Introduction to Dynamic Programming

Kathi Fisler and Milda Zizyte

April 4, 2022

Motivating Question

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

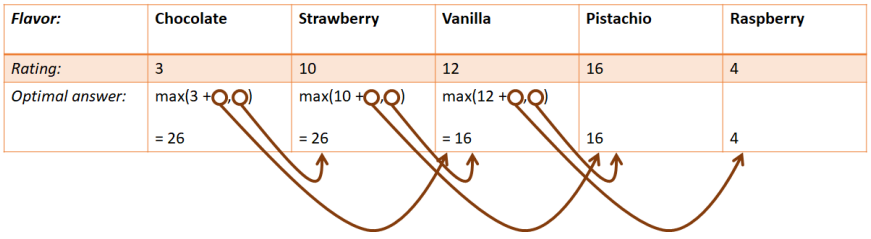
Objectives

By the end of these notes, you will know

- how to optimize a recursive program that does the same computation multiple times by storing the result
- how to implement a dynamic programming solution in Python
- how to compute the set of inputs for a dynamic programming problem, rather than just the optimal value

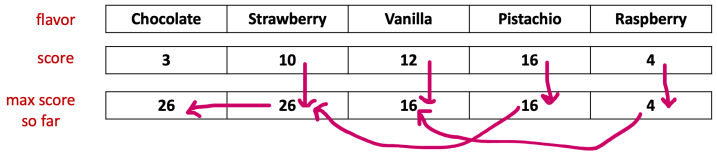
1 Optimizing the sweets problem using iteration

In the last lecture, we saw that we could compute the optimal sweets rating by storing intermediate answers in a list and moving backwards through the list:



If the computation is essentially going to fill in the array 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):



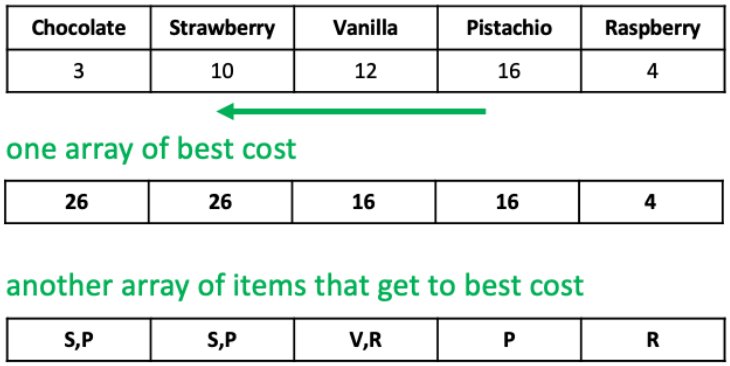

```
Given the three arrays above (max score starts out as all 0, others are given)
- initialize the two rightmost spots of max-score from the score array
- for each index i from length - 3 down to 0
    max-score[i] = max(score[i] + max-score[i + 2], max-score[i+1])
- the overall result is in score[0]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

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).

2 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):



How do we build up these lists of flavors? Remember that we are computing the flavors that get us the optimal score for the sublist starting with each index. We can do this at the same time as when we compute the optimal tastiness – in the sketch of the code above, the line

```
max-score[i] = max(score[i] + max-score[i + 2], max-score[i + 1])
```

got us the optimal score because we chose between picking the flavor (the `score[i] + max-score[i + 2]` input to `max`) and skipping the flavor (the `max-score[i + 1]` input to `max`). If we pick the flavor at index i, we add it on to the sweets we picked at index i + 2 (for example, at i = 1, we added strawberry to the list with pistachio, which was the list of flavors we had at i = 3). If, instead, we skip the flavor at index i, our optimal list of flavors is the list at index i + 1 (for example, at i = 0, we skipped chocolate, so we go with strawberry and pistachio (the optimal flavors at i = 1)).

3 Another example

In the candy example, we reduced an exponential ($O(2^N)$) runtime to a linear runtime by storing the intermediate result – for every element in the input list, we took the max of two numbers, which is a constant-time operation. If we use this technique, are we always guaranteed a linear runtime in the 1-dimensional case (the case where our optimal computations can be stored as a list)? The answer turns out to be no (depending on the problem), but we still make significant gains over the recursive solution if the recursive solution has a lot of repetition.

Consider the "maximum increasing subsequence" problem: given a list of integers, find the longest subsequence (sequence of elements from the list that are in the same order as in the list, not necessarily consecutive) that is increasing. For the list [3, 7, 4, 2, 5], the answer is [3, 4, 5]. The spreadsheet handout that came with this lecture has you try out examples of computing the longest increasing subsequence which starts with each element in the list, in turn. The logic behind this is that the optimal subsequence is guaranteed to start with one of the elements in the original list, so if we compute all of these answers, we can find out which one will have the optimal result by choosing the one that started the longest subsequence.

The supplementary notes to this lecture go through this example and its implementation in Python in more detail. The naive recursive solution actually runs in factorial time – for each index in the list, you might potentially do as many computations as there are remaining elements in the list, which adds up very fast if you don't store the intermediate results. When we do store the intermediate results, the problem reduces to quadratic time – for every element in the list, we are taking the maximum of potentially every subsequent element in the array where we store the optimal result (rather than comparing a constant number of elements, as we did for candy). This means that, when you are reasoning about the runtime of these kinds of problems, you should be careful to consider how many operations the code is doing to fill out every cell of the computation.

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`