Practice Set 10: solutions

Problem 1

Search through the upper **diagonals** of the table, beginning with the top right corner, until you find the first occurrence of a 1. The position of this 1 indicates the substring that is a palindrome. Therefore, it remains only to output all the palindrome substrings with that length. The outer loop goes from $k = n$ to $k = 1$. Note that $n - k + 1$ represents the number of entires in the diagonal. When $k = n - 1$ for example, there are 2 entires in the diagonal, and $n - k + 1 = 2$. The index $i + k - 1$ represents the indices of the rows in a specific diagonal. Again, when $k = n - 1$, the values we consider are $L[1, 1 + (n - 1) - 1] = L[1, n - 1]$ and $L[2, 2 + (n - 1) - 1] = L[2, n]$.

```
Initialize FoundMax = false.
for k = n to 1
    for i = 1 to i = n − k + 1
        if L [i, i + k − 1] = 1
            FoundMax = true
            Print substring s [i, i + k − 1]
    if FoundMax=true break
```

In the worst case, this searches through the entire upper triangle of the $n \times n$ table, which has approximately $n^2/2$ entries. Since each iteration of the loop above runs in constant time, the runtime is $O(n^2)$.

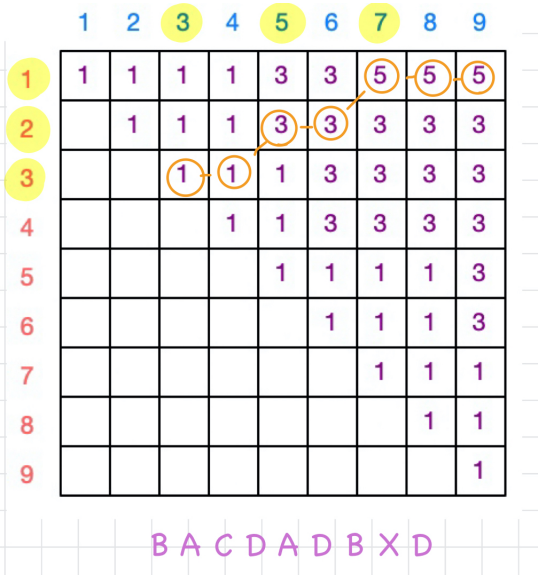
Problem 2

In this problem, we need to keep track of which characters are "selected" to be part of the palindrome subsequence. In the example table below, note that the length of the longest palindrome subsequence is stored in $P[1, n]$. We can back-track through the table in such a way that we select the next subsequence that contains the longest palindrome subsequence. We stop the search when we arrive at the diagonal.

```
Set CharsToPrint[i] = True for character at index i if it is part of the longest palindrome subsequence:
    Initialize array CharsToPrint[1 . . . n] with all False entries.
    i = 1
    j = n
    while i ≤ j
        if s[i] = s[j]
            CharsToPrint[i] = True
            CharsToPrint[j] = True
            i = i + 1
            j = j − 1
        else if P[i + 1, j] > P[i, j − 1]
            i = i + 1
        else
            j = j − 1
Next, print out the characters of the Palindrome subsequence:
    for i = 1 to n
        if CharsToPrint[i] = True
            Print s[i]
```

The above algorithm either increments i or decrements j at each iteration. Therefore the maximum number of iterations is $O(n)$. Since each iterations performs a constant number of operations, the runtime

is $O(n)$.



The characters we should print are at indices $1, 2, 3, 5, 7$ corresponding to the palindrome subsequence $BACAB$

Problem 3

Brute force: We consider all possible subset of weights, and for each possible subset, we total the weight to determine if it is exactly T. There are 2^n possible subsets, and for each subset it takes time $O(n)$ to sum the weights. Therefore the runtime of the brute force approach is $O(2^n \cdot n)$.

Dynamic Programming solution: We refer to the set of selected weights as S. We begin by formalizing the problem recursively. There are n items and let's assume their weights are stored in the array $w[1 \ldots n]$. Consider the following two cases for the last weight w_n:

Case 1:. Weight w_n is included in set S, if w_n is not too heavy. In this case, the remaining target weight is $T - w[n]$.
Case 2:. The last weight is not placed in S. In this case, the remaining total weight is still T.

Notice that this recursive relationship references looking up subproblems based on subsets of the weights, and the target sum. Therefore, we build a table $M[i, j]$ that stores the true/false values corresponding to whether or not target weight j can be made using weights $1 \ldots i$.

Defining the table: $M[0 \ldots n, 0 \ldots T]$

- Entry $M[i, j]$ = **true** if there is a set of weights from $1 \ldots i$ that can be selected to make a sum of j.
- If $j = 0$ then by selecting *no* weights we have found a valid solution: $M[i, 0] = \text{true}$
- If there are no weights to select from, then it is impossible to create any sum $j > 0$: $M[0, j] = \text{false}$
- If weight w_i is too heavy ($w_i > j$) then we must consider the set of weights $1 \ldots (i-1)$: set $M[i, j] = M[i-1, j]$
- Otherwise, we could either include or not include weight w_i:

$$\text{If } M[i-1, j] = \textbf{true} \text{ OR if } M[i-1, j-w[i]] = \textbf{true} \text{ then set } M[i, j] = \text{true}$$

- If entry $M[n, T] = \text{true}$ then there is a selection of weights that have a total sum of T.

The table can be filled up in a row by row, left to right order since each recursive reference above is to an entry that is either in a previous row or a previous column. The algorithm is described below:

WeightSet(w,T)

Step 1: Initialize the all cells with F.

Initialize the first column of the table: for $i = 0$ to n set $M[i, 0] = true$

Initialize the first row of the table: for $j = 1$ to T set $M[0, j] = false$.

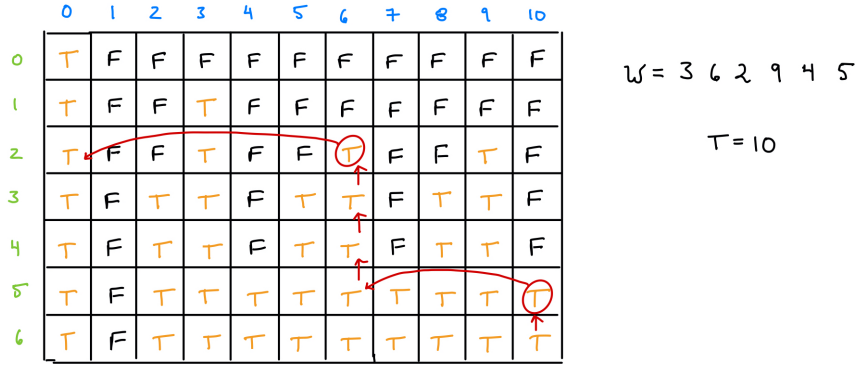
Step 2: Loop through the table row by row from left to right, filling in the entries using the recurrence relationship above.

```
for i = 1 to n
    for j = 1 to T
        if w[i] > j
            M[i, j] = M[i − 1, j]
        else if M[i − 1, j] = true
            M[i, j] = true
        else if M[i − 1, j − w[i]] = true
            M[i, j] = true
            Selected[i, j] = true
Return M[n, T]
```

Runtime: The algorithm runs through a table of size $n \cdot T$ and performs a constant number of operations per cell. Therefore the total runtime is $\Theta(n \cdot T)$.

Output the subset: How do you find the subset that creates this sum? Trace back the true values through the table. Below is an example for a table run on the input weights $w = \{3, 6, 2, 9, 4, 5\}$ and $T = 10$. The red arrows represent the path followed by the search below. Notice that whenever an item is "chosen" the target sum is updated. The red circles represent the cases for which $Selected[i, j] = true$. (You can also write the pseudocode without the use of $Selected[]$.)

```
Initialize j = T and i = n
While j > 0 :
    if Selected[i, j] is true:
        Output w[i]
        j = j − w[i]
    Update i = i − 1
```



Recursive Solution: The recursive algorithm below takes as input the size of the weights $w[]$ and indices i and j, and returns TRUE if there is a subset of weights from $w[1, \ldots, i]$ that has total sum j. The initial call to the recursive algorithm is RecursiveWeightSet(w, n, T). The procedure follows the cases of the dynamic programming solution above, but makes recusive calls instead of making look-ups in a table.

RecursiveWeightSet(w,i,j)

```
if j = 0
    return true
else if i = 0
    return false
else if w[i] > j
```

return RecursiveWeightSet$(w, i-1, j)$
else if RecursiveWeightSet$(w, i-1, j) =$ true
return true
else if RecursiveWeightSet$(, i-1, j-w[i]) =$ true
return true

Problem 4

Assume that the set of weights is stored in array $w[1 \ldots n]$, and their corresponding values are stored in array $v[1 \ldots n]$. The goal is to select a maximum-value subset S of the weights whose total weight is at most T.

We begin the dynamic programming solution in the usual way, by formulating the problem recursively. The optimal subproblems are defined by selecting from smaller subsets of the weights. For example, consider the maximum-value using only weight 1, and next the maximum-value selected from weights 1 and 2, and next the maximum-value selected from weights 1, 2 and 3. If we continue in this way, eventually we will determine the maximum-value set selected from *all* weights.

We now formalize this recursive relationship. If we focus on the last weight in the sequence, this weight could be either *included* or *excluded* from set S. We summarize these two options below:

Option 1: If we include $w[i]$ in set S, then the maximum total weight is now $T - w[i]$, and we have added a value of $v[i]$. Therefore, we would like to solve the optimal subproblem of selecting the maximum-value subset from the weights $w_1 \ldots w_{i-1}$ having total weight at most $T - w[i]$.

Option 2: If we do not include $w[i]$ in set S, then the solution is equivalent to solving the subproblem of selecting the maximum-value subset from the weights $1 \ldots (i-1)$ having total weight at most T.

Since the goal is to select the set S that maximizes the total value, then the solution should select the maximum value of the above two options. The optimal solutions to the subproblems need to be stored for easy look-up by the dynamic programming solution. The table $V[i, j]$ can be used to store the optimal solution to the problem of using weights $w_1, w_2, \ldots, w_i$ having a maximum weight of j.

Defining the table: $V[0 \ldots n, 0 \ldots T]$

- Let $V[i, j]$ store the maximum-value subset selected from weights $w_1, \ldots, w_i$ having total weight at most j
- If we select no items, then the maximum value is 0. Therefore $V[0, j] = 0$ for all $j \geq 0$
- If the maximum allowable weight is 0, then we must select no items (assume that the weights are positive). Therefore $V[i, 0] = 0$ for all $i \geq 0$.
- If $w[i] \leq j$ then we consider the maximum of either including weight w_i or not. The entry in red corresponds to option 1 above, and the entry in blue corresponds to option 2.

$$V[i, j] = \max\{v[i] + V[i-1, j-w[i]], V[i-1, j]\}$$

- Otherwise if $w[i] > j$, then we must exclude this weight:

$$V[i, j] = V[i-1, j]$$

- The final optimal value to the original problem is stored in $V[n, T]$.

The above recursion references entries in either a previous column or a previous row. Therefore the table can be filled row by row, from left to right.

MaxValueSet(w,T)

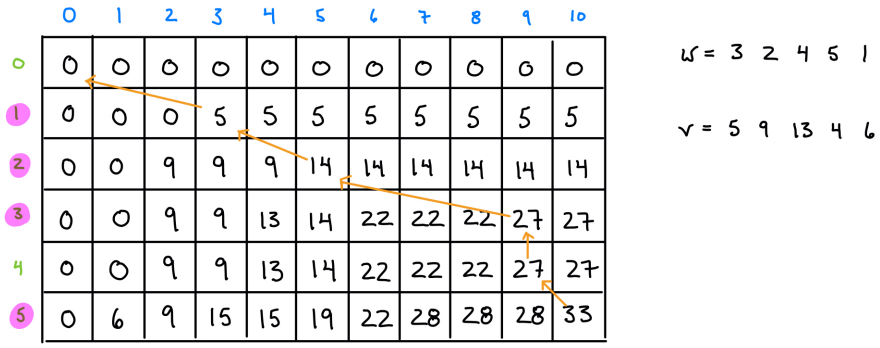
Step 1: Initialize the first row: for $j = 0$ to T set $V[0, j] = 0$
Initialize the first column: for $i = 0$ to n set $V[i, 0] = 0$

Step 2: Loop through the table row by row from left to right, filling in the entires using the above recursive relationship.

```
for i = 1 to n
    for j = 1 to T
        if w[i] ≤ j
            V[i, j] = max{v[i] + V[i − 1, j − w[i]], V[i − 1, j]}
        else V[i, j] = V[i − 1, j]
return V[n, T]
```

Runtime: The algorithm loops through a table of size $n \cdot T$ performing a constant number of steps for each entry. Therefore the overall runtime is $\Theta(n \cdot T)$.

Below is an example using the weights $w = \{3, 2, 4, 5, 1\}$ with corresponding values $v = \{5, 9, 13, 4, 6\}$ and maximum weight $T = 10$.



The optimal solution is the set of weights $\{3, 2, 4, 1\}$ with a total value of 33.

Finding the set of weights: We need to output the set of weights that correspond to the optimal value in entry $V[n, T]$. We can trace back through the table following the orange arrows in the above diagram. These diagonal arrows represent the case where entry $V[i, j]$ was made by selecting weight $w[i]$. The vertical arrows represent the case where weight $w[i]$ was not selected.

```
Initialize j = T and i = n
While i > 0 :
    if V[i, j] ≠ V[i − 1, j]
        j = j − w[i]
        Output "weight w[i] is selected with value v[i]"
    Update i = i − 1
```

Problem 5

We begin by identifying subproblems. In order to arrive at station n, there must have been some previous stop, which was the last stop made before arriving at station n. Suppose that stop was station i. Then the battery at station i must be long enough to travel $n - i$ miles. Furthermore, the number of stops we took to *get* to station i must have been *optimal*. Certainly, we don't *know* what that last stop was. However, like with the rod-cutting problem, we can take the **minimum** over all possible previous stops in order to determine the minimum number of stops to station n.

The table:

Define a table $M[0 \ldots n]$ where $M[i]$ represents the minimum number of stops needed to get to station i. We do no include station 0 as a "stop", but the last station is included as a final "stop". We initialize $M[0] = 0$. The final answer is stored in $M[n]$.

The recurrence:

To get to station i, we must have had some previous stop, at either $0, 1, \ldots, i-1$. The value of $M[i]$ is the **minimum** over *all possible* previous stops. Note that a previous stop at station j is only possible if the amount of water at station j is enough to get you to station i. Therefore, $M[i]$ is the *minimum* of all $M[j] + 1$ where $j = 0..i-1$, as long as $w[j] \geq i - j$.

We fill in the table from left to right as in the rod-cutting problem:

MinStops(b)

```
Set M[0] = 0
for i = 1 to n
        mymin = n
        for j = 0 to i − 1
            if b[j] ≥ i − j
                mymin = min(M[j] + 1, mymin)
         M[i] = mymin
Return M[n]
```

The runtime:

The runtime to fill entry $M[i]$ of the table is ci, for some constant c, since the for loop must run through all table entries for which $j < i$. Therefore the overall runtime is at most $c + 2c + 3c + \ldots nc = \Theta(n^2)$. (Same as rod-cutting).