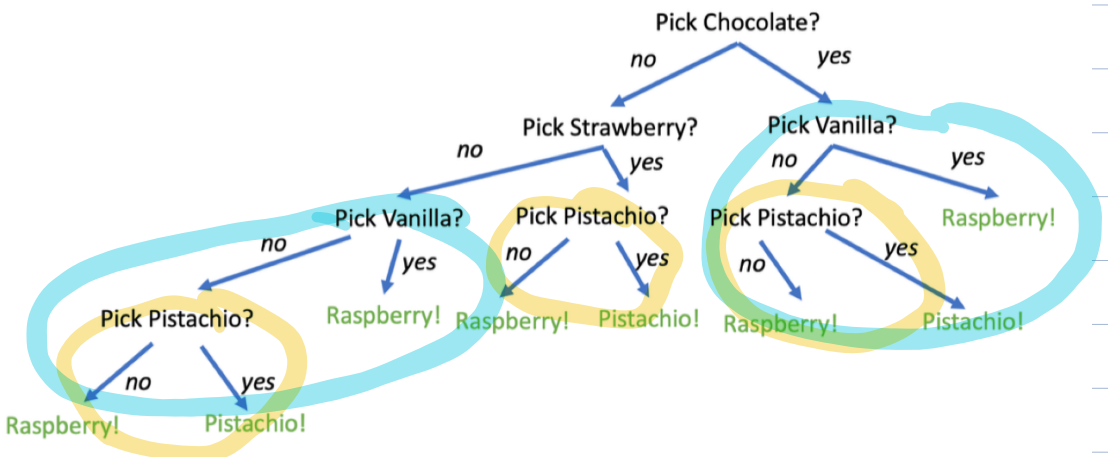
CHOC	STRAWBERRY	VANILLA	PISTACHIO	RASPBERRY
3	10	12	16	4

Problem: write a program to select the maximum total rating of candies you can buy, with constraint that you can't select adjacent candies

Initial idea: to solve, break problem down into smaller problems, solve smaller problems
=> Recursive version, works, but has lots of repeated computations
=> involves solving the subproblem multiple times



=> Dynamic programming

The basic idea:
- Store the result of each subproblem as you go (list, map, etc.)
- When you compute each subproblem, use the results from previous subproblems to find result

=> Certain types of programs can take advantage of this style of programming

How it works
- Loop through array right to left

- At each step i, keep track of max rating seen so far
make an array (best) which stores the maximum rating seen so far (ie, from i to end of array)

- At end, result is in best[0]

Problem-specific part:
for any candy you consider:
- Pick that candy
- Don't pick it
=> This affects what you pick after that

INDEX	0	1	2	3	4
ratings	C 3	S 10	V 12	P 16	R 4
best	6	26	16	16	4
choices	S,P	S,P	P *	P	R

(3) (2) (1) (0)

(0) *First step is base case => max rating if array was size 1 (only raspberry)*

BEST = 4

```
best[4] = ratings[4]
```

PICK: R (4)
DON'T: — (0)

(1) *Consider array of size 2: [P, R], need to choose max*

BEST = MAX(16, 4)

```
best[3] = max(ratings[3], ratings[4])
best[3] = max(ratings[3], best[4]) (could also write this)
```

PICK: P (16)
DON'T: R (4)

To find the max rating, we use best to look up results we've previously computed!
- Pick: consider current rating + max rating two steps ahead (can't pick adjacent)
- Don't pick: consider max if we ignored this slot (rest of array)

(2) *Consider array size 3: [V, R, P]*

BEST = MAX(12+4, 16) = 16
(ALSO WRITE AS...) = MAX(12+4, MAX(16,4))

PICK: V+R (16)
DON'T: P

```
best[2] = max(ratings[2] + best[4], best[3])
```

Consider array size 4: [S, V, R, P]

(3) BEST = MAX(10+16, 16)
= MAX(10+MAX(16,4), MAX(12+4, MAX(16,4)))

PICK: S+P
DON'T: (V+R) OR P

```
best[1] = max(ratings[1] + best[3], best[2])
```

(4) BEST = MAX(3+16, 26)
= MAX(3+ . . . , . . .)

PICK: C+ [ ]
DON'T: [ ]

```
best[0] = max(ratings[0] + best[2], best[1])
```

After working this out on paper, we can write a general-case solution by inspection (see page 5 for code):

```
best[i] = max(ratings[i] + best[i + 2], best[i + 1])
```

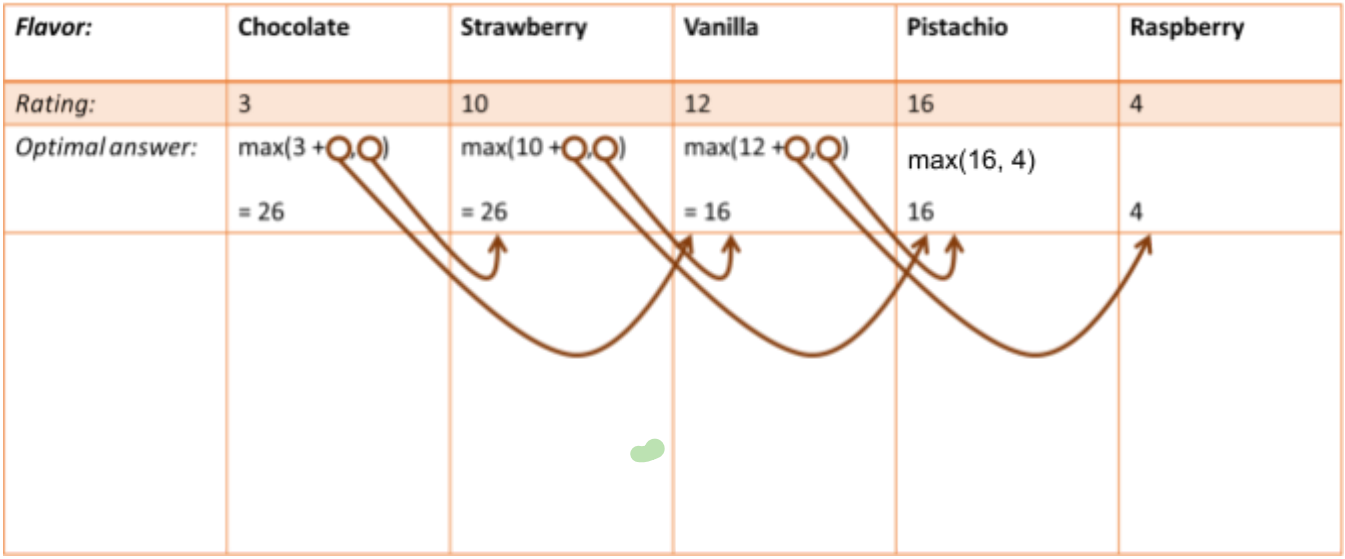
General strategy for thinking about these types of problems
- Start with the base case (simplest solution you already know)
- Starting from base case, write out expected result and write it down (what goes in 'best' array)
- Continue for more steps, using past info you wrote down
- Try and write a general solution for how to find any result based on your stored info

When do we use dynamic programming?
- Use when you can break original problem down into subproblems such that the subproblem is the same no matter when you compute it

The function must be a "function" in the mathematical sense, meaning it doesn't modify any variables that compute the subproblem

Topic: Adding Path Tracking

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, 4) 16	4



(This example is from ChatGPT. See next page for a version of the code that matches the example on page 2, which we did in class.)

```
 # ChatGPT solution (last class)
def maximize_total_rating(sweets):
    n = len(sweets)
    if n == 0:
        return 0
    elif n == 1:
        return sweets[0]

    # Initialize a list to store the maximum total ratings for each position
    max_ratings = [0] * n
    max_ratings[0] = sweets[0]
    max_ratings[1] = max(sweets[0], sweets[1])

    for i in range(2, n):
        # Calculate the maximum total rating for the current position
        max_ratings[i] = max(max_ratings[i - 1], max_ratings[i - 2] + sweets[i])

    return max(max_ratings[-1], max_ratings[-2])

# Example usage
sweets = [3, 10, 12, 16, 4]
result = maximize_total_rating(sweets)
```

```python
# A right-to-left solution
def max_sweets_dp_rtl(ratings):
    if len(ratings) == 1:
        return ratings[0]

    best = [0] * len(ratings)

    best[-1] = ratings[-1]  # Base case (last item)
    best[-2] = max(ratings[-1], ratings[-2])  # Base case (second to last)

    # Fill in remaining spots, working right to left
    for i in range(len(ratings) - 3, -1, -1):
        best[i] = max(ratings[i] + best[i + 2],
                      best[i + 1])

    return best[0]
```