Ex. No: 6 20-05-2023	UIT2201 — Programming and Data Structures

Aim:

To execute the following programs and note the output.

PART – A

1. Arrange n elements either in ascending or descending order using Merge sort. Write a Python function to sort n numbers and analyze the time complexity of your code and express the same in asymptotic notation. Also, write a recursive binary search function to search for an element in the above sorted list. Analyze the time complexity of your code and express the same in asymptotic notation.

Code:

```
"""
    This module provides functionality for checking the execution time and
    number of comparisons for sorting of various sequences using merge sort

    This output can be used for ratio analysis of algorithms by taking n as
the size of the array or
    as the number of comparisons.

    Original Author: Pranesh Kumar

    Created on 10 May 2023
"""

# importing the necessary modules
import random
import timeit

count = 0
starttime = 0


def merge(seq1, seq2):
    """
    This function performs merge operation on the two lists which are
passsed to the function.

    Args:
        seq1 (Iterable): First sequence to be merged
        seq2 (Iterable): Second sequence to be merged

    Returns:
        Merged list of seq1 and seq2 which is sorted
    """

    global count
    i = j = 0
```

```
    m = len(seq1)
    n = len(seq2)
    result = []
    while i < m and j < n:
        count += 1
        if seq1[i] < seq2[j]:
            result.append(seq1[i])
            i += 1
        else:
            result.append(seq2[j])
            j += 1
    while (i < m):
        result.append(seq1[i])
        i += 1
    while (j < n):
        result.append(seq2[j])
        j += 1
    return result


def mergesort(seq):
    """
    Performs recursive merge operation on list by using divide and conquer
technique

    Args:
        seq (Iterable) Sequene to be sorted

    Returns:
        Sorted sequence
    """
    length = len(seq)
    if length < 2:  # Base Condition
        return seq[:]
    else:
        mid = length // 2
        return merge(mergesort(seq[:mid]), mergesort(seq[mid:]))  #
Recursive call


def bestcasecomplexity(size):
    """
    Finds out the best case complexity of merge sort by passing an already
sorted list
    """
    myseq = [random.randint(-10000, 10000) for _ in range(size)]
    myseq.sort()
    mergesort(myseq)


def worstcasecomplexity(size):
    """
    Finds out the worst case complexity of merge sort by passing an already
sorted list in descending order
    """
    myseq = [random.randint(-10000, 10000) for _ in range(size)]
    myseq.sort(reverse=True)
    mergesort(myseq)


# driver code
```

```
if __name__ == "__main__":
    f1 = open("merge-sort-comp.txt", "w")
    f2 = open("merge-sort-time.txt", "w")

    sizes = [1, 10, 50, 100, 500, 1000, 5000, 10000]
    for size in sizes:
        count = 0
        starttime = timeit.default_timer()
        myseq = [random.randint(-10000, 10000) for _ in range(size)]
        mergesort(myseq)
        print("=" * 50, file=f1)
        print(f"Size: {size} \nNumber of comparisons: {count}", file=f1)
        print("=" * 50, file=f1)

        exectime = timeit.default_timer() - starttime
        print("=" * 50, file=f2)
        print(f"Size: {size} \nExecution Time: {exectime}", file=f2)
        print("=" * 50, file=f2)

    count = 0
    starttime = timeit.default_timer()
    bestcasecomplexity(10)
    print("=" * 50, file=f1)
    print(f"Best Case - Size: {10} \nNumber of comparisons: {count}",
file=f1)
    print("=" * 50, file=f1)

    exectime = timeit.default_timer() - starttime
    print("=" * 50, file=f2)
    print(f"Best Case - Size: {10} \nExecution Time: {exectime}", file=f2)
    print("=" * 50, file=f2)

    count = 0
    starttime = timeit.default_timer()
    worstcasecomplexity(10)
    print("=" * 50, file=f1)
    print(f"Worst Case - Size: {10} \nNumber of comparisons: {count}",
file=f1)
    print("=" * 50, file=f1)

    exectime = timeit.default_timer() - starttime
    print("=" * 50, file=f2)
    print(f"Worst Case - Size: {10} \nExecution Time: {exectime}", file=f2)
    print("=" * 50, file=f2)

    f1.close()
    f2.close()
```

Ratio Analysis of Merge Sorting												
Degree(n)	Number of comparisons (f(n))	Execution time	n	log n	n * log n	n^2	n^3	f(n)/n	f(n)/log n	f(n)/n * log n	$f(n)/n^2$	$f(n)/n^3$
1	0	9.06E-05	1	0	0	1	1	0	#DIV/0!	#DIV/0!	0	0
10	23	9.19E-05	10	3.32192809	33.2192809	100	1000	2.3	6.9236899	0.69236899	0.23	0.023
50	221	0.0003611	50	5.64385619	282.192809	2500	125000	4.42	39.157624	0.783152485	0.0884	0.001768
100	535	0.0007345	100	6.64385619	664.385619	10000	1000000	5.35	80.525524	0.805255238	0.0535	0.000535
500	3855	0.0040721	500	8.96578428	4482.89214	250000	125000000	7.71	429.96796	0.859935925	0.01542	0.00003084
1000	8721	0.0189045	1000	9.96578428	9965.78428	1000000	1000000000	8.721	875.0942	0.875094197	0.008721	8.721E-06
5000	55201	0.0470443	5000	12.2877124	61438.5619	25000000	1.25E+11	11.0402	4492.374	0.898474806	0.002208	4.4161E-07
10000	120477	0.1043038	10000	13.2877124	132877.124	100000000	1E+12	12.0477	9066.7977	0.90667977	0.0012048	1.2048E-07

Best Case	Size	10
	Number of Comparisons	15
	Execution Time	9.49E-05
Worst Case	Size	10
	Number of Comparisons	19
	Execution Time	8.26E-05

Observation:

From this ratio analysis table, we observe that f(n)/n*log n remains almost a constant around 0.8. So, the time complexity of merge sort is O(nlogn).

Ratio Analysis of Binary Search											
Degree(n)	Number of comparisons (f(n))	n	log n	n * log n	n^2	n^3	f(n)/n	f(n)/log n	f(n)/n * log n	$f(n)/n^2$	$f(n)/n^3$
1	2	1	0	0	1	1	2	#DIV/0!	#DIV/0!	2	2
10	8	10	3.32192809	33.219281	100	1000	0.8	2.40824	0.240823997	0.08	0.008
50	11	50	5.64385619	282.19281	2500	125000	0.22	1.949022	0.03898044	0.0044	0.000088
100	17	100	6.64385619	664.38562	10000	1000000	0.17	2.558755	0.02558755	0.0017	0.000017
500	20	500	8.96578428	4482.8921	250000	125000000	0.04	2.2307028	0.004461406	0.00008	1.6E-07
1000	29	1000	9.96578428	9965.7843	1000000	1E+09	0.029	2.9099566	0.002909957	0.000029	2.9E-08
5000	29	5000	12.2877124	61438.562	25000000	1.25E+11	0.0058	2.3600813	0.000472016	1.16E-06	2.32E-10
10000	35	10000	13.2877124	132877.12	1E+08	1E+12	0.0035	2.6340125	0.000263401	3.5E-07	3.5E-11

Observation:

From this ratio analysis table, we observe that f(n)/log n remains almost a constant around 1 to 2. So, the time complexity of binary search is O(logn).

2. Write a Python function to sort n numbers using Quick sort and analyze the time complexity of your code using the number of comparisons required and express the same in asymptotic notation.

Code:

```
"""
This module provides functionality for sorting a sequence using quicksort
and analyse
the time complexity by using the number of comparisons as f(n)

Original Author: Pranesh Kumar

Created on: 20 May 2023
"""
```

```
import random

count = 0


def qsort(seq, begin, end):
    """
    This functions sorts a sequence using quicksort algorithm recursively.
    Sorts the original sequence.
    Original sequence gets changed.
    Args:
        begin: Starting index from which sorting is to be done
        end: Ending index up to which sorting is to be done.
        seq: Iterable which needs to be sorted

    Returns:
        None
    """
    if begin < end:
        k = partition(seq, begin, end - 1)
        qsort(seq, begin, k - 1)
        qsort(seq, k + 1, end)


def partition(seq, begin, end):
    """
    Partitions a list based on a pivot so that the elements on the left of
pivot
    is less than pivot and elements on the right of pivot is greater than
the pivot.

    This function returns the middle element/ the index of the pivot in the
list
    Args:
        seq: sequence which needs to be partitioned based on pivot
        begin: starting index from which portioning is to be done
        end: ending index up to which portioning is to be done

    Returns:
        i (int): the index of pivot element in the sequence
    """
    global count
    find_median(begin, end, seq)
    pivot = seq[end]
    i = begin - 1
    for j in range(begin, end):
        count += 1
        if seq[j] <= pivot:
            i += 1

            seq[i], seq[j] = seq[j], seq[i]

    seq[i+1], seq[end] = seq[end], seq[i+1]

    return i+1


def find_median(begin, end, seq):
    """
    This function finds the median element and movies it to the end of the
list, so that
```

```
    it can be the pivot while using in the partition function.

    This changes the original sequence

    Args:
        begin: starting index from which median is to be found
        end: ending index up to which median is to be found
        seq: sequence of which median is to be found

    Returns:
        None
    """
    mid = (begin + end) // 2
    if seq[begin] > seq[mid]:
        seq[begin], seq[mid] = seq[mid], seq[begin]
    if seq[mid] > seq[end]:
        seq[mid], seq[end] = seq[end], seq[mid]
    if seq[begin] > seq[end]:
        seq[begin], seq[end] = seq[end], seq[begin]

    seq[mid], seq[end] = seq[end], seq[mid]


# driver code
if __name__ == "__main__":
    sizes = [1, 10, 50, 100, 500, 1000, 5000, 10000]
    for size in sizes:
        # size = int(input("Enter the size of list: "))
        my_seq = [random.randint(-10000, 10000) for _ in range(size)]
        qsort(my_seq, 0, size)
        print(my_seq)
        print(size, count)
        count = 0
```

Ratio Analysis of Quick Sort											
Degree(n)	Number of comparisons (f(n))	n	log n	n * log n	n^2	n^3	f(n)/n	f(n)/log n	f(n)/n * log n	$f(n)/n^2$	$f(n)/n^3$
1	0	1	0	0	1	1	0	#DIV/0!	#DIV/0!	0	0
10	16	10	3.32192809	33.219281	100	1000	1.6	4.8164799	0.481647993	0.16	0.016
50	194	50	5.64385619	282.19281	2500	125000	3.88	34.373661	0.687473222	0.0776	0.001552
100	593	100	6.64385619	664.38562	10000	1000000	5.93	89.255394	0.892553937	0.0593	0.000593
500	3914	500	8.96578428	4482.8921	250000	125000000	7.828	436.54854	0.873097071	0.015656	3.131E-05
1000	9225	1000	9.96578428	9965.7843	1000000	1E+09	9.225	925.66724	0.925667237	0.009225	9.225E-06
5000	60487	5000	12.2877124	61438.562	25000000	1.25E+11	12.0974	4922.5599	0.984511976	0.002419	4.839E-07
10000	146,965	10000	13.2877124	132877.12	1E+08	1E+12	14.6965	11060.218	1.106021833	0.00147	1.47E-07

Observation:

From this ratio analysis table, we observe that f(n)/n*log n remains almost a constant around 0.8 to 0.9. So, the time complexity of merge sort is O(nlogn).

==========