Assignment Report

Full Name : Cheng-Chun Hsu (Kevin Hsu)	Student ID: s5390209
Course Name: Data Structures and Algorithms (7011ICT)	Date: 05/28/2025

I. Overview

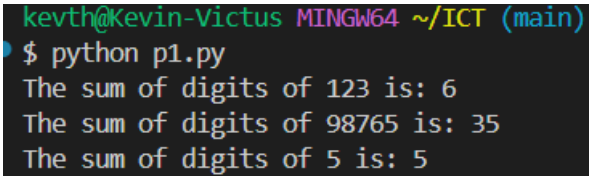
- **Programming Language**: Python 3.10.11
- **Editor**: VS Code

II. Problem 1

Code:

```
# Calculate the sum of digits of a number using recursion
def digitSum(n):
    # Return the number itself if it is a single-digit number
    if n < 10:
        return n

    # Convert the number to string to access each digit
    strNum = str(n)

    # Retrive the first digit and convert it to int
    firstDigit = int(strNum[0])
    # Retrive the rest of digits and convert it to int
    restDigit = int(strNum[1:])

    return firstDigit + digitSum(restDigit)

# Test usage
result = digitSum(123)
print(f"The sum of digits of 123 is: {result}")

result = digitSum(98765)
print(f"The sum of digits of 98765 is: {result}")

result = digitSum(5)
print(f"The sum of digits of 5 is: {result}")
```

Output:



Algorithm Explanation:
The function digitSum(n) recursively computes the sum of the digits of a positive integer:

- It checks if the number is a single digit (base case), and if so, returns it directly.
- Otherwise, it splits the number into two parts:
  - The first digit, and
  - The remaining digits.
- It then recursively sums the first digit with the sum of the remaining digits.

Time Complexity Analysis:

- **In each call:**
  - strNum = str(n) →**O(d)**
  - restDigit = int(strNum[1:]) →**O(d)**
- There are d calls in total

Therefore, the time complexity is **O(d^2)**

Space Complexity Analysis:

- Each recursive call adds a frame to the call stack → d recursive calls

Therefor the space complexity is **O(d)**

Performance Optimization Suggestion:

```
def digitSum(n):
    if n < 10:
        return n
    return n % 10 + digitSum(n // 10)
```

This approach does not need to convert type between int and string, so the time complexity is only O(d) which is more efficient than the previous one.

###----------***END PROBLEM 1***----------###

III. Problem 2

Code:

```
class orderManager():
    def __init__(self):
        self.order = []

    # Add order to the queue
    def addOrder(self, order):
        self.order.append(order)
        print("Order is added!\n")

    # Process the first order in the queue
    def processOrder(self):
        if self.order:
            print(f"{ self.order[0]} is ready to process!\n")
            self.order.pop(0)
        else:
            print("No order to process!\n")

    # Remove a specific order from the queue
    def removeOrder(self, order_to_remove):
        if order_to_remove in self.order:
            self.order.remove(order_to_remove)
            print("The order is cancelled!\n")
        else:
            print("Order not found!\n")

    # View the next order without removing it
    def nextOrder(self):
        if len(self.order)>0:
            print(f"Next order is {self.order[0]}")
            return self.order[0]
        else:
            print("There's no order left.\n")

    # Check if there are pending orders
    def checkOrder(self):
        if len(self.order)>0:
            print(f"There are {len(self.order)} orders in the line!\n")
        else:
            print("There are no pending order!\n")
```

Time Complexity Analysis:

- Function addOrder: Simply append an element to the queue → **O(1)**
- Function processOrder: When pop an element, each rest element in the array needs to be shifted. → **O(n)**
- Function removeOrder: The best case is to remove the last element so the time complexity will be **O(1)**. The worst case is to remove the first element so the time complexity will be **O(n)**.

- Function nextOrder: Simply check the first element in the array → O(1)
- Function checkOrder: Simply check the length of the array → O(1)

Space Complexity Analysis:

- All operations are done in-place on a list. So the space complexity is O(n).

Performance Optimization Suggestion:

- If using type dict to store the order data, when removing an order, we don't need to use pop() function and don't need to shift the left elements so the time complexity will be O(1) when we do the remove operation.

###----------***END PROBLEM 2***----------###

IV. Problem 3

Code:

```
# Quick Sort
def quickSort(arr):
    # If the list has less than 2 elements, it's already sorted
    if len(arr)<=1:
        return arr

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

    # Partition elements into three groups: less than the pivot, greater than
    the pivot, and equal to the pivot
    greaterNum = []
    for num in arr:
        if num > pivot:
            greaterNum.append(num)

    lessNum = []
    for num in arr:
        if num < pivot:
            lessNum.append(num)

    equalNum = []
    for num in arr:
        if num == pivot:
            equalNum.append(num)

    # Recursively sort the less and greater lists and combine with the equal
    list
    return quickSort(lessNum) + equalNum + quickSort(greaterNum)
```

```
# Merge Sort
def mergeSort(arr):
    # If the list has less than 2 elements, it's already sorted
    if len(arr)<=1:
        return arr

    # Find the middle index to divide the array into two halves (left and right)
    midNumIndex = len(arr)//2
    arrLeft = arr[0:midNumIndex]
    arrRight = arr[midNumIndex:]

    # Recursively sort both halves
    leftSort = mergeSort(arrLeft)
    rightSort = mergeSort(arrRight)

    # Merge the sorted left and right halves into a single sorted list
    mergedArr = []
    i = 0
    j = 0

    # Compare elements from both halves and append the smaller one to
    mergedArr
    while i < len(leftSort) and j < len(rightSort):
        if leftSort[i] <= rightSort[j]:
            mergedArr.append(leftSort[i])
            i += 1
        else:
            mergedArr.append(rightSort[j])
            j += 1

    mergedArr = mergedArr + leftSort[i:]
    mergedArr = mergedArr + rightSort[j:]

    return mergedArr
```

```
# Insertion Sort
def insertSort(arr):
    # Initialize the sorted array with the first element
    target = arr[0]
    sortedArr = []
    sortedArr.append(target)

    # Loop through the remaining elements of the input array
    for i in range(1, len(arr)):
        compareNum = arr[i]

        # If the current number is smaller than the first element in sortedArr,
insert it at the beginning
        if compareNum < sortedArr[0]:
            sortedArr.insert(0, compareNum)
            continue

        inserted = False

        # Find the correct position in sortedArr for insertion
        for _ in sortedArr:
            if compareNum < _:
                sortedArr.insert(sortedArr.index(_), compareNum)
                inserted = True
                break

        # If the number is larger than all elements in sortedArr, append it to the
end
        if not inserted:
            sortedArr.append(compareNum)

    return sortedArr
```

```
# Selection Sort
def SelectionSort (arr):
    # Create an empty list to store the sorted elements
    sortedArr = []

    while len(arr) > 0:
        # Assume the first element is the minimum
        minNum = arr[0]

        # Find the real minimum number in the array
        for num in arr:
            if num < minNum:
                minNum = num

        # Append the minimum number to the sorted array
        sortedArr.append(minNum)

        # Remove the minimum number from the originla array
        arr.remove(minNum)


    return sortedArr
```

Performance Test:
The performance test is based on the code following:

```
import time, random

# Quick Sort
def quickSort(arr):
    # If the list has less than 2 elements, it's already sorted
    if len(arr)<=1:
        return arr

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

    # Partition elements into three groups: less than the pivot, greater than
the pivot, and equal to the pivot
    greaterNum = []
    for num in arr:
        if num > pivot:
            greaterNum.append(num)

    lessNum = []
    for num in arr:
        if num < pivot:
            lessNum.append(num)

    equalNum = []
    for num in arr:
        if num == pivot:
            equalNum.append(num)

    # Recursively sort the less and greater lists and combine with the equal
list
    return quickSort(lessNum) + equalNum + quickSort(greaterNum)

# Merge Sort
def mergeSort(arr):
    # If the list has less than 2 elements, it's already sorted
    if len(arr)<=1:
        return arr

    # Find the middle index to divide the array into two halves (left and right)
    midNumIndex = len(arr)//2
    arrLeft = arr[0:midNumIndex]
    arrRight = arr[midNumIndex:]

    # Recursively sort both halves
    leftSort = mergeSort(arrLeft)
    rightSort = mergeSort(arrRight)

    # Merge the sorted left and right halves into a single sorted list
    mergedArr = []
```

```
    i = 0
    j = 0

    # Compare elements from both halves and append the smaller one to
mergedArr
    while i < len(leftSort) and j < len(rightSort):
        if leftSort[i] <= rightSort[j]:
            mergedArr.append(leftSort[i])
            i += 1
        else:
            mergedArr.append(rightSort[j])
            j += 1

    mergedArr = mergedArr + leftSort[i:]
    mergedArr = mergedArr + rightSort[j:]

    return mergedArr

# Insertion Sort
def insertSort(arr):
    # Initialize the sorted array with the first element
    target = arr[0]
    sortedArr = []
    sortedArr.append(target)

    # Loop through the remaining elements of the input array
    for i in range(1, len(arr)):
        compareNum = arr[i]

        # If the current number is smaller than the first element in sortedArr,
insert it at the beginning
        if compareNum < sortedArr[0]:
            sortedArr.insert(0, compareNum)
            continue

        inserted = False

        # Find the correct position in sortedArr for insertion
        for _ in sortedArr:
            if compareNum < _:
                sortedArr.insert(sortedArr.index(_), compareNum)
                inserted = True
                break

        # If the number is larger than all elements in sortedArr, append it to the
end
        if not inserted:
            sortedArr.append(compareNum)
```

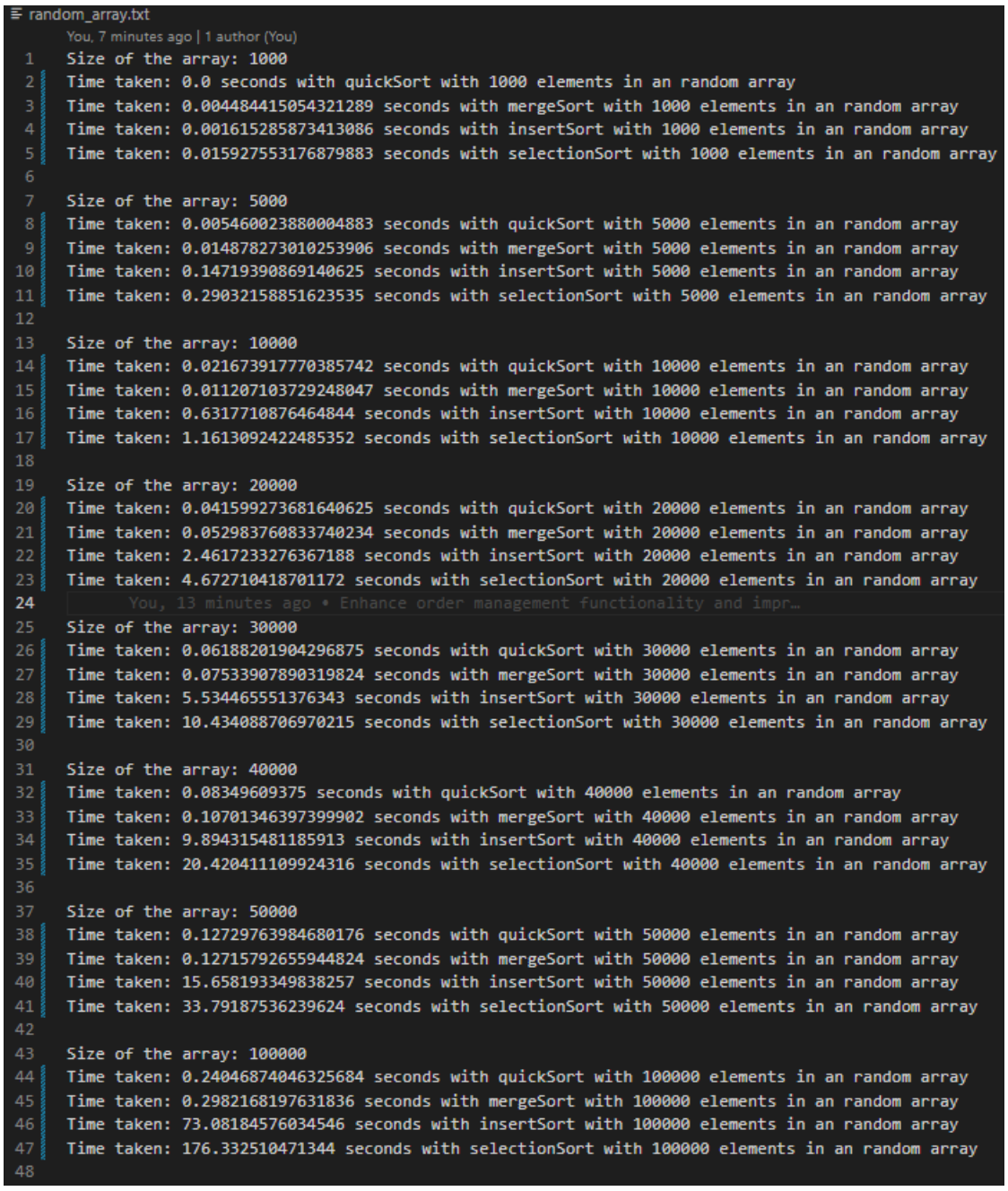
```
    return sortedArr

# Selection Sort
def selectionSort (arr):
    # Create an empty list to store the sorted elements
    sortedArr = []

    while len(arr) > 0:
        # Assume the first element is the minimum
        minNum = arr[0]

        # Find the real minimum number in the array
        for num in arr:
            if num < minNum:
                minNum = num

        # Append the minimum number to the sorted array
        sortedArr.append(minNum)

        # Remove the minimum number from the originla array
        arr.remove(minNum)


    return sortedArr



# Performance Testing
sizes = [1000, 5000, 10000, 20000, 30000, 40000, 50000, 100000]

# ------------------------------- Random Array Test-------------------------------
# Generate a random array
for size in sizes:
    randomArray = []
    for n in range(size):
        randomArray.append(n)

    # Shuffle the array
    random.shuffle(randomArray)

    with open("random_array.txt", "a") as f:
        f.write(f"Size of the array: {size}\n")



    startTime = time.time()
    result = quickSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
```

```
        f.write(f"Time taken: {endTime - startTime} seconds with QuickSort
with {size} elements in an random array\n")
    print(f"Time taken: {endTime - startTime} seconds with QuickSort with
{size} elements in an random array")

    startTime = time.time()
    mergeSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with MergeSort
with {size} elements in an random array\n")
    print(f"Time taken: {endTime - startTime} seconds with MergeSort with
{size} elements in an random array")

    startTime = time.time()
    insertSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with InsertSort
with {size} elements in an random array\n")
    print(f"Time taken: {endTime - startTime} seconds with InsertSort with
{size} elements in an random array")

    startTime = time.time()
    selectionSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with selectionSort
with {size} elements in an random array\n\n")
    print(f"Time taken: {endTime - startTime} seconds with selectionSort with
{size} elements in an random array")


# ------------------------------ Reversed Array Test------------------------------
# Generate a reversed sorted array
for size in sizes:
    reverseArray = []
    for n in range(size):
        reverseArray.append(n)

    reverseArray.reverse()

    with open("reversed_array.txt", "a") as f:
        f.write(f"Size of the array: {size}\n")

    startTime = time.time()
    result = quickSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
```

```
        f.write(f"Time taken: {endTime - startTime} seconds with QuickSort
with {size} elements in an reversed array\n")
    print(f"Time taken: {endTime - startTime} seconds with QuickSort with
{size} elements in an reversed array")

    startTime = time.time()
    mergeSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with MergeSort
with {size} elements in an reversed array\n")
    print(f"Time taken: {endTime - startTime} seconds with MergeSort with
{size} elements in an reversed array")

    startTime = time.time()
    insertSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with InsertSort
with {size} elements in an reversed array\n")
    print(f"Time taken: {endTime - startTime} seconds with InsertSort with
{size} elements in an reversed array")

    startTime = time.time()
    selectionSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with selectionSort
with {size} elements in an reversed array\n\n")
    print(f"Time taken: {endTime - startTime} seconds with selectionSort with
{size} elements in an reversedom array")

# ------------------------------ Duplicate Value Array Test----------------------------
---
# Generate a duplicate value array
for size in sizes:
    duplicateArray = []
    for n in range(size):
        duplicateArray.append(random.randint(0, 10))


    with open("duplicate_array.txt", "a") as f:
        f.write(f"Size of the array: {size}\n")


    startTime = time.time()
    result = quickSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
```

```
        f.write(f"Time taken: {endTime - startTime} seconds with QuickSort
with {size} elements in an duplicate value array\n")
    print(f"Time taken: {endTime - startTime} seconds with QuickSort with
{size} elements in an duplicate value array")

    startTime = time.time()
    mergeSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with MergeSort
with {size} elements in an duplicate value array\n")
    print(f"Time taken: {endTime - startTime} seconds with MergeSort with
{size} elements in an duplicate value array")

    startTime = time.time()
    insertSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with InsertSort
with {size} elements in an duplicate value array\n")
    print(f"Time taken: {endTime - startTime} seconds with InsertSort with
{size} elements in an duplicate value array")

    startTime = time.time()
    selectionSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with selectionSort
with {size} elements in an duplicate value array\n\n")
    print(f"Time taken: {endTime - startTime} seconds with selectionSort with
{size} elements in an duplicate value array")
```

Random Array:



Reversed Array:

reversed_array.txt

You, 5 minutes ago | 1 author (You)

```
Size of the array: 1000
Time taken: 0.003204822540283203 seconds with quickSort with 1000 elements in an reversed array
Time taken: 0.0006313323974609375 seconds with mergeSort with 1000 elements in an reversed array
Time taken: 0.0 seconds with insertSort with 1000 elements in an reversed array
Time taken: 0.015502452850341797 seconds with selectionSort with 1000 elements in an reversed array

Size of the array: 5000
Time taken: 0.006669044494628906 seconds with quickSort with 5000 elements in an reversed array
Time taken: 0.011598825454711914 seconds with mergeSort with 5000 elements in an reversed array
Time taken: 0.004581928253173828 seconds with insertSort with 5000 elements in an reversed array
Time taken: 0.3723735809326172 seconds with selectionSort with 5000 elements in an reversed array

Size of the array: 10000
Time taken: 0.022426366806030273 seconds with quickSort with 10000 elements in an reversed array
Time taken: 0.018366575241088867 seconds with mergeSort with 10000 elements in an reversed array
Time taken: 0.015448570251464844 seconds with insertSort with 10000 elements in an reversed array
Time taken: 1.5490210056304932 seconds with selectionSort with 10000 elements in an reversed array

Size of the array: 20000
Time taken: 0.0408477783203125 seconds with quickSort with 20000 elements in an reversed array
Time taken: 0.04641890525817871 seconds with mergeSort with 20000 elements in an reversed array
Time taken: 0.04637598991394043 seconds with insertSort with 20000 elements in an reversed array
Time taken: 6.314342498779297 seconds with selectionSort with 20000 elements in an reversed array

Size of the array: 30000
Time taken: 0.05135369300842285 seconds with quickSort with 30000 elements in an reversed array
Time taken: 0.05869293212890625 seconds with mergeSort with 30000 elements in an reversed array
Time taken: 0.11419343948364258 seconds with insertSort with 30000 elements in an reversed array
Time taken: 14.588743209838867 seconds with selectionSort with 30000 elements in an reversed array

Size of the array: 40000
Time taken: 0.08524823188781738 seconds with quickSort with 40000 elements in an reversed array
Time taken: 0.06187081336975098 seconds with mergeSort with 40000 elements in an reversed array
Time taken: 0.1994340419769287 seconds with insertSort with 40000 elements in an reversed array
Time taken: 25.285762310028076 seconds with selectionSort with 40000 elements in an reversed array

Size of the array: 50000
Time taken: 0.08826518058776855 seconds with quickSort with 50000 elements in an reversed array
Time taken: 0.09984326362609863 seconds with mergeSort with 50000 elements in an reversed array
Time taken: 0.31672048568725586 seconds with insertSort with 50000 elements in an reversed array
Time taken: 59.8646035194397 seconds with selectionSort with 50000 elements in an reversed array

Size of the array: 100000
Time taken: 0.18355035781860352 seconds with quickSort with 100000 elements in an reversed array
Time taken: 0.19094514846801758 seconds with mergeSort with 100000 elements in an reversed array
Time taken: 1.4366059303283691 seconds with insertSort with 100000 elements in an reversed array
Time taken: 161.44582772254944 seconds with selectionSort with 100000 elements in an reversed array
```

Duplicate Value Array:

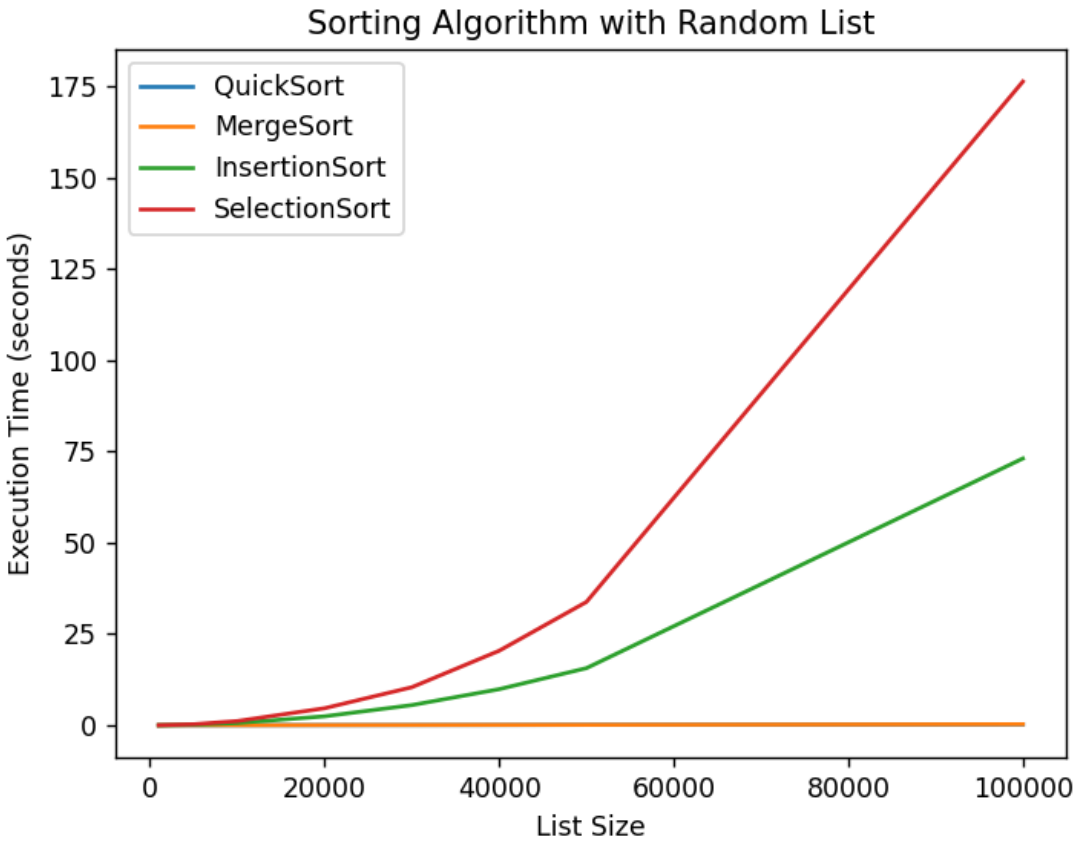
duplicate_array.txt

You, 6 minutes ago | 1 author (You)

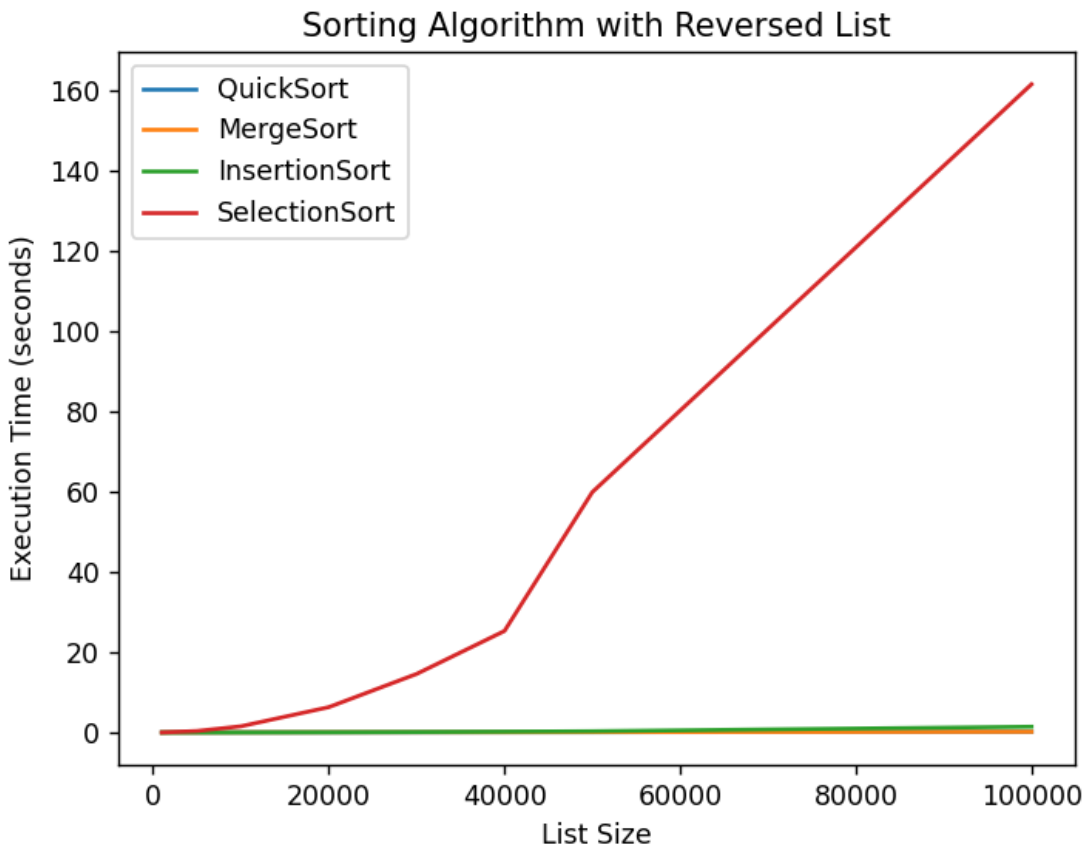
```
Size of the array: 1000
Time taken: 0.0 seconds with quickSort with 1000 elements in an duplicate value array
Time taken: 0.007704973220825195 seconds with mergeSort with 1000 elements in an duplicate value array
Time taken: 0.0034034252166748047 seconds with insertSort with 1000 elements in an duplicate value array
Time taken: 0.01827073097229004 seconds with selectionSort with 1000 elements in an duplicate value array

Size of the array: 5000
Time taken: 0.0019626617431640625 seconds with quickSort with 5000 elements in an duplicate value array
Time taken: 0.0115966796875 seconds with mergeSort with 5000 elements in an duplicate value array
Time taken: 0.15789127349853516 seconds with insertSort with 5000 elements in an duplicate value array
Time taken: 0.3714919090270996 seconds with selectionSort with 5000 elements in an duplicate value array

Size of the array: 10000
Time taken: 0.007247447967529297 seconds with quickSort with 10000 elements in an duplicate value array
Time taken: 0.025011062622070312 seconds with mergeSort with 10000 elements in an duplicate value array
Time taken: 0.6947760581970215 seconds with insertSort with 10000 elements in an duplicate value array
Time taken: 1.3271369934082031 seconds with selectionSort with 10000 elements in an duplicate value array

Size of the array: 20000
Time taken: 0.01677227020263672 seconds with quickSort with 20000 elements in an duplicate value array
Time taken: 0.04911613464355469 seconds with mergeSort with 20000 elements in an duplicate value array
Time taken: 2.6906583309173584 seconds with insertSort with 20000 elements in an duplicate value array
Time taken: 5.717264413833618 seconds with selectionSort with 20000 elements in an duplicate value array

Size of the array: 30000
Time taken: 0.012505769729614258 seconds with quickSort with 30000 elements in an duplicate value array
Time taken: 0.07962656021118164 seconds with mergeSort with 30000 elements in an duplicate value array
Time taken: 6.282073736190796 seconds with insertSort with 30000 elements in an duplicate value array
Time taken: 12.712151765823364 seconds with selectionSort with 30000 elements in an duplicate value array

Size of the array: 40000
Time taken: 0.019968509674072266 seconds with quickSort with 40000 elements in an duplicate value array
Time taken: 0.09972381591796875 seconds with mergeSort with 40000 elements in an duplicate value array
Time taken: 11.663087606430054 seconds with insertSort with 40000 elements in an duplicate value array
Time taken: 22.454686641693115 seconds with selectionSort with 40000 elements in an duplicate value array

Size of the array: 50000
Time taken: 0.018741846084594727 seconds with quickSort with 50000 elements in an duplicate value array
Time taken: 0.1324143409729004 seconds with mergeSort with 50000 elements in an duplicate value array
Time taken: 16.438893795013428 seconds with insertSort with 50000 elements in an duplicate value array
Time taken: 31.222820520401 seconds with selectionSort with 50000 elements in an duplicate value array

Size of the array: 100000
Time taken: 0.029422760009765625 seconds with quickSort with 100000 elements in an duplicate value array
Time taken: 0.25153446197509766 seconds with mergeSort with 100000 elements in an duplicate value array
Time taken: 62.55976176261902 seconds with insertSort with 100000 elements in an duplicate value array
Time taken: 132.16945481300354 seconds with selectionSort with 100000 elements in an duplicate value array
```

Graph Analysis:

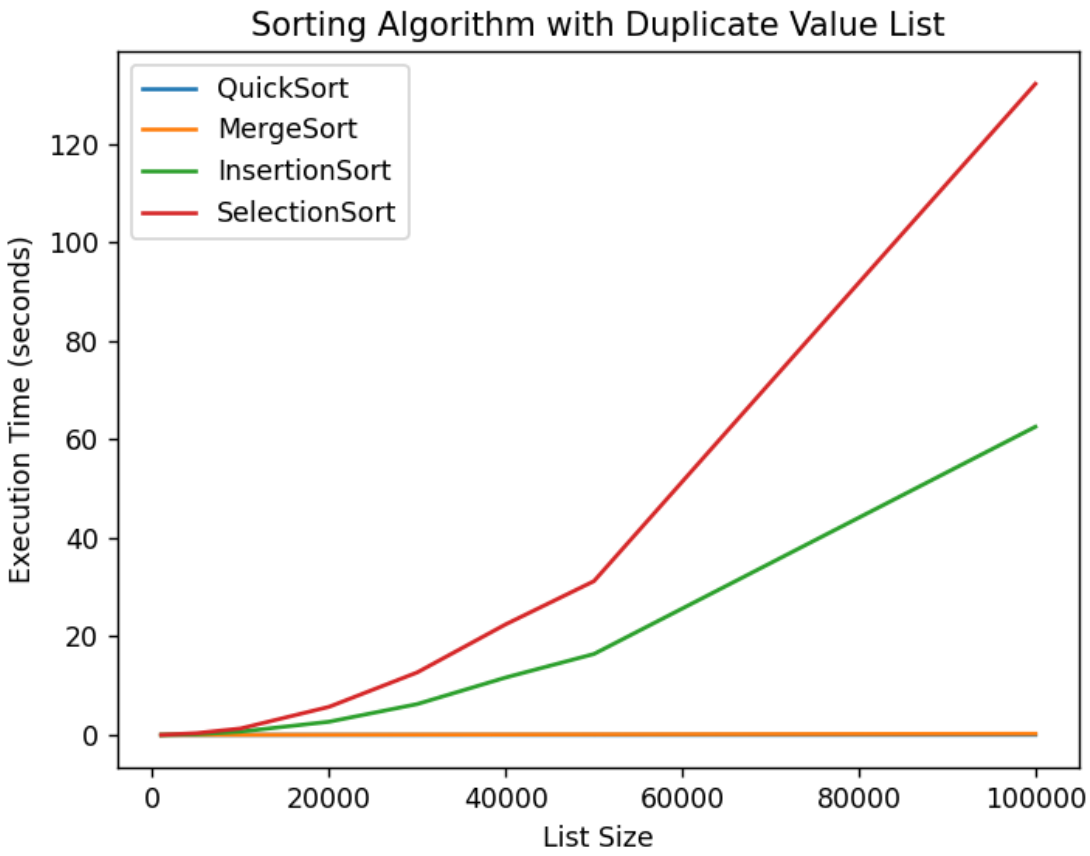
Random List:



Reversed List:



Duplicate Value List:



Time Complexity:
QuickSort:
Best: O(n log n)
Worst: O(n^2)
Average: O(n log n)

MergeSort:
Best: O(n log n)
Worst: O(n log n)
Average: O(n log n)

InsertionSort:
Best: O(n)
Worst: O(n^2)
Average: O(n^2)

SelectionSort:
Best: O(n^2)
Worst: O(n^2)
Average: O(n^2)

Space Complexity:
QuickSort: O(log n)

MergeSort: O(n)

InsertionSort: O(1)

SelectionSort: O(1)

Analysis:
Input Type: Random List
Best Choice: Quick Sort

Input Type: Reverse Sorted List
Best Choice: Merge Sort

Input Type: List with duplicate values
Best Choice: QuickSort

Input Type: Small Size List (<1000)
Best Choice: Quick Sort/ Insertion Sort

Why QuickSort usually perform better than others?
The QuickSort has O(log n) in average case, so in most cases it can perform better than others.

In what scenario does QuickSort perform poorly?
In reversed sorted array scenario, we can see that with a significant number of list, QuickSort is worser than MergeSort and InsertSort as QuickSort always choose the fixed position number of the pivot, such as first one, last one or the middle one.

###----------***END PROBLEM 3***----------###

V. Problem 4

Code:

```
def searchItem(targetWeight, targetInventory, startIndex=0,
endIndex=None):
    if endIndex is None:
        endIndex = len(targetInventory) - 1

    print(f"Start Index: {startIndex}, End Index: {endIndex}")
    midItemIndex = (startIndex + endIndex) // 2

    # If search range is invalid (target not found)
    if startIndex > endIndex:
        # Handle edge cases where startIndex or endIndex is out of bounds
        if startIndex >= len(targetInventory):
            closestIndex = len(targetInventory) - 1
        elif endIndex < 0:
            closestIndex = 0
        else:
            # Choose the closest between endIndex and startIndex
            left = endIndex
            right = startIndex
            if abs(targetInventory[left] - targetWeight) <=
abs(targetInventory[right] - targetWeight):
                closestIndex = left
            else:
                closestIndex = right
        print(f"Item is not found. The weight of the closest available item is
{targetInventory[closestIndex]}\n")
        return closestIndex

    if targetInventory[midItemIndex] == targetWeight:
        print(f"Item is found at index {midItemIndex}.\n")
        return midItemIndex
    elif targetInventory[midItemIndex] < targetWeight:
        return searchItem(targetWeight, targetInventory, midItemIndex + 1,
endIndex)
    else:
        return searchItem(targetWeight, targetInventory, startIndex,
midItemIndex - 1)
```

Explanation:
I use Binary Search which is more efficient than linear search in this case

Time Complexity:

Best Case: If the target is right in the middle → **O(1)**
Worst Case: Keep slicing the array in half until it can be done → **O(log n)**

Space Complexity: O(log n)

###----------***END PROBLEM 4***----------###

VI. Problem 5

Code:

```
def compute_cost(distance, toll, mode):
    if mode == "distance":
        return distance
    elif mode == "toll":
        return toll
    elif mode == "balanced":
        return distance + toll

def dijkstra(graph, start, end, mode):
    nodes = {}
    for v in graph:
        nodes[v] = {
            "dist": float("inf"),
            "toll": float("inf"),
            "cost": float("inf"),
            "prev": None
        }

    nodes[start]["dist"] = 0
    nodes[start]["toll"] = graph[start]["toll"]
    nodes[start]["cost"] = compute_cost(0, graph[start]["toll"], mode)

    queue = [start]
    visited = set()

    while queue:
        # Select node with lowest cost
        current = min(queue, key=lambda x: nodes[x]["cost"])
        queue.remove(current)

        if current == end:
            break

        visited.add(current)

        for neighbor in graph[current]["roads"]:
            if neighbor in visited:
                continue

            edge_dist = graph[current]["roads"][neighbor]
            new_dist = nodes[current]["dist"] + edge_dist
            new_toll = nodes[current]["toll"] + graph[neighbor]["toll"]
            new_cost = compute_cost(new_dist, new_toll, mode)

            if new_cost < nodes[neighbor]["cost"]:
                nodes[neighbor]["dist"] = new_dist
                nodes[neighbor]["toll"] = new_toll
                nodes[neighbor]["cost"] = new_cost
                nodes[neighbor]["prev"] = current
```

```
            if neighbor not in queue:
                queue.append(neighbor)

    # Reconstruct path
    path = []
    node = end
    while node is not None:
        path.insert(0, node)
        node = nodes[node]["prev"]

    return {
        "path": path,
        "distance": nodes[end]["dist"],
        "toll": nodes[end]["toll"],
        "cost": nodes[end]["cost"]
    }
```

Time Complexity: $O(V^2 + E)$

Space Complexity: $O(V + E)$

###----------***END PROBLEM 5***----------###

Appendix

1. Problem 1

```
# Calculate the sum of digits of a number using recursion
def digitSum(n):
    # Return the number itself if it is a single-digit number
    if n < 10:
        return n

    # Convert the number to string to access each digit
    strNum = str(n)

    # Retrive the first digit and convert it to int
    firstDigit = int(strNum[0])
    # Retrive the rest of digits and convert it to int
    restDigit = int(strNum[1:])

    return firstDigit + digitSum(restDigit)

# Test usage
result = digitSum(123)
print(f"The sum of digits of 123 is: {result}")

result = digitSum(98765)
print(f"The sum of digits of 98765 is: {result}")

result = digitSum(5)
print(f"The sum of digits of 5 is: {result}")
```

2. Problem 2

```
class orderManager():
    def __init__(self):
        self.order = []

    # Add order to the queue
    def addOrder(self, order):
        self.order.append(order)
        print("Order is added!\n")

    # Process the first order in the queue
    def processOrder(self):
        if self.order:
            print(f"{ self.order[0]} is ready to process!\n")
            self.order.pop(0)
        else:
            print("No order to process!\n")

    # Remove a specific order from the queue
    def removeOrder(self, order_to_remove):
        if order_to_remove in self.order:
            self.order.remove(order_to_remove)
            print("The order is cancelled!\n")
        else:
            print("Order not found!\n")

    # View the next order without removing it
    def nextOrder(self):
        if len(self.order)>0:
            print(f"Next order is {self.order[0]}")
            return self.order[0]
        else:
            print("There's no order left.\n")

    # Check if there are pending orders
    def checkOrder(self):
        if len(self.order)>0:
            print(f"There are {len(self.order)} orders in the line!\n")
        else:
            print("There are no pending order!\n")
```

3. Problem 3

```
import time, random

# Quick Sort
def quickSort(arr):
    # If the list has less than 2 elements, it's already sorted
    if len(arr)<=1:
        return arr

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

    # Partition elements into three groups: less than the pivot, greater than
the pivot, and equal to the pivot
    greaterNum = []
    for num in arr:
        if num > pivot:
            greaterNum.append(num)

    lessNum = []
    for num in arr:
        if num < pivot:
            lessNum.append(num)

    equalNum = []
    for num in arr:
        if num == pivot:
            equalNum.append(num)

    # Recursively sort the less and greater lists and combine with the equal
list
    return quickSort(lessNum) + equalNum + quickSort(greaterNum)

# Merge Sort
def mergeSort(arr):
    # If the list has less than 2 elements, it's already sorted
    if len(arr)<=1:
        return arr

    # Find the middle index to divide the array into two halves (left and right)
    midNumIndex = len(arr)//2
    arrLeft = arr[0:midNumIndex]
    arrRight = arr[midNumIndex:]

    # Recursively sort both halves
    leftSort = mergeSort(arrLeft)
    rightSort = mergeSort(arrRight)

    # Merge the sorted left and right halves into a single sorted list
    mergedArr = []
```

```
    i = 0
    j = 0

    # Compare elements from both halves and append the smaller one to
mergedArr
    while i < len(leftSort) and j < len(rightSort):
        if leftSort[i] <= rightSort[j]:
            mergedArr.append(leftSort[i])
            i += 1
        else:
            mergedArr.append(rightSort[j])
            j += 1

    mergedArr = mergedArr + leftSort[i:]
    mergedArr = mergedArr + rightSort[j:]

    return mergedArr

# Insertion Sort
def insertSort(arr):
    # Initialize the sorted array with the first element
    target = arr[0]
    sortedArr = []
    sortedArr.append(target)

    # Loop through the remaining elements of the input array
    for i in range(1, len(arr)):
        compareNum = arr[i]

        # If the current number is smaller than the first element in sortedArr,
insert it at the beginning
        if compareNum < sortedArr[0]:
            sortedArr.insert(0, compareNum)
            continue

        inserted = False

        # Find the correct position in sortedArr for insertion
        for _ in sortedArr:
            if compareNum < _:
                sortedArr.insert(sortedArr.index(_), compareNum)
                inserted = True
                break

        # If the number is larger than all elements in sortedArr, append it to the
end
        if not inserted:
            sortedArr.append(compareNum)
```

```
    return sortedArr

# Selection Sort
def selectionSort (arr):
    # Create an empty list to store the sorted elements
    sortedArr = []

    while len(arr) > 0:
        # Assume the first element is the minimum
        minNum = arr[0]

        # Find the real minimum number in the array
        for num in arr:
            if num < minNum:
                minNum = num

        # Append the minimum number to the sorted array
        sortedArr.append(minNum)

        # Remove the minimum number from the originla array
        arr.remove(minNum)


    return sortedArr



# Performance Testing
sizes = [1000, 5000, 10000, 20000, 30000, 40000, 50000, 100000]

# ------------------------------- Random Array Test-------------------------------
# Generate a random array
for size in sizes:
    randomArray = []
    for n in range(size):
        randomArray.append(n)

    # Shuffle the array
    random.shuffle(randomArray)

    with open("random_array.txt", "a") as f:
        f.write(f"Size of the array: {size}\n")



    startTime = time.time()
    result = quickSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
```

```
        f.write(f"Time taken: {endTime - startTime} seconds with QuickSort
with {size} elements in an random array\n")
    print(f"Time taken: {endTime - startTime} seconds with QuickSort with
{size} elements in an random array")

    startTime = time.time()
    mergeSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with MergeSort
with {size} elements in an random array\n")
    print(f"Time taken: {endTime - startTime} seconds with MergeSort with
{size} elements in an random array")

    startTime = time.time()
    insertSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with InsertSort
with {size} elements in an random array\n")
    print(f"Time taken: {endTime - startTime} seconds with InsertSort with
{size} elements in an random array")

    startTime = time.time()
    selectionSort(randomArray)
    endTime = time.time()
    with open("random_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with selectionSort
with {size} elements in an random array\n\n")
    print(f"Time taken: {endTime - startTime} seconds with selectionSort with
{size} elements in an random array")


# ------------------------------ Reversed Array Test------------------------------
# Generate a reversed sorted array
for size in sizes:
    reverseArray = []
    for n in range(size):
        reverseArray.append(n)

    reverseArray.reverse()

    with open("reversed_array.txt", "a") as f:
        f.write(f"Size of the array: {size}\n")

    startTime = time.time()
    result = quickSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
```

```
        f.write(f"Time taken: {endTime - startTime} seconds with QuickSort
with {size} elements in an reversed array\n")
    print(f"Time taken: {endTime - startTime} seconds with QuickSort with
{size} elements in an reversed array")

    startTime = time.time()
    mergeSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with MergeSort
with {size} elements in an reversed array\n")
    print(f"Time taken: {endTime - startTime} seconds with MergeSort with
{size} elements in an reversed array")

    startTime = time.time()
    insertSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with InsertSort
with {size} elements in an reversed array\n")
    print(f"Time taken: {endTime - startTime} seconds with InsertSort with
{size} elements in an reversed array")

    startTime = time.time()
    selectionSort(reverseArray)
    endTime = time.time()
    with open("reversed_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with selectionSort
with {size} elements in an reversed array\n\n")
    print(f"Time taken: {endTime - startTime} seconds with selectionSort with
{size} elements in an reversedom array")

# ------------------------------ Duplicate Value Array Test----------------------------
---
# Generate a duplicate value array
for size in sizes:
    duplicateArray = []
    for n in range(size):
        duplicateArray.append(random.randint(0, 10))


    with open("duplicate_array.txt", "a") as f:
        f.write(f"Size of the array: {size}\n")


    startTime = time.time()
    result = quickSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
```

```
        f.write(f"Time taken: {endTime - startTime} seconds with QuickSort
with {size} elements in an duplicate value array\n")
    print(f"Time taken: {endTime - startTime} seconds with QuickSort with
{size} elements in an duplicate value array")

    startTime = time.time()
    mergeSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with MergeSort
with {size} elements in an duplicate value array\n")
    print(f"Time taken: {endTime - startTime} seconds with MergeSort with
{size} elements in an duplicate value array")

    startTime = time.time()
    insertSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with InsertSort
with {size} elements in an duplicate value array\n")
    print(f"Time taken: {endTime - startTime} seconds with InsertSort with
{size} elements in an duplicate value array")

    startTime = time.time()
    selectionSort(duplicateArray)
    endTime = time.time()
    with open("duplicate_array.txt", "a") as f:
        f.write(f"Time taken: {endTime - startTime} seconds with selectionSort
with {size} elements in an duplicate value array\n\n")
    print(f"Time taken: {endTime - startTime} seconds with selectionSort with
{size} elements in an duplicate value array")
```

4. Problem 4

```
def searchItem(targetWeight, targetInventory, startIndex=0,
endIndex=None):
    if endIndex is None:
        endIndex = len(targetInventory) - 1

    print(f"Start Index: {startIndex}, End Index: {endIndex}")
    midItemIndex = (startIndex + endIndex) // 2

    # If search range is invalid (target not found)
    if startIndex > endIndex:
        # Handle edge cases where startIndex or endIndex is out of bounds
        if startIndex >= len(targetInventory):
            closestIndex = len(targetInventory) - 1
        elif endIndex < 0:
            closestIndex = 0
        else:
            # Choose the closest between endIndex and startIndex
            left = endIndex
            right = startIndex
            if abs(targetInventory[left] - targetWeight) <=
abs(targetInventory[right] - targetWeight):
                closestIndex = left
            else:
                closestIndex = right
        print(f"Item is not found. The weight of the closest available item is
{targetInventory[closestIndex]}\n")
        return closestIndex

    if targetInventory[midItemIndex] == targetWeight:
        print(f"Item is found at index {midItemIndex}.\n")
        return midItemIndex
    elif targetInventory[midItemIndex] < targetWeight:
        return searchItem(targetWeight, targetInventory, midItemIndex + 1,
endIndex)
    else:
        return searchItem(targetWeight, targetInventory, startIndex,
midItemIndex - 1)



inventory = [10, 20, 30, 40, 50, 60, 70, 80, 90, 91.49, 92.5000001, 92.52]
result = searchItem(92, inventory)
print(f"Item is found at index {result}.\n")
```

5. Problem 5

```
def compute_cost(distance, toll, mode):
    if mode == "distance":
        return distance
    elif mode == "toll":
        return toll
    elif mode == "balanced":
        return distance + toll

def dijkstra(graph, start, end, mode):
    nodes = {}
    for v in graph:
        nodes[v] = {
            "dist": float("inf"),
            "toll": float("inf"),
            "cost": float("inf"),
            "prev": None
        }

    nodes[start]["dist"] = 0
    nodes[start]["toll"] = graph[start]["toll"]
    nodes[start]["cost"] = compute_cost(0, graph[start]["toll"], mode)

    queue = [start]
    visited = set()

    while queue:
        # Select node with lowest cost
        current = min(queue, key=lambda x: nodes[x]["cost"])
        queue.remove(current)

        if current == end:
            break

        visited.add(current)

        for neighbor in graph[current]["roads"]:
            if neighbor in visited:
                continue

            edge_dist = graph[current]["roads"][neighbor]
            new_dist = nodes[current]["dist"] + edge_dist
            new_toll = nodes[current]["toll"] + graph[neighbor]["toll"]
            new_cost = compute_cost(new_dist, new_toll, mode)

            if new_cost < nodes[neighbor]["cost"]:
                nodes[neighbor]["dist"] = new_dist
                nodes[neighbor]["toll"] = new_toll
                nodes[neighbor]["cost"] = new_cost
                nodes[neighbor]["prev"] = current
```

```
                if neighbor not in queue:
                    queue.append(neighbor)

    # Reconstruct path
    path = []
    node = end
    while node is not None:
        path.insert(0, node)
        node = nodes[node]["prev"]

    return {
        "path": path,
        "distance": nodes[end]["dist"],
        "toll": nodes[end]["toll"],
        "cost": nodes[end]["cost"]
    }


Map = {
    "A": {"toll": 0, "roads": {"B": 4, "C": 8}},
    "B": {"toll": 2, "roads": {"A": 4, "C": 2, "D": 5}},
    "C": {"toll": 3, "roads": {"A": 8, "B": 2, "D": 3, "E": 6}},
    "D": {"toll": 2, "roads": {"B": 5, "C": 3, "E": 2}},
    "E": {"toll": 5, "roads": {"C": 6, "D": 2}},
}

result = dijkstra(Map, "A", "E", "toll")
print(f"Path: {result['path']}")
```