Introduction to Dynamic Programming

Kathi Fisler
Adapted for CS200 by 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 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.

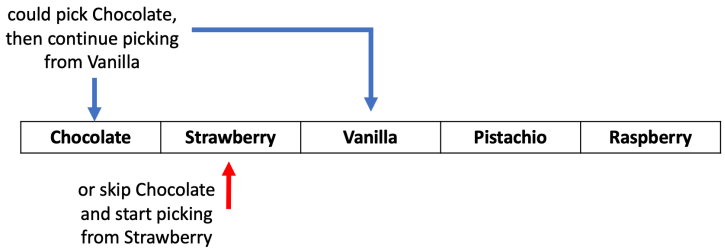
Each flavor of sweet has a positive (non-negative and nonzero) tastiness value based on how tasty that flavor is. Our goal will be to help Brock figure out the best set of sweets to purchase—that is, the set of sweets with the maximum sum of their tasty values while following the shop owner's rule.

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):



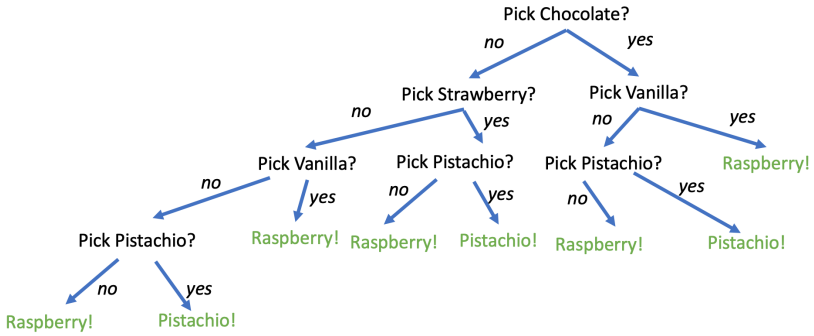
Look at the following code that gives a naive recursive solution to computing the max tastiness. This version assumes that the tastiness scores for these flavors are stored in an array. Try to convince yourself that it would compute the same answer that you worked out for the concrete example above.

Note: this is slightly different code than given in the handout. This code computes just the maximum tastiness, NOT the list of the candies that got us to the tastiness. That is covered in the next section.

```
1  class pickSweets:
2
3      def __init__(self, sweets_lst : "list[tuple[str, int]]"):
4          self.sweets_list = sweets_lst
5
6      def tail(self, lst : list) -> list:
7          return lst[1:]
8
9      def first(self, lst : list):
10     return lst[0]
11
12
13     # the input is the tastiness array -- a list of (sweet name, tastiness) tuples
14     # the output is the maximum tastiness of that input array
15     def tastiness_helper(self, lst : "list[tuple[str, int]]") -> int:
16         if len(lst) == 0:
17             return 0
18         elif len(self.tail(lst)) == 0:
19             return self.first(lst)[1] # the tastiness is the second item in the tuple
20         else:
21         pick_this = self.first(lst)[1] +\
22           self.tastiness_helper(self.tail(self.tail(lst)))
23             skip_this = self.tastiness_helper(self.tail(lst))
24             if pick_this > skip_this:
25                 return pick_this
26             else:
27                 return skip_this
28
29     def max_tastiness(self) -> int:
30         return self.tastiness_helper(self.sweets_list)
31
32 our_sweets_lst = [("choc", 3), ("straw", 10), ("vanilla", 12), ("pistachio", 16), ("rasp", 4)]
33 sweets_picker = pickSweets(our_sweets_lst)
34 print(sweets_picker.max_tastiness())
```

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:

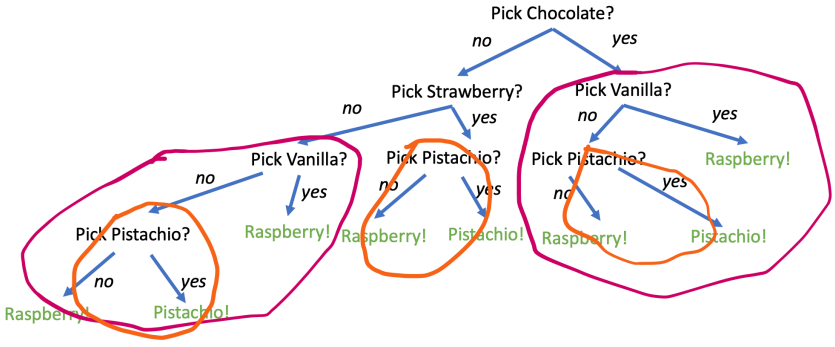


Stop and Think: Now that you see the tree, what is the running time of this code? Think about how many nodes are in this tree.

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?

If we look down the tree, we see that smaller redundant computations fit inside larger redundant computations. That is, to compute whether we pick vanilla, we first compute whether to pick pistachio. It seems like the solution is to work from the end of the list backwards, first computing which sweets to pick in the {Raspberry} sublist, then which sweets to pick in the {Pistachio, Raspberry} sublist, then {Vanilla, Pistachio, Raspbery}, and so on. We need to have a way of storing the previously computed results and associate it with the sublist (for example, associate the sublist starting with Raspberry with the tastiness of 4, the sublist starting with Pistachio with the tastiness of 16, the sublist starting with Vanilla with the tastiness of 16, and so on).

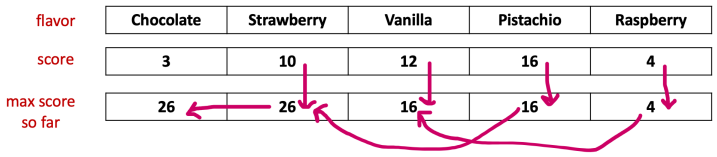
Many problems like this store the "previously computed" data in arrays. This makes sense when there is already a way to associate the items with array indices. For the sweets problem, we can use position in

the sequence of flavors as the indices. Hence, we would store the tastiness result of the sublist starting with Vanilla at index 2 of the array, because Vanilla is at index 2 of the input array.

1.2 Using Iteration instead of Recursion

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):



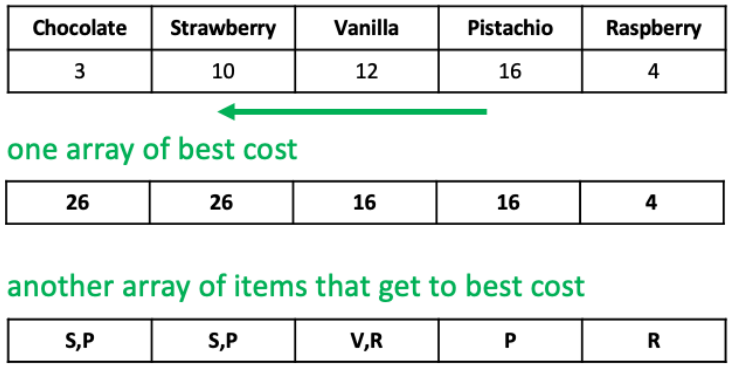

```
Given the three arrays above (max score starts out as all 0, others are given)
- initialize the two rightmost spots of max-score from the score array
- for each index i from length - 3 down to 0
    max-score[i] = max(score[i] + max-score[i + 2], max-score[i+1])
- the overall result is in score[0]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

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):



3 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`