1. A greedy algorithm to solve the 0-1 knapsack problem is as follows.

   - Sort the items in decreasing order of v_i/w_i. (In other words, prioritize items with higher value per weight.)
   - Keep adding items into the knapsack until the knapsack can no longer accommodate items.

   Give a small counterexample to show that this greedy algorithm is not optimal.

   Solution.

   Let's consider the case when the maximum capacity of the knapsack is $W = 20$.

   We have $n = 3$ items and the following weights and values:

item i	1	2	3
value v_i	22	19	19
weight w_i	11	10	10

   According to our greedy algorithm, we will sort the items as follows: 1, 2, 3. First, we add item 1 to the knapsack, so we have a total value of 22 and a remaining weight of 9. However, we will be unable to add items 2 or 3 to the knapsack since we will be going over the maximum capacity.

   However, the optimal solution is to add items 2 and 3 to the knapsack with a total value of 38.

   Thus, this greedy algorithm is not optimal.

2. This problem is about segmented least squares. Consider the data set in the file. The first line of the file is the number of data points n, and each of the succeeding lines gives the (x, y) coordinates of one data point. **The points are not ordered. Don't forget to sort the points based on x-coordinate**. It is not guaranteed that no two data points have the same x-coordinate.

   (a) Implement the first DP algorithm in the handout to find the optimal way to partition the points to fit a piecewise linear model with $l = 3$ lines. Give a top-down DP implementation which also outputs the regression coefficients for each line. Your DP must have a time complexity of $O(n^2 l)$.

   (b) Implement the second DP algorithm in the handout to find the optimal way to partition the points to fit a piecewise linear model with an arbitrary number of lines, where the cost per line is $C = 10,000$. Give a bottom-up DP implementation which also outputs the regression coefficients for each line. Your DP must have a time complexity of $O(n^2)$.

   Solution.

   (a) The python code is in the attached jupyter notebook with comments for the explanations.

Recursive Algorithm.

For this first case, the DP recurrence $f(j,k)$ outputs the minimum sum of squared errors (SSE) obtained by using k lines to cover j points: $p_1, p_2, ..., p_j$. The function has two parameters: j for the index of the last point covered by the least squares lines and k for the number of lines used. Note that each line must fit at least two points and that for this problem, $k = 3$. Let $L_{i,j}$ be the SSE for points $p_i, p_{i+1}, ..., p_j$ after performing simple linear regression.

$$f(j,k) = \begin{cases} L_{1,j} & \text{if } k = 1 \\ \min_{2k-1 \le i \le j-1}(L_{i,j} + f(i-1, k-1)) & \text{if } k > 1 \end{cases}$$

- Every time we run the recursive function, we need to save the left index i of the best-fitting line so that we can calculate regression coefficients of the least squares line.

After running the DP function, we obtained the following piecewise linear function that had the lowest SSE. Note that we have a total of 1000 distinct points denoted by $p_1, p_2, ..., p_{1000}$ sorted by increasing x-coordinate.

Least Squares Line.

$$f(x) = \begin{cases} 2.6199x + 41.4170 & \text{if } 0.599(p_1) \le x \le 168.6967(p_{323}) \\ -0.0828x + 499.5263 & \text{if } 170.4067(p_{324}) \le x \le 341.5237(p_{676}) \\ -2.7978x + 1428.2028 & \text{if } 343.1398(p_{677}) \le x \le 498.5574(p_{1000}) \end{cases}$$

Time Complexity/Space Complexity.

The time complexity of this DP algorithm is $O(n^2 l)$. The recursive function has to go through n states for all points. Each function call can be done in $O(nl)$ because the parameter space has size nl, where at most, $j = n$ and $k = l$. Thus, in total, the DP algorithm runs in $O(n^2 l)$. Note that prefix sum arrays for the sums of the x and y coordinates of the points that were used in computing the SSE were computed before the DP recurrence. This means that the subarrays computed in the main DP algorithm can be done in constant $O(1)$ time. The space complexity of the algorithm is $O(nl)$ since this is the size of the memo table used. The memo table has form (j,k), where j has n possible values and k has l possible values.

(b) The bottom-up python implementation is in the attached jupyter notebook.

Recursive Algorithm.

Similar to the first case, the function outputs the minimum SSE obtained from the piecewise linear model. The main difference is that the dp function f has only one parameter j for the index of the last point covered. Instead of fixing the number of lines, the recursive algorithm can create as many lines as possible given that each line has a cost of $C = 10,000$ which prevents our model from connecting all pairs of adjacent points.

$$f(j) = \begin{cases} 0 & \text{if } j = 1 \\ \min_{1 \le i \le j-1}(L_{i,j} + C + f(i-1)) & \text{if } j > 0 \end{cases}$$

- Similar to the first case, we save the optimal index i for each j that minimizes the SSE.

Least Squares Line.

$$f(x) = \begin{cases} 3.2699x + 11.4751 & \text{if } 0.599(p_1) \leq x \leq 86.9667(p_{156}) \\ 1.9756x + 125.4943 & \text{if } 88.7357(p_{157}) \leq x \leq 163.5283(p_{312}) \\ 0.7051x + 334.1240 & \text{if } 164.7958(p_{313}) \leq x \leq 250.3958(p_{496}) \\ -0.7672x + 704.0777 & \text{if } 252.0358(p_{497}) \leq x \leq 341.5237(p_{676}) \\ -2.1598x + 1182.4614 & \text{if } 343.1398(p_{677}) \leq x \leq 421.9327(p_{840}) \\ -3.3707x + 1693.9691 & \text{if } 422.7515(p_{841}) \leq x \leq 498.5574(p_{1000}) \end{cases}$$

Time Complexity/Space Complexity.

The time complexity of this DP algorithm is $O(n^2)$. In generating the array that stored the minimum errors, the algorithm had a nested for loop that went through all possible i and j values, which means the for loop had nm iterations. This case also used prefix sum arrays in computing the SSE. Each iteration of the for loop was done in constant O(1). Thus, the total time complexity of this algorithm is $O(n^2)$. The space complexity of the algorithm is $O(n)$ since this is the size of the array used to store all the minimum errors for all the possible values of j.

3. You are given n trash bags, numbered 1 to n, as well as a table w, where w_i (positive integer) is the weight of trash bag i. You cannot split or otherwise modify the weight of each bag.

   You wish to transfer the trash bags into processing containers. Each container has a maximum weight capacity of M (positive integer).

   In the following items, you must write programs which determine if it is possible to fit all the trash bags into k containers such that

   - Each trash bag is in one container.
   - While a container may contain multiple trash bags, the total weight of the bags in a single container does not exceed M.

   You may assume that your program is given n, the list w, and the container capacity M (positive integer).

   (a) Suppose there are exactly $k = 3$ containers available. Your program must have a time complexity of $O(nM^3)$ or better.

   (b) Suppose there are exactly $k = 2$ containers available. Your program must have a time complexity of $O(nM)$ or better.

   Solution.

   (a) The python code is in the attached jupyter notebook.

   Code Explanation.

   Let n be the be the number of trash bags and let w be the array of weights. Trash bag i has weight $w[i]$, where $i = 1, 2, 3, ..., n$. Each container has a maximum capacity of M.

   - Each trash bag with weight w_i can be placed in any one of the three available containers, as long as its remaining capacity c_j , where $j = 1, 2, 3$ is greater than or equal to w_i. Note that $c_j \geq 0$ since capacity is nonnegative.

- We can define $f(i, c_1, c_2, c_3)$ for this decision problem as follows:

$$f(i, c_1, c_2, c_3) = \begin{cases} \text{True} & \text{if } w[i] \leq c_1 \text{ or } w[i] \leq c_2 \text{ or } w[i] \leq c_3 (i = 1) \\ \text{False} & \text{if } w[i] > c_1 \text{ and } w[i] > c_2 \text{ and } w[i] > c_3 (i = 1) \\ f(i-1, c_1 - w[i], c_2, c_3) & \text{if } w[i] \leq c_1 (1 < i \leq n) \\ f(i-1, c_1, c_2 - w[i], c_3) & \text{if } w[i] \leq c_2 (1 < i \leq n) \\ f(i-1, c_1, c_2, c_3 - w[i]) & \text{if } w[i] \leq c_3 (1 < i \leq n) \\ \text{False} & \text{if } w[i] > c_1 \text{ and } w[i] > c_2 \\ & \text{ and } w[i] > c_3 (1 < i \leq n) \end{cases}$$

- The first two cases of the piecewise function are when we consider when there is one trash bag left to place in the containers ($i = 1$). We output True when at least one of the containers has enough remaining capacity to hold the last trash bag and False otherwise.
- The following three cases are the DP recurrence since we have the choice to put trash bag i in any of the three containers as long as the container has enough remaining capacity to hold the weight. We adjust the parameters accordingly by subtracting the weight of trash bag i from the container it was placed in. We then run the function again considering the next trash bag and the updated remaining capacities.

Time Complexity/Space Complexity.

The time complexity of this DP algorithm is $O(nM^3)$. The recursive function has to go through n states for each of the trash bags from 1 to n. In each function call, the total number of possible combinations of c_1, c_2, and c_3 is $(M+1)^3$ since each container can have a remaining capacity from 0 to M. Thus, in total, the recurrence algorithm has to do $n(M+1)^3$ operations, which is $O(nM^3)$ time. The space complexity of this algorithm is also $O(nM^3)$. The memo table is a dictionary of the form (i, c_1, c_2, c_3). The total number of possible keys in the memo dictionary is $n(M+1)^3$, which is $O(nM^3)$.

(b) This case is very similar to the first case with three containers. The only adjustment is that there is one less parameter since there are only two containers.

When $k = 2$, we have the following recursive algorithm:

$$f(i, c_1, c_2) = \begin{cases} \text{True} & \text{if } w[i] \leq c_1 \text{ or } w[i] \leq c_2 (i = 1) \\ \text{False} & \text{if } w[i] > c_1 \text{ and } w[i] > c_2 (i = 1) \\ f(i-1, c_1 - w[i], c_2) & \text{if } w[i] \leq c_1 (1 < i \leq n) \\ f(i-1, c_1, c_2 - w[i]) & \text{if } w[i] \leq c_2 (1 < i \leq n) \\ \text{False} & \text{if } w[i] > c_1 \text{ and } w[i] > c_2 (1 < i \leq n) \end{cases}$$

Time Complexity/Space Complexity.

The time and space complexity of this case is $O(nM^2)$ by similar reasoning as the previous case. The DP function can be called for every combination of i (index of the current bag), c_1 (remaining capacity of container 1), and c_2 (remaining capacity of container 2). This leads to a maximum of $n((M+1)^2)$ recursive calls, which is $O(nM^2)$. The space complexity of this case is $O(nM^2)$ since the memo table stores all combinations of i, c_1, c_2.

4. Given an integer array A of length n, find L and R such that $\sum_{i=L}^{R} A[i]$ is maximized and $R - L + 1 \geq X$. You may assume that your program is given the integer array A and a positive integer $X \leq n$. Your solution must have a time complexity of $O(n)$ or better, and it must output L and R, not just the maximum sum. Suboptimal solutions are accepted but will receive a maximum of 3 out of 5 pts.

   Solution.

   I used a top down approach to solve the DP problem. The code can be seen in the jupyter notebook.

   Recursive Algorithm.

   Let L and R be the left and right indices of the subarray with the maximum sum. Let $f(R)$ be the maximum subarray sum that ends exactly at index R. Let P be the prefix-sum array of A.

$$f(R) = \begin{cases} 0 & \text{if } R < X - 1 \\ P[R] & \text{if } R = X - 1 \\ \max(P[R] - P[R - X], A[R] + f(R - 1)) & \text{if } X \leq R \leq n - 1 \end{cases}$$

   The maximum subarray sum of A is $\max_{X-1 \leq i \leq n-1} f(i)$. We don't consider the subarrays that end at index less than $X - 1$ because the subarrays don't meet the required length.

   Note that we can only pick the maximum subarray if it has a length of at least X. This is equivalent to $R - L + 1 \geq X$, where R is the right endpoint of the maximum subarray and L is the left endpoint of the maximum subarray.

   Time Complexity/Space Complexity.

   The time complexity of this DP algorithm is $O(n)$. The prefix-sum array of A was precomputed and ran in $O(n)$ time. There are n states, corresponding to each iteration of the for loop from X to $n - 1$. In the worst case, when $X = 1$, the for loop will run n times. Each iteration of the for loop can be done in constant $O(1)$ time. The space complexity of the algorithm is also $O(n)$ since the memo table only has one parameter R, which has size n.

5. A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order.

   Given three strings A (length m), B (length n), C (length $m+n$), describe and analyze an algorithm to determine the number of ways that C is a shuffle of A and B. It is possible that C is not a shuffle of A and B, in which case the answer is 0.

   Your algorithm must have a time complexity of at most $O((m + n)mn)$. Suboptimal solutions are accepted but will receive a maximum of 3 out of 5 pts.

   Solution.

   I used a bottom-up approach to solve the DP problem. The code can be seen in the jupyter notebook.

   Recursive Algorithm.

   We consider the $(i + j)th$ character of C. Let i be the number of characters from A and let j be the number of characters from B. Remember from the problem that the order of characters of X and Y is preserved in C.

   We save the number of ways that C is a shuffle of A and B in the two-dimensional array, dp.

Let $f(i, j)$ be the number of ways that the first $i + j$ characters of C are a shuffle of the first i characters of A and the first j characters of B. We can define the recurrence relation for $f(i, j)$ as the following:

$$f(i,j) = \begin{cases} 1 & \text{if } A[:i] = C[:i] \text{ or } B[:j] = C[:j] \text{ when either i or j} = 0. \\ 0 & \text{if } A[:i] \neq C[:i] \text{ or } B[:j] \neq C[:j] \text{ when either i or j} = 0. \\ f(i-1,j) + f(i,j-1) & \text{if } i, j \neq 0 \end{cases}$$

- The first two cases are the base cases when all of the characters of C up to the ith position are from A or if all the characters of C up to the jth position are from B. The function will return 1 if the characters match and 0 otherwise.
- When $i, j \neq 0$, we consider two possibilities: whether the $(i + j)$th character of C comes from A or if the $(i + j)$th character of C comes from B.
  - If the $(i + j)$th character comes from A, in the previous recurrence, C had the first $i - 1$ characters of A and the first j characters of B. Thus, the total number of ways for this case is equal to $f(i - 1, j)$.
  - If the $(i + j)$th character comes from B, in the previous recurrence, C was made up of the first i characters of A and the first $j - 1$ characters of B. Thus, the total number of ways for this case is equal to $f(i, j - 1)$.
- Since we are only concerned with the total number of ways that C is a shuffle of A and B, we add the two cases. The number of ways to form the $(i + j)$th character of C depends on the number of ways to form the previous characters using either characters from A or B (but not both for the same character).

Time Complexity/Space Complexity.

The time and space complexity of this DP algorithm is $O(mn)$. When generating the dp list, we used a nested for loop that went through all the characters in A and B. Since A had m characters and B had n characters, the for loop ran mn times. Each iteration of the for loop can be done in constant $O(1)$ time so the total time complexity is $O(mn)$, which is under the required time complexity of $O((m + n)mn)$. The space complexity is also $O(mn)$ because the 2-D array dp has size $(m + 1)$ x $(n + 1)$. Since the size of the table grows proportionally to the product of m and n, the space complexity is considered $O(mn)$.