Quicksort Algorithm: Implementation, Analysis, and Randomization

Umesh Dhakal

Computer Science: University of Cumberland

MSCS532

Dr. Satish Penmatsa

October 17,2025

Quicksort Algorithm: Implementation, Analysis, and Randomization

GitHub link- https://github.com/udhakal1s/MSCS532A5

Quicksort Algorithm

The quick sort algorithm works on the principle of the divide and conquer algorithm; it breaks a problem into smaller parts, solves them separately, and then combines the results. The steps that Quicksort does in sorting are

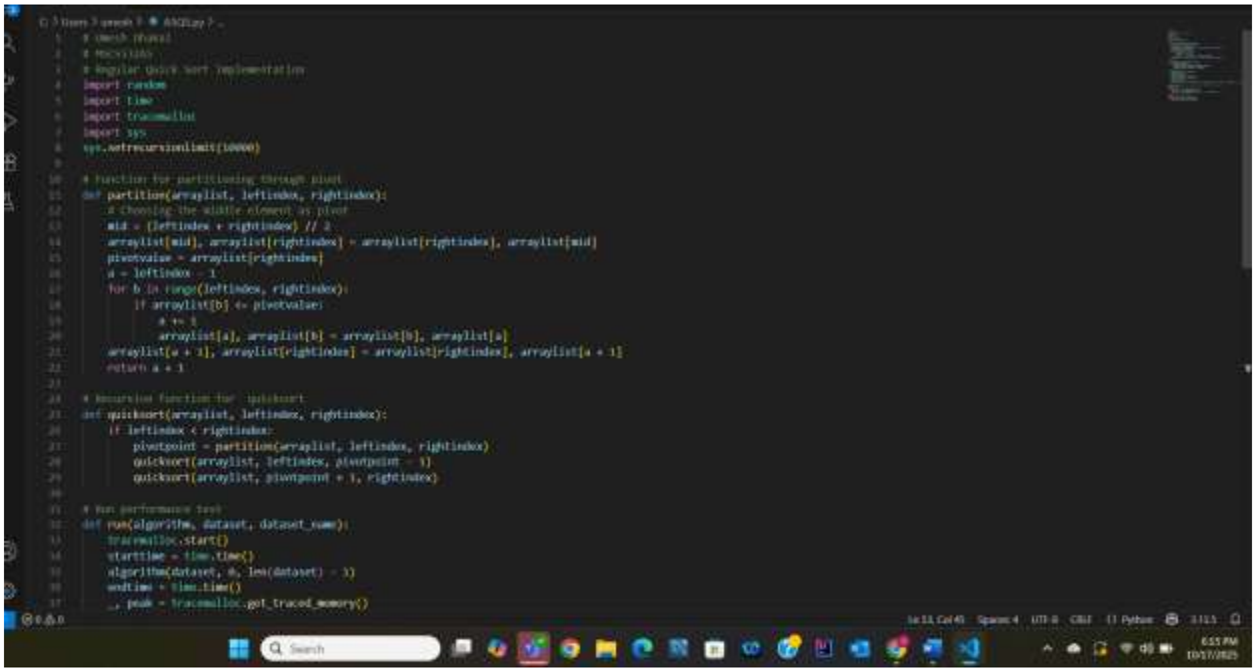
1. Choose a pivoted element from the dataset.
2. It Partition the dataset into two parts, where smaller elements than the pivot stay on the left side of pivot Partition and larger elements than the pivot stay on the right side of Partition
3. Then, it recursively applies the same technique to both the subarrays until the whole array is fully sorted.

Example demonstration

Let say we have 6 element is dataset containing the element [10,6,9,4,7,11]

1. Let's say 9 is the pivot
2. Then partition after choosing pivot is – [4,6,7] [9] [10,11]
3. The we apply the partition in subarray too – [4] [6] [7] and [10] [11]
4. Combine and get the result – [4,6,7,9,10,11]

Source code for Quicksort

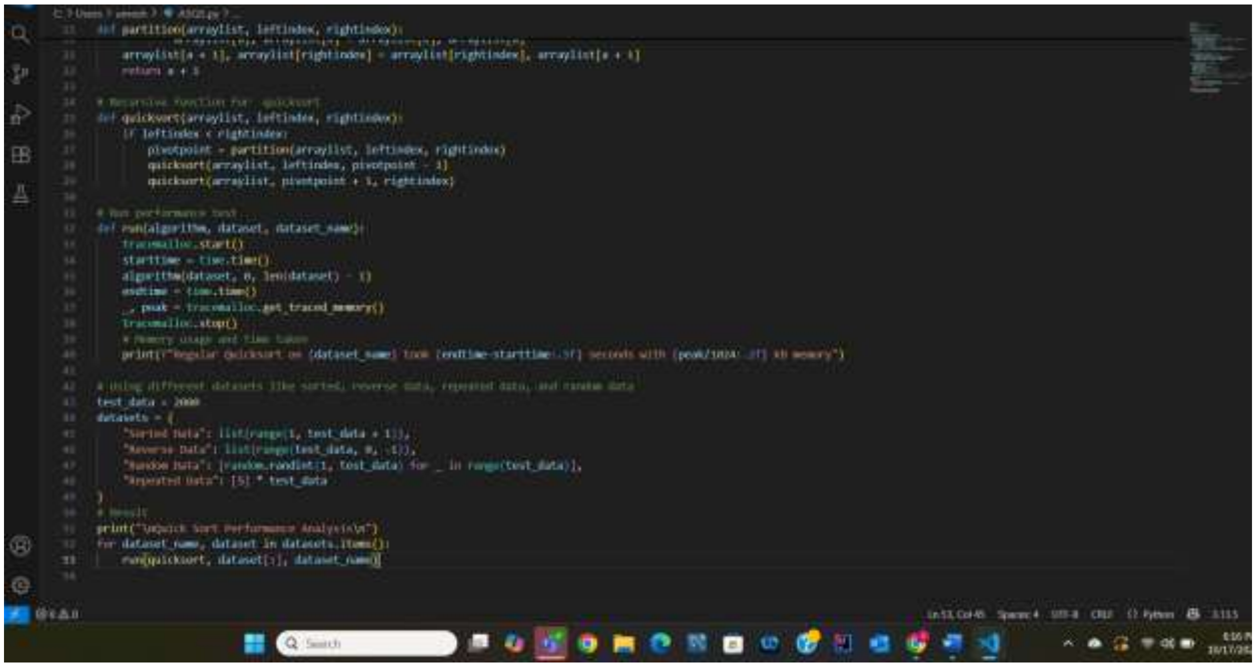


```python
# Umesh Dhakal
# MSCS532
# Regular Quick Sort Implementation
import random
import time
import tracemalloc
import sys
sys.setrecursionlimit(10000)

# Function for partitioning through pivot
def partition(arraylist, leftindex, rightindex):
    # Choosing the middle element as pivot
    mid = (leftindex + rightindex) // 2
    arraylist[mid], arraylist[rightindex] = arraylist[rightindex], arraylist[mid]
    pivotvalue = arraylist[rightindex]
    a = leftindex - 1
    for b in range(leftindex, rightindex):
        if arraylist[b] <= pivotvalue:
            a += 1
            arraylist[a], arraylist[b] = arraylist[b], arraylist[a]
    arraylist[a + 1], arraylist[rightindex] = arraylist[rightindex], arraylist[a + 1]
    return a + 1

# Recursive function for  quicksort
def quicksort(arraylist, leftindex, rightindex):
    if leftindex < rightindex:
        pivotpoint = partition(arraylist, leftindex, rightindex)
        quicksort(arraylist, leftindex, pivotpoint - 1)
        quicksort(arraylist, pivotpoint + 1, rightindex)

# Run performance test
def run(algorithm, dataset, dataset_name):
    tracemalloc.start()
    starttime = time.time()
    algorithm(dataset, 0, len(dataset) - 1)
    endtime = time.time()
    _, peak = tracemalloc.get_traced_memory()
```



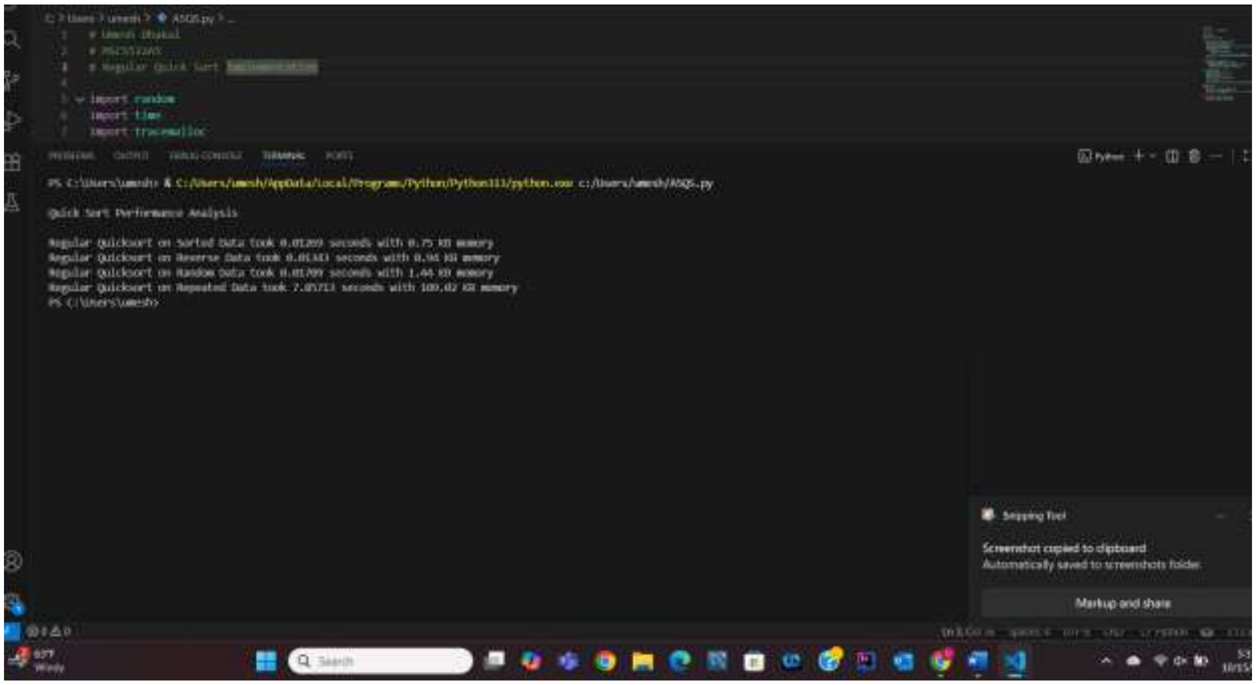
```python
def partition(arraylist, leftindex, rightindex):
    arraylist[a + 1], arraylist[rightindex] = arraylist[rightindex], arraylist[a + 1]
    return a + 1

# Recursive function for  quicksort
def quicksort(arraylist, leftindex, rightindex):
    if leftindex < rightindex:
        pivotpoint = partition(arraylist, leftindex, rightindex)
        quicksort(arraylist, leftindex, pivotpoint - 1)
        quicksort(arraylist, pivotpoint + 1, rightindex)

# Run performance test
def run(algorithm, dataset, dataset_name):
    tracemalloc.start()
    starttime = time.time()
    algorithm(dataset, 0, len(dataset) - 1)
    endtime = time.time()
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    # Memory usage and time taken
    print(f"Regular Quicksort on {dataset_name} took {endtime-starttime:.5f} seconds with {peak/1024:.2f} KB memory")

# Using different datasets like sorted, reverse data, repeated data, and random data
test_data = 2000
datasets = {
    "Sorted Data": list(range(1, test_data + 1)),
    "Reverse Data": list(range(test_data, 0, -1)),
    "Random Data": [random.randint(1, test_data) for _ in range(test_data)],
    "Repeated Data": [5] * test_data
}
# Result
print("\nQuick Sort Performance Analysis\n")
for dataset_name, dataset in datasets.items():
    run(quicksort, dataset[:], dataset_name)
```

Output



```
C: > Users > umesh > ASQS.py > ...
# Umesh Dhakal
# MSCS532AX
# Regular Quick Sort Implementation

import random
import time
import tracemalloc
```

```
PS C:\Users\umesh> & C:/Users/umesh/AppData/Local/Programs/Python/Python311/python.exe c:/Users/umesh/ASQS.py

Quick Sort Performance Analysis

Regular Quicksort on Sorted Data took 0.01269 seconds with 0.75 KB memory
Regular Quicksort on Reverse Data took 0.01343 seconds with 0.94 KB memory
Regular Quicksort on Random Data took 0.01709 seconds with 1.44 KB memory
Regular Quicksort on Repeated Data took 7.05713 seconds with 109.02 KB memory
PS C:\Users\umesh>
```

Snipping Tool
Screenshot copied to clipboard
Automatically saved to screenshots folder
Markup and share

Performance Analysis

The performance of Quicksort depends on how well the pivot divides the dataset.

Best case

In the best case, each partition divides the dataset into two equal parts. For example, if we have eight elements, After the partition, we get two subarrays of 4 elements each, after the next step, each subarray becomes 2, then 1. The number of steps in the recursion tree is now log n, and at each level, we process n elements. Therefore, total time it take would be n log n. So, time complexity in this case would be O (n log n).

Average case

In the average case, each partition divides the dataset into fairly two equal parts even if the pivot is not selected as a center element. In this case data might still be divided into ratio of 60 to 40 or 30 to 70, which is somewhat still balanced. As a result, most divisions are still balanced. On the case the number of steps in the recursion tree is log n, and at each level, we process n elements. Therefore, in average case total time it take would still be n log n. So, time complexity in this case would be O (n log n).

Worst case

In the worst case, the pivot is always chosen as the smallest or largest element in the dataset, as a result one of the partitions is always empty while another partition have all the elements other than pivot itself. In this case recursion depth becomes n, and each step processes n elements. Therefore, in worst case total time it take would be n^2. o time complexity in this case would be O(n^2).

Case	**Time Complexity**	**Reason**
Best	O (n log n)	Pivot divide data equally
Average	O (n log n)	Pivot divide data almost equally
Worst	O (n^2)	Pivot is always smallest or largest element

Space Complexity

Even though Even though Quicksort uses recursion, it is considered an in-place algorithm, and it does not need extra space for sorting.

Best case- In case of best case the recursion depth is about log n, so only log n function call are on the stack at a time. As a result, the space complexity in best case is O (log n)

Average case- In case of average case the recursion depth is still about log n, so only log n function call are on the stack at a time. As the result the space complexity in average case is O (log n)

Worst case - In case of worst case, the partitions are very unbalanced, and recursion can go as depth as n level. As a result, the space complexity in average case is O (n)

Additional Overheads

Although quicksort is known for its speed and efficiency there are still few things that affect performance

1. Function call overhead from recursion which add amount of extra processing for function management
2. Stack memory used for recursive calls and this usage increases when the recursion becomes deep especially in case of worst case.
3. Time taken to choose and swap the pivot which involves minor but repeated operations throughout the sorting process

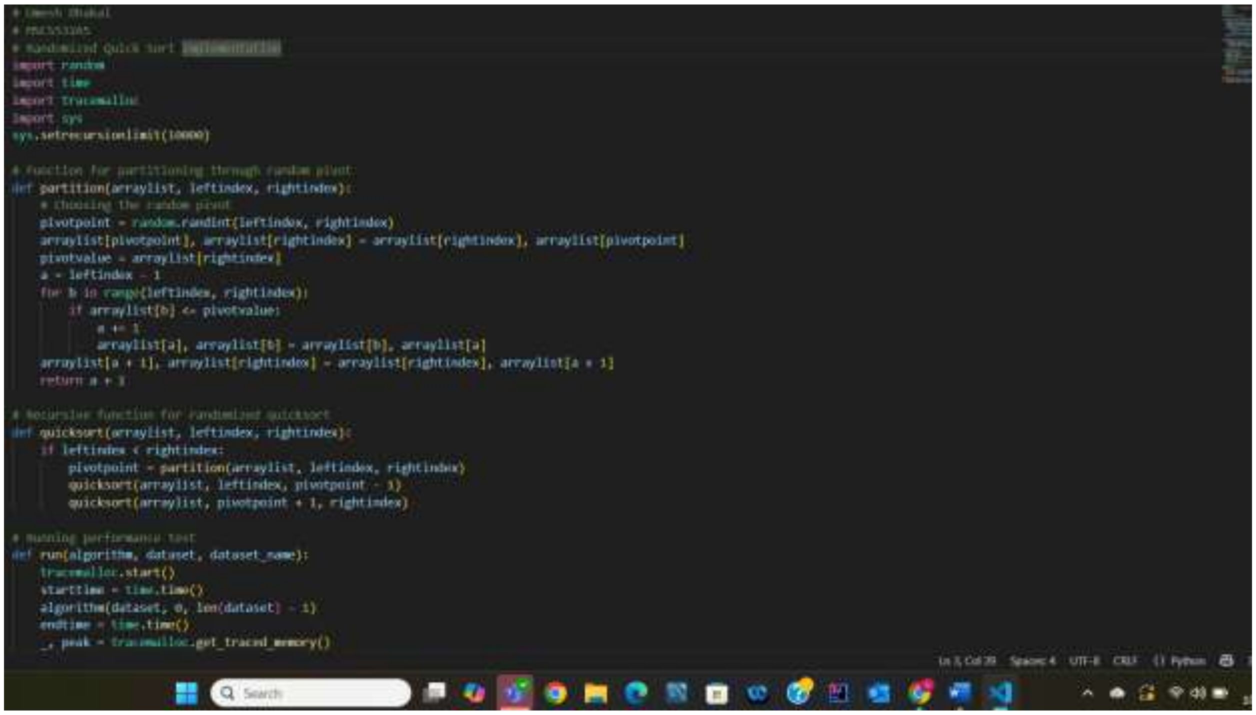
Randomized Quicksort Analysis

In Randomized Quicksort, the pivot is chosen randomly. When a pivot is chosen randomly, it avoids the worst pivot selections every time. It is almost impossible for a bad pivot to be selected every time with randomized quicksort. No matter what kind of dataset there is, the pivot chosen is not predictable, which makes it nearly impossible to get unbalanced partitions all the time. Therefore, in average and best-case number of steps in the recursion tree is log n, and at each level, we process n elements. So, the time complexity for best and average case would be same as regular quicksort, which is O (n log n). While regular quick sort might take O (n^2) time on already sorted data when the pivot is the smallest or largest but randomized avoids that and always runs in O (n log n), no matter the input datasets. The expected performance is always O (n log n) on average, and the risk of getting the worst case with randomized quicksort is very low. Whereas the risk of getting the worst case on deterministic quicksort is high, it always depends on the input order. The only time we would get time complexity O (n^2) with randomize Quicksort is when random pivot selected is always small or large number.

Space Complexity

On randomized quicksort, partitions are almost balanced, so on average, the recursion depth occurs O (log n), so the space complexity will be O (n). In the worst case, where the pivot is always selected as the smallest or largest, the recursion depth occurs O (n), so the space complexity will be O (n).

Source code for Randomize Quicksort

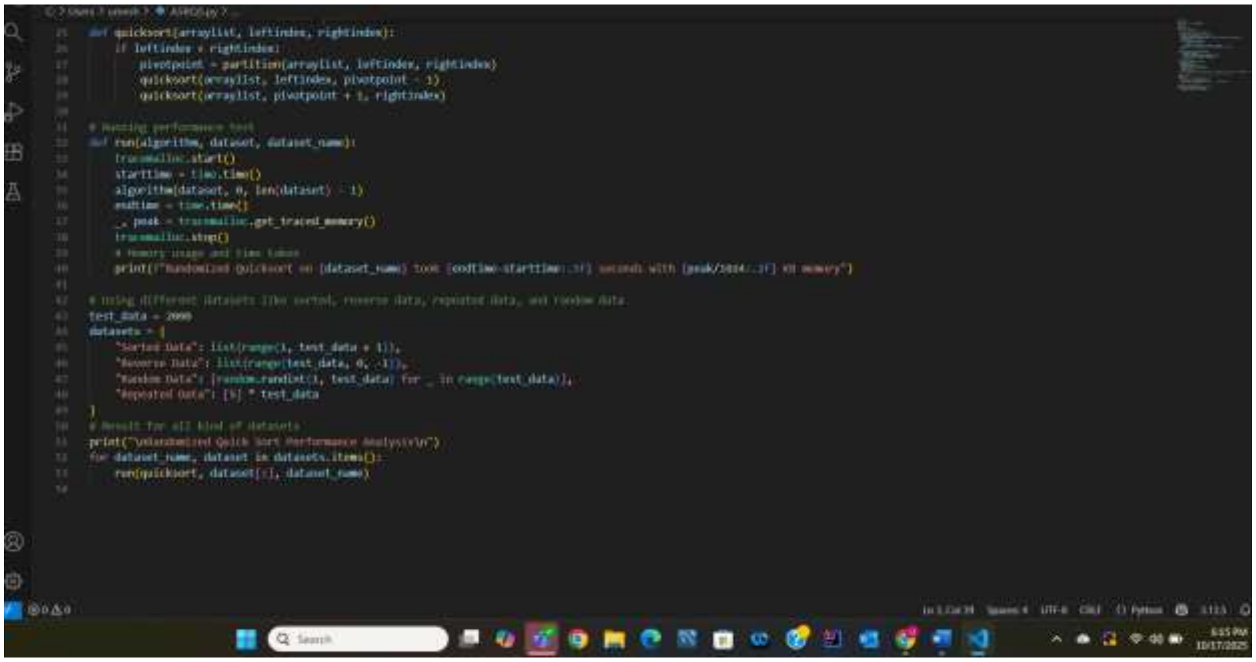


```
# [illegible]
# [illegible]
# Randomized Quick Sort implementation
import random
import time
import tracemalloc
import sys
sys.setrecursionlimit(10000)

# Function for partitioning through random pivot
def partition(arraylist, leftindex, rightindex):
    # Choosing the random pivot
    pivotpoint = random.randint(leftindex, rightindex)
    arraylist[pivotpoint], arraylist[rightindex] = arraylist[rightindex], arraylist[pivotpoint]
    pivotvalue = arraylist[rightindex]
    a = leftindex - 1
    for b in range(leftindex, rightindex):
        if arraylist[b] <= pivotvalue:
            a += 1
            arraylist[a], arraylist[b] = arraylist[b], arraylist[a]
    arraylist[a + 1], arraylist[rightindex] = arraylist[rightindex], arraylist[a + 1]
    return a + 1

# Recursive function for randomized quicksort
def quicksort(arraylist, leftindex, rightindex):
    if leftindex < rightindex:
        pivotpoint = partition(arraylist, leftindex, rightindex)
        quicksort(arraylist, leftindex, pivotpoint - 1)
        quicksort(arraylist, pivotpoint + 1, rightindex)

# Running performance test
def run(algorithm, dataset, dataset_name):
    tracemalloc.start()
    starttime = time.time()
    algorithm(dataset, 0, len(dataset) - 1)
    endtime = time.time()
    _, peak = tracemalloc.get_traced_memory()
```

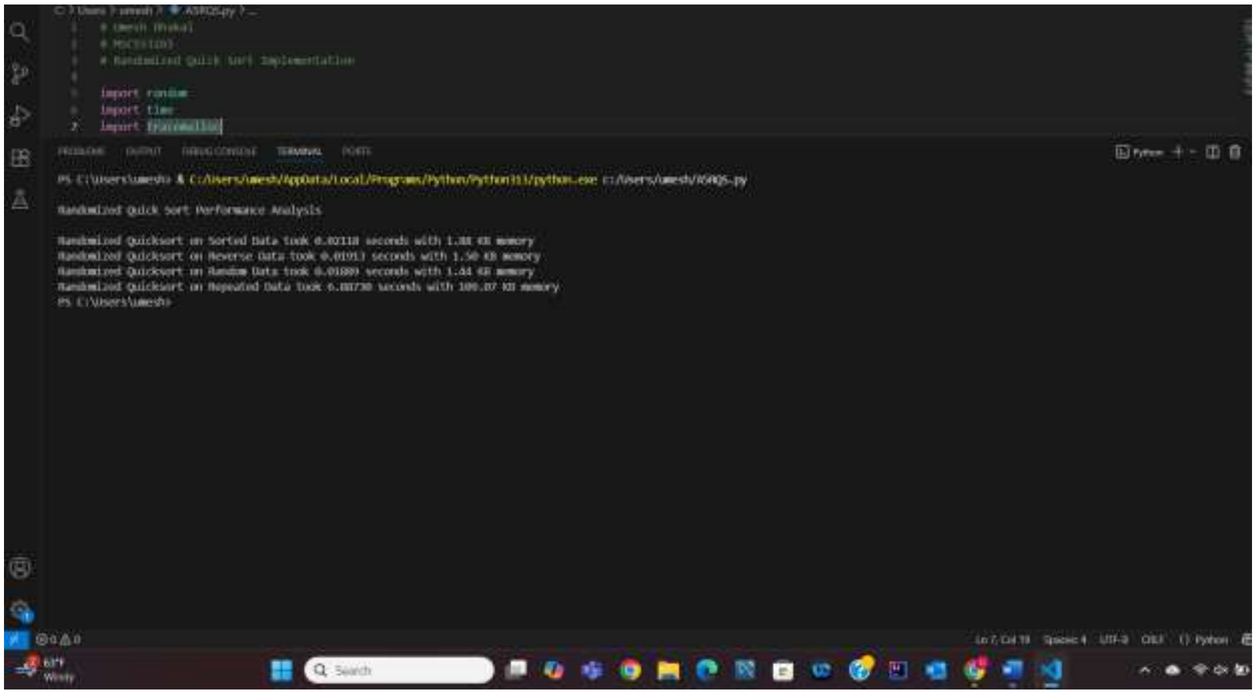


```python
def quicksort(arraylist, leftindex, rightindex):
    if leftindex < rightindex:
        pivotpoint = partition(arraylist, leftindex, rightindex)
        quicksort(arraylist, leftindex, pivotpoint - 1)
        quicksort(arraylist, pivotpoint + 1, rightindex)

# Running performance test
def run(algorithm, dataset, dataset_name):
    tracemalloc.start()
    starttime = time.time()
    algorithm(dataset, 0, len(dataset) - 1)
    endtime = time.time()
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    # Memory usage and time taken
    print(f"Randomized Quicksort on {dataset_name} took {endtime-starttime:.5f} seconds with {peak/1024:.2f} KB memory")

# Using different datasets like sorted, reverse data, repeated data, and random data
test_data = 2000
datasets = {
    "Sorted Data": list(range(1, test_data + 1)),
    "Reverse Data": list(range(test_data, 0, -1)),
    "Random Data": [random.randint(1, test_data) for _ in range(test_data)],
    "Repeated Data": [5] * test_data
}
# Result for all kind of datasets
print("\nRandomized Quick Sort Performance Analysis\n")
for dataset_name, dataset in datasets.items():
    run(quicksort, dataset[:], dataset_name)
```

Output



```python
# Umesh Dhakal
# [illegible]
# Randomized Quick Sort Implementation

import random
import time
import tracemalloc
```

```
PS C:\Users\umesh> & C:/Users/umesh/AppData/Local/Programs/Python/Python311/python.exe c:/Users/umesh/ASRQS.py

Randomized Quick Sort Performance Analysis

Randomized Quicksort on Sorted Data took 0.02118 seconds with 1.88 KB memory
Randomized Quicksort on Reverse Data took 0.01913 seconds with 1.50 KB memory
Randomized Quicksort on Random Data took 0.01889 seconds with 1.44 KB memory
Randomized Quicksort on Repeated Data took 6.88730 seconds with 109.07 KB memory
PS C:\Users\umesh>
```

Empirical Analysis

We tested both regular Quicksort and randomized Quicksort using Python on four different types of datasets such as sorted Data, reverse Data, random Data, and repeated Data. For both algorithms, we made the input size 2,000 elements. After that, we noted the total running time and memory usage for both kinds of algorithms. The memory usage was generally low for all datasets other than repeated datasets because in repeated data recursion depth increases.

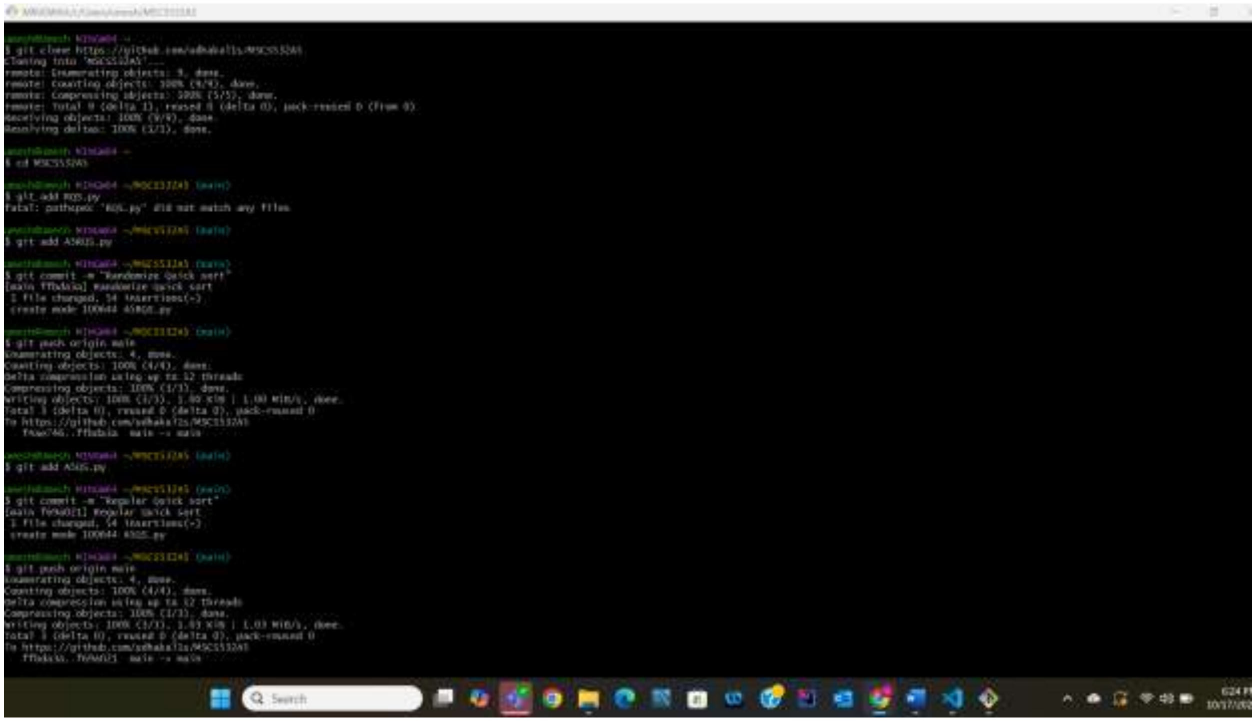
The Regular Quicksort was slightly faster than the Randomized Quicksort on sorted and reverse dataset. The time taken by regular quicksort was 0.01269 on sorted data and 0.01343 on reverse data whereas the time taken by randomize quicksort was 0.02118 on sorted data and 0.01913 on reverse data. In this case both algorithms performed very efficiently and avoid worst case.

On the random data both algorithm showed almost identical performance. The time taken by regular quicksort was 0.01709 and the time taken by randomize quicksort was 0.01889. This align with theoretical average case time complexity which is O (n log n) where both algorithms perform very efficiently as data has no particular order.

On the repeated data both algorithm took long time. The time taken by regular quicksort was 7.85713 and the time taken by randomize quicksort was 6.88730. This shows quicksort is less efficient when data are repeated. However, from the result we saw randomize quicksort perform slightly better than regular quicksort indicating the random pivot selection help.

From the result analysis, we can say both algorithms show efficient average performance on most datasets other than the repeated dataset, which causes the most slowdown due to poor partitioning. For real-world applications with unpredictable input data, randomized quicksort is better as it maintains consistent O (nlogn) performance and provide greater reliability and stability.

Git Commit



```
[illegible] MINGW64 ~
$ git clone https://github.com/sdhakal1s/MSCS532A5
Cloning into 'MSCS532A5'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 1), reused 8 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (1/1), done.

[illegible] MINGW64 ~
$ cd MSCS532A5

[illegible] MINGW64 ~/MSCS532A5 (main)
$ git add RQS.py
fatal: pathspec 'RQS.py' did not match any files

[illegible] MINGW64 ~/MSCS532A5 (main)
$ git add A5RQS.py

[illegible] MINGW64 ~/MSCS532A5 (main)
$ git commit -m "Randomize Quick sort"
[main ffbda1a] Randomize Quick sort
 1 file changed, 54 insertions(+)
 create mode 100644 A5RQS.py

[illegible] MINGW64 ~/MSCS532A5 (main)
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 1.00 KiB | 1.00 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/sdhakal1s/MSCS532A5
   f4ae746..ffbda1a  main -> main

[illegible] MINGW64 ~/MSCS532A5 (main)
$ git add A5QS.py

[illegible] MINGW64 ~/MSCS532A5 (main)
$ git commit -m "Regular Quick sort"
[main fe9a021] Regular Quick sort
 1 file changed, 54 insertions(+)
 create mode 100644 A5QS.py

[illegible] MINGW64 ~/MSCS532A5 (main)
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 1.03 KiB | 1.03 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/sdhakal1s/MSCS532A5
   ffbda1a..fe9a021  main -> main
```