- Many questions in this problem set ask you to design *non-recursive* DP algorithms. For many of these questions, 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 algorithm you get in each case.
- For each question where the expected solution is a non-recursive DP algorithm, 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 running-time bounds. This will be the case in the exams as well.
- See previous practice problem sets for other instructions.

1. Recall the following recursive algorithm that we saw in class, for solving the Rod Cutting problem. Here n is the length of the rod to be cut up and sold, and P is an array of selling prices for different lengths of rod[1].

```
CutRod(n, P)
1  if n == 0
2      return 0
3  maxRevenue ← −1
4  for i ← 1 to n
5      iFirstRevenue = P[i] + CutRod((n − i), P)
6      if iFirstRevenue > maxRevenue
7          maxRevenue ← iFirstRevenue
8  return maxRevenue
```

   (a) Explain why CutRod correctly solves the Rod Cutting problem.

   (b) Note that—assuming that each array access and each operation involving up to two numbers can be done in constant time—the running time of the algorithm is proportional to the *number of times* that CutRod is invoked, starting with the initial call CutRod(n, P). Write a recurrence for this number, and solve it to get an estimate of this number.

   (c) Write pseudocode that *memoizes* the above recursive solution. Derive an upper bound on the running time of this memoized algorithm, as a function of n. How does this bound compare to the bound that you derived for the recursive solution in part (b)?

2. We derived the following dynamic programming solution to Rod Cutting in class:

[1] See the Dynamic Programming chapter in CLRS for more details on this problem.

```
CutRod-DP(n, P)
 1  R ← an array of length n + 1        // R = R[0 ... n]
 2  R[0] = 0
 3  for j ← 1 to n
 4      jMaxRevenue = −1
 5      for i = 1 to j
 6          iRevenue = P[i] + R[j − i]
 7          if iRevenue > jMaxRevenue
 8              jMaxRevenue ← iRevenue
 9      R[j] ← jMaxRevenue
10  return R[n]
```

We have now seen three solutions to this problem: the recursive solution given as part of Question 1, the memoized version of Question 1(c), and the above DP. Each of these algorithms only gave us the maximum revenue; they did not tell us *where/how to cut* the input rod to *realize* this maximum revenue.

Modify each of these three algorithms for Rod Cutting so that it also returns a collection of lengths (starting with zero at one end of the rod, and ending with n at the other end) where we can cut the input rod of length n to realize the maximum possible revenue. A sample output for $n = 10$ and some array P of prices might look like: "2, 4, 7, 9".

*Hint: Simplify and conquer. See if it suffices to find, for each length $0 \leq i \leq n$, the best place to make the **first** cut on a rod of length i.*

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

Least Cost Matrix-chain Multiplication

Input: An array $D[0 \ldots n]$ 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.

Let $OPT(s, t)$ denote the least ("optimum") cost for multiplying the sub-sequence $A_s, A_{s+1}, \ldots, A_t$. If $s = t$ then $OPT(s, t) = 0$.

(a) Explain why there must exist an index $1 \leq i < n$ such that $OPT(1, n) = OPT(1, i) + OPT(i + 1, n) + d_0 d_i d_n$.

Note: The main thing to show is *not* the fact that such an i exists, but the claim that it is OK to just add together the $OPT()$ values of the sub-sequences. In particular: Why is it that the global optimum is not *smaller* than this sum?

(b) Write the pseudocode for a recursive algorithm that solves Least Cost Matrix-chain Multiplication, using the claim proved in part (a). Argue that this algorithm correctly

solves the problem.

(c) Prove that your algorithm from part (b) makes $\Omega(2^n)$ recursive calls.

(d) Memoize your algorithm from part (c). Show that it now runs in $\mathcal{O}(n^c)$ time for some fixed constant c. What is the value of c that you get?

4. In this question you will derive *non-recursive* DP algorithms for the Least Cost Matrix-chain Multiplication problem defined in the previous question, in two different ways.

   (a) Write the pseudocode for a *non-recursive* algorithm that solves Least Cost Matrix-chain Multiplication using *dynamic programming*, in time polynomial in n. For this try the idea of projecting onto the "breakpoint" that we discussed in class. Recall that this breakpoint is defined based on the outermost right-parenthesis that encloses the *first* matrix in the input sequence. As an example: In the parenthesization $((A_1A_2)(A_3A_4))A_5$ the breakpoint that we discussed in class is 4.

   (b) Argue that your algorithm of part (a) correctly solves the problem. What is the worst-case running time of your algorithm in the asymptotic notation?

   (c) A different projection might give you a different DP algorithm for the same problem. Write the pseudocode for a *non-recursive* algorithm that solves Least Cost Matrix-chain Multiplication using *dynamic programming*, in time polynomial in n. This time, 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 innermost breakpoint is 2.

   Do you get an algorithm which looks different from your algorithm of part (a)?

   (d) Argue that your algorithm of part (c) correctly solves the problem. What is the worst-case running time of your algorithm in the asymptotic notation?

5. The 0-1 Knapsack Without Repetition problem is defined as follows:

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 $Value[1 \ldots n]$ and $Weight[1 \ldots n]$, where $Value[j]$ is the value and $Weight[j]$ is the weight of item I_j.

(a) Write the pseudocode for a *recursive* procedure NoRepKnapSackRec which solves the 0-1 Knapsack Without Repetition problem for these inputs.

Argue that your procedure correctly solves the problem.

Write a recurrence for the running time of the algorithm, and solve it to obtain a worst-case upper bound on the running time of the algorithm on these inputs.

(b) Memoize your algorithm of part (a). What is the running time of this version?

(c) Write the pseudocode for a *non-recursive* procedure NoRepKnapSackDP which solves the 0-1 Knapsack Without Repetition problem for these inputs in time polynomial in $(n + W)$, *using dynamic programming*. *Hint:* Try projecting onto a *prefix* of the list of items. That is, for each $1 \leq i \leq n$, let projection S_i denote the set of all solutions which pick items *only* from the subset $\{I_1, I_2, \ldots, I_i\}$. See if you can develop this into a DP following the method that we saw in class.

Argue that your procedure correctly solves the problem.

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

6. The 0-1 Knapsack With Repetition problem is defined as follows:

0-1 Knapsack With Repetition

Input: A non-negative integer n; a set of n item *types* $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ where each item of type T_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 multisubsets of $\mathcal{T}$ of total weight at most W, of the total value of the items represented by that multiset.

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

$$x_i \in (\mathbb{N} \cup \{0\}) \text{ for } 1 \leq i \leq n.$$

The difference from 0-1 Knapsack Without Repetition is that here we are allowed to pick more than one item of each type into the collection.

Assume, as before, that the weights and values are given, respectively, as arrays $Value[1 \ldots n]$ and $Weight[1 \ldots n]$, where $Value[j]$ is the value and $Weight[j]$ is the weight of item type T_j.

(a) Write the pseudocode for a *recursive* procedure RepKnapSackRec which solves the 0-1 Knapsack With Repetition problem for these inputs.

Argue that your procedure correctly solves the problem.

Write a recurrence for the running time of the algorithm, and solve it to obtain a worst-case upper bound on the running time of the algorithm on these inputs.

(b) Memoize your algorithm of part (a). What is the running time of this version?

(c) Write pseudocode for a *non-recursive* procedure RepKnapSack1 which solves the 0-1 Knapsack With Repetition problem for these inputs, using *dynamic programming*. *Hint:* Try projecting onto the *item types* in a solution.

Argue that your procedure correctly solves the problem.

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

(d) Write pseudocode for a *non-recursive* procedure RepKnapSack2 which solves the 0-1 Knapsack With Repetition problem for these inputs, using *dynamic programming*. *Hint:* Try projecting onto the *number of items* in a solution.

Argue that your procedure correctly solves the problem.

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

7. A *subsequence* of an array A is any sub-array of A, obtained by deleting zero or more elements of A *without* changing the order of the remaining elements. An *increasing subsequence* of an integer array A is a sub-sequence of A which is in *strict* increasing order.

   The input to the *Longest Increasing Subsequence* problem consists of an integer array A, and the goal is to find an increasing subsequence of A with the most number of elements. See Chapters 2 and 3 of Jeff Erickson's book for various solutions to this problem. In this question we will deal with the (slightly) simpler problem of finding the *length* of a longest increasing subsequence of the input array.

   (a) Write the pseudocode for a *recursive* algorithm LenLis(A, n) that takes an integer array $A[1 \ldots n]$ of length n as input and returns the length of a longest increasing subsequence of A. As always, simplifying the task makes it easier to solve:

      i. First, write the pseudocode for LenLis(A, n) *assuming* that you have access to a function LenLisBigger(A, n, i, j) which takes A, n, and two indices $1 \leq i < j \leq n$ as inputs, and returns the length of a longest increasing subsequence S of $A[j \ldots n]$ with the property that every element of S is *larger* than $A[i]$.

      ii. Now write *recursive* pseudocode for LenLisBigger(A, n, i, j). Make sure that you correctly deal with the base cases.

   (b) Write a recurrence for the running time of your LenLis(A, n) function and solve it.

   (c) Memoize your LenLis(A, n) function. Derive an upper bound on the running time of the memoized version. How does this compare with the running time of the pure recursive version?

8. In this question you will design DP algorithms for the problem of finding the *length* of a longest *increasing subsequence*, as defined in the previous question.

   (a) Write the pseudocode for a *non-recursive* procedure LenLisDP1(A, n)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:* 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 $A[i]$.

   Argue that your procedure correctly solves the problem.

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

   (b) Write the pseudocode for a *non-recursive* procedure LenLisDP2(A, n)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:* 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 $A[i]$.

   Argue that your procedure correctly solves the problem.

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

9. Refer the previous problems 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 procedures correctly solve the problem.

   (c) What are the running times of your algorithms in the asymptotic notation?

10. 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)

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 10, with two examples of valid paths.

$C[i+1][j+1]$; $i < m, j < n$. 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.

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

(b) Argue that your procedures correctly solve the problem.

(c) What are the running times of your algorithms in the asymptotic notation?

11. 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 procedures correctly solve the problem.

(c) What are the running time of your algorithms in the asymptotic notation?

> The remaining 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.

12. Climbing Stairs
13. Min Cost Climbing Stairs
14. Word Break
15. Integer Replacement
16. Partition Array for Maximum Sum