

Lecture 27: Dynamic Programming Part 2

11:00 AM, Mar 24, 2021

Contents

1	Searching for Solutions: the Macaron problem	1
1.1	The plain recursive solution	2
2	Fibonnaci vs Macarons	2
3	What is Dynamic Programming?	4
4	Maximizing Halloween Candy	4
4.1	Searching for paths	5
5	Summary	7
6	Study Questions	7

Objectives

By the end of these notes, you will know

- how to optimize a 1-dimensional recursive problem using Dynamic Programming
- how to search for a solution in dynamic programming problem
- how to approach a 2-dimensional dynamic programming problem

1 Searching for Solutions: the Macaron problem

The setup notes for today described the macaron-selection problem, in which we needed to search for the best set of macarons to purchase. We defined “best” as the set of macarons that yielded the highest total review rating, where each individual macaron had a given rating. We could have used other attributes, such as sugar content, to optimize on. The point is simply that we have (a) a set of items with (b) attributes and (c) a metric that we want to optimize for in choosing a subset of those items.

The point is that we were searching for an optimal solution based on a metric on our data. Searching-based problems are often solved using recursion. These problems also often involve many repeated intermediate computations, which is why we want to optimize them.

1.1 The plain recursive solution

Here is a plain recursive solution to the Macaron problem (without optimization):

```
class Macaron(val ratings: Array[Int]) {
  // The input is the array of ratings -- this is part of
  // the original data, not the array that we create
  // to optimize performance

  /**
   * Finds the optimal rating sublist of the Macarons.
   *
   * @param i - an int between 0 and the number of Macarons minus 1
   * @return the max total rating that can be achieved using Macarons up to
   *         index i
   */
  def maxRec(i: Int): Int =
    if (i == 0)
      this.tastinessValues(0)
    else if (i == 1)
      if (this.tastinessValues(0) > this.tastinessValues(1))
        this.tastinessValues(0)
      else this.tastinessValues(1)
    else {
      val twoAgo = maxRec(i - 2)
      val oneAgo = maxRec(i - 1)
      if (oneAgo > (twoAgo + this.tastinessValues(i)))
        oneAgo
      else twoAgo + this.tastinessValues(i)
    }

  def maxTastiness = maxRec(tastinessValues.length-1)
}

object Main extends App {
  print(new Macaron(Array(3, 10, 12, 16, 4)).maxTastiness + " should be 26")
}
```

To convert this to dynamic programming, we have to add an auxiliary data structure to store the results of calls to the recursive functions. Let's sketch out the code at a high level. Figure 1 shows a copy of the code annotated with the changes that need to be made to remember and use the values of previous calls to `maxRec`. The annotations are in all-caps comments

We did not actually fill in this code in class. The next lecture has a completed version of code for a DP problem.

2 Fibonacci vs Macarons

In **Macarons**, some subproblems were used and some were skipped in the final solution. In contrast, **Fibonacci** used all of the lower values of the input to compute the output. In Macarons, we could reasonably ask “which macarons should we buy?”, which signals that there is a choice of which

```

class MacaronTD(val tastinessValues: Array[Int]) {
  // The input is the tastiness array -- this is part of
  // the original data, not the array that we create
  // to optimize performance

  // SET UP ARRAY TO HOLD OUTPUTS AS THEY GET COMPUTED
  // INITIALIZE ARRAY WITH DEFAULT VALUES (USING NONE)

  /**
   * Finds the optimal tastiness value sublist of the Macarons.
   *
   * @param i - an int between 0 and the number of Macarons minus 1
   * @return the max tastiness that can be achieved using Macarons up to
   *         index i
   */
  def maxRec(i: Int): Int =
    // CHECK TO SEE IF OUTPUT ON i HAS ALREADY BEEN COMPUTED, IF SO RETURN
    // IT
    if (i == 0)
      // STORE VALUE IN THE COMPUTED ARRAY
      this.tastinessValues(0)
    else if (i == 1)
      // STORE VALUE IN COMPUTED ARRAY
      if (this.tastinessValues(0) > this.tastinessValues(1))
        this.tastinessValues(0)
      else this.tastinessValues(1)
    else {
      val twoAgo = maxRec(i - 2) // GOT STORED WHEN CALLED RECURSIVE
      val oneAgo = maxRec(i - 1)
      if (oneAgo > (twoAgo + this.tastinessValues(i)))
        // STORE VALUE IN COMPUTED ARRAY
        oneAgo
      else
        // STORE VALUE IN COMPUTED ARRAY
        twoAgo + this.tastinessValues(i)
      // RETURN NEWLY COMPUTED VALUE
    }

  def maxTastiness = maxRec(tastinessValues.length-1)
}

```

Figure 1: Annotated Macaron code showing the changes to make for dynamic programming.

pieces of your data contribute to the final solution. This searching aspect is a key difference between the two problems.

Macarons 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.

3 What is Dynamic Programming?

The Wikipedia entry on dynamic programming has a history section that explains the context and origin of the term. It ties heavily to the search-style problems like Macarons.

https://en.wikipedia.org/wiki/Dynamic_programming

4 Maximizing Halloween Candy

At the end of the last lecture, we briefly introduced a problem of searching for the best route to maximize how much candy you can get from trick-or-treating on Halloween. The idea is that you will start from some house in the top row, then make your way down to the bottom row. You can pick up candy from one house in each row. From each house you visit, the next one has to be either directly below or diagonally adjacent. The following image (left) shows the idea (only some of the “next” arrows are present to avoid clutter). The blue highlights show the optimal choice.

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

To see the computation that underlies the table, you can look at the following sample Excel sheet, which computes two tables from the input data (in gray in the sheet). The pink table shows the table that holds the max candy you can get at each house, assuming you filled the table from the top row to the last row. The green table shows the max-candy table, this time filled from the last row to the top row. Each table shows the same max answer (circled in blue). While the circled cell in each table is part of the optimal path (from the blue circles on the gray table), our code will still need to compute the actual path.

A DP-solution would compute one of these two tables (either choice works) to store the results of computing the max candy available at each house. The solution we build today will follow the formulas that generated the green table.

<https://brown-cs18-master.github.io/content/lectures/27dynprog2/tables.xlsx>

```

class Halloween() {

  def maxCandy(candyAvailable: Array[Array[Int]]): Int = {
    val height = candyAvailable.length - 1
    val width = candyAvailable(0).length - 1

    // get the available candy from the input, returning
    // 0 if the requested row/col is not in the table
    def candyVal(row: Int, col: Int): Int = {
      if ((row > height) | (col > width) | (col < 0))
        0
      else
        candyAvailable(row)(col)
    }

    def maxRec(row: Int, col: Int): Int = {
      if (row == height)
        candyVal(row, col)
      else
        candyVal(row, col) +
          max(maxRec(row + 1, col - 1),
              max(maxRec(row + 1, col), maxRec(row + 1, col + 1)))
    }

    // get the largest candy total from the top row
    List.range(0, width).map(c => maxRec(0, c)).max
  }
}

```

Figure 2: The unoptimized recursive solution

Figure 2 shows the un-optimized recursive program that solves this problem (the heart of the recursive function matches the computation in the pink/green tables). Figure 3 is the DP solution that computes the green table, storing it in an array. See the code comments for additional information.

4.1 Searching for paths

To also search for the optimal *path* (instead of just the optimal *value*), we need to compute a second table that holds the path of cells that got to the corresponding value in the green table. Slide 5 of the deck shows the path table in the upper right. Each cell in the darker-green table holds a list of tuples for the coordinates that were used to compute the max value in the lighter-green table. If you look at the list of coordinates in the top-left corner of the dark green table, you see that they correspond to the path taken with the blue circles on the gray table.

Figures 4 and 5 shows the code augmented with a second table (called `bestPath`) that mimics the dark green table. The two tables get computed simultaneously – as we figure out which cell to use to fill the next cell in the light-green table, we add that cell to the list accumulating in the dark-green table. This version has moved the code for initializing the arrays and populating

```

class HalloweenLtoF() { // LtoF stands for "last to first row"

  def maxCandy(candyAvailable: Array[Array[Int]]): Int = {
    val height = candyAvailable.length - 1
    val width = candyAvailable(0).length - 1
    // the array for holding previously computed values
    val candyComputed = Array.ofDim[Int](width+1, height+1)

    // get the available candy from the input, returning
    // 0 if the requested row/col is not in the table.
    def candyVal(row: Int, col: Int): Int = {
      if ((col > width) | (col < 0))
        0
      else
        candyComputed(row)(col)
    }

    // set up the base cases of the computed values
    for (col <- 0 to width) {
      candyComputed(height)(col) = candyAvailable(height)(col)
    }

    // set up the rest of the computed values
    for (row <- height - 1 to 0 by -1)
      for (col <- 0 to width)
        candyComputed(row)(col) =
          candyAvailable(row)(col) +
            max(candyVal(row + 1, col - 1),
              max(candyVal(row + 1, col), candyVal(row + 1, col + 1)))

    // lookup the computed answer
    // get the largest candy total from the top row
    List.range(0, width).map(c => candyComputed(0)(c)).max
  }
}

```

Figure 3: The optimized solution for computing only max candy available

them into helper functions. The main reason for that change is to align with the homework format, which was designed to show you a structuring technique that would also work in Java (where these methods would get called within the constructor).

5 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, hashables) 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.

6 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).

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: <https://cs.brown.edu/courses/cs018/feedback>.

```

class HalloweenLtoFPaths(candyAvailable: Array[Array[Int]]) {
  val height = candyAvailable.length - 1
  val width = candyAvailable(0).length - 1
  val candyComputed = Array.ofDim[Int](width + 1, height + 1)
  // make lists of tuple of coordinates to store paths
  val bestPath = Array.ofDim[List[(Int, Int)]](width + 1, height + 1)

  // suppressed in this version for space
  def candyVal(row: Int, col: Int): Int = { ... }

  // a new helper to return which of col-1, col, and col+1 has
  // highest value in row -- use this to update both tables
  def bestCol(row: Int, col: Int): Int = {
    if (candyVal(row, col - 1) > candyVal(row, col))
      if (candyVal(row, col - 1) > candyVal(row, col + 1))
        col - 1
      else col + 1
    else if (candyVal(row, col) > candyVal(row, col + 1))
      col
    else col + 1
  }

  def initTable() = {
    // set up the base cases of the computed values
    for (col <- 0 to width) {
      candyComputed(height)(col) = candyAvailable(height)(col)
      bestPath(height)(col) = List((height, col))
    }
  }

  def fillTable() = {
    // set up the rest of the computed values
    for (row <- height - 1 to 0 by -1) {
      for (col <- 0 to width) {
        val maxCol = bestCol(row + 1, col)
        candyComputed(row)(col) =
          candyAvailable(row)(col) + candyVal(row + 1, maxCol)
        bestPath(row)(col) = (row, col) :: bestPath(row + 1)(maxCol)
      }
    }
  }

  // run the setup methods
  // in Scala, these don't really need to be methods. However, the homework
  // handout asks for them because this is how you would set up such a
  // problem in Java -- these methods would get called in the constructor
  initTable()
  fillTable()
}

```

Figure 4: Most of the solution that also computes paths.


```
// get the largest candy total from the top row of the candyComputed table
def maxCandy(): Int = {
  List.range(0, width).map(c => candyComputed(0)(c)).max
}

// get the path that leads to the max candy
def maxPath(): List[(Int, Int)] = {
  val indices = List.range(0, width)
  val topRowVals = indices.map(c => candyComputed(0)(c))
  val indexOfMax = topRowVals.indexOf(topRowVals.max)
  bestPath(0)(indexOfMax)
}
```

Figure 5: The rest of the paths-based solution .