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 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 Edit Distance subproblem 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:

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

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 $O(2\min\{n, m\}) = O(\min\{n, m\})$.

2 Optimal Set Covering

Recall the set covering problem that you saw in lecture. Given a collection $S_1, S_2 \ldots S_m$ where each S_i is a subsets of the universe $U = \{1, 2, \ldots n\}$, we want to pick the minimum number of sets from the collection such that their union contains all of U. We saw in lecture that Greedy cannot do better than an approximation. 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 $O(2^m mn)$. m can be up to $O(2^n)$ in the worst case, so this runtime is huge.

(b) As seen in lecture, we cannot hope for a polynomial time algorithm here but we can try to do better than exponential time in m by aiming for exponential time in 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 $S_1, S_2 \dots S_i$ 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 $S_1 \dots S_{i-1}$, or we don't pick S_i and need cover all of T using the $S_1 \dots S_{i-1}$.

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 $O(2^n)$.

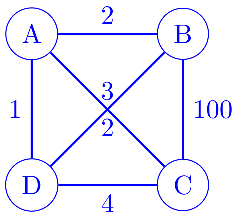
3 Finding More Counterexamples

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 $[3, 9]$.

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 $g_{left} \cdot g_{popped} \cdot g_{right}$, where g_{popped} is the size of the popped balloon and g_{left} and g_{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 $g_{left} = 1$. Similarly, if there are no balloons to the right then we set $g_{right} = 1$, 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:

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

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 subproblem? What is the space complexity?

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

(d) Is it possible to further optimize the space complexity? In either case, explain your reasoning.

Solution:

(a) Base case: When the size of sublist to pop out is only one balloon g_{popped}, return the noise $g_{left} \cdot g_{popped} \cdot g_{right}$. In addition, we need to initialize $g_0 = 1$ and $g_{n+1} = 1$ 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)
$$C(i,j) = \max_{i \le k \le j} \{C(i,k-1) + C(k+1,j) + g_{i-1} \cdot g_k \cdot g_{j+1}\}$$

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 LP Basics

Linear Program. A *linear program* is an optimization problem that seeks the optimal assignment for a linear objective over linear constraints. Let $x \in \mathbb{R}^n$ be the set of variables and $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$. The canonical form of a linear program is

$$\begin{aligned} \text{maximize } & c^\top x \\ \text{subject to } & Ax \leq b \\ & x \geq 0 \end{aligned}$$

Any linear program can be written in canonical form.

Let's check this is the case:

(i) What if the objective is maximization?

(ii) What if you have a constraint $Ax \leq b$?

(iii) What about $Ax = b$?

(iv) What if the constraint is $x \leq 0$?

(v) What about unconstrained variables $x \in \mathbb{R}$?

Solution:

(i) Take the negative of the objective.

(ii) Negate both sides of the inequality.

(iii) Write both $Ax \leq$ and $Ax \geq b$ into the constraint set.

(iv) Change of variable: replace every x by $-z$, and add constraint $z \geq 0$.

(v) Replace every x by $x^+ - x^-$, add constraints $x^+, x^- \geq 0$. Note that for every solution to the original LP, there is a solution to the transformed LP (with the same objective value). Similarly, if there is a feasible solution for the transformed problem, then there is a feasible solution for the original problem with the same objective value.

Dual. The dual of the canonical LP is

$$\begin{aligned} \text{minimize } & y^\top b \\ \text{subject to } & y^\top A \geq c^\top \\ & y \geq 0 \end{aligned}$$

Weak duality: The objective value of any feasible primal $\leq$ objective value of any feasible dual

Strong duality: The *optimal* objective values of these two are equal.

Both are solvable in polynomial time by the Ellipsoid or Interior Point Method.

6 Taking a Dual

Consider the following linear program:

$$\begin{aligned} &\max\ 4x_1 + 7x_2 \\ x_1 + 2x_2 &\le 10 \\ 3x_1 + x_2 &\le 14 \\ 2x_1 + 3x_2 &\le 11 \\ x_1, x_2 &\ge 0 \end{aligned}$$

Construct the dual of the above linear program.

Solution: If we scale the first constraint by $y_1 \ge 0$, the second by $y_2 \ge 0$, the third by $y_3 \ge 0$, and we add them up, we get an upperbound of $(y_1 + 3y_2 + 2y_3)x_1 + (2y_1 + y_2 + 3y_3)x_2 \le (10y_1 + 14y_2 + 11y_3)$. We need y_1, y_2, y_3 to be non-negative, otherwise the signs in the inequalities flip. Minimizing for a bound for $4x_1 + 7x_2$, we get the tightest possible upperbound by

$$\begin{aligned} &\min\ 10y_1 + 14y_2 + 11y_3 \\ y_1 + 3y_2 + 2y_3 &\ge 4 \\ 2y_1 + y_2 + 3y_3 &\ge 7 \\ y_1, y_2, y_3 &\ge 0 \end{aligned}$$

7 LP Meets Linear Regression

One of the most important problems in the field of *statistics* is the *linear regression problem.* Roughly speaking, this problem involves fitting a straight line to statistical data represented by points (x_1, y_1), $(x_2, y_2), \dots, (x_n, y_n)$ on a graph. Denoting the line by $y = a + bx$, the objective is to choose the constants a and b to provide the "best" fit according to some criterion. The criterion usually used is the *method of least squares*, but there are other interesting criteria where linear programming can be used to solve for the optimal values of a and b.

Suppose instead we wish to minimize the sum of the absolute deviations of the data from the line:

$$\min \sum_{i=1}^{n} |y_i - (a + bx_i)|$$

Write a linear program with variables a, b to solve this problem.

Hint: Create new variables z_i and new constraints to help represent $|y_i - (a + bx_i)|$ in a linear program.

Solution:

Note that the smallest value of z that satisfies $z \geq x, z \geq -x$ is $z = |x|$. Now, consider the following linear programming problem:

$$\begin{aligned} &\min \sum_{i=1}^{n} z_i \\ &\text{subject to} \begin{cases} y_i - (a + bx_i) \leq z_i & \text{for } 1 \leq i \leq n \\ (a + bx_i) - y_i \leq z_i & \text{for } 1 \leq i \leq n \end{cases}. \end{aligned}$$

Since $\sum_i z_i$ is minimized, z_i will be set to the $\max(y_i - (a + bx_i), (a + bx_i) - y_i)$. Note that $\max(y_i - (a + bx_i), (a + bx_i) - y_i)$ is, in fact, $|y_i - (a + bx_i)|$.

If for some solution we have that $z_i > |y_i - (a + bx_i)|$, then by setting $z_i = |y_i - (a + bx_i)|$ we will get a solution with a smaller value of the objective function, therefore the initial solution was not optimal. Hence, the constraints requires that the optimal solution will set $z_i = |y_i - (a + bx_i)|$, so the new problem is indeed equivalent to the original problem. However, now it is a linear programming problem.