Assignment 11

▼ Question R-5.1

Let S = {a, b, c, d, e, f, g} be a collection of objects with benefit-weight values as follows: a:(12,4), b:(10,6), c:(8,5), d:(11,7), e:(14,3), f:(7,1), g:(9,6). What is an optimal
solution to the fractional knapsack problem for S assuming we have a knapsack that can hold objects with total weight 15? Show your work.

Benefit-to-weight ratio

Rank	Item	Ratio	Weight	Benefit
1	f	7.00	1	7
2	e	4.67	3	14
3	a	3.00	4	12
4	b	1.67	6	10
5	c	1.60	5	8
6	d	1.57	7	11
7	g	1.50	6	9

Greedy selection

Item	Take	Weight Used	Benefit Gained	Remaining Capacity
f	all	1	7	14
e	all	3	14	11
a	all	4	12	7
b	all	6	10	1
c	1/5	1	1.6	0

Total benefit: `7 (f) + 14 (e) + 12 (a) + 10 (b) + 1.6 (1/5 of c)` **= 44.6**

▼ Question R-5.11

Solve Exercise R-5.1 above for the 0-1 Knapsack Problem.

i	Item	Benefit	Weight	B/W Ratio
1	a	12	4	3
2	b	10	6	1.67
3	c	8	5	1.6
4	d	11	7	1.57
5	e	14	3	4.666
6	f	7	1	7
7	g	9	6	1.5

Optimal subset:

- **f**: Benefit = 7, Weight = 1
- **e**: Benefit = 14, Weight = 3

- **a**: Benefit = 12, Weight = 4
- **d**: Benefit = 11, Weight = 7

Total Benefit = 7 + 14 + 12 + 11 = 44

Total Weight = 1 + 3 + 4 + 7 = 15

▼ Question R-5.12

Sally is hosting an Internet auction to sell n widgets. She receives m bids, each of the form “I want ki widgets for di dollars,” for i = 1, 2, ..., m. Characterize her
optimization problem as a knapsack problem. Under what conditions is this a 0-1 versus fractional problem?

Case	Condition	Explanation
0-1 Knapsack	Each bid must be **either fully accepted or rejected**	You can’t split a bid across multiple buyers. For example, if someone wants 5 widgets, you either sell them all 5 or none.
Fractional Knapsack	You are allowed to **partially fulfill** a bid	You can sell part of the requested widgets (e.g., someone asks for 5, you sell them 2). This is less realistic for auction platforms unless explicitly allowed.

▼ Question A

Based only on the characterizing equations (B[k,w]), give a recursive pseudo code algorithm for the 0-1 knapsack problem (do this from the equations and without looking at my solution in the notes), then memoize it so it is efficient. Compare your algorithm to the two given in the lecture notes (iterative dynamic programming version and recursive non-memoized algorithm) in terms of time and space complexity.

```
Algorithm KnapsackMemo(k, w)

   memo = Array of size [n+1][W+1] initialized to UNDEFINED
   Memo(memo, k, w)


Algorithm Memo(memo, k, w):
  if k == 0 or w == 0:
     return 0

  if memo[k][w] ≠ UNDEFINED:
     return memo[k][w]

  if weight[k] > w:
     memo[k][w] = KnapsackMemo(k-1, w)
  else:
     memo[k][w] = max(
        KnapsackMemo(k-1, w),
        value[k] + KnapsackMemo(k-1, w - weight[k])
     )
```

```
return memo[k][w]
```

▼ **Question B**

How can we modify the dynamic programming algorithm from simply computing the best benefit value for the 0-1 knapsack problem (like A above) to computing the
assignment (subset) that gives the maximum benefit? Design a pseudo code algorithm to do the trace back through the 2-dimensional array as we described in the lecture.

```
Algorithm KnapsackWithTraceback(values ← new array [1..n], weights ← new array [1..n], W):
    // Step 1: Build DP table
    DP ← array [0..n][0..W] all zeros

    for k = 1 to n:
        for w = 0 to W:
            if weights[k] > w:
                DP[k][w] = DP[k-1][w]
            else:
                DP[k][w] = max(
                    DP[k-1][w],
                    values[k] + DP[k-1][w - weights[k]]
                )

    // Step 2: Traceback to find items selected
    w = W
    chosen_items = new empty List

    for k = n downto 1:
        if DP[k][w] != DP[k-1][w]:
            chosen_items.add(k)
            w = w - weights[k]

    return DP[n][W], chosen_items
```

▼ **Question C**

Suppose we have a set of objects that have different sizes s1, s2, ..., sn, and we have some positive upper limit L. Design an efficient pseudo code algorithm to determine the subset of objects that produces the largest sum of sizes that is no greater than L. Hint: dynamic programming similar to 0-1 knapsack problem, except only size/weight and no benefit.

```
Algorithm MaxSubsetSum(s ← [1..n], L):
    DP ← new array [0..n][0..L] all false
    DP[0][0] = true

    for k = 1 to n:
```

```
    for w = 0 to L:
        if DP[k-1][w] == true:
            DP[k][w] = true
            if w + s[k] ≤ L:
                DP[k][w + s[k]] = true

for w = L downto 0:
    if DP[n][w] == true:
        max_sum = w
        break

subset = new empty List
w = max_sum

for k = n downto 1:
    if w ≥ s[k] and DP[k-1][w - s[k]] == true:
        subset.add(k)
        w = w - s[k]

return max_sum, subset
```