Colorado School of Mines

Trebuchet

Chris Deal
Roll: *Student*
Class: *CSCI 406*
Session: *Fall 2021*
Email: *cdeal@mines.edu*

Course: *Algorithms* – Teacher: *Dinesh Mehta*
Submission date: *November 13, 2021*

Contents

Trebuchet Problem

The Mines E-Days Trebuchet Competition has been given a new twist. This year, competitors will use their apparatus to throw pumpkins at a series of targets with the goal of hitting the furthest target possible without the pumpkin exploding on impact. There are t targets to be attempted, each 1-meter further than the previous. To make things more interesting, competitors are given only a limited amount of pumpkins, denoted by p. A pumpkin can be reused for multiple throws as long as it remains intact. Thus if p = 1, the only sure way to find the maximum distance is to throw at the first target, then the second, then the third, and so on until the pumpkin breaks or the t-th target is hit successfully. In the worst case, this strategy would take t throws. However, larger values of p allow for some pumpkins to be strategically put at risk in order for their maximum throwing distance to be more efficiently discovered. The end goal of the trebuchet problem is to find the smallest number of throws necessary to determine the maximum possible distance in the worst-case for given values of p and t.

ASSUMPTIONS:

1. The trebuchet can be calibrated exactly to hit the intended target no matter how far.
2. A pumpkin that survives a throw can be used again. Assume no damage is done to the
3. A pumpkin that shatters from being thrown too far cannot be reused.
4. All pumpkins are equal. They share the same maximum throwing distance and will explode if thrown 1 meter further.
5. If a pumpkin survives a throw, it would survive a shorter throw.
6. It is possible that a pumpkin shatters at the 1st target. Then the maximum distance is 0 meters.
7. It is also possible that a pumpkin survives a throw to the t-th target. Since there are no further targets, the maximum distance is t meters.

Recurrence Relation:

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

$$\text{Base Cases: } \begin{cases} T(p,0) = 0 \\ T(p,1) = 1 \\ T(1,t) = t \end{cases}$$

The full code base for this project may be examined in the appendix [**1**]

Recursive Implementation:

```
    private int recursiveSoln(int p, int t) {
        calls++;
        int survive, smash, max, min;
        min = Integer.MAX_VALUE;

        if (t == 0) return 0;
        if (t == 1) return 1;
        if (p == 1) return t;

        for (int x = 1; x <= t; x++) {
            smash = recursiveSoln(p - 1, x - 1);
            survive = recursiveSoln(p, t - x);
            max = Math.max(smash, survive);
            if (max < min) min = max;
        }
        return 1 + min;
    }
```

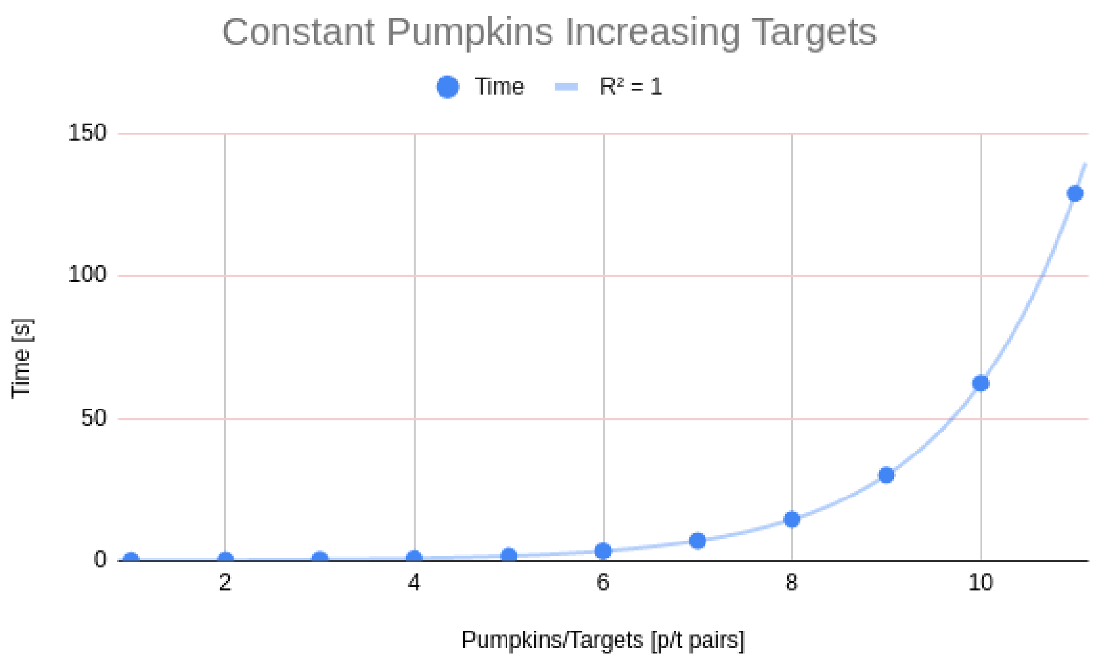
This project was written in Java. This recursive algorithm strictly followed the given recurrence relation for the trebuchet problem. The recursiveSoln(...) method took in the values for the supplied pumpkins and targets as parameters. A calls integer was incremented each time this method was called. Variables were initialized each time the method was called which reset them appropriately. The base cases are contained in the if-conditional statements which short circuited if the condition was met and returned the appropriate value out of the method call. If it was not a base case, the for loop iterated through all possible target combinations with successive recursive calls that each also were evaluated and terminated at base cases until the maximum of of the survival or no survival (smash) cases could be determined. For each recursive call the min of these two max cases was assigned and then 1 was added and this was returned for each pumpkin and target combination that was not returned in the base cases.

For the case of 3 pumpkins and 16 targets, there were 753,365 recursive calls. Please see the Appendix for the full call breakdown[2].

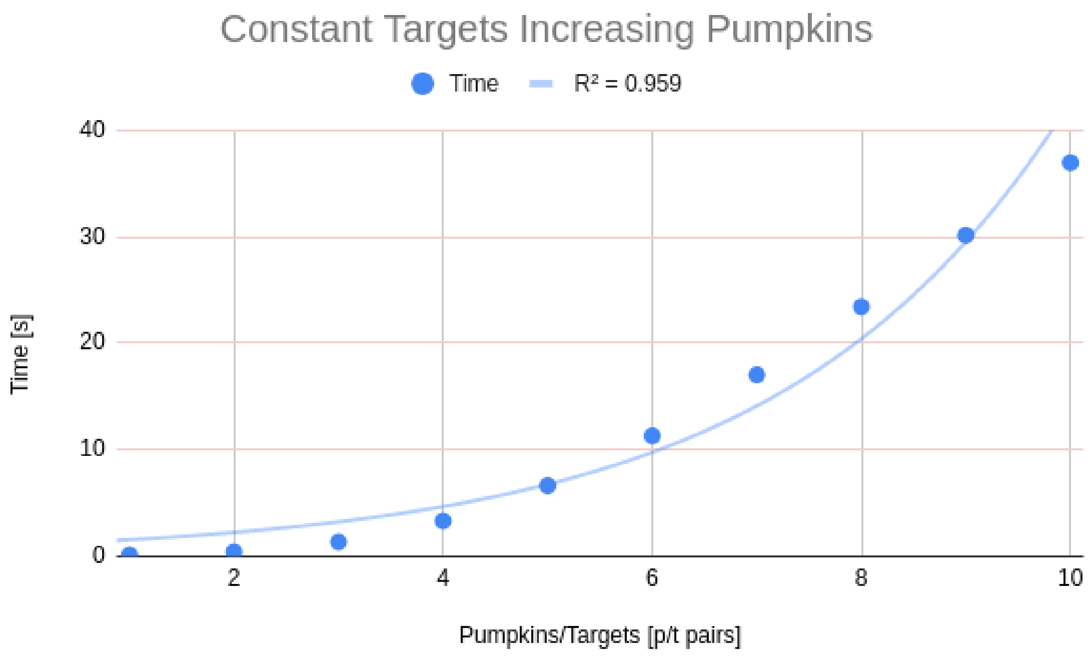
Recursive Algorithm Complexity Analysis

I ran three tests to determine the run time complexity. The first test held pumpkins constant while increasing targets. The second test held targets constant while increasing pumpkins, and the final incrementally increased targets and pumpkins together by a factor of one.

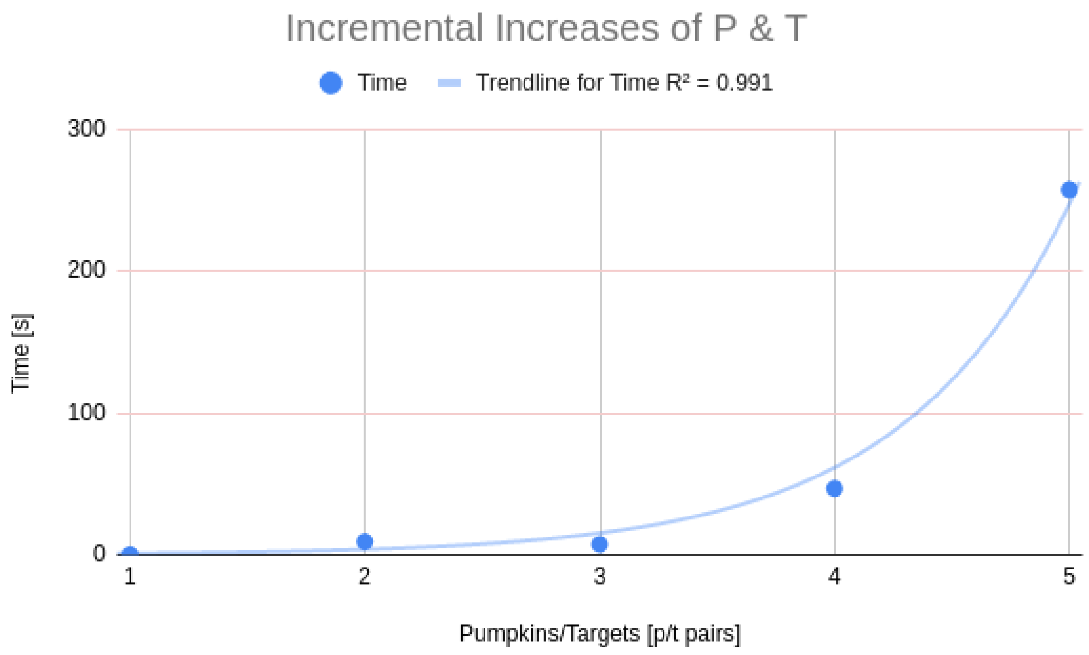
(Pumpkins,Targets)	Time(s)
(3,20)	0.089
(3,21)	0.178
(3,22)	0.373
(3,23)	0.776
(3,24)	1.617
(3,25)	3.352
(3,26)	6.969
(3,27)	14.488
(3,28)	30.034
(3,29)	62.255
(3,30)	128.940



(Pumpkins,Targets)	Time(s)
(3,20)	0.085
(4,20)	0.407
(5,20)	1.333
(6,20)	3.302
(7,20)	6.621
(8,20)	11.301
(9,20)	17.025
(10,20)	23.414
(11,20)	30.134
(12,20)	36.944



(Pumpkins,Targets)	Time(s)
(3,20)	0.118
(4,21)	0.922
(5,22)	7.314
(6,23)	46.677
(7,24)	257.389



From observation it would appear that increasing targets causes a worse time complexity than simply increasing pumpkins. The first test shows an exponential relationship when p was held constant, but the next test where t was held constant showed a cubic relationship. The final test, where t and p were both increased by one, showed t dominating p. In conclusion, increases in t have a harsher impact on the recursion run time complexity.

Dynamic Implementation

```
private void dpSoln(int p, int t){
    boolean deathFlag = false;
    int survive, smash, max, min;
    min = Integer.MAX_VALUE;
    resize(optimalThrowTable, p+1, t+1); //Size tables
    resize(x_Table, p+1, t+1);

    for (int i = 1; i <= p; i++) {
        for (int j = 0; j <= t; j++) {
            if (j == 0) optimalThrowTable.get(i).add(0);
            else if (j == 1) {
                optimalThrowTable.get(i).set(j, 1);
                x_Table.get(i).set(j, -1);
            } else if (i == 1) {
                optimalThrowTable.get(i).set(j, j);
                x_Table.get(i).set(j, 1);
            } else {
                for (int x = 1; x <= j; x++) {
                    smash = optimalThrowTable.get(i - 1).get(x - 1);
                    survive = optimalThrowTable.get(i).get(j - x);

                    if (smash >= survive) {
                        max = smash;
                        deathFlag = true;
                    } else {
                        max = survive;
                    }
                    if (max < min) {
                        min = max;
                        if (deathFlag) {
                            x_Table.get(i).set(j, x * -1);
                            deathFlag = false;
                        } else x_Table.get(i).set(j, x);
                        }
                        optimalThrowTable.get(i).set(j, 1 + min);
                    }
                }
            min = Integer.MAX_VALUE;
            }
        }
}
```

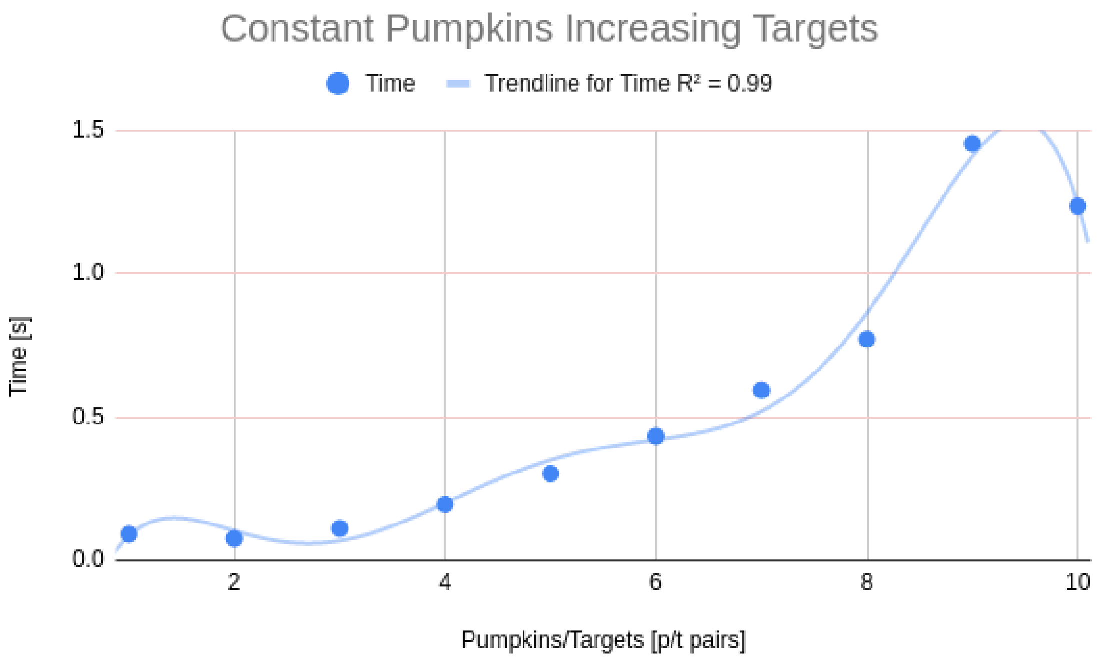
This algorithm uses table look-ups to complete the same task as the recurrence method but much more efficiently. Pumpkins and targets are passed to this method as parameters. A number of variables are set, identical to the recursive call if you ignore the deathFlag and the x_Table which are utilized to build the traceback table. The optimalThrowTable is sized to fit all the data. The x_Table is also sized and referenced in the code but it is used for the traceback so it will not be further mentioned in this breakdown. A nested for-loop is initialized to run through all the successive pumpkins and targets with the first three conditionals handling the base cases per the recurrence relationship. The fourth conditional statement is handling every other case and uses another for-loop to cycle through all possible targets that is not a base case and uses table look-ups to accomplish the task versus recursion to find the optimal throw.

The minimum number of throws in the worst case for 3 pumpkins and 16 targets is 5.

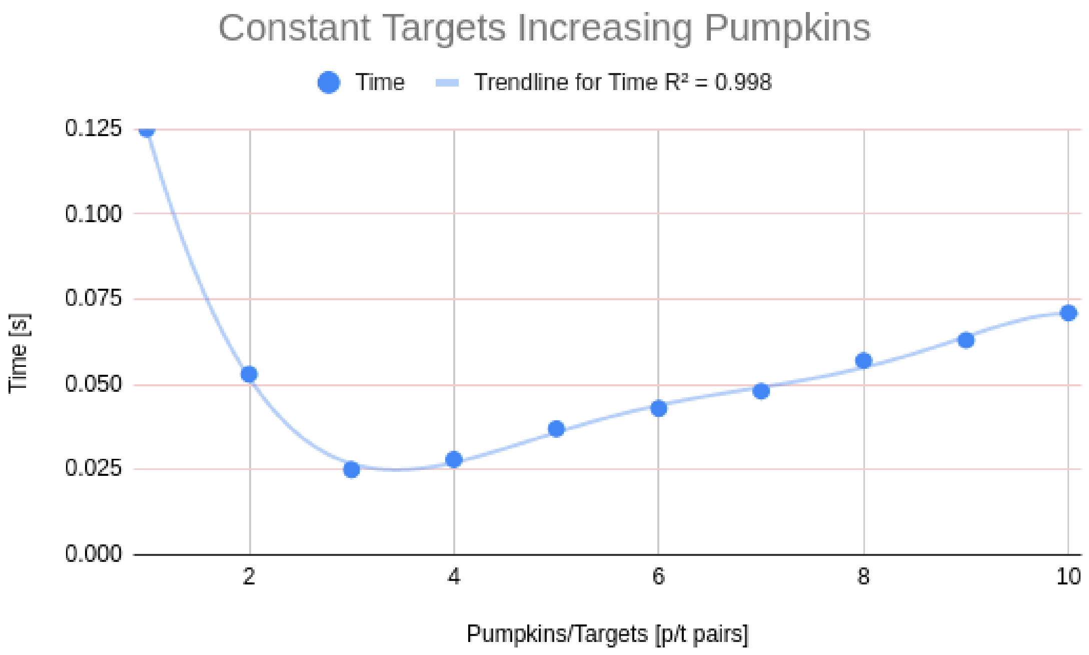
Dynamic Programming Complexity Analysis

The analysis here will be identical to the one I performed for the recursive implementation. The first analysis will hold pumpkins constant with targets increasing by 1000 on each successive run. The second will hold targets constant with pumpkins increasing by one each time. The final run will have both increase simultaneously by appropriate amounts for a thorough dynamic programming analysis.

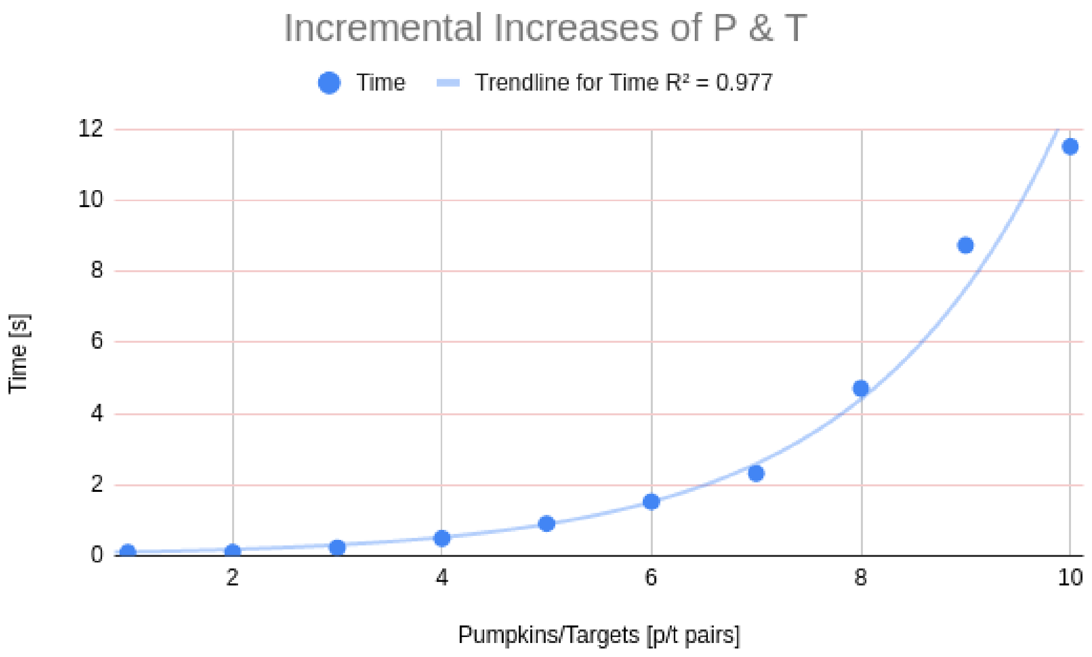
(Pumpkins,Targets)	Time(s)
(3,1000)	0.091
(3,2000)	0.076
(3,3000)	0.111
(3,4000)	0.195
(3,5000)	0.302
(3,6000)	0.433
(3,7000)	0.593
(3,8000)	0.771
(3,9000)	1.454
(3,10000)	1.236



(Pumpkins,Targets)	Time(s)
(3,1000)	0.091
(4,1000)	0.076
(5,1000)	0.111
(6,1000)	0.195
(7,1000)	0.302
(8,1000)	0.433
(9,1000)	0.593
(10,1000)	0.771
(11,1000)	1.454
(12,1000)	1.236



(Pumpkins,Targets)	Time(s)
(3,1000)	0.094
(4,2000)	0.098
(5,3000)	0.225
(6,4000)	0.487
(7,5000)	0.904
(8,6000)	1.522
(9,7000)	2.317
(10,8000)	4.707
(11,9000)	8.738
(12,10000)	11.512



From observation the dynamic programming approach gives polynomial time complexity functions. I am getting polynomial trendlines to match my scatter plots for these successive runs which would seem to indicate why we can increase targets and pumpkins with a much smaller effect to the overall run time versus the recursive algorithm where exponential held dominance. The dramatic increases in run time are due to the dynamic programming storing all the calculations and not having to constantly recalculate with each successive recursive call.

Traceback Implementation

```
        private ArrayList<Integer> traceIt(int p, int t){
        int offSet = 0;
        int survivalIndicator = 1;
        int optimalThrows = optimalThrowTable.get(p).get(t);
        for(int i = 0; i < optimalThrows; i++) {
            int val = x_Table.get(p).get(t);
            if (val > 0) { //She lives
                survivalIndicator = 1;
                t = t - Math.abs(val);

            } else { //She dies
                p -= 1;
                survivalIndicator = -1;
                t = Math.abs(val) -1;

            }
            tracer.add((Math.abs(val) + offSet) * survivalIndicator);
            if (val > 0) offSet += Math.abs(val);
        }
        return tracer;
    }
```

Now, let's return to the dynamic implementation where I had previous mentioned the x_Table. This table is tasked with tracking the throw location for any pumpkin/target combo and a positive number indicates survival and conversely a negative number indicates pumpkin death. A deathFlag which is a boolean takes care of the logic in assigning positive or negative values to the table at a given location based on what the maximum chosen is from the table.

Let's now turn to the above algorithm which is what returns the desired output which is the target order and whether they lived or died. This output is returned as an ArrayList and printed from main. The variables that are not self-explanatory are the offset which keeps track of the relative target based on if there was a survival at the target. The survivalIndicator flips the returned value to positive or negative accordingly to the rule described above. The optimalThrow is the answer from the dynamic solution which is stored in the optimalTable. A for-loop cycles from 0 to this number and val is assigned to the p/t combination which happens to be the same for the first iteration. Two conditional statements handle a survival or death and then this is added to the Arraylist with the appropriate offset and +/- sign and then the offset is added if the pumpkin survived.

Output Demo

$p = 5$ t=100

7

$-38 \quad 7 \quad -22 \quad -14 \quad -10 \quad 8 \quad -9$

Appendix

[1]*Project Code:*

```
import java.util.*;

public class Trebuchet {
    long calls = 0;
    ArrayList<ArrayList<Integer>> optimalThrowTable = new ArrayList<>();
    ArrayList<ArrayList<Integer>> x_Table = new ArrayList<>();
    ArrayList<Integer> tracer = new ArrayList<>();

    private int recursiveSoln(int p, int t) {
        calls++;
        int survive, smash, max, min;
        min = Integer.MAX_VALUE;

        if (t == 0) return 0;
        if (t == 1) return 1;
        if (p == 1) return t;

        for (int x = 1; x <= t; x++) {
            smash = recursiveSoln(p - 1, x - 1);
            survive = recursiveSoln(p, t - x);
            max = Math.max(smash, survive);
            if (max < min) min = max;
        }
        return 1 + min;
    }

    private void dpSoln(int p, int t){
        boolean deathFlag = false;
        int survive, smash, max, min;
        min = Integer.MAX_VALUE;
        resize(optimalThrowTable, p+1, t+1); //Size tables
        resize(x_Table, p+1, t+1);

        for (int i = 1; i <= p; i++) {
            for (int j = 0; j <= t; j++) {
                if (j == 0) optimalThrowTable.get(i).add(0);
                else if (j == 1) {
                    optimalThrowTable.get(i).set(j, 1);
                    x_Table.get(i).set(j, -1);
                } else if (i == 1) {
                    optimalThrowTable.get(i).set(j, j);
                    x_Table.get(i).set(j, 1);
                } else {
                    //Think of i as the pumpkins and j as the targets for this conditi
                    for (int x = 1; x <= j; x++) {
```

```
                    smash = optimalThrowTable.get(i - 1).get(x - 1);
                    survive = optimalThrowTable.get(i).get(j - x);

                    if (smash >= survive) {
                        max = smash;
                        deathFlag = true;
                    } else {
                        max = survive;
                    }
                    if (max < min) {
                        min = max;
                        if (deathFlag) {
                            x_Table.get(i).set(j, x * -1);
                            deathFlag = false;
                        } else x_Table.get(i).set(j, x);
                        }
                        optimalThrowTable.get(i).set(j, 1 + min);
                    }
                }
            min = Integer.MAX_VALUE;
            }
        }
}

public static void resize(ArrayList<ArrayList<Integer>> list, int arrays, int indi
    for(int i = 0; i < arrays; i++) {
        list.add(new ArrayList<Integer>());
        for(int j = 0; j < indices; j++) {
            list.get(i).add(0);
        }
    }
}

private ArrayList<Integer> traceIt(int p, int t){
    int offSet = 0;
    int survivalIndicator = 1;
    int optimalThrows = optimalThrowTable.get(p).get(t);
    for(int i = 0; i < optimalThrows; i++) {
        int val = x_Table.get(p).get(t);
        if (val > 0) { //She lives
            survivalIndicator = 1;
            t = t - Math.abs(val);

        } else { //She dies
            p -= 1;
            survivalIndicator = -1;
            t = Math.abs(val) -1;
        }
        tracer.add((Math.abs(val) + offSet) * survivalIndicator);
        if (val > 0) offSet += Math.abs(val);
    }
    return tracer;
}

public static void main(String[] args) {
    whatDoYouWantToRun(args);
}

private static void whatDoYouWantToRun(String[] args) {
    Trebuchet trebuchet = new Trebuchet();
    int pumpkins = Integer.parseInt(args[0]);
    int targets = Integer.parseInt(args[1]);
    Scanner input = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Pumpkins:\t" + pumpkins + "\ttargets:\t" + targets + "\nS
    String choice = input.nextLine();  // Read user input
    while (!choice.equalsIgnoreCase("r") && !choice.equalsIgnoreCase("d") && !cho
        System.out.println("Enter an appropriate response");
        choice = input.nextLine();
        if (choice.equalsIgnoreCase("r") || choice.equalsIgnoreCase("d") || choic
            break;
        }
    }
    if (choice.equalsIgnoreCase("r")) {
        recursive(trebuchet, pumpkins, targets);
    } else if (choice.equalsIgnoreCase("d")) {
        dynamic(trebuchet, pumpkins, targets);
    }
    else if (choice.equalsIgnoreCase("b")){
        recursive(trebuchet, pumpkins, targets);
        dynamic(trebuchet, pumpkins, targets);
    }
}

private static void recursive(Trebuchet trebuchet, int pumpkins, int targets) {
    System.out.println("*****************RECURSIVE*******************");
    System.out.println("******************PROGRAM********************");

    ArrayList<ArrayList<Integer>> recursiveArray = new ArrayList<>();
    long totalTime = 0;

    for (int p = 1; p <= pumpkins; p++) {
        recursiveArray.add(new ArrayList<>());
        for (int t = 0; t <= targets; t++) {
            trebuchet.calls = 0;
            long start = System.currentTimeMillis();
            recursiveArray.get(p - 1).add(trebuchet.recursiveSoln(p, t));
            long finish = System.currentTimeMillis();
            long timeElapsed = finish - start;
            totalTime += timeElapsed;
//                System.out.println("Pumpkins:\t" + p + "\tTargets:\t" + t + "\tAnswe
//                System.out.println("Time Elapsed:\t" + timeElapsed + "ms\t\t\tCalls
//                System.out.println("**********************************************
//                System.out.println("**********************************************
        }
    }
    System.out.println("Total Time Elapsed for Recursive:\t" + totalTime + "ms\n\r
    System.out.print("Targets: ->\t\t");
    for(int i = 0; i <= targets; i++){
        System.out.print( i + "\t");
    }
    System.out.println();
    for (int i = 0; i < recursiveArray.size(); i++) {
        System.out.print("Pumpkin: " + (1+i) + "\t\t");
        for (int j = 0; j < recursiveArray.get(i).size(); j++) {
            System.out.print(recursiveArray.get(i).get(j) + "\t");
        }
        System.out.println();
    }
    System.out.println("\n");
//        if (targets >= 21) return;  //testing
//        if (pumpkins >= 4) return;
//        for (int i = 4; i <= 12; i++){
//            System.out.println("**********************" + i + "********************
//            recursive(trebuchet, i, targets);
//        }
}

private static void dynamic(Trebuchet trebuchet, int pumpkins, int targets) {
    System.out.println("***************DYNAMIC******************");
    System.out.println("***************PROGRAM******************");
    long start = System.currentTimeMillis();
    long start2 = System.nanoTime();

    trebuchet.dpSoln(pumpkins, targets);
    int answer = trebuchet.optimalThrowTable.get(pumpkins).get(targets);
    long finish2 = System.nanoTime();
    long finish = System.currentTimeMillis();
    long timeElapsed = finish - start;
    long timeElapsed2 = finish2 - start2;
    System.out.println("Pumpkins: " + pumpkins + " Targets: " + targets);
    System.out.println("Total Time Elapsed for Dynamic:\t" + timeElapsed + "ms\t\t
    System.out.println("The throws necessary is: " + answer);
    System.out.print("Targets -> \t\t");
    for(int i = 0; i <= targets; i++){
        System.out.print( i + "\t");
    }
    System.out.println();
    for (int p = 1; p <= pumpkins; p++) {
        System.out.print("Pumpkin: " + p + "\t\t");
        for (int t = 0; t <= targets; t++) {
            System.out.print(trebuchet.optimalThrowTable.get(p).get(t) + "\t");
        }
        System.out.println();
    }
//        if(pumpkins >= 4) return;
//        for(int i = 0; i <= 8; i++) {
//              pumpkins++;
////            targets += 1000;
//            dynamic(trebuchet, pumpkins, targets);
//        }
    System.out.println("\n\n***************X*****************");
    System.out.println("************TABLE***************");
    System.out.print("Remaining T -> ");
    for(int i = 1; i <= targets; i++){
        System.out.print( "\t\t" + i);
    }
    System.out.println();
    for (int p = 1; p <= pumpkins; p++) {
        System.out.print("Pumpkin: " + p + "\t");
        for (int t = 1; t <= targets; t++) {
            System.out.print("\t\t" + trebuchet.x_Table.get(p).get(t));
        }
        System.out.println();
    }
    trebuchet.traceIt(pumpkins, targets);
    System.out.println("\n\n**********TRACER*****************");
    System.out.println("*************************************");
    for (var x : trebuchet.tracer){
        System.out.print(x + " ");
    }
}
}
```

[2] *p:3 t:16 Calls:*

```
*****************RECURSIVE********************
******************PROGRAM*********************
Pumpkins: 1 Targets: 0 Answer: 0
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 1 Answer: 1
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 2 Answer: 2
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 3 Answer: 3
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 4 Answer: 4
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 5 Answer: 5
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 6 Answer: 6
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 7 Answer: 7
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 8 Answer: 8
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 9 Answer: 9
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 10 Answer: 10
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 11 Answer: 11
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 12 Answer: 12
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 13 Answer: 13
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 14 Answer: 14
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 15 Answer: 15
Time Elapsed: 0ms Calls: 1
Pumpkins: 1 Targets: 16 Answer: 16
Time Elapsed: 0ms Calls: 1
Pumpkins: 2 Targets: 0 Answer: 0
Time Elapsed: 0ms Calls: 1
Pumpkins: 2 Targets: 1 Answer: 1
Time Elapsed: 0ms Calls: 1
Pumpkins: 2 Targets: 2 Answer: 2
Time Elapsed: 0ms Calls: 5
Pumpkins: 2 Targets: 3 Answer: 2
Time Elapsed: 0ms Calls: 11
Pumpkins: 2 Targets: 4 Answer: 3
Time Elapsed: 0ms Calls: 23
```

```
Pumpkins: 2 Targets: 5 Answer: 3
Time Elapsed: 0ms Calls: 47
Pumpkins: 2 Targets: 6 Answer: 3
Time Elapsed: 0ms Calls: 95
Pumpkins: 2 Targets: 7 Answer: 4
Time Elapsed: 0ms Calls: 191
Pumpkins: 2 Targets: 8 Answer: 4
Time Elapsed: 0ms Calls: 383
Pumpkins: 2 Targets: 9 Answer: 4
Time Elapsed: 0ms Calls: 767
Pumpkins: 2 Targets: 10 Answer: 4
Time Elapsed: 0ms Calls: 1535
Pumpkins: 2 Targets: 11 Answer: 5
Time Elapsed: 0ms Calls: 3071
Pumpkins: 2 Targets: 12 Answer: 5
Time Elapsed: 0ms Calls: 6143
Pumpkins: 2 Targets: 13 Answer: 5
Time Elapsed: 1ms Calls: 12287
Pumpkins: 2 Targets: 14 Answer: 5
Time Elapsed: 0ms Calls: 24575
Pumpkins: 2 Targets: 15 Answer: 5
Time Elapsed: 0ms Calls: 49151
Pumpkins: 2 Targets: 16 Answer: 6
Time Elapsed: 1ms Calls: 98303
Pumpkins: 3 Targets: 0 Answer: 0
Time Elapsed: 0ms Calls: 1
Pumpkins: 3 Targets: 1 Answer: 1
Time Elapsed: 0ms Calls: 1
Pumpkins: 3 Targets: 2 Answer: 2
Time Elapsed: 0ms Calls: 5
Pumpkins: 3 Targets: 3 Answer: 2
Time Elapsed: 0ms Calls: 15
Pumpkins: 3 Targets: 4 Answer: 3
Time Elapsed: 0ms Calls: 41
Pumpkins: 3 Targets: 5 Answer: 3
Time Elapsed: 0ms Calls: 105
Pumpkins: 3 Targets: 6 Answer: 3
Time Elapsed: 0ms Calls: 257
Pumpkins: 3 Targets: 7 Answer: 3
Time Elapsed: 0ms Calls: 609
Pumpkins: 3 Targets: 8 Answer: 4
Time Elapsed: 0ms Calls: 1409
Pumpkins: 3 Targets: 9 Answer: 4
Time Elapsed: 0ms Calls: 3201
Pumpkins: 3 Targets: 10 Answer: 4
Time Elapsed: 0ms Calls: 7169
Pumpkins: 3 Targets: 11 Answer: 4
Time Elapsed: 0ms Calls: 15873
```

```
Pumpkins: 3 Targets: 12 Answer: 4
Time Elapsed: 1ms Calls: 34817
Pumpkins: 3 Targets: 13 Answer: 4
Time Elapsed: 1ms Calls: 75777
Pumpkins: 3 Targets: 14 Answer: 4
Time Elapsed: 0ms Calls: 163841
Pumpkins: 3 Targets: 15 Answer: 5
Time Elapsed: 1ms Calls: 352257
Pumpkins: 3 Targets: 16 Answer: 5
Time Elapsed: 2ms Calls: 753665
Total Time Elapsed for Recursive: 7ms
```