Section 5: Dynamic Programming

1. Lots of fun, with a normal sleep schedule

You are planning your social calendar for the month. For each day, you can choose to go do a social event or stay in and catch-up on sleep. If you go to a social event, you will enjoy yourself. But you can only go out for two consecutive days – if you go to a social event 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 social events, and have assigned each day an (integer) numerical happiness score (and you know you get 0 enjoyment from staying in and catching up on sleep). You have an array $H[]$ which gives the happiness you would get by going out each day. Your goal is to maximize the sum of the happinesses for the days you do go out, while not going out for more than two consecutive days.

1.1. Read and understand the problem

Read the problem and answer the usual quick-check-questions

- Are any words in the problem technical terms? Do you know them all?
- What is the input type?
- What is the output type?

1.2. Generate Examples

Generate at least two examples along with their correct answers. It often helps at this point to ask yourself "what would a greedy algorithm be?" and design a counter-example for that algorithm

1.3. Write the Dynamic Program

Since we know we're writing DP algorithms this week, we're going to make these steps a bit more specific to the DP process.

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation?

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

1.4. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

(b) Describe a filling order for your memoization structure.

(c) State and justify the running time of an iterative solution.

More Problems!

2. Longest Increasing Subsequence AGAIN

We've already seen a recurrence for Longest Increasing Subsequence. Let's write another!

As before, $[10, -2, 5, 0, 3, 11, 8]$ has a longest increasing subsequence of 4 elements: $[-2, 0, 3, 8]$

2.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation? To make sure you get a different solution than the one from class, you should ask yourself to answer the question "what's the longest increasing subsequence where the first included element is the one at index i, and how would I find that?"

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

2.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

(b) Describe a filling order for your memoization structure.

(c) State and justify the running time of an iterative solution.

3. Longest Common Subsequence

Given two strings s with length m and t with length n, find the length of their longest common subsequence. A subsequence is a (possibly non-contiguous) substring.

Examples:

Input s="backs", t="arches": the longest common subsequence is "acs", so the output should be 3.

Input s="skaters", t="hated": the longest common subsequence is "ate", so the output should be 3.

3.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation?

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

3.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

(b) Describe a filling order for your memoization structure.

(c) State and justify the running time of an iterative solution.

4. k Minimum Disjoint Subarrays, Each with Target Sum t

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

Examples:

Input $k = 4$, $t = 2$, and the array $[1, 1, 8, 2, 3, 2, 1, 1, 2]$: the four smallest disjoint subarrays whose elements each sum to 2 are $[[1, 1], 8, [2], 3, [2], 1, 1, [2]]$, so the output should be 5.

Input $k = 2$, $t = 5$, and the array $[2, 2, 2, 2]$: there aren't two subarrays whose elements each sum to 5, so the output should be ∞.

Input $k = 3$, $t = 4$, and the array $[1, 3, 1, 2, 1, 4, 2, 2]$: the three smallest disjoint subarrays whose elements each sum to 4 are $[[1, 3], 1, 2, 1, [4], [2, 2]]$ so the output should be 5.

4.1. Write the Dynamic Program

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation?

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

4.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

(b) Describe a filling order for your memoization structure.

(c) State and justify the running time of an iterative solution.