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: 15-10-2022

Date of Submission: 22-10-2022

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.

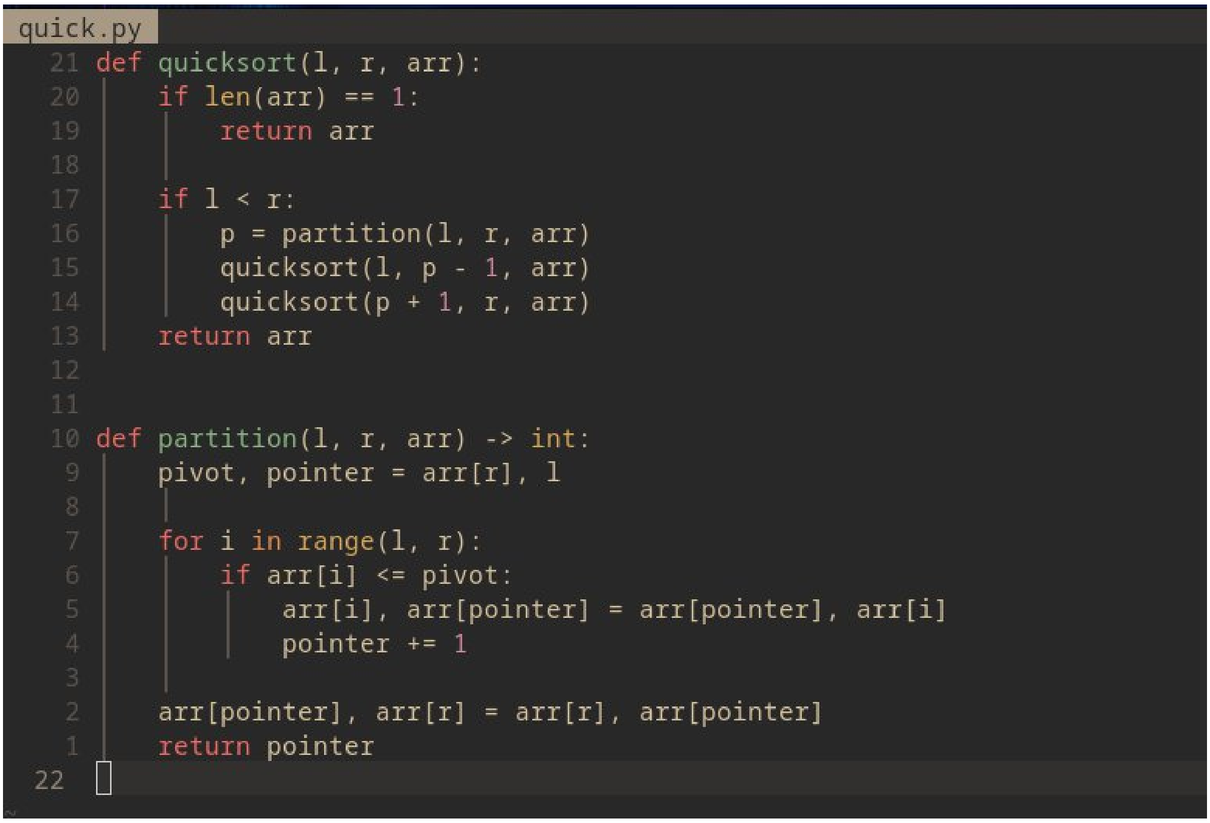
Quick Sort :-

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.

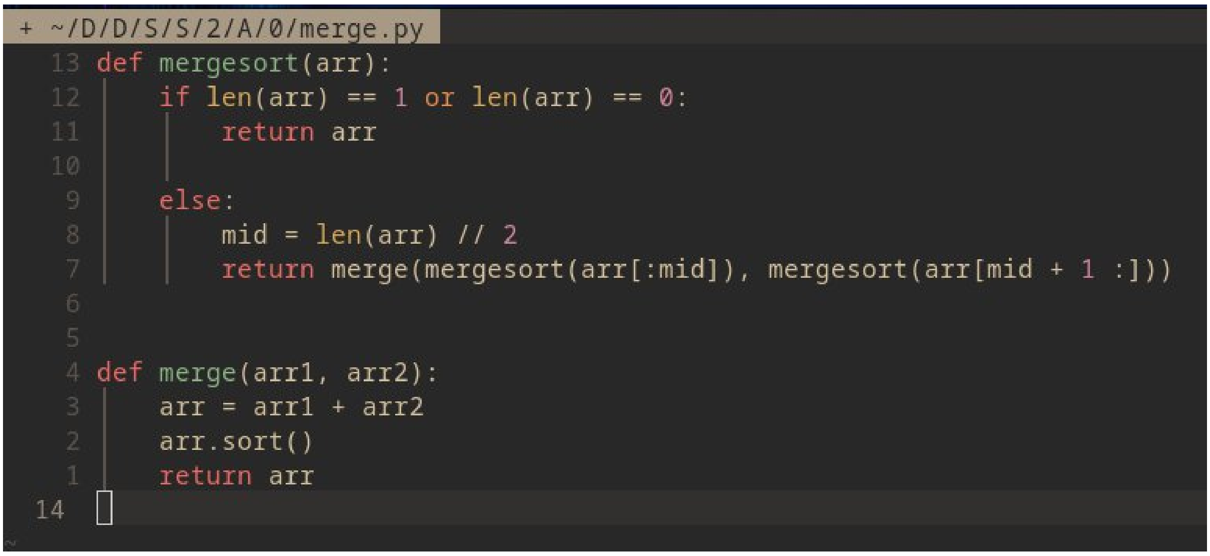
- Always pick the first element as a pivot.
- Always pick the last element as a pivot
- Pick a random element as a pivot.
- Pick median as the pivot.

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.

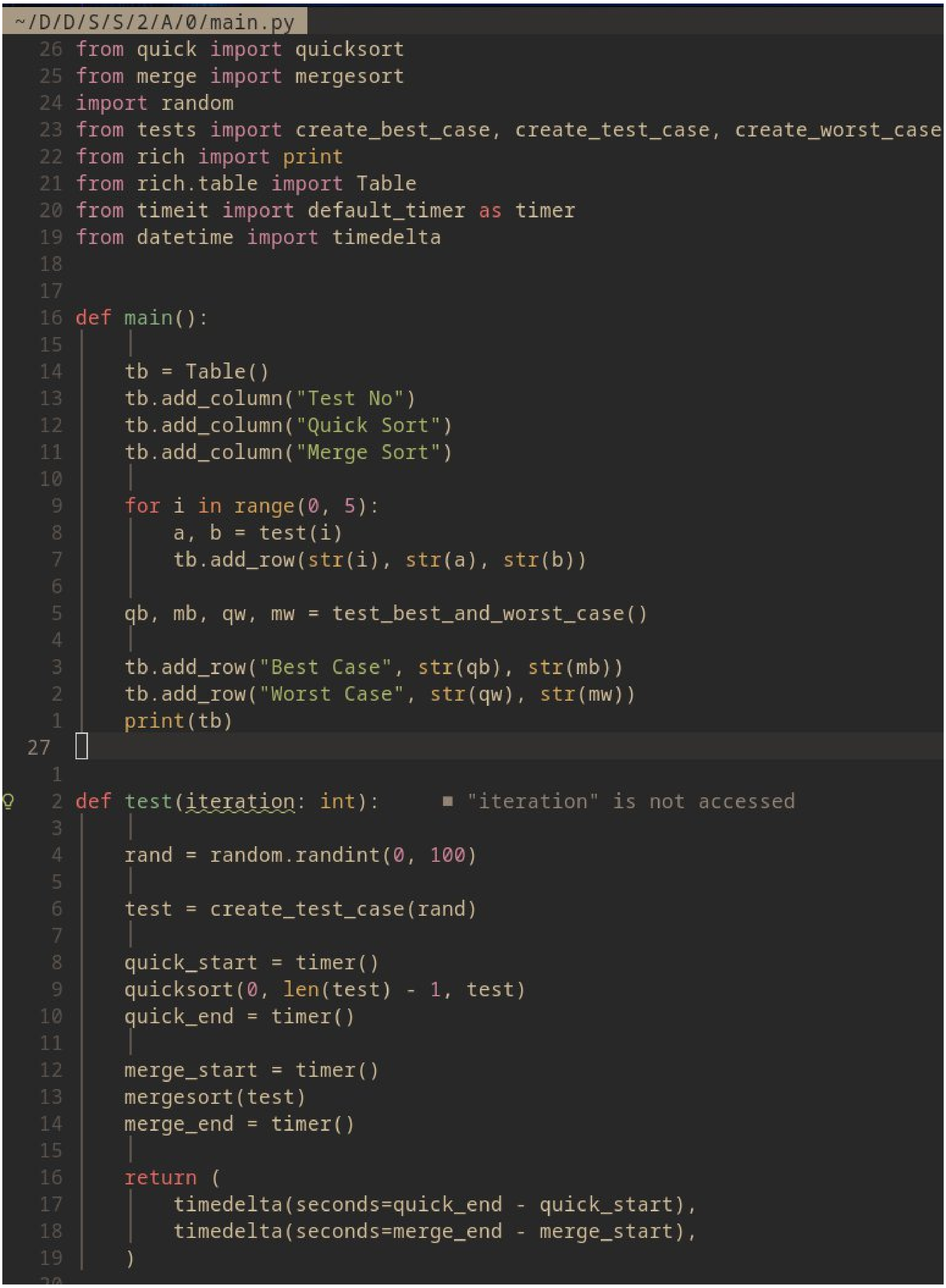
Code:



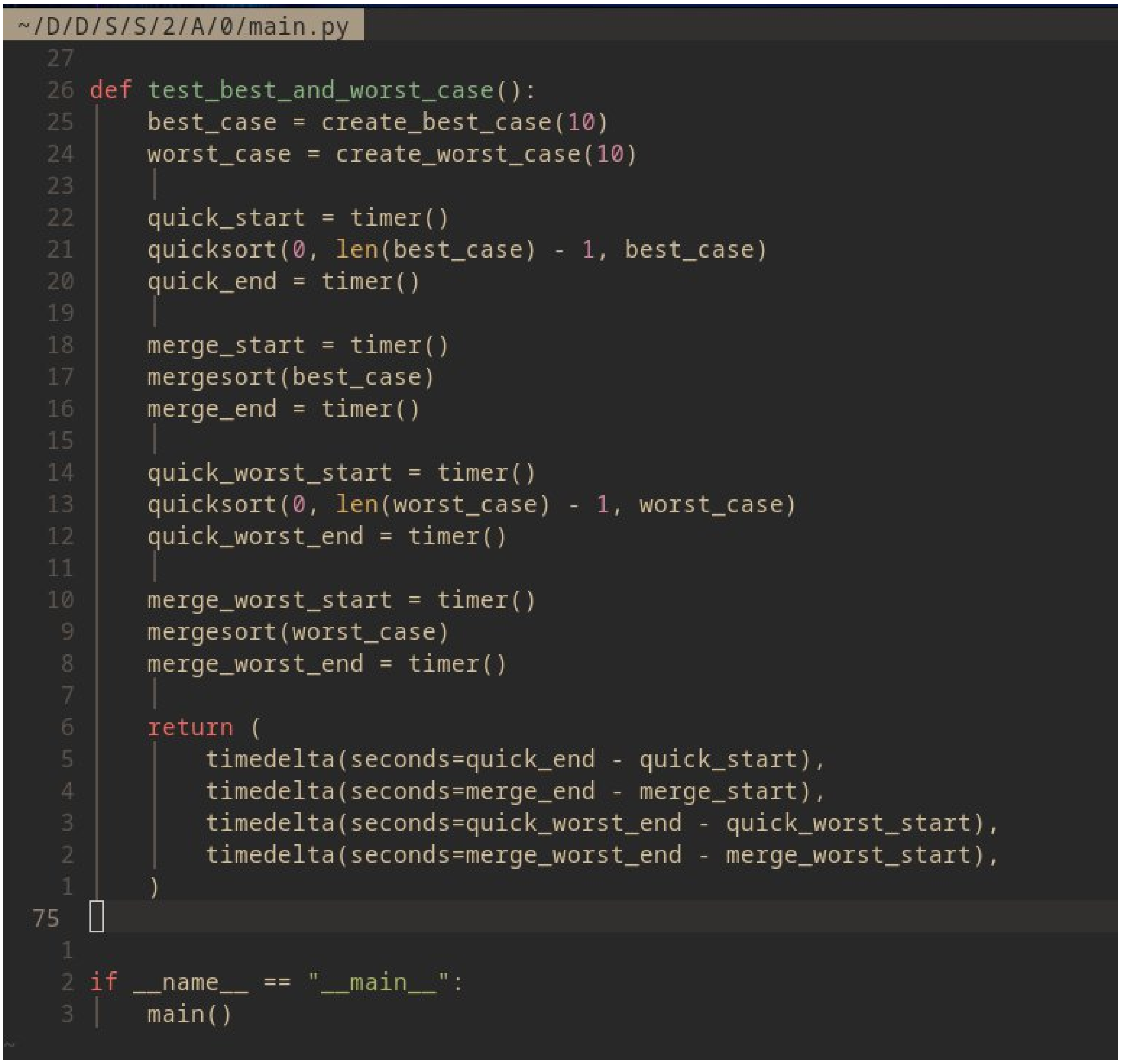
```python
def quicksort(l, r, arr):
    if len(arr) == 1:
        return arr

    if l < r:
        p = partition(l, r, arr)
        quicksort(l, p - 1, arr)
        quicksort(p + 1, r, arr)
    return arr


def partition(l, r, arr) -> int:
    pivot, pointer = arr[r], l

    for i in range(l, r):
        if arr[i] <= pivot:
            arr[i], arr[pointer] = arr[pointer], arr[i]
            pointer += 1

    arr[pointer], arr[r] = arr[r], arr[pointer]
    return pointer
```



```python
def mergesort(arr):
    if len(arr) == 1 or len(arr) == 0:
        return arr

    else:
        mid = len(arr) // 2
        return merge(mergesort(arr[:mid]), mergesort(arr[mid + 1 :]))


def merge(arr1, arr2):
    arr = arr1 + arr2
    arr.sort()
    return arr
```

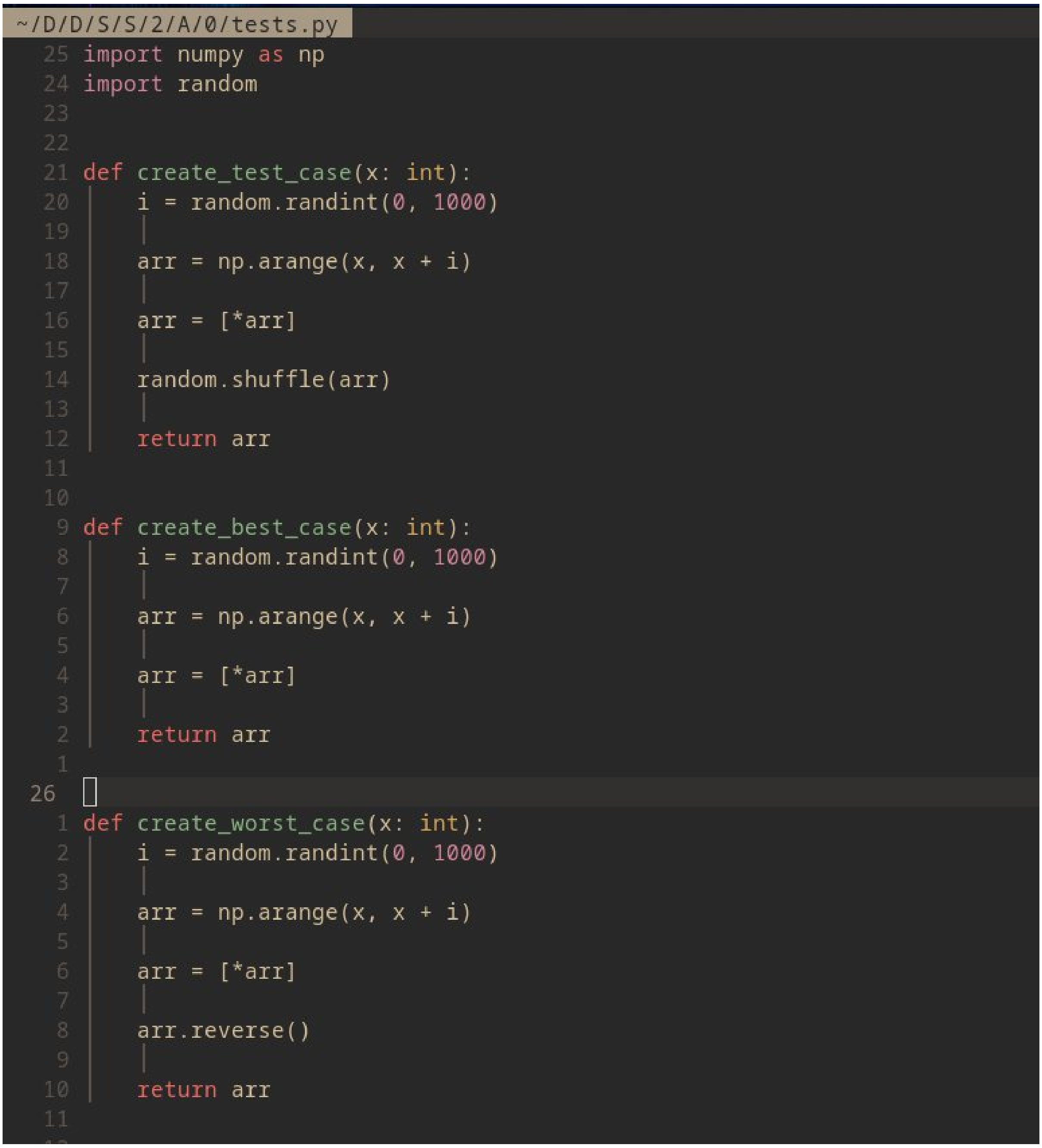


```
~/D/D/S/S/2/A/0/main.py
from quick import quicksort
from merge import mergesort
import random
from tests import create_best_case, create_test_case, create_worst_case
from rich import print
from rich.table import Table
from timeit import default_timer as timer
from datetime import timedelta


def main():

    tb = Table()
    tb.add_column("Test No")
    tb.add_column("Quick Sort")
    tb.add_column("Merge Sort")

    for i in range(0, 5):
        a, b = test(i)
        tb.add_row(str(i), str(a), str(b))

    qb, mb, qw, mw = test_best_and_worst_case()

    tb.add_row("Best Case", str(qb), str(mb))
    tb.add_row("Worst Case", str(qw), str(mw))
    print(tb)


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

    rand = random.randint(0, 100)

    test = create_test_case(rand)

    quick_start = timer()
    quicksort(0, len(test) - 1, test)
    quick_end = timer()

    merge_start = timer()
    mergesort(test)
    merge_end = timer()

    return (
        timedelta(seconds=quick_end - quick_start),
        timedelta(seconds=merge_end - merge_start),
    )
```



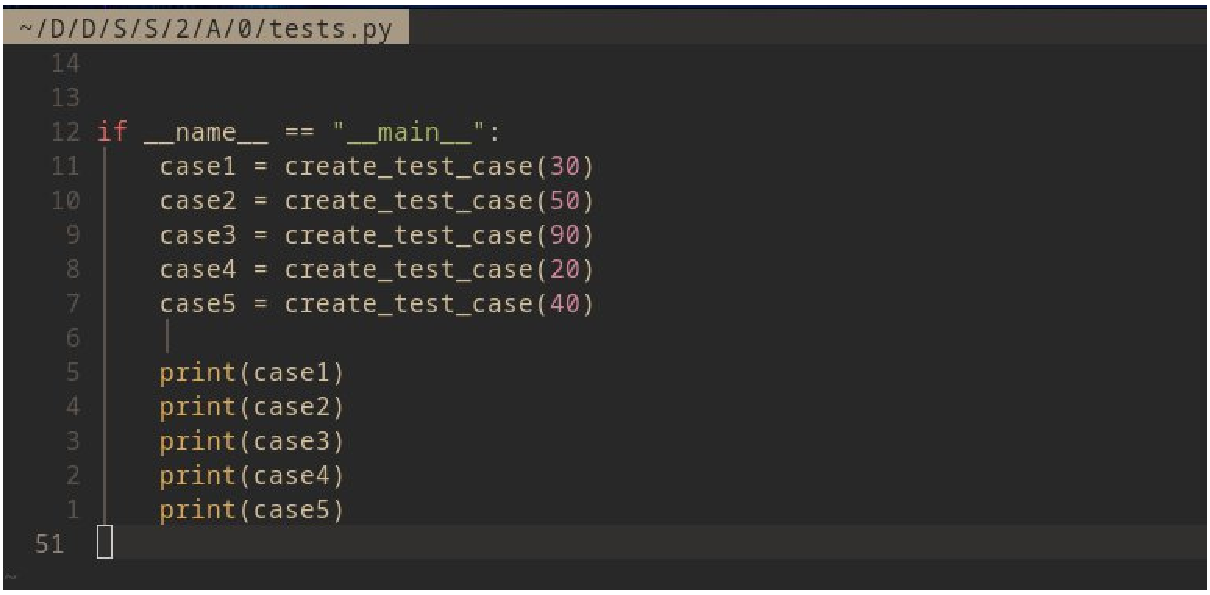
~/D/D/S/S/2/A/0/main.py

```python

def test_best_and_worst_case():
    best_case = create_best_case(10)
    worst_case = create_worst_case(10)

    quick_start = timer()
    quicksort(0, len(best_case) - 1, best_case)
    quick_end = timer()

    merge_start = timer()
    mergesort(best_case)
    merge_end = timer()

    quick_worst_start = timer()
    quicksort(0, len(worst_case) - 1, worst_case)
    quick_worst_end = timer()

    merge_worst_start = timer()
    mergesort(worst_case)
    merge_worst_end = timer()

    return (
        timedelta(seconds=quick_end - quick_start),
        timedelta(seconds=merge_end - merge_start),
        timedelta(seconds=quick_worst_end - quick_worst_start),
        timedelta(seconds=merge_worst_end - merge_worst_start),
    )


if __name__ == "__main__":
    main()
```



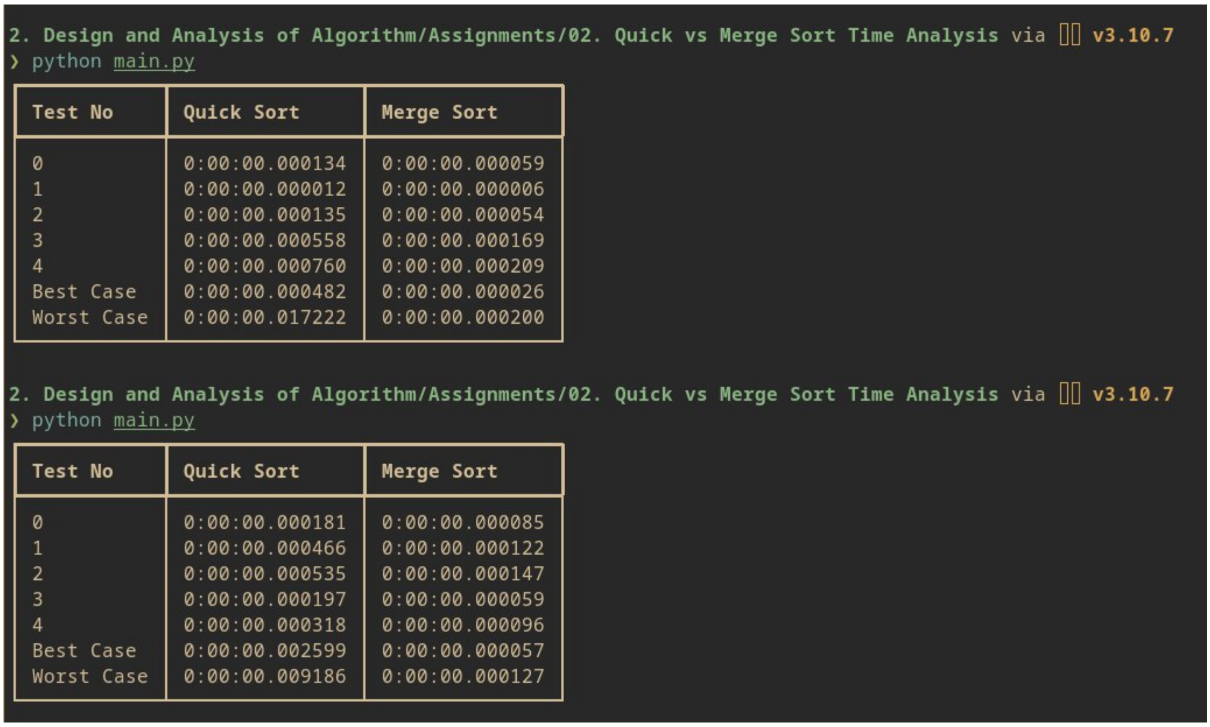
~/D/D/S/S/2/A/0/tests.py

```python
import numpy as np
import random


def create_test_case(x: int):
    i = random.randint(0, 1000)

    arr = np.arange(x, x + i)

    arr = [*arr]

    random.shuffle(arr)

    return arr


def create_best_case(x: int):
    i = random.randint(0, 1000)

    arr = np.arange(x, x + i)

    arr = [*arr]

    return arr


def create_worst_case(x: int):
    i = random.randint(0, 1000)

    arr = np.arange(x, x + i)

    arr = [*arr]

    arr.reverse()

    return arr

```

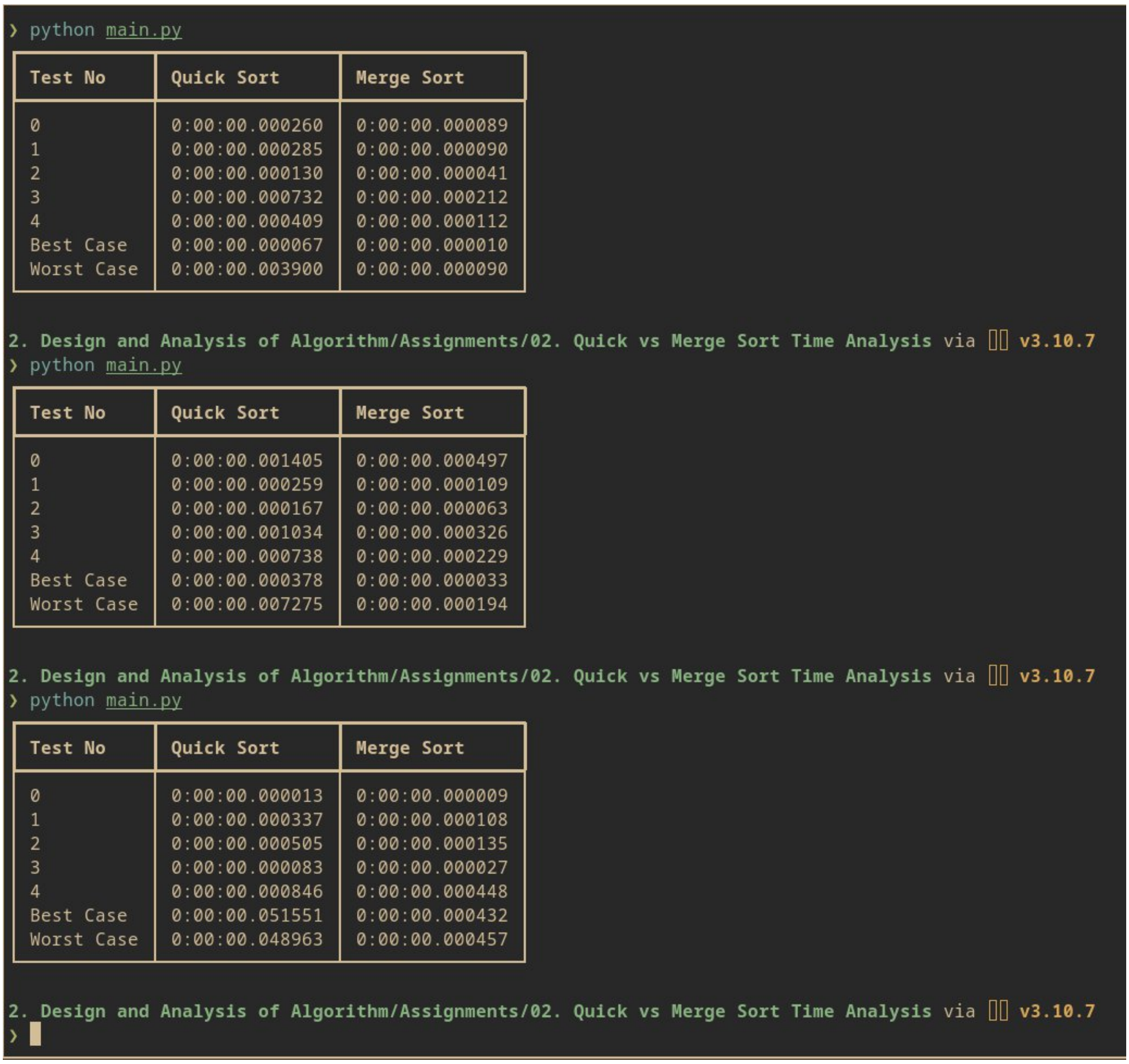


```
~/D/D/S/S/2/A/0/tests.py


if __name__ == "__main__":
    case1 = create_test_case(30)
    case2 = create_test_case(50)
    case3 = create_test_case(90)
    case4 = create_test_case(20)
    case5 = create_test_case(40)

    print(case1)
    print(case2)
    print(case3)
    print(case4)
    print(case5)

```

Output:



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

Conclusion:

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