Hwa Chong Institution

Group 8-21

Strike It Out

Written Report 2021

Leader: Lim Hengyi Gareth (15) 3i1

Aidan Ong Ming Feng (2) 3i1

Tan Yu Yao (26) 3i1

CONTENTS

Contents *Page*

1) Introduction

Strike It Out is a game for two players with a number line from 0 to 20, where players take turns to write an addition or subtraction equation with three numbers on the number line. The winner is the player that makes the last move. For example, Player A can choose the equation $10 - 1 = 9$ when he makes the first move. He strikes out the numbers 1 and 10, while circling the number 9, as illustrated in Fig.1.

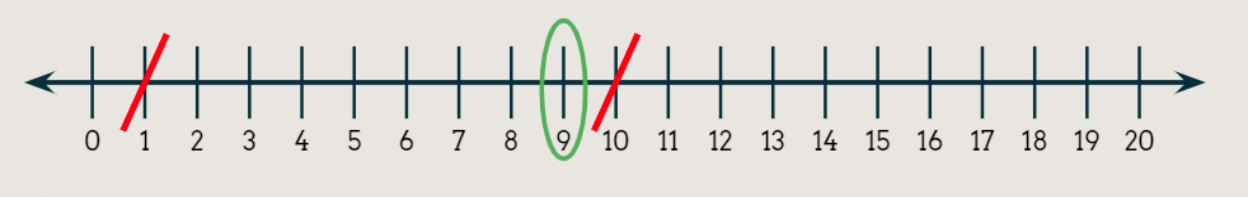

Fig. 1: Player A's first move

When Player A ends his turn with the number 9 circled, Player B must use 9 as the first number in his equation. For example, Player B can make the move $9 - 7 = 2$, striking 7 and 9 off, while circling 2, as illustrated in Fig.2. A player cannot use equations with numbers that have been struck off, and each number can only be used once in forming equations.

Fig. 2: Player B's first move

The players will take turns forming equations on the number line with the numbers remaining until a player is unable to form any equations with the remaining numbers, and the player who made the last move will be the winner.

2) Rationale

We found the game *Strike It Out* in a magazine called Mathematics in School in January this year. We decided to investigate this game because we were fascinated by the complexities behind this seemingly simple game and were interested to investigate further. We were also curious as to how extensions, like negative numbers, would affect the gameplay.

3) Objectives

We have several objectives we hope to achieve at the end of the research project. We hope:

1. To understand the mathematics behind *Strike It Out*
2. To explore *Strike It Out* and find out winning strategies behind it
3. To investigate how extensions and restrictions could affect the strategy of *Strike It Out*

4) Research Questions

To achieve our objectives, we have crafted three research questions to help us with the research. They are:

1. What are the winning strategies of *Strike It Out*?
2. What is the strategy for both players to strike out or circle all the numbers, if possible?
3. How does the addition of negative numbers change the strategy of *Strike It Out*?

5) Methodology

We used the following methods to conduct our research in an efficient and effective manner.

1. Conduct thorough research and investigation on the game
2. Derive strategies for the original version of the game
3. Derive strategies for the modified versions of the game
4. Verify our strategies mathematically or computationally

6) Fields of Mathematics

We identified two fields of mathematics that this project is related to.

1. Number Theory
2. Combinatorics

7) Terminology

1. Move: Each player makes one move per turn. A player's move refers to the equation he or she forms by selecting two numbers from the remaining numbers on the line, and then adding or subtracting those numbers. The numbers chosen are struck off, and the result of the equation is circled on the number line.
2. Impartial game: An impartial game is a game in which the available moves for a certain player at a certain point in the game depends only on the current state of the game and not on which of the two players is currently moving, and where the payoffs are symmetric, which means that the only difference between the two players is that Player 1 starts first. Another condition for a game to be an impartial game is that there is no way for there to be a draw, and that it is played with perfect information with no randomisation involved.
3. Number line states: A game state where a specific few numbers are crossed out and a number is circled, with an exception being the start of the game when no player has made a move and there are no numbers circled or crossed. Since two numbers are struck off and one new number is circled whenever a player makes a move, we can find out which player has to make a move next just from the number of numbers that have already been struck off.

8) Literature Review

8.1) *Strike It Out*

An article by Lynne McClure introduced *Strike It Out*, and clearly explained the rules. The idea of a co-operative game was also mentioned in the article, with the goal being to strike out or circle all numbers. This inspired our second research question, where we aim to find a way to achieve the aforementioned goal.

8.2) Zero-sum game

While conducting our research, an article on zero-sum games provided ideas on theorems that could aid us in finding a strategy. Since *Strike It Out* uses a finite number line, and each move will remove the option of using the numbers from that particular round of gameplay there will always be a point where one of the players will not be able to make a move, while the other will win. Hence, *Strike It Out* is a zero-sum game without the possibility of ties.

8.3) Zermelo's Theorem

Zermelo's Theorem states that in a finite two-player zero sum game with perfect information and without other factors like luck that could disrupt optimal gameplay, one of the two players must have a winning strategy. The proof of this theorem is by backwards induction. This provided ideas as to how a program could be created to simulate the gameplay of *Strike It Out*, and to derive its optimal strategy.

8.4) Sprague-Grundy Theorem

The Sprague-Grundy Theorem states that impartial games are equivalent to an infinite generalisation of Nim. Since the available moves of a player in *Strike It Out* is only dependent on the number line state, and there can be no draws with no randomisation, *Strike It Out* is an impartial game. This means that one player will always have a winning strategy since there is always a move that player can make in response to the other player to ensure their victory no matter what moves are made thereafter by the opposing player. This theorem is useful in helping us know about the basic concepts behind *Strike It Out*, like the guaranteed assumption that one player will always have a winning strategy.

8.5) Dynamic Programming

Dynamic Programming refers to an algorithm to solve a problem by breaking the problem into simpler subproblems, such that the optimal solution depends on the optimal solution to its subproblems. Most of the time, it is considered useful if the subproblems can be repeatedly broken down into its own subproblems using similar methods. As such, we decided to break down *Strike It Out* into multiple scenarios as the game progressed over multiple turns to find a winning strategy. This is illustrated in Fig.3.

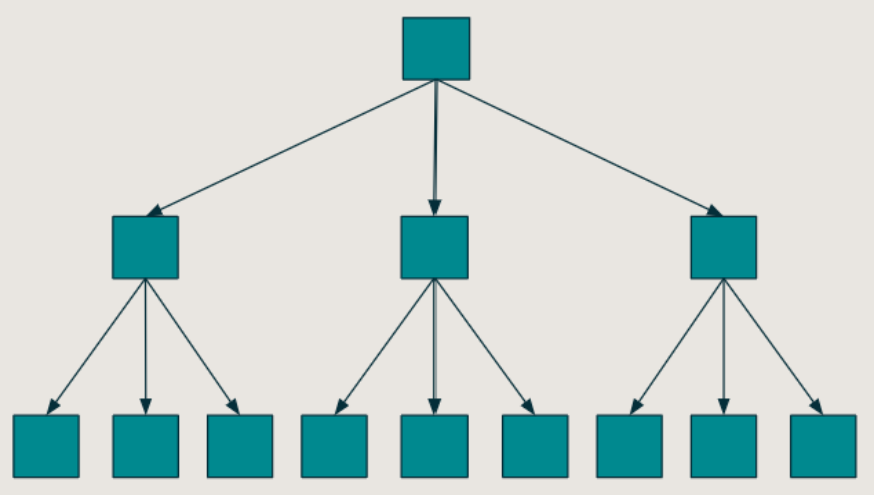

Fig. 3: Illustration of different possible game states branching out

We can represent the possible states of the game into a tree, which is a non-cyclic graph with n nodes and (n − 1) edges. We can let each node represent a certain number line state, while each edge represents a move taken by the player. To find out at a certain number line state which player has a definitive winning strategy even if both players play optimally, we can treat the finding of which player having a winner strategy as the "problem" while the "subproblems" be that of the many number line states the current node branches out to. Fig. 4 shows an example of different number line states, where the red arrow represents an optimal move that can be made by the current player on his turn.

As long as one of the subproblems show that the current player will win if they end up in that number line state, we can assume that the player will always pick that move, and we can answer the main problem that the current player will always win when the same number line state is reached. Only when all options that can be made by the current player result in them losing, can we say that the opposing player will win when

the current number line state is achieved. This is illustrated in Fig.4, with the red arrow representing one of the optimal moves that can be made by Player 1.

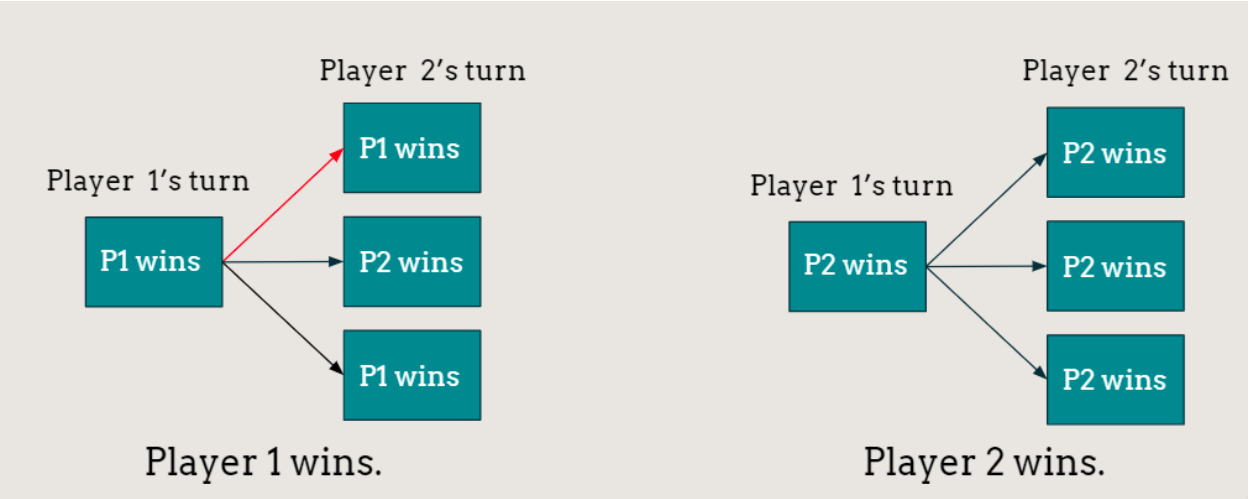

Fig. 4: Breaking down of number line states into subproblems

Through this, we can repeatedly split the possible number line states of the game into its own problem and subproblems, where the problem can be solved with the knowledge of the subproblems. Since the game will always come to an end and that there is a fixed number of number line states which result in the game ending, we can recursively work up the tree and come to a conclusion of which player will decisively win in future rounds of the game.

9) Results

9.1) Research Question 1: What are the winning strategies of *Strike It Out*?

We first attempted finding out which player has a winning strategy by making use of dynamic programming, before deciding to list a tree graph of all the game states. It is clear that there is always one winner as *Strike It Out* is a zero-sum game. We coded a program on C++ (refer to code in Appendix A and B) to do so. The results will be discussed below.

The results in the code given in Appendix A allows the user to freely set the desired starting number line state and finds out which player will have the winning strategy. The code returns the winning player and one winning game progression involving the starting number line state. It prints out the starting number and the end results of each equation, and also gives a visualisation of the progress of the game, as seen in Fig. 5.

```
Input a string made of 0s and 1s to represent to the number line with 0s representing
 uncrossed numbers and 1s representing crossed numbers: 0000000000000000000

Do you want the first player to start with an already circled number? Input 0 if no o
r the number if yes: 0

Player 1 Wins
1 8 20 15 2 6 17 14
10000000000000000000
10000011000000000000
10000011000100000001
10001011000100100001
11001011000110100001
11011111000110100001
11011111001110101001
```

Fig. 5: Screenshot of the code console output

It uses binary to store number line states, with 0 representing an unused number and 1 representing a number that has been struck off. Each digit in the string represents the numbers in the number line in ascending order, from 1 to 20, as 0 cannot be used in any addition or subtraction operation the players choose, giving 20 separate boolean variables. For the purpose of optimization of the code, circled numbers are assumed to be automatically struck off.

If there is no way for Player 1 to win, the code will always display "Player 2 Wins" and similarly outputs a string of numbers representing one winning game progression whereby the player who starts second wins. This can only be done if the user inputs a customised number line state, as the first player will always win given that both players play optimally.

Another program, in Appendix B, finds out all possible progressions of the game assuming both players play optimally which results in the first player winning. The program asks the user to input the length of the number line. The code then exhausts all possible number line states throughout the course of the game and prints out all the ways in which the first player will win. Fig. 6 shows the output of the code for a number line from 1 to 5, and Fig. 7 shows the entire tree graph of the game state progressions. In Fig. 7, red boxes represent number line states where the first player wins, and blue boxes represent number line states where the second player wins. From the graph, it can be seen that the first player has the winning strategy for a number line with numbers 1 to 5.

```
How long do you want the number line to be? 5
01 04
   03
02 03
03 04
   02
04 01
05 02
   04
```

Fig. 6: Screenshot of the code console output

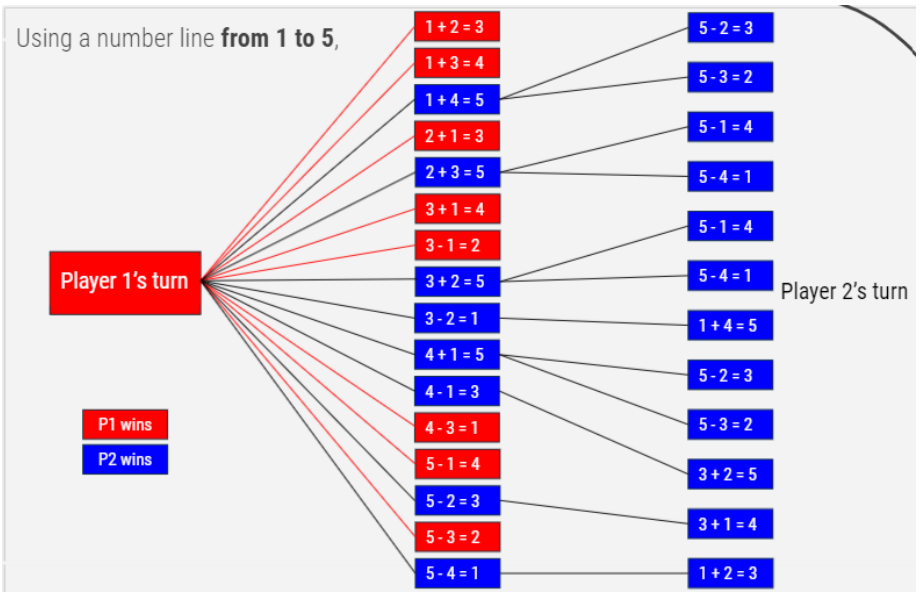

Fig. 7: Tree graph of all possible game progressions for a number line of 1 to 5

Using the program, we obtained the winning strategy for Player 1 in the original game with a number line of 1 to 20. Analysing the strategy, there are a total of 75397 possible winning game progressions. We then looked through the starting equations and compared the number of winning game progressions under that equation and the total number of winning lines for a specific starting number. We plotted the number of winning progressions for each starting number onto a graph, as seen in Fig. 8.

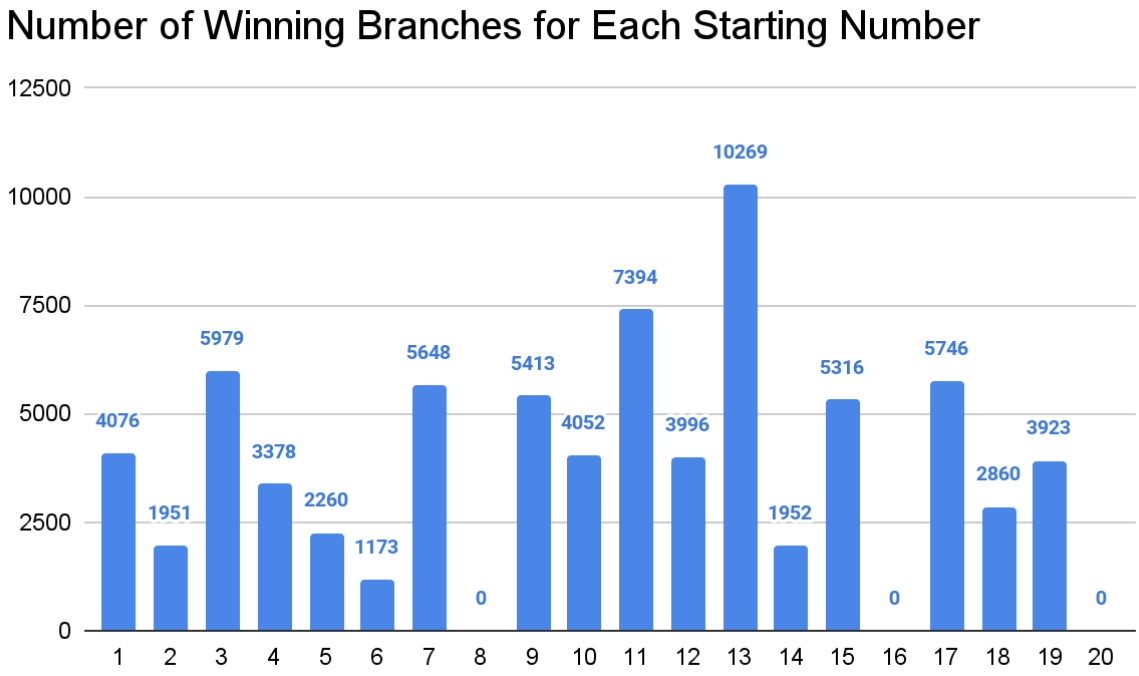

Fig. 8: Graph of the number of winning branches for each starting number on a number line from 1 to 20

From these results, we have observed a few trends in the total number of winning game progressions and their corresponding strategies. One such trend is that only starting numbers coprime to 20 tend to have a higher number of winning game progressions, with the exceptions of 10 and 15 having rather high numbers of winning game progressions. 19 is also such an exception with a lower number of winning lines but this can be attributed to being closer to the end point of the number line and having less options for move choices.

The full list of first equations used that is winning for the first player is stated in Appendix D.

9.2) Research Question 2: What is the strategy for both players to strike out or circle all the numbers, if possible?

Immediately, we note that 0 cannot be used in any of the equations, due to it being the additive identity. Because $x \pm 0 = x$, we will need to strike x out twice if we were to strike out 0. Therefore, there is no way to strike out 0.

To make the research question meaningful, the number '0' is not considered. Hence, we use a number line with the numbers from 1 to 20, which has 20 numbers. Note that each new equation strikes out 2 more numbers, and leaves 1 number circled. Therefore, at each new move, there is an odd quantity of numbers striked out or circled, and hence, there is no way to strike out or circle all numbers from 1 to 20.

However, there may exist ways for every number to be struck off or circled for number lines with an odd quantity of numbers. This led us to examine a number line with the numbers from 1 to 21. We wrote a code, in Appendix C, to brute force a strategy to strike out or circle all numbers from 1 to 21.

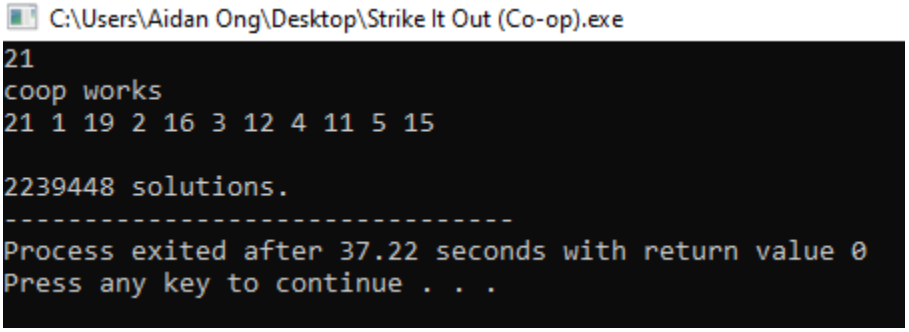

Fig. 9: Screenshot of the code console output

The code returned one of the possible lines whereby all numbers can be struck off and circled. In this line, the sequence of equations are:

Round number:	Player 1's move:	Player 2's move:
1.	21 – 20 = 1	1 + 18 = 19
2.	19 – 17 = 2	2 + 14 = 16
3.	16 – 13 = 3	3 + 9 = 12
4.	12 – 8 = 4	4 + 7 = 11
5.	11 – 6 = 5	5 + 10 = 15

This sequence of equations yields all numbers from 1 to 21 on the number line crossed or circled. This is 1 of 2239448 different strategies to strike off or circle all numbers on a number line from 1 to 21.

9.3) Research Question 3: How does the addition of negative numbers change the strategy of *Strike It Out*?

The addition of negative numbers suggests that unlike the previous two research questions, the number '0' can actually be used in equations. However, note that '0' can only be used in equations where 0 is the first number and the operation has to be a subtraction operation, like the equation $0 - 2 = -2$, or if 0 is the result of an equation adding two numbers of the same magnitude, like the equation $8 + (-8) = 0$.

In tackling this question, we wrote a program in C++, in Appendix E. We did not use a number line from -20 to 20, as the computation time would be too long. A number

line from –8 to 8 was used, with 0 inclusive. This program allows the user to specify the lowest and highest number in the number line and finds out all possible progressions of the game within that range assuming both players play optimally where the first player wins. However, it is different from the code in Appendix B because it now outputs the operation and the second number involved in each equation, since the addition of negative numbers means that there are multiple ways to end up with the same number when forming equations, like + 2 and $-(-2)$. Using the new program, we obtained the winning strategy for Player 1. Analysing the strategy, we found out that for a number line between -8 and 8, there are a total of 149196 possible winning game progressions. We then looked through the starting equations and compared the number of winning lines under that equation in Appendix F, and the total number of winning lines for a specific starting number. We plotted the number of winning game progressions for each starting number onto a graph, as seen in Fig. 10.

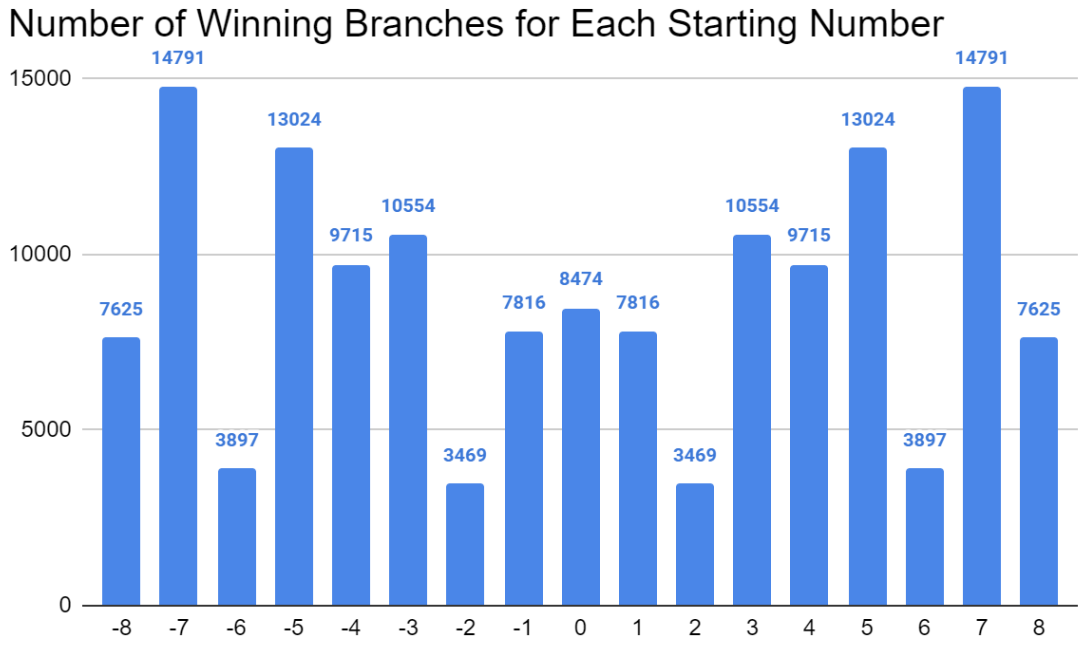

Fig. 10: Graph of the number of winning branches for each starting number on a number line from -8 to 8

We also used a number line from -10 to 10, inclusive of 0. Using the same program, we obtained the winning strategy for Player 1. Analysing the strategy, there were a total of 15520722 winning game progressions. Because of the large amount of possible winning game progressions, we found it impractical to look through all of the starting equations. We noted down the number of winning game progressions for each starting number and plotted the values onto a graph, as seen in Fig. 11.

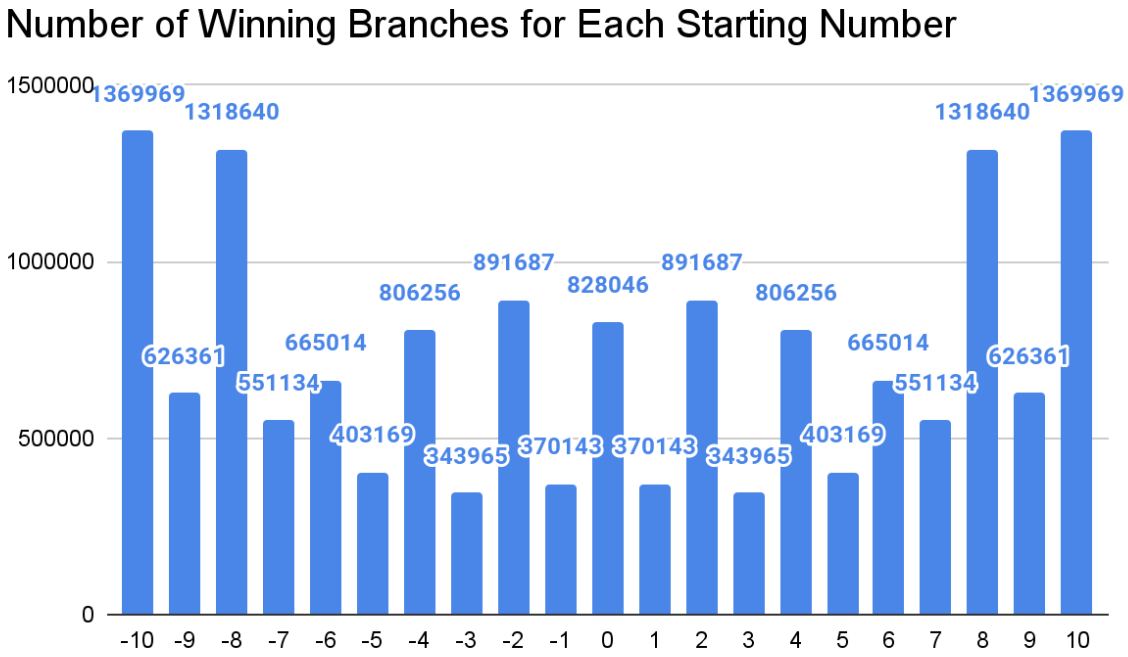

Fig. 11: Graph of the number of winning branches for each starting number on a number line from -10 to 10

From these results, we have observed a few trends in the total number of winning game progressions and their corresponding strategies. Similar to Research Question 1, we noticed that starting numbers coprime with 8 have a greater number of winning game progressions for the number line from -8 to 8. For the number line from -10 to 10, we noticed that all the even starting numbers had more winning progressions than the odd starting numbers.

In both number lines, it can also be observed that the number of winning lines for each starting number is symmetrical about '0'. For any winning line with the starting number x and proceeding numbers k_1, k_2, ..., k_n, a winning line for a game starting with

the number *-x* will have the proceeding numbers $-k_1$, $-k_2$, ..., $-k_n$. For example, one winning line starting with -8 in the number from -8 to 8 is:

$-8+5=-3$	$-3+7=4$	$4-8=-4$	$-4-(-5)=1$	$1+2=3$

so a corresponding winning line starting with 8 is:

$8+(-5)=3$	$3+(-7)=-4$	$-4-(-8)=4$	$4-5=-1$	$-1+(-2)=-3$

This property is attributed to the number lines being symmetrical around '0', and results in the graph of the number of winning lines for each starting number being symmetrical.

10) Future Extensions

We have several aspects of the research project that we hope to be able to explore more of.

1. Our solution, though accurate and working, is not practical for actual gameplay in that it is difficult to spot common trends when picking actual equations. Given more time, more trends can be observed so that a more combinatorial and comprehensive strategy can be obtained for Research Questions 1, 2, and 3.
2. Apart from our computational proofs for Research Questions 1, 2, and 3, we can also mathematically verify them for improved clarity.
3. *Strike It Out,* as a game, is fluid and highly flexible. We can certainly explore more variations of the game, like involving the use of multiplication and division operations.

11) References

"Grokking Dynamic Programming Patterns for Coding Interviews," (n.d.) *Subset Sum - Grokking Dynamic Programming Patterns for Coding Interviews*. https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3j64vRY6JnR.

John McQuarrie. *Mathematics and Chess* Retrieved March 31, 2021. https://mathshistory.st-andrews.ac.uk/Projects/MacQuarrie/chapter-4/

Kaeslin. (2009) *A Gentle Introduction to Dynamic Programming and the Viterbi Algorithm* Retrieved April 1, 2021. http://www.cambridge.org/resources/0521882672/7934_kaeslin_dynpro_new.pdf

Lynne McClure *Strike It Out* (2020) Retrieved March 14, 2021. https://www.cambridgemaths.org/blogs/striking-up-a-mathematical-conversation/

Schwalbe, Ulrich; Walker, Paul. (1997) *Zermelo and the Early History of Game Theory* Retrieved March 31, 2021. http://abel.math.harvard.edu/~elkies/FS23j.03/zermelo.pdf

Sjostrand, Jonas. (2013) *Impartial Games and Sprague-Grundy Theory* Retrieved May 2, 2021. https://www.math.kth.se/matstat/gru/sf2972/2013/lecture9_2013.pdf

12) Appendices

12.1) Appendix A

Finding who has the winning strategy:

```
#include<bits/stdc++.h>
#include<tuple>
using namespace std;

map<tuple<string, int, int>, pair<int, vector<int> > > memo;

pair<int, vector<int> > dp(string nlstate, int turn, int circled){
  tuple<string, int, int> tmp = make_tuple(nlstate, turn, circled);
  if (memo.find(tmp) != memo.end()) return memo[tmp];
 pair<int, vector<int> > result;
 int valid = 0;
 for(int i = 1; i <= nlstate.length(); i++){
    if(nlstate[i - 1] == '0'){
      nlstate[i - 1] = '1';
      if(circled - i > 0 && nlstate[circled - i - 1] == '0'){
        nlstate[circled - i - 1] = '1';
       valid = circled - i;
        result = dp(nlstate, turn + 1, circled - i);
        if(result.first == turn % 2){
         result.second.push_back(circled - i);
         return memo[tmp] = make_pair(turn % 2, result.second);
        }
        nlstate[circled - i - 1] = '0';
      }
      if(circled + i <= nlstate.length() && nlstate[circled + i - 1] == '0'){
        nlstate[circled + i - 1] = '1';
       valid = circled + i;
        result = dp(nlstate, turn + 1, circled + i);
        if(result.first == turn % 2){
         result.second.push_back(circled + i);
         return memo[tmp] = make_pair(turn % 2, result.second);
        }
        nlstate[circled + i - 1] = '0';
      }
            nlstate[i - 1] = '0';
    }
  }
 if(valid != 0) result.second.push_back(valid);
  return memo[tmp] = make_pair((turn  + 1) % 2, result.second);
}
```

```
int main(){
 cout << "Input a string made of 0s and 1s to represent to the number line with 0s representing uncrossed numbers and 1s representing crossed numbers: ";
  string nl, b;
 cin >> nl;
 cout << "\nDo you want the first player to start with an already circled number? Input 0 if no or the number if yes: ";
 int s;
 cin >> s;
 cout << '\n';
 b = nl;
 queue<int> q;
 pair<int, vector<int> > winner;
 if(s > 0 && s <= nl.length()){
   winner = dp(nl, 1, s);
   if(winner.first == 1){
          cout << "Player 1 Wins" << '\n';
     cout << s << " ";
     q.push(s);
     while(!winner.second.empty()){
       cout << winner.second.back() << " ";
       q.push(winner.second.back());
       winner.second.pop_back();
     }

     int c = 0;
     while(!q.empty()){
       if(c != 0){
         b[abs(q.front() - c) - 1] = '1';
         b[q.front() - 1] = '1';
         cout << '\n' << b;
         c = q.front();
       }else{
         b[q.front() - 1] = '1';
         cout << '\n' << b;
         c = q.front();
       }
       q.pop();
     }
          return 0;
   }
 }else{
   for(int i = 1; i <= nl.length(); i++){
     if(nl[i - 1] == '0'){
           nl[i - 1] = '1';
       winner = dp(nl, 1, i);
```

```
            if(winner.first == 1){
              cout << "Player 1 Wins" << '\n';
        cout << i << " ";
        q.push(i);
        while(!winner.second.empty()){
          cout << winner.second.back() << " ";
          q.push(winner.second.back());
          winner.second.pop_back();
        }
        int c = 0;
        while(!q.empty()){
          if(c != 0){
            b[abs(q.front() - c) - 1] = '1';
            b[q.front() - 1] = '1';
            cout << '\n' << b;
            c = q.front();
          }else{
            b[q.front() - 1] = '1';
            cout << '\n' << b;
            c = q.front();
          }
          q.pop();
        }
              return 0;
            }
      nl[i - 1] = '0';
    }
   }
}
 cout << "Player 2 Wins" << '\n';
if(s > 0 && s <= nl.length()){
  cout << s << " ";
  q.push(s);
}else{
  cout << nl.length() << " ";
  q.push(nl.length());
}
while(!winner.second.empty()){
  cout << winner.second.back() << " ";
  q.push(winner.second.back());
  winner.second.pop_back();
}

int c = 0;
while(!q.empty()){
  if(c != 0){
```

```
    b[abs(q.front() - c) - 1] = '1';
    b[q.front() - 1] = '1';
    cout << '\n' << b;
    c = q.front();
   }else{
    b[q.front() - 1] = '1';
    cout << '\n' << b;
    c = q.front();
   }
   q.pop();
 }
}
```

12.2) Appendix B

Exhausting all winning states:

```
#include<bits/stdc++.h>
#include<tuple>
using namespace std;
bool ps = true;

map<tuple<string, int, int>, pair<int, vector<int> > > memo;

pair<int, vector<int> > dp(string nlstate, int turn, int circled){
 tuple<string, int, int> tmp = make_tuple(nlstate, turn, circled);
  if (memo.find(tmp) != memo.end()) return memo[tmp];
 pair<int, vector<int> > result;
 vector<int> valid;
 for(int i = 1; i <= nlstate.length(); i++){
    if(nlstate[i - 1] == '0'){
      nlstate[i - 1] = '1';
      if(circled - i > 0 && nlstate[circled - i - 1] == '0'){
        nlstate[circled - i - 1] = '1';
       valid.push_back(circled - i);
        result.first = dp(nlstate, turn + 1, circled - i).first;
        if(result.first == turn % 2){
         result.second.push_back(circled - i);
       }
        nlstate[circled - i - 1] = '0';
      }
      if(circled + i <= nlstate.length() && nlstate[circled + i - 1] == '0'){
        nlstate[circled + i - 1] = '1';
       valid.push_back(circled + i);
        result.first = dp(nlstate, turn + 1, circled + i).first;
        if(result.first == turn % 2){
         result.second.push_back(circled + i);
```

```
      }
       nlstate[circled + i - 1] = '0';
      }
           nlstate[i - 1] = '0';
    }
  }
 if(result.second.empty()){
   result.second = valid;
    return memo[tmp] = make_pair((turn  + 1) % 2, result.second);
 }else{
    return memo[tmp] = make_pair(turn % 2, result.second);
 }
}

void display(string nls, int tur, int cir){
 if(ps){
   for(int i = 0; i < tur - 1; i++) cout << "    ";
   ps = false;
 }
 tuple<string, int, int> dis = make_tuple(nls, tur, cir);
 pair<int, vector<int> > branch;
 branch = memo[dis];
 if(branch.first == 1){
   if(branch.second.empty()){
     if(cir < 10){
       string z = "0";
       z += ('0' + cir);
       cout << z << '\n';
     }else cout << cir << '\n';
     ps = true;
   }
   else{
     if(cir < 10){
       string z = "0";
       z += ('0' + cir);
       cout << z << " ";
     }else cout << cir << " ";
     while(!branch.second.empty()){
       nls[branch.second.back() - 1] = '1';
       nls[abs(branch.second.back() - cir) - 1] = '1';
       display(nls,tur + 1, branch.second.back());
       nls[branch.second.back() - 1] = '0';
       nls[abs(branch.second.back() - cir) - 1] = '0';
       branch.second.pop_back();
     }
   }
```

```
 }
}


int main(){
 cout << "How long do you want the number line to be? ";
 int nll;
 ofstream cout("2021PWMovelist.txt");
 cin >> nll;
 string nl;
 char tnl[nll];
 pair<int, vector<int> > winner;
 for(int i = 0; i < nll; i++) tnl[i] = '0';
 nl = tnl;
 for(int i = 0; i < nll; i++){
   nl[i] = '1';
   winner = dp(nl, 1, i + 1);
   display(nl, 1, i + 1);
   nl[i] = '0';
 }
}
```

12.3) Appendix C

```
#include <bits/stdc++.h>
using namespace std;

int n, a[50]={};
int sola[50]={}, ta[50]={}, kkk, x=0;

void print(int o, int c){
        for (int i=0; i<c; i++) cout<<"\t";
        for (int i=1; i<=n; i++){
                if(a[i]==0) cout<<i;
                else cout<<"/";
                if(i==o) cout<<" <-";
                cout<<'\t';
        }
        cout<<'\n';
        return;
}

bool check(int m){
```

```
	for (int i=1; i<=n; i++) if(a[i]==0 && i!=m) return false;
	x++;
	return true;
}

void recur(int i, int c){
	ta[c+1]=i;
	if(check(i)){
		kkk=c+1;
		for (int i=1; i<=c+1; i++) sola[i]=ta[i];
	}
	for (int j=1; j<=n; j++){
		if (a[j]==1) continue;
		if (j!=i && i+j<=n && a[i+j]==0){
			a[i]=a[j]=1;
			//print(i+j,c+1);
			recur(i+j,c+1);
			a[i]=a[j]=0;
		}
		if (j!=i && i-j>=1 && i-j!=j && a[i-j]==0){
			a[i]=a[j]=1;
			//print(i-j,c+1);
			recur(i-j,c+1);
			a[i]=a[j]=0;
		}
	}
	return;
}

int main(){
	cin>>n;
	if(n>=50) return 0;
	//print(0,0);
	for (int i=1; i<=n; i++){
		//cout<<'\n';
		recur(i,0);
	}
	if(x>0){
		cout<<"coop works\n";
		for (int i=1; i<=kkk; i++) cout<<sola[i]<<" ";
	}
	else cout<<"coop does not work";
	cout<<"\n\n"<<x<<" solutions.";

}
```

12.4) Appendix D

Exhaustive list of winning first equations:

First equation used	Number of winning lines
1 + 13 = 14	2251
1 + 7 = 8	1825
2 + 12 = 14	1951
3 + 15 = 18	2812
3 + 7 =10	1806
3 + 4 = 7	1361
4 + 7 = 11	2017
4 + 3 = 7	1361
5 – 4 = 1	2260
6 – 5 = 1	1173
7 + 4 = 11	2017
7 + 3 = 10	1806
7 + 1 = 8	1825
9 + 11 = 20	3720
9 – 1 = 8	1693
10 – 9 = 1	1963
10 – 7 = 3	2089
11 + 9 = 20	3720
11 – 3 = 8	1900
11 – 1 = 10	1774
12 + 2 = 14	1951
12 – 2 = 10	2045

13 – 11 = 2	3650
13 – 5 = 8	2247
13 – 4 = 9	2121
13 + 1 = 14	2251
14 – 6 = 8	1952
15 – 9 = 6	2504
15 + 3 = 18	2812
17 – 13 = 4	3435
17 – 1 = 16	2312
18 – 14 = 4	2860
19 – 17 = 2	3923

12.5) Appendix E

```
#include<bits/stdc++.h>
#include<tuple>
using namespace std;
int mini, maxi, nll;
bool el = false, newsta = false;
map<tuple<string, int, int>, pair<int, vector<string> > > memo;

string disfor(int i){
 string ret;
 if(i > -1){
   ret = "0";
   ret += ('0' + floor(i / 10));
   ret += ('0' + (i % 10));
 }else{
   ret = "-";
   ret += ('0' + abs(floor(i / 10)));
   ret += ('0' + abs(i % 10));
 }
 return ret;
}

pair<int, vector<string> > dp(string nlstate, int turn, int circled){
 tuple<string, int, int> tmp = make_tuple(nlstate, turn, circled);
```

```
 if (memo.find(tmp) != memo.end()) return memo[tmp];
pair<int, vector<string> > result;
vector<string> valid;
for(int i = mini; i <= maxi; i++){
  if(nlstate[i - mini] == '0'){
     nlstate[i - mini] = '1';
    string op;
     if(circled - i >= mini && circled - i <= maxi && nlstate[circled - i - mini] == '0'){
       nlstate[circled - i - mini] = '1';
      op = "-" + disfor(i);
      valid.push_back(op);
       result.first = dp(nlstate, turn + 1, circled - i).first;
       if(result.first == turn % 2){
        result.second.push_back(op);
      }
       nlstate[circled - i - mini] = '0';
     }
     if(circled + i <= maxi && circled + i >= mini && nlstate[circled + i - mini] == '0'){
       nlstate[circled + i - mini] = '1';
      op = "+" + disfor(i);
      valid.push_back(op);
       result.first = dp(nlstate, turn + 1, circled + i).first;
       if(result.first == turn % 2){
        result.second.push_back(op);
      }
       nlstate[circled + i - mini] = '0';
     }
           nlstate[i - mini] = '0';
   }
}
if(result.second.empty()){
  result.second = valid;
   return memo[tmp] = make_pair((turn  + 1) % 2, result.second);
}else{
   return memo[tmp] = make_pair(turn % 2, result.second);
}
}

void display(string nls, int tur, int cir){
tuple<string, int, int> dis = make_tuple(nls, tur, cir);
pair<int, vector<string> > branch;
branch = memo[dis];
if(branch.first == 1){
  if(newsta){
    cout << disfor(cir) << "  ";
    newsta = false;
```

```
  }
  if(branch.second.empty()){
    cout << '\n';
    el = true;
  }else while(!branch.second.empty()){
    if(el){
      for(int i = 0; i < tur; i++) cout << "      ";
      el = false;
    }
    cout << branch.second.back() << " ";
    int ab = 0;
    ab += ((branch.second.back()[2] - '0') * 10);
    ab += (branch.second.back()[3] - '0');
    if(branch.second.back()[0] == '-'){
      if(branch.second.back()[1] == '-'){
        nls[0 - ab - mini] = '1';
        nls[cir + ab - mini] = '1';
        display(nls,tur + 1, cir + ab);
        nls[0 - ab - mini] = '0';
        nls[cir + ab - mini] = '0';
      }else{
        nls[ab - mini] = '1';
        nls[cir - ab - mini] = '1';
        display(nls,tur + 1, cir - ab);
        nls[ab - mini] = '0';
        nls[cir - ab - mini] = '0';
      }
    }else{
      if(branch.second.back()[1] == '-'){
        nls[0 - ab - mini] = '1';
        nls[cir - ab - mini] = '1';
        display(nls,tur + 1, cir - ab);
        nls[0 - ab - mini] = '0';
        nls[cir - ab - mini] = '0';
      }else{
        nls[ab - mini] = '1';
        nls[cir + ab - mini] = '1';
        display(nls,tur + 1, cir + ab);
        nls[ab - mini] = '0';
        nls[cir + ab - mini] = '0';
      }

    }
    branch.second.pop_back();
  }
}
```

```
}

int main() {
 cout << "What is the lowest number of the number line? ";
 cin >> mini;
 cout << "What is the highest number of the number line? ";
 cin >> maxi;
 string nl;
 nll = maxi - mini + 1;
 char tnl[nll];
 pair<int, vector<string> > winner;
 for(int i = 0; i < nll; i++) tnl[i] = '0';
 nl = tnl;
 for(int i = 0; i < nll; i++){
   nl[i] = '1';
   winner = dp(nl, 1, i + mini);
   el = false;
   newsta = true;
   display(nl, 1, i + mini);
   nl[i] = '0';
 }
}
```

12.6) Appendix F

Exhaustive list of winning first equations (for RQ3):

First equation used	Number of winning lines
-8 + 5 = -3	1933
-8 - (-2) = -6	2771
-8 - (-6) = -2	2921
-7 + 7 =0	4592
-7 + 4 = -3	1918
-7 + 1 = -6	1492
-7 - 1 = -8	1414
-7 + (-1) = -8	1351
-7 - (-3) = -4	2009
-7 - (-4) = -3	2015
-6 + (-2) = -8	1879

-6 - (-2) = -4	2018
-5 + 8 = 3	1933
-5 + 5 = 0	2904
-5 + 3 = -2	2026
-5 - 3 = -8	1508
-5 + (-3) = -8	1269
-5 - (-3) = -2	1638
-5 - (-7) = 2	1746
-4 + 7 = 3	1918
-4 - 2 = -6	1915
-4 + 1 = -3	1287
-4 + (-2) = -6	1590
-4 + (-3) = -7	975
-3 + 5 = 2	2030
-3 - 5 = -8	1471
-3 + 3 = 0	1568
-3 + (-4) = -7	975
-3 + (-5) = -8	1269
-3 - (-5) = 2	1513
-3 - (-7) = 4	1728
-2 + (-4) = -6	1590
-2 + (-6) = -8	1879
-1 + 7 = 6	1492
-1 - 7 = - 8	1018
-1 + 4 = 3	1287
-1 - (-4) = 3	1210
-1 + (-7) = -8	1351
-1 - (-7) = 6	1458
0 - 6 = -6	4237
0 - (-6) = 6	4237

1 + 7 = 8	1351
1 - 7 = -6	1458
1 - 4 = -3	1210
1 + (-4) =-3	1287
1 - 7 = 6	1018
1 + 7 = 8	1492
2 + 6 = 8	1879
2 + 4 = 6	1590
3 - 7 = -4	1728
3 - 5 = -2	1513
3 + 5 = 8	1269
3 + 4 = 7	975
3 + (-3) = 0	1568
3 - (-5) = 8	1471
3 + (-5) = 2	2030
4 + 3 = 7	975
4 + 2 = 6	1590
4 + (-1) = 3	1287
4 - (-2) = 6	1915
4 + (-7) = 3	1918
5 - 7 = -2	1746
5 - 3 = 2	1638
5 + 3 = 8	1269
5 - (-3) = 8	1508
5 + (-3) = 2	2026
5 + (-5) = 0	2904
5 + (-8) = 3	1933
6 - 2 = 4	2018
6 + 2 = 8	1879
7 - 4 = 3	2015

7 - 3 = 4	2009
7 + 1 = 8	1351
7 - (-1) = 8	1414
7 + (-1) = 6	1492
7 - 4 = 3	1918
7 + (-7) = 0	4592
8 - 6 = 2	2921
8 - 2 = 6	2771
8 + (-5) = 3	1933