Lecture 27: Dynamic Programming Setup
11:00 AM, Apr 11, 2020

Contents

We’re going to practice and extend our work with optimizing recursive functions that compute the same answers multiple times. Look over this problem before class tomorrow.

Here’s a depiction of a store-counter of macarons:

| Chocolate | Strawberry | Vanilla | Pistachio | Raspberry |

Brock wants to purchase a number of macarons, since they’re easy to carry around when he’s got work to do, but the shop owner has a particular (and odd) rule: he may not purchase two adjacent macarons. For example, in the above arrangement, he cannot purchase both strawberry and vanilla macarons.

Each flavor of macaron has a positive (non-negative and nonzero) tastiness value based on how tasty that flavor is:

- Flavor 1: Strawberry (value 35),
- Flavor 2: Salted Caramel (value 90),
- Flavor 3: Chocolate (value 40),
- …
- Flavor n: Lavender (value 15).

Our goal will be to help Brock figure out the best set of macarons to purchase—that is, the set of macarons with the maximum sum of their tasty values while following the shop owner’s rule.

To prep for class, we want you to think about how to compute the maximum total tastiness that Brock could achieve under the shop-keeper’s rule (set aside which macarons to purchase – we’re just totaling for max tastiness). The idea is to go down the sequence of tastiness values and ask ourselves whether we are better off keeping or skipping the current macaron.

Let’s see the idea with an example. Assume that we have five macaron options with the following tastiness values: [3, 10, 12, 16, 4].

1. Enumerate the combinations that you could purchase following the non-adjacency rule. For example, you could achieve the sum 3+12+4 while following the rule. What other combinations are possible?

2. What is the maximum tastiness sum you can achieve on this example? (you’ll put your answer in Canvas)
3. Look at the following code that gives a naive recursive solution to computing the max tastiness. Try to convince yourself that it would compute the same answer that you worked out for the concrete example above. Enter questions about this solution and the problem setup in Canvas.

```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")
}
```

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](https://cs.brown.edu/courses/cs018/feedback)