# Assignment - 5

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

# Course : 2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

# Date : November 17, 2024

Github : https://github.com/VijayaKrishnaSameerajJonnavithula/Assignment-5

**Quick Sort :**

Code:

def quicksort(arr):

if len(arr) <= 1:

return arr

else:

pivot = arr[len(arr) // 2] # Choosing the middle element as the pivot

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

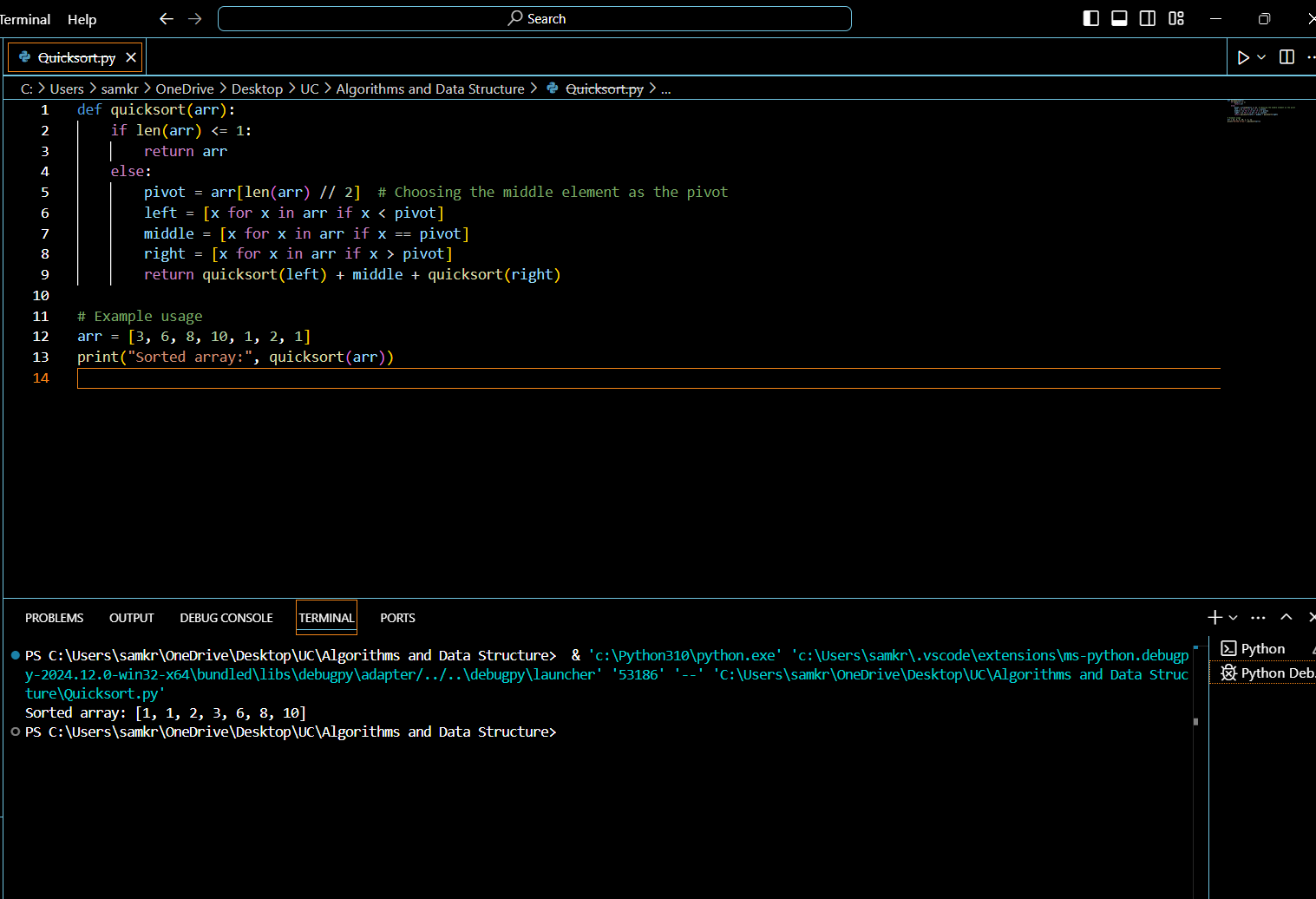
right = [x for x in arr if x > pivot]

return quicksort(left) + middle + quicksort(right)

# Example usage

arr = [3, 6, 8, 10, 1, 2, 1]

print("Sorted array:", quicksort(arr))



**Performance Analysis :**

Time Complexity in the Best-Case (wt(nlogn)): The array is divided into two almost equal halves at each step by the pivot, which is the best-case scenario. The recurrence relation is the outcome of this:  
  
𝑇 (𝑛) = 2 𝑇 (𝑛 2) + 𝑂 (𝑛)  
2T( 2n)+O(n) = T(n)  
Resolving this provides   
𝑇 (𝑛) = 𝑂 (𝑛 log ⁡ 𝑛)  
T(n)=O(nlogn).  
  
Average-case Complexity of Time (𝑂(𝑛 log ⁡ O(nlogn)): This is true when the array is divided by the pivot in a fashion that is sufficiently close to being completely balanced but not quite. In both average situations and numerous recursive calls, the algorithm's performance is roughly O(nlogn).

The worst situation Complexity of Time (O(n 2)): When sorting an array that has already been sorted or reverse-sorted, for example, this happens when the pivot is the largest or smallest element in the array, creating extremely uneven partitions. The recurrence relationship turns into:  
  
𝑇 (𝑛) = 𝑇 (𝑛 − 1) + 𝑂 (𝑛)  
T(n)=T(n−1)+O(n)  
When this is solved, it produces   
T(n) = O(n 2) = 𝑇 (𝑛) = 𝑂 (𝑛 2).

Space Complexity: Because of the recursion stack, Quicksort's space complexity is O(logn) = 𝑂 (log ⁡ 𝑛). The stack depth is usually 𝑂(log⁡𝑛) O(logn) for balanced partitioning, but in the worst scenario, it can approach 𝑂(𝑛) O(n). With the exception of the recursion stack, Quicksort does not require more space because it is already in place.

**Randomized Quicksort Implementation :**

import random

def randomized\_quicksort(arr):

if len(arr) <= 1:

return arr

else:

pivot\_index = random.randint(0, len(arr) - 1)

pivot = arr[pivot\_index]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

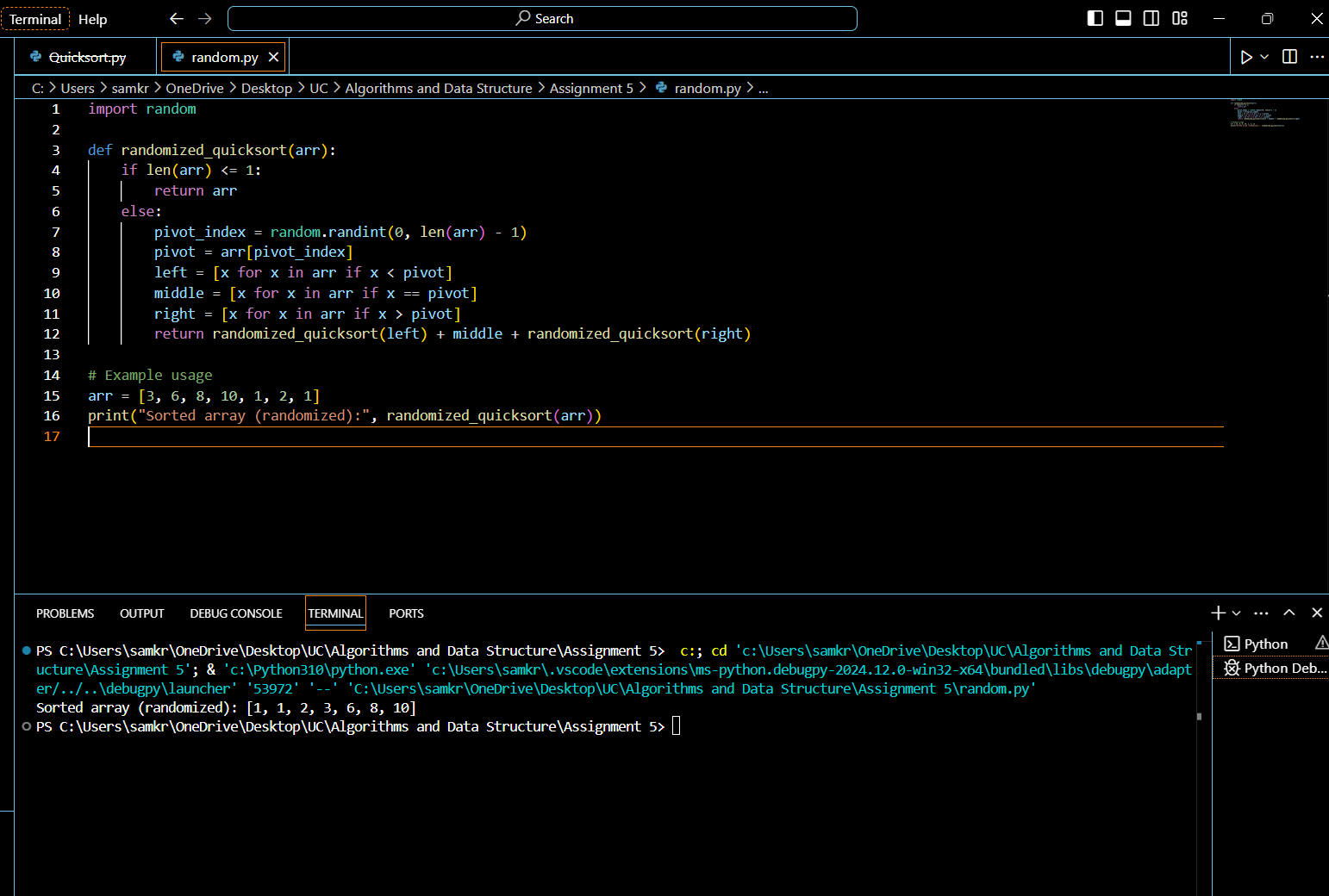
right = [x for x in arr if x > pivot]

return randomized\_quicksort(left) + middle + randomized\_quicksort(right)

# Example usage

arr = [3, 6, 8, 10, 1, 2, 1]

print("Sorted array (randomized):", randomized\_quicksort(arr))



Randomized Quicksort Analysis:

Impact on Performance: By decreasing the probability of selecting a continuously unbalanced pivot, randomized pivot selection dramatically lowers the chance of running into the worst-case 𝑂(𝑛2) O(n 2) time complexity. The average-case temporal complexity is still O(nlogn) = 𝑂 (𝑛 log ⁡ 𝑛).

**Empirical Analysis :**

To empirically compare the performance, you can run both versions on various input types and sizes.

import time

import random

from Quicksort import quicksort

from Randomsort import randomized\_quicksort

# Function to time the execution of a sorting algorithm

def time\_sorting\_algorithm(sort\_func, arr):

start\_time = time.time()

sort\_func(arr.copy())

return time.time() - start\_time

# Generate different input types

input\_sizes = [100, 1000, 5000, 10000]

input\_types = {

"random": lambda size: [random.randint(0, 10000) for \_ in range(size)],

"sorted": lambda size: list(range(size)),

"reverse\_sorted": lambda size: list(range(size, 0, -1))

}

# Compare deterministic and randomized Quicksort

for size in input\_sizes:

for input\_type, generator in input\_types.items():

arr = generator(size)

time\_deterministic = time\_sorting\_algorithm(quicksort, arr)

time\_randomized = time\_sorting\_algorithm(randomized\_quicksort, arr)

print(f"Size: {size}, Type: {input\_type}")

print(f" Deterministic Quicksort: {time\_deterministic:.5f} sec")

print(f" Randomized Quicksort: {time\_randomized:.5f} sec")

A screenshot of a computer program

Description automatically generated

qA screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

Interpretation of the Results: By avoiding worst-case pivot selections, the randomized version frequently performs better than the deterministic version on inputs such as sorted or reverse-sorted arrays.

Alignment with Theory: These findings are consistent with the theoretical analysis, which holds that randomization keeps the average performance at O(nlogn) by mitigating worst-case scenarios.

**REFERENCE:**

Python Documentation and Programming References:

Official Python Documentation: The Python Standard Library provides insights into built-in modules like random and how to use Python functions effectively.

Python's time module: Useful for timing code. Check the time module documentation.

Algorithm and Data Structures:

Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein (CLRS): This is a highly regarded textbook for understanding the theory behind algorithms, including Quicksort, with detailed time and space complexity analysis.

The Algorithm Design Manual by Steven Skiena: A practical guide with examples and performance analysis of algorithms.