## W1D4

**QUESTION 1.**

**Write a Java program to solve the subset problem.**

**(a) T or F.**

**(b) One solution.**

**(c) All solutions.**

```
import java.util.*;
public class Question1_SubsetSum {
    // (a) True/False – existe subconjunto com soma k?
    public static boolean existsSubsetSum(int[] S, int k) {
        return backtrackExists(S, 0, k);
    }

    private static boolean backtrackExists(int[] S, int i, int rem) {
        if (rem == 0) return true;          // atingiu o alvo
        if (i == S.length || rem < 0) return false;
        // incluir ou não incluir o elemento atual
        return backtrackExists(S, i + 1, rem - S[i]) ||
               backtrackExists(S, i + 1, rem);
    }

    // (b) One solution
    public static List<Integer> findOneSubset(int[] S, int k) {
        List<Integer> path = new ArrayList<>();
        if (backtrackOne(S, 0, k, path)) return path;
        return null;
    }

    private static boolean backtrackOne(int[] S, int i, int rem, List<Integer> path)
{
        if (rem == 0) return true;
        if (i == S.length || rem < 0) return false;

        path.add(S[i]);
        if (backtrackOne(S, i + 1, rem - S[i], path)) return true;
        path.remove(path.size() - 1);

        return backtrackOne(S, i + 1, rem, path);
    }

    // (c) All solutions
    public static List<List<Integer>> findAllSubsets(int[] S, int k) {
        List<List<Integer>> ans = new ArrayList<>();
        backtrackAll(S, 0, k, new ArrayList<>(), ans);
        return ans;
    }

    private static void backtrackAll(int[] S, int i, int rem, List<Integer> path,
List<List<Integer>> ans) {
        if (rem == 0) {
            ans.add(new ArrayList<>(path));
```

```
            return;
        }
        if (i == S.length || rem < 0) return;

        path.add(S[i]);
        backtrackAll(S, i + 1, rem - S[i], path, ans);
        path.remove(path.size() - 1);

        backtrackAll(S, i + 1, rem, path, ans);
    }
}
```

**QUESTION 2.**

**Solve the subset problem where S = {3, 4, 7, 8} and k = 15.**

**(a) T or F.**

**(b) One solution.**

**(c) All solutions.**

```
import java.util.*;
public class Question2_SubsetSum {
    public static boolean existsSubsetSum(int[] S, int k) {
        if (k == 0) return true;
        return backtrackExists(S, 0, k);
    }

    private static boolean backtrackExists(int[] S, int i, int rem) {
        if (rem == 0) return true;
        if (i == S.length || rem < 0) return false;
        return backtrackExists(S, i + 1, rem - S[i]) ||
               backtrackExists(S, i + 1, rem);
    }

    public static List<Integer> findOneSubset(int[] S, int k) {
        List<Integer> path = new ArrayList<>();
        backtrackOne(S, 0, k, path);
        return path.isEmpty() ? null : path;
    }

    private static boolean backtrackOne(int[] S, int i, int rem, List<Integer> path)
{
        if (rem == 0) return true;
        if (i == S.length || rem < 0) return false;
        path.add(S[i]);
        if (backtrackOne(S, i + 1, rem - S[i], path)) return true;
        path.remove(path.size() - 1);
        return backtrackOne(S, i + 1, rem, path);
    }

    public static List<List<Integer>> findAllSubsets(int[] S, int k) {
        List<List<Integer>> ans = new ArrayList<>();
        backtrackAll(S, 0, k, new ArrayList<>(), ans);
```

```
        return ans;
    }

    private static void backtrackAll(int[] S, int i, int rem, List<Integer> path,
List<List<Integer>> ans) {
        if (rem == 0) { ans.add(new ArrayList<>(path)); return; }
        if (i == S.length || rem < 0) return;
        path.add(S[i]);
        backtrackAll(S, i + 1, rem - S[i], path, ans);
        path.remove(path.size() - 1);
        backtrackAll(S, i + 1, rem, path, ans);
    }
}
```

## QUESTION 3

**Solve the integer Knapsack problem given below:**

***The maximum allowable total weight in the knapsack is Wmax = 20.***

| Item | a | b | c | d | e |
|---|---|---|---|---|---|
| value | 25 | 12 | 24 | 16 | 28 |
| Weight | 5 | 6 | 8 | 2 | 7 |

**Integer (0/1) Knapsack Problem**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **a** | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 |
| **b** | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 | 25 | 37 | 37 | 37 | 37 | 37 | 37 | 37 | 37 | 37 | 37 |
| **c** | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 | 25 | 37 | 37 | 49 | 49 | 49 | 49 | 49 | 49 | 61 | 61 |
| **d** | 0 | 0 | 16 | 16 | 25 | 25 | 41 | 41 | 41 | 41 | 41 | 41 | 53 | 53 | 65 | 65 | 65 | 65 | 65 | 65 |
| **e** | 0 | 0 | 16 | 16 | 16 | 16 | 16 | 41 | 41 | 44 | 44 | 53 | 53 | 69 | 69 | 69 | 69 | 69 | 69 | 81 |

**QUESTION 4.** Solve the fractional Knapsack problem given below:

*The maximum allowable total weight in the knapsack is Wmax = 20.*

| Item | a | b | c | d | e |
|---|---|---|---|---|---|
| value | 25 | 12 | 24 | 16 | 28 |
| Weight | 5 | 6 | 8 | 2 | 7 |

| Item | value | weight | Value/weight |
|---|---|---|---|
| a | 25 | 5 | 5.0 |
| b | 12 | 6 | 2.0 |
| c | 24 | 8 | 3.0 |
| d | 16 | 2 | 8.0 |
| e | 28 | 7 | 4.0 |

**Step 2: Order items by value/weight**

| Item | Value |
|---|---|
| d | 8.0 |
| a | 5.0 |
| e | 4.0 |
| c | 3.0 |
| b | 2.0 |

**Step 3: greedily fill the backpack(knapsack)**

*Wmax = 20.*

| Item | value | weight | value/weight | Fraction | Remaining weight | Accumulated value |
|---|---|---|---|---|---|---|
| d | 16 | 2 | 5.0 | 1.00 | 20 - 2 = **18** | 16 |
| a | 25 | 5 | 2.0 | 1.00 | 18 - 5 = **13** | 16 + 25 = 41 |
| e | 28 | 7 | 3.0 | 1.00 | 13 - 7 = **6** | 41 + 28 = 69 |
| c | 24 | 8 | 8.0 | 0.75 | 6 / 8 = 0.75 ➔ 6 – 6 = **0** | 69 + (0.75 *24) = 87 |
| b | 12 | 6 | 4.0 | 0.00 | 0 | 87 |

Total weight: 2 + 5 + 7 + 6 = 20

Total value: 16 + 25 + 28 + 18 = **87**

## QUESTION 5.

https://leetcode.com/problems/climbing-stairs/description/

```
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) return n;
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

## QUESTION 6.

https://leetcode.com/problems/house-robber/description/

```
class Solution {
     public int rob(int[] nums) {
          if (nums.length == 1){
               return nums[0];

          }

          int prev1 = 0; // previous of previous house (i-2)

          int prev2 = 0; // previous house (i-1)

          int val = 0;  // current max value


          for (int num: nums){
               val = Math.max(prev1 + num, prev2);
               prev1 = prev2;
               prev2 = val;
          }

          return prev2;
     }

}
```