Section 5: Dynamic Programming

1. Going to parties

You are planning your calendar for n days, where every day, there is a party that you can go to. Every day, you can choose to go to that party or stay in and catch-up on sleep. If you party, you will enjoy yourself. But you can only party for two consecutive days — if you party three days in a row, you'll fall too far behind on sleep and miss class. Luckily, you have an excellent social sense, so you know exactly how much you will enjoy any of the parties, and you have assigned each day a positive integer happiness score in an array $H[1\,..\,n]$. You get 0 happiness if you do not go to the party. Maximize the sum of your happiness for these n days, while not going out for more than two consecutive days.

(a) Write a summary for this problem.

Dynamic programming is difficult because examples rarely help until you know your subproblems. You need to abstractly analyze the problem to determine subproblems first.

You've seen a strategy in lecture before. Here it is:

1. Let OPT(j) = the optimal solution on inputs up to j.
2. Divide OPT(j) into cases based on what to do with the jth element.
3. For each case, can you use OPT$(j-1)$ to handle it? Why or why not?
4. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

(b) Now you try. Say how to adapt each step to the party problem at hand.

(i) Let OPT(j) = the optimal solution on inputs up to j.

(ii) Divide OPT(j) into cases based on what to do with the jth element.

(iii) For each case, can you use OPT$(j-1)$ to handle it? Why or why not?

(iv) If you could not handle it, what additional information would help you? What kinds of subproblems are these?

(c) Now that you know what form you want your subproblems to take, try this example to flesh out the details and convince yourself that it works.

[9, 2, 3, 8, 6, 6, 4, 7, 1]

(d) Write the recurrence relation. Don't forget the base cases.

In dynamic programming, the pseudocode will end up being a fairly direct translation of the recurrence, so we'll do the proof first. In this class, we will focus on just proving the recursive case. A complete formal proof is, of course, induction.

(e) Prove your recurrence to be correct.

Now, on to the implementation details. Even though we're use a recurrence relation, do not call your function recursively! Calling the function recursively can lead to blowing up the running time, so we need to consider how to remember the solutions to subproblems.

(f) (i) State the parameters for your subproblems and what kind of structure you will use to store them.

(ii) Describe the order for evaluating your subproblems.

(g) Write the pseudocode for your iterative algorithm.

(h) What is the running time of your algorithm?

Lastly, you will sometimes be asked to return the optimal object, rather than the optimal value. However, doing it naively can cost you. You've seen this in lecture already, but a brief review:

- A first idea might be to track the optimal object at each j, instead of the optimal value. In today's problem, this will use $O(n^2)$ total time and space, very bad!
- Thus, we usually try to backtrack to find the optimal object after finding the optimal value. You may need to leave some hints for yourself to know where to go.

(i) How would you modify the algorithm if you were asked to return the optimal party schedule?

The following problems will not be covered in section, but may be useful to think about.
We recommend trying them by yourself first. Solutions will be posted in the evening.

2. Longest common subsequence

Given two strings, $s = s_1 \dots s_m$ with length m and $t = t_1 \dots t_n$ with length n, find the length of their longest common subsequence. (A subsequence may not be contiguous. That is, one finds a subsequence by taking any subset of the indices, and putting together the letters at those indices in their original order.)

Here are a few examples:

- **Input**: $s =$ `backs`, $t =$ `arches`
  Solution: The longest common subsequence is `acs`, so the output should be 3.
- **Input**: $s =$ `skaters`, $t =$ `hated`
  Solution: The longest common subsequence is `ate`, so the output should be 3.

(a) Define, in English, the quantities that you will use for your recursive solution.

(b) Given a recurrence relation for the quantities you have defined.

(c) Argue for the correctness of your recurrence relation.

(d) Describe the parameters for the subproblems in your recursion and how you will store their solutions.

(e) In what order can you evaluate them iteratively?

(f) Write the pseudocode for your iterative algorithm.

(g) Analyze the running time of your algorithm.

3. Short disjoint subarrays

You are given an array of positive integers $A[1\,..\,n]$ and an positive integer target t. You need to find the minimum total length of k disjoint (non-overlapping) contiguous subarrays of A that each have a sum of their elements equal to the target. If there are not k such subarrays, return ∞.

Here are a few examples:

- **Input**: $k = 4$, $t = 2$, and the array $[1, 1, 8, 2, 3, 2, 1, 1, 2]$
  Solution: We could do $[[1, 1], 8, [2], 3, [2], 1, 1, [2]]$, so the output should be 5. (not the only solution)
- **Input**: $k = 2$, $t = 4$, and the array $[2, 1, 2, 2, 2]$
  Solution: No two disjoint, contiguous subarrays each have sum 4, so the output should be ∞.
- **Input**: $k = 3$, $t = 4$, and the array $[1, 3, 1, 2, 1, 4, 2, 2]$
  Solution: We could do $[[1, 3], 1, 2, 1, [4], [2, 2]]$, so the output should be 5.

(a) Define, in English, the quantities that you will use for your recursive solution.

(b) Given a recurrence relation for the quantities you have defined.

(c) Argue for the correctness of your recurrence relation.

(d) Describe the parameters for the subproblems in your recursion and how you will store their solutions.

(e) In what order can you evaluate them iteratively?

(f) Write the pseudocode for your iterative algorithm.

(g) Analyze the running time of your algorithm.