**Lab 1**

**Problem 1**

To solve the Knapsack problem, pseudo code below is the solution.

// Input:

// Values (stored in array v)

// Weights (stored in array w)

// Number of distinct items (n)

// Knapsack capacity (W)

// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.

for j from 0 to W do:

m[0, j] := 0

for i from 1 to n do:

for j from 0 to W do:

if w[i] > j then:

m[i, j] := m[i-1,j]

else:

m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])

The m[i,w] value will be the maximum value that can be attained with weight ≤ w using items up to i (first i items). It gives us the following Java code to implement this algorithm

public List<Item> knapsack(Item[] items, int W) {

int NB\_ITEMS = items.length;

int[][] matrix = new int[NB\_ITEMS + 1][W + 1];

// first line is initialized to 0

for (int i = 0; i <= W; i++)

matrix[0][i] = 0;

for (int i = 1; i <= NB\_ITEMS; i++) {

for (int j = 0; j <= W; j++) {

if (items[i - 1].weight > j)

matrix[i][j] = matrix[i-1][j];

else

matrix[i][j] = Math.max(matrix[i-1][j], matrix[i-1][j - items[i-1].weight] + items[i-1].value);

}

}

int res = matrix[NB\_ITEMS][W]; int wt = capacity;

List<Item> finalItems = new ArrayList<>();

for (int i = NB\_ITEMS; i > 0 && res > 0; i--) {

if (res != matrix[i-1][wt]) {

finalItems.add(items[i-1]);

res -= items[i-1].value;

wt -= items[i-1].weight;

}

}

return finalItems; }

where **Item** is an object with the **weight** and **value** properties:

**Problem 2**

If rearranging v[] and w[] with respect to weight in ascending or descending, knapsack method can still give the correct output as result.