

		Bansilal Ramnath Agarwal Charitable Trust's Vishwakarma Institute of Information Technology	
		Department of Artificial Intelligence and Data Science	
Student Name: Mohammad Faiz Nishat Parvej Saiyad			
Class: TY	Division: A	Roll No: 371034	
Semester: V		Academic Year: 2022-23	
Subject Name & Code: Design and Analysis of Algorithms			
Title of Assignment: Implementation the following algorithm using Divide & Conquer method. (a)Merge sort (b) Quick Sort Also display execution time for different size of input and perform the analysis.			
Date of Performance:		Date of Submission:	

Aim:

Implementation the following algorithm using Divide & Conquer method.

(a)Merge sort (b) Quick Sort

Also display execution time for different size of input and perform the analysis.

Problem Statement:

Implement quick and merge sort and display execution time and perform analysis for different size of input

Software Requirements:

Text Editor: VSCode, Neovim, etc

Environment: Python 3.10

Terminal Emulator

Background Information:

Divide and Conquer :-

A divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems solving the sub-problems, and combining them to get the desired output.

Here are the steps involved:

1. Divide: Divide the given problem into sub-problems using recursion.
2. Conquer: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Merge Sort :-

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Quick Sort :-

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Code:

```
quick.py
21 def quicksort(l, r, arr):
20     if len(arr) == 1:
19         return arr
18
17     if l < r:
16         p = partition(l, r, arr)
15         quicksort(l, p - 1, arr)
14         quicksort(p + 1, r, arr)
13     return arr
12
11
10 def partition(l, r, arr) -> int:
9     pivot, pointer = arr[r], l
8
7     for i in range(l, r):
6         if arr[i] <= pivot:
5             arr[i], arr[pointer] = arr[pointer], arr[i]
4             pointer += 1
3
2     arr[pointer], arr[r] = arr[r], arr[pointer]
1     return pointer
22
```

+ ~/D/D/S/S/2/A/0/merge.py

```
13 def mergesort(arr):
12     if len(arr) == 1 or len(arr) == 0:
11         return arr
10
9     else:
8         mid = len(arr) // 2
7         return merge(mergesort(arr[:mid]), mergesort(arr[mid + 1 :]))
6
5
4 def merge(arr1, arr2):
3     arr = arr1 + arr2
2     arr.sort()
1     return arr
14
```

~/D/D/S/S/2/A/0/main.py

```
26 from quick import quicksort
25 from merge import mergesort
24 import random
23 from tests import create_best_case, create_test_case, create_worst_case
22 from rich import print
21 from rich.table import Table
20 from timeit import default_timer as timer
19 from datetime import timedelta
```

18

17

```
16 def main():
```

15

```
14     tb = Table()
```

```
13     tb.add_column("Test No")
```

```
12     tb.add_column("Quick Sort")
```

```
11     tb.add_column("Merge Sort")
```

10

```
9     for i in range(0, 5):
```

```
8         a, b = test(i)
```

```
7         tb.add_row(str(i), str(a), str(b))
```

6

```
5     qb, mb, qw, mw = test_best_and_worst_case()
```

4

```
3     tb.add_row("Best Case", str(qb), str(mb))
```

```
2     tb.add_row("Worst Case", str(qw), str(mw))
```

```
1     print(tb)
```

27

1

```
2 def test(iteration: int):      ■ "iteration" is not accessed
```

3

```
4     rand = random.randint(0, 100)
```

5

```
6     test = create_test_case(rand)
```

7

```
8     quick_start = timer()
```

```
9     quicksort(0, len(test) - 1, test)
```

```
10    quick_end = timer()
```

11

```
12    merge_start = timer()
```

```
13    mergesort(test)
```

```
14    merge_end = timer()
```

15

```
16    return (
```

```
17        timedelta(seconds=quick_end - quick_start),
```

```
18        timedelta(seconds=merge_end - merge_start),
```

```
19    )
```

20

~/D/D/S/S/2/A/0/main.py

```
27
26 def test_best_and_worst_case():
25     best_case = create_best_case(10)
24     worst_case = create_worst_case(10)
23     |
22     quick_start = timer()
21     quicksort(0, len(best_case) - 1, best_case)
20     quick_end = timer()
19     |
18     merge_start = timer()
17     mergesort(best_case)
16     merge_end = timer()
15     |
14     quick_worst_start = timer()
13     quicksort(0, len(worst_case) - 1, worst_case)
12     quick_worst_end = timer()
11     |
10     merge_worst_start = timer()
9     mergesort(worst_case)
8     merge_worst_end = timer()
7     |
6     return (
5         timedelta(seconds=quick_end - quick_start),
4         timedelta(seconds=merge_end - merge_start),
3         timedelta(seconds=quick_worst_end - quick_worst_start),
2         timedelta(seconds=merge_worst_end - merge_worst_start),
1     )
75 |
1
2 if __name__ == "__main__":
3     main()
```

~/D/D/S/S/2/A/0/tests.py

```
25 import numpy as np
24 import random
23
22
21 def create_test_case(x: int):
20     i = random.randint(0, 1000)
19     |
18     arr = np.arange(x, x + i)
17     |
16     arr = [*arr]
15     |
14     random.shuffle(arr)
13     |
12     return arr
11
10
9 def create_best_case(x: int):
8     i = random.randint(0, 1000)
7     |
6     arr = np.arange(x, x + i)
5     |
4     arr = [*arr]
3     |
2     return arr
1

26 ☐
1 def create_worst_case(x: int):
2     i = random.randint(0, 1000)
3     |
4     arr = np.arange(x, x + i)
5     |
6     arr = [*arr]
7     |
8     arr.reverse()
9     |
10    return arr
11
```

```
~/D/D/S/S/2/A/0/tests.py
14
13
12 if __name__ == "__main__":
11     case1 = create_test_case(30)
10     case2 = create_test_case(50)
9     case3 = create_test_case(90)
8     case4 = create_test_case(20)
7     case5 = create_test_case(40)
6
5     print(case1)
4     print(case2)
3     print(case3)
2     print(case4)
1     print(case5)
51
```

Output:

2. Design and Analysis of Algorithm/Assignments/02. Quick vs Merge Sort Time Analysis via `pip v3.10.7`
> python main.py

Test No	Quick Sort	Merge Sort
0	0:00:00.000134	0:00:00.000059
1	0:00:00.000012	0:00:00.000006
2	0:00:00.000135	0:00:00.000054
3	0:00:00.000558	0:00:00.000169
4	0:00:00.000760	0:00:00.000209
Best Case	0:00:00.000482	0:00:00.000026
Worst Case	0:00:00.017222	0:00:00.000200

2. Design and Analysis of Algorithm/Assignments/02. Quick vs Merge Sort Time Analysis via `pip v3.10.7`
> python main.py

Test No	Quick Sort	Merge Sort
0	0:00:00.000181	0:00:00.000085
1	0:00:00.000466	0:00:00.000122
2	0:00:00.000535	0:00:00.000147
3	0:00:00.000197	0:00:00.000059
4	0:00:00.000318	0:00:00.000096
Best Case	0:00:00.002599	0:00:00.000057
Worst Case	0:00:00.009186	0:00:00.000127


```
> python main.py
```

Test No	Quick Sort	Merge Sort
0	0:00:00.000260	0:00:00.000089
1	0:00:00.000285	0:00:00.000090
2	0:00:00.000130	0:00:00.000041
3	0:00:00.000732	0:00:00.000212
4	0:00:00.000409	0:00:00.000112
Best Case	0:00:00.000067	0:00:00.000010
Worst Case	0:00:00.003900	0:00:00.000090

2. Design and Analysis of Algorithm/Assignments/02. Quick vs Merge Sort Time Analysis via `pip v3.10.7`

```
> python main.py
```

Test No	Quick Sort	Merge Sort
0	0:00:00.001405	0:00:00.000497
1	0:00:00.000259	0:00:00.000109
2	0:00:00.000167	0:00:00.000063
3	0:00:00.001034	0:00:00.000326
4	0:00:00.000738	0:00:00.000229
Best Case	0:00:00.000378	0:00:00.000033
Worst Case	0:00:00.007275	0:00:00.000194

2. Design and Analysis of Algorithm/Assignments/02. Quick vs Merge Sort Time Analysis via `pip v3.10.7`

```
> python main.py
```

Test No	Quick Sort	Merge Sort
0	0:00:00.000013	0:00:00.000009
1	0:00:00.000337	0:00:00.000108
2	0:00:00.000505	0:00:00.000135
3	0:00:00.000083	0:00:00.000027
4	0:00:00.000846	0:00:00.000448
Best Case	0:00:00.051551	0:00:00.000432
Worst Case	0:00:00.048963	0:00:00.000457

2. Design and Analysis of Algorithm/Assignments/02. Quick vs Merge Sort Time Analysis via `pip v3.10.7`

```
> █
```

Conclusion:

Implemented Quick Sort and Merge Sort and Performed Comparison and Analysis on various input sizes