Ally Smith

Dynamic Programming Project

1. **Recursive Implementation**

   Below is the code I used for my recursive implementation of the algorithm. When run with the parameters $p = 3,\ t = 16$ the function is called 753,665 times.

```python
# recursive algorithm
def recursive(self, p: int, t: int) -> int:
    self.recursive_counter += 1  # increment counter for report
    # base cases
    if (t == 0 or t == 1):
        return t
    if (p == 1):
        return t

    # general cases for all values of x from 1 to t, inclusive
    results = []  # list to hold calculated costs, used to choose minimum
    for x in range(1, t + 1):
        breaksCase = self.recursive(p=p - 1, t=x - 1)
        intactCase = self.recursive(p=p, t=t - x)
        maxThrows = max(breaksCase,
                        intactCase)  # maximum of the two cases is chosen
        results.append(maxThrows)
    results.sort()  # sorts in place in ascending order
    return 1 + results[
        0]  # gets the minimum value from all values of x + 1
```

2. **Recursive Runtime Analysis** (measured in ms)

$\frac{t}{p}$	10	12	15	18	20
2	1	2	20	170	716
4	7	39	499	5,942	29,772
8	13	117	2,958	65,222	
12	13	134	3,398		

   As can be seen in the table above, the runtime increases greatly with slight changes in t. Additionally, higher values of p begin to affect the runtime more as t increases. For example, the runtime of $p = 8,\ t = 10$ is fairly close to that of $p = 4,\ t = 10$, but $p = 8,\ t = 15$ is far greater than $p = 4,\ t = 15$.

3. **Dynamic Programming Implementation**

Below is the code I used for my dynamic programming implementation of the algorithm.

```python
    # dynamic programming algorithm
    def dynamic(self, p: int, t: int):
        self.dynamic_counter += 1  # implement counter for report
        # base cases
        if (t == 0 or t == 1):
            self.table[p - 1][t - 1] = t
            return
        if (p == 1):
            self.table[p - 1][t - 1] = t
            return

        # general case for all values of x from 1 to t
        results = []
        for x in range(1, t + 1):
            # if the value is none, it is calculated by calling the dynamic function on that spot
            if self.table[p - 2][x - 2] == None:
                self.dynamic(p=p - 1, t=x - 1)
            # after that, the value is read for comparison
            breaks_case = self.table[p - 2][x - 2]

            # same as above but for the case where the pumpkin stays intact
            if self.table[p - 1][t - x - 1] == None:
                self.dynamic(p=p, t=t - x)
            intact_case = self.table[p - 1][t - x - 1]

            max_throws = max(breaks_case, intact_case)
            results.append(max_throws)
        results.sort()

        # assigns minimum value to the table slot corresponding to p and t
        self.table[p - 1][t - 1] = 1 + results[0]
```

4. **Dynamic Runtime Analysis** (measured in ms)

$\frac{t}{p}$	20	40	60	80	100	120	140	160	180	200
20	1	6	14	26	41	64	86	114	141	176
40	1	7	20	39	70	107	148	202	256	328
80	1	7	20	47	89	143	217	308	411	538
120	1	7	21	47	91	153	239	353	481	634
160	1	7	21	48	91	153	239	359	517	695

As can be seen in the above table, growth in t leads to an increased runtime with p held constant. Similarly, an increase p can lead to an increased runtime, however this value seems to max out as $p \to t$. If p and t are increasing simultaneously, then the runtime will increase dramatically.

5. **Traceback Step**

```python
# input: num of pumpkins and targets to be traced
# call after filling table for correct functionality
def traceback(self, p: int, t: int):
    # base cases
    if (t == 0 or t == 1):
        return [t]
    if (p == 1):
        return list(range(1, t + 1))
    min_val = 1e9
    min_x = -1
    for x in range(1, t + 1):
        breaks_case = self.table[p - 2][x - 2]
        intact_case = self.table[p - 1][t - x - 1]
        max_val = max(breaks_case, intact_case)
        if max_val <= min_val:
            min_val = max_val
            min_x = x
            multiplier = -1 if breaks_case > intact_case else 1

    if multiplier == -1:  # breaks case
        return self.traceback(p=p - 1, t=min_x - 1) + [multiplier * t]
    else:  # intact
        return self.traceback(p=p, t=t - min_x) + [multiplier * t]
```

6. **Traceback Verification**

My output for $p = 5,\ t = 100$ is as follows:

7

1, 3, 7, 15, 30, -43, 100

Appendix

Here are some of the other functions I wrote for this project to set up my algorithm.

```python
class Algorithm:
    # initializes all values in the table to None
    def init_table(self):
        self.table = [[None] * self.targets for _ in range(self.pumpkins)]

    def __init__(self, pumpkins: int, targets: int):
        self.recursive_counter = 0
        self.dynamic_counter = 0
        self.pumpkins = pumpkins
        self.targets = targets
        self.init_table()

    # prints out values in the table separated by a tab character
    def print_table(self):
        for r in range(self.pumpkins):
            for c in range(self.targets):
                print(self.table[r][c], end="\t")
            print()

    # runs the dynamic algorithm on each spot in the table
    def fill_table(self) -> None:
        for r in range(self.pumpkins):
            for c in range(self.targets):
                self.dynamic(p=r + 1, t=c + 1)
```