Advanced Algorithms and Data Structures Report

Exercise 1: Binary Search

Problem Representation::

Entrée : nums = [1, 3, 5, 7, 9, 11, 15], target = 6 → Sortie : -1

Solution

Given a sorted array of integers, the goal is to find the index of a target element using a more efficient method than linear search. The Binary Search algorithm works by repeatedly dividing the search interval in half. If the value of the target is less than the item in the middle, the search continues in the left half, otherwise in the right half.

Algorithm: Binary Search operates in the following steps:

1. Initialize two pointers, `left` and `right`, to the beginning and end of the array.
2. Find the middle element using the formula:
   `mid = (left + right) // 2`.
3. If the middle element is equal to the target, return the index.
4. If the middle element is less than the target, set `left` to `mid + 1`.
5. If the middle element is greater than the target, set `right` to `mid - 1`.
6. Repeat the process until the target is found or the search space is exhausted.

Code Snippet:

```
def binary_search(nums, target):
left, right = 0, len(nums) - 1
while left <= right:
mid = (left + right) // 2
if nums[mid] == target:
return mid
elif nums[mid] < target:
left = mid + 1
else:
right = mid - 1
return -1
```

Exercise 2 : Graph Traversal

Problem Representation:

Exemple d'entrée :

Graph avec les connexions suivantes :
1 → 2
1 → 3

```
2 → 4
2 → 5
3 → 6
3 → 7
```

```
Exemple de sortie :
```

```
BFS : [1, 2, 3, 4, 5, 6, 7]
DFS : [1, 2, 4, 5, 3, 6, 7]
```

Solution

Breadth-First Search (BFS): BFS explores the graph level by level, starting from the source node. It uses a queue to store the nodes and visits each node at the current level before moving to the next.

Depth-First Search (DFS): DFS explores the graph by going as deep as possible from the starting node before backtracking. It can be implemented either recursively or using a stack. Here, we implement it recursively.

Code snippet

```
def bfs(self, 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.get(node, []))
return result
```

```
def dfs(self, start, visited=None):
 if visited is None:
    visited = set()
 visited.add(start)
 result = [start]
 for neighbor in self.graph.get(start, []):
    if neighbor not in visited:
       result.extend(self.dfs(neighbor, visited))
 return result
```

Conclusion

Effectiveness of the Algorithms:

Each of the algorithms described—Merge Sort, Binary Search, Merge Intervals, and Kadane's Algorithm—provides an efficient and optimal solution for the problems at hand.

1. **Binary Search** significantly reduces the search time to O(logn) compared to linear search.
2. **Graph Traversal (BFS et DFS)**
3. **Merge Intervals** efficiently merges overlapping intervals with a time complexity of O(nlogn) due to sorting.
4. **Kadane's Algorithm** solves the maximum subarray sum problem in linear time O(n), making it highly efficient for large datasets.**Repository Link:**

You can access the code and full report on my GitHub repository here:
https://github.com/ousmanelagrange/td-advanced-algorithms