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

Kathi Fisler and Milda Zizyte

November 11, 2022

Objectives

By the end of these notes, you will know

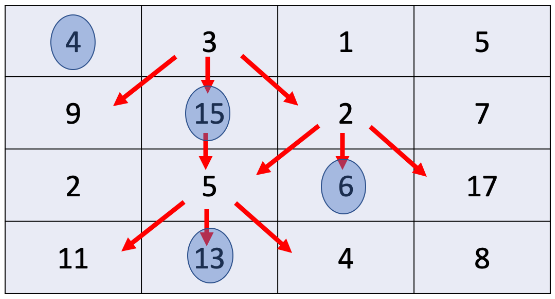
- how to approach a 2-dimensional dynamic programming problem

In the previous two lectures, 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. 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.



1.1 Recursive solution

Below we show the un-optimized recursive program that solves this problem, in pseudocode. The key computation this code makes on every house (except for houses in the last row) is that the maximum pieces of candy we can get by going through a certain house is:

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

The expression depends on "the 2 or 3 neighboring houses" because some houses are at the edge of the neighborhood and do not have a house diagonally-left (for leftmost houses) or diagonally-right (for rightmost houses).

```
# assume candy_available is a global variable that is a 2d array
# of how much candy you can get at each house; indexed By
# candy_available[house_row][house_col]

# num_rows is the number of rows in candy_available
# num_cols is the number of columns in candy_available


# this function computes the maximum candy we can get by going through the house
# at the address (house_row, house_col)
max_candy_through_house(house_row, house_col):
  if house_row == num_rows - 1: # if we are at the last row of houses,
                                # just give back the candy at that house
    return candy_available[house_row][house_col]
  else:
    prev_houses_candy = []
    for each candidate_col (the possible houses diagonally and directly below):
        compute max_candy_through_house(house_row + 1, candidate_col)
        append result to prev_houses_candy

    # candy at this house + the max we can get by going through the houses below
    return candy_available[house_row][house_col] + max(prev_houses_candy)
```

To get the final answer, we would call `max_candy_through_house` on every house in row 0, and take the maximum over all of those results.

1.2 The DP solution

We can convert this solution to DP by storing the result of each recursive call in a table (2d array), in the order in which the calls for the recursive code would finish (that is, last row to first row). 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). There are some light orange tables that show example computations for this dynamic programming approach, which we discuss in the next paragraph. For now, try changing the values in the gray cells to see how they affect the final outcome (the bottom-right light orange table).

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). The table in the middle-left of the sheet shows numerical examples of these computations, and the table in the bottom-left of the sheet translates these examples to spreadsheet formulas. The bottom-right table actually computes the values according to these formulas.

Note that the first part of the computation comes from the gray table, because it is the new piece of information we are introducing about the house. The next part of the computation comes from the lower row in 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).

To code the DP solution up, we would essentially create the orange table from the sheet (as a 2d array), following the examples/formulas shown on the left side of 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 performing this same computation, but with max replaced by min, so we are leaving the Python DP code to you. The code/console that is posted on the class lectures page shows some examples of working with 2d arrays in Python, which we haven't done before.

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.