Algorithms for Problem Solving

Exercise 1: Binary Search

Problem Representation

Binary search efficiently locates elements in sorted arrays:

```
Array: [1, 2, 3, 4, 5, 6, 7, 8, 9]  Target: 7
 Index: 0  1  2  3  4  5  6  7  8

Step 1: Mid = 4  [1, 2, 3, 4, |5|, 6, 7, 8, 9]    5 < 7, go right
Step 2: Mid = 6  [6, |7|, 8, 9]                    Found at index 6!
```

Solution

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

Results

Input: `[1, 2, 3, 4, 5]`, Target: `3` Output: `2` (index of target)

Conclusion

Achieves O(log n) time complexity, significantly faster than linear search for large datasets.

Exercise 2: Graph Traversal

Problem Representation

Graph structure representing connected nodes:

```
0 --- 1
|     |
2 --- 3
```

Solution

```
def bfs(self, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    result = []
    while queue:
        vertex = queue.popleft()
        result.append(vertex)
        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return result
```

Results

Input: Graph with edges `[(0,1), (0,2), (1,2), (2,3)]` Output: BFS order `[0, 1, 2, 3]`

Conclusion

BFS ensures shortest paths in unweighted graphs, while DFS explores graph properties, both with O(V + E) complexity.

Exercise 3: Knapsack Problem

Problem Representation

Optimize value selection under weight constraints:

```
Items:
1. ($60, 10kg)
2. ($100, 20kg)
3. ($120, 30kg)
Capacity: 50kg
```

Solution

```
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]
```

Results

Input: Values=`[60,100,120]`, Weights=`[10,20,30]`, Capacity=`50` Output: Maximum value=`220`

Conclusion

Dynamic programming solution achieves optimal results with O(nW) complexity.

Exercise 4: Merge Intervals

Problem Representation

Merge overlapping time intervals:

```
[1,3] [2,6] [8,10]
1---3
  2-----6
        8--10
```

Solution

```
def merge_intervals(intervals):
    if not intervals:
        return []
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]
    for interval in intervals[1:]:
        if merged[-1][1] >= interval[0]:
            merged[-1] = (merged[-1][0], max(merged[-1][1], interval[1]))
        else:
```

```
            merged.append(interval)
    return merged
```

Results

Input: `[(1,3), (2,6), (8,10)]` Output: `[(1,6), (8,10)]`

Conclusion

Efficiently handles interval merging in O(n log n) time complexity.

Exercise 5: Maximum Subarray Sum

Problem Representation

Find contiguous subarray with largest sum:

```
[-2, 1, -3, 4, -1, 2, 1] → [4, -1, 2, 1]
```

Solution

```python
def kadane(arr):
    max_ending_here = max_so_far = arr[0]
    for i in range(1, len(arr)):
        max_ending_here = max(arr[i], max_ending_here + arr[i])
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far
```

Results

Input: `[-2, 1, -3, 4, -1, 2, 1]` Output: Maximum sum = `6`

Conclusion

Kadane's algorithm achieves optimal O(n) time complexity.

GitHub Repository

Complete implementation available at GitHub Repository URL.

```
.
|-- algorithms.py
|-- README.md
|-- technical_report.md
|-- test_cases/
`-- LICENSE
```