Assignment 8

CS 577 *Dynamic Programming* Fall 2024

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Related Readings: `http://pages.cs.wisc.edu/~hasti/cs240/readings/`

Name: ______________________ Wisc id: ______________________

- **Purpose of Homework:**
  - Algorithm design and analysis, like any skill, can only be developed through consistent practice and feedback. Whether it's cooking, playing basketball, integration, gardening, interviewing, or teaching, theoretical knowledge alone is not sufficient. The comfortable feeling of "Oh, sure, I get it" after following a well-presented lecture or hearing a TA explain a homework solution is a seductive, yet dangerous trap. True understanding comes from doing the thing—by actually solving the problems yourself.
  - The homework assignments are your opportunity to practice. Lectures, textbooks, office hours, labs, and guided problem sets are designed to build intuition and provide justification for the skills we want you to develop. However, the most effective way to develop those skills is by attempting to solve the problems on your own. The process is far more important than the final solution.
  - Expect to get stuck. It's normal to have no idea where to start on some problems. That's why you have access to a textbook, lecture slides, and discussions. The journey of wrestling with the problem is an essential part of the learning process.

Scoring Guidelines

You do not need to solve all the exercises!!!

Each exercise is worth 12.5+2.5 points. Your final score can be calculated as:

$$\text{Score} = \left\lceil \sum_{k\in[20]} \text{Exercise}_k \right\rceil$$

Instructions for Solving the Problems: Please solve each of the following problems using **dynamic programming**. For each problem:

- Define a set of subproblems.
- Relate the subproblems recursively.
- Check that the relation is acyclic or Plot the dependencies arrows.
- Provide base cases.
- Construct a solution from the subproblems.
- Analyze the running time.

Correct but inefficient dynamic programs will be awarded significant partial credit.

For each problem, indicate whether the requested running time is one of the following:

1. **Polynomial**
2. **Pseudopolynomial**
3. **Exponential** (in the size of the input)

This categorization will be worth **2.5 points per problem**.

Dynamic Programming Guidelines

Guidelines for Describing a Dynamic Programming Solution Dynamic Programming generalizes Divide and Conquer recurrences by solving subproblems where dependencies form a directed acyclic graph (DAG) rather than a tree. It applies to optimization problems, such as maximizing or minimizing a scalar value, or to counting problems, where all possibilities must be evaluated. Below are key steps to describe a dynamic programming problem effectively:

1. **Define Subproblems**
    - Clearly describe the meaning of each subproblem in words and in terms of relevant parameters.
    - Identify subsets of the input: prefixes, suffixes, or contiguous subsequences.
    - Incorporate partial state tracking, such as using auxiliary variables to extend subproblems.
2. **Relate Subproblems Recursively**
    - Define the recursive relationship: $x(i) = f(x(j), \dots)$ for one or more indices $j < i$.
3. **Establish Topological Order**
    - Ensure the recursive relations are acyclic and form a DAG.
    - Order the subproblems based on this DAG structure to enable correct evaluation.
4. **State the Base Cases**
    - Clearly specify the solutions for all independent subproblems where the recursive relation does not apply.
    - These base cases will serve as the foundation for building more complex solutions.
5. **Solve the Original Problem**
    - Show how to derive the solution to the original problem from the solutions of the subproblems.
    - Optionally, use parent pointers or a similar structure to recover the actual solution, not just the objective value.
6. **Analyze Time Complexity**
    - Sum the work done for all subproblems: $\sum_{x \in X} \text{work}(x)$.
    - If the work for each subproblem is $O(W)$ and the number of subproblems is $|X|$, the total time complexity is $|X| \cdot O(W)$.
    - Ensure that recursive calls are counted as taking $O(1)$ time per step to avoid overestimating the complexity.

Example: *How to write a DP Solution*

Tim the Beaver always attends the career fair, not to find a career, but to collect free swag. There are n booths at the career fair, each giving out one known type of swag. To collect a single piece of swag from booth i, having integer coolness c_i and integer weight w_i, Tim must stand in line at that booth for exactly t_i minutes. After obtaining a piece of swag from one booth, it will take Tim exactly 1 minute to get back in line at the same booth or any other. Tim's backpack can hold at most weight b in swag, but at any time Tim may spend h minutes to run home, empty the backpack, and return to the fair, taking 1 additional minute to get back in line. Given that the career fair lasts exactly k minutes, describe an $O(nbk)$-time algorithm to determine the maximum total coolness of swag Tim can collect during the career fair.

1. **Subproblems**
   - $x(i, j)$: the maximum total coolness of swag collectible in the next i minutes with j weight remaining in Tim's backpack (where Tim is at the career fair and may immediately stand in line).
2. **Relate**
   - Tim can either:
     - Collect no more swag.
     - Stand in a line and collect a piece of swag.
     - Go home and empty the backpack.
   - The recursive relation is:

$$x(i,j) = \max \begin{cases} 0 & \text{always} \\ c_{k'} + x(i - t_{k'} - 1, j - w_{k'}) & \text{for } k' \in \{1, \dots, n\} \text{ where } w_{k'} \leq j, t_{k'} < i \\ x(i - h - 1, b) & \text{when } i > h \end{cases}$$

3. **Topological Order**
   - Subproblem $x(i, j)$ only depends on subproblems with strictly smaller i, so the dependency graph is acyclic.
4. **Base**
   - Tim needs time to collect swag, so nothing is possible if the time is zero.
   - $x(0, j) = 0$ for all $j \in \{0, \dots, b\}$.
5. **Original Problem**
   - $x(k, b)$ is the maximum total coolness of swag collectible in k minutes, starting with an empty backpack, as desired.
6. **Time**
   - # subproblems: $\leq (k + 1)(b + 1) = O(kb)$, one $x(i, j)$ for each $i \in \{0, \dots, k\}$ and $j \in \{0, \dots, b\}$.
   - Work per subproblem: $O(n)$ to try all items and check constraints.
   - Total time: $O(nbk)$, which is pseudo-polynomial in b and k.

Homework Guidelines

- **Collaboration and Academic Integrity:**
  - You are encouraged to work together on homework problems, but you must list everyone you worked with for each problem.
  - You must write everything in your own words and properly cite every external source you use, including ideas from other students. The only sources that you are not required to cite are the official course materials (lectures, notes, homework solutions).
  - Plagiarism is strictly prohibited. Using ideas from other sources or people without citation is considered plagiarism. Copying verbatim from any source, even with citation or permission, is also considered plagiarism. Don't cheat.
- **Submission Instructions:**
  - Submit your homework solutions as PDF files on Gradescope. Submit one PDF file per numbered homework problem.
  - Gradescope will not accept other text file formats such as plain text, HTML, LaTeX source, or Microsoft Word (.doc or .docx).
  - Homework submitted as images (.png or .jpg) will not be graded.
  - Each submitted PDF file should include the following information prominently at the top of the first page: [your full name]_[course title]_[homework assignment number].pdf
- **Solution Writing:**
  - When writing an algorithm, a clear description in English is sufficient. Pseudo-code is not required.
  - Ensure that your algorithm is correct by providing a justification, and analyze the asymptotic running time of your solution. Even if your algorithm does not meet the requested time bounds, you may receive partial credit for a correct, albeit inefficient, solution.
  - Pay close attention to the instructions for each problem. Partial credit may be awarded for incomplete or partially correct answers.

Rod Cutting

1. Once upon a time, a skilled carpenter named Theo was given a long rod of wood, measuring exactly L units in length. Theo knew that different-sized pieces of wood could fetch different prices at the market, depending on their length and quality. His goal was to cut the rod into smaller segments to maximize his earnings. However, Theo could choose to make as many or as few cuts as he wanted—he could even sell the entire rod uncut! The challenge was that the total length of the pieces had to sum exactly to L. Each possible segment of length ℓ had a pre-determined value, denoted by $v(\ell)$. Given the values for segments of all lengths from 1 to L, Theo needed to decide on the best way to divide the rod. Additionally, show that the standard greedy heuristic of cutting based on the highest value per unit length is not optimal. Now, Theo wonders how he can systematically find the optimal way to cut any rod, no matter its length or the values associated with each segment. Your task is to help Theo by developing an algorithm that determines the optimal way to cut a rod of length L to maximize the total value of the pieces.

Sunny Studies

2. Tim the Beaver needs to study for finals, but the weather is getting warmer, and he is tempted to spend more time outside. Tim enjoys being outside when it's warm: if the temperature outside on a given day is t, his happiness increases by t after spending the day outside. If the temperature is negative, his happiness decreases. Tim will either study or play outside each day for n days leading up to the finals, but he cannot play outside two days in a row. To stay on top of coursework, Tim wants to determine the optimal days to study such that he maximizes his happiness. Given the weather forecast for the next n days, describe a dynamic programming algorithm that finds the days Tim should study to maximize his total happiness.
Hint: Design a $O(n)$-time algorithm that accounts for the constraints on outdoor play.

Diffing Data

3. Operating system Menix has a `diff` utility that can compare files. A *file* is an ordered sequence of strings, where the i^{th} string is called the i^{th} line of the file. A single *change* to a file is either: • the insertion of a single new line into the file; • the removal of a single line from the file; or • swapping two adjacent lines in the file.
In Menix, swapping two lines is cheap, as they are already in the file, but inserting or deleting a line is expensive. A `diff` from a file A to a file B is any sequence of changes that, when applied in sequence to A, will transform it into B. The conditions are:

   - *Any line may be swapped at most once.* • *Any pair of swapped lines must appear adjacent in both A and B.*

   Given two files A and B, each containing exactly n lines, describe an $O(kn + n^2)$-time algorithm to return a `diff` from A to B that minimizes the number of changes that are not swaps. Assume that any line from either file is at most k ASCII characters long.

Princess Plum and the Haunted Forest

4. Princess Plum is a video game character collecting mushrooms in a digital haunted forest. The forest is an $n \times n$ square grid where each grid square contains either a tree, mushroom, or is empty. Princess Plum can move from one grid square to another if the two squares share an edge, but she cannot enter a grid square containing a tree. Princess Plum starts in the upper-left grid square and wants to reach her home in the bottom-right grid square. The haunted forest is scary, so she wants to reach home via a quick path: a route from start to home that goes through at most $2n - 1$ grid squares (including start and home). If Princess Plum enters a square

with a mushroom, she will pick it up. Let k be the maximum number of mushrooms she could pick up along any quick path, and let a quick path be optimal if it collects k mushrooms along that path. Given a map of the forest grid, describe an $O(n^2)$-time algorithm to return the number of distinct optimal quick paths through the forest, assuming that some quick path exists.

Building Blocks

5. Saggie Mimpson is a toddler who likes to build block towers. Each of her blocks is a 3D rectangular prism, where each block b_i has a positive integer width w_i, height h_i, and length ℓ_i. She has at least three of each type of block. Each block can be oriented so that any opposite pair of its rectangular faces may serve as its top and bottom faces. The height of the block in that orientation is the distance between those faces. Saggie wants to construct a tower by stacking her blocks as high as possible. However, she can only stack an oriented block b_i on top of another oriented block b_j if the dimensions of the bottom of block b_i are strictly smaller[1] than the dimensions of the top of block b_j. Given the dimensions of each of her n blocks, describe an $O(n^2)$-time algorithm to determine the height of the tallest tower Saggie can build from her blocks.

Subset Sum Variants

6. Partition - Given a set of n positive integers A, describe an algorithm to determine whether A can be partitioned into two non-intersecting subsets A_1 and A_2 of equal sum, i.e., $A_1 \cap A_2 = \emptyset$ and $A_1 \cup A_2 = A$ such that $\sum_{a \in A_1} a = \sum_{a \in A_2} a$. Example: $A = \{1, 4, 3, 12, 19, 21, 22\}$ has partition $A_1 = \{1, 19, 21\}$, $A_2 = \{3, 4, 12, 22\}$.

7. Close Partition - Given a set of n positive integers A, describe an algorithm to find a partition of A into two non-intersecting subsets A_1 and A_2 such that the difference between their respective sums is minimized.

8. Can you adapt the subset sum to work with negative integers?

Peter Piper's Pepper Problem

9. Peter Piper wants a peck of pickled peppers, so he planted a patch of n pepper plants in a line. The i^{th} pepper plant possesses a known positive integer p_i representing the number of peppers. Peter Piper didn't plant his peppers properly, and now the plants are overgrown. He must decide which plants to prune to ensure unpicked plants can prosper. For each plant, Peter Piper will either leave it unpicked or prune it along with its peppers. However, pruning a plant requires pruning both its neighboring plants to provide space for unpicked plants. Peter Piper aims to prune as few plants as possible to allow maximum growth while minimizing the total number of peppers picked. He wants to minimize the sum of all picked peppers.

Coin Crafting

10. Ceal Naffrey is a thief in desperate need of money. He recently acquired n identical gold coins. Each coin has distinctive markings that would easily identify them as stolen if sold. However, using his amateur craftsman skills, Ceal can melt down gold coins to craft other golden objects. Ceal has a buyer willing to purchase golden objects at different rates but will only purchase one of any object. Ceal has compiled a

[1]If the bottom of block b_i has dimensions $p \times q$ and the top of block b_j has dimensions $s \times t$, then b_i can be stacked on b_j in this orientation if either: $p < s$ and $q < t$; or $p < t$ and $q < s$.

list of the n golden objects, listing both the positive integer **purchase price** the buyer would be willing to pay for each object and each object's positive integer **melting number**: the number of gold coins that would need to be melted to craft that object. Given this list, describe an efficient algorithm to determine the maximum revenue that Ceal could make by melting down his coins to craft into golden objects to sell to his buyer.

Arithmetic Parenthesization

11. A favorite discovery among students is how the order of operations affects the outcome of arithmetic expressions. Consider the arithmetic expression $a_0 *_1 a_1 *_2 a_2 \cdots *_{n-1} a_{n-1}$, where each a_i is an integer, and each operator $*_i \in \{+, \times\}$. Let's play with this concept by placing parentheses strategically. Determine where to place parentheses to maximize the evaluated expression. **Example:**

$$\begin{cases} 7 + 4 \times 3 + 5 \rightarrow ((7) + (4)) \times ((3) + (5)) = 88 \\ 7 + (-4) \times 3 + (-5) \rightarrow ((7) + ((-4) \times ((3) + (-5)))) = 15 \end{cases}$$

Remark: We allow negative integers!

Optimal Finger Assignment Problem

12. We will solve the problem of the *orchestra maestro*. Given a sequence of notes $t_0, t_1, \ldots, t_{n-1}$, where each note is to be played with a finger from the right hand, we aim to assign fingers in a way that minimizes the total difficulty of transitions between notes. Each finger f is taken from the set $\{1, 2, \ldots, F\}$, with $F = 5$ for most humans. For each transition from note t with finger f to note t' with finger f', a difficulty metric $d(t, f, t', f')$ is given. The difficulty includes various penalties, such as:

- Using a higher finger $f > f'$ on a lower note $t < t'$ is uncomfortable.
- Smooth transitions (legato) require $t \neq t'$; otherwise, a high penalty applies ($+\infty$).
- Avoid using weak fingers, such as $f' \in \{4, 5\}$, in succession.
- A transition between fingers $\{3, 4\}$ is especially annoying.

The goal is to assign fingers to notes to minimize the total difficulty across the sequence.

Treasureship!

13. The new boardgame Treasureship is played by placing 2×1 ships within a $2 \times n$ rectangular grid. Just as in regular battleship, each 2×1 ship can be placed either horizontally or vertically, occupying exactly 2 grid squares, and each grid square may only be occupied by a single ship. Each grid square has a positive or negative integer value, representing how much treasure may be acquired or lost at that square. You may place as many ships on the board as you like, with the score of a placement of ships being the value sum of all grid squares covered by ships. Design an efficient dynamic-programming algorithm to determine a placement of ships that will maximize your total score.

Wafer Circuit Matching

14. A start-up is developing a novel circuit design for parallel computing. The circuit is structured around a circular wafer containing n ports, evenly spaced along the perimeter. Each port functions either as a **power source** or a **computing unit**. To power the computing units, the design requires matching

each computing unit to a power source through **non-crossing wires** etched on the wafer's top surface. However, the following constraints apply:

- No two adjacent ports may be connected.
- Wires may not cross each other, and they must connect computing units (white) to power sources (black).
- If a computing unit connects to a power source that is too close, the connection could overload the circuit.

The goal is to *maximize the number of powered computing units* while adhering to the above constraints. You are tasked to determine the maximum number of valid non-crossing matches between computing units and power sources.

Exercise: Simplified Blackjack

15. We define a simplified version of the game **blackjack** between one player and a dealer. A *deck of cards* is represented as an ordered sequence $D = (c_1, \ldots, c_n)$, where each card c_i is an integer between 1 and 10 inclusive. (Unlike in real blackjack, aces always have a value of 1.) The game proceeds in *rounds.*

During each round:

- The dealer draws the first two cards from the deck: c_1 and c_2.
- The player draws the next two cards: c_3 and c_4.
- The player then chooses to either draw an additional card (a *hit*) or stop.

The player wins the round if the value of their hand (the sum of their drawn cards) is ≤ 21 and exceeds the value of the dealer's hand. Otherwise, the player loses the round. The game ends if fewer than 5 cards remain in the deck after a round.

The objective is to determine the *maximum number of rounds the player can win.* Given the known order of the cards in the deck, describe an $O(n)$-time algorithm to find the optimal strategy.

Exercise: Text Justification

16. Text Justification is the task of fitting a sequence of n space-separated words into a column of lines with a fixed width s, in order to minimize the total amount of white space between the words. Each word can be represented by its width $w_i < s$. A suitable way to minimize white space in each line is to minimize the **badness** of the line. Given a line that contains words from w_i to w_j, the badness of that line is defined as: $b(i, j) = (s - (w_i + \cdots + w_j))^3 \quad \text{if} \quad s \geq (w_i + \cdots + w_j)$, and $b(i, j) = \infty$ otherwise. A good text justification strategy would partition the words into lines such that the sum of badness across all lines is minimized. The cubic power term heavily penalizes large gaps, discouraging excessive white space between words. Microsoft Word, for example, uses a greedy algorithm to justify text by putting as many words as possible into a line before moving to the next. However, this strategy often results in poorly formatted lines. In contrast, LaTeX uses a dynamic programming approach to minimize white space efficiently. Your task is to describe an $O(n^2)$-time algorithm that fits n words into a column of width s in a way that minimizes the total badness over all lines.

Oil Well that Ends Well

17. The oil wells of tycoon Ron Jockefeller will produce m oil barrels this month. Ron has received n orders from potential buyers, where the i-th order specifies a willingness to buy a_i barrels for a total price

of p_i (not per barrel). Importantly, the price p_i may be negative, indicating that fulfilling the order is undesirable (during CoViD19 period, oil futures contract prices went negative: people were paying money to not accept delivery of oil because demand for oil had fallen dramatically and there was a shortage of places to store oil). Each order must either be filled completely or not at all, and each order can only be fulfilled once. Ron does not need to sell all his barrels but incurs a storage cost of s dollars for each unsold barrel. The challenge is to determine which orders Ron should fulfill to maximize his profit (which could potentially be negative). This optimization problem needs to be solved in $O(nm)$-time.

Exercise: Split Bowling

18. In old fashioned **Bowling** setting we have that a one-player played on a sequence of n pins, where pin i has integer value v_i (possibly negative). The player repeatedly knocks down pins in two ways: (a) Knock down a single pin i, providing v_i points or (b) Knock down two adjacent pins i and $i+1$, providing $v_i \cdot v_{i+1}$ points. Pins may be knocked down at most once, though the player may choose not to knock down some pins. A Bowling variant, **Split Bowling**, adds a third way the player can knock down two pins forming a split, specifically:

    - Knock down two pins i and $j > i+1$ if all pins in $\{i+1, \ldots, j-1\}$ between them have already been previously knocked down, providing $v_i \cdot v_j$ points.

    Describe an $O(n^3)$-time algorithm to determine the maximum score possible playing Split Bowling on a given input sequence of n pins.

Quarter Partition

19. **Given:** A set $A = \{a_0, a_1, \ldots, a_{n-1}\}$ containing n distinct positive integers, where the total sum of the elements is given by $m = \sum_{a_i \in A} a_i$. The task is to describe an $O(m^3 n)$-time algorithm that returns a partition of A into four subsets A_1, A_2, A_3, and A_4 such that $A_1 \cup A_2 \cup A_3 \cup A_4 = A$, and the maximum of their individual sums is minimized. Specifically, we aim to find a partition such that: $\min\left\{\max\left\{\sum_{a_i \in A_j} a_i \,\middle|\, j \in \{1,2,3,4\}\right\}\right\}$.

Protein Parsing

20. Prof. Leric Ander's lab performs experiments on DNA. After experimenting on any **strand of DNA** (a sequence of nucleotides, either A, C, G, or T), the lab will cut it up so that any useful protein markers can be used in future experiments. Ander's lab has compiled a list P of known protein markers, where each **protein marker** corresponds to a sequence of at most k nucleotides. A **division** of a DNA strand S is an ordered sequence $D = (d_1, \ldots, d_m)$ of DNA strands, where the ordered concatenation of D results in S. The **value** of a division D is the number of DNA strands in D that appear as protein markers in P. Given a DNA strand S and set of protein markers P, describe an $O(k(|P| + k|S|))$-time algorithm to determine the maximum value of any division of S.

When Greedy Fails, Dynamic Programming Prevails

21. We start by exploring a famous NP-complete problem in the worst case: the maximum weight independent set problem in two settings: (1) on a tree, and (2) on a sequence of integers.

(a) *Part 1: Maximum Weight Independent Set in a Tree*:
Given a set of vertices $I \subseteq V$ of an undirected graph $G(V, E)$, the set is called independent if no two vertices in I are connected by an edge. In this part, we consider a tree $T(V, E)$ where each vertex $v \in V$ has a weight $w(v) \geq 0$. The objective is to compute an independent set in T with the maximum total weight.

(b) *Part 2: Maximum Weight Independent Set in a Sequence*:
Given a sequence $X = (x_1, \ldots, x_n)$ of n positive integers, a set of indices $I \subseteq \{1, \ldots, n\}$ is called independent if for every pair of indices $i, j \in I$, $|i - j| > 1$, meaning that I contains no consecutive indices. The weight $W(J)$ of a set of indices $J \subseteq \{1, \ldots, n\}$ is defined as the sum of the corresponding values x_i, i.e.,
$$W(J) = \sum_{i \in J} x_i.$$
The goal is to compute an independent set of maximum weight for the given sequence X.