Lecture 13: Dynamic Programming Part 2
10:00 AM, Feb 23, 2018

Contents

1 Recap

2 Maximizing Halloween Candy
   2.1 Implementing this in Code

3 A Second Example: Longest Common Substring

Objectives

By the end of these notes, you will know:

- how to approach a 2-dimensional dynamic programming problem

By the end of these notes, you will be able to:

- follow a sequence of steps to solve a recursive problem using dynamic programming

1 Recap

So far, we have solved two problems using DP:

In Fibonacci, every subproblem (each \( \text{fib}(i) \)) was used to compute the final answer.

In Macarons, some subproblems were used and some were skipped in the final solution. The recursive algorithm searches for which subproblems to use in the final solution. This is an example of what’s called an optimization problem (which arise all over computer science). Since multiple candidate solutions build off the same subproblems, DP enables us to consider a subproblem multiple times while only computing its answer once.

The underlying data for both of these problems was 1-dimensional. Today, we look at what happens when we have to search in a 2-dimensional space.

2 Maximizing Halloween Candy

It’s Halloween and you’re going trick-or-treating! Your parents have dropped you off at the north edge of the neighborhood and will pick you up at the south edge in an hour. The houses in the neighborhood are laid out on a rectangular grid, and you only have time to visit one house in each row as you move south procuring candy. Also, lest you risk insulting any of your friendly neighbors
by skipping over them, the next house you visit must be adjacent to the current one, meaning
directly southwest, south, or southeast.

Because you love candy, yet your time is limited, you really want to optimize your path through the
neighborhood to maximize the amount of candy you can procure. To help you and others meet this
lofty goal, the neighborhood’s data scientists have analyzed data from past years to produce reliable
estimates of how much candy each house is expected to give out this year, as shown below:

```
  4  3  1  5
  9 15  2  7
  2  5  6 17
 11 13  4  8
```

Given this rectangular grid of numbers, a neighborhood whiz kid calculated the optimal path from
top to bottom, abiding by the above-mentioned constraints. That is, she figured out which path
accrues the most candy. How might we do this programmatically?

We’re going to do this by systematically exploring the neighborhood data (before the night we go
trick or treating): we’ll calculate the best path starting from every house on the north edge, then
check which one yields the most candy.

Here, we introduced the dynamic programming worksheet, that is linked to the webpage alongside
todays’s notes. All of the steps we’re discussing here are part of that worksheet.

The first step is to **play with an example to make sure you understand the constraints of the problem.**
Use our example from above. Imagine that we had started at the second house on the north edge
(the one giving out 3 pieces of candy). Where could we go next? According to the constraints, we
could go to any one of the houses offering 9, 15, or 2 pieces of candy in the second row. If we chose
the house offering 2 pieces, we could go to any one of the houses offering 5, 6, or 17 pieces of candy
in the third row, and so on.

Always make sure you have worked through one or two examples by hand (on paper)
to make sure you understand the constraints of the problem before you go further. If
you can’t explain what needs to happen on a concrete example, you won’t be able to
express a general case in code.

The next step is to figure out **what table dimensions (and contents) you need to store the results of
the subproblem.** Here, we are trying to store the max amount of candy at each house, so we need a
table of the same dimensions as our input data, holding numbers.

The next step is to figure out **the meaning of a cell in the subproblem table.** For this problem, we
have two options:

1. How much candy you can accumulate from this house to the south edge
2. How much candy you could have accumulated before reaching this house, having started at
   the north edge

Either of these works. In the first, you would fill the table from the bottom row to the top. In the
second, you would fill the table from the top row to the bottom. You would get the same answer in
either case, but you’d have to look for it in a different spot in the table. In the first version, the max
candy would be stored in the top row, while in the second version the max candy would be stored in the bottom row. If all we want is the max amount of candy, it doesn’t matter which way we go.

But we also want the path that gets the most candy. Either version would correctly compute that path. In the first version, the house with the max candy in the top row is also our starting point. In the second version, we’d look in the bottom row to find the max candy, then have to lookup which north-edge house got us there. This isn’t an expensive operation, but it is a bit of extra work. As a result, we work with the first version in these notes.

Now, we formalize the informal work we’ve done so far by writing a recurrence relation to capture the relationships between the subproblems and the input data. To write the recurrence, you will need the original input data (the per-house candy totals) as a formal piece of program data. So let’s nail that down first.

Let’s assume our data table is actually a two-dimensional array \( A \) with \( n \) rows and \( m \) columns (we’ll see how to program 2D arrays before lecture is done). If row 0 represents the north edge, and column 0, the west, then there is a 3 in cell \((0, 1)\) (i.e., \( a_{0,1} = 3 \)), a 2 in cell \((2, 0)\) (i.e., \( a_{2,0} = 2 \)), and so on:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( j = 0 )</th>
<th>( j = 1 )</th>
<th>( j = 2 )</th>
<th>( j = 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i = 0 )</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>( i = 1 )</td>
<td>9</td>
<td>15</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>( i = 2 )</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>( i = 3 )</td>
<td>11</td>
<td>13</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

To write the recurrence, we also need to name our table of intermediate computations. We’ll call that \( C \) (for candy). Here’s the recurrence relation. It captures that we choose a neighboring house in the next row with the max candy and add the candy available at the current house to determine the max candy at the current house:

\[
C(i, j) = \begin{cases} 
A_{ij}, & \text{if } i = n \\
A_{ij} + \max\{C(i + 1, j - 1), C(i + 1, j), C(i + 1, j + 1)\}, & \text{otherwise}
\end{cases} \tag{1}
\]

The recurrence shows that any constraints on the problem are reflected in the general case of the recurrence relation. Here, the only constraint was on which houses can be considered. For macarons, the constraint was that you had to skip adjacent options.

The final step is to identify how you get the final (optimal) answer from the table. If we filled the table from bottom to top, then the starting house is the one in the top row of \( C \) with the maximum computed candy value.

### 2.1 Implementing this in Code

The code file is posted on the lectures page. Let’s walk through the high-level points:

**Creating 2-dimensional tables** We capture 2-dimensional tables by creating an array of arrays. Each row of the table is an array (with an index for each column). An array of rows captures an entire table. In Java, here’s how we would write or original input table:
int[][] H = new int[][] { {4, 3, 1, 5},
(9, 15, 2, 7),
(2, 5, 6, 17),
(11, 13, 4, 8})

The notation int[][] creates a two-dimensional array of ints. You can use any type in place of int. You access elements of this 2D array just as you would a 1D array. For example, if you wanted the third row of houses (from the top), you would write H[2]. If you wanted to extract the 5 from the third row, you would write H[2][1] (third row, second column, off by 1 since indices are numbered from zero).

**Setting up the DP Solution**  The architecture of our DP solution is a file as follows:

```java
public class Halloween {

    // define fields for the input table, dimensions, and subproblem table
    private int[][] itemValues; // the candy available at each house
    private int height; // the height of the town
    private int width; // the width of the town
    private int[][] optTable; // table of optimal values, where
    // optTable[i][j] is the amt of candy
    // from this cell to south end of town

    // constructor
    public Halloween(int[][] itemValues) {
        // any error checking on input goes here
        // initialize the fields
        this.itemValues = itemValues;
        this.height = itemValues.length;
        this.width = itemValues[0].length;
        this.optTable =
        new int[this.itemValues.length][this.itemValues[0].length];
        initTable();
        fillTable();
    }

    // initialize the optTable with base cases of the recurrence
    // and a default value in all other cells
    private void initTable() { ... }

    // use the recurrence to populate the optTable
    private void fillTable() { ... }

    // find the maximal value in the filled optTable
    public int optimalValue() { ... }
}
```

The posted code file shows all the details. The posted file also shows how we track the optimal paths as well as the optimal amounts of candy. Basically, we maintain a second table that holds a LinkedList of the column indices on the best path from each cell to the south edge of town. You could have used an array or an ArrayList in place of the LinkedList just as well.
Initializing Cells with Paths  
There is one Java annoyance which comes up when you create the lists to hold the paths. Assume we have set up a 2D array to hold those paths as follows:

```java
private LinkedList<Integer>[][] optPaths;
```

Inside the constructor, you want to initialize this field to hold an array of the needed dimensions. We would like to write this code as:

```java
this.optPaths = new LinkedList<Integer>[height][width];
```

If you do this, you will get a weird error about Java not letting you define arrays with generic types. The fix is to leave the Integer annotation off the array that you create, then add it back on with the parenthesized expression before the new as shown below:

```java
this.optPaths = (LinkedList<Integer>[][]) new LinkedList[height][width];
```

The parenthesized expression at the front is something called casting. We’ll review it in class next week.

3  A Second Example: Longest Common Substring

You’ve now seen two path-based optimization problems. Let’s look at a different kind of problem that is suited to dynamic programming. Assume we are given two strings. What is the longest substring that is common to both strings?

This problem, named longest common substring is used particularly in computational biology, to detect subsequences of genes with shared patterns.

As an example, let’s take the words “bluno” and “bun”. We should get the string “un” as the result on these two words (we don’t include “b” since it isn’t part of the same substring as “un” in “bluno”).

As with the halloween problem, we will set up a 2D table, this time with the letters of each word populating the rows and columns. We’ll call this table $S$ (for “substring”)

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>l</th>
<th>u</th>
<th>n</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What now goes in each cell of the table? One idea would be that a cell represents the longest common substring in the suffixes of the two strings from that cell (suffix is a sequence of characters from a position in the string to the end). If we did that, we might get a table like the following as we fill it from the bottom (lower two rows filled so far):

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>l</th>
<th>u</th>
<th>n</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>un</td>
<td>un</td>
<td>un</td>
<td>n</td>
<td>-</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>-</td>
</tr>
</tbody>
</table>
So now let’s fill the top row. what would the rule look like for the top-left corner? Both words have a “b”, so it would seem we should add the “b” onto the longest substring from “luno” and “un”. But this would suggest the final answer “bun”, which isn’t correct. But we’re supposed to build off the neighboring cells somehow, so what do we do?

The problem here is that we have the wrong meaning for the contents of the cells. If we want to be able to extend the answer in a cell to add another letter as we make each suffix longer, we have to make sure that the common substring is at the front of each suffix. In other words, our rule really needs to be:

A cell represents the longest common prefix of the suffixes of the words corresponding to the given cell.

With this interpretation, our table would fill as follows:

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>l</th>
<th>u</th>
<th>n</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>b</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>u</td>
<td>-</td>
<td>-</td>
<td>un</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n</td>
<td>-</td>
<td>-</td>
<td>n</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Note something interesting here – the final answer we wanted (“un”) is not in the first or last rows of the table! So for this problem, we have to look throughout the table to find the largest substring when we compute our final answer.

Does having to do that mean that the DP solution is no less expensive than the naive recursive solution? Think about that and ask us if you have questions.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)