Lecture 27: Dynamic Programming Part 2
11:00 AM, Apr 11, 2020

Contents

Objectives

By the end of these notes, you will know how to optimize a 1-dimensional recursive problem using DP.

1 Recap

Last lecture, we looked at the Fibonacci problem. We showed how to modify the code to remember prior computed results so that we could optimize the running time of the function.

2 Practicing DP: the Macaron problem

This lecture, we look at another 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 “tastiness” rating, where each individual macaron had a given tastiness score. We could have used other attributes, such as calorie counts, to optimize on.

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.

2.1 The plain recursive solution

Here again is the plain recursive solution from the setup notes:

```scala
class Macaron(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

  /**
   * 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 =
```
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.

3 Fibonnaci vs Macarons

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

4 What is Dynamic Programming?

In the setup survey answers for today, several students asked where the term “dynamic programming” means. 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.

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 =

 if (i == 0)
   // STORE VALUE IN THE COMPUTED ARRAY
   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 FUNCTION
   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.
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