Introduction to Dynamic Programming

Kathi Fisler and Milda Zizyte

November 7, 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 analyze a 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.

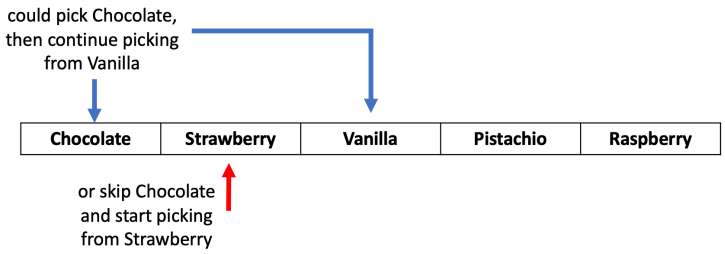
Each flavor of sweet has a positive (non-negative and nonzero) tastiness rating based on how tasty that flavor is. Our goal will be to help Brock figure out the best set of sweets to purchase—that is, the set of sweets with the maximum sum of their tasty values while following the shop owner's rule.

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):



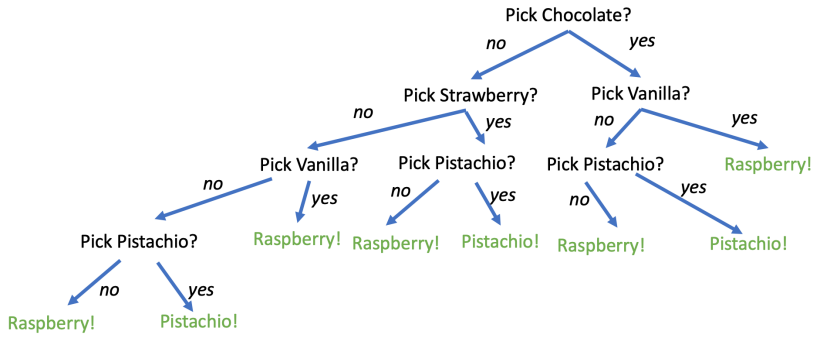
Look at the following code that gives a naive recursive solution to computing the max tastiness. This version assumes that the tastiness scores for these flavors are stored in a list. Try to convince yourself that it would compute the same answer that you worked out for the concrete example above.

```
class pickSweets:

    def __init__(self, sweets_lst : "list[tuple[str, int]]"):
        self.sweets_list = sweets_lst

    def flavor(self, i) -> str:
        return self.sweets_list[i][0]

    def rating(self, i) -> int:
        return self.sweets_list[i][1]

    def pick_sweets_recursive(self, start_ind: int) -> int:
        ''' returns the optimal tastiness rating for the sublist of sweets_list
            starting with start_ind '''
        if start_ind == len(self.sweets_list) - 1: # last element
            return self.rating(start_ind)
        elif start_ind == len(self.sweets_list) - 2: # second-to-last element
            # two choices: pick this, or skip it and choose the next element
            # (which is the last in the list)
            return max(self.rating(start_ind), self.rating(start_ind) + 1)
        else:
            # pick this and get the optimal rating after skipping the next element:
            pick_this_rating = self.rating(start_ind) +\\
                self.pick_sweets_recursive(start_ind + 2)

            # skip this and get the optimal rating starting with the next element:
            skip_this_rating = self.pick_sweets_recursive(start_ind + 1)

            # the choices are to pick or to skip; choose the best one:
            return max(pick_this_rating, skip_this_rating)

    def pick_sweets(self) -> int:
        ''' picks a subset of sweets according to maximum tastiness rating,
            satisfying the constraint that two sweets cannot be adjacent'''
        return self.tastiness_recursive(0)

our_sweets_lst = [("choc", 3), ("straw", 10), ("vanilla", 12), ("pistachio", 16), ("rasp", 4)]
```

```
38 sweets_picker = pickSweets(our_sweets_lst)
39 print(sweets_picker.pick_sweets())
```

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:

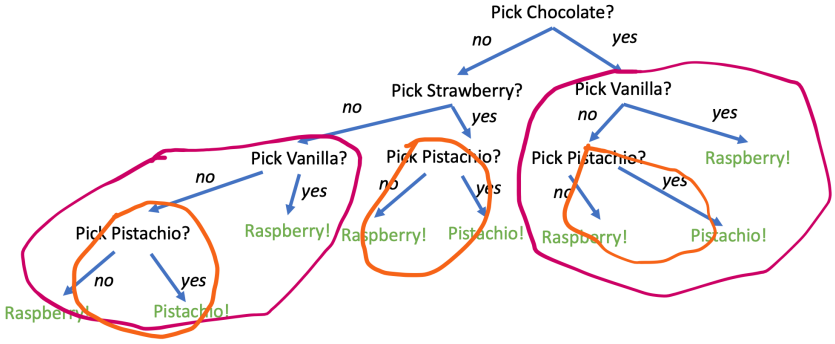


Stop and Think: Now that you see the tree, what is the running time of this code? Think about how many nodes are in this tree.

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.

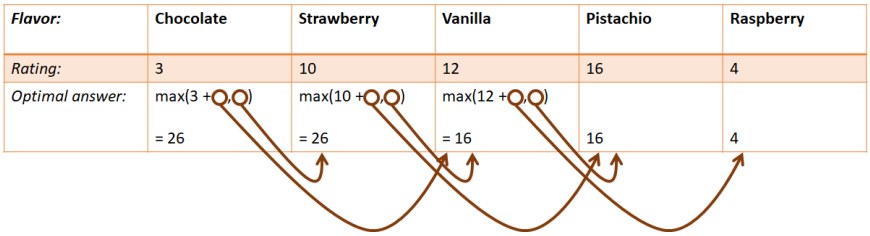


If we are worried about runtime here, perhaps we could avoid repeating a computation to save time. Specifically, if we actually expanded out the recursive calls only once per flavor, somehow saving the results of each call, this computation could be done in linear time instead.

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).

If we move backwards through the list (initializing the optimal value for the {Raspberry} sublist to be 4 and the optimal value for the {Pistachio, Raspberry} sublist to be 16), we can use the pre-computed values to directly compute the next values:



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. We will explore what this looks like in code in the next lecture!