Medians and Order Statistics & Elementary Data Structures

Umesh Dhakal

Computer Science: University of Cumberland

MSCS532

Dr. Satish Penmatsa

October 31,2025

Medians and Order Statistics & Elementary Data Structures

GitHub link- https://github.com/udhakal1s/MSCS532A6

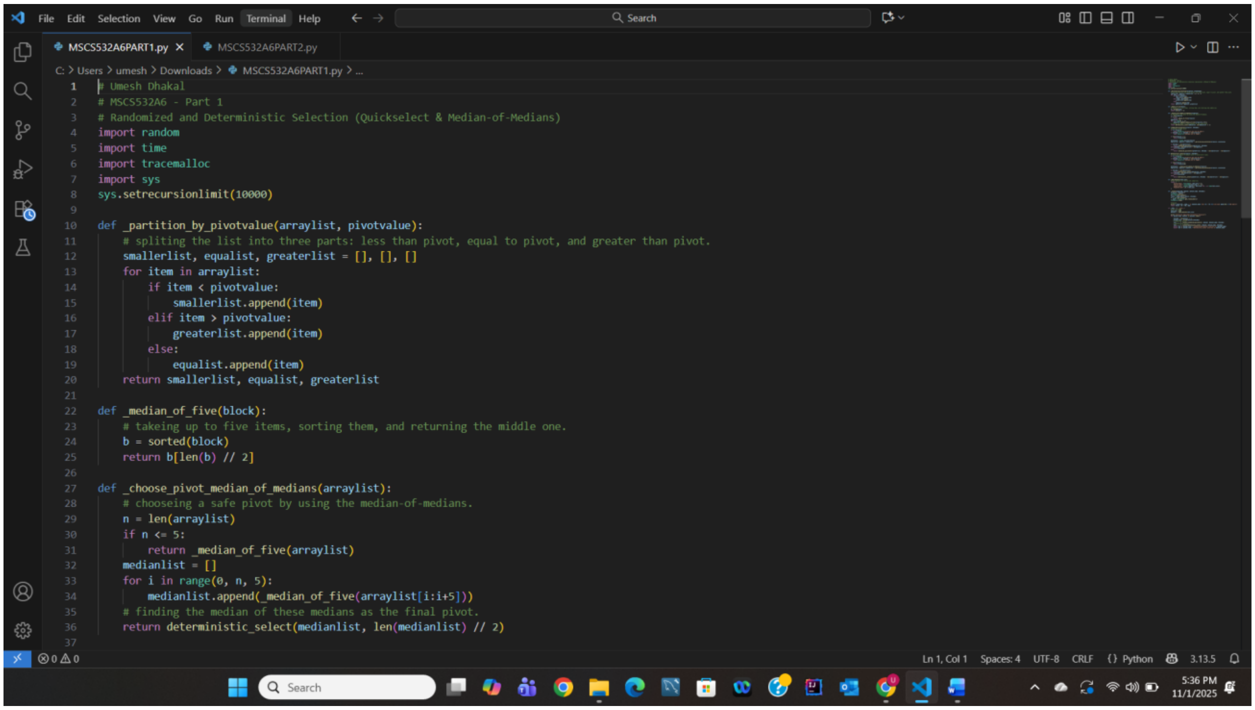
Implementation and Analysis of Selection Algorithms

Implementation

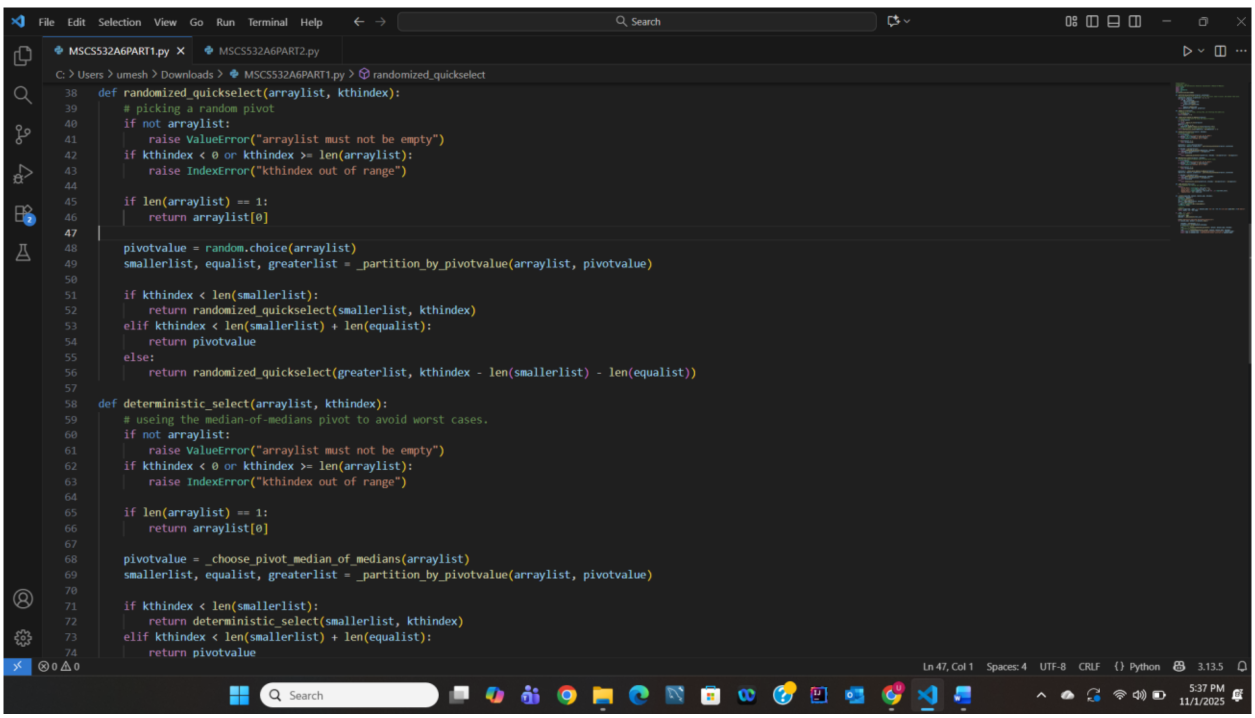
Here I implemented two different algorithms to find the smallest k-th element in a list. and both of the algorithm follow the general idea of breaking a complex problem into simpler problems. In the randomized version uses a random pivot to divide the datasets into smaller parts. As discussed in other assignments, doing so makes the problem easier to solve as large dataset is divided into smaller and we only needs to focus on one divided section at a time. The deterministic Median-of-Medians algorithm uses a step-by-step approach to choose the pivot. In this version instead of picking pivots randomly, it groups the datasets into smaller datasets, finds the median of each dataset, and then uses the median of those medians as the pivot. Even though it might be time consuming, it makes the algorithm more consistent and helps avoid the worst-case scenarios.

The screenshot below shows the coding implementation and output we got after implementation

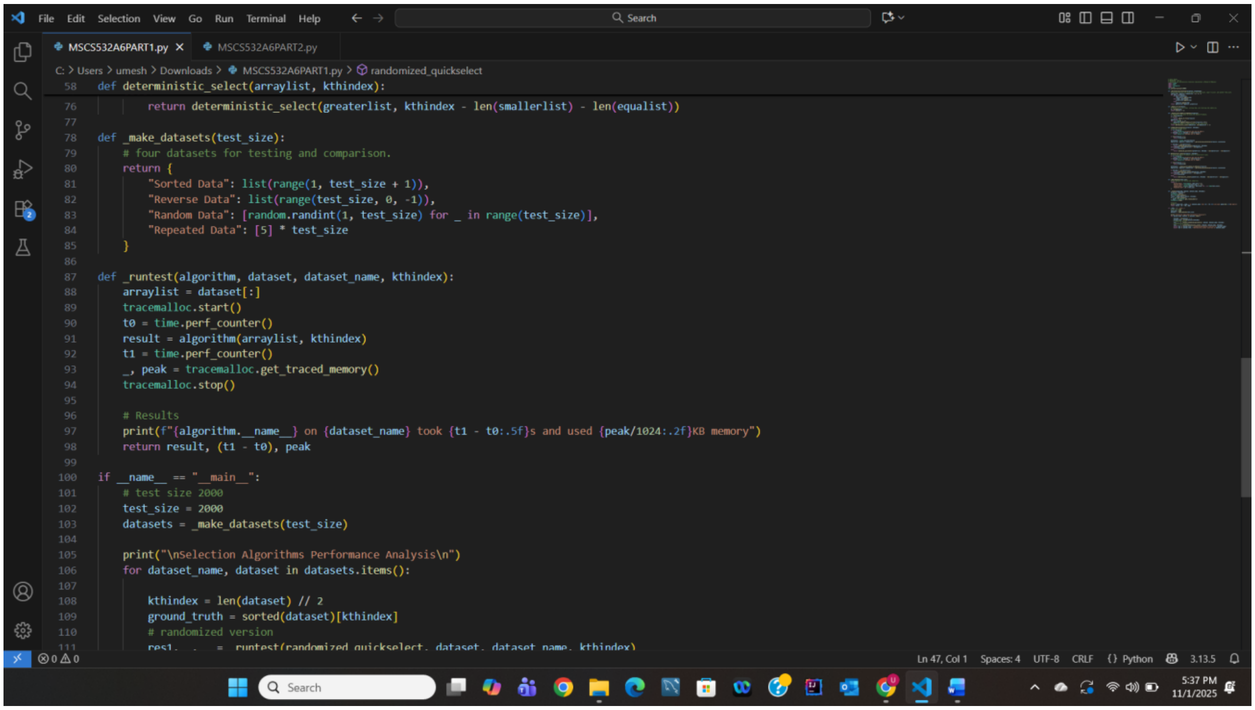
Source Code snippets



```python
# Umesh Dhakal
# MSCS532A6 - Part 1
# Randomized and Deterministic Selection (Quickselect & Median-of-Medians)
import random
import time
import tracemalloc
import sys
sys.setrecursionlimit(10000)

def _partition_by_pivotvalue(arraylist, pivotvalue):
    # spliting the list into three parts: less than pivot, equal to pivot, and greater than pivot.
    smallerlist, equalist, greaterlist = [], [], []
    for item in arraylist:
        if item < pivotvalue:
            smallerlist.append(item)
        elif item > pivotvalue:
            greaterlist.append(item)
        else:
            equalist.append(item)
    return smallerlist, equalist, greaterlist

def _median_of_five(block):
    # takeing up to five items, sorting them, and returning the middle one.
    b = sorted(block)
    return b[len(b) // 2]

def _choose_pivot_median_of_medians(arraylist):
    # chooseing a safe pivot by using the median-of-medians.
    n = len(arraylist)
    if n <= 5:
        return _median_of_five(arraylist)
    medianlist = []
    for i in range(0, n, 5):
        medianlist.append(_median_of_five(arraylist[i:i+5]))
    # finding the median of these medians as the final pivot.
    return deterministic_select(medianlist, len(medianlist) // 2)
```

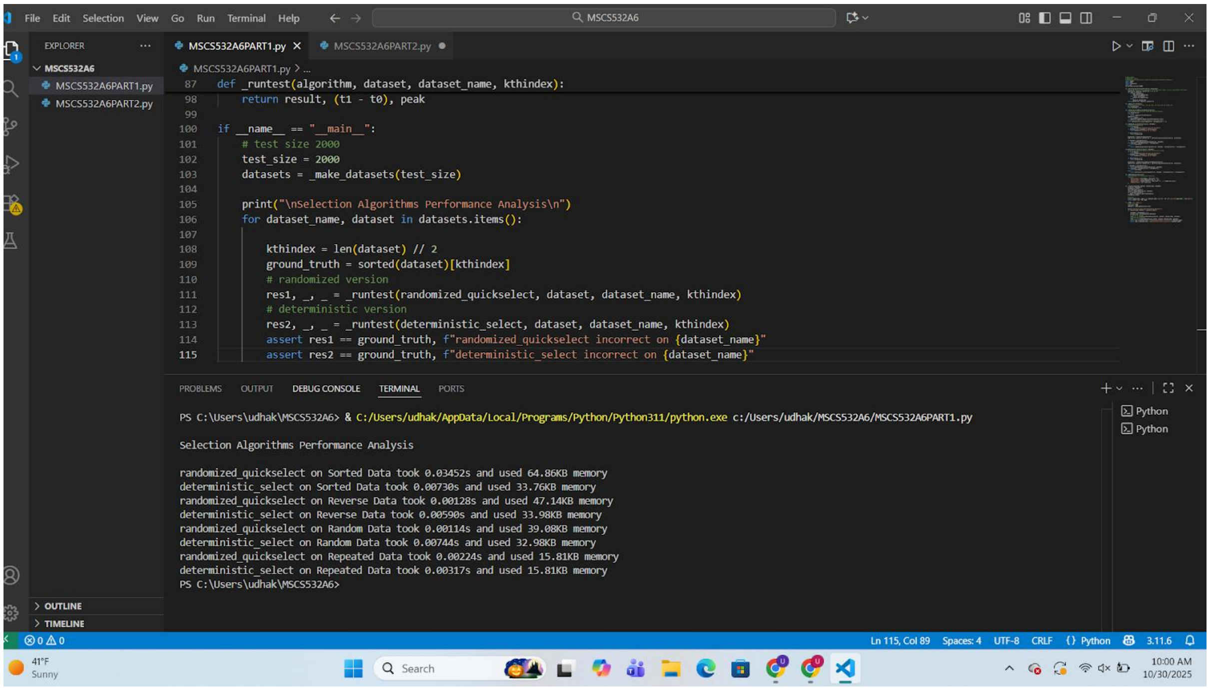


```python
def randomized_quickselect(arraylist, kthindex):
    # picking a random pivot
    if not arraylist:
        raise ValueError("arraylist must not be empty")
    if kthindex < 0 or kthindex >= len(arraylist):
        raise IndexError("kthindex out of range")

    if len(arraylist) == 1:
        return arraylist[0]

    pivotvalue = random.choice(arraylist)
    smallerlist, equalist, greaterlist = _partition_by_pivotvalue(arraylist, pivotvalue)

    if kthindex < len(smallerlist):
        return randomized_quickselect(smallerlist, kthindex)
    elif kthindex < len(smallerlist) + len(equalist):
        return pivotvalue
    else:
        return randomized_quickselect(greaterlist, kthindex - len(smallerlist) - len(equalist))

def deterministic_select(arraylist, kthindex):
    # useing the median-of-medians pivot to avoid worst cases.
    if not arraylist:
        raise ValueError("arraylist must not be empty")
    if kthindex < 0 or kthindex >= len(arraylist):
        raise IndexError("kthindex out of range")

    if len(arraylist) == 1:
        return arraylist[0]

    pivotvalue = _choose_pivot_median_of_medians(arraylist)
    smallerlist, equalist, greaterlist = _partition_by_pivotvalue(arraylist, pivotvalue)

    if kthindex < len(smallerlist):
        return deterministic_select(smallerlist, kthindex)
    elif kthindex < len(smallerlist) + len(equalist):
        return pivotvalue
```



```python
def deterministic_select(arraylist, kthindex):
        return deterministic_select(greaterlist, kthindex - len(smallerlist) - len(equalist))

def _make_datasets(test_size):
    # four datasets for testing and comparison.
    return {
        "Sorted Data": list(range(1, test_size + 1)),
        "Reverse Data": list(range(test_size, 0, -1)),
        "Random Data": [random.randint(1, test_size) for _ in range(test_size)],
        "Repeated Data": [5] * test_size
    }

def _runtest(algorithm, dataset, dataset_name, kthindex):
    arraylist = dataset[:]
    tracemalloc.start()
    t0 = time.perf_counter()
    result = algorithm(arraylist, kthindex)
    t1 = time.perf_counter()
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Results
    print(f"{algorithm.__name__} on {dataset_name} took {t1 - t0:.5f}s and used {peak/1024:.2f}KB memory")
    return result, (t1 - t0), peak

if __name__ == "__main__":
    # test size 2000
    test_size = 2000
    datasets = _make_datasets(test_size)

    print("\nSelection Algorithms Performance Analysis\n")
    for dataset_name, dataset in datasets.items():

        kthindex = len(dataset) // 2
        ground_truth = sorted(dataset)[kthindex]
        # randomized version
        res1, _, _ = _runtest(randomized_quickselect, dataset, dataset_name, kthindex)
```

Output



```python
def _runtest(algorithm, dataset, dataset_name, kthindex):
    return result, (t1 - t0), peak

if __name__ == "__main__":
    # test size 2000
    test_size = 2000
    datasets = _make_datasets(test_size)

    print("\nSelection Algorithms Performance Analysis\n")
    for dataset_name, dataset in datasets.items():

        kthindex = len(dataset) // 2
        ground_truth = sorted(dataset)[kthindex]
        # randomized version
        res1, _, _ = _runtest(randomized_quickselect, dataset, dataset_name, kthindex)
        # deterministic version
        res2, _, _ = _runtest(deterministic_select, dataset, dataset_name, kthindex)
        assert res1 == ground_truth, f"randomized_quickselect incorrect on {dataset_name}"
        assert res2 == ground_truth, f"deterministic_select incorrect on {dataset_name}"
```

```
PS C:\Users\udhak\MSCS532A6> & C:/Users/udhak/AppData/Local/Programs/Python/Python311/python.exe c:/Users/udhak/MSCS532A6/MSCS532A6PART1.py

Selection Algorithms Performance Analysis

randomized_quickselect on Sorted Data took 0.03452s and used 64.86KB memory
deterministic_select on Sorted Data took 0.00730s and used 33.76KB memory
randomized_quickselect on Reverse Data took 0.00128s and used 47.14KB memory
deterministic_select on Reverse Data took 0.00590s and used 33.98KB memory
randomized_quickselect on Random Data took 0.00114s and used 39.08KB memory
deterministic_select on Random Data took 0.00744s and used 32.98KB memory
randomized_quickselect on Repeated Data took 0.00224s and used 15.81KB memory
deterministic_select on Repeated Data took 0.00317s and used 15.81KB memory
PS C:\Users\udhak\MSCS532A6>
```

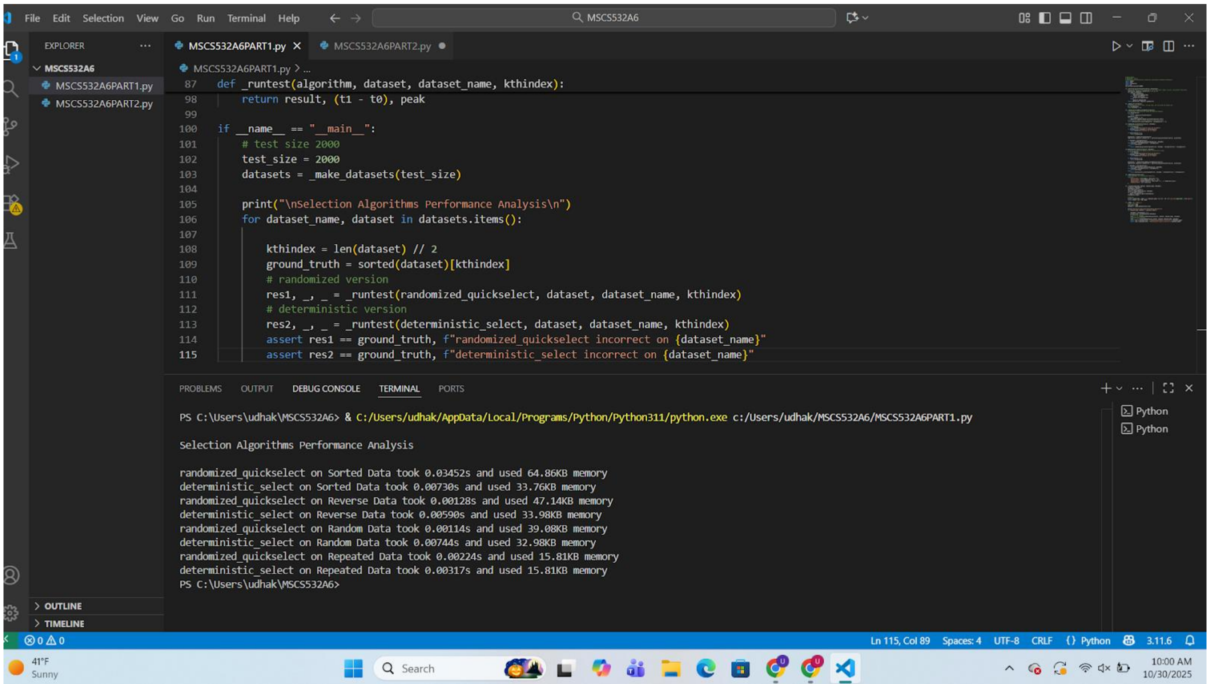
Performance Analysis

The performance of all kinds of quicksort depends on how the pivot is selected during sorting processes. In the case of selection algorithm, the performance depends on how they choose their pivot and how they break the large datasets into smaller parts. The randomized selection algorithm works fast in almost all situations as it picks a pivot randomly. When the pivot is good, it results into balanced partitions and as a result the randomized algorithm normally runs in O(n) time. However, if the pivot selection is bad this makes the data split unevenly. And If pivot selection happen to be bad all the time than the algorithm becomes slower as there is too many uneven partitions and that make the run time $O(n^2)$ making it worst case. It is very unlikely that this scenario happen in real life.

The deterministic Median-of-Medians algorithm uses a step-by-step approach to choose the pivot. In this version instead of picking pivots randomly, it groups the datasets into smaller datasets, finds the median of each dataset, and then uses the median of those medians as the pivot. Even though it might be time consuming, it makes the algorithm more consistent and helps avoid the worst-case scenarios and it always runs in O(n) worst-case time. This is why the deterministic algorithm is more predictable and reliable.

For space complexity, as both algorithms in Python use extra memory because they split the dataset into smaller parts during partitioning. Both the versions take O(n) space in the partitioning process. The deterministic algorithm also has extra overhead as it should build groups of five and store medians in it, as a result it uses more memory and does more work in each step. As it involve extra steps it could be slower in practice, but it always guarantee that it will not fall into a worst-case scenario. Overall, the randomized version is faster on average, while the deterministic version is more stable and predictable.

Empirical Analysis



Here the analysis we tested both the randomized selection algorithm and the deterministic selection algorithm on four different types of datasets such as sorted data, reverse data, random data, and repeated data. Each dataset had 2,000 elements. For each dataset we recorded the total running time and the memory used. This helped us analyze how each algorithm behaves under different kinds of input.

On the sorted dataset, the randomized selection algorithm version took 0.03452 seconds and used 64.86 kb of memory. Whereas deterministic algorithm only took 0.00730 seconds and use 33.76 kb which performed faster compared to randomized version. The results shows that the deterministic algorithm works better on sorted data because it always chooses a safe pivot, while the randomized algorithm depends on better the pivot choice. On the reverse dataset, the randomized algorithm version took 0.00128 seconds with 47.14 kb of memory use. The deterministic method took 0.00509 seconds and used 33.90 kb memory.

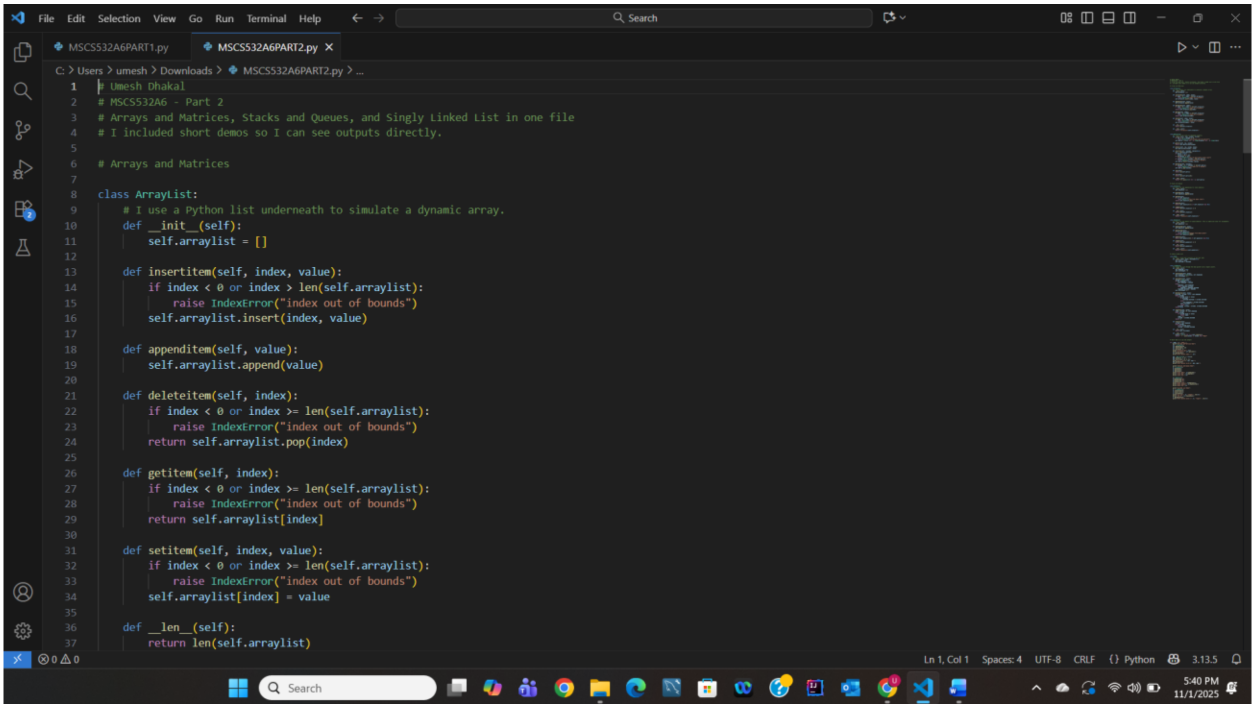
On the random dataset, both deterministic and randomized algorithms performed very efficiently. The randomized algorithm only took 0.00114 seconds and used 39.90 kb memory, where the deterministic algorithm took 0.00744 seconds and used 32.98 kb of memory. This fits what we expect in theory because random data usually creates balanced partitions for both algorithms, helping them run in linear time.

On the repeated dataset, both algorithms used less memory. The randomized algorithm took 0.00224 seconds, and the deterministic one took 0.00317 seconds, and both used 15.81 kb of memory. Repeated data usually makes the partition step easier because many elements are equal, so the algorithms do not have to search as deeply.

Elementary Data Structures Implementation

For data structure implementation I created several basic data structures such as Array, Matrices, Queue and Linked list. First, I started with arrays and matrices, where I inserted, deleted, and accessed values. After that I implemented a stack and a queue using Python lists. At last, I implemented a singly linked list by creating nodes that point to the next node as shown in the source code in the GitHub file.

Source Code snippets


MSCS532A6PART1.py
MSCS532A6PART2.py
C: > Users > umesh > Downloads > MSCS532A6PART2.py > ...
Umesh Dhakal
MSCS532A6 - Part 2
Arrays and Matrices, Stacks and Queues, and Singly Linked List in one file
I included short demos so I can see outputs directly.

Arrays and Matrices

class ArrayList:
    # I use a Python list underneath to simulate a dynamic array.
    def __init__(self):
        self.arraylist = []

    def insertitem(self, index, value):
        if index < 0 or index > len(self.arraylist):
            raise IndexError("index out of bounds")
        self.arraylist.insert(index, value)

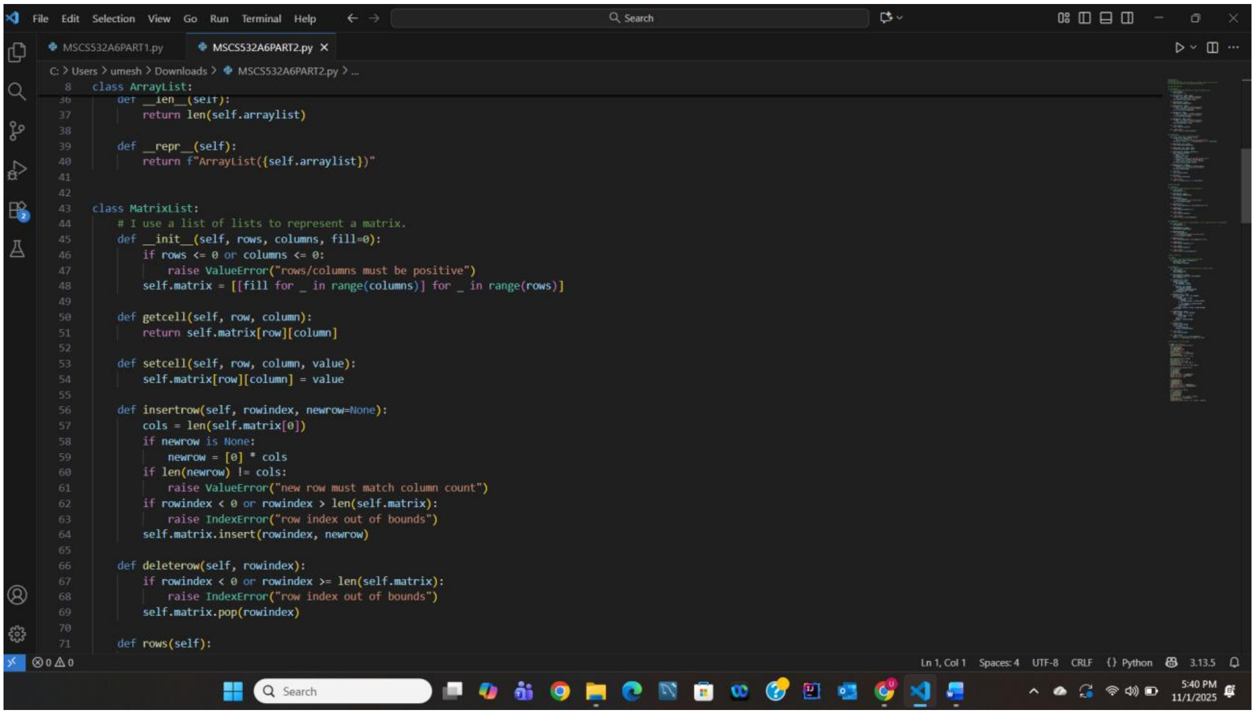
    def appenditem(self, value):
        self.arraylist.append(value)

    def deleteitem(self, index):
        if index < 0 or index >= len(self.arraylist):
            raise IndexError("index out of bounds")
        return self.arraylist.pop(index)

    def getitem(self, index):
        if index < 0 or index >= len(self.arraylist):
            raise IndexError("index out of bounds")
        return self.arraylist[index]

    def setitem(self, index, value):
        if index < 0 or index >= len(self.arraylist):
            raise IndexError("index out of bounds")
        self.arraylist[index] = value

    def __len__(self):
        return len(self.arraylist)
Ln 1, Col 1   Spaces: 4   UTF-8   CRLF   Python   3.13.5
5:40 PM
11/1/2025


MSCS532A6PART1.py
MSCS532A6PART2.py
C: > Users > umesh > Downloads > MSCS532A6PART2.py > ...
class ArrayList:
        return len(self.arraylist)

    def __repr__(self):
        return f"ArrayList({self.arraylist})"

class MatrixList:
    # I use a list of lists to represent a matrix.
    def __init__(self, rows, columns, fill=0):
        if rows <= 0 or columns <= 0:
            raise ValueError("rows/columns must be positive")
        self.matrix = [[fill for _ in range(columns)] for _ in range(rows)]

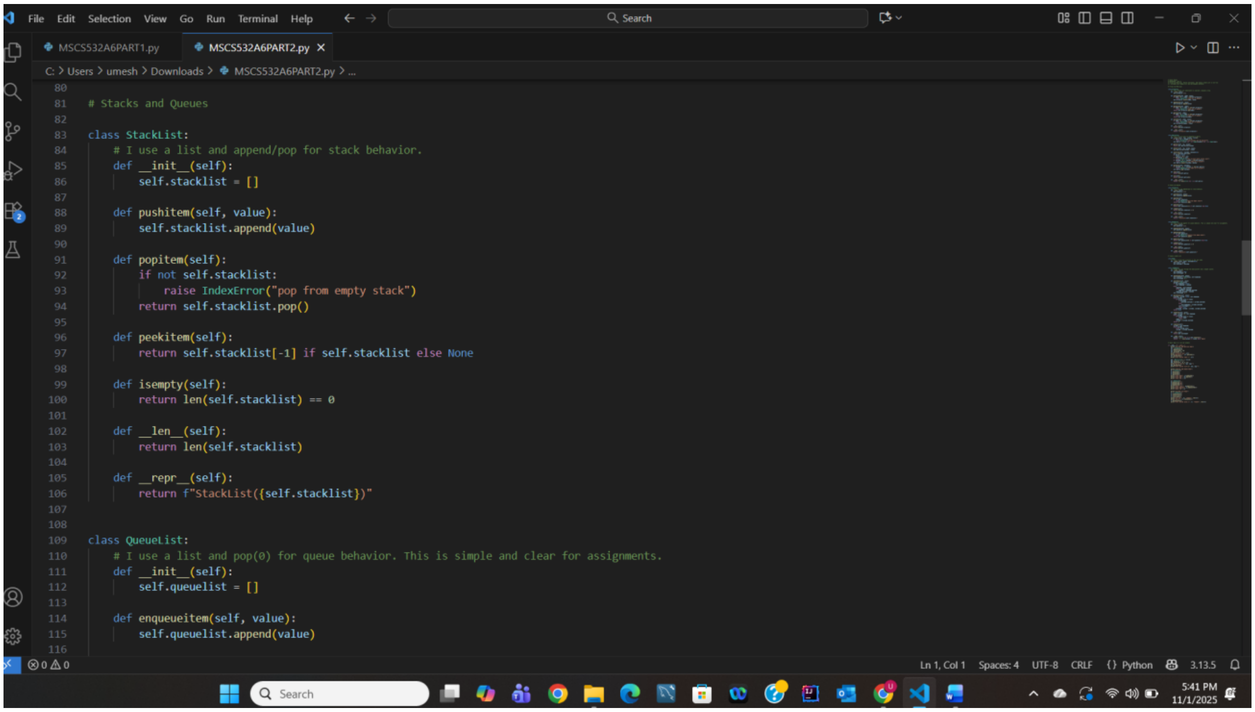
    def getcell(self, row, column):
        return self.matrix[row][column]

    def setcell(self, row, column, value):
        self.matrix[row][column] = value

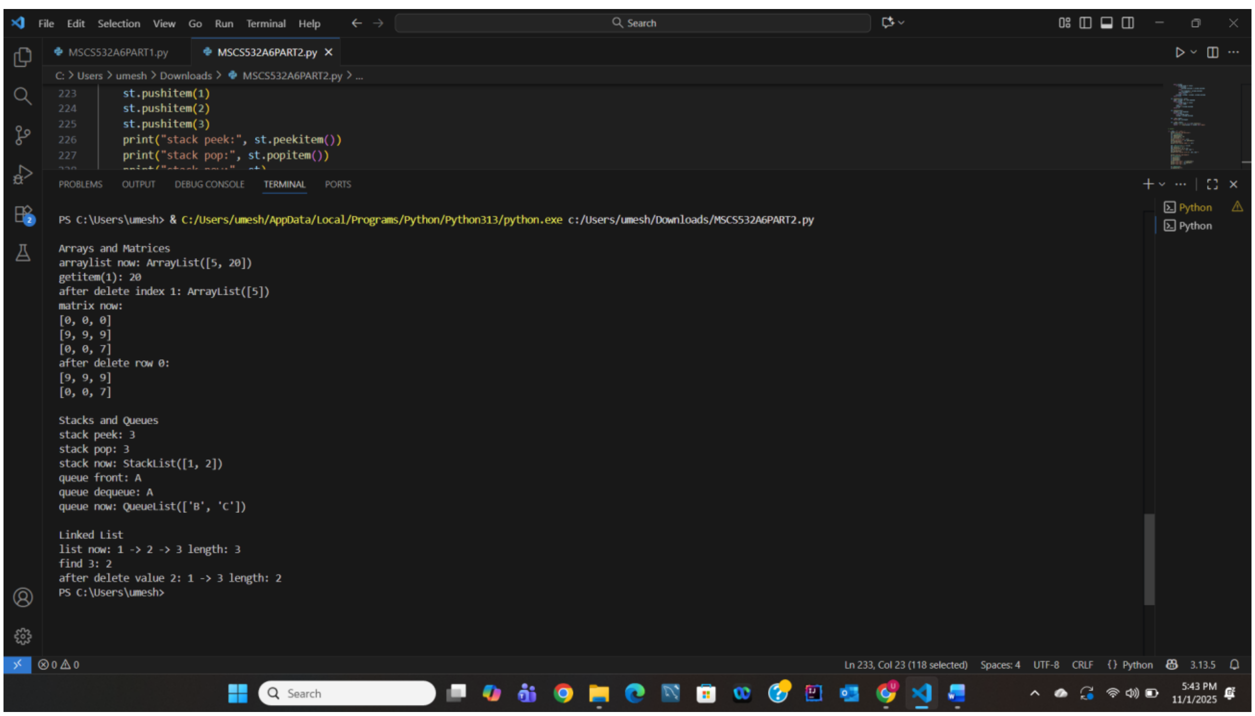
    def insertrow(self, rowindex, newrow=None):
        cols = len(self.matrix[0])
        if newrow is None:
            newrow = [0] * cols
        if len(newrow) != cols:
            raise ValueError("new row must match column count")
        if rowindex < 0 or rowindex > len(self.matrix):
            raise IndexError("row index out of bounds")
        self.matrix.insert(rowindex, newrow)

    def deleterow(self, rowindex):
        if rowindex < 0 or rowindex >= len(self.matrix):
            raise IndexError("row index out of bounds")
        self.matrix.pop(rowindex)

    def rows(self):
Ln 1, Col 1   Spaces: 4   UTF-8   CRLF   Python   3.13.5
5:40 PM
11/1/2025



Output



Performance Analysis

Arrays and matrices are very fast when we want to get or change an item by its position. This just takes O (1) time because the computer knows exactly where the value is. Adding to the end is also fast and also take O (1) time, but adding or removing in the item from middle is slower and taking O(n) as many items must move in between. As Matrices are just arrays inside another array the time complexity of matrices are same as array.

Stacks that are with arrays are also fast as array. Pushing and popping from the end take just O (1) time. For queues, adding to the end is fast, but removing from the front is slow and take O(n)) as everything need to shift left before element can be fully removed. In Linked lists adding or removing the items take O (1) time but finding an item or deleting something in the middle is slower and take O(n) as we must move through each node one by one.

Arrays and linked lists have different strengths. Arrays are good when we want fast access to any position and when the data doesn't change a lot. Linked lists are better when we add and remove items often, especially at the front or back. In conclusion arrays work best for quick lookups, and linked lists work best when the size changes often.

Discussion and application of these data structures

The data structures used in many real-life applications and understanding them is important for solving problems smoothly and efficiently. Arrays and matrices are used in almost every part of computing, such as in storing images, in organizing data in tables, and running scientific or machine learning calculations. Stacks are very common in programming to do tasks like undo features, function calls, and commonly used depth-first search. Queues are used in systems that need to process tasks in order, like print queues, customer service systems, and scheduling processes in an operating system and Linked lists are more helpful where data needs to change often because inserting or deleting items is easier compared to arrays. LinkedList are used in memory management, graph algorithms, and implementing hash tables. Understanding these core structures also helps when learning more advanced topics data structure like trees, graphs, and hashing.

GitHub Commit