Lecture 26: Dynamic Programming Part 1

11:00 AM, Mar 22, 2021

Contents

Motivating Question

How can we quickly search for optimal answers among sets of items?

Objectives

By the end of these notes, you will know

- how to optimize a functional recursive program that does the same computation multiple times
- how to approach a 1-dimensional dynamic programming problem

1 Searching for Solutions: the Sweets problem

Here's a depiction of a store-counter of sweets:

Chocolate	Strawberry	Vanilla	Pistachio	Raspberry

Brock wants to purchase a number of sweets, 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 sweets**. For example, in the above arrangement, he cannot purchase both strawberry and vanilla sweets.

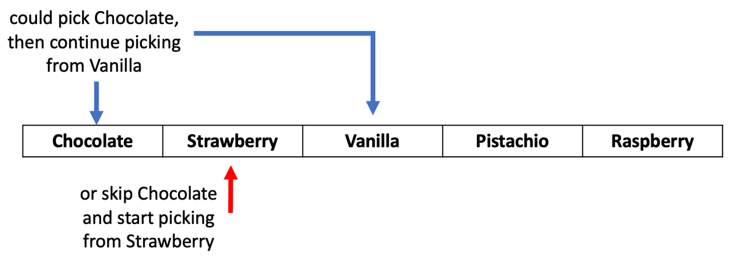
Each flavor of sweet has a positive (non-negative and nonzero) tastiness value based on how tasty that flavor is. Our goal will be to help Brock figure out the best set of sweets to purchase—that is, the set of sweets with the maximum sum of their tasty values while following the shop owner's rule.

Let's see the idea with an example. Assume that the five sweet flavors have the following tastiness values: $[3, 10, 12, 16, 4]$. By the rules, you could select any single sweet, or one of the following combinations (in terms of tastiness values):

- 3 + 12 + 4
- 3 + 16
- 3 + 4
- 10 + 16
- 10 + 4
- 12 + 4

The best score comes from taking 10 + 16 (Strawberry and Pistachio) – fewer sweets, but more tasty.

This is a brute-force method, but it would better for us to *search* for a good choice. The search will consider all of the combinations, but systematically. Here's a sketch of how we might do this (where recursive calls would start new searches from Strawberry and Vanilla as part of computing the answer for Chocolate):

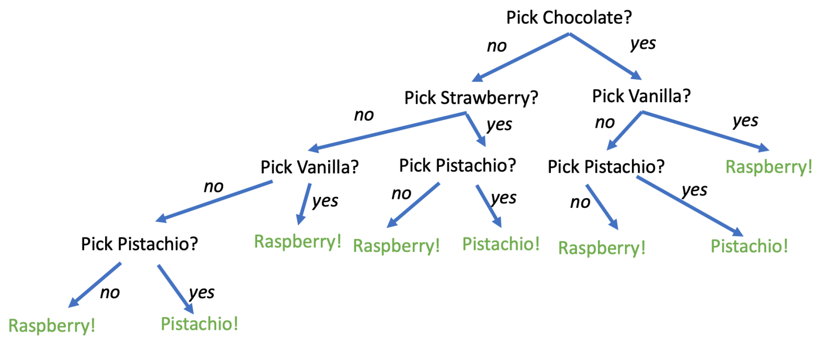

could pick Chocolate, then continue picking from Vanilla
Chocolate
Strawberry
Vanilla
Pistachio
Raspberry
or skip Chocolate and start picking from Strawberry

Look at the following code that gives a naive recursive solution to computing the max tastiness. This version assumes that the tastiness scores for these flavors are stored in an array. Try to convince yourself that it would compute the same answer that you worked out for the concrete example above.

```
class Sweet(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 Sweets.
 *
 * @param i - an int between 0 and the number of Sweets minus 1
 * @return the max tastiness that can be achieved using Sweets 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 Sweet(Array(3, 10, 12, 16, 4)).maxTastiness + " should be 26")
}
```

Stop and Think: What is the running time of this code, in terms of the number of sweets in the collection?

One way to think about this is to unroll the recursive calls that get made into a tree, as follows:

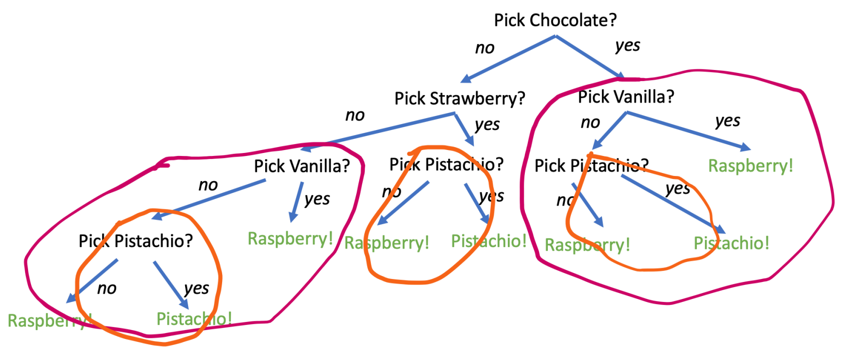


Stop and Think: Now that you see the tree, what is the running time of this code? Think about how many nodes are in this tree.

A mostly-balanced tree of height n has $O(2^n)$ nodes (exponential). Looking down the leftmost branch, the depth of the tree matches the number of flavors. Our naive recursive solution is therefore exponential-time. That's not a problem with 5 flavors, but optimization problems often have large amounts of data.

1.1 Avoiding Redundant Computation

Looking at the tree, we see some subtrees appear more than once. The tree from Vanilla appears twice, and that from Pistachio appears three times. The same value gets returned from each computation on Vanilla, and the same for Pistachio.



If we are worried about runtime here, perhaps we could avoid repeating a computation to save time. Specifically, if we actually expanded out the recursive calls only once per flavor, somehow saving the results of each call, this computation could be done in linear time instead.

How might we do this in code, however?

The basic idea is straightforward: Create a hashmap from flavor names to the result of the @codemaxRec function for that flavor name. Each time we would return a result on a flavor name, we store the result in the hashmap first. Before we start making recursive calls, we check the hashmap to see whether we have already computed the answer. 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

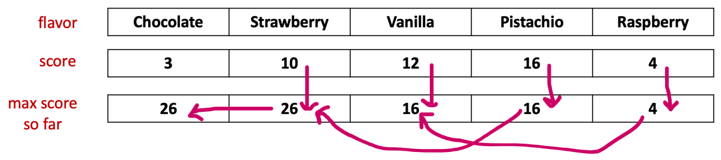
This approach takes the run-time down to linear, once you fill in the code.

Note: Many problems like this store the "previously computed" data in arrays rather than hashmaps. This makes sense when there is already a way to associate the items with array indices. For the sweets problem, we can use position in the sequence of flavors as the indices. That's what the above code does.

2 Using Iteration instead of Recursion

As we see if we look down the left branch of the tree, the optimized linear recursive computation is going to store the values for the flavors in reverse order from the list (Raspberry first, then Pistachio, working back to Chocoloate). If the computation is essentially going to fill in the array/hashmap in reverse, couldn't we just compute the stored-value array iteratively using a for-loop?

Yes, and that is a common approach in practice. Here's a sketch of the code (along with the array contents that get computed):



flavor	Chocolate	Strawberry	Vanilla	Pistachio	Raspberry
score	3	10	12	16	4
max score so far	26	26	16	16	4

```
Given the three arrays above (max score starts out as all 0, others are given)
- initialize the two rightmost spots of max-score from the score array
- for each index i from length - 3 down to 0
   max-score[i] = max(score[i] + max-score[i + 2], max-score[i+1])
- the overall result is in score[0]
```

```scala
class SweetsTD(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 HASHMAP TO HOLD OUTPUTS AS THEY GET COMPUTED
  // INITIALIZE HASHMAP WITH DEFAULT VALUES (USING NONE)

  /**
   * Finds the optimal tastiness value sublist of the Sweets.
   *
   * @param i - an int between 0 and the number of Sweets minus 1
   * @return the max tastiness that can be achieved using Sweets up to index
       i
   */
  def maxRec(i: Int): Int =
    // CHECK TO SEE IF OUTPUT ON i IS IN HASHMAP. IF SO RETURN IT
    if (i == 0)
      // STORE VALUE IN THE HASHMAP
      this.tastinessValues(0)
    else if (i == 1)
      // STORE VALUE IN HASHMAP 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 HASHMAP
        oneAgo
      else
        // STORE VALUE IN HASHMAP
        twoAgo + this.tastinessValues(i)
      // RETURN NEWLY COMPUTED VALUE
    }

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

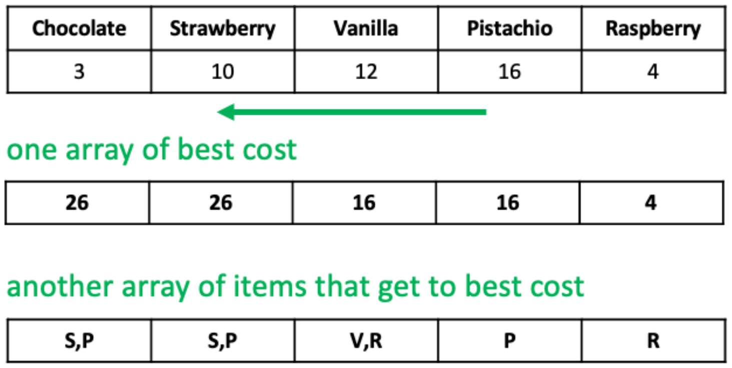
Figure 1: Annotated Sweets code showing the changes to make for dynamic programming.

Note that you can have the for loop go through the flavors either front to back or back to front. Either gets you the same max tastiness values (but with different array contents).

3 How to Know Which Sweets to Buy??

So far, our code has stored the max tastiness value that can be obtained, but not the set of sweets that yields that value. Often, we want that information, not just the end value.

To do that, the code maintains two arrays: one of the max values, and one of the flavors that you used while computing those values. The following diagram shows what the array looks like (the flavors are stored as lists within the array):



Chocolate	Strawberry	Vanilla	Pistachio	Raspberry
3	10	12	16	4

one array of best cost

26	**26**	**16**	**16**	**4**

another array of items that get to best cost

S,P	**S,P**	**V,R**	**P**	**R**

4 Dynamic Programming

This technique, of building up the solution to a problem from solutions to subproblems is called *dynamic programming*. Here, we motivated dynamic programming as a run-time optimization strategy for an initial recursive program. In the real world, you won't necessarily write the recursive program first. If you do already have the recursive version, however, you can augment it to save the results the function on different input values as you go.

When you augment the recursive version, the approach is often termed "memoization". In languages where you can pass functions as arguments, you can memoize a function without modifying its internal code. In Java, where you can't do this, you modify the function to save and fetch values from the array or hashmap.

For purposes of this class, we don't care that you know the differences between the terms memoization and dynamic programming. What we do care about is that you are able to take a problem that gets solved in terms of subproblems on smaller parts of the input and write an iterative solution that performs the computation more efficiently.

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 Sweets.

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