INFO: 5059 SEARCH BASE SOFTWARE ENGINEERING		
Students' Names	UNIVERSITÉ DE YAOUNDÉ I Sapientia - Collativa - Cognitio Cameroun	Deadline: Undefined
FOTSO WAFO Omer		**January 28, 2025**
		Ac. Year: 2024 - 2025

TD 00: Advanced Algorithms and Data Structures

Exercise 1: Binary Search

The problem asks us to search for a target value in a sorted list of integers. Given that the list is sorted, Binary Search divides the list in half with each step, focusing the search on the relevant half, reducing the search space exponentially. Consider the following sorted list:

$$\text{List} = [1, 3, 5, 7, 9, 11, 13, 15, 17]$$

We are tasked with finding the index of the target value, say 9, using Binary Search (**Output:** 4).

Analyze the time complexity of the binary search and compare it with a linear search.

- At the beginning, the search range spans the entire sorted list, i.e., indices:

$$[0, n-1]$$

- The search space reduces from:

$$n = \frac{n}{2^0}, \quad \frac{n}{2^1}, \quad \frac{n}{2^2}, \quad \ldots, \quad \frac{n}{2^k} = 1$$

until the size of the search space becomes 1.

The number of iterations required is equal to the number of times n can be divided by 2:

$$k = \log_2(n)$$

- Time Complexity Expression:

$$T(n) = c_1 \cdot \log_2(n) + c_2$$

As n grows large, the dominant term is $\log_2(n)$. Thus, we can conclude:

$$T(n) \sim O(\log_2(n))$$

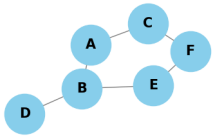
In contrast, Linear Search has a time complexity of $O(n)$, since it iterates through the entire list element by element. This makes Binary Search much faster, especially for large lists.

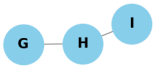
- Python Implementation:GitHub repository

Exercise 2: Graph Traversal (BFS and DFS)

Breadth-First Search (BFS) explores the graph level by level, starting from a source node, and uses a queue for traversal. It is particularly useful for finding the shortest path in an unweighted graph. **Depth-First Search (DFS)** explores as deeply as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack.

0.1 Test the traversal algorithms with a graph that has multiple connected components.





Test	Result
BFS Traversal	`['A', 'B', 'C', 'D', 'E', 'F']`
DFS Recursive Traversal	`['A', 'B', 'D', 'E', 'F', 'C']`
Connectivity Check: Are 'A' and 'F' connected?	`True`, Path: `['A', 'C', 'F']`
Connectivity Check: Are 'A' and 'G' connected?	`False`, Path: `[]`

Table 2: Results of Graph Traversal Tests

0.2 Time Complexity Analysis of BFS and DFS

Let V be the number of vertices and E the number of edges in the graph.

Breadth-First Search (BFS)

- **Initialization:** Initializing the queue takes $O(1)$.
- **Vertex Visits:** Each vertex is dequeued and processed exactly once, resulting in a time complexity of $O(V)$.
- **Edge Exploration:** Each edge is traversed once, resulting in a time complexity of $O(E)$.

Total Time Complexity:

$$T(V, E) = O(V) + O(E) = O(V + E)$$

Depth-First Search (DFS)

- **Vertex Visits:** Each vertex is visited once, resulting in $O(V)$.
- **Edge Exploration:** Each edge is traversed once, resulting in $O(E)$.
- **Recursive Calls:** The cost of recursive calls is negligible for large graphs.

Total Time Complexity:

$$T(V, E) = O(V) + O(E) = O(V + E)$$

- Python Implementation:GitHub repository

Exercise 3: Dynamic Programming (Knapsack Problem)

0.3 Test cases:

- **Items:** $[(60, 10), (100, 20), (120, 30)]$; Maximum Weight: 50
  Output:
  Maximum Value: 220
  Selected Items: $[(100, 20), (120, 30)]$
- **Items:** $[(20, 5), (30, 10), (50, 15), (70, 20)]$; Maximum Weight: 25
  Output:
  Maximum Value: 90
  Selected Items: $[(20, 5), (70, 20)]$

0.4 Time Complexity of the 0/1 Knapsack Algorithm (Summary)

The 0/1 Knapsack algorithm fills a 2D table $dp[i][w]$, where i is the number of items (0 to n) and w is the weight limit of the knapsack (0 to W).

- **Table Construction:**
  - Two nested loops iterate over n items and W weight limits.
  - Each cell computation is done in constant time $O(1)$:

    $dp[i][w] = \max(dp[i-1][w], dp[i-1][w-\text{weight}]+\text{value})$
  - Total iterations: $n \cdot W$, giving a time complexity:

    $$T_{\text{fill}} = O(n \cdot W)$$
- **Backtracking:** After constructing the table, backtracking involves n iterations to determine the items included in the optimal solution. Its time complexity is:

  $$T_{\text{backtrack}} = O(n)$$

Overall Time Complexity: Combining the table construction and backtracking:

$$T_{\text{total}} = O(n \cdot W) + O(n)$$

Since $O(n \cdot W)$ dominates for large W, the final time complexity is:

$$\boxed{T_{\text{total}} = O(n \cdot W)}$$

- Python Implementation:GitHub repository

Exercise 4: Merge Intervals

0.5 Test cases:

- **Intervals with overlaps:** $[(1, 3), (2, 6), (8, 10), (15, 18)]$.
  Output: $[(1, 6), (8, 10), (15, 18)]$
- **Intervals with no overlaps:** $[(1, 2), (3, 4), (5, 6)]$.
  Output: $[(1, 2), (3, 4), (5, 6)]$
- **Fully overlapping intervals:** $[(1, 10), (2, 6), (3, 8)]$.
  Output: $[(1, 10)]$

0.6 Analyze the Time Complexity of Merge Intervals

- **Sorting:**

  $$O(n \log n)$$
- **Merging:**

  $$O(n)$$

Total Time Complexity:

$$T(n) = O(n \log n) + O(n) = O(n \log n)$$

Exercise 5: Maximum Subarray Sum (Kadane's Algorithm)

The goal is to identify the contiguous subarray of a given array with the maximum sum. For example:

6 (Maximum Sum)

-2	1	-3	4	-1	2	1	-5	4

0.7 Solutiion

Kadane's Algorithm is a dynamic programming-based solution to efficiently find the maximum sum of a contiguous subarray. The algorithm works by iterating through the array, maintaining the current subarray sum, and resetting it to zero whenever it becomes negative.

0.8 Test cases:

- **Input: Mixed positive and negative numbers** $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$.
  Output: 6
- **Input: All positive numbers** $[1, 2, 3, 4]$.
  Output: 10
- **Input: All negative numbers** $[-1, -2, -3, -4]$.
  Output: -1

0.9 Compare with Brute-Force Approach:

Approach	Time Complexity	Space Complexit
Kadane's Algorithm	$O(n)$	$O(1)$
.Brute-Force Approach	$O(n^2)$	$O(1)$

- Python Implementation:GitHub repository