Ally Smith

Dynamic Programming Project

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
```

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 $p = 4,\ t = 15$.

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]
```

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.