Week 7 Applied Sheet

Objectives: The applied sessions, in general, give practice in problem solving, in analysis of algorithms and data structures, and in mathematics and logic useful in the above.

Instructions to the class: You should actively participate in the class. The preparation problems are not assessed. We strongly recommend that everyone tries to solve the preparation problems before the applied session as you will benefit the most from the applied session if you come properly prepared. However, you can still attend the applied session if you have not solved those problems beforehand.

Instructions to Tutors: The purpose of the applied session is not to solve the practical exercises! The purpose is to check answers, and to discuss particular sticking points, not to simply make answers available.

Supplementary problems: The supplementary problems provide additional practice for you to complete after your applied session, or as pre-exam revision. Problems that are marked as **(Advanced)** difficulty are beyond the difficulty that you would be expected to complete in the exam, but are nonetheless useful practice problems as they will teach you skills and concepts that you can apply to other problems.

Problems

Problem 1. **(Preparation)** Implement the solution to the coin change problem described in the lectures. Your solution should return the number of coins needed, along with how many of each denomination are required.

(a) Use the bottom-up strategy to compute the solutions.

(b) Use the top-down strategy to compute the solutions.

Consult the notes if you are unclear on the difference between the two approaches.

Problem 2. Suppose that you are a door-to-door salesman, selling the latest innovation in vacuum cleaners to less-than-enthusiastic customers. Today, you are planning on selling to some of the n houses along a particular street. You are a master salesman, so for each house, you have already worked out the amount c_i of profit that you will make from the person in house i if you talk to them. Unfortunately, you cannot sell to every house, since if a person's neighbour sees you selling to them, they will hide and not answer the door for you. Therefore, you must select a subset of houses to sell to such that none of them are next to each other, and such that you make the maximum amount of money.

For example, if there are 10 houses and the profits that you can make from each of them are 50, 10, 12, 65, 40, 95, 100, 12, 20, 30, then it is optimal to sell to the houses $1,4,6,8,10$ for a total profit of \$252. Devise a dynamic programming algorithm to solve this problem, i.e. to return the maximum profit you can obtain.

(a) Describe in plain English, the optimal substructure present in the problem.

(b) Define a set of overlapping subproblems that are based on the optimal substructure.

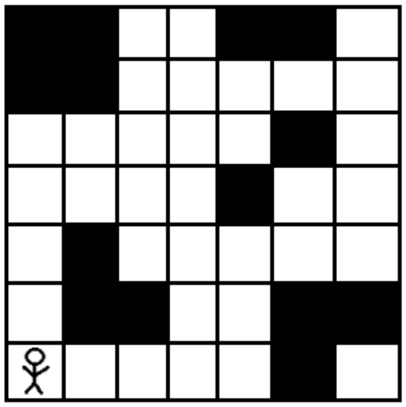
(c) What are the base case subproblems and what are their values?

(d) Write a recurrence relation that describes the solutions to the subproblems.

(e) Write pseudocode that implements all of this as a dynamic programming algorithm.

Problem 3. Extend your solution to Problem 2 so that it returns an optimal subset of houses to sell to, in addition to the maximum possible profit.

Problem 4. You find yourself curiously stranded on an $n \times n$ grid, unsure of how you got there, or how to leave. Some of the cells of the grid are blocked and cannot be walked through. Anyway, while you're here, you decide to solve the following problem. You are currently standing at the bottom-left corner of the grid, and are only able to move up (to the next row) and to the right (to the next column). You wonder, how many ways can you walk to the top-right corner of the grid while avoiding blocked cells? You may assume that the bottom-left and top-right cells are not blocked. For example, in the following grid, the answer is 19.



Write a dynamic programming algorithm that given a grid as input, counts the number of valid paths from the bottom-left cell to the top-right cell. Your algorithm should run in $O(n^2)$ time.

Problem 5. You somehow find yourself on yet another $n \times n$ grid, but this time, it is more exciting. Each cell of the grid has a certain non-negative amount of money on it! Denote the amount of money in the cell of row i, column j by $c_{i,j}$. You are standing on the bottom-left corner $(1,1)$ of the grid. From any cell, you can only move up (to the next row), or right (to the next column). What is the maximum amount of money that you can collect?

Given $c_{i,j}$ for every cell, describe a dynamic programming algorithm to solve this problem. Your algorithm should run in $O(n^2)$ time.

Problem 6. Consider a sequence $a_1, a_2, ..., a_n$ of length n. A *subsequence* of a sequence a is any sequence that can be obtained by deleting any of the elements of a. Devise a dynamic programming algorithm that finds the length of a *longest increasing subsequence* of a. That is, a longest possible subsequence that consists of elements in strictly increasing order. Your algorithm should run in $O(n^2)$ time.

For example, given the sequence $\{\mathbf{0}, 8, 4, 12, \mathbf{2}, 10, \mathbf{6}, 14, 1, \mathbf{9}, 5, 13, 3, \mathbf{11}, 7, \mathbf{15}\}$, the longest increasing subsequence is $\{0, 2, 6, 9, 11, 15\}$ of length 6 (shown in **bold** in the original sequence).

Problem 7. Consider a pair of sequences $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_m$ of length n and m. Devise a dynamic programming algorithm that finds the length of a *longest common subsequence* of a and b. A common subsequence is a sequence that is both a subsequence of a and a subsequence of b. Your algorithm should run in $O(nm)$. [Hint: This problem is very similar to the edit distance problem]

Problem 8. Consider a sequence of integers $a_1, a_2, ..., a_n$ of length n. Devise a dynamic programming algorithm that finds the subarray of a with maximum sum. A subarray or substring of a is a contiguous subsequence, ie. a subsequence consisting of consecutive elements.

(a) Your algorithm should run in $O(n^2)$ time.

(b) Your algorithm should run in $O(n)$ time. [Hint: Be greedy]

Problem 9. You are a young child, out for a walk with your parent. You are walking down a quiet street. Your parent is in a hurry, but you are more interested in collecting pretty rocks. Your parent has agreed to a compromise where they are happy for you to cross the street back and forth to pick up rocks, but only if you keep up.

Since you are a very organised child, you have determined the value to you of each rock along both sides of the streets, and represented this data in 2 arrays, `left` and `right`, of equal length. The i^{th} element in an array represents the value of the rock on the i^{th} square of pavement on that side of the street.

Because of your parent's rule about keeping up, you have the following restrictions to your movement. If you are on pavement square i on one side of the street you can either:

- Pick up the rock on square i on that side, and move to square $i+1$ on the same side.
- Cross the street diagonally to square $i+1$ on the other side (but not pick up the rock).

You start on pavement 0 on the left side of the street.

Devise a dynamic programming algorithm to determine the maximum value of rocks that you can obtain. Your algorithm should run in O(n), where n is the length of `left`.

Supplementary Problems

Problem 10. A ferry that is going to carry cars across the bay has two lanes in which cars can drive onto. Each lane has a total length of L. The cars that wish to board the ferry line up in a single lane, and are directed one by one onto one of the two lanes of the ferry until the next car cannot fit in either lane. Depending on which lanes the cars are directed onto, fewer or more cars may fit on the ferry. Given the lengths of each of the cars (which will not be longer than the length of the ferry L), and assuming that the distance between cars when packed into the ship is negligible, write a dynamic programming solution that determines the maximum number of cars that can be loaded onto the ferry if distributed optimally.

(a) Solve the problem in whatever time complexity you can (without resorting to brute force).

(b) Improve your solution to $O(nL)$ time complexity (if it is not already).

For example, suppose that the car lengths are 2, 2, 7, 4, 9, 8, 1, 7, 3, 3 and the ferry is 20 units long. An optimal solution is to load 8 cars in lanes arranged like 2, 2, 7, 9 and 4, 8, 1, 7.

Problem 11. You and your friend are going to play a game. You start with a row of n coins with values $a_1, a_2, ..., a_n$. You will each take turns to remove either the first or last coin. You continue until there are no coins left, and your score is the total value of the coins that you removed. You will make the first move of the game.

(a) Show that the greedy strategy in which you simply pick the most valuable coin every time is not optimal.

(b) Devise a dynamic programming algorithm that determines the maximum possible score that you can achieve if both you and your friend make the best possible moves.

For example, given the coins 6, 9, 1, 2, 16, 8, the maximum value that you can achieve when going first is 23.

Problem 12. A sequence is a *palindrome* if it is equal to its reversal. For example, `racecar` and $5,3,2,3,5$ are palindromes. Given a sequence $a_1, a_2, ..., a_n$ of length n, solve the following problems.

(a) Devise a dynamic programming algorithm that determines the length of the longest palindromic **subsequence** of a. Your algorithm should run in $O(n^2)$ time.

(b) Devise an algorithm that determines the length of the longest palindromic **substring** of a. Your algorithm should run in $O(n^2)$ time. (You do not have to use dynamic programming).

(c) **(Advanced)** Devise a dynamic programming algorithm that determines the length of the longest palindromic **substring** of a. Your algorithm should run in $O(n)$ time.

Problem 13. Consider a pair of sequences $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_m$ of length n and m. A supersequence of a sequence a is a sequence that contains a as a subsequence. Devise an algorithm that finds the length of a *shortest common supersequence* of a and b. A common supersequence is a sequence that is both a supersequence of a and a supersequence of b. Your algorithm should run in $O(nm)$.

Problem 14. Suppose that I give you a string S of length n and a list L consisting of m words with length at most n. Devise a dynamic programming algorithm to determine the minimum number of strings from L that must be concatenated to form the string S. You may use a word from L multiple times. Your algorithm should run in $O(n^2 m)$ time.

Problem 15. Given three strings A, B, and C of length n, m, and $n+m$ respectively, determine whether C can be formed by interleaving the characters of A and B. C is said to be interleaving A and B if there are subsequences C_1 and C_2 of C such that they are complementary (i.e., each index of C is in either C_1 or C_2, and there are no common indices in C_1 and C_2) and that $C_1 = A$ and $C_2 = B$. For example, string "XXXXZY" interleaves strings "XXY" and "XXZ", but string "XXY" does not interleaves strings "YX" and "X". Your algorithm should run in $O(nm)$ time.

Problem 16. Given an $n \times n$ matrix A, your task is to find the submatrix with the maximum possible sum. A submatrix of a matrix A is a matrix consisting of the elements of some contiguous set of rows and columns of A.

1. Devise a dynamic programming algorithm that runs in $O(n^4)$ time.
2. **(Advanced)** Devise an algorithm that runs in $O(n^3)$ time. [Hint: Use your solution to Problem 8(b)]

Problem 17. Consider the problem of nicely justifying text on a page. Given a sequence of word lengths $l_1, l_2, ..., l_n$ and a page width w, we wish to figure out the best way to justify the words on a page of width w. We define the quality of a line containing the words from index i to j inclusive by the following scoring function

$$\texttt{score}[i..j] = \begin{cases} \infty & \text{if } \text{sum}(l[i..j]) > w \\ (w - \text{sum}(l[i..j]))^3 & \text{otherwise} \end{cases}$$

The aim is to split the words into lines such that the total score of all the lines is minimised. Devise a dynamic programming algorithm that computes the minimum possible score. Implement your solution in Python and extend it such that it also prints the optimal justification for a given list of words. Your algorithm should run in $O(n^2)$ time to compute the minimum score, and $O(T)$ time to print the justified words, where $T = \text{sum}(l[1..n])$.

Problem 18. Along a particular river, there are $2n$ houses, exactly n of them on each side of the river. The owner of each house has a friend in one of the houses on the other side of the river. Each person is friends with exactly one other person. The house owners are tired of having to swim across the river to meet their friends each day, so they are going to connect some of the houses by bridges. Each person would like to have a bridge that connects their house directly to their friend's house, but this is not always possible since some bridges would have to cross. What is the maximum number of bridges that can be built between friends' houses without any of them crossing? Devise a dynamic programming algorithm to solve this problem. Your algorithm should run in $O(n^2)$ time.

Problem 19. You have access to a set of n different types of cardboard boxes. Each type of box is rectangular, with a particular length, width, and height. Your goal is to build a tower of boxes that is as tall as possible. For stability purposes, you can only stack a box on top of another if the dimensions of the base of the top box are strictly smaller than the dimensions of the top of the lower box. You may rotate the boxes in any way that you wish, and may use as many of the same type of box as necessary (but each layer can only have one box). For example, you can stack a box with a 2×3 base on top of a box with a 4×3 base (by rotating the box so that $2 < 3$ and $3 < 4$), but you cannot stack a 2×3 box on top of a 3×3 box. Write a dynamic programming algorithm that computes the height of the tallest possible tower. Your algorithm should run in $O(n^2)$ time.

Problem 20. Given a sequence of n integers $a_1, a_2, ..., a_n$, and two integers k and d, determine whether it is possible to partition the integers a_i into groups such that each group contains at least k items, and for each pair of items a_i, a_j in the same group, $|a_i - a_j| \leq d$. Come up with a dynamic programming algorithm for this problem.

(a) Find an algorithm that runs in $O(n^2)$.

(b) **(Advanced)** Find an algorithm that runs in $O(n \log(n))$.

(c) **(Advanced)** Assume that the sequence (a_i) is sorted. Find an algorithm that runs in $O(n)$.

Problem 21. (Advanced) Given an unparenthesised boolean expression containing n literals (ie. an expression containing n symbols T (**True**) or F (**False**), each separated by $\wedge$ (and), or $\vee$ (or)), there are many ways to add parenthesis to the expression to change the order in which it is evaluated. For example, given the expression $T \vee T \wedge F$, there are two orders in which we could evaluate the operands:

$$(T \vee T) \wedge F, \qquad \text{or} \qquad T \vee (T \wedge F).$$

Our goal is to count the number of different orders of evaluating the expression such that it evaluates to **True**. In the example above, of the two orders, the second evaluate to **True**, so the answer is 1. Devise a dynamic programming algorithm to solve this problem in $O(n^3)$ time.

Problem 22. Consider a distributed computer network consisting of n computers. The computers are connected together such that there is exactly one path in the network between any pair of computers (in other words, the network is a tree). One of the computers contains an important message that needs to be sent to every other computer. In a single *round*, each computer that knows the message can send a copy of the message to one of the computers that it is connected to. Write a dynamic programming algorithm that computes the minimum number of rounds required to distribute the message to every computer if done optimally. Your algorithm should run in $O(n)$ time.

Problem 23. (Advanced) You wake up early in the morning and step outside to head to your favourite class (Algorithms and Data Structures of course), only to find that it is raining! Consider the path from your home to class to be a straight line from position 0 to position C. There are n segments on which it is currently raining. Each segment can be described by an interval $[l, r]$ such that it is raining at all points between l and r inclusive. Conveniently, along the way there are m umbrellas on the ground that you may pick up. Each umbrella has a particular location and a particular weight. You can only move through the rain if you are currently carrying an umbrella (you do not want your laptop to get wet). However, since the umbrellas are heavy, you would like to carry them for as little time as possible. In particular, if you carry an umbrella of weight w for a distance of d, you will use up $w \times d$ energy. You may put down an umbrella at any time, or swap your current umbrella with another umbrella (including while in the rain, if an umbrella is located in the rain). Given the location of the rain and the locations and weights of the umbrellas, what is the minimum amount of energy that you must expend to get to class without getting wet? Assume that all of the raining segments do not intersect. Write a dynamic programming algorithm that runs in $O((n+m)^2)$ time.

Problem 24. Let us define a *pattern* string as follows. A pattern string P is a string consisting of lowercase letters, stars (*) and full-stops (.). We say that a text string T of lowercase letters *matches* the pattern P if we can form T from P by replacing the full-stops in P with any lowercase character (each full-stop may be replaced with a different character), and replacing the stars by any (possibly empty) string of lowercase characters (each star may be replaced by a different string). Devise a dynamic programming algorithm to determine whether a given text T of length n matches a given pattern P of length m.

(a) Your algorithm should run in $O(n^2 m)$ time.

(b) **(Advanced)** Your algorithm should run in $O(nm)$ time.