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

Jonny 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, he knows that from checkpoint i, he can hit the ball up to $d(i) \geq 1$ checkpoints away in one stroke. That is, if the ball is at checkpoint i, he can hit the ball to any of checkpoints $\{i+1, i+2, \ldots, i+d(i)\}$ in one stroke.

(a) Suppose he hits the ball as far as possible every stroke, i.e. if he is at checkpoint i, he 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. Give a $O(nD)$ time algorithm for computing the minimum number of strokes Jonny needs to reach point n. **Please provide a 4-part solution.**

Solution:

(a) Consider when $n = 5, d(1) = 2, d(2) = 3, d(3) = 1, d(4) = 1$. The greedy algorithm will visit checkpoints $1, 3, 4, 5$, the optimal route is to visit checkpoints $1, 2, 5$.

(b) **Solution 1:** let $s(i)$ be the minimum number of strokes needed to reach checkpoint n from checkpoint i. Then the base case is $s(n) = 0$, and for $i < n$, $s(i) = 1 + \min_{i+1 \leq j \leq i+d(i)} s(j)$. The DP table has n entries, and each takes $O(D)$ time to update, so the runtime is $O(nD)$. Since each entry of the DP table only relies on the next D entries, it suffices to only store D entries, i.e. only $O(D)$ space is needed.

Solution 2: let $s(i)$ be the minimum number of strokes needed to reach checkpoint i from checkpoint 1. Then the base case is $s(1) = 0$, and for $i > 1$, $s(i) = 1 + \min_{i-D \leq j \leq i-1 : i \leq j+d(j)} s(j)$. The DP table has n entries, and each takes $O(D)$ time to update, so the runtime is $O(nD)$. Since each entry of the DP table only relies on the previous D entries, it suffices to only store D entries, i.e. only $O(D)$ space is needed.

2 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. **Please provide a 4-part solution**.

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

Solution:

Main Idea: We define our subproblems as follows: let $A[i]$ be a boolean variable representing whether $s[:i]$ can be represented by a sequence of encodings. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$\begin{aligned} A[i] &= \bigvee_{\substack{\text{any symbol } a \text{ whose encoding} \\ \text{matches the end of } s[:i]}} (\text{whether it is possible to interpret } s[:i] \text{ with } a \text{ as its last symbol}) \\ &= \bigvee_{\substack{\text{any symbol } a \text{ whose encoding} \\ \text{matches the end of } s[:i]}} (\text{whether it is possible to interpret } s[:i - \text{length}(d[a]])) \\ &= \bigvee_{\substack{\text{symbol } a \text{ in } d \\ s[i-\text{length}(d[a])+1:i]=d[a]}} A[i - \text{length}(d[a])] \end{aligned}$$

Note here that we set $A[0] = \mathsf{TRUE}$. Our algorithm simply computes the above formula in a trivial manner.

Pseudocode:

```
procedure TRANSLATE($s$):
    Create an array $A$ of length $n + 1$ and initialize all entries to FALSE.
    Let $A[0]$ = TRUE
    for $i := 1$ to $n$ do
        for each symbol $a$ in $d$ do
            if $i \geq \text{length}(d[a])$ and $d[a] = s[i - \text{length}(d[a]) + 1 : i]$ then
                $A[i] = A[i] \vee A[i - \text{length}(d[a])]$
    return $A[n]$
```

Proof of Correctness: We can show this via a simple induction argument.

Base Case. When $i = 0$, we can interpret $s[:0]$ as just the empty string. Hence, $A[0] = \mathsf{TRUE}$.

Inductive Step. Suppose that $A[0], \ldots, A[i-1]$ contains the right value. We will show that the above recurrence relation gives the right value for $A[i]$. To do this, we partition interpretations of $s[:i]$ as a sequence of symbols $a_1 \ldots a_k$ based on the ending symbol a_k. For $a_k = a$, if the suffix of $s[:i]$ coincides with $d[a]$, every interpretation $a_1 \ldots a_k$ has a one-to-one correspondence with an interpretation $a_1 \ldots a_{k-1}$ of $s[:i - \text{length}(d[a])]$. From our inductive hypothesis, we know whether exactly whether $s[:i - \text{length}(d[a])]$ is interpretable based on $A[:i - \text{length}(d[a])]$. On the other hand, if the suffix of $s[:i]$ differs from $d[a]$, then there is no interpretation of $s[:i]$ ending with symbol a. Performing a logical or over all symbols a's implies that our recurrence relation yields the right value for $A[i]$.

Finally, note that our program below implements this recurrence in a straightforward way, so the output of our program is indeed $A[n]$, representing whether s is interpretable.

Runtime Analysis: There are n iterations of the outer for loop and k iterations of the inner for loop. Inside each of these loops, checking that the two strings are equal takes $O(\text{length}(d[a])) \leq O(\ell)$ time. Hence, the total running time is $O(nk\ell)$.

Note that it is possible to speed up the algorithm running time to $O((n + k)\ell)$ using a trie instead of reconstructing the string every time, but this is not required to receive full credit for the problem.

Space Complexity Analysis: We store n subproblems in $A[\cdot]$, each subproblem taking up constant space. Thus, the overall space complexity is $O(n)$.

Note that since any $A[i]$ only depends on the previous ℓ subproblems, you can optimize the space to $O(\ell)$.

3 Longest Common Subsequence

In lecture, we covered the longest increasing subsequence problem (LIS). Now, let us consider the longest common subsequence problem (LCS), which is a bit more involved. Given two arrays A and B of integers, you want to determine the length of their longest common subsequence. If they do not share any common elements, return 0.

For example, given $A = [1, 2, 3, 4, 5]$ and $B = [1, 3, 5, 7]$, their longest common subsequence is $[1, 3, 5]$ with length 3.

We will design an algorithm that solves this problem in $O(nm)$ time, where n is the length of A and m is the length of B.

(a) Define your subproblem.

Hint: looking at the subproblem for Edit Distance may be helpful.

(b) Write your recurrence relation.

(c) In what order do we solve the subproblems?

(d) What is the runtime of this dynamic programming algorithm?

(e) What is the space complexity of your DP algorithm? Is it possible to optimize it?

Solution:

Algorithm Description: Let $L[i][j]$ be the length of the LCS between the first i characters of A and j characters of B. (i.e between $A[0:i]$ and $B[0:j]$). Then the final answer will be $L[n][m]$. The recurrence is given as follows:

$$L[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1 & \text{if } A[i] = B[j] \\ \max\{L[i-1][j], L[j-1][i]\} & \text{otherwise} \end{cases} \tag{1}$$

Correctness: The correctness follows by induction. Observe that if the last characters of A and B match, then any longest common subsequence should have the same last character. Otherwise, at least one of $A[i-1]$ and $B[j-1]$ will not be in the solution, so we take the maximum over the possibilities.

Runtime: Our DP has $(n+1)(m+1)$ states and calculating each state takes $O(1)$. Hence, the algorithm runs in $O(nm)$.

Space: Naively, we can just store all the states, which would take up $O(nm)$ space. However, we can note that to compute any $L[i][j]$, we only need the subproblems involving $i-1$, $j-1$, and i, j. Thus, we simply need to store the last 2 "rows" or "columns" of the DP table, yielding a space complexity of $O(2\min\{n,m\}) = O(\min\{n,m\})$.

4 Balloon Popping Problem.

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 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:

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

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.

Solution:

(a) Base case: When the size of sublist to pop out is only one balloon n_{popped}, return the noise $n_{left} \cdot n_{popped} \cdot n_{right}$. In addition, we need to initialize $n_0 = 1$ and $n_{n+1} = 1$ assuming the input is from 1 to n. In our algorithm we'll never actually pop these two balloons as they are just dummy balloons on the left and right.

(b)
$$C(i, j) = \max_{i \le k \le j} \{C(i, k-1) + C(k+1, j) + n_{i-1} \cdot n_k \cdot n_{j+1}\}$$

Justification: The leaves of the tree represent the last balloon being popped, so here, k represents the index of the last balloon being popped. Then we can recurse up, and at each level find the splitting point k that maximizes the value of the subtree (noise produced for that sequence).

(c) The runtime of this algorithm is $O(n^3)$ because there are $O(n^2)$ subproblems, each of which requires $O(n)$ time to compute given the answers to smaller subproblems. The space complexity is $O(n^2)$, as there are $O(n^2)$ subproblems and each subproblem has size $O(1)$.

(d) No, it is not possible to optimize the space complexity to be $o(n^2)$. This is because when computing each $C(i, j)$, we need *all* smaller subproblems within the range $[i, j]$. So in every iteration of the DP algorithm we need all subproblems $O(n^2)$.

5 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?

Solution: With a collection of size m, there are 2^m possible choices. We can check each possibility in $O(mn)$ so the runtime is $O(2^m \cdot mn)$. m can be up to $O(2^n)$ in the worst case, so this runtime is huge.

(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.

Solution: For a subset $T \subseteq U$, define $f(T, i)$ to be the minimum number of sets from the collection $S_1, S_2 \ldots S_i$ needed to cover the subset T. Then, the final answer will be $f(U, m)$. For each S_i, we consider two possibilities, we either pick S_i and obtain the subproblem of covering the set $T \setminus S_i$ with $S_1 \ldots S_{i-1}$, or we don't pick S_i and need cover all of T using the $S_1 \ldots S_{i-1}$.

Thus, the recurrence is given by,

$$f(T,i) = \begin{cases} 0 & \text{if } T = \emptyset \\ \infty & \text{if } i = 0 \text{ and } |T| > 0 \\ \min\{f(T, i-1), f(T \setminus S_i, i-1) + 1\} & \text{otherwise} \end{cases} \tag{2}$$

where we iterate through the subproblems in increasing i. We can iterate in any order of T for a fixed i.

Since we have $2^n \cdot m$ states and calculating $T \setminus S_i$ takes $O(n)$, this can be implemented in $O(2^n \cdot mn)$ time.

Naively, the space complexity is proportional to the number of states $O(2^n \cdot m)$. However, we note that for any $f(T, i)$, we only require the subproblems $f(\cdot, i-1)$. Thus, we only need to store $2 \cdot 2^n$ subproblems (for $f(\cdot, i)$ and $f(\cdot, i-1)$), yielding an optimized space complexity of $O(2^n)$.

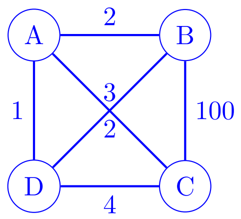
6 Finding More Counterexamples

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.

Solution: This algorithm is equivalent to doing the following: for every possible start node, repeatedly pick the unvisited node with the smallest edge weight to the current node to visit next.

Consider a counter-example where $N = 4$:



Here the edge B-D has weight 2 and A-C has wight 3.

Any such DFS will always have the edge A-D, since it is the smallest weight edge. However that will cause the cycle to contain edge B-C, which blows up the weight. The optimal solution doesn't use the edge A-D and instead does A-B-D-C-A.

(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.

Solution: $A = [7, 6, 8, 5]$. LIS is $[7, 8]$ but greedy outputs $[5]$.

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

Solution: $A = [3, 9, 7, 8]$. Now LIS is $[3, 7, 8]$ but greedy outputs $[3, 9]$.