- This problem set is about designing polynomial-time *non-recursive* DP algorithms. For each problem I have provided one or more sample projections to try. *These are just hints.* Some of the hints may *not* lead to polynomial-time DP algorithms; I have not tried designing DP algorithms using many of these suggestions. I have provided these just to give some *examples* of projections for you to try. I strongly encourage you to try all the hints, to see what you get in each case.
- The expected solution for each problem in this set is a non-recursive DP algorithm that correctly solves the problem within the stated time bound. You are not required to explain *how* you arrived at the algorithm. In particular, the solution does *not* have to be based on the idea of starting with a projection. And even if it is, you don't have to mention the projection in your solution; all that is required is a correct non-recursive DP algorithm that runs within the stated bounds. This will be the case in the exams as well.
- See previous practice problem sets for other instructions.

1. Recall the problem of efficiently multiplying a chain of matrices that we saw in class:

Least Cost Matrix-chain Multiplication

Input: An array $D[0 \dots n]$ of $n + 1$ positive integers.

Output: The least cost (total number of arithmetic operations required) for computing the matrix product $A_1 A_2 \cdots A_n$ where each A_i ; $1 \leq i \leq n$ has dimensions $D[i-1] \times D[i]$, given that multiplying a $p \times q$ matrix with a $q \times r$ matrix requires $\mathcal{O}(pqr)$ arithmetic operations.

   (a) Write the pseudocode for a *non-recursive* algorithm that solves this problem using *dynamic programming*, in time polynomial in n.

   Hint 1: Try the idea of projecting onto the "breakpoint" that we discussed in class.

   Hint 2: The breakpoint of Hint 1 is based on the outermost right-parenthesis that encloses the *first* matrix in the input sequence. Try projecting on to an "innermost breakpoint": the location of the right-bracket that encloses the first matrix and is *closest* to it. As an example: In the parenthesization $((A_1A_2)(A_3A_4))A_5$ the breakpoint that we discussed in class is 4, whereas the innermost breakpoint is 2.

   (b) Argue that your procedure correctly solves the problem.

   (c) What is the running time of your algorithm in the asymptotic notation?

2. Recall the 0-1 Knapsack Without Repetition problem that we saw in class:

0-1 Knapsack Without Repetition

Input: A non-negative integer n; a set of n items $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ where item I_j has value v_j and weight w_j; and a maximum weight capacity W. The values, weights, and W are all integers.

Output: The maximum sum, taken over all subsets of $\mathcal{I}$ of total weight at most W, of the total value of the items in that set.

That is, the goal is to maximize

$$\sum_{i=1}^{n} v_i x_i$$

subject to the conditions

$$\left(\sum_{i=1}^{n} w_i x_i\right) \leq W,$$

and

$$x_i \in \{0, 1\} \text{ for } 1 \leq i \leq n.$$

For this question, assume that the weights and values are given, respectively, as arrays $\mathsf{Value}[1 \ldots n]$ and $\mathsf{Weight}[1 \ldots n]$, where $\mathsf{Value}[j]$ is the value and $\mathsf{Weight}[j]$ is the weight of item I_j.

(a) Write the pseudocode for a *non-recursive* algorithm that solves this problem using *dynamic programming*, in time polynomial in $(n + W)$.

Hint: Try the idea of projecting onto a *prefix* of the list of items, that we discussed in class.

(b) Argue that your procedure correctly solves the problem.

(c) What is the running time of your algorithm in the asymptotic notation?

3. Recall the problem of finding the *length* of a longest *increasing subsequence*, from the previous problem set.

(a) Write the pseudocode for a *non-recursive* procedure that finds the length of a longest increasing subsequence of integer array $A[1 \ldots n]$ using *dynamic programming*, in time polynomial in n.

Hint 1: Try projecting the solution space—the set of all increasing subsequences of A—onto *starting indices*. That is, for $1 \leq i \leq n$ let S_i denote all those solutions whose first element is $A[i]$.

Hint 2: Try projecting the solution space onto *ending indices*. That is, for $1 \leq i \leq n$ let S_i denote all those solutions whose *last* element is $A[i]$.

(b) Argue that your procedure correctly solves the problem.

(c) What is the running time of your algorithm in the asymptotic notation?

4. Refer the previous problem set for the definition of a *subsequence* of an array. Let A, B be two arrays. An array X is said to be a *common subsequence* of A and B if X is a subsequence both of A and of B. The input to the *Longest Common Subsequence* problem consists of two arrays A and B, and the goal is to find a common subsequence of A and B with the most number of elements. In this question we look at the (slightly) simpler problem of finding the *length* of a longest common subsequence of the input arrays.

   (a) Write the pseudocode for a *non-recursive* procedure that finds the length of a longest common subsequence of input arrays $A[1 \ldots m]$ and $B[1 \ldots n]$ using *dynamic programming*, in time polynomial in $(m + n)$.

   Hint 1: Try projecting the solution space—the set of all common subsequences of A and B—onto *starting indices*. That is, for $1 \leq i \leq m, 1 \leq j \leq n$ let $S_{i,j}$ denote all those common subsequences whose first elements in the two arrays are $A[i], B[j]$, respectively.

   Hint 2: Try projecting the solution space onto *ending indices*. That is, for $1 \leq i \leq m, 1 \leq j \leq n$ let $S_{i,j}$ denote all those common subsequences whose *last* elements in the two arrays are $A[i], B[j]$, respectively.

   (b) Argue that your procedure correctly solves the problem.

   (c) What is the running time of your algorithm in the asymptotic notation?

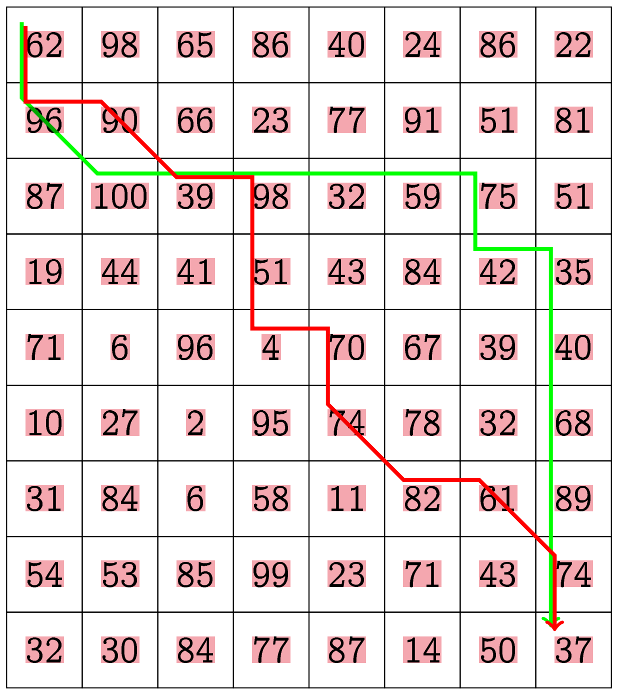
5. The input to this problem is an $m \times n$ array $C[0 \ldots (m-1)][0 \ldots (n-1)]$ of positive integers. A *valid path* through this array is a path that starts at location $C[0][0]$ and ends at location $C[m-1][n-1]$ using (only) the following three types of steps: (i) move to the right by one cell, (ii) move down by one cell, and (iii) move diagonally down and to the right by one cell. That is, if the path is currently at location $C[i][j]$ then after one step it can be only at one of the following three locations: (i) $C[i][j+1]$; $j < n$, (ii) $C[i+1][j]$; $i < m$, and (iii) $C[i+1][j+1]$; $i < m, j < n$. The *cost* of a valid path is the sum of all the array elements which the path touches, including the first and last elements in the path.

   Figure 1 on the next page shows two valid paths through such an array with $m = 9, n = 8$. The cost of the green path is $62+96+100+39+98+32+59+75+42+35+40+68+89+74+37 = 946$. The cost of the red path is $62+96+90+39+98+51+4+70+74+82+61+74+37 = 838$.

   The goal is to compute the *least cost of a valid path* through the input array C.

   (a) Write the pseudocode for a *non-recursive* procedure that takes C, m, n as arguments and finds the least cost of a valid path through C using *dynamic programming*, in time polynomial in $(m + n)$.

   Hint 1: Try projecting the solution space—the set of all valid paths through C—onto *the direction of the first step* that the path takes.



62	98	65	86	40	24	86	22
96	90	66	23	77	91	51	81
87	100	39	98	32	59	75	51
19	44	41	51	43	84	42	35
71	6	96	4	70	67	39	40
10	27	2	95	74	78	32	68
31	84	6	58	11	82	61	89
54	53	85	99	23	71	43	74
32	30	84	77	87	14	50	37

Figure 1: A sample input for Question 5, with two examples of valid paths.

Hint 2: Try projecting the solution space onto the direction of the *last step* that the path takes.

(b) Argue that your procedure correctly solves the problem.

(c) What is the running time of your algorithm in the asymptotic notation?

6. The input to this problem consists of an array $C[0 \ldots (n-1)]$ of n distinct positive integers, and a target positive integer T. The elements of C are coin denominations, and T is a target amount to be made up using these denominations. The goal is to find the *least number* of coins, of these denominations, that can be used to make up the amount T. Note that each denomination may be used any number of times. The goal is to minimize the total number of *coins*, not of denominations, used. If a certain denomination $C[i]$ is used d times, this adds d to the number of coins.

   (a) Write the pseudocode for a *non-recursive* algorithm that accepts C, n, T as arguments and computes the smallest *number of coins* of the denominations specified by C, that can be used to make up the target amount T. If the given denominations cannot be used to make up the specified target—Example: $C = [2, 4, 6], T = 15$—then the algorithm should return False. The algorithm should solve this problem using *dynamic programming*, in time polynomial in $(n + T)$.

   Hint 1: Try projecting onto *the coin denominations* in a solution.

Hint 2: Try projecting onto *the number of coins* in a solution.

(b) Argue that your procedure correctly solves the problem.

(c) What is the running time of your algorithm in the asymptotic notation?

The following questions are programming problems from LeetCode. Solve each of them in three different ways: (i) using a pure recursive algorithm, (ii) using a memoized version of the recursive algorithm, and (iii) using a non-recursive DP algorithm. Use Python3 to implement your solutions, and test your programs using LeetCode's testing mechanism.

7. ★ Climbing Stairs
8. ★ Min Cost Climbing Stairs
9. ★ Word Break
10. ★ Integer Replacement
11. ★ Partition Array for Maximum Sum