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 Planting Trees

This problem will guide you through the process of writing a dynamic programming algorithm.

You have a garden and want to plant some apple trees in your garden, so that they produce as many apples as possible. There are n adjacent spots numbered 1 to n in your garden where you can place a tree. Based on the quality of the soil in each spot, you know that if you plant a tree in the ith spot, it will produce exactly x_i apples. However, each tree needs space to grow, so if you place a tree in the ith spot, you can't place a tree in spots $i-1$ or $i+1$. What is the maximum number of apples you can produce in your garden?

(a) Give an example of an input for which:

- Starting from either the first or second spot and then picking every other spot (e.g. either planting the trees in spots $1, 3, 5\ldots$ or in spots $2, 4, 6\ldots$) does not produce an optimal solution.
- The following algorithm does not produce an optimal solution: While it is possible to plant another tree, plant a tree in the spot where we are allowed to plant a tree with the largest x_i value.

(b) To solve this problem, we'll think about solving the following, more general problem: "What is the maximum number of apples that can be produced using only spots 1 to i?". Let $f(i)$ denote the answer to this question for any i. Define $f(0) = 0$, as when we have no spots, we can't plant any trees. What is $f(1)$? What is $f(2)$?

(c) Suppose you know that the best way to plant trees using only spots 1 to i does not place a tree in spot i. In this case, express $f(i)$ in terms of x_i and $f(j)$ for $j < i$. (Hint: What spots are we left with? What is the best way to plant trees in these spots?)

(d) Suppose you know that the best way to plant trees using only spots 1 to i places a tree in spot i. In this case, express $f(i)$ in terms of x_i and $f(j)$ for $j < i$.

(e) Describe a linear-time algorithm to compute the maximum number of apples you can produce. (Hint: Compute $f(i)$ for every i. You should be able to combine your results from the previous two parts to perform each computation in $O(1)$ time).

Solution:

(a) For the first algorithm, a simple input where this fails is $[2, 1, 1, 2]$. Here, the best solution is to plant trees in spots 1 and 4. For the second algorithm, a simple input where this fails is $[2, 3, 2]$. Here, the greedy algorithm plants a tree in spot 2, but the best solution is to plant a tree in spots 1 and 3.

(b) $f(1) = x_1$, $f(2) = \max\{x_1, x_2\}$

(c) If we don't plant a tree in spot i, then the best way to plant trees in spots 1 to i is the same as the best way to plant trees in spots 1 to $i-1$. Then, $f(i) = f(i-1)$.

(d) If we plant a tree in spot i, then we get x_i apples from it. However, we cannot plant a tree in spot $i-1$, so we are only allowed to place trees in spots 1 to $i-2$. In turn, in this case we can pick the best way to plant trees in spots 1 to $i-2$ and then add a tree at i to this solution to get the best way to plant trees in spots 1 to i. So we get $f(i) = f(i-2) + x_i$.

(e) Initialize a length n array, where the ith entry of the array will store $f(i)$. Fill in $f(1)$, and then use the formula $f(i) = \max\{f(i-1), x_i + f(i-2)\}$ to fill out the rest of the table in order. Then, return $f(n)$ from the table.

2 Change making

You are given an unlimited supply of coins of denominations $v_1, \ldots, v_n \in N$ and a value $W \in N$. Your goal is to make change for W using the minimum number of coins, that is, find a smallest set of coins whose total value is W.

1. Design a dynamic programming algorithm for solving the change making problem. What is its running time?

2. You now have the additional constraint that there is only one coin per denomination. Does your previous algorithm still work? If not, design a new one.

Solution:

1. For $0 \le w \le W$, define

$$f(w) = \text{the minimum number of coins needed to make a change for } w.$$

It satisfies

$$f(w) = \min \begin{cases} 0 & \text{if } w = 0, \\ 1 + \min_{j:\, v_j \le w} f(w - v_j) & \\ \infty & \end{cases}$$

The answer is $f(W)$, where ∞ means impossible. It takes $O(nW)$ time.

2. For $0 \le i \le n$ and $0 \le w \le W$, define

$$f(i, w) = \text{the minimum number of coins (among the first } i \text{ coins) needed to make a change for } w, \text{ having one coin per denomination.}$$

It satisfies

$$f(i, w) = \min \begin{cases} 0 & \text{if } i = 0 \text{ and } w = 0, \\ f(i-1, w) & \text{if } i > 0, \\ 1 + f(i-1, w - v_i) & \text{if } i > 0 \text{ and } v_i \le w, \\ \infty & \end{cases}$$

The answer to the problem is $f(n, W)$. It takes $O(nW)$ time.

3 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 how many ways one can interpret s. 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. Your algorithm should run in time at most $O(nm\ell)$ where n is the length of the input bit string s, m is the number of symbols, and ℓ is an upper bound on the length of the bit strings representing symbols.

Please give a 3-part solution.

Solution:

Main Idea: We define our subproblems as follows: let $A[i]$ be the number of ways of interpreting the string $s[:i]$. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$A[i] = \sum_{\substack{\text{symbol } a \text{ in } d \\ s[i-\text{length}(d[a])+1:i]=d[a]}} A[i - \text{length}(d[a])].$$

Note here that we set $A[0] = 1$. 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 with zeros.
  Let $A[0] = 1$
  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**
        $A[i] += A[i - \text{length}(d[a])]$
  return $A[n]$

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

Base Case. When $i = 0$, there is only one way to interpret $s[:0]$ (the empty string). Hence, $A[0] = 1$.

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, there are exactly $A[i - \text{length}(d[a])]$ of the latter. 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. Summing this up 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 $A[n]$, the number of ways to interpret s.

Runtime Analysis: There are n iterations of the outer for loop and m 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 $O(nm\ell)$.

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

4 String Shuffling

Let x, y, and z be strings. We want to know if z can be obtained only from x and y by interleaving the characters from x and y such that the characters in x appear in order and the characters in y appear in order. For example, if $x =$ **efficient** and $y =$ **ALGORITHM**, then it is true for $z =$ **effALGiORciIenTHMt**, but false for $z =$ **efficientALGORITHMS** (extra characters), $z =$ **effALGORITHMicien** (missing the final t), and $z =$ **effOALGRicieITHMnt** (out of order). How can we answer this query efficiently? Your answer must be able to efficiently deal with strings with lots of overlap, such as $x =$ **aaaaaaaaaab** and $y =$ **aaaaaaaac**.

1. Design an efficient algorithm to solve the above problem and state its runtime.

2. Consider an iterative implementation of our DP algorithm in part (a). Naively if we want to keep track of every solved sub-problem, this requires $O(|x||y|)$ space (double check to see if you understand why this is the case). How can we reduce the amount of space our algorithm uses?

Solution:

1. First, we note that we must have $|z| = |x| + |y|$, so we can assume this. Let $S(i, j)$ be true if and only if the first i characters of x and the first j characters of y can be interleaved to make the first $i + j$ characters of z. Then x and y can be interleaved to make z if and only if $S(|x|, |y|)$ is true.

   For the recurrence, if $S(i, j)$ is true then either $z_{i+j} = x_i$, $z_{i+j} = y_j$, or both. In the first case it must be that the first $i - 1$ characters of x and the first j characters of y can be interleaved to make the first $i + j - 1$ characters of z; that is, $S(i - 1, j)$ must be true. In the second case $S(i, j - 1)$ must be true. In the third case we can have either $S(i - 1, j)$ or $S(i, j - 1)$ or both being true. This yields the recurrence:

$$S(i, j) = (S(i - 1, j) \wedge (x_i = z_{i+j})) \vee (S(i, j - 1) \wedge (y_j = z_{i+j}))$$

   The base case is $S(0, 0) = T$; we also set $\forall i \in [0, |x|], S(i, -1) = F$ and $\forall j \in [0, |y|], S(-1, j) = F$. The running time is $O(|x||y|)$.

   Somewhat naively if we'd like an iterative solution, we can keep track of the solutions to all subproblems with a 2D array where the entry at row i, column j is $S(i, j)$. If we iterate over this array row by row, going left to right, we'll always be able to fill in the next entry using values we've already computed.

   Notice, however, that to compute any entry, we only really need the infomration in the previous row, and the current row we're filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing us from $O(m * n)$ space to $O(m)$ space.

2. We can keep track of the solutions to all subproblems with a 2D array of size $|x||y|$ where the entry at row i, column j is $S(i, j)$. If we iterate over this array row by row, going left to right, we'll always be able to fill in the next entry using values we've already computed.

   Notice, however, that to compute any entry, we only really need the infomration in the previous row, and the current row we're filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing from $O(|x||y|)$ space to $O(\min(|x|, |y|))$ space.