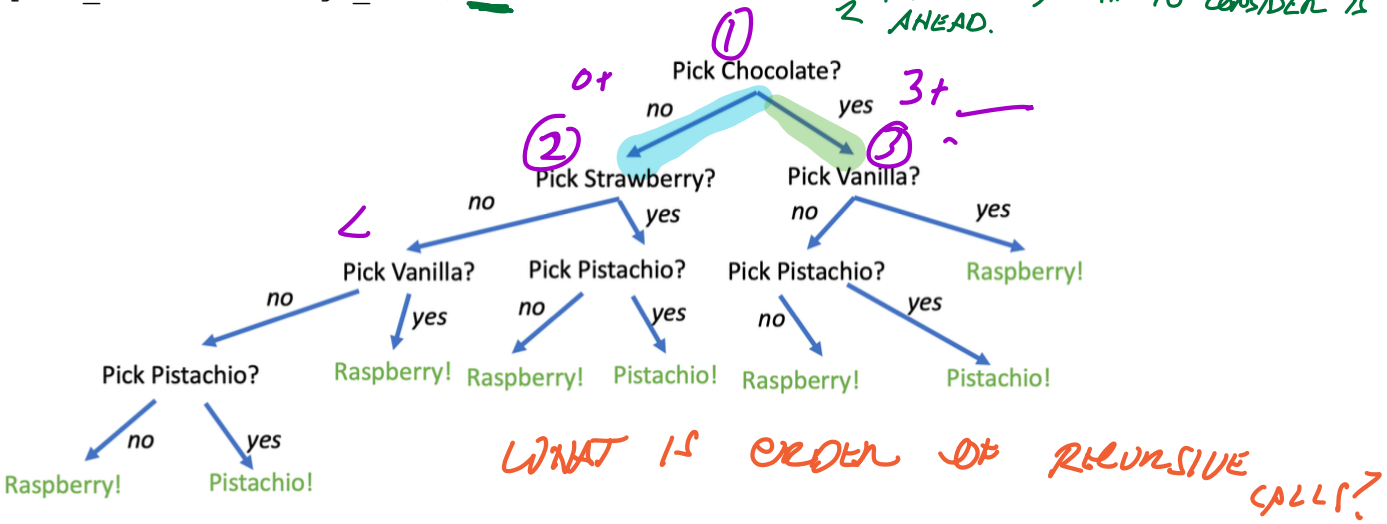
Problem statement: (last lecture) Write a program to compute the maximal total rating (a number) of flavors that you can buy, under the constraint that you cannot select adjacent flavors.

Flavor	Chocolate	Strawberry	Vanilla	Pistachio	Raspberry
Rating	3	10	12	16	4
Optimal answer					

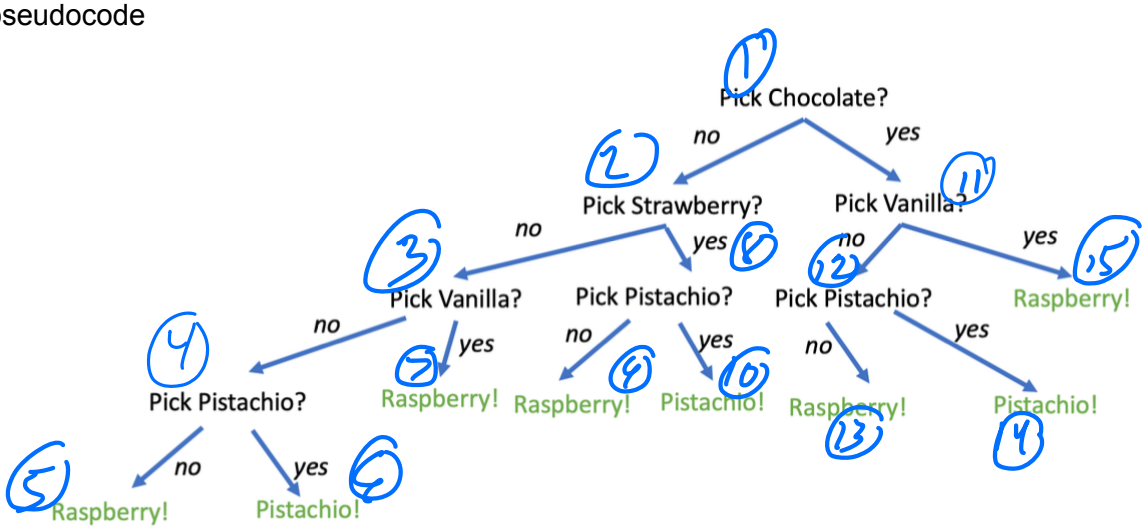
```
def pick_sweets(ratings, from_pos) -> int:
    if from_pos == len(ratings) - 1:
        return ratings[from_pos]
    else if from_post == len(ratings) - 2:
        return max(ratings[from_pos], ratings[from_pos + 1])
    else return max(pick_sweets(ratings, from_pos + 1),
                    ratings[from_pos] + pick_sweets(ratings, from_pos + 2))

pick_sweets(ratings_list, 0)
```



Try it: what is the order of the recursive calls?

Label the following decision tree with the order in which each recursive call happens while running the pseudocode

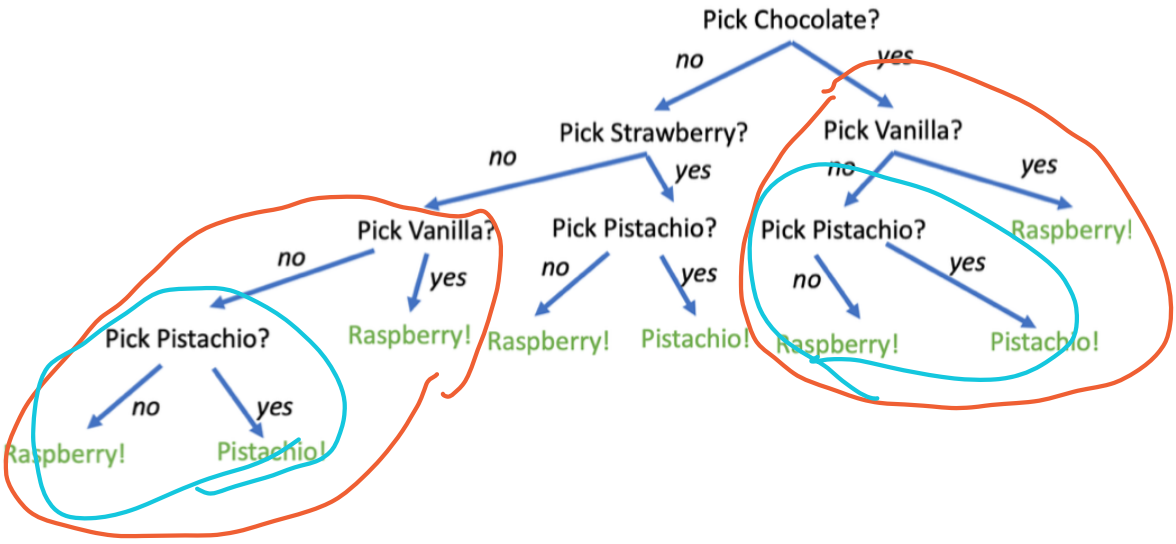


For N flavors, we end up with 2^N nodes in the tree. The recursive version visits all of them => O(2^N) runtime

Pseudocode:

```
pick_sweets(ratings, from_pos) -> int:
    if from_pos == len(ratings) - 1:
        return ratings[from_pos]
    else if from_pos == len(ratings) - 2:
        return max(ratings[from_pos], ratings[from_pos - 1])
    else return max(pick_sweets(ratings,from_pos + 1),
                    pick_sweets(ratings, from_pos + 2) + ratings[from_pos])


pick_sweets(ratings_list, 0)
```



=> Same computation, same answer

When we look to make code faster, one thing we look for is repeated computations

To speed up, two choices:

1. **Remember results we've already computed and reuse them**
2. **Rewrite code so that each computation is only done once**

=> We focus on option 2

How do we break down option 2?

```
pick_sweets(ratings, from_pos) -> int:
    if from_pos == len(ratings) - 1:
        return ratings[from_pos]
    else if from_pos == len(ratings) - 2:
        return max(ratings[from_pos], ratings[from_pos - 1])
    else return max(pick_sweets(ratings,from_pos + 1),
                    pick_sweets(ratings, from_pos + 2) +
ratings[from_pos])
```

```
pick_sweets(ratings_list, 0)
```

END ← LOOP ← START

BASE CASE BASE CASE

Flavor	Chocolate	Strawberry	Vanilla	Pistachio	Raspberry
Rating	3	10	12	16	4
Optimal answer	max(3+__, __) = 26	max(10 + __, __) = 26	max(12 + __, __) = 16	max(16, __) = 16	4

0 1

⇒ COMPUTING EACH CELL DEPENDS ON PREVIOUS RESULTS

Goal: compute everything exactly once

If we read from the right edge back—we know raspberry, it doesn't depend on anything

Pistachio: rating for raspberry, which I already know

Vanilla: I need pistachio, which I already know, and raspberry

=> Could start at the base cases, save the values, work our way backwards

Note: it's okay if this still seems weird right now—we'll build an example in the next lecture (also see typed notes for more details)

Dynamic programming: building a strategy

- **Set up an array of results,**
- **Array has same length as number of flavors**
- **Fill in last cell of array with rating (rasp)**
- **Fill in next to last cell with max of last 2 cells**
- **Loop backwards through array to fill in each result based on the next two cells in the array**

This seems annoying to write. . . Let's ask ChatGPT