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.

Alg 1: $[\textcircled{2}, 1, 1, \textcircled{2}]$ Alg 2: $[\textcircled{9}, \underline{10}, \textcircled{8}]$ (greedy)

(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 $f(2)$?

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

(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?)

same as best way to plant trees just using spots $1 \to i-1$

$f(i) = f(i-1)$

(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 $j < i$.

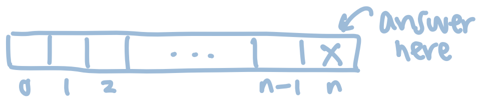
plant tree in spot i, can't use spot i-1

$f(i) = x_i + f(i-2)$

(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).

$f(i) = \max\{f(i-1), x_i + f(i-2)\}$

either plant a tree in spot i or don't

DP array: 

Overall Alg runtime: $O(n)$

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?

* decrease problem size — less coins? smaller W?

$f(w)$ = min number of coins needed to make change for w

$f(w) = 1 + \min_{v_i \leq w} f(w - v_i)$ or $f(w) = \infty$ if impossible (no $v_i \leq w$)

Base case: $f(0) = 0$

Answer at $f(W)$

Runtime: $O(nW)$

- W subproblems
- each subproblem takes $O(n)$ time $(|N| = n)$

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.

No, previous algorithm reuses coins

No longer enough to just keep track of the value w, also need to track which coins are available.

$f(i, w)$ = min number of coins among first i coins needed to make change for w, each coin used at most once

Either use the ith coin or don't.

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

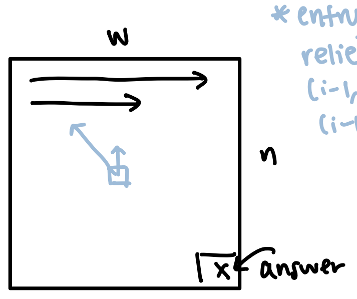
$f(i,w) = \infty$ if impossible (e.g. $i = 0$ and $w > 0$)

Base case: $f(0,0) = 0$

Answer at $f(n, W)$ ← can use all n coins to make W

Runtime: $O(nW)$

- nW subproblems
- each subproblem takes $O(1)$ time



* entry (i,w) relies on $(i-1, w)$ and $(i-1, w - v_i)$

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.

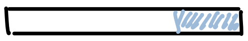
Main Idea:

- smaller problem $\Rightarrow$ smaller bit string

A[i] = number of ways to interpret string s[:i] (first i bits)

For each subproblem, check all valid symbol encodings for the last bits

k = key in d

s[:i] 

d[k]

$$A[i] = \sum_{\substack{k \in d \\ \text{such that} \\ s[i-\text{len}(d[k]):i] = d[k]}} A[i-\text{len}(d[k])]$$

Base case: A[0] = 1

Proof of correctness:

Base case: one way to interpret empty string

Induction:

Assume A[0] ... A[i-1] contains the right value.

Will show recurrence is correct for A[i]

Let's partition s[:i] into symbols $a_1 a_2 \ldots a_k$

If suffix of s[:i] a_k = d[k], s[:i] can have interpretation $a_1 a_2 \ldots a_{k-1}$ d[k]

possible interpretations : s[:i-len(d[k])]

We sum up all possible interpretations with all possible d[k] suffixes.

By inductive hypothesis, s[:i-len(d[k])] has correct count.

Runtime:

Each subproblem takes O(mℓ) time. There are n subproblems.

Overall runtime O(nmℓ)

tinyurl.com/cs170disc6

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.

First check $|z| = |x| + |y|$.

$S[i,j]$ = true if first i characters from x and first j characters from y can form first $i+j$ characters from z

$(i+j)^{th}$ character from z can come from either x or y

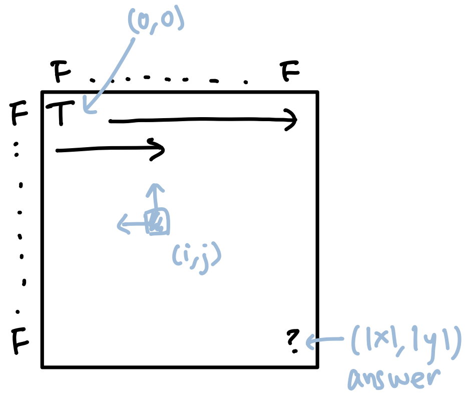
$$S[i,j] = [S(i-1,j) \wedge (x_i = z_{i+j})] \vee [S(i,j-1) \wedge (y_j = z_{i+j})]$$

Base case: $S[0,0]$ = True

*just having this True base case not enough

$\forall i \in [0,|x|]$ $S(i,-1)$ = False
$\forall j \in [0,|y|]$ $S(-1,j)$ = False

This means matrix dimensions are:
$(|x|+1)$ by $(|y|+1)$



Runtime: $|x||y|$ subproblems, each take $O(1)$ time.
$O(|x||y|)$

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?

$S[i,j]$ only need entries $(i-1,j)$ and $(i,j-1)$

Can store just rows i and $i-1$ or columns j and $j-1$

Space reduced to $O(\min\{|x|,|y|\})$