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 \( i \)th spot, it will produce exactly \( x_i \) apples. However, each tree needs space to grow, so if you place a tree in the \( i \)th 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... or in spots 2, 4, 6...) 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 $i$th 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 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 $AB$ or as $A$. Note here that we set $A[0] = 1$. Our algorithm simply computes the above formula in a straightforward manner.

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 $s[0 : i]$. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$A[i] = \sum_{\text{symbol } a \text{ in } d} A[i - \text{length}(d[a])]$$

Note here that we set $A[0] = 1$. Our algorithm simply computes the above formula in a trivial manner.

**Pseudocode:**

```plaintext
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 $s[0 : 0]$ (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 $s[0 : i]$ 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 $s[0 : i]$ 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 $s[0 : i - \text{length}(d[a])]$. From our inductive hypothesis, there are exactly $A[i - \text{length}(d[a])]$ of the latter. On the other hand, if the suffix of $s[0 : i]$ differs from $d[a]$, then there is no interpretation of $s[0 : i]$ 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 $s$. 

2
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 \( O(nmf) \).

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.

3 Equivalent Strings

We are given two strings \( A, B \) of length \( n, m \) respectively. These two strings can contain English characters a to z, as well the special character \( ? \). We say \( A \) and \( B \) are equivalent if it is possible to replace every instance of \( ? \) with a (possibly empty) string of English characters, such that the resulting strings (containing only English letters) are the exact same.

For example, “ab?” is equivalent to “a?cd”, since with the above replacements we can transform both strings into “abcd”. Similarly, “a?bc” is equivalent to “abc”, since we are allowed to replace \( ? \) with the empty string.

Give an efficient dynamic programming algorithm to determine if two strings are equivalent. Give a three-part solution.

Solution:

Main idea: Let \( E(i, j) \) be True if the first \( i \) characters of \( A \) and \( j \) characters of \( B \) are equivalent, and False otherwise.

As a base case, \( E(0,0) = True \) and \( E(i,0) = True \) if and only if the first \( i \) characters of \( A \) are \( ? \) for all \( i > 0 \). We initialize all \( E(0,j) \) analogously.

For \( E(i,j) \) where \( i, j > 0 \) we set it to be the OR of the following cases:

1. \( E(i - 1, j - 1) \) AND \( (A[i] = B[i]) \)
2. \( (E(i, j - 1) OR E(i - 1, j)) \) AND \( (A[i] =? OR B[j] =?) \)

Correctness: For the base cases, two empty strings are of course equivalent, and the only non-empty strings that an empty string can be equivalent to are those consisting only of \( ? \) characters.

For the remaining values, \( A[1 : i] \) and \( B[1 : j] \) are equivalent if and only if one of the following possibilities happens:

- The last character of \( A[1 : i] \) is matched to the last character of \( B[1 : j] \) and vice-versa, and \( A[1 : i - 1], B[1 : j - 1] \) are also equivalent.
- The last character of \( A[i] \) is \( ? \). In this case, if \( A[1 : i - 1] \) and \( B[1 : j] \) are equivalent, then so are \( A[1 : i] \) and \( B[1 : j] \) since we can replace \( ? \) with the empty string. The same is true if \( B[j] =? \) and \( A[1 : i] \) and \( B[1 : j - 1] \) are equivalent.

The first possibility is covered by our first case, and the second and third possibility are covered by our second case.

Runtime analysis: There are \( O(nm) \) values of \( E(i,j) \) to compute, each takes \( O(1) \) time, so the runtime is \( O(nm) \).

Partial credit \( O(nm(n + m))^2 \)-time algorithm:

A slower but still correct solution is to use the same \( E(i,j) \) and base cases as before, and use the following recurrence relation: \( E(i,j) \) is the OR of:

For \( E(i,j) \) where \( i, j > 0 \) we set it to be the OR of the following cases:
1. \( E(i-1, j-1) \) AND \( A[i] = B[i] \)

2. There exists \( k \) such that \( E(i-1, j-k) \) AND \( A[i] =? \).

3. There exists \( k \) such that \( E(i-k, j-1) \) AND \( B[j] =? \).

This takes \( O(nm(n + m)) \) time since there are potentially \( n + m \) subproblems that need to be checked to compute each \( E(i,j) \).