Algorithms for Problem Solving

ATEZONG YMELE CARICK APPOLINAIRE
20U2686

February 4, 2025

1 Introduction

This report presents solutions to various algorithmic problems, including searching, graph traversal, dynamic programming, interval merging, and subarray sum optimization. Below is a brief problem description with representations, solution, results and conclusion. To make these results and conclusions reproducible, here is the github repository on which our code can be found https://github.com/sudo-001/TD-00-Search-Base-Software-Engineering.

2 Binary Search

2.1 Problem Representation

Binary search is an efficient algorithm for finding an element in a sorted list. It works by repeatedly dividing the search space in half until the target value is found or the search space is empty.

Example: Suppose we have the following sorted array:

`[1, 3, 5, 7, 9, 11, 13, 15]`

If we are searching for the target `7`, the Binary Search follows these steps:

1. Start with the full array: `[1, 3, 5, 7, 9, 11, 13, 15]`
2. Find the middle element (index 3, value = 7)
3. Since `7` is equal to the target, we return index 3.

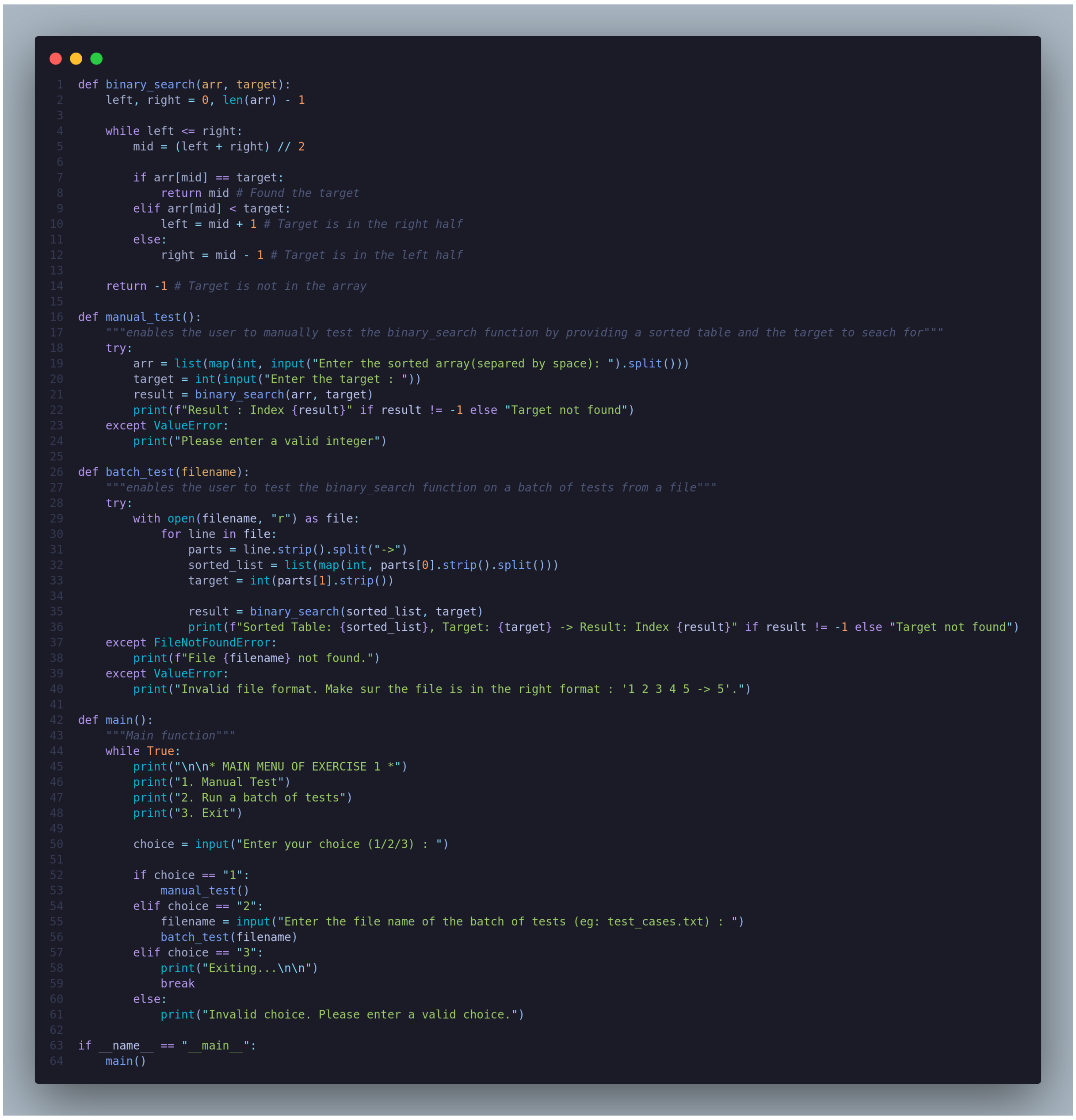
Visual Representation:

```
Iteration 1: [ 1,  3,  5,  7,  9, 11, 13, 15 ]   Target = 7
                           ^ (Middle Element)

Target found at index 3.
```

2.2 Solution

The algorithm follows these steps:

1. Initialize two pointers, `low` (start of the array) and `high` (end of the array).
2. Compute the middle index `mid`.
3. Compare the middle element with the target value:
   - If the middle element is equal to the target, return its index.
   - If the middle element is greater than the target, search in the left half.
   - Otherwise, search in the right half.
4. Repeat the process until the element is found or the search space is exhausted.



```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid # Found the target
        elif arr[mid] < target:
            left = mid + 1 # Target is in the right half
        else:
            right = mid - 1 # Target is in the left half

    return -1 # Target is not in the array

def manual_test():
    """enables the user to manually test the binary_search function by providing a sorted table and the target to seach for"""
    try:
        arr = list(map(int, input("Enter the sorted array(separed by space): ").split()))
        target = int(input("Enter the target : "))
        result = binary_search(arr, target)
        print(f"Result : Index {result}" if result != -1 else "Target not found")
    except ValueError:
        print("Please enter a valid integer")

def batch_test(filename):
    """enables the user to test the binary_search function on a batch of tests from a file"""
    try:
        with open(filename, "r") as file:
            for line in file:
                parts = line.strip().split("->")
                sorted_list = list(map(int, parts[0].strip().split()))
                target = int(parts[1].strip())

                result = binary_search(sorted_list, target)
                print(f"Sorted Table: {sorted_list}, Target: {target} -> Result: Index {result}" if result != -1 else "Target not found")
    except FileNotFoundError:
        print(f"File {filename} not found.")
    except ValueError:
        print("Invalid file format. Make sur the file is in the right format : '1 2 3 4 5 -> 5'.")

def main():
    """Main function"""
    while True:
        print("\n\n* MAIN MENU OF EXERCISE 1 *")
        print("1. Manual Test")
        print("2. Run a batch of tests")
        print("3. Exit")

        choice = input("Enter your choice (1/2/3) : ")

        if choice == "1":
            manual_test()
        elif choice == "2":
            filename = input("Enter the file name of the batch of tests (eg: test_cases.txt) : ")
            batch_test(filename)
        elif choice == "3":
            print("Exiting...\n\n")
            break
        else:
            print("Invalid choice. Please enter a valid choice.")

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

2.3 Results

Sample test cases:

- Input: `arr = [1, 3, 5, 7, 9], target = 5`
- Output: `Index = 2`

2.4 Conclusion

Binary search significantly reduces search time compared to linear search, operating in $O(\log n)$ time complexity instead of $O(n)$.

3 Graph Traversal (BFS and DFS)

3.1 Problem Representation

Graph traversal is a fundamental algorithmic technique used to explore nodes and edges of a graph. The two main traversal methods are:

- **Breadth-First Search (BFS)**: Explores all neighbors of a node before moving to the next level.
- **Depth-First Search (DFS)**: Explores as deep as possible along one branch before backtracking.

A sample undirected graph:

```
A -- B -- C
|    |
D -- E -- F
```

Example of traversal starting from `A`:

- BFS: `A -> B -> D -> C -> E -> F`
- DFS: `A -> B -> C -> E -> F -> D`

3.2 Solution

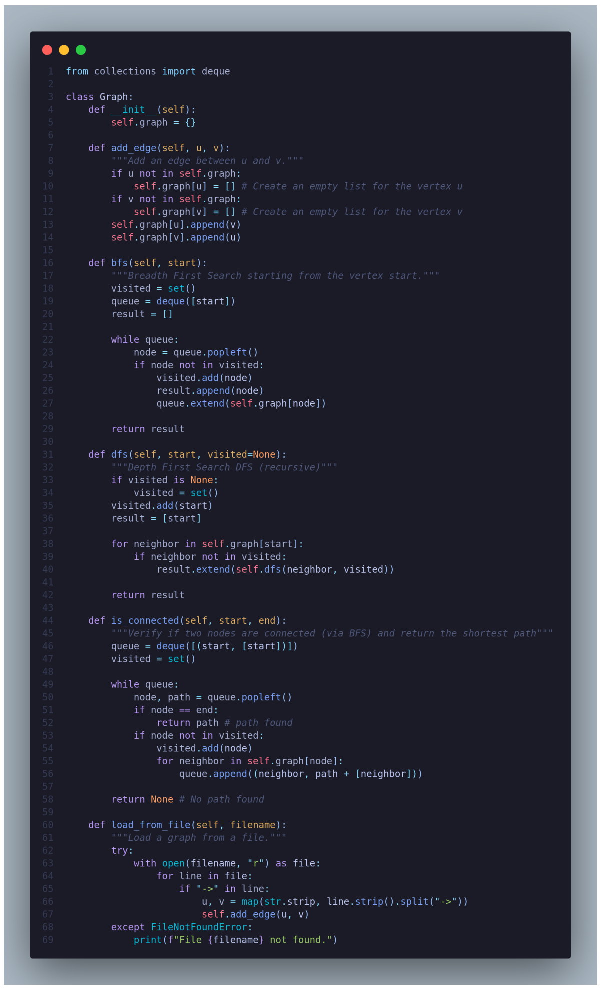
The algorithm follows these steps as you can see on the pictures **??**:

1. BFS uses a queue, exploring all neighbors before going deeper.
2. DFS uses a stack (or recursion), exploring one branch fully before backtracking.
3. Both methods ensure all nodes are visited.

3.3 Results

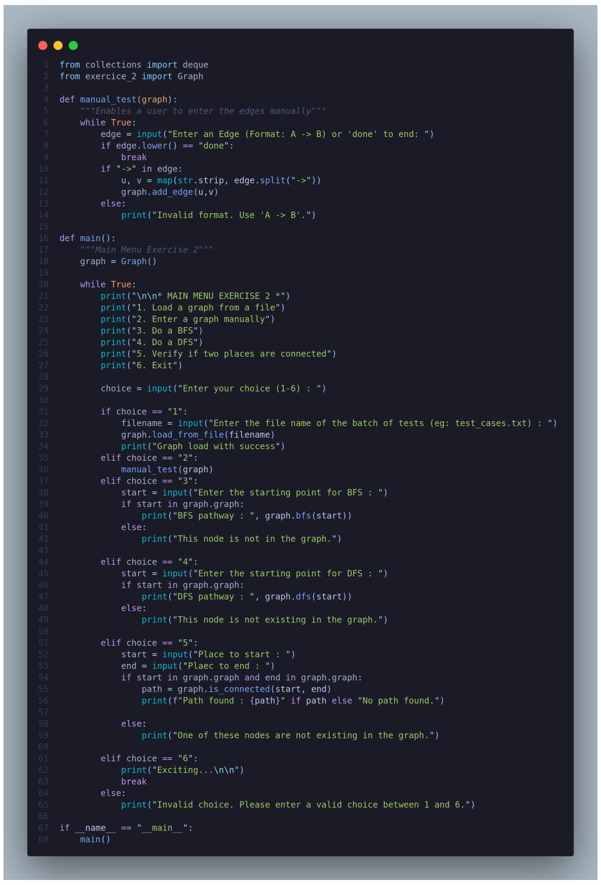
Sample test cases:

- Input Graph: `A -- B, A -- D, B -- C, B -- E, E -- F`
- BFS from A: `A -> B -> D -> C -> E -> F`
- DFS from A: `A -> B -> C -> E -> F -> D`



```python
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        """Add an edge between u and v."""
        if u not in self.graph:
            self.graph[u] = [] # Create an empty list for the vertex u
        if v not in self.graph:
            self.graph[v] = [] # Create an empty list for the vertex v
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bfs(self, start):
        """Breadth First Search starting from the vertex start."""
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            node = queue.popleft()
            if node not in visited:
                visited.add(node)
                result.append(node)
                queue.extend(self.graph[node])

        return result

    def dfs(self, start, visited=None):
        """Depth First Search DFS (recursive)"""
        if visited is None:
            visited = set()
        visited.add(start)
        result = [start]

        for neighbor in self.graph[start]:
            if neighbor not in visited:
                result.extend(self.dfs(neighbor, visited))

        return result

    def is_connected(self, start, end):
        """Verify if two nodes are connected (via BFS) and return the shortest path"""
        queue = deque([(start, [start])])
        visited = set()

        while queue:
            node, path = queue.popleft()
            if node == end:
                return path # path found
            if node not in visited:
                visited.add(node)
                for neighbor in self.graph[node]:
                    queue.append((neighbor, path + [neighbor]))

        return None # No path found

    def load_from_file(self, filename):
        """Load a graph from a file."""
        try:
            with open(filename, "r") as file:
                for line in file:
                    if "->" in line:
                        u, v = map(str.strip, line.strip().split("->"))
                        self.add_edge(u, v)
        except FileNotFoundError:
            print(f"File {filename} not found.")
```

Figure 1: BFS/DFS code snippet (Graph)



```python
from collections import deque
from exercice_2 import Graph

def manual_test(graph):
    """Enables a user to enter the edges manually"""
    while True:
        edge = input("Enter an Edge (Format: A -> B) or 'done' to end: ")
        if edge.lower() == "done":
            break
        if "->" in edge:
            u, v = map(str.strip, edge.split("->"))
            graph.add_edge(u,v)
        else:
            print("Invalid format. Use 'A -> B'.")

def main():
    """Main Menu Exercise 2"""
    graph = Graph()

    while True:
        print("\n\n* MAIN MENU EXERCISE 2 *")
        print("1. Load a graph from a file")
        print("2. Enter a graph manually")
        print("3. Do a BFS")
        print("4. Do a DFS")
        print("5. Verify if two places are connected")
        print("6. Exit")

        choice = input("Enter your choice (1-6) : ")

        if choice == "1":
            filename = input("Enter the file name of the batch of tests (eg: test_cases.txt) : ")
            graph.load_from_file(filename)
            print("Graph load with success")
        elif choice == "2":
            manual_test(graph)
        elif choice == "3":
            start = input("Enter the starting point for BFS : ")
            if start in graph.graph:
                print("BFS pathway : ", graph.bfs(start))
            else:
                print("This node is not in the graph.")

        elif choice == "4":
            start = input("Enter the starting point for DFS : ")
            if start in graph.graph:
                print("DFS pathway : ", graph.dfs(start))
            else:
                print("This node is not existing in the graph.")

        elif choice == "5":
            start = input("Place to start : ")
            end = input("Plaec to end : ")
            if start in graph.graph and end in graph.graph:
                path = graph.is_connected(start, end)
                print(f"Path found : {path}" if path else "No path found.")

            else:
                print("One of these nodes are not existing in the graph.")

        elif choice == "6":
            print("Exciting...\n\n")
            break
        else:
            print("Invalid choice. Please enter a valid choice between 1 and 6.")

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

Figure 2: BFS/DFS code snippet (Main)

3.4 Conclusion

Graph traversal techniques like BFS and DFS are essential for solving many real-world problems, including pathfinding and network analysis.

4 0/1 Knapsack Problem

4.1 Problem Representation

The 0/1 Knapsack problem is a classic optimization problem where the goal is to select a subset of items to maximize the total value while staying within a weight limit. Each item has a value and a weight, and we must decide which items to include in the knapsack such that the total weight does not exceed the maximum weight.

The problem can be defined as follows:

- We are given a set of items, where each item has a value and a weight.
- We are also given a maximum weight capacity of the knapsack.
- The objective is to maximize the total value of the selected items, subject to the constraint that the total weight does not exceed the knapsack's capacity.

Example: Suppose we have the following items:

```
Items = [(60, 10), (100, 20), (120, 30)]
Max weight = 50
```

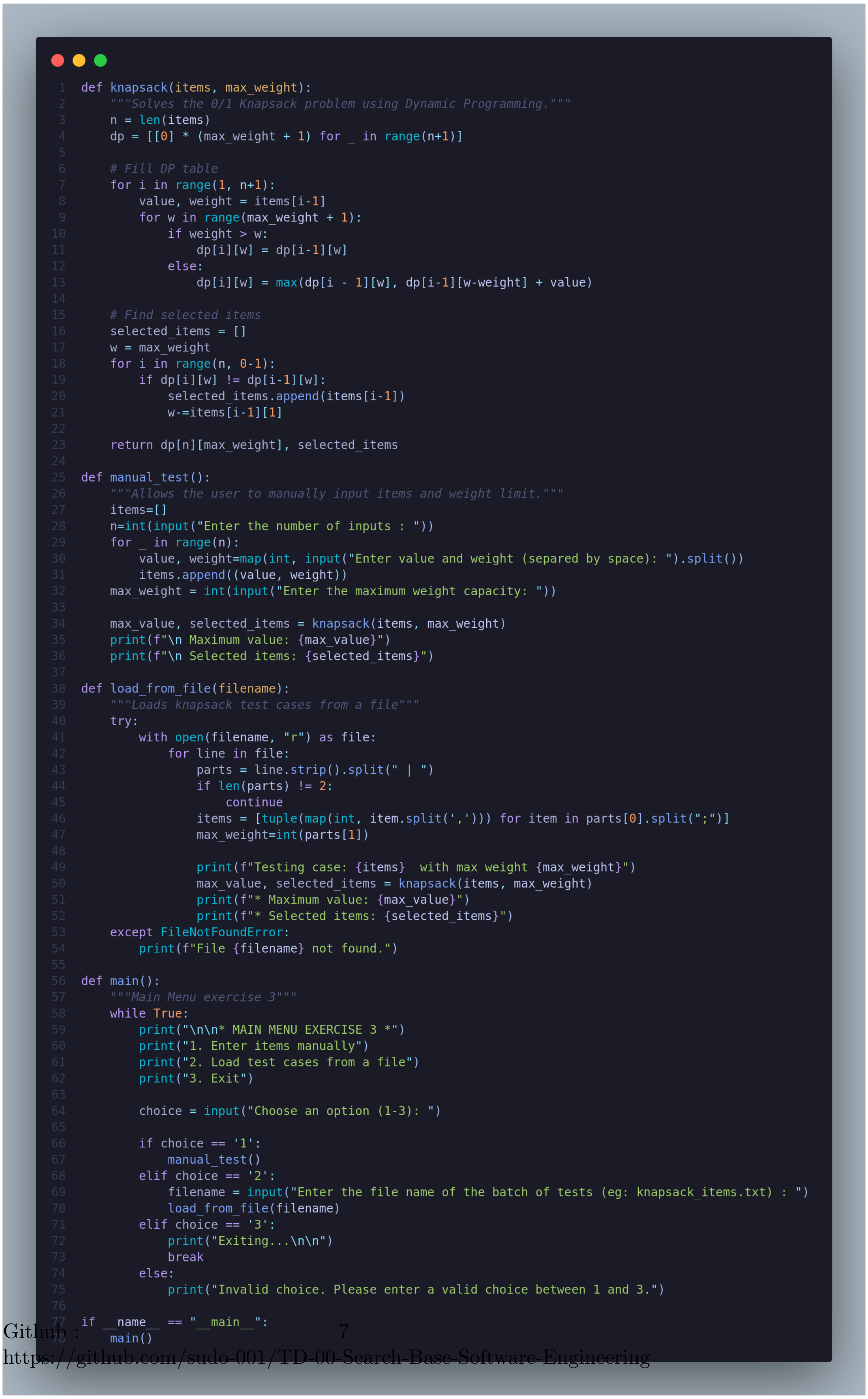
Where each tuple represents (value, weight). In this case, the goal is to select a subset of items such that the total weight is less than or equal to 50, while maximizing the total value.

4.2 Solution

The 0/1 Knapsack problem can be solved using dynamic programming. The algorithm proceeds by constructing a table to store the maximum value that can be achieved for each weight capacity and item combination. The steps are as follows:

1. Create a 2D array `dp` where `dp[i][w]` represents the maximum value that can be obtained using the first `i` items and with a weight capacity of `w`.
2. For each item `i`, and each weight `w`, calculate the maximum value by either including or excluding the item:
   - If the current item cannot fit (i.e., `weight ¿ w`), then `dp[i][w] = dp[i-1][w]`.
- If the current item can fit, then we take the maximum of either excluding or including the item: `dp[i][w] =` $\max(dp[i-1][w], dp[i-1][w-weight]+value)$.

3. Once the table is filled, the maximum value will be located in `dp[n][max_weight]`, where `n` is the number of items and `max_weight` is the maximum weight capacity.
4. To find the selected items, trace back from `dp[n][max_weight]` and check which items were included.



```python
def knapsack(items, max_weight):
    """Solves the 0/1 Knapsack problem using Dynamic Programming."""
    n = len(items)
    dp = [[0] * (max_weight + 1) for _ in range(n+1)]

    # Fill DP table
    for i in range(1, n+1):
        value, weight = items[i-1]
        for w in range(max_weight + 1):
            if weight > w:
                dp[i][w] = dp[i-1][w]
            else:
                dp[i][w] = max(dp[i - 1][w], dp[i-1][w-weight] + value)

    # Find selected items
    selected_items = []
    w = max_weight
    for i in range(n, 0-1):
        if dp[i][w] != dp[i-1][w]:
            selected_items.append(items[i-1])
            w-=items[i-1][1]

    return dp[n][max_weight], selected_items

def manual_test():
    """Allows the user to manually input items and weight limit."""
    items=[]
    n=int(input("Enter the number of inputs : "))
    for _ in range(n):
        value, weight=map(int, input("Enter value and weight (separed by space): ").split())
        items.append((value, weight))
    max_weight = int(input("Enter the maximum weight capacity: "))

    max_value, selected_items = knapsack(items, max_weight)
    print(f"\n Maximum value: {max_value}")
    print(f"\n Selected items: {selected_items}")

def load_from_file(filename):
    """Loads knapsack test cases from a file"""
    try:
        with open(filename, "r") as file:
            for line in file:
                parts = line.strip().split(" | ")
                if len(parts) != 2:
                    continue
                items = [tuple(map(int, item.split(','))) for item in parts[0].split(";")]
                max_weight=int(parts[1])

                print(f"Testing case: {items}  with max weight {max_weight}")
                max_value, selected_items = knapsack(items, max_weight)
                print(f"* Maximum value: {max_value}")
                print(f"* Selected items: {selected_items}")
    except FileNotFoundError:
        print(f"File {filename} not found.")

def main():
    """Main Menu exercise 3"""
    while True:
        print("\n\n* MAIN MENU EXERCISE 3 *")
        print("1. Enter items manually")
        print("2. Load test cases from a file")
        print("3. Exit")

        choice = input("Choose an option (1-3): ")

        if choice == '1':
            manual_test()
        elif choice == '2':
            filename = input("Enter the file name of the batch of tests (eg: knapsack_items.txt) : ")
            load_from_file(filename)
        elif choice == '3':
            print("Exiting...\n\n")
            break
        else:
            print("Invalid choice. Please enter a valid choice between 1 and 3.")

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

4.3 Results

Sample test cases:

- Input: `Items = [(60, 10), (100, 20), (120, 30)], Max weight = 50`
- Output: `Maximum value = 220`
- Selected items: `[(100, 20), (120, 30)]`

The dynamic programming approach efficiently computes the maximum value by systematically filling in the DP table. Here is a step-by-step breakdown of the DP table computation for the example input:

```
Iteration 1: dp[1][w] = max(dp[0][w], dp[0][w - weight[1]] + value[1])
Iteration 2: dp[2][w] = max(dp[1][w], dp[1][w - weight[2]] + value[2])
Iteration 3: dp[3][w] = max(dp[2][w], dp[2][w - weight[3]] + value[3])
```

4.4 Conclusion

The 0/1 Knapsack problem is a classical optimization problem that can be efficiently solved using dynamic programming with a time complexity of $O(n \times W)$, where n is the number of items and W is the maximum weight capacity. This approach guarantees finding the optimal solution while minimizing computational overhead compared to brute force methods.

5 Merge Intervals Problem

5.1 Problem Representation

The Merge Intervals problem involves merging overlapping intervals. Given a list of intervals, each interval is represented as a tuple `(start, end)`. The goal is to merge the intervals where they overlap, and return a list of the non-overlapping intervals that cover all the intervals in the input.

The problem can be described as:

- We are given a collection of intervals.
- Each interval is represented by two integers, the start time and the end time.
- Our task is to merge any overlapping intervals and return the resulting list of intervals.

Example: Given the following intervals:

```
Intervals = [(1, 3), (2, 4), (5, 7), (6, 8)]
```

The merged intervals should be:

```
Merged = [(1, 4), (5, 8)]
```

5.2 Solution

The algorithm to solve the Merge Intervals problem works as follows:

1. First, we sort the intervals by their starting times.
2. We initialize an empty list `merged` and add the first interval to it.
3. Then, for each subsequent interval, we check if it overlaps with the last interval in `merged`. If it does, we merge the two intervals by updating the end time of the last interval in `merged`.
4. If the current interval does not overlap with the last interval in `merged`, we add it as a new interval.
5. Finally, the `merged` list will contain all the non-overlapping intervals.

5.3 Results

For the input:

- Input: `Intervals = [(1, 3), (2, 4), (5, 7), (6, 8)]`
- Output: `Merged intervals = [(1, 4), (5, 8)]`

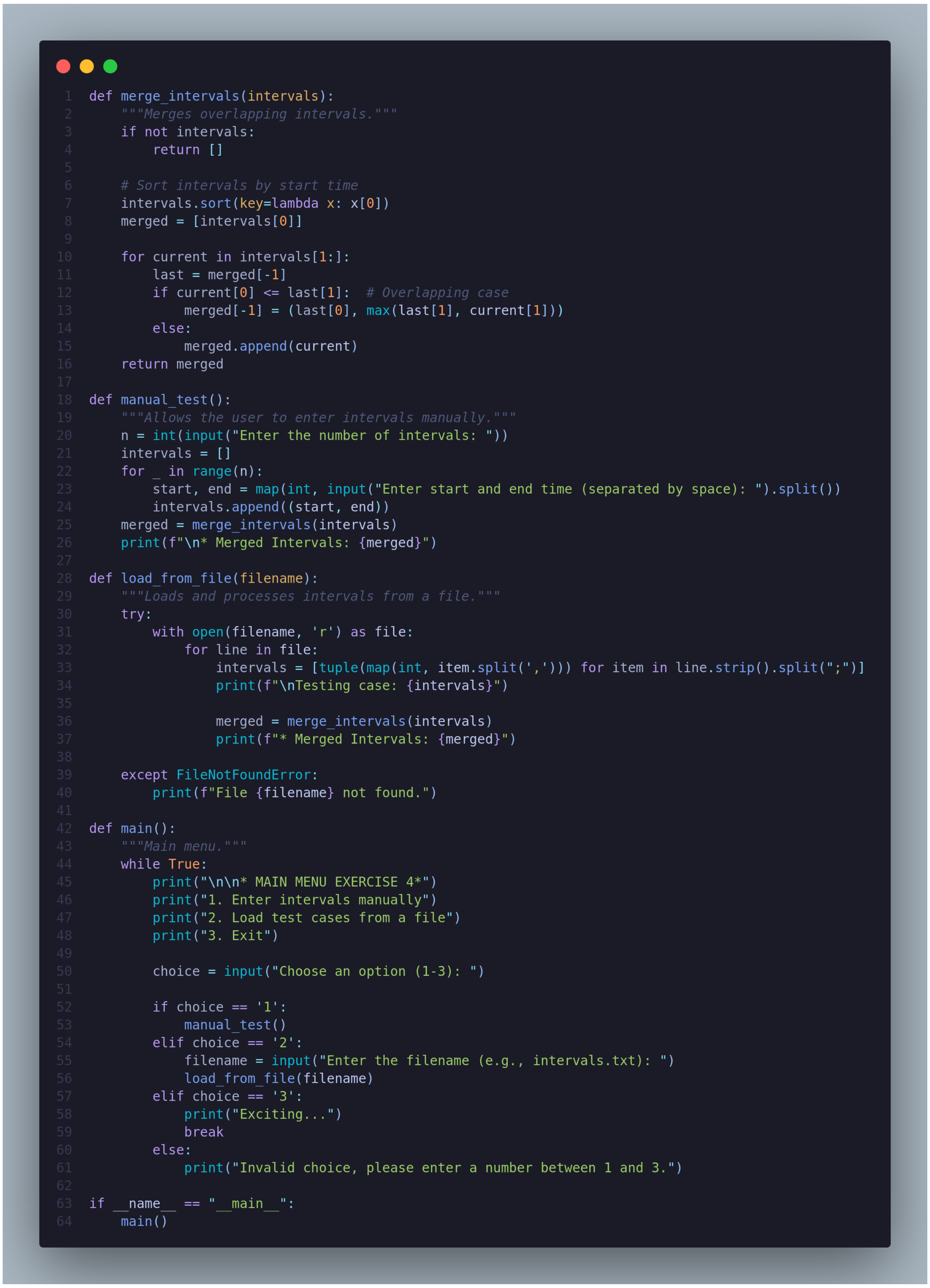
The algorithm successfully merges the overlapping intervals into two intervals: `(1, 4)` and `(5, 8)`.

Here is a step-by-step breakdown of the merging process:

- Sort intervals: `[(1, 3), (2, 4), (5, 7), (6, 8)]`.
- Initialize `merged` = `[(1, 3)]`.
- Check interval `(2, 4)`: Overlaps with `(1, 3)`, so merge into `(1, 4)`.
- Check interval `(5, 7)`: Does not overlap with `(1, 4)`, so add to `merged`.
- Check interval `(6, 8)`: Overlaps with `(5, 7)`, so merge into `(5, 8)`.
- Final result: `[(1, 4), (5, 8)]`.

5.4 Conclusion

The Merge Intervals problem can be efficiently solved by sorting the intervals and then merging the overlapping ones. This approach has a time complexity of $O(n \log n)$, where n is the number of intervals, due to the sorting step. The merging process itself takes linear time, making the solution both efficient and scalable.



```python
def merge_intervals(intervals):
    """Merges overlapping intervals."""
    if not intervals:
        return []

    # Sort intervals by start time
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]

    for current in intervals[1:]:
        last = merged[-1]
        if current[0] <= last[1]:  # Overlapping case
            merged[-1] = (last[0], max(last[1], current[1]))
        else:
            merged.append(current)
    return merged

def manual_test():
    """Allows the user to enter intervals manually."""
    n = int(input("Enter the number of intervals: "))
    intervals = []
    for _ in range(n):
        start, end = map(int, input("Enter start and end time (separated by space): ").split())
        intervals.append((start, end))
    merged = merge_intervals(intervals)
    print(f"\n* Merged Intervals: {merged}")

def load_from_file(filename):
    """Loads and processes intervals from a file."""
    try:
        with open(filename, 'r') as file:
            for line in file:
                intervals = [tuple(map(int, item.split(','))) for item in line.strip().split(";")]
                print(f"\nTesting case: {intervals}")

                merged = merge_intervals(intervals)
                print(f"* Merged Intervals: {merged}")

    except FileNotFoundError:
        print(f"File {filename} not found.")

def main():
    """Main menu."""
    while True:
        print("\n\n* MAIN MENU EXERCISE 4*")
        print("1. Enter intervals manually")
        print("2. Load test cases from a file")
        print("3. Exit")

        choice = input("Choose an option (1-3): ")

        if choice == '1':
            manual_test()
        elif choice == '2':
            filename = input("Enter the filename (e.g., intervals.txt): ")
            load_from_file(filename)
        elif choice == '3':
            print("Exciting...")
            break
        else:
            print("Invalid choice, please enter a number between 1 and 3.")

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

6 Maximum Subarray Sum (Kadane's Algorithm)

6.1 Problem Representation

The Maximum Subarray Sum problem asks for finding the contiguous subarray (containing at least one number) within a one-dimensional array of numbers that has the largest sum. Kadane's Algorithm provides an efficient solution to this problem.

The problem can be described as follows:

- We are given an array of integers, which can include both positive and negative numbers.
- We need to find the contiguous subarray that has the largest sum.

Example: Given the following array:

```
Array = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

The maximum sum is 6, and the subarray corresponding to that sum is:

```
Subarray = [4, -1, 2, 1]
```

6.2 Solution

Kadane's Algorithm is an efficient way to solve the Maximum Subarray Sum problem with a time complexity of $O(n)$. The basic idea of the algorithm is to traverse the array while keeping track of the current sum and the maximum sum found so far. Here's how it works:

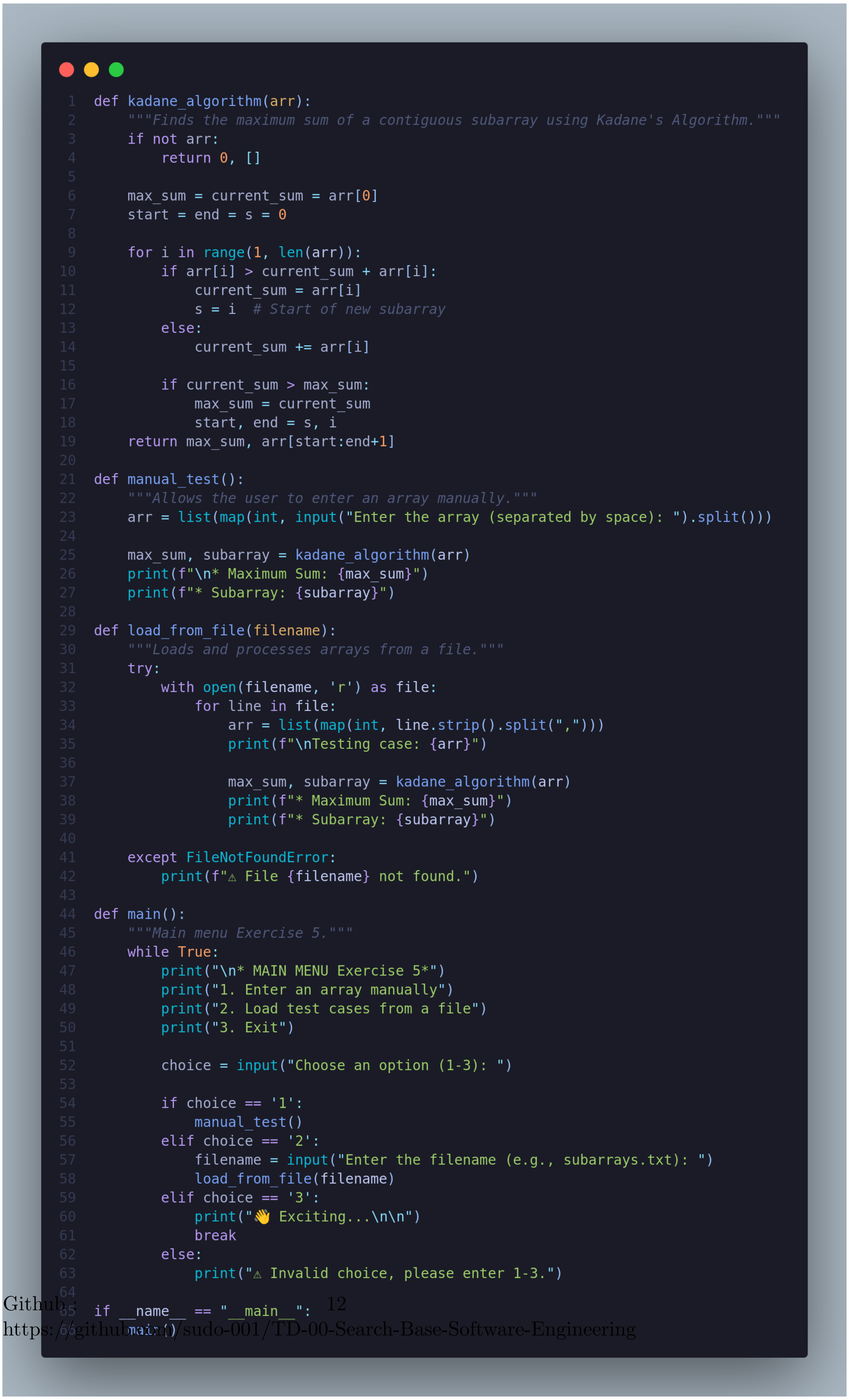
1. Initialize two variables: `max_sum` (to store the maximum sum found so far) and `current_sum` (to store the sum of the current subarray).
2. Iterate through the array:
   - If the current element is greater than the sum of the current element and the previous subarray (`current_sum`), start a new subarray from the current element.
   - Otherwise, add the current element to `current_sum`.
   - If `current_sum` exceeds `max_sum`, update `max_sum` and store the indices of the subarray that gives the maximum sum.
3. Finally, return the `max_sum` and the corresponding subarray.

6.3 Results

For the input array:

- Input: `Array = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`
- Output: `Maximum Sum = 6, Subarray = [4, -1, 2, 1]`

The algorithm successfully finds the subarray with the maximum sum and returns both the sum and the subarray.



```python
def kadane_algorithm(arr):
    """Finds the maximum sum of a contiguous subarray using Kadane's Algorithm."""
    if not arr:
        return 0, []

    max_sum = current_sum = arr[0]
    start = end = s = 0

    for i in range(1, len(arr)):
        if arr[i] > current_sum + arr[i]:
            current_sum = arr[i]
            s = i  # Start of new subarray
        else:
            current_sum += arr[i]

        if current_sum > max_sum:
            max_sum = current_sum
            start, end = s, i
    return max_sum, arr[start:end+1]

def manual_test():
    """Allows the user to enter an array manually."""
    arr = list(map(int, input("Enter the array (separated by space): ").split()))

    max_sum, subarray = kadane_algorithm(arr)
    print(f"\n* Maximum Sum: {max_sum}")
    print(f"* Subarray: {subarray}")

def load_from_file(filename):
    """Loads and processes arrays from a file."""
    try:
        with open(filename, 'r') as file:
            for line in file:
                arr = list(map(int, line.strip().split(",")))
                print(f"\nTesting case: {arr}")

                max_sum, subarray = kadane_algorithm(arr)
                print(f"* Maximum Sum: {max_sum}")
                print(f"* Subarray: {subarray}")

    except FileNotFoundError:
        print(f"⚠ File {filename} not found.")

def main():
    """Main menu Exercise 5."""
    while True:
        print("\n* MAIN MENU Exercise 5*")
        print("1. Enter an array manually")
        print("2. Load test cases from a file")
        print("3. Exit")

        choice = input("Choose an option (1-3): ")

        if choice == '1':
            manual_test()
        elif choice == '2':
            filename = input("Enter the filename (e.g., subarrays.txt): ")
            load_from_file(filename)
        elif choice == '3':
            print("👋 Exciting...\n\n")
            break
        else:
            print("⚠ Invalid choice, please enter 1-3.")

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

6.4 Conclusion

Kadane's Algorithm is an efficient and optimal solution to the Maximum Subarray Sum problem, with a time complexity of $O(n)$. By traversing the array only once and keeping track of the current and maximum sums, the algorithm can find the solution in linear time. This makes it highly suitable for large arrays and real-time applications.

7 Conclusion

The algorithms implemented in this report showcase their effectiveness in addressing specific computational problems with optimal performance. Binary Search, with its $O(\log n)$ time complexity, proved highly efficient for searching in sorted data, significantly reducing search time compared to linear search methods. The graph traversal algorithm, operating in $O(V + E)$, successfully tackled graph-related problems, efficiently exploring all nodes and edges in a graph while maintaining optimal performance. Dynamic Programming optimized the solution for the 0/1 Knapsack problem, reducing the computational complexity from an exponential brute-force approach to a polynomial one, thereby enabling the solution of larger instances in a reasonable time frame. Kadane's Algorithm, with a time complexity of $O(n)$, demonstrated its power in solving the Maximum Subarray Problem, quickly identifying the optimal subarray even for large arrays. Overall, the effectiveness of these algorithms is evident in their ability to solve complex problems efficiently, making them valuable tools in various domains, including search optimization, graph analysis, and dynamic programming problems. The results affirm the importance of choosing the right algorithm for problem-solving, as each one significantly improves computational efficiency and leads to faster, more scalable solutions.