Introduction to Dynamic Programming, pt.3 (in two dimensions!)

Kathi Fisler
Adapted for CS200 by Milda Zizyte

April 8, 2022

Objectives

By the end of these notes, you will know

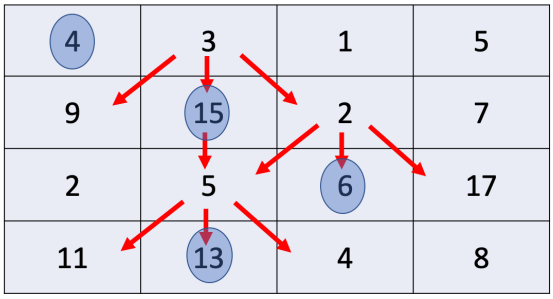
- how to approach a 2-dimensional dynamic programming problem

In the previous two classes, we searched for a way to optimize our selection of sweets from a display case, and found the longest increasing subsequence in a list of numbers. Our input data for each problem (the rating of each sweet or the list of numbers) was uni-dimensional, in that we had a set of options to choose among that were ordered only in one dimension.

Today. we turn to a problem in which the options are ordered in two dimensions, each of which contributes to which information can be considered as we optimize our solution.

1 Maximizing Halloween Candy

Imagine that you live in a neighborhood that is laid out in a grid. It's Halloween, and you get to stop at once house on each east-west street (in each row) to collect candy. The idea is that you will start from some house in the top row, then make your way down to the bottom row. From each house you visit, the next one has to be either directly below or diagonally adjacent. The following image shows the idea (only some of the red "next" arrows are present to avoid clutter). The blue highlights show the optimal choice.



To see the computation that allows us to reach the optimal outcome, you can look at the Google Sheet linked to the notes page of this lecture, which computes a table from the input data (gray in the sheet). The light orange table shows the table that holds the max candy you can get from going through each house, assuming you filled the table from the bottom row to the top row. The dark orange highlighted cells show the path you take to get to the optimal solution. Try changing the values in the blue cells to see how they affect the final outcome.

The idea behind the computation is that the orange table shows the maximum pieces of candy it is possible to get by going through the corresponding house, assuming that you already visited one of the houses in the row below it. We fill out the bottom row of the orange table to be equal to the pieces of candy at each of the houses in the bottom street, as our *base case*. Then, we can build up the rest of the table using the information about the houses (gray table) and the information we have computed so far about the optimal route (orange table).

If you look at the formula for each cell in the table, you will see that the idea of the computation is:

`(num. pieces of candy from that house) + max(candy from going through the 2 or 3 neighboring house`

Because some houses are at the edge of the neighborhood, they do not have 3 neighboring houses in the street below, so we have to account for the edge cases (for example, house B1 has three neighboring houses in the street below: A2, B2, and C2; while house A1 only has two: A2 and B2, because there is no down-left neighboring house).

Note that the first part of the computation comes from the blue table, because it is the new piece of information we are introducing about the house. The next part of the computation comes from the orange table, because it is the computation we are using that tells us the optimal route we have computed so far (through the houses below the house in question).

Below we show the un-optimized recursive program that solves this problem, in pseudocode (the heart of the recursive function matches the computation in the orange table).

```
1    # the number of rows is len(candy_available)
2    # the number of columns is len(candy_available[0])
3
4    # this function computes the maximum candy we can get by going through the house
5    # at the address (house_row, house_col)
6    max_candy_through_house(candy_available, house_row, house_col):
7                        # candy_available is 2d array of houses
8      if house_row == len(candy_available) - 1: # if we are at the last row of houses
9        return candy_available[house_row][house_col]
10     else:
11       houses_below = [] # look at the answers for the 2 or 3 houses below
12
13       if house_col > 0 # not at the leftmost house:
14         houses_below.append(
15           max_candy_through_house(candy_available, house_row + 1, house_col - 1)
16
17       # house directly below:
18       houses_below.append(
19         max_candy_through_house(candy_available, house_row + 1, house_col)
20
21       if house_col < len(candy_available[0]) - 1: # not at rightmost house
22         houses_below.append(
23           max_candy_through_house(candy_available, house_row + 1, house_col + 1)
24
25       # candy at this house + the maxi we can get by going through the houses below
26       return candy_available[house_row][house_col] + max(houses_below)
27
28   # get the maximum value in the top row
29   # notice that this is a list comprehension that helps us loop through the top row
30    max([max_candy_through_house(candy_available, 0, c)
31      for c in range(len(candy_available[0]))])
```

To optimize this solution using DP, we would essentially create the orange table from the sheet: we would first initialze a table the same size as our input, and populate it bottom-to-top. The recursive calls would be replaced with lookups into the relevant places in the table.

In your seamcarve assignment, you are essentially performing this same computation, but with max

replaced by min, so we are leaving the Python DP code to you.

2 Summary

What should you take from the DP segment?

- If you have a piece of functional (no mutation) code that makes the same call on the same inputs multiple times, you can save time by storing the previously-computed results in a data structure and retrieving them later.
- Search-based problems often satisfy this pattern. In search-based problems, we are looking to find (a) a solution, that (b) optimizes for some attribute of the data. In these notes, we have shown you how to make data structures to hold the results of both the optimized attribute and the solution paths as you run the program.
- A program modified to use dynamic programming will run only once on each unique input value. Dynamic programming saves on time by using more space.
- We used arrays as the data structure for previously-computed inputs here (rather than, say, hashtables) partly because you will often see DP problems solved with arrays (and we want you to be prepared for interviews). You could have used different data structures in practice.
- Starting from a working recursive solution without optimization can ease the process of writing the optimized solution.

3 Study Questions

1. Why did we say that DP only works if the original recursive program is functional (rather than mutating data)?
2. If you do write both the unoptimized and the optimized solutions, how might you use both in testing?
3. DP seems to introduce a sizeable space overhead, especially if we were to be searching in large data. Think about our final Halloween solution, and the patterns of array cell access that get used in the final solution. Do the patterns suggest ways that we could use less space than our current approach (which needs two arrays, each the same dimensions as the input neighborhood).