Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Discrete Golf

Carolyn loves to play golf, specifically a version called discrete golf. The goal of discrete golf is to hit a golf ball from checkpoint 1 to checkpoint n on a golf course in as few strokes as possible. Based on the terrain of the course, she knows that from checkpoint i, she can hit the ball up to $d(i) \geq 1$ checkpoints away in one stroke. That is, if the ball is at checkpoint i, she can hit the ball to any of checkpoints $\{i+1, i+2, \ldots, i+d(i)\}$ in one stroke.

(a) Suppose she hits the ball as far as possible every stroke, i.e. if she is at checkpoint i, she hits the ball to checkpoint $i + d(i)$. Give a counterexample where this greedy algorithm does not achieve the minimum number of strokes needed to reach checkpoint n from checkpoint 1.

(b) Suppose that all $d(i)$ are at most D. Describe a $O(nD)$ time dynamic programming algorithm for computing the minimum number of strokes Carolyn needs to reach point n. Justify your algorithm and analyze its runtime and space complexity.

2 Grid of Obstacles

PNPenguin is waddling from the top-left cell to the bottom-right cell of an $m \times n$ grid, only moving between adjacent cells down or to the right. However, some cells contain obstacles, which cause him to become stuck.

Determine the number of valid paths PNPenguin can take without passing through any obstacles.

(a) Design an $O(mn)$-time dynamic programming algorithm that, given m, n, and the locations of the obstacles as input, outputs this count modulo 1337.

Note: We ask you to output your answer modulo 1337 because arithmetic can no longer be treated as $O(1)$ when numbers grow exponentially in the input size. Therefore, during your computation, you should avoid tracking numbers that might be exponentially large in m and n.

(b) What is the space complexity of your algorithm? Can you improve the space complexity of your algorithm?

3 Optimal Set Covering

The Set Cover problem is defined as follows: given a collection $S_1, S_2 \dots S_m$ where each S_i is a subsets of the universe $U = \{1, 2, \dots, n\}$, we want to pick the minimum number of sets from the collection such that their union contains all of U. In this problem we will use dynamic programming to come up with an exact solution.

(a) How long does a brute force solution of trying all the possibilities take? How large can m be compared to n in the worst case?

(b) As you will learn later on in the class, there does not exist a sub-exponential time algorithm for Set Cover. However, we can try to do better than exponential time in m by aiming for exponential time in only n instead. Describe a dynamic programming algorithm that accomplishes this in $O(2^n \cdot nm)$ time.

(a) Define your subproblem.

(b) Write your recurrence relation.

(c) Describe the order in which we should solve the subproblems.

(d) Analyze the runtime of this dynamic programming algorithm.

(e) Analyze the space complexity.

4 Balloon Popping

You are given a sequence of n balloons with each one of a different size. If a balloon is popped, then it produces noise equal to $n_{left} \cdot n_{popped} \cdot n_{right}$, where n_{popped} is the size of the popped balloon and n_{left} and n_{right} are the sizes of the balloons to its left and to its right. If there are no balloons to the left, then we set $n_{left} = 1$. Similarly, if there are no balloons to the right then we set $n_{right} = 1$, while calculating the noise produced.

After popping a balloon, the balloons to its left and right become neighbors. (Note that the total noise produced depends on the order in which the balloons are popped.)

In this problem we will design a polynomial-time dynamic programming algorithm to compute the maximum noise that can be generated by popping the balloons.

Example:
Input (Sizes of the balloons in a sequence): ④ ⑤ ⑦
Output (Total noise produced by the optimal order of popping): 175

Walkthrough of the example:

- **Current State** ④ ⑤ ⑦
  Pop Balloon ⑤
  Noise Produced $= 4 \cdot 5 \cdot 7$

- **Current State** ④ ⑦
  Pop Balloon ④
  Noise Produced $= 1 \cdot 4 \cdot 7$

- **Current State** ⑦
  Pop Balloon ⑦
  Noise Produced $= 1 \cdot 7 \cdot 1$

- **Total Noise Produced** $= 4 \cdot 5 \cdot 7 + 1 \cdot 4 \cdot 7 + 1 \cdot 7 \cdot 1$.

(a) Define your subproblem as follows:

$$C(i, j) = \text{maximum amount of noise produced by popping balloons in the sublist } i, i+1, \ldots, j \text{ first before the other balloons.}$$

What are the base cases? Are there any special cases to consider?

(b) Write down the recurrence relation for your subproblems $C(i, j)$.

Hint: suppose the k-th balloon (where $k \in [i, j]$) is the last balloon popped, after popping all other balloons in $[i, j]$. What is the total noise produced so far?

(c) What is the runtime of a dynamic programming algorithm using this recurrence? What is its space complexity?

Note: you may assume that all arithmetic operations take constant time.

(d) Is it possible to modify your algorithm above to use less space? If so, describe your modification and re-analyze the space complexity. If not, briefly justify.

5 String Shuffling

Let x, y, and z be strings. We want to know if z can be obtained only from x and y by interleaving the characters from x and y such that the characters in x appear in order and the characters in y appear in order.

For example, if $x =$ **butter** and $y =$ **scotch**, then it is true for $z =$ **butterscotch** and $z =$ **sbcuotctterh**, but false for $z =$ **scotchbuttercandy** (extra characters), $z =$ **scotchbutte** (missing a final **r**), and $z =$ **scottchbuter** (out of order). How can we answer this query efficiently? Your answer must be able to efficiently deal with strings with lots of overlap, such as $x =$ **aaaaaaaaaab** and $y =$ **aaaaaaaac**.

(a) Design an efficient algorithm to solve the above problem, briefly justify its correctness, and analyze its runtime. You do *not* need to provide a space complexity analysis (you'll do this in the next part!).

(b) Consider a bottom-up approach to our DP algorithm in part (a). Naively if we want to keep track of every solved sub-problem, this requires $O(|x||y|)$ space (double check to see if you understand why this is the case). How can we reduce the amount of space our algorithm uses?

6 Finding More Counterexamples

In this problem, we will explore why we had to resort to Dynamic Programming for some problems from lecture instead of using a fast greedy approach. Give counter examples for the following greedy algorithms.

(a) For the travelling salesman problem, first sort the adjacency list of each vertex by the edge weights. Then, run DFS N times starting at a different vertex $v = 1, 2, \ldots N$ for each run. Every time we encounter a back edge, check if it forms a cycle of length N, and if it does then record the weight of the cycle. Return the minimum weight of any such cycle found so far.

(b) For the Longest Increasing Subsequence problem, consider this algorithm: Start building the LIS with the smallest element in the array and go iterate through the elements to its right in order. Every time we enounter an element $A[i]$ that is larger than the current last element in our LIS, we add $A[i]$ to the LIS.

(c) Can you construct a counter example for the previous algorithm where the smallest element in the array is $A[0]$?

7 Non-Prefix Code

As we have learned in lecture, the Huffman code satisfies the *Prefix Property*, which states that the bit string representing each symbol is not a prefix of the bit string representing any other symbol. One nice property of such codes is that, given a bit string, there is at most one way to decode it back to a sequence of symbols. However, this is not true anymore once we are working with codes that do not satisfy the Prefix Property. For example, consider the code that maps A to 1, B to 01 and C to 101. A bit string 101 can be interpreted in two ways: as C or as AB.

Your task is to, given a bit string s, determine whether it is possible to interpret s as a sequence of symbols. The mapping from symbols to bit strings of the code will be given to you as a dictionary d (e.g., in the example, $d = \{A : 1, B : 01, C : 101\}$); you may assume that you can access each symbol in the dictionary in constant time.

(a) Describe an algorithm that solves this problem in at most $O(nk\ell)$ where n is the length of the input bit string s, k is the number of symbols, and ℓ is an upper bound on the length of the bit strings representing symbols. Give a proof of correctness and analyze the runtime and space complexity.

(b) **(Optional)** How can you modify the algorithm from part (a) to speed up the runtime to $O((n+k)\ell)$?

