MINISTRY OF EDUCATION | وزارة التعليم
IMAM ABDULRAHMAN BIN | جامعة الإمام
FAISAL UNIVERSITY | عبد الرحمن بن
COLLEGE OF COMPUTER | فيصل
SCIENCE | كلية علوم
& INFORMATION | الحاسب
TECHNOLOGY | وتقنية المعلومات

جامعة الإمام عبدالرحمن بن فيصل
IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY

Algorithm Analysis and Design

Academic Year (2022 – 2023) –First semester
CS 412– Final Project

Student name	Student ID
Dlayel Al-uhaideb	2200004891
Fatima Al-gharash	2200002685
Sara Al-talib	2200004233
Shahad Al-jafaari	2200003431
Shahad Al-otaibi	2200003282

Instrctoer:

Dr.Fahd Al-haidari

Section number:

TFA2

1 Table of Contents

Table of graphs and figures 3

1. *The system used in this project* 4

2. *Insertion Sort*.. 5

2.1. **Source code:** 5

2.2. **Insertion sort analysis** 6

2.2.1. Best case: 6

2.2.2. Average case:..7

2.2.3. Worst case:...8

2.3. **Graph:** 9

3. *Quick Sort:* 10

3.1. **Source code:** 10

3.2. **Quick sort analysis:** 11

3.2.1. Best case 11

3.2.2. Average case 12

3.2.3. Worst case..13

3.3. **Graph:** 14

4. *Comparison between quick sort and insertion sort* 15

5. *Hybrid Sort (Quick & insertion):* 16

5.1 **Source Code**...16

5.2 **Hybrid sort analysis**...18

5.2.1 Average case 18

Sample:...18

6. *Comparison between quick sort and insertion sort and the hybrid*................................ 19

6.1 **Graph:** 19

7. *Questions & Answers* 20

7.1 **Describe how you obtained the value of the threshold.**...............................20

7.2 **Describe the numbers of input data you have tested of each length. Take care to choose the sizes of your input appropriately.**...20

7.3 **A description of the observed behavior of the three algorithms for values of n both smaller and larger than the cross-over value determined experimentally.** 20

Table of graphs and figures

Graph 1: Insertion sort 9
Graph 2: Quick sort 14
Graph 3: Average case (insertion vs quick sort) 15
Graph 4: Average case (hybird vs insertion vs quick sort) 19
Graph 5: Average case (hybird vs insertion vs quick sort) 20

Table 1: Characteristics for machine 4
Table 2: Data in ascending order 6
Table 3: Data in random order 7
Table 4: Data in reverse order 8
Table 5: Quick sort bst case 11
Table 6: Quick sort average case 12
Table 7: Quick sort worst case 13
Table 8: Hybird sort average case 18

Sample 1: insertion sort best case 6
Sample 2: insertion sort average case 7
Sample 3: insertion sort worst case 8
Sample 4: quicksort best case 9
Sample 5: quicksort average case 10
Sample 6: quicksort worst case 11
Sample 7: hybird sort average case 18

1. The system used in this project

Characteristics	The Machines
Name of Machines	OMEN by HP Laptop 15-dh0000nx
Operating System	Windows 11
Processer	Intel Core i7
RAM	16 GB
System Type	64-bit

Table 1: Characteristics for machine

2. Insertion Sort

Insertion sort is a famous sorting algorithm that is best used when an array is almost sorted. Thus, insertion sort belongs to the in-place sort family, and the reason why it is called “insertion” sort is for that is literally what it does. The working principle is based on the initial assumption that a single-element array is always sorted. The array will be divided into 2 sub-arrays, one is sorted while the other is not. In each iteration, the sorted part is going to be extended while the unsorted one will shrink until the array is fully sorted. Comparison is done by comparing every 2 adjacent elements and based on the result of the comparison, the smaller element will be moved to its appropriate position in the sorted array (in case the desired result is ascending order) until the unsorted sub-array is empty and the sorted sub-array include all the elements, thus the whole array is fully sorted.

2.1. Source code:

```
#insertion_sort
import time
st = time.time()

def insertionSort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

    #return print1(arr)

def readFile(filename, l):

    with open(filename, "r") as f:
        for line in f:
            l.append(int(line))

    if f.closed == False:
        f.close()

def main():
    txt_array = []
    readFile("1000_random.txt", txt_array)
    size = len(txt_array)
    insertionSort(txt_array)

main()
#End time
et = time.time()

# get the execution time
elapsed_time = (et - st )
print('Execution time:', elapsed_time, 's')
elapsed_time_ms = round(elapsed_time *1000,2)
print('Execution time:', elapsed_time_ms, 'ms')
```

2.2. Insertion sort analysis

2.2.1. Best case:

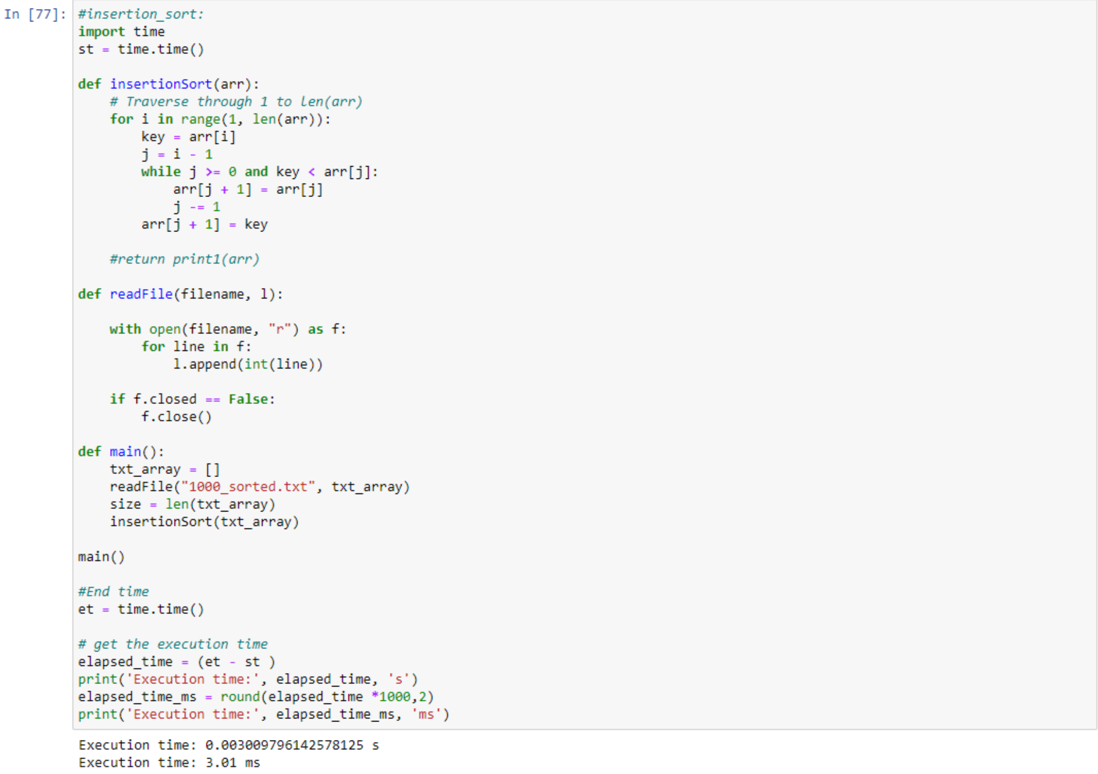
The best-case analysis for the insertion sort is **θ(n),** and that is the case when the input array is already sorted. The reason why it is **n** even though the array is sorted is because of the condition. The outer loop will be executed every time and so does the condition of the inner loop to make sure that every element is in its appropriate position. The inner loop condition in this case is executed (n-1) times. Yet, it will not be entered due to the fact that the condition will always be false since the array is sorted.

Calculate execution time:

Data in ascending order				
Input	Time (1 ms)	Time (2 ms)	Time (3 ms)	Average Of Times
50	0.75	0.88	1.00	0.88
100	1.99	1.33	1.02	1.45
500	2.50	2.00	1.99	2.16
1000	3.65	3.55	3.01	3.41

Table 2: Data in ascending order

Sample:



```
In [77]: #insertion_sort:
import time
st = time.time()

def insertionSort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

    #return print1(arr)

def readFile(filename, l):

    with open(filename, "r") as f:
        for line in f:
            l.append(int(line))

    if f.closed == False:
        f.close()

def main():
    txt_array = []
    readFile("1000_sorted.txt", txt_array)
    size = len(txt_array)
    insertionSort(txt_array)

main()

#End time
et = time.time()

# get the execution time
elapsed_time = (et - st )
print('Execution time:', elapsed_time, 's')
elapsed_time_ms = round(elapsed_time *1000,2)
print('Execution time:', elapsed_time_ms, 'ms')
```

```
Execution time: 0.003009796142578125 s
Execution time: 3.01 ms
```

Sample 1: insertion sort best case

2.2.2. Average case:

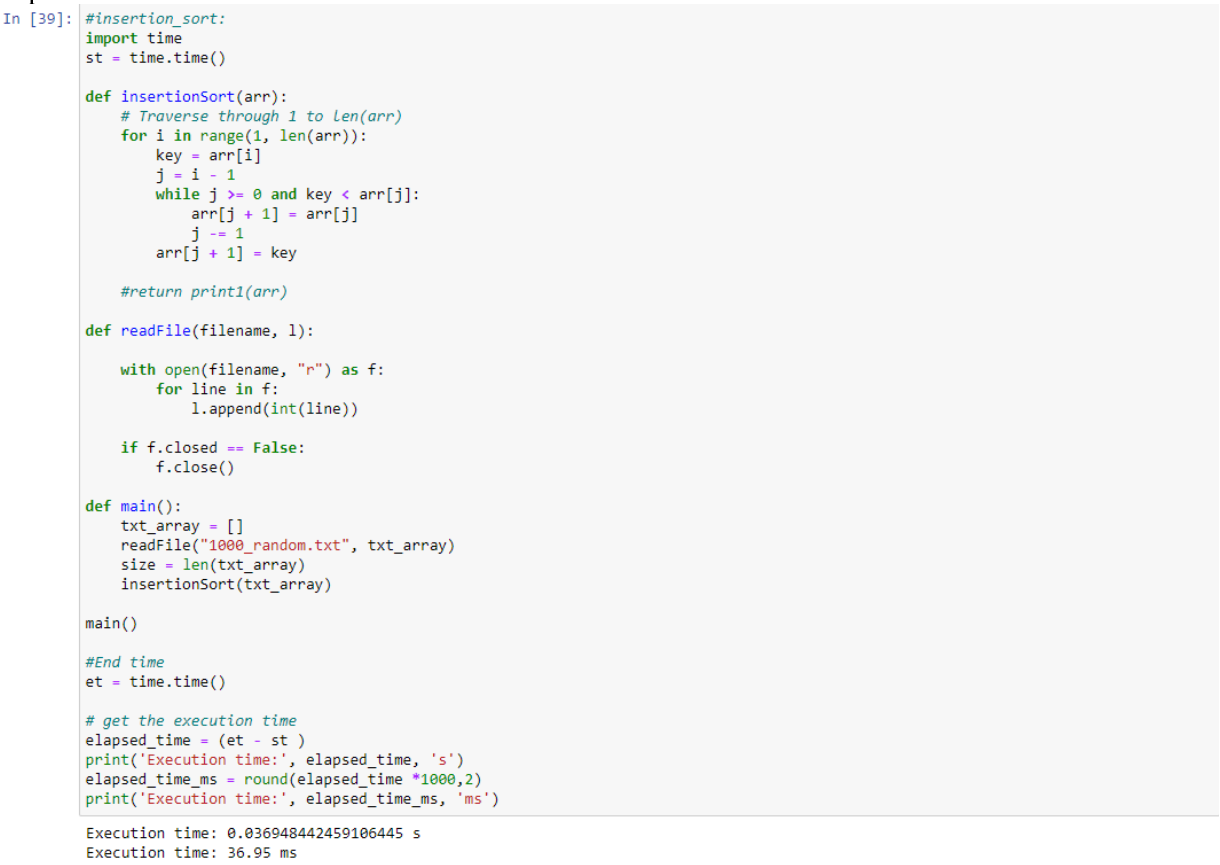
The average case analysis for the insertion sort is **θ(n^2)**. The average case occurs when an array is half sorted, and the reason why it is **n^2** is that both inner and outer loops will be executed, and even if both were not executed every time it will be considered as a quadratic running time because the focus of the time complexity is on the growth of a function.

Calculate execution time:

Data in random order				
Input	Time (1 ms)	Time (2 ms)	Time (3 ms)	Average Of Times
50	1.32	1.00	1.50	1.27
100	2.52	2.03	2.01	2.19
500	19.95	14.65	17.95	17.52
1000	35.14	42.35	36.95	38.15

Table 3: Data in random order

Sample:



```
In [39]: #insertion_sort:
         import time
         st = time.time()

         def insertionSort(arr):
             # Traverse through 1 to len(arr)
             for i in range(1, len(arr)):
                 key = arr[i]
                 j = i - 1
                 while j >= 0 and key < arr[j]:
                     arr[j + 1] = arr[j]
                     j -= 1
                 arr[j + 1] = key

             #return print1(arr)

         def readFile(filename, l):

             with open(filename, "r") as f:
                 for line in f:
                     l.append(int(line))

             if f.closed == False:
                 f.close()

         def main():
             txt_array = []
             readFile("1000_random.txt", txt_array)
             size = len(txt_array)
             insertionSort(txt_array)

         main()

         #End time
         et = time.time()

         # get the execution time
         elapsed_time = (et - st )
         print('Execution time:', elapsed_time, 's')
         elapsed_time_ms = round(elapsed_time *1000,2)
         print('Execution time:', elapsed_time_ms, 'ms')

         Execution time: 0.036948442459106445 s
         Execution time: 36.95 ms
```

Sample 2: insertion sort average case

2.2.3. Worst case:

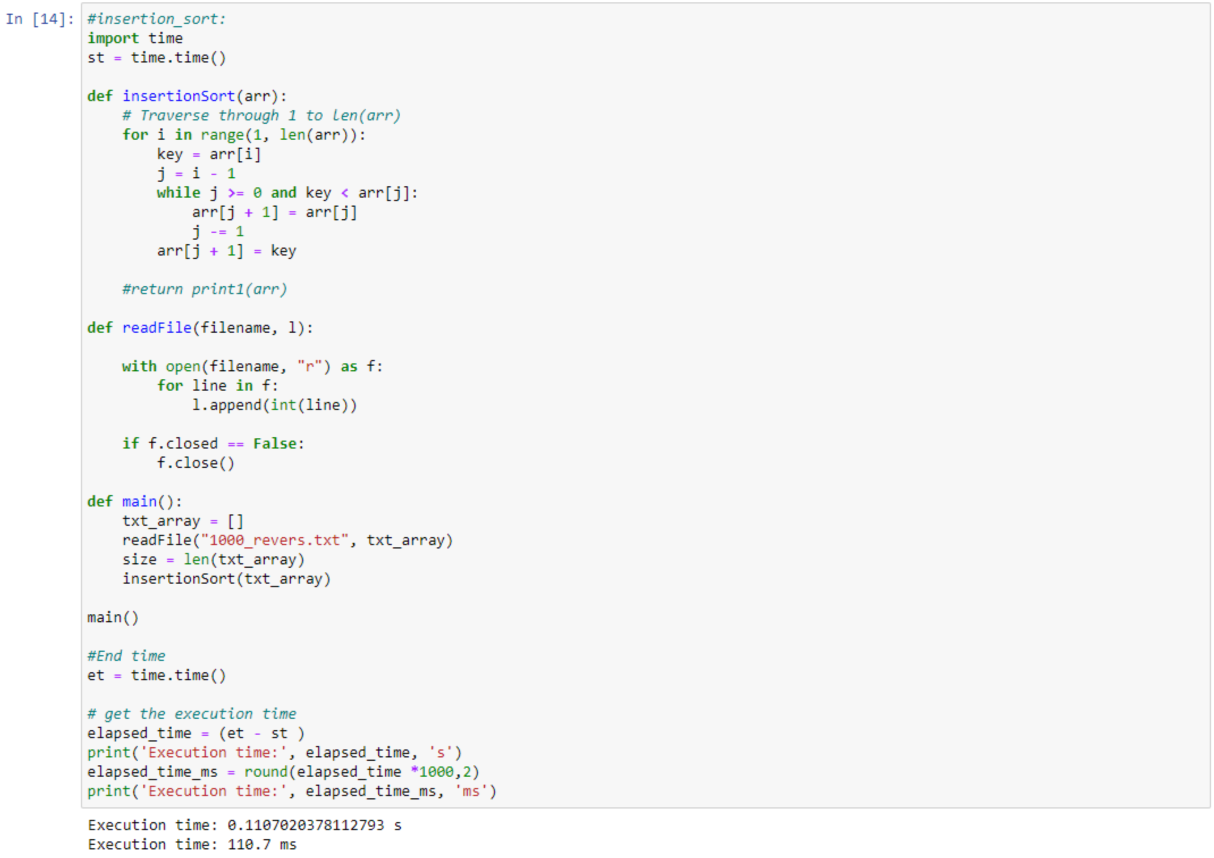
The worst-case analysis for the insertion sort is **θ(n^2),** which happens when an array is reversely sorted. For each element in the array, both loops will be executed leading the time complexity to be **n^2** for each and every iteration in the array

Calculate execution time:

Data in reverse order				
Input	Time (1 ms)	Time (2 ms)	Time (3 ms)	Average Of Times
50	1.02	1.00	1.99	1.34
100	3.99	2.99	1.52	2.83
500	34.13	32.12	31.20	32.48
1000	109.19	110.72	98.73	106.21

Table 4: Data in reverse order

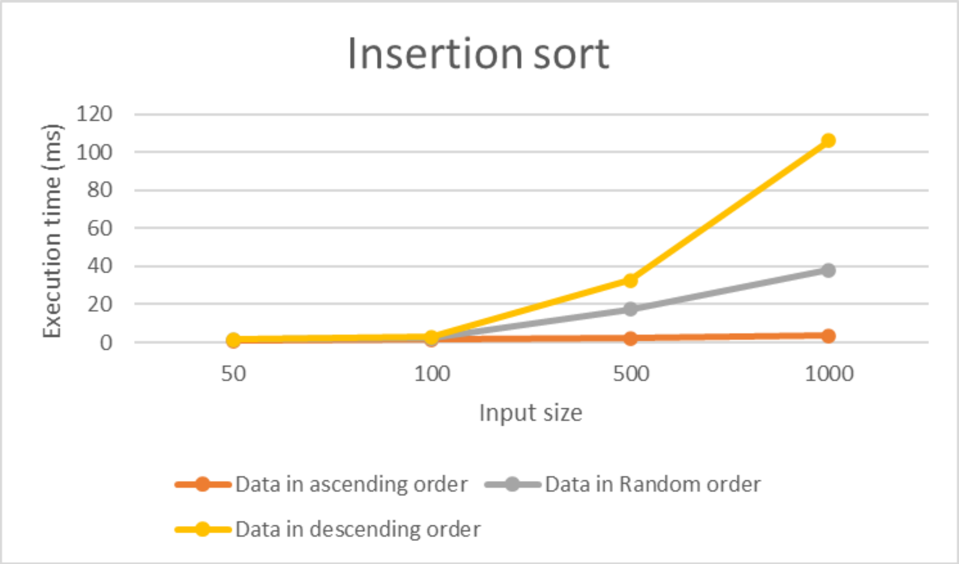
Sample:



```
In [14]: #insertion_sort:
         import time
         st = time.time()

         def insertionSort(arr):
             # Traverse through 1 to len(arr)
             for i in range(1, len(arr)):
                 key = arr[i]
                 j = i - 1
                 while j >= 0 and key < arr[j]:
                     arr[j + 1] = arr[j]
                     j -= 1
                 arr[j + 1] = key

             #return print1(arr)

         def readFile(filename, l):

             with open(filename, "r") as f:
                 for line in f:
                     l.append(int(line))

             if f.closed == False:
                 f.close()

         def main():
             txt_array = []
             readFile("1000_revers.txt", txt_array)
             size = len(txt_array)
             insertionSort(txt_array)

         main()

         #End time
         et = time.time()

         # get the execution time
         elapsed_time = (et - st )
         print('Execution time:', elapsed_time, 's')
         elapsed_time_ms = round(elapsed_time *1000,2)
         print('Execution time:', elapsed_time_ms, 'ms')

         Execution time: 0.1107020378112793 s
         Execution time: 110.7 ms
```

Sample 3: insertion sort worst case

2.3. Graph:



Graph 1: Insertion sort

This figure represents the resulting execution time of 4 inputs size which are: 50,100,500 and 1000. We applied these inputs based on three cases, the first is when the array is in ascending order then random order and lastly descending order. We observed that the case of the ascending has the least execution time making it the best case.

3. Quick Sort:

As a divide-and-conquer algorithm, Quicksort greatly depends on the dividing step to determine its running time. Arrays are divided into two subarrays by an element called the pivot. Following the resolution of the subproblems, the original algorithm is recombined using the subproblems' results.

3.1. Source code:

```
#quick_sort
import time
st = time.time()

def partition(arr, low, high):
    i = (low - 1)
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)

def readFile(filename, l):
    with open(filename, "r") as f:
        for line in f:
            l.append(int(line))
    if f.closed == False:
        f.close()

def main():
    txt_array = []
    readFile('1000_random_with_middle.txt', txt_array)
    size = len(txt_array)
    quickSort(txt_array, 0, size - 1)

main()
#End time
et = time.time()
# get the execution time
elapsed_time = (et - st )
print('Execution time:', elapsed_time, 's')
elapsed_time_ms = round(elapsed_time *1000,2)
print('Execution time:', elapsed_time_ms, 'ms')
```

3.2. Quick sort analysis:

3.2.1. Best case

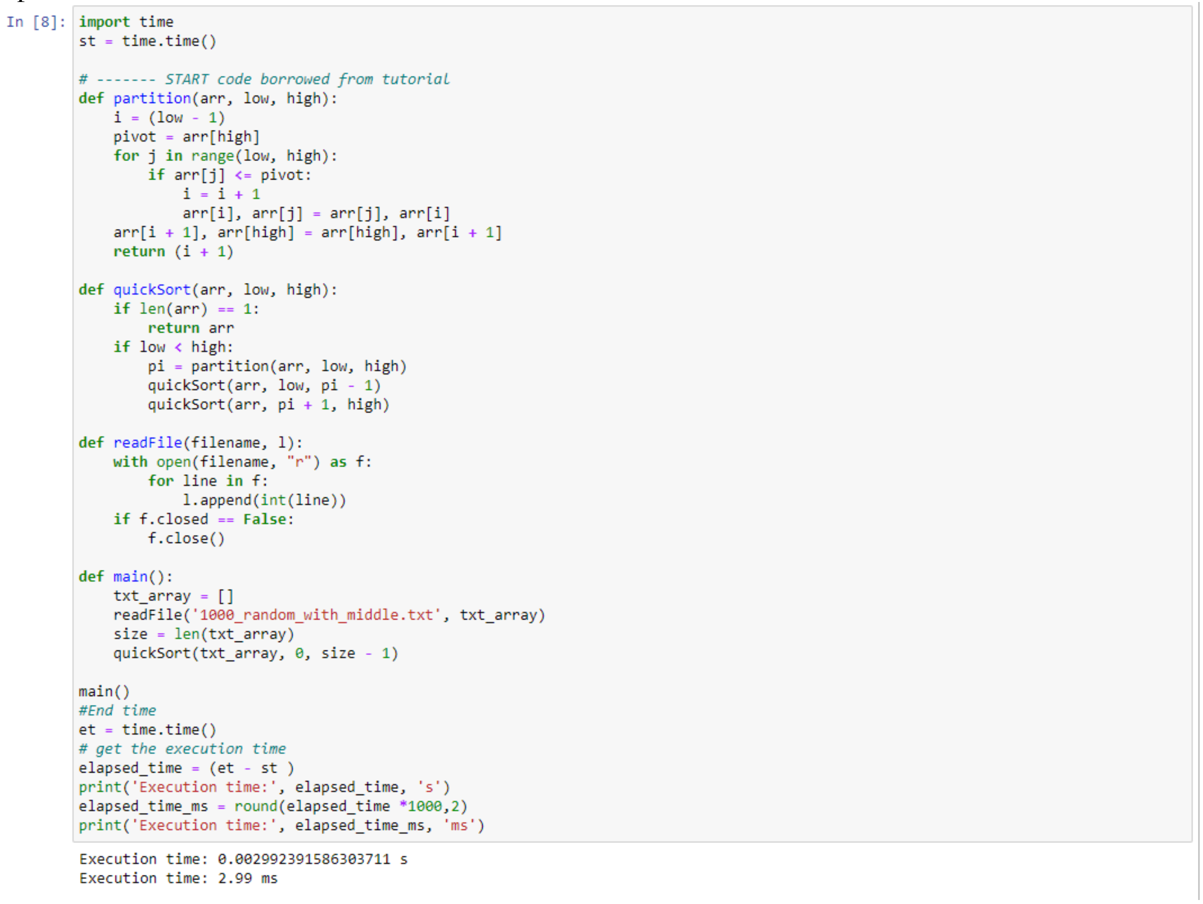
In the best-case scenario, quicksort takes Q(nlgn). Typically, it occurs when two halves of the input list are balanced, and the partitioning subroutine returns the middle index. It is only possible to achieve balance by making the pivot the median of the input list. Therefore, a pivot chosen from the middle of an input list is expected to give a running time that aligns with the theoretical one.

Calculate execution time:

Best Case				
Input	Time (1 ms)	Time (2 ms)	Time (3 ms)	Average Of Times
50	1.00	1.00	0.64	0.88
100	1.07	1.00	1.72	1.26
500	1.99	2.01	1.54	1.85
1000	3.03	2.99	3.86	3.29

Table 5: Quick sort best case

Sample:



```
In [8]: import time
st = time.time()

# ------- START code borrowed from tutorial
def partition(arr, low, high):
    i = (low - 1)
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)

def readFile(filename, l):
    with open(filename, "r") as f:
        for line in f:
            l.append(int(line))
    if f.closed == False:
        f.close()

def main():
    txt_array = []
    readFile('1000_random_with_middle.txt', txt_array)
    size = len(txt_array)
    quickSort(txt_array, 0, size - 1)

main()
#End time
et = time.time()
# get the execution time
elapsed_time = (et - st )
print('Execution time:', elapsed_time, 's')
elapsed_time_ms = round(elapsed_time *1000,2)
print('Execution time:', elapsed_time_ms, 'ms')
```

```
Execution time: 0.002992391586303711 s
Execution time: 2.99 ms
```

Sample 4: quicksort best case

3.2.2. Average case

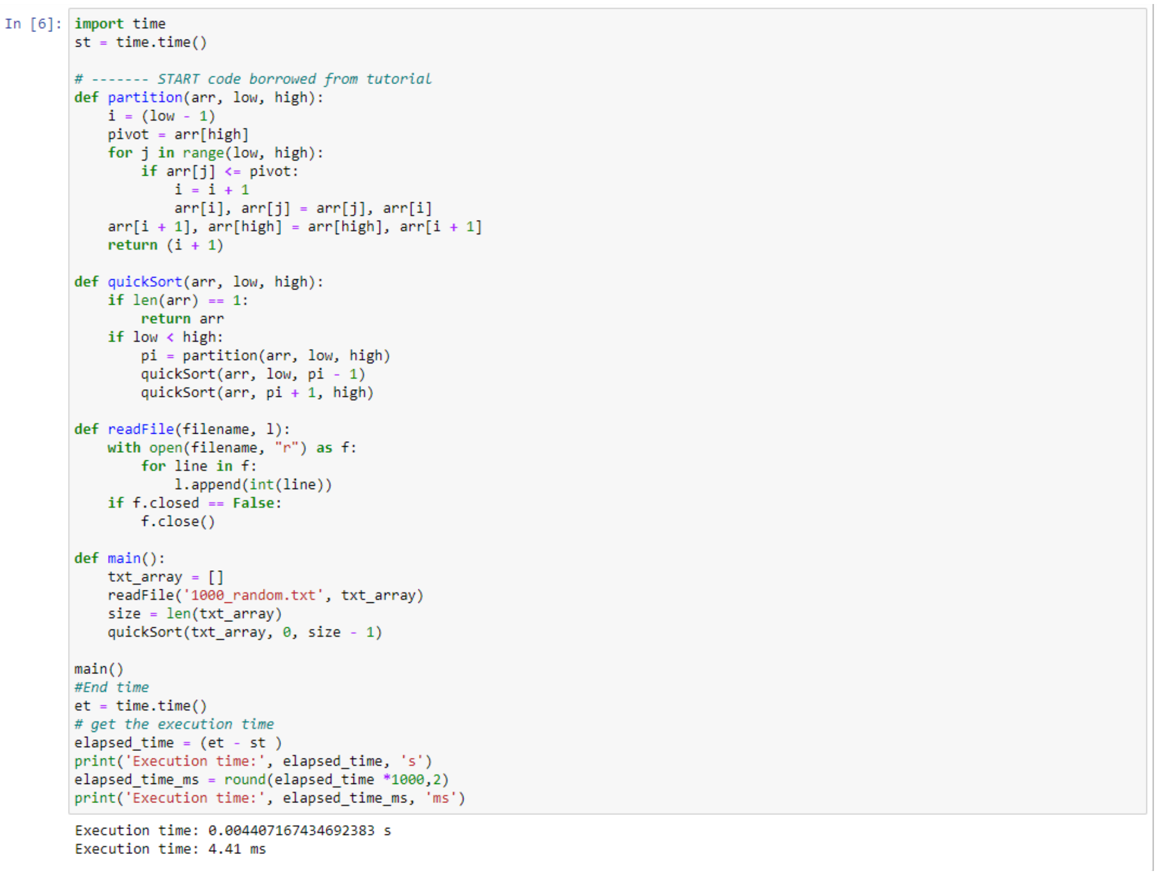
As a general rule, the quicksort algorithm runs in Q(nlgn). Usually, it occurs when the returned by the partitioning subroutine is not the maximum, minimum, or median value. As a result, the two sides of the partition will not be exactly the same size or are not equally balanced. Therefore, if the pivot is not the maximum, minimum, or median element, we expect the experimental results to confirm the theoretical.

Calculate execution time:

Average Case				
Input	Time (1 ms)	Time (2 ms)	Time (3 ms)	Average Of Times
50	1.00	1.99	1.06	1.35
100	1.98	1.48	2.99	2.15
500	2.97	3.01	1.99	2.66
1000	3.43	4.41	3.67	3.84

Table 6: Quick sort average case

Sample:



```
In [6]: import time
        st = time.time()

        # ------- START code borrowed from tutorial
        def partition(arr, low, high):
            i = (low - 1)
            pivot = arr[high]
            for j in range(low, high):
                if arr[j] <= pivot:
                    i = i + 1
                    arr[i], arr[j] = arr[j], arr[i]
            arr[i + 1], arr[high] = arr[high], arr[i + 1]
            return (i + 1)

        def quickSort(arr, low, high):
            if len(arr) == 1:
                return arr
            if low < high:
                pi = partition(arr, low, high)
                quickSort(arr, low, pi - 1)
                quickSort(arr, pi + 1, high)

        def readFile(filename, l):
            with open(filename, "r") as f:
                for line in f:
                    l.append(int(line))
            if f.closed == False:
                f.close()

        def main():
            txt_array = []
            readFile('1000_random.txt', txt_array)
            size = len(txt_array)
            quickSort(txt_array, 0, size - 1)

        main()
        #End time
        et = time.time()
        # get the execution time
        elapsed_time = (et - st )
        print('Execution time:', elapsed_time, 's')
        elapsed_time_ms = round(elapsed_time *1000,2)
        print('Execution time:', elapsed_time_ms, 'ms')

        Execution time: 0.004407167434692383 s
        Execution time: 4.41 ms
```

Sample 5: quicksort average case

3.2.3. Worst case

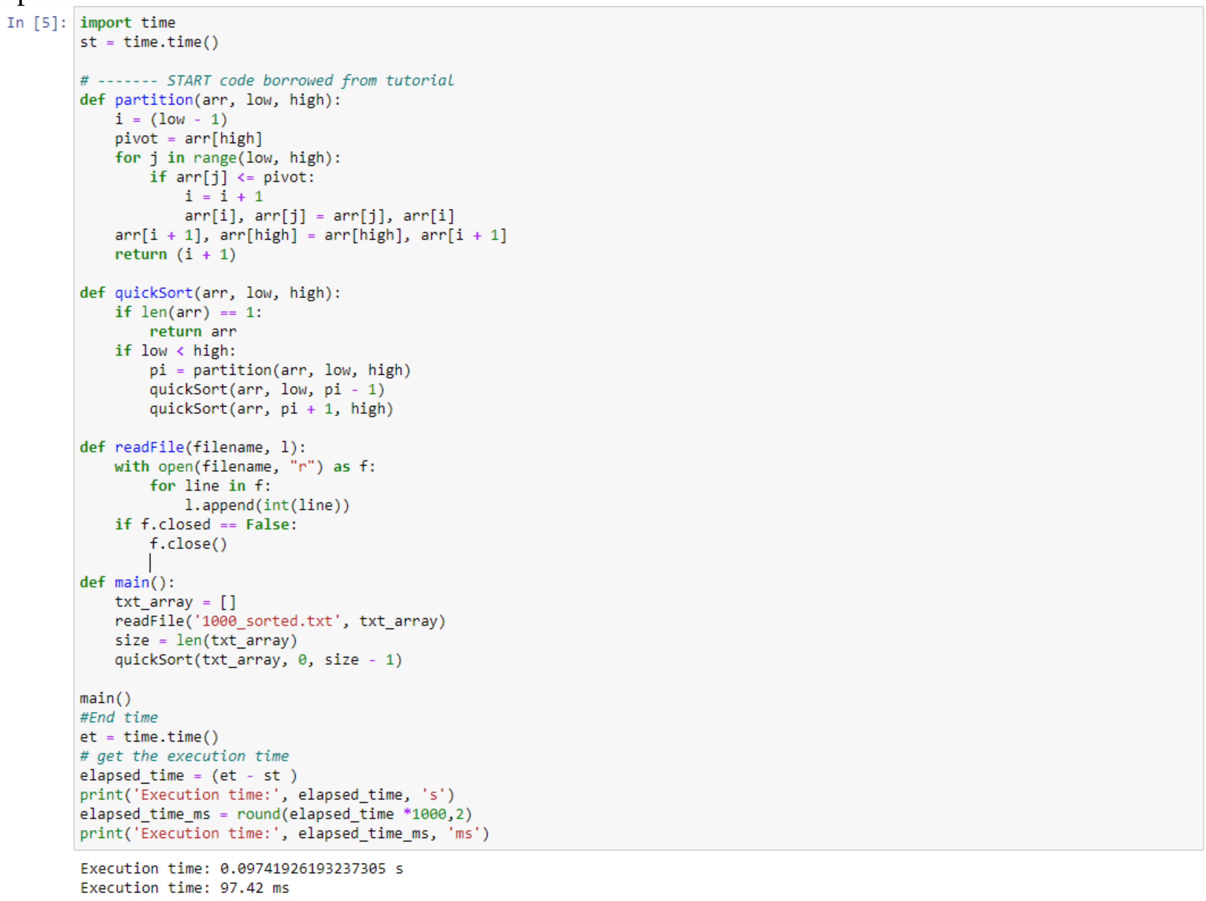
A quicksort algorithm running time in the worst-case is Q(n ^{2}). In this case, we select the pivot based on the highest or lowest element in the list. Consequently, the partition will have no elements on one side. As long as the pivot is the maximum or minimum element in the input list, we expect the experimental results to be in line with the theoretical results.

Calculate execution time:

Worst Case				
Input	Time (1 ms)	Time (2 ms)	Time (3 ms)	Average Of Times
50	1.00	1.09	1.06	1.05
100	1.98	1.48	2.99	2.15
500	32.06	28.93	30.53	30.51
1000	101.36	97.81	97.42	98.86

Table 7: Quick sort worst case

Sample:

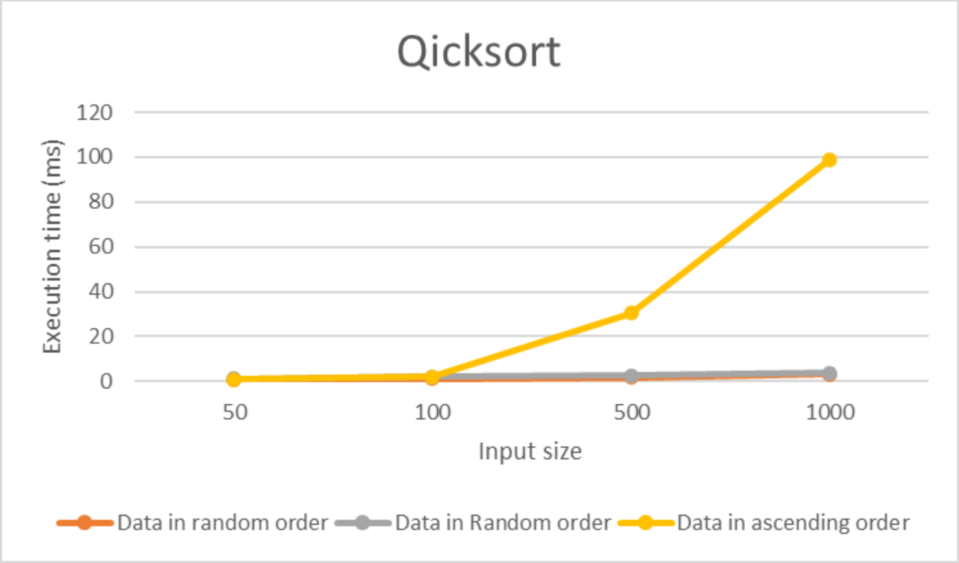


```
In [5]: import time
st = time.time()

# ------- START code borrowed from tutorial
def partition(arr, low, high):
    i = (low - 1)
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)

def readFile(filename, l):
    with open(filename, "r") as f:
        for line in f:
            l.append(int(line))
    if f.closed == False:
        f.close()

def main():
    txt_array = []
    readFile('1000_sorted.txt', txt_array)
    size = len(txt_array)
    quickSort(txt_array, 0, size - 1)

main()
#End time
et = time.time()
# get the execution time
elapsed_time = (et - st )
print('Execution time:', elapsed_time, 's')
elapsed_time_ms = round(elapsed_time *1000,2)
print('Execution time:', elapsed_time_ms, 'ms')
```

```
Execution time: 0.09741926193237305 s
Execution time: 97.42 ms
```

Sample 6: quicksort worst case

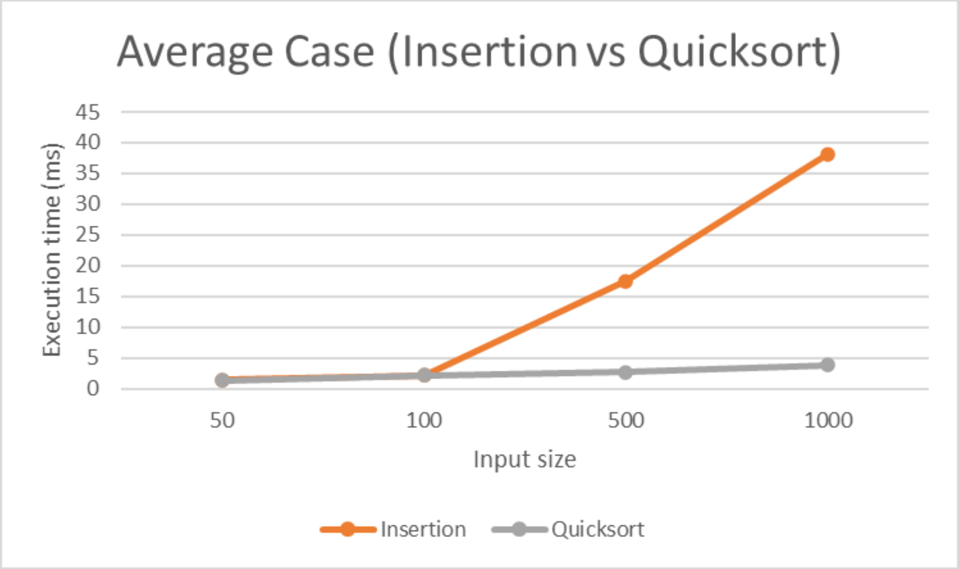
3.3. Graph:



Graph 2: Quick sort

The following graph illustrates the execution time of four inputs with sizes of 50,100,500, and 1000. We applied these inputs three times: once when the array was sorted ascendingly, once with a randomly chosen pivot, and once with a median element as a pivot. Compared to the other cases, the random sorting case with the median pivot has the shortest execution time, making it the best candidate for the quick sorting algorithm.

4. Comparison between quick sort and insertion sort



Graph 3: Average case (insertion vs quick sort)

In the above figure, we illustrate the average case for both the insertion and quick sort implementation for four different input sizes: 50, 100, 500, and 1000. Considering the significant difference in execution times between the two algorithms, we found that **100 is our threshold value**. And that the quick sort is faster than the insertion sort after the threshold value.

5. Hybrid Sort (Quick & insertion):

By combining more than one algorithm, the Hybrid algorithm achieves its name. In general, hybrid sorting is used to enhance the performance of the algorithms and make them run more efficiently. In some cases, the algorithm might switch back and forth between the two algorithms based on whether it is below or above the threshold.

5.1 Source Code

```
#hybird_sorting
import time
st = time.time()

def insertion_sort(arr, low, n):
    for i in range(low + 1, n + 1):
        val = arr[i]
        j = i
        while j > low and arr[j - 1] > val:
            arr[j] = arr[j - 1]
            j -= 1
        arr[j] = val


def partition(arr, low, high):
    pivot = arr[high]
    i = j = low
    for i in range(low, high):
        if arr[i] < pivot:
            arr[i], arr[j] = arr[j], arr[i]
            j += 1
    arr[j], arr[high] = arr[high], arr[j]
    return j

def quick_sort(arr, low, high):
    if low < high:
        pivot = partition(arr, low, high)
        quick_sort(arr, low, pivot - 1)
        quick_sort(arr, pivot + 1, high)
        return arr


# Hybrid function -> Quick + Insertion sort
def hybrid_quick_sort(arr, low, high):
    while low < high:

        # If the size of the array is less
        # than threshold apply insertion sort
        # and stop recursion
        if high - low + 1 < 100:
            insertion_sort(arr, low, high)
            break

        else:
            pivot = partition(arr, low, high)

            # Optimised quicksort which works on
            # the smaller arrays first

            # If the left side of the pivot
            # is less than right, sort left part
            # and move to the right part of the array
            if pivot - low < high - pivot:
                hybrid_quick_sort(arr, low, pivot - 1)
                low = pivot + 1
            else:
                # If the right side of pivot is less
                # than left, sort right side and
                # move to the left side
                hybrid_quick_sort(arr, pivot + 1, high)
                high = pivot - 1


def readFile(filename, l):
    with open(filename, "r") as f:
        for line in f:
            l.append(int(line))

    if f.closed == False:
        f.close()


def main():
    txt_array = []
    readFile("1000_random_with_middle.txt", txt_array)
    size = len(txt_array)
    hybrid_quick_sort(txt_array, 0, size - 1)

main()

#End time
et = time.time()

# get the execution time
elapsed_time = (et - st )
print('Execution time:', elapsed_time, 's')
elapsed_time_ms = round(elapsed_time *1000,2)
print('Execution time:', elapsed_time_ms, 'ms')
```

5.2 Hybrid sort analysis

It is a combination of the two algorithms (insertion and quick sort) both are applied but the decision of which one to use depends on the value of the input. Whether it is below or above the threshold. The hybrid sort has the least execution time out of the three.

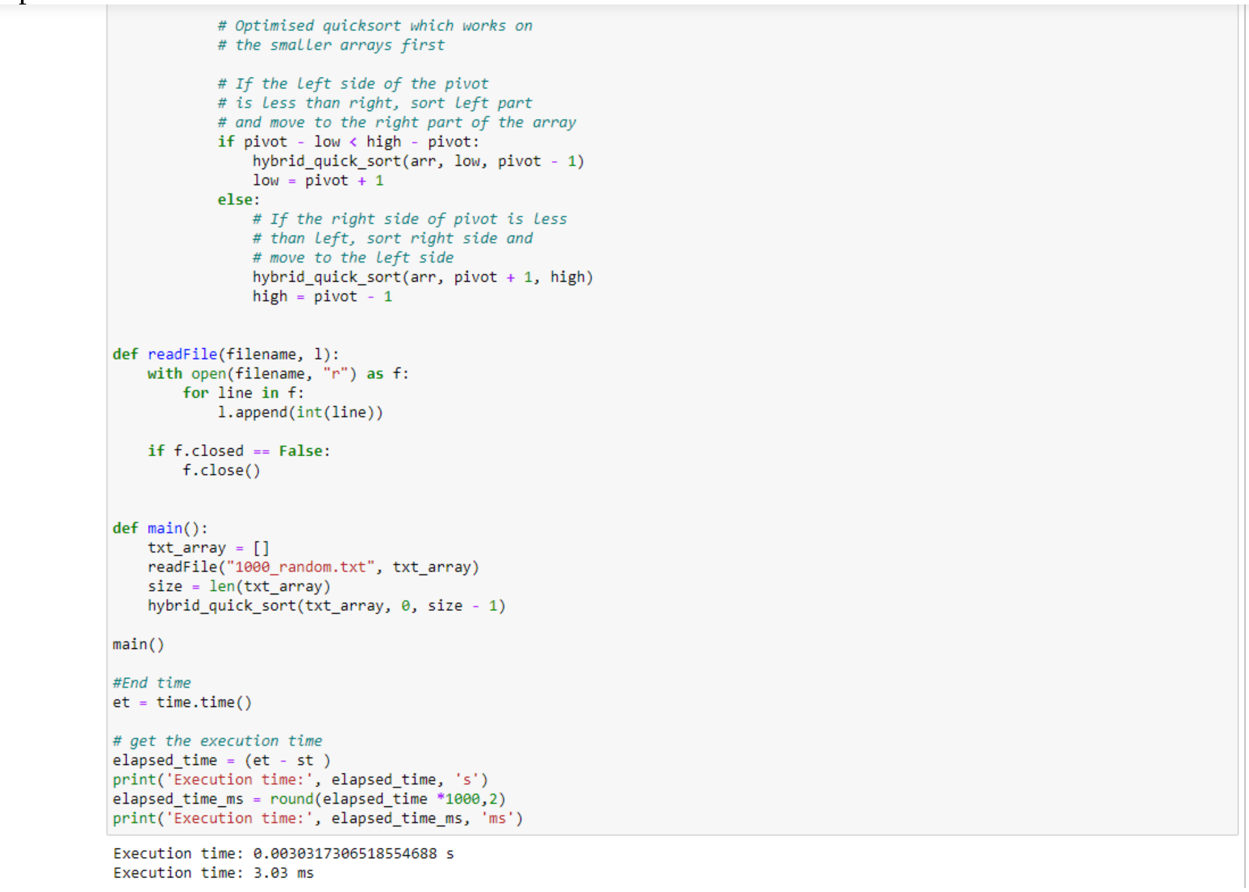
5.2.1 Average case

Calculate execution time:

Average Case				
Input	Time (1 ms)	Time (2 ms)	Time (3 ms)	Average Of Times
50	1.00	1.00	0.72	0.91
100	1.99	1.00	1.99	1.66
500	2.99	1.99	2.99	2.66
1000	4.67	3.66	3.03	3.79

Table 8: Hybird sort average case

Sample:



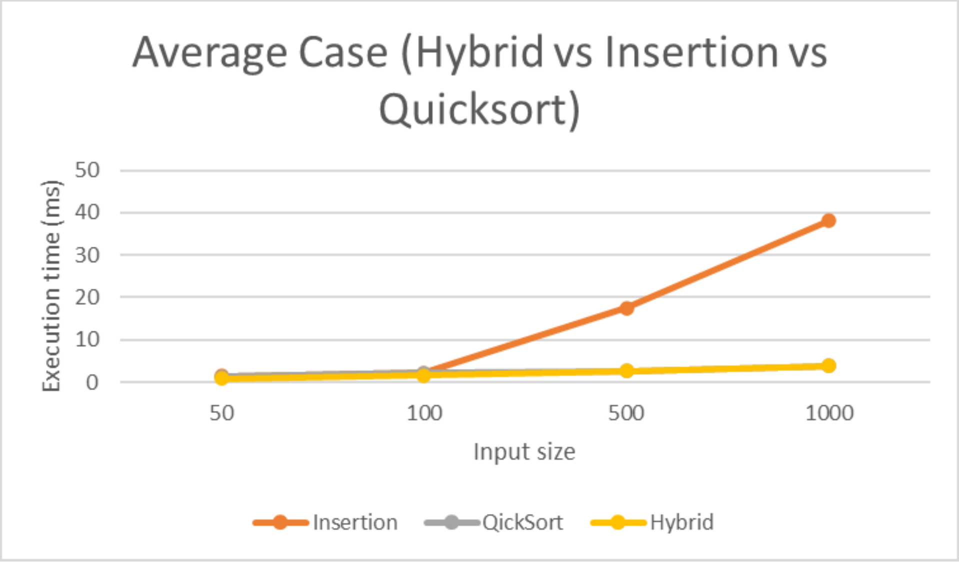
```
            # Optimised quicksort which works on
            # the smaller arrays first

            # If the left side of the pivot
            # is less than right, sort left part
            # and move to the right part of the array
            if pivot - low < high - pivot:
                hybrid_quick_sort(arr, low, pivot - 1)
                low = pivot + 1
            else:
                # If the right side of pivot is less
                # than left, sort right side and
                # move to the left side
                hybrid_quick_sort(arr, pivot + 1, high)
                high = pivot - 1


def readFile(filename, l):
    with open(filename, "r") as f:
        for line in f:
            l.append(int(line))

    if f.closed == False:
        f.close()


def main():
    txt_array = []
    readFile("1000_random.txt", txt_array)
    size = len(txt_array)
    hybrid_quick_sort(txt_array, 0, size - 1)

main()

#End time
et = time.time()

# get the execution time
elapsed_time = (et - st )
print('Execution time:', elapsed_time, 's')
elapsed_time_ms = round(elapsed_time *1000,2)
print('Execution time:', elapsed_time_ms, 'ms')
```

```
Execution time: 0.0030317306518554688 s
Execution time: 3.03 ms
```

Sample 7: hybrid sort average case

6. Comparison between quick sort and insertion sort and the hybrid

6.1 Graph:



Graph 4: Average case (hybird vs insertion vs quick sort)

In the above figure, we illustrate the average case for the insertion and quick sort and the hybrid sort implementation on four different input sizes: 50, 100, 500, and 1000. Due to the significant difference in execution times between the two algorithms, we found that **100 is our threshold value**. Thus, it is the transitioning value for the hybrid sort algorithm.

7. Questions & Answers

7.1 Describe how you obtained the value of the threshold.

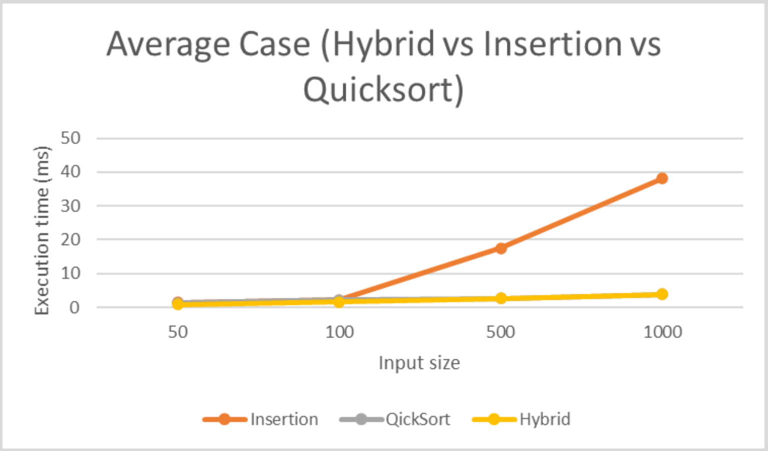
After trying out multiple input values for the 2 algorithms, we observed the behavior of the execution time. The result we obtained is that before the value 100 (the threshold value) the insertion sort produced better execution times than the quicksort. Yet, after 100, the quick sort gave better results, which is why we used the value 100 as the threshold value while designing the hybrid sort algorithm.

7.2 Describe the numbers of input data you have tested of each length. Take care to choose the sizes of your input appropriately.

We have used 4 datafiles with different inputs: 50,100,500 and 1000. The reason why we chose these values specifically is because they gave us valuable information about the threshold value, leading to a better visualization when designing the hybrid algorithm.

7.3 A description of the observed behavior of the three algorithms for values of n both smaller and larger than the cross-over value determined experimentally.

In the below figure, we illustrate the average case for the insertion and quick sort and the hybrid sort implementation on four different input sizes: 50, 100, 500, and 1000. We observed that based on the smaller input (50) the insertion sort gave a better execution time, so when we designed the hybrid to be using the code of the insertion sort algorithm when inputs are below the threshold value. On the other hand, when we used a larger input (1000) we saw that the quick sort algorithm produces better execution times, which is the reason why we designed the hybrid algorithm to use its code with larger inputs.



Graph 5: Average case (hybrid vs insertion vs quick sort)