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 Tiring Resorts

PNPenguin has grown tired of delivering ice-cakes and has decided to start a new resort business that serves penguins who want to go on vacations. They have chosen two locations for their resort business – Premium Resort and Common Resort. Unfortunately due to understaffing, each resort can only serve one penguin at a time. There are n total penguins wanting to go on vacations and they don't mind which resort they end up going to. The i-th penguin stays $d[i]$ days and will pay $2 \cdot c[i]$ per day at Premium Resort or $c[i]$ per day at Common Resort. After each penguin leaves, PNPenguin needs ℓ days (for Common Resort) or 2ℓ days (for Premium Resort) to clean up the resort and get it ready to serve the next penguin.

PNPenguin wants to find a way to assign the n penguins to resorts to maximize their profit (assume no operation costs). Note that, although a bit scummy, PNPenguin does not have to assign all the vacationers to resorts.

(a) Design a dynamic programming algorithm that determines the maximum profit PNPenguin can make over T days by serving the penguin vacationers. Please provide the following:

(i) Subproblem definition.
(ii) Recurrence Relation (make sure to include base cases!).
(iii) Time Complexity Analysis.
(iv) Space Complexity Analysis.

Solution:

Subproblem:

$p(t_1, t_2, i)$ = the maximum profit PNPenguin can make with t_1 days left for Premium Resort and t_2 days left for Common Resort using the first i penguins.

Recurrence Relation:

$$p(t_1, t_2, i) = \max \begin{cases} p(t_1 - d[i] - 2\ell, t_2, i-1) + 2 \cdot d[i] \cdot c[i], \\ p(t_1, t_2 - d[i] - \ell, i-1) + d[i] \cdot c[i], \\ p(t_1, t_2, i-1) \end{cases}$$

Base cases are $p(\cdot, \cdot, 0) = 0$.

Time Complexity Analysis:

We have a total of $T \cdot T \cdot n$ subproblems and each subproblem takes $O(1)$ to compute. Therefore, the final runtime of this algorithm will be $O(T^2 n)$.

Space Complexity Analysis:

The naive space complexity is $O(T^2 n)$, as we're storing all the elements in the full DP table.

However, notice that in order to compute $p(\cdot, \cdot, i)$ for a given i, we only need the $p(\cdot, \cdot, i-1)$ values. Thus, we only need to store $2 \cdot T^2 = O(T^2)$ elements (for $p(\cdot, \cdot, i)$ and $p(\cdot, \cdot, i-1)$). This optimization allows us to achieve a space complexity of $O(T^2)$.

(b) Now suppose each penguin has a specific start date and end date; that is, the ith penguin will arrive on $start[i]$ and leave on $end[i]$. Two penguins can start and end on the same date.

Assume for this subpart that all vacationers are super-duper tidy, meaning that clean up is no longer needed after each penguin leaves. Modify your algorithm from part (a) to find the new maximum profit PNPenguin can make.

Solution: First, sort all the penguins in increasing order of end date. Then, we proceed with the following DP algorithm:

Subproblem:

$p(t_1, t_2, i)$ = the maximum profit PNPenguin can make with t_1 days for Premium Resort and t_2 days for Common Resort using the first i penguins.

Recurrence Relation:

$$p(t_1, t_2, i) = \max \begin{cases} p(start[i], t_2, i-1) + 2 \cdot d[i] \cdot c[i] \\ p(t_1, start[i], i-1) + d[i] \cdot c[i], \\ p(t_1, t_2, i-1) \end{cases}$$

Base cases are $p(\cdot, \cdot, 0) = 0$.

Note the first case is only possible if $end[i] \leq t_1$, the second case is only possible if $end[i] \leq t_2$.

Time Complexity Analysis: Same as part (a).

Space Complexity Analysis: Same as part (a).

2 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 $S_1, S_2 \dots S_i$ 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,

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

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 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:

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

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 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 $O(n^2)$.

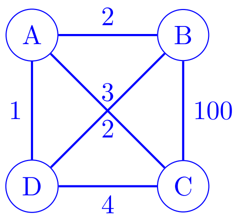
4 Finding More Counterexamples

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 $v = 1, 2, \dots N$ 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 $[3, 9]$.