{O}(n)$ time.

[6]:
```
#mergesort function
def SortMERGE(a):
    if type(a) == np.ndarray:
        a = a.tolist() #need to convert arrays to lists for interlacing␣
 ↪function to work
    if type(a) == list:
        n = len(a)
        if n == 1:
            return a #re-return the list if it contains only one element
        else:
            m = int(np.floor(n/2))
            return interlace(SortMERGE(a[0:m]),SortMERGE(a[m:n])) #recursive␣
 ↪part - interlace first half of list with the second half

    else:
        return 'Error: arguments must be lists'
```

Testing:

[7]:
```
#test case
#visible by eye - arrays up to 100 in size
test4 = np.array(random.sample(range(1,2000),10))
print(SortMERGE(test4))

#for larger arrays
test4 = np.array(random.sample(range(1,5000),1000))
for l in range(2,len(test4)):
    if test4[l] < SortINSR(test4)[:l-1].max() or test4[l] < test4[l-1]:
        print('whoops') #should have no output if correct
```

```
[171, 669, 695, 811, 912, 1334, 1445, 1476, 1564, 1957]
```

1.4 Task 3 - fixing mergesort

[8]:
```
#testing for the maximum recursion depth for mergesort
test5 = random.sample(range(1,10**12),2**12)
result = SortMERGE(test5)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Input In [8], in <cell line: 3>()
      1 #testing for the maximum recursion depth for mergesort
      2 test5 = random.sample(range(1,10**12),2**12)
----> 3 result = SortMERGE(test5)
```

```
Input In [6], in SortMERGE(a)
      9     else:
     10         m = int(np.floor(n/2))
---> 11         return interlace(SortMERGE(a[0:m]),SortMERGE(a[m:n])) #recursiv␣
 ↪part - interlace first half of list with the second half
     13 else:
     14     return 'Error: arguments must be lists'

Input In [4], in interlace(list1, list2)
     12     else:
     13         alist.append(list2[0]) #add list2[0] to alist if smaller than␣
 ↪list1[0]
---> 14         return alist + interlace(list1, list2[1:]) #recursive part -␣
 ↪will compare next element of list1 with list1[0] etc
     16 else:
     17     return 'Error: arguments must be lists'

Input In [4], in interlace(list1, list2)
     12     else:
     13         alist.append(list2[0]) #add list2[0] to alist if smaller than␣
 ↪list1[0]
---> 14         return alist + interlace(list1, list2[1:]) #recursive part -␣
 ↪will compare next element of list1 with list1[0] etc
     16 else:
     17     return 'Error: arguments must be lists'

Input In [4], in interlace(list1, list2)
      9 elif list1[0] < list2[0]:
     10     alist.append(list1[0]) #add list1[0] to alist if smaller than␣
 ↪list2[0]
---> 11     return alist + interlace(list1[1:], list2) #recursive part - will␣
 ↪compare next element of list1 with list2[0] etc
     12 else:
     13     alist.append(list2[0]) #add list2[0] to alist if smaller than␣
 ↪list1[0]

Input In [4], in interlace(list1, list2)
      9 elif list1[0] < list2[0]:
     10     alist.append(list1[0]) #add list1[0] to alist if smaller than␣
 ↪list2[0]
---> 11     return alist + interlace(list1[1:], list2) #recursive part - will␣
 ↪compare next element of list1 with list2[0] etc
     12 else:
     13     alist.append(list2[0]) #add list2[0] to alist if smaller than␣
 ↪list1[0]

    [… skipping similar frames: interlace at line 14 (1494 times), interlace at␣
 ↪line 11 (1470 times)]
```

```
Input In [4], in interlace(list1, list2)
     12     else:
     13         alist.append(list2[0]) #add list2[0] to alist if smaller than␣
 ↪list1[0]
---> 14         return alist + interlace(list1, list2[1:]) #recursive part -␣
 ↪will compare next element of list1 with list1[0] etc
     16 else:
     17     return 'Error: arguments must be lists'

    [… skipping similar frames: interlace at line 11 (1 times)]

Input In [4], in interlace(list1, list2)
      9 elif list1[0] < list2[0]:
     10     alist.append(list1[0]) #add list1[0] to alist if smaller than␣
 ↪list2[0]
---> 11     return alist + interlace(list1[1:], list2) #recursive part - will␣
 ↪compare next element of list1 with list2[0] etc
     12 else:
     13     alist.append(list2[0]) #add list2[0] to alist if smaller than␣
 ↪list1[0]

Input In [4], in interlace(list1, list2)
      2 def interlace(list1, list2):
----> 3     if type(list1) == list and type(list2) == list:
      4         alist = [] #new empty list to hold the interlaced elements
      5         if (len(list1) == 0):

RecursionError: maximum recursion depth exceeded while calling a Python object
```

It seems to exceed the maximum recursion depth for integer arrays of length 2^{12}. We'll now edit the interlacing function to be non-recursive so we do not overflow the call stack.

(Non-recursive) interlace function: This function is the same as the recursive interlace function in terms of allowable arguments. To alter the function to be non-recursive, we use two variables `j1` and `j2` to keep track of the number of elements from `list1` and `list2` respectively which have already been interlaced into `alist`. For each comparison we increment the suitable `j` variable by one until the list is exhausted.

[9]:
```python
#Interlacing function (non-recursive)
def interlace_adapted(list1, list2):
    if type(list1) == list and type(list2) == list: #designed to work for lists
        alist = [] #new empty list to hold the interlaced elements
        if (len(list1) == 0):
            return list2 #return second list if first list empty
        elif (len(list2) == 0):
            return list1 #return first list if second list empty
        else:
            j1 = 0
            j2 = 0 #index holders for j1 and j2 respectively
            while j1 < len(list1) and j2 < len(list2): #repeats iteration until␣
 ↪one of lists exhausted
                if list1[j1] < list2[j2]: #if j1-th element in list1 is␣
 ↪smallest...
                    alist.append(list1[j1]) #...add it to alist...
                    j1 += 1 #...and for further comparisons we will consider␣
 ↪the subsequent element in list1
                else: #if j2-th element in list2 is smallest, repeat above␣
 ↪process for list2
                    alist.append(list2[j2])
                    j2 += 1
            alist += list1[j1:] + list2[j2:] #append alist with remainder of␣
 ↪both lists (one of which will be non-empty)
            return alist #alist is interlaced

    else:
        return 'Error: arguments must be lists'
```

Testing:

[10]:
```
#test case
size_diff = np.random.randint(-5,5) #also will test if it works for lists of␣
 ↪different sizes
test6 = random.sample(range(1,10**3),2**3)
test7 = random.sample(range(1,10**3),2**3+size_diff)
print('test6 = ',test6)
print('test7 = ',test7)
print(interlace_adapted(test6,test7))
```

```
test6 =  [690, 188, 24, 67, 355, 348, 592, 884]
test7 =  [275, 701, 142, 681, 243, 37, 222, 677]
[275, 690, 188, 24, 67, 355, 348, 592, 701, 142, 681, 243, 37, 222, 677, 884]
```

We will update the `SortMERGE()` function. The sole change is switching to the non-recursive `interlace_adapted()` function above.

[11]:
```
#mergesort function
def SortMERGE(a):
    if type(a) == np.ndarray:
        a = a.tolist() #need to convert arrays to lists for interlacing␣
 ↪function to work
    if type(a) == list:
        n = len(a)
        if n == 1:
            return a #re-return the list if it contains only one element
        else:
            m = int(np.floor(n/2))
            return interlace_adapted(SortMERGE(a[0:m]),SortMERGE(a[m:n]))␣
↪#recursive part - interlace first half of list with the second half

    else:
        return 'Error: arguments must be lists'
```

Testing:

[12]:
```
#test case
#visible by eye - arrays up to 100 in size
test8 = np.array(random.sample(range(0,2000),10))
print(SortMERGE(test8))

#for larger arrays
test8 = np.array(random.sample(range(0,10000),1000))
for l in range(2,len(test8)):
    if test8[l] < SortINSR(test8)[:l-1].max() or test8[l] < test8[l-1]:
        print('whoops') #should have no output if correct
```

```
[666, 1084, 1199, 1220, 1316, 1433, 1458, 1541, 1558, 1845]
```

How does it compare to the recursive interlacing function in terms of achieveable list length?

[13]:
```
#timed case of size 2^18
import timeit

test9 = random.sample(range(0,2**20),2**18)
starttime = timeit.default_timer() #restart timer
result = SortMERGE(test9)
runtime = timeit.default_timer() - starttime #calulate runtime
runtime
```

[13]:
```
0.9773085599999831
```

The test case runs, so we appear to have overcome recursion depth. In the subsequent task we will exploit this to investigate the run time of the mergesort algorithm for random arrays of integers up to 2^{20} in length.

1.5 Task 4 - runtime

[14]:
```
#timing
max_pow = 20
sizesMERGE = np.array([2**i for i in range(1,max_pow+1)]) #array of powers of 2␣
↪up to max desired power
```

To smoothen our results, for each power of 2 we run five times and take an average for the final runtime. With the exception of 2^1, which is so small that we average over $10,000$ runs.

[15]:
```
#merge sort
runtimesMERGE = [np.float32(0.0)] * len(sizesMERGE) #stores runtimes

j = 0
for i in sizesMERGE: #for each power of 2...
    if i == 2:
        repeat = 10**4
    else:
        repeat = 5
    for k in range(repeat):
        A = random.sample(range(0,i*2**3),i) #generate random list
        starttime = timeit.default_timer() #restart timer
        resultMERGE = SortMERGE(A) #perform sorting algorithm
        runtimesMERGE[j] += timeit.default_timer() - starttime #calulate runtime
    runtimesMERGE[j] = runtimesMERGE[j]/repeat
    j = j + 1

print(runtimesMERGE)
```

```
[2.175594099975342e-06, 6.215599989900511e-06, 1.2669800014464272e-05,
2.7095199993709684e-05, 5.7104599989088456e-05, 0.00012446560000398677,
0.0002624828000080015, 0.0005648834000226089, 0.0011863184000048931,
0.00250384999999369, 0.0052762078000114345, 0.011067353599992202,
0.023440521800011993, 0.048708507400010606, 0.10364681040000505,
0.21366818039999771, 0.47014778240001076, 0.9756239602000051,
2.1982883912000033, 4.476444912600004]
```

Insertion sort takes much longer, so we will adjust our max desired power to 16 so the code is executed in a reasonable time. As above, we run 2^1 length 10,000 times and larger lengths thereafter five times, however for length 2^{16} we only run once due to its large runtime (in the ballpark of 160s).

[16]:
```
#insertion method
max_pow = 16
sizesINSR = np.array([2**i for i in range(1,max_pow+1)]) #array of powers of 2

runtimesINSR = [np.float32(0.0)] * len(sizesINSR) #stores runtimes

j = 0
for i in sizesINSR: #for each power of 2...
    if i == 2:
        repeat = 10**4
    elif i > 2**15+1:
        repeat = 1
    else:
        repeat = 5
    for k in range(repeat):
        A = random.sample(range(0,i*2**3),i) #generate random list
        starttime = timeit.default_timer() #restart timer
        resultINSR = SortINSR(A) #perform sorting algorithm
        runtimesINSR[j] += timeit.default_timer() - starttime #calulate runtime
    runtimesINSR[j] = runtimesINSR[j]/repeat
    j = j + 1

print(runtimesINSR)
```

```
[6.752075991585116e-07, 1.4211999996405212e-06, 2.5769999865588035e-06,
7.405799965454207e-06, 2.8095600009692133e-05, 0.0001032023999869125,
0.00046645200002330965, 0.0018955512000027512, 0.007725339000035092,
0.03246548839999832, 0.13364746259999266, 0.535838488399986, 2.2004971465999916,
8.96522726400001, 38.35076225359999, 152.36277569899994]
```

[17]:
```
#rough gague on total runtime for insertion sort - ideally keeping this below␣
 ↪10 minutes
total_time = 10000*runtimesINSR[0] + runtimesINSR[-1]
for l in range(1,len(runtimesINSR)-1):
    total_time += 5*runtimesINSR[l]
total_time
```

[17]:
```
403.5128685139915
```

Save our runtimes to csv files

[18]:
```
print(np.shape(runtimesINSR))
print(np.shape(runtimesMERGE))
```

```
(16,)
(20,)
```

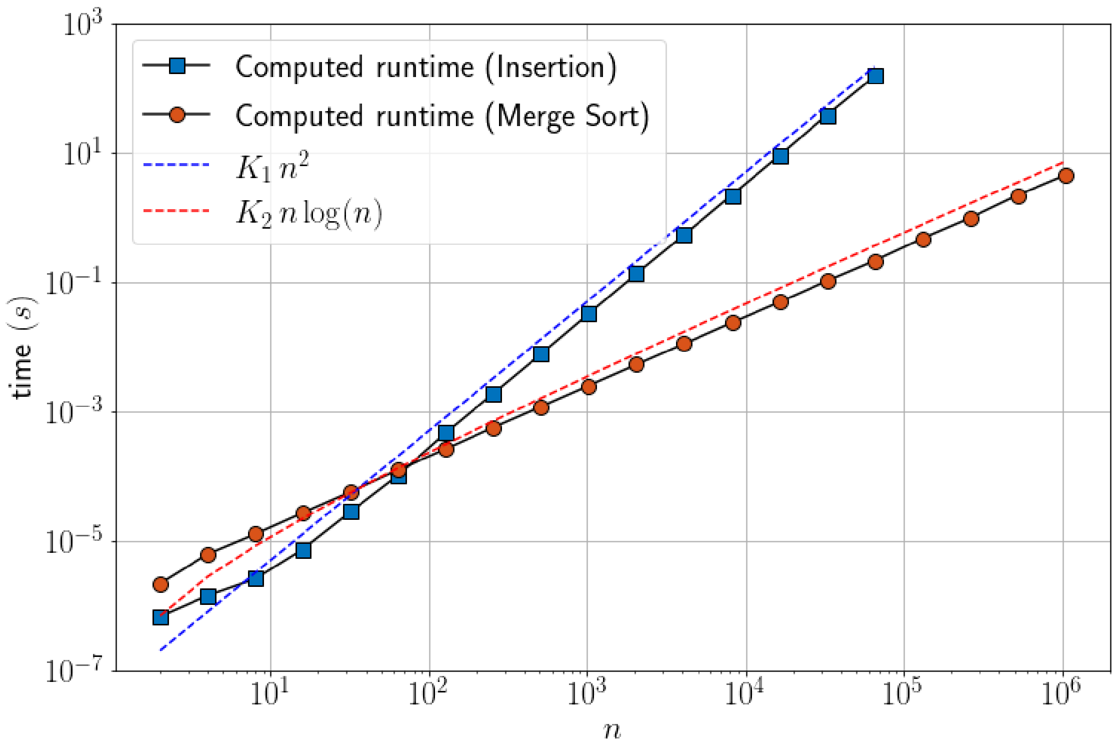
[19]:
```
np.savetxt('./insertion_performance.csv', runtimesINSR, delimiter=",")
np.savetxt('./mergesort_performance.csv', runtimesMERGE, delimiter=",")
```

1.6 Task 5 - empirical analysis of computational complexity

[20]:
```
#reload runtime data
insertion_speed = np.genfromtxt('insertion_performance.csv')
mergesort_speed = np.genfromtxt('mergesort_performance.csv')
print(insertion_speed)
print(mergesort_speed)
```

```
[6.75207599e-07 1.42120000e-06 2.57699999e-06 7.40579997e-06
 2.80956000e-05 1.03202400e-04 4.66452000e-04 1.89555120e-03
 7.72533900e-03 3.24654884e-02 1.33647463e-01 5.35838488e-01
 2.20049715e+00 8.96522726e+00 3.83507623e+01 1.52362776e+02]
[2.17559410e-06 6.21559999e-06 1.26698000e-05 2.70952000e-05
 5.71046000e-05 1.24465600e-04 2.62482800e-04 5.64883400e-04
 1.18631840e-03 2.50385000e-03 5.27620780e-03 1.10673536e-02
 2.34405218e-02 4.87085074e-02 1.03646810e-01 2.13668180e-01
 4.70147782e-01 9.75623960e-01 2.19828839e+00 4.47644491e+00]
```

[21]:
```
#plotting
plt.rcParams.update({'font.size': 22})

plt.figure(figsize=(12, 8))

plt.plot(sizesINSR, [i for i in insertion_speed], "-ks", label=r"Computed␣
↪runtime (Insertion)", markersize=10, markerfacecolor=(0, 0.447, 0.741, 1))
plt.plot(sizesMERGE, [i for i in mergesort_speed], "-ko", label=r"Computed␣
↪runtime (Merge Sort)", markersize=10, markerfacecolor=(0.85, 0.325, 0.098,␣
↪1))

plt.plot(sizesINSR, 1e-6*0.05*np.power(sizesINSR, 2.0), "--b", label=r"$K_1 \,␣
↪n^2$")
plt.plot(sizesMERGE, 1e-6*0.5*sizesMERGE*np.log(sizesMERGE), "--r",␣
↪label=r"$K_2 \, n \log(n)$")

plt.legend(loc="upper left")

plt.xlabel(r"$n$")
plt.ylabel(r"time $(s)$")
plt.xscale("log")
plt.yscale("log")
plt.ylim([1e-7, 1e3])
plt.grid()
plt.show()
```



How do our run times compare to theoretical expectations?

Insertion sort: The worst-case theoretical complexity for the insertion algorithm is $F(n) \sim \mathcal{O}(n^2)$, arising from the fact that for all n elements in our array we make up to $n-1$ comparisons. Our run times for insertion sort scale in a similar fashion to $K_1\, n^2$, with $K_1 = 5 \times 10^{-8}$ independent of n. This indicates that the complexity of our insertion sorting algorithm is $F(n) \sim \mathcal{O}(n^2)$, matching the theoretical expectations.

Merge sort: Despite changing the `interlace_adapted()` function to be iterative rather than recursive, the theoretical complexity of the merge sort algorithm remains at $F(n) \sim \mathcal{O}(n \log_2 n)$. Our `SortMERGE()` function is still recursive with number of levels L. If we suppose our list size n is a power of 2, the number of recursion levels L for a list of size n satisfies $n = 2^L$, so $L = \log_2 n$. With regards to the complexity of `interlace_adapted()`, at level m in the recursion there are $2m$ sub-problems each of size $n/2m$ (best case), or size $(2n-1)/2m$ (worst case). Hence, the complexity of each level of the recursion is $\sim \mathcal{O}(n)$. This gives us a total complexity of $F(n) \sim \mathcal{O}(n \log_2 n)$ as desired.
Our run times for the mergesort algorithm scale in a similar fashion to $K_2\, n \log n$, with $K_2 = 5 \times 10^{-7}$ independent of n. This indicates that the complexity of our insertion sorting algorithm is $F(n) \sim \mathcal{O}(n \log n)$, matching the theoretical expectations.

```
[ ]:
```