Trebuchet Project

Algorithms Project 3 - Liam Dempsey (10754465)

1. RECURSIVE ALGORITHM:

The following code is a recursive solution to the Trebuchet problem, implementing the specified recurrence relation and base cases from lecture:

$$T(p,t) = 1 + \min_{1 \le x \le t} \left(\max \left[T(p-1, x-1), T(p, t-x) \right] \right)$$

```python
def trebuchet(p, t):
    # Base cases
    if p == 1 or t <= 1: return t

    # Check best case for each target
    minthrows = t + 1
    for x in range(1, t + 1):
        broken = trebuchet(p - 1, x - 1)
        intact = trebuchet(p, t - x)

        minthrows = min(max(broken, intact), minthrows)

    return 1 + minthrows
```

Executing this algorithm with the initial parameters, p = 3 and t = 16 yields a value of 5 upon completion.

To calculate this value, this function is recursively called 753,665 times, including the initial call.

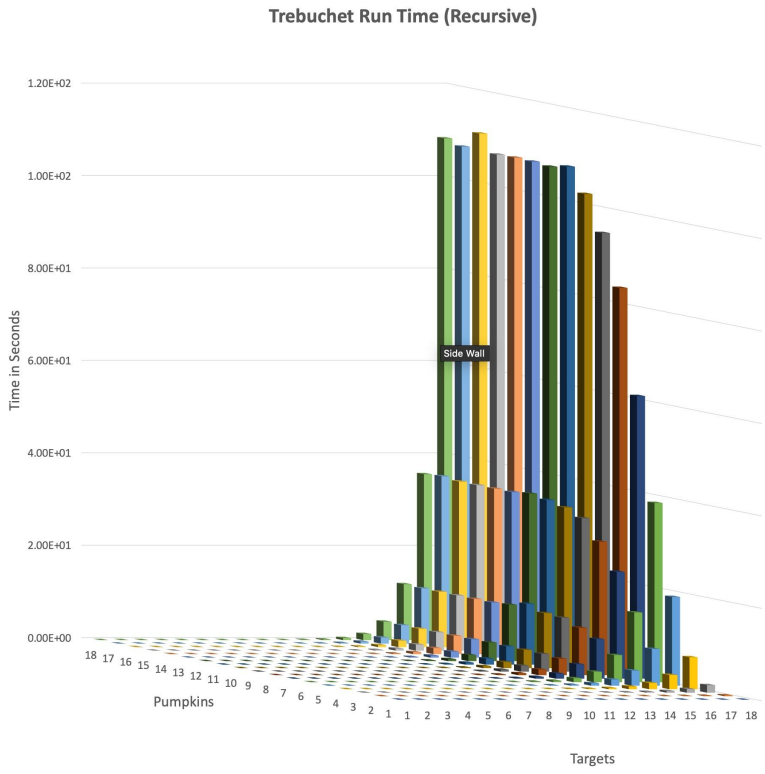
2. GROWTH COMPLEXITY (RECURSIVE):

As indicated by the chart, the recursive trebuchet implementation grows in complexity at an astronomical rate, primarily driven by the number of targets. By rough estimation, this solution has an asymptotic complexity of $O(n^n)$.

This rate of growth makes it hard to collect a significant number of data points, as each increasing value of n (or t in this example) requires considerably more time to compute than the last.

This chart was generated for every combination of $1 \le p \le 18$ and $1 \le t \le 18$. To provide a bit of perspective, each of the largest inputs took nearly two minutes to compute.

**Due to time constraints, only one trial per input combination was performed, which corresponds to slight variations in the data from what would be expected.*



3. DYNAMIC PROGRAMMING ALGORITHM:

The following code is my dynamic programming implementation of the Trebuchet problem, written for Python 3.

```
def trebuchetdp(p, t):
    # Build an empty table
    table = [[0 for i in range(t + 1)] for j in range(p)]

    # Set up the base case values
    for i in range(1, t + 1): table[0][i] = i
    for i in range(1, p): table[i][1] = 1

    # Compute recursive entries
    for row in range(1, p):
        for col in range(2, t + 1):
            entrymin = t

            for x in range(1, col + 1):
                broken = table[row - 1][x - 1]
                intact = table[row][col - x]

                entrymin = min(max(broken, intact), entrymin)

            table[row][col] = entrymin + 1

    return table[p - 1][t]
```

The table generated in this algorithm differs slightly from that of the implied results table. The rows correspond to pumpkins and the columns, targets, however the rows are indexed by p - 1. The reason for this is the nature of pythonic lists, which are indexed beginning at zero. The lowest value of P we will compute is one, so the table is mapped accordingly.

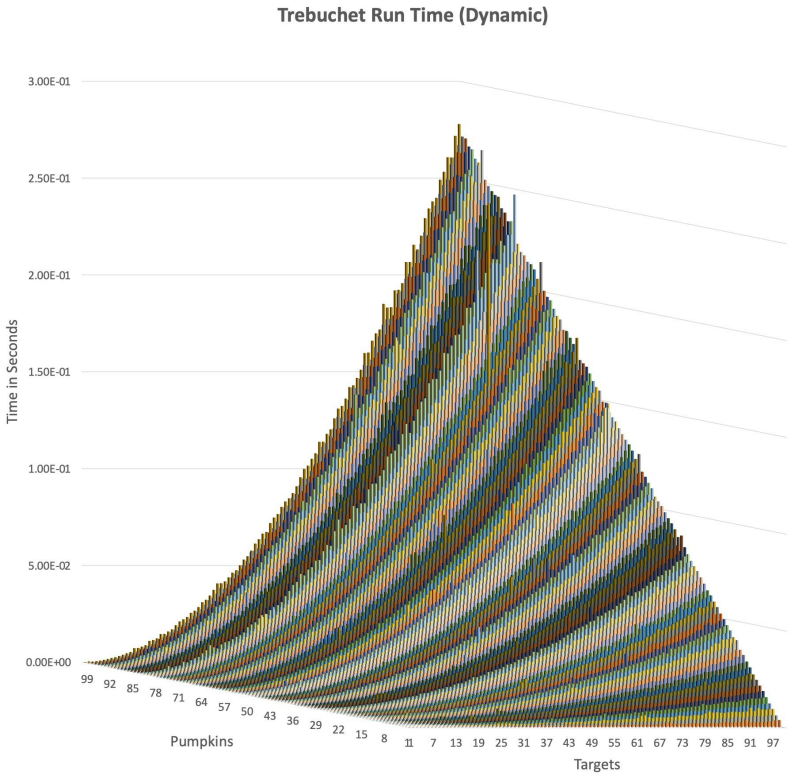
Executing this function with the parameters: p = 3, t = 16, yields the following table:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
[0, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6]
[0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 5, 5]
```

4. GROWTH COMPLEXITY (DYNAMIC):

Since the dynamic programming implementation has a considerably better complexity–$O(n^3)$–we are able to collect considerably more data points to characterize the growth of the function. The following table tests every combination of inputs where $1 \leq p \leq 100$ and $1 \leq t \leq 100$.

**Due to time constraints, only one trial per input combination was performed, which corresponds to slight variations in the data from what would be expected.*



5. TRACEBACK STEP:

The following code is an updated version of the dynamic programming implementation that includes the traceback functionality. This enables the program to compute the series of targets attempted by the trebuchet by taking advantage of the existing table of values.

My implementation of the traceback generates a secondary "traceback" table which I use to store the intermediate 'best target' values. This allows me to easily compute both, what the next lookup entry should be, and the respective 'target offset' as the algorithm does not otherwise account for it.

Running the *updated* algorithm with parameters: p = 2 and t = 4, generates the following *traceback* table:

```
[0, -1, 1, 1, 1]
[0, -1, 1, -2, 1]
```

Where, as before, the rows correspond to the number of pumpkins left to throw, and the column the number of targets left to check.

6. TRACEBACK VERIFICATION:

Running the traceback algorithm with parameters p = 5, t = 100 yields the following output:

```
3
1 -3 -2
```

```python
def trebuchetdp(p, t, output=False):
    # Build an empty table
    table = [[0 for i in range(t + 1)] for j in range(p)]  # Primary table
    tb = [[0 for i in range(t + 1)] for j in range(p)]  # Traceback table

    # Set up the base case values
    # Primary table
    for i in range(1, t + 1): table[0][i] = i
    for i in range(1, p): table[i][1] = 1

    # Traceback table
    for i in range(2, t + 1): tb[0][i] = 1
    for i in range(0, p): tb[i][1] = -1

    # Compute recursive entries
    for row in range(1, p):
        for col in range(2, t + 1):
            entrymin = t
            besttarget = 0

            for x in range(1, col + 1):
                broken = table[row - 1][x - 1]
                intact = table[row][col - x]

                tmp = max(broken, intact)
                if tmp < entrymin:
                    entrymin = tmp
                    besttarget = x * ((-1) if broken >= intact else 1)

            table[row][col] = entrymin + 1
            tb[row][col] = besttarget

    # Traceback step
    targets = []
    offset = 0
    row = p - 1
    col = t

    while True:
        # Check if we're done
        if col <= 0: break

        # Determine the next entry
        entry = tb[row][col]
        x = abs(entry)
        if entry < 0 and row > 1:
            row -= 1
            col -= col - x + 1

        else:
            col -= x

        # Append to the list and update the offset
        targets.append((x + offset) * ((-1) if entry < 0 else 1))
        if entry > 0: offset += x

    return table[p - 1][t], targets
```