Algorithms for Problem Solving

Exercise 1 - Binary Search

Problem: Given a sorted array of integers, use binary search algorithm to find a target value.
Eg: Consider input array [4,9,17,-3,107,-2,9], search value: 9. Expected result: index 6
Solution: Binary Search works by repeatedly dividing the search space into two, comparing the searched value to the middle element to decide whether to continue search on the left or right half
Code snippet:

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

Result: Input(array: [4,9,17,-3,107,-2,9], target: -3). Result: index 3
Conclusion: Binary Search is faster(O(logn)) than Linear Search (complexity: O(n)).

Exercise 2 - Graph Traversal (BFS and DFS)

Problem: Find shortest path between two nodes, check connection between two nodes of an undirected graph using BFS and DFS.
Solution: - BFS (Breadth-First Search): Explores nodes level by level using a queue to find shortest path.
- DFS (Depth-First Search): Explores as deeply as possible before backtracking, using either a stack or recursion.

Code snippet for BFS:

```
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            for neighbor in graph[node]:
                queue.append(neighbor)
    return visited
```

Result: For input graph {'X': {'Y', 'Z'},'Y': {'X'},'Z': {'X'}}, Output BFS: X Y Z
Conclusion: BFS and DFS both have linear time complexity(O(V + E)) where V is the number of nodes, and E is the number of edges. Thus, both are optimal for graph traversal.

Exercise 3 - Knapsack Problem (Dynamic Programming)

Problem: Pack products in bag with weight limit to maximize value while minimizing weight.
Solution: Use DP by implementing a 2D table dp[i][w], where i represents the items and w represents the limit weight. Every cell of the 2D table contains the highest possible value for the first i items and the weight limit w.

Code snippet:

```
def knapsack(elements, W):
    for value, weight in elements:
        for w in range (W, weight-1, -1):
            dp[w]=max(dp[w-weight]+value, dp[w])
    max_val=dp[W]
    choosen_elmts=[]
    w=W
    for x in range(len(elements)-1,-1,-1):
        value, weight=elements[x]
        if dp[W]==dp[w-weight]+value and w>=weight:
            choosen_elmts.append(elements[x])
            w = w - weight
    choosen_elmts.reverse()
    return choosen_elmts, max_val
```

Result: For input items: [(5,70),(10,120),(7,70),(5,80),(13,100)], limit weight: 200, Output: selected=(7,70),(13,100)
Conclusion: Using DP is faster (O(n * W)) than the brute-force approach (complexity: O(2^n)).

Exercise 4 - Merge Intervals

Problem: Given a set of start/end time intervals, merge intervals that overlap and return the merged list.
Solution: Sort intervals by start time. Iterate on sorted list and merge if start time of current interval is less than or equal to the end time of the previous time interval.

Code snippet:
```
def merge_intervals(intervals):
    intervals.sort(key=start_time)
    merged_intervals=[]
    for interval in intervals:
        if not merged_intervals or merged_intervals[-1][1] < interval[0] :
            merged_intervals.append(interval)
        else:
            last_merged=merged_intervals[-1]
            merged_intervals[-1] = (last_merged[0], max(last_merged[1], interval[1]))
    return merged_intervals
```

Result: Given input intervals:[(5,7),(10,13),(9,11)], Result: merged intervals: [(5, 7), (9, 13)]
Conclusion: This approach is faster(O(nlogn)) than brute-force approach (O(n^2)).

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

Problem: Given an array of integers, find the maximum sum of contiguous elements of sub array in the array.
Solution: Kadane's algorithm is used to iterate through the array while computing the sum of current subarray. If current sum is negative, reset to zero, and store maximum computed sum.

Code:
```
def max_subarray_sum(numbers):
    if len(numbers) == 0:
        return 0
    max_sum = numbers[0]
    current_sum = numbers[0]
    for i in range(1, len(numbers)):
        current_sum = max(numbers[i], current_sum + numbers[i])
        max_sum = max(max_sum, current_sum)
    return max_sum
```

Result: Given input array**: [9,10,14,-4,3,-1,9],** output: max__sum_sub_array=40
Conclusion: Kadane's algorithm is more efficient(O(n)) than brute-force approach (complexity: O(n^2)).

Conclusion:
The aim of the exercises was to implement optimal problem solving algorithms:
These algorithms can be used for solving real world problems like data search, city connectivity, resource optimization, scheduling.