|  |  |
| --- | --- |
| Ex. No: 5 | UIT2201 — Programming and Data Structures |
| 05-05-2023 |

**Aim:**

To execute the following programs and note the output.

**PART – A**

1. Arrange n elements either in ascending or descending order using Bubble sort and Selection sort. Write a Python function to sort n numbers and analyze the time complexity of your code and express the same in asymptotic notation. Give your inference on the performance of these sorting algorithms in terms of the number of comparisons and number of swaps performed. Try the best case, worst case and average case scenarios and report your observations.

**Code:**

*"""  
 This module provides functionality for checking the number of comparisons,  
 number of swappings and the time taken for sorting for various sizes of arrays.  
  
 This output can be used for ratio analysis of algorithms by taking n as the size of the array.  
  
 Original Author: Pranesh Kumar  
  
 Created on: 03 May 2023  
"""  
  
# importing the necessary modules*import timeit  
import random  
  
  
def bubblesort(seq, reverse=False):  
 *"""This function sorts the given sequence and returns a new sorted sequence based on bubble sorting.  
  
 Args: seq (Iterable): Sequence to be sorted  
 reverse (bool, optional): It should be set as True if the sequence is  
 to be sorted in descending order. Defaults to False.  
  
 Returns:  
 List: Sorted sequence  
 """* start = timeit.default\_timer() *# setting the start time* myseq = seq  
 seqlen = len(seq)  
 comparisoncount = 0  
 swappingcount = 0  
  
 for idx in range(seqlen - 1): *# outer for loop for iterating through the elements* for j in range(seqlen - idx - 1): *# inner for loop for number of passes for each iteration* if reverse: *# if reverse is True, then sort in descending* comparisoncount += 1  
 if myseq[j] < myseq[j + 1]:  
 swappingcount += 1  
 myseq[j], myseq[j + 1] = myseq[j + 1], myseq[j] *# swapping the current element and the next element* else: *# if reverse is False, then sort in ascending* comparisoncount += 1  
 if myseq[j] > myseq[j + 1]:  
 swappingcount += 1  
 myseq[j], myseq[j + 1] = myseq[j + 1], myseq[j] *# swapping the current element and the next element* exectime = timeit.default\_timer() - start *# finding the time taken* runtimedata = {"comparisons": comparisoncount, "swappings": swappingcount,  
 "exec": exectime} *# dictionary of runtime data* return myseq, runtimedata  
  
  
def selectionsort(seq, reverse=False):  
 *"""This function sorts the given sequence and returns a new sorted sequence based on selection sorting.  
  
 Args: seq (Iterable): Sequence to be sorted reverse (bool, optional): It should be set as True if the sequence is  
 to be sorted in descending order. Defaults to False.  
  
 Returns:  
 List: Sorted Sequence  
 """* start = timeit.default\_timer() *# setting the start time* seqlen = len(seq)  
 comparisoncount = 0  
 swappingcount = 0  
  
 for idx in range(seqlen): *# iterating through the list* min\_idx = idx *# setting the minimum index(element) as the first index(element)* for j in range(idx + 1, seqlen): *# eliminating the previous index element as it is already sorted* if reverse: *# if reverse is True, then sort in ascending* comparisoncount += 1  
 if seq[j] > seq[min\_idx]:  
 min\_idx = j  
 else: *# if reverse is False, then sort in ascending* comparisoncount += 1  
 if seq[j] < seq[min\_idx]:  
 min\_idx = j  
 swappingcount += 1  
 seq[idx], seq[min\_idx] = seq[min\_idx], seq[idx] *# Swapping current element and the relative first element* exectime = timeit.default\_timer() - start *# finding the time taken* runtimedata = {"comparisons": comparisoncount, "swappings": swappingcount,  
 "exec": exectime} *# dictionary of runtime data* return seq, runtimedata  
  
  
*# driver code*if \_\_name\_\_ == "\_\_main\_\_":  
 f = open("sorting.txt", "w")  
 sizes = [1, 10, 50, 100, 500, 1000, 5000, 10000]  
 for size in sizes:  
 bcomparisons = 0.0  
 bswappings = 0.0  
 bexecutiontime = 0.0  
 scomparisons = 0.0  
 sswappings = 0.0  
 sexecutiontime = 0.0  
 testcasecount = 5  
 for i in range(testcasecount):  
 mylist = [random.randint(-10000, 10000) for \_ in range(size)]  
  
 brunningdata = bubblesort(mylist.copy())[1]  
 bcomparisons += brunningdata['comparisons']  
 bswappings += brunningdata['swappings']  
 bexecutiontime += brunningdata['exec']  
  
 srunningdata = selectionsort(mylist.copy())[1]  
 scomparisons += srunningdata['comparisons']  
 sswappings += srunningdata['swappings']  
 sexecutiontime += srunningdata['exec']  
 print("==========================Bubble Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: {size}", file=f)  
 print(f"Number of comparisons: {bcomparisons / testcasecount}", file=f)  
 print(f"Number of swappings: {bswappings / testcasecount}", file=f)  
 print(f"Execution time: {bexecutiontime / testcasecount}", file=f)  
 print("================================================================", file=f)  
  
 print("==========================Selection Sort========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: {size}", file=f)  
 print(f"Number of comparisons: {scomparisons / testcasecount}", file=f)  
 print(f"Number of swappings: {sswappings / testcasecount}", file=f)  
 print(f"Execution time: {sexecutiontime / testcasecount}", file=f)  
 print("================================================================", file=f)  
  
 print("Best Case for size 10".center(64, "="), file=f)  
 mylist = [random.randint(-10000, 10000) for \_ in range(10)]  
 mylist.sort()  
  
 brunningdata = bubblesort(mylist.copy())[1]  
 srunningdata = selectionsort(mylist.copy())[1]  
  
 print("==========================Bubble Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {brunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {brunningdata['swappings']}", file=f)  
 print(f"Execution time: {brunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 print("==========================Selection Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {srunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {srunningdata['swappings']}", file=f)  
 print(f"Execution time: {srunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 print("Worst Case for size 10".center(64, "="), file=f)  
 mylist = [random.randint(-10000, 10000) for \_ in range(10)]  
 mylist.sort(reverse=True)  
  
 brunningdata = bubblesort(mylist.copy())[1]  
 srunningdata = selectionsort(mylist.copy())[1]  
  
 print("==========================Bubble Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {brunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {brunningdata['swappings']}", file=f)  
 print(f"Execution time: {brunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 print("==========================Selection Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {srunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {srunningdata['swappings']}", file=f)  
 print(f"Execution time: {srunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 f.close()

**Explanation:**

This module provides functionality to analyze the performance of two sorting algorithms, bubble sort and selection sort, for different sizes of arrays. It calculates the number of comparisons, number of swappings, and the time taken for sorting, and outputs the results to a file named "sorting.txt".

The module defines two functions, **bubblesort** and **selectionsort**, which take an iterable and an optional **reverse** boolean argument as input, and return a sorted list and a dictionary of runtime data including the number of comparisons, number of swappings, and execution time.

The **main** function runs five test cases for each array size defined in the **sizes** list, and calculates the average number of comparisons, number of swappings, and execution time for both bubble sort and selection sort. The results are then written to the file "sorting.txt".

Overall, this module provides a useful tool for comparing the efficiency of bubble sort and selection sort for different array sizes.

**Output:**

sorting.txt

==========================Bubble Sort===========================  
================================================================  
Size: 1  
Number of comparisons: 0.0  
Number of swappings: 0.0  
Execution time: 1.200009137392044e-06  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 1  
Number of comparisons: 0.0  
Number of swappings: 1.0  
Execution time: 2.500019036233425e-06  
================================================================  
==========================Bubble Sort===========================  
================================================================  
Size: 10  
Number of comparisons: 45.0  
Number of swappings: 22.4  
Execution time: 1.1299969628453255e-05  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 10  
Number of comparisons: 45.0  
Number of swappings: 10.0  
Execution time: 7.940037176012992e-06  
================================================================  
==========================Bubble Sort===========================  
================================================================  
Size: 50  
Number of comparisons: 1225.0  
Number of swappings: 622.0  
Execution time: 0.00023409996647387742  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 50  
Number of comparisons: 1225.0  
Number of swappings: 50.0  
Execution time: 0.00012672001030296088  
================================================================  
==========================Bubble Sort===========================  
================================================================  
Size: 100  
Number of comparisons: 4950.0  
Number of swappings: 2389.0  
Execution time: 0.0009060000069439411  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 100  
Number of comparisons: 4950.0  
Number of swappings: 100.0  
Execution time: 0.00048786001279950143  
================================================================  
==========================Bubble Sort===========================  
================================================================  
Size: 500  
Number of comparisons: 124750.0  
Number of swappings: 61773.6  
Execution time: 0.02402332001365721  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 500  
Number of comparisons: 124750.0  
Number of swappings: 500.0  
Execution time: 0.01284205997362733  
================================================================  
==========================Bubble Sort===========================  
================================================================  
Size: 1000  
Number of comparisons: 499500.0  
Number of swappings: 244615.2  
Execution time: 0.09847341999411582  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 1000  
Number of comparisons: 499500.0  
Number of swappings: 1000.0  
Execution time: 0.050922240037471055  
================================================================  
==========================Bubble Sort===========================  
================================================================  
Size: 5000  
Number of comparisons: 12497500.0  
Number of swappings: 6256318.8  
Execution time: 2.498145160009153  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 5000  
Number of comparisons: 12497500.0  
Number of swappings: 5000.0  
Execution time: 1.2344569799955933  
================================================================  
==========================Bubble Sort===========================  
================================================================  
Size: 10000  
Number of comparisons: 49995000.0  
Number of swappings: 24976031.0  
Execution time: 10.779710840038025  
================================================================  
==========================Selection Sort========================  
================================================================  
Size: 10000  
Number of comparisons: 49995000.0  
Number of swappings: 10000.0  
Execution time: 5.313462179969065  
================================================================  
=====================Best Case for size 10======================  
==========================Bubble Sort===========================  
================================================================  
Size: 10  
Number of comparisons: 45  
Number of swappings: 0  
Execution time: 1.0899966582655907e-05  
================================================================  
==========================Selection Sort===========================  
================================================================  
Size: 10  
Number of comparisons: 45  
Number of swappings: 10  
Execution time: 7.999944500625134e-06  
================================================================  
=====================Worst Case for size 10=====================  
==========================Bubble Sort===========================  
================================================================  
Size: 10000  
Number of comparisons: 45  
Number of swappings: 45  
Execution time: 1.750001683831215e-05  
================================================================  
==========================Selection Sort===========================  
================================================================  
Size: 10000  
Number of comparisons: 45  
Number of swappings: 10  
Execution time: 8.999952115118504e-06  
================================================================

2. Write a Python function to sort n numbers using Insertion sort and analyze the time complexity of your code using the number of comparisons and swaps required and express the same in asymptotic notation. Try the best case, worst case and average case scenarios and report your observations.

**Code:**

*"""  
 This module provides functionality for checking the number of comparisons,  
 number of swappings and the time taken for sorting for various sizes of arrays.  
  
 This output can be used for ratio analysis of algorithms by taking n as the size of the array.  
  
 Original Author: Pranesh Kumar  
  
 Created on: 03 May 2023  
"""  
  
# importing the necessary modules*import timeit  
import random  
  
  
def bubblesort(seq, reverse=False):  
 *"""This function sorts the given sequence and returns a new sorted sequence based on bubble sorting.  
  
 Args: seq (Iterable): Sequence to be sorted  
 reverse (bool, optional): It should be set as True if the sequence is  
 to be sorted in descending order. Defaults to False.  
  
 Returns:  
 List: Sorted sequence  
 """* start = timeit.default\_timer() *# setting the start time* myseq = seq  
 seqlen = len(seq)  
 comparisoncount = 0  
 swappingcount = 0  
  
 for idx in range(seqlen - 1): *# outer for loop for iterating through the elements* for j in range(seqlen - idx - 1): *# inner for loop for number of passes for each iteration* if reverse: *# if reverse is True, then sort in descending* comparisoncount += 1  
 if myseq[j] < myseq[j + 1]:  
 swappingcount += 1  
 myseq[j], myseq[j + 1] = myseq[j + 1], myseq[j] *# swapping the current element and the next element* else: *# if reverse is False, then sort in ascending* comparisoncount += 1  
 if myseq[j] > myseq[j + 1]:  
 swappingcount += 1  
 myseq[j], myseq[j + 1] = myseq[j + 1], myseq[j] *# swapping the current element and the next element* exectime = timeit.default\_timer() - start *# finding the time taken* runtimedata = {"comparisons": comparisoncount, "swappings": swappingcount,  
 "exec": exectime} *# dictionary of runtime data* return myseq, runtimedata  
  
  
def selectionsort(seq, reverse=False):  
 *"""This function sorts the given sequence and returns a new sorted sequence based on selection sorting.  
  
 Args: seq (Iterable): Sequence to be sorted reverse (bool, optional): It should be set as True if the sequence is  
 to be sorted in descending order. Defaults to False.  
  
 Returns:  
 List: Sorted Sequence  
 """* start = timeit.default\_timer() *# setting the start time* seqlen = len(seq)  
 comparisoncount = 0  
 swappingcount = 0  
  
 for idx in range(seqlen): *# iterating through the list* min\_idx = idx *# setting the minimum index(element) as the first index(element)* for j in range(idx + 1, seqlen): *# eliminating the previous index element as it is already sorted* if reverse: *# if reverse is True, then sort in ascending* comparisoncount += 1  
 if seq[j] > seq[min\_idx]:  
 min\_idx = j  
 else: *# if reverse is False, then sort in ascending* comparisoncount += 1  
 if seq[j] < seq[min\_idx]:  
 min\_idx = j  
 swappingcount += 1  
 seq[idx], seq[min\_idx] = seq[min\_idx], seq[idx] *# Swapping current element and the relative first element* exectime = timeit.default\_timer() - start *# finding the time taken* runtimedata = {"comparisons": comparisoncount, "swappings": swappingcount,  
 "exec": exectime} *# dictionary of runtime data* return seq, runtimedata  
  
  
*# driver code*if \_\_name\_\_ == "\_\_main\_\_":  
 f = open("sorting.txt", "w")  
 sizes = [1, 10, 50, 100, 500, 1000, 5000, 10000]  
 for size in sizes:  
 bcomparisons = 0.0  
 bswappings = 0.0  
 bexecutiontime = 0.0  
 scomparisons = 0.0  
 sswappings = 0.0  
 sexecutiontime = 0.0  
 testcasecount = 5  
 for i in range(testcasecount):  
 mylist = [random.randint(-10000, 10000) for \_ in range(size)]  
  
 brunningdata = bubblesort(mylist.copy())[1]  
 bcomparisons += brunningdata['comparisons']  
 bswappings += brunningdata['swappings']  
 bexecutiontime += brunningdata['exec']  
  
 srunningdata = selectionsort(mylist.copy())[1]  
 scomparisons += srunningdata['comparisons']  
 sswappings += srunningdata['swappings']  
 sexecutiontime += srunningdata['exec']  
 print("==========================Bubble Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: {size}", file=f)  
 print(f"Number of comparisons: {bcomparisons / testcasecount}", file=f)  
 print(f"Number of swappings: {bswappings / testcasecount}", file=f)  
 print(f"Execution time: {bexecutiontime / testcasecount}", file=f)  
 print("================================================================", file=f)  
  
 print("==========================Selection Sort========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: {size}", file=f)  
 print(f"Number of comparisons: {scomparisons / testcasecount}", file=f)  
 print(f"Number of swappings: {sswappings / testcasecount}", file=f)  
 print(f"Execution time: {sexecutiontime / testcasecount}", file=f)  
 print("================================================================", file=f)  
  
 print("Best Case for size 10".center(64, "="), file=f)  
 mylist = [random.randint(-10000, 10000) for \_ in range(10)]  
 mylist.sort()  
  
 brunningdata = bubblesort(mylist.copy())[1]  
 srunningdata = selectionsort(mylist.copy())[1]  
  
 print("==========================Bubble Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {brunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {brunningdata['swappings']}", file=f)  
 print(f"Execution time: {brunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 print("==========================Selection Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {srunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {srunningdata['swappings']}", file=f)  
 print(f"Execution time: {srunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 print("Worst Case for size 10".center(64, "="), file=f)  
 mylist = [random.randint(-10000, 10000) for \_ in range(10)]  
 mylist.sort(reverse=True)  
  
 brunningdata = bubblesort(mylist.copy())[1]  
 srunningdata = selectionsort(mylist.copy())[1]  
  
 print("==========================Bubble Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {brunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {brunningdata['swappings']}", file=f)  
 print(f"Execution time: {brunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 print("==========================Selection Sort===========================", file=f)  
 print("================================================================", file=f)  
 print(f"Size: 10", file=f)  
 print(f"Number of comparisons: {srunningdata['comparisons']}", file=f)  
 print(f"Number of swappings: {srunningdata['swappings']}", file=f)  
 print(f"Execution time: {srunningdata['exec']}", file=f)  
 print("================================================================", file=f)  
  
 f.close()

**Explanation:**

This code is an implementation of the insertion sort algorithm, which sorts a given sequence in ascending order. It also provides functionality for measuring the number of comparisons, number of overwritings, and the time taken for sorting for various sizes of arrays.

The **insertionsort** function takes an iterable **seq** as input and sorts it using the insertion sort algorithm. It returns a tuple consisting of the sorted sequence and a dictionary containing runtime data like the number of comparisons, overwritings, and execution time.

The **main** function uses the **insertionsort** function to sort arrays of various sizes (1, 10, 50, 100, 500, 1000, 5000, and 10000) and measures their performance. For each size, it generates five test cases of random integers between -10000 and 10000 and measures the number of comparisons, overwritings, and execution time. It then writes the results to a file called **insertionsort.txt**.

Finally, the code tests the best and worst cases of size 10, where the best case is a sorted list, and the worst case is a reversed list. It measures the same runtime data as before for these cases and writes them to the same file.

Overall, this code is useful for analyzing the runtime performance of the insertion sort algorithm and comparing it with other sorting algorithms by using the ratio analysis method.

**Output:**

insertionsorting.txt

==========================Insertion Sort========================  
================================================================  
Size: 1  
Number of comparisons: 0.0  
Number of overwritings: 0.0  
Execution time: 1.1600000000000499e-06  
================================================================  
==========================Insertion Sort========================  
================================================================  
Size: 10  
Number of comparisons: 19.4  
Number of overwritings: 19.4  
Execution time: 4.319999999999324e-06  
================================================================  
==========================Insertion Sort========================  
================================================================  
Size: 50  
Number of comparisons: 663.0  
Number of overwritings: 663.0  
Execution time: 8.716000000000279e-05  
================================================================  
==========================Insertion Sort========================  
================================================================  
Size: 100  
Number of comparisons: 2543.0  
Number of overwritings: 2543.0  
Execution time: 0.00026323999999999516  
================================================================  
==========================Insertion Sort========================  
================================================================  
Size: 500  
Number of comparisons: 62857.6  
Number of overwritings: 62857.6  
Execution time: 0.0061543000000000014  
================================================================  
==========================Insertion Sort========================  
================================================================  
Size: 1000  
Number of comparisons: 252100.2  
Number of overwritings: 252100.2  
Execution time: 0.026573080000000006  
================================================================  
==========================Insertion Sort========================  
================================================================  
Size: 5000  
Number of comparisons: 6222192.4  
Number of overwritings: 6222192.4  
Execution time: 0.6533121800000001  
================================================================  
==========================Insertion Sort========================  
================================================================  
Size: 10000  
Number of comparisons: 25013194.8  
Number of overwritings: 25013194.8  
Execution time: 2.6780067599999997  
================================================================  
=====================Best Case for size 10======================  
==========================Insertion Sort========================  
================================================================  
Size: 10  
Number of comparisons: 0  
Number of overwritings: 0  
Execution time: 2.9999999995311555e-06  
================================================================  
=====================Worst Case for size 10=====================  
==========================Bubble Sort===========================  
================================================================  
Size: 10  
Number of comparisons: 45  
Number of overwritings: 45  
Execution time: 5.699999999109195e-06  
================================================================

**Ratio Analysis:**

A picture containing table

Description automatically generated

Table

Description automatically generated with medium confidence

Table

Description automatically generated

Here, if we analyse the table, we can notice that the value of f(n)/n2 tends to some finite value in all the 3 cases. So, we infer that the time complexity of three of the cases is the same and is O(n2).

Even though the time complexities are the same, insertion sort is the best sorting technique among the 3, because, in the best case, insertion sort performs better than selection or bubble and the elements are immediately sorted in the list with each iteration.

**========================**