

# CMSC 441 Algorithms Project - Games People Play

An introduction to Winning Positions and Dynamic Programming

By

Adam Fort

Student

University of Maryland Baltimore County

[afort1@umbc.edu](mailto:afort1@umbc.edu)

<http://userpages.umbc.edu/~afort1/>

This paper is available online with full downloadable source code at

<http://userpages.umbc.edu/~afort1/cs441/>

## Keywords:

asymmetric, no – repeat, pumping lemma, stones, winning conditions

## Abstract:

An experiential analysis of a class of two-player games where the outcome of the game depends solely on the input conditions into the game. There are three cases that have been analyzed. The warm-up game where there are no restrictions on the movement of the player which has a run time of  $O(nm)$ . The more interesting game is where the players are not allowed to repeat the last move which has a runtime of  $O(nm)$ . And finally, the asymmetric game where each player has a different move set and has a runtime of  $O(n*m*s)$ . In each case there is a mathematical recurrence that can reproduce the output of the game, and in many cases dynamic programming examples are given.

## Table of Contents

<b>CMSC 441 ALGORITHMS PROJECT - GAMES PEOPLE PLAY</b>	<b>1</b>
Keywords:	1
Abstract:	1
<b>TABLE OF CONTENTS</b>	<b>2</b>
<b>INTRODUCTION</b>	<b>3</b>
<b>THE WARM-UP GAME</b>	<b>3</b>
<b>A MORE INTERESTING GAME</b>	<b>4</b>
<b>ASYMMETRIC GAMES</b>	<b>8</b>
<b>REFERENCES</b>	<b>12</b>
<b>APPENDIX A – SOURCE CODE</b>	<b>13</b>
The Simple Game	13
Coderunner	15
Makefile	16
Project.h	17
Timer.h	22
Timer.cpp	23

## Introduction

In the following game there are two players that have a pile of stones between them. The rules are simple, both players agree to a set or sets of legal moves and the last person who can remove a stone from the table will be declared the winner. Because of the way the game is designed there is no chance in the game and if both player play logically.

## The Warm-up Game

In the warm-up game, the first-player will win provided that there is a possible move for the first-player that will move him into a second player win. For the second-player to be able to win they would have to move into a position where every possible position is a first player win. This movement is governed by the recursion  $f(n)$  which corresponds to a set  $S = (S_1, \dots, S_t)$  of possible moves.

$$f(n) = \left\{ \begin{array}{l} 2 \quad \text{if } n = 0 \\ 1 \quad \text{if } \exists p \in S \text{ s.t. } f(n - p) = 2 \\ 2 \quad \text{all other cases} \end{array} \right\}$$

The algorithm works by looping from 0 to  $n$  and at each step it calls the recurrence  $f(n)$  witch results in a very slow run time. This function is a dynamic programming algorithm that will result in one of five cases and has a runtime of  $O(nm)$

- If  $n$  is 0 the function will return a second player win and return. –  $O(1)$
- If  $n$  is in  $S$ , the set of legal moves, the function will return a first player win and return. –  $O(m)$
- For every element in  $S$ , each element is subtracted from  $n$  and the result is checked in memory. If a match is found that results to 2, the function will return a 1. –  $O(1)$

- If no match is found in memory the function then calls  $f(n - e)$  where  $e$  is each element that is in  $S$ . If any of these returns a second player win, the function will return a 1. – Let  $m=|S|$ .  $O(n*m)$
- If everything else fails then the function returns a second player win. –  $O(1)$

A complete working example of this algorithm coded using this dynamic programming method is provided in Appendix A. Using this algorithm a list of all of the winning positions for when  $n \leq 100$  and  $S = (1, 5, 8, 10)$  is listed below:

Player 1 Winning Conditions:

1 3 5 7 8 9 10 11 12 14 16 18 20 21 22 23 24 25 27  
 29 31 33 34 35 36 37 38 40 42 44 46 47 48 49 50 51 53  
 55 57 59 60 61 62 63 64 66 68 70 72 73 74 75 76 77 79  
 81 83 85 86 87 88 89 90 92 94 96 98 99 100

Player 2 Winning Conditions:

0 2 4 6 13 15 17 19 26 28 30 32 39 41 43 45 52 54 56  
 58 65 67 69 71 78 80 82 84 91 93 95 97

## A More Interesting Game

The game rules have been changed that now forbid the repetition of the previous players move. So the recursion now has to keep track of two things, one the recursion, and two the previous move. For example if the Left player removes 5 stones, then the right player can make any legal move except for 5. In order to force a win for the Left player tries to move into a safe position, which is a position on the where no matter what the previous move there will always be a winning strategy for the second player. In the example that was given, where  $S = (1, 2, 3, 4, 5)$  the first two safe positions are where  $n = 7$  and  $n = 13$ . The positions follow a pattern where the spacing between the numbers alternates between +7 and +6.

This is a representation of the internal structure of the algorithm, the first few rows have been filled in due to following the rules of the algorithm, if to win you need  $S_p$  and you are  $S_p$  away then it is impossible for you to win at that position.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S1	2	2	1	1	1	1															
S2	2	1	2	1	1	1															
S3	2	1	1	2	1	1															
S4	2	1	1	1	2	1															
S5	2	1	1	1	1	2															

At each step we look  $S_p$  backwards to see if a 2 is in the position there; if it is then due to the history rule we cannot move into that position so assign it as 2. Every other position can force a win in 1 move.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S1	2	2	1	1	1	1	1														
S2	2	1	2	1	1	1	1														
S3	2	1	1	2	1	1	2														
S4	2	1	1	1	2	1	1														
S5	2	1	1	1	1	2	1														

When  $n = 7$  every element in the diagonal is equal to one, therefore that is a safe position.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S1	2	2	1	1	1	1	1	2													
S2	2	1	2	1	1	1	1	2													
S3	2	1	1	2	1	1	2	2													
S4	2	1	1	1	2	1	1	2													
S5	2	1	1	1	1	2	1	2													

At  $n = 13$  the full series is visible

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S1	2	2	1	1	1	1	1	2	1	1	1	1	1	2							
S2	2	1	2	1	1	1	1	2	1	1	1	1	1	2							
S3	2	1	1	2	1	1	2	2	1	1	1	1	1	2							
S4	2	1	1	1	2	1	1	2	1	1	1	2	1	2							
S5	2	1	1	1	1	2	1	2	1	1	1	1	2	2							

At  $n=20$ , the next safe value in the series you can see the pattern starting to repeat.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S1	2	2	1	1	1	1	1	2	1	1	1	1	1	2	2	1	1	1	1	1	2
S2	2	1	2	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1	1	1	2
S3	2	1	1	2	1	1	2	2	1	1	1	1	1	2	1	1	2	1	1	1	2
S4	2	1	1	1	2	1	1	2	1	1	1	2	1	2	1	1	1	1	1	1	2
S5	2	1	1	1	1	2	1	2	1	1	1	1	2	2	1	1	1	1	2	1	2

Using this method the positions for up to  $n = 100$  have been filled out below:

Next Safe Positions for  $n=100$ :

0 7 13 20 26 33 39 46 52 59 65 72 78 85 91 98

The safe positions are those which have values that when divided by 13 leave a remainder that is either 0 or 7. To prove this we make use of the *Pumping Lemma* which is a tool that is gained from automata theory, the theory is used to show that given an infinite language which there is a finite set of states ( $m$ , the number of items in the move set  $S$ ) and that some of the strings have a length that is greater than  $m$ , then there must be a cycle in graph. The proof of this is as follows:

The pumping lemma states that anything that is in an infinite language but has a finite grammar will eventually have strings that repeat. Based off of the first 13 results of  $f(n)$  that only have 7 and 13 as safe values will eventually be forced to repeat itself. Because the pattern will repeat every 13 values the safe positions will be those that occur when  $n \% 13 = 0$  or  $n \% 13 = 7$

To compute the safe positions using recursion only would have an extremely long run time. If the problem is rewritten using dynamic programming methods, like

memoization will speed up the algorithm greatly. A dynamic programming program algorithm that will return if a given  $n$  is safe when given an arbitrary set  $S$  is as follows.

In this algorithm, the history array that is used for memoization is an  $n*m$  sized matrix.

The only values that will be returned are 1's and 2's that represent the player positions; to flip the value means to set 1->2 and 2->1.

- The history array for this function is an  $n*m$  sized matrix.
- If  $n = 0$ , loop through and set all the values for the history array at  $n$  equal to 2
- For each value in  $S$  place in  $value$ 
  - If  $value == previous\_move$ , then return 2, otherwise
  - If  $n - value \geq 0$ , then check to see if there is already a position memoized for  $history[n-value][value]$ , if there is return the flip of the value.
  - since there is no memo saved for this position recurse then recurse calling this function and passing into it  $n = n - value$  and  $previous\_move = value$ .  
If the recurrence returns a 2 then return a 1.
- If the function passes all of the previous conditions and still has not returned then return 2.

This function has a time complexity of  $O(n*m)$ . Now suppose the function depends on the previous  $k$  moves. To modify the preceding function to find safe positions biased off of the previous  $k$  moves we need to keep track of  $k$  tables, when we chose a position we recurse into the table for the previous move. The time complexity for this version of the algorithm is only  $O(k*n*m)$ .

## Asymmetric Games

The game is now modified to allow for each player to have a different set of legal moves. Because each player has different moves it no longer makes sense to find the “first-player-winning positions”. Traditionally the players are named Left and Right and are the winning positions L-winning positions and R-winning positions.

The refined dynamic programming algorithm that follows these conditions is as follows:

Let  $L\_condition$ ,  $R\_condition$  be an array of the respective winning conditions for each player and let  $S\_condition$  be the array of safe conditions, where both players are safe.

Let History be the memorization data set to store the results, history is a two dimensional array where vertically is set to the number of players and horizontally it is set to  $n$ .

recurrence(int n)

- Check if the left player can move, if not Right wins, save the result into the history.
- Check if the right player can move, if not Left wins, save the result into the history.
- Check to see if the left player is in a winning position at the start, if they are save the result in to the history.
- Check to see if the right player is in a winning position at the start, if they are save the result in to the history.
- Check to see if the history has already been written for this value of n, if so then just return.
- For each of the allowed left player moves that are stored into value
  - Check to see if moving by value will cause n to go negative, if so R will win at that position, otherwise check to see what the history was for that position. If it is a winning position for you then you will win, save the result into the history. If it is not then right will win and save that result into history.
  - Check to see if moving by value will cause n to go negative, if so L will win at that position, otherwise check to see what the history was for that position. If it is a winning position for you then you will win, save the

result into the history. If it is not then Left will win and save that result into history.

Let  $m$  and  $s$  be the sizes of the sets for the left hand player and the right hand player respectively. Let  $n$  be the length of the list that is being searched through. The asymptotic complexity for the algorithm is  $O(nms)$ .

Suppose that we did not have to list every safe position for each player then we can reduce the size that the algorithm takes up drastically. Because the memoization does not need to check every entry in the history, but only on the ones that can be effected by the elements in the move sets  $L$  and  $R$  we can reduce the size needed to be only the  $\max(\max(L\_hand), \max(R\_hand))$ . There will be no real asymptotic speed increase by doing this; however the program will run faster by reducing the strain on the machine.

When player  $L$ 's move set is  $\{2, 5, 9\}$  and player  $R$ 's move set is  $\{3, 4, 8\}$  then winning conditions for both players are as follows:

```
Left Player Win Conditions
  2  6 10 14 18 22 26 30 34 38 42 46 50 54 58 62 66 70
74 78 82 86 90 94 98
Right Player Win Conditions
```

When player  $R$ 's set is changed to  $\{3, 5, 9\}$  the result are as follows.

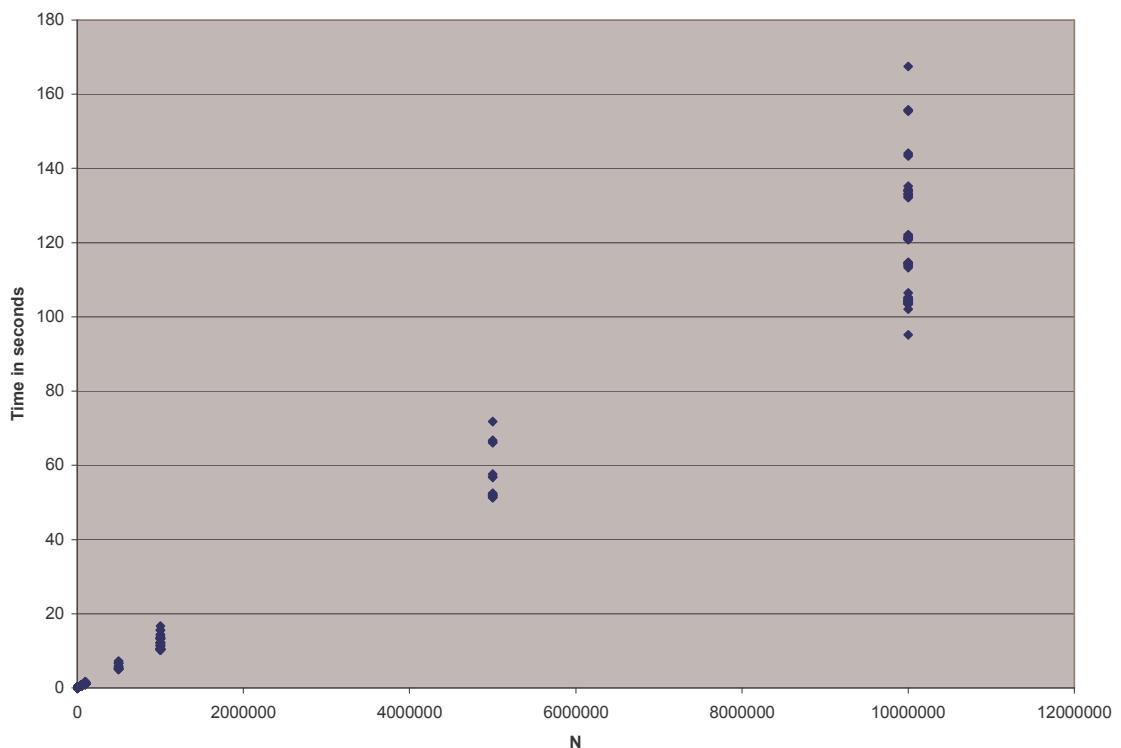
```
Left Player Win Conditions
  2  7 11 16 20 25 29 34 38 43 47 52 56 61 65 70 74 79
83 88 92 97
Right Player Win Conditions
```

Suppose that player  $R$ 's set has been changed to  $\{2, 4, 8\}$  the resulting set is as follows:

```
Left Player Win Conditions
  6 10 14 18 26 30 42 46 50 54 62 66 78 82 86 90 98
Right Player Win Conditions
  4  7 11 13 16 20 25 29 31 40 43 47 49 52 56 61 65 67
76 79 83 85 88 92 97
Safe positions
  2 22 34 38 58 70 74 94
```

When a player has a lower minimum value in their set then the other player they will be the only player to have safe positions. If both players have the same minimum value then the results will be split between the players.

I was wondering how much the size difference of the players hands effected the final run time of the algorithm. I know that the complexity of the algorithm is  $O(n*m*s)$  where  $m$  and  $s$  are the sizes of the Left and Right player set. However I wanted to actually see the difference. To find this the project ran 1000 times over the course of a few days with different data sets sizes and iterations. The details of how the project was run can be found in the acknowledgements section. The data was then gathered into a CSV file and then plotted using Microsoft Excel.



Each of the points on the graph represents a test of the project, because the project was tested using different sets for left and right hands you can see that at each

grouping they are arranged in a vertical line. The hands that had the longest length are at the top while the hands that had the shortest length were at the bottom. On this graph the growth of the function is roughly linear and is close to our prediction of  $O(n*m*s)$ .

## Acknowledgements

- Dr. While for providing assistance when my project was not going in the proper direction.
- Dr. Sherman for providing a very easy to follow guide to writing a technical paper.
- UMBC – OIT for providing computer time to compute this project.  
All test cases where data was gathered were taken from linux3.gl.umbc.edu, one

of the Linux servers available for student use. The program was run over night when the user count was between 20-50 and there were about 10 active uses. The load average was watched to make sure it goes no higher than 5.0. This was to help keep constant results with the data that was gathered. Linux3.gl.umbc.edu is a dual processor Pentium III 850 with 2 GB of memory.

## References

CSC 4170 the Pumping Lemma. 25Nov 2003

CSC 4170. 18Feb 1998.

<<http://www.netaxs.com/people/nerp/automata/pumping0.html>>

Theory of Nim. 24Nov 2003

Combination Games. 2003

<[http://www.cut-the-knot.org/nim\\_theory.shtml](http://www.cut-the-knot.org/nim_theory.shtml)>

## Appendix A – Source Code

The examples for the dynamic programming algorithms in the first two sections are provided here as working examples in Perl Code. I used Perl because of its ability to rapidly prototype functions and provided a great help in testing the recurrences that are in this project. For the last part, the Asymmetric Game is done in C++ with the results tabulated into a CSV file.

### The Simple Game

```
#!/usr/bin/perl
$n          = 100;
@set_s     = (1,5,8,10);
@history   = ();

#function to check to see if the supplied number
# is in the allowed moves set.
sub in_set
{
    my $n = $_[0];
    foreach $value (@set_s)
        { if ($n == $value) { return 1; } }
    return 0;
}

# The Recurrence
sub recurrence
{
    my $n = $_[0];
    my $v = -1;

    if ($n == 0)
        { return $history[$n] = 2; }

    if (in_set($n))
    {
        if (!$history[$n]) { $history[$n] = 1; }
        return 1;
    }

    foreach $value (@set_s)
    {
        if (($history[$n - $value]) && ($n - $value >= 0))
        {
            $v = $history[$n - $value];
            if ($v == 2)
                { return $history[$n] = 1; }
        }
    }
}
```

```

else
{
    if ($n - $value >= 0)
    {
        $v = recurrence($n - $value);
        if ($v == 2)
            { return $history[$n - $value] = 1; }
    }
}
$history[$n] = 2;
return 2;
}

#Loop through 0...n and call the recurrence.
for ($i = 0; $i <= $n; $i++)
{
    recurrence($i);
}

#Output player 1 winning conditions
print "Player 1 Winning Conditions:\n";
for ($i = 0; $i <= @history; $i++)
{
    if ($history[$i] == 1)
        { print " " ".$i; }
}
print "\n";

#Output player 2 winning conditions
print "Player 2 Winning Conditions:\n";
for ($i = 0; $i <= @history; $i++)
{
    if ($history[$i] == 2)
        { print " " ".$i; }
}
print "\n";

```

## Coderunner

This Perl script contains the code that will run the project code without user intervention.

```
#!/usr/local/bin/perl -w
# The name of the project executable file
$program = "project";
# The output file where the runtime data is to be stored
$csv_file = "data.csv";

# These are sample sets that I decided to use to run the project
@sets_L = ("2,5,9", "4,5,6", "4,6,8,10", "3,6,9");
@sets_R = ("3,4,8", "3,5,9", "1,2,3", "3,4,5,6,7,8");
# At each pairing of sample sets these data points are generated
@intervals = (100, 500, 1000, 5000, 10000, 50000, 100000, 500000,
1000000, 5000000, 10000000);

$total = @sets_L * @sets_R * @intervals;
$current = 1;
# Adds the start date and time to the csv file
`date > $csv_file`;
foreach $set_L (@sets_L)
{
    foreach $set_R (@sets_R)
    {
        foreach $interval (@intervals)
        {
            # run the program, append the results to a csv file
            print "$current / $total - $interval $set_L $set_R\n";
            print ` $program $interval $set_L $set_R >> $csv_file `;
            $current++;
        }
    }
}
# Adds the end date and time to the csv file
`date >> $csv_file`;
```

## Makefile

```
#####
##  Filename: Makefile
##  Creator:  Adam Fort - afort1@umbc.edu
##  Version:  1.0
##
##  Builds the program by typeing make
##
##  Requires G++ and a the source code placed into
##  a writeable directory.
##
##  This makefile was built off of a makefile used
##  in CS341 and some makefiles that are used to
##  compile programs for G++.
##
##  List of revisions:
##  Date | Coder | Description
##
#####

compiler      = /usr/local/bin/g++
compilerflags = -ansi -Wall -g
PROJECT       = project

OBJECTS       = Timer.o Project.o

$(PROJECT): $(OBJECTS)
    $(compiler) $(compilerflags) -o $(PROJECT) $(OBJECTS)

Project.o: Project.cpp Project.h
    $(compiler) $(compilerflags) -c Project.cpp

Timer.o: Timer.cpp Timer.h
    $(compiler) $(compilerflags) -c Timer.cpp

#-----

clean:
    touch foo.o
    /bin/rm -rf *.o
    /bin/rm -f core.*
    /bin/rm -f $(PROJECT)
```

## Project.h

```
/*
*****
**
** CMSC 441 Algorithms Project
** Project description available at
** http://www.csee.umbc.edu/~tadwhite/441.f03/games.html
**
** Code written by Adam Fort
**
** Filename: project.h
** Date: 11/20/03
**
** This file contains the entire code for the project.
*****/
#ifndef __project_h__
#define __project_h__

#include <vector>
using namespace std;

void parse_string(vector < int >* set, char* hand);
void game(int n);
void reccurance(int n);
bool in_set(vector < int > array, int target);
int vector_min(vector < int > array);
#endif
```

## Project.cpp

```

/*****
**
** CMSC 441 Algorithms Project
** Project description available at
** http://www.csee.umbc.edu/~tadwhite/441.f03/games.html
**
** Code written by Adam Fort
**
** Filename: project.c
** Date: 11/20/03
**
** This file contains the entire code for the project.
*****/
#include <iostream>
#include <stdlib.h>
#include <cstdlib>
#include <vector>
#include <limits.h>
#include "Project.h"
#include "Timer.h"
// If defined, this turns on additional output about the program
// Turned off to speed out output
// #define VERBOSE

using namespace std;
using namespace _TIMER_;

// Global variables are bad...But I am lazy, so here they are good.
// The history vector, contains the memoization for the algorithm
vector < vector < int > > history;
// The allowed moves for each player
vector < int > l_player;
vector < int > r_player;
// Winning conditions of each player, could just save the one
// but saved them all to help debugging
vector < int > L_condition;
vector < int > R_condition;
vector < int > S_condition;

int main(int argc, char **argv)
{
    int    n;
    timer  stopwatch;
    double total_time;

    if (argc < 4)
    {
        cout << "Usage: " << argv[0] << " n hand1 hand2" << endl;
        exit(-1);
    }
    // Build the player hands
    n = atoi(argv[1]);
    parse_string(&l_player, argv[2]);
    parse_string(&r_player, argv[3]);

```

```

// Declare the array
history.resize(2);
for (unsigned int i = 0; i < 2; i++)
    history[i].resize(n);
for (unsigned int j = 0; j < 2; j++)
    for(int i = 0; i < n; i++)
        history[j][i] = ' ';

// Start the timer, run the game, then stop and record the time
stopwatch.start();
game(n);
total_time = stopwatch.stop();

// Additional output used for debugging
#ifdef VERBOSE
    cerr << "Starting run with n = " << n;
    cerr << endl << "Left players move table is:" << endl;
    for(unsigned int i = 0; i < l_player.size(); i++)
        { cerr << "\t" << l_player[i]; }
    cerr << endl << "Right players move table is:" << endl;
    for(unsigned int i = 0; i < r_player.size(); i++)
        { cerr << "\t" << r_player[i]; }
    cerr << endl;

    cerr << "Left Player Win Conditions" << endl;
    for(unsigned int i = 0; i < L_condition.size(); i++)
        {
            cerr << " " << L_condition[i];
        }
    cerr << endl;

    cerr << "Right Player Win Conditions" << endl;
    for(unsigned int i = 0; i < R_condition.size(); i++)
        {
            cerr << " " << R_condition[i];
        }
    cerr << endl;

    cerr << "Safe positions" << endl;
    for(unsigned int i = 0; i < S_condition.size(); i++)
        {
            cerr << " " << S_condition[i];
        }
    cerr << endl;
#endif
// Output the data to SDOUT to be redirected to a csv file
cout << n << "," << total_time << endl;
return 0;
}

void parse_string(vector < int >* set, char* hand)
{
    char* buffer = NULL;
    while (true)
    {
        if (buffer == NULL)

```

```

        {
            buffer = strtok(hand, ",");
        } else {
            buffer = strtok(NULL, ",");
        }
        if (buffer == NULL) { break; }

        set->push_back(atoi(buffer));
    }
    return;
}

void game(int n)
{
    for (int i = 0; i < n; i++)
    {
        reccurance(i);
        // Check for each winning condition and place it on the data
        if ((history[0][i] == 'L') && (history[1][i] == 'L'))
        {
            L_condition.push_back(i);
        }
        if ((history[0][i] == 'R') && (history[1][i] == 'R'))
        {
            R_condition.push_back(i);
        }
        if ((history[0][i] == 'L') && (history[1][i] == 'R'))
        {
            S_condition.push_back(i);
        }
    }
}

void reccurance(int n)
{
    // You cant move, then you lose
    if (n < vector_min(l_player))
    {
        if (history[0][n] == ' ')
            history[0][n] = 'R';
    }
    // You cant move, then you lose
    if (n < vector_min(r_player))
    {
        if (history[1][n] == ' ')
            history[1][n] = 'L';
    }

    if (in_set(l_player, n))
    {
        if (history[0][n] == ' ')
            history[0][n] = 'L';
    }

    if (in_set(r_player, n))
    {

```

```

    if (history[1][n] == ' ')
        history[1][n] = 'R';
}

if ((history[0][n] != ' ') && (history[1][n] != ' '))
{
    return;
}
for ( unsigned int li = 0; li < l_player.size(); li++)
{
    if (n - l_player[li] >= 0)
    {
        if (history[1][n - l_player[li]] == 'L')
            history[1][n] = 'L';
        else
            history[1][n] = 'R';
    }
}
if (history[0][n] == ' ')
    history[0][n] = 'R';

for ( unsigned int ri = 0; ri < r_player.size(); ri++)
{
    if (n - r_player[ri] >= 0)
    {
        if (history[0][n - r_player[ri]] == 'R')
            history[0][n] = 'R';
        else
            history[0][n] = 'L';
    }
}
if (history[1][n] == ' ')
    history[1][n] = 'L';
return;
}
// check if the current move is legal
bool in_set(vector < int > array, int target)
{
    for(unsigned int i = 0; i < array.size(); i++)
    {
        if (array[i] == target)
            return true;
    }
    return false;
}
// Find the smallest legal move
int vector_min(vector < int > array)
{
    int min = INT_MAX;
    for(unsigned int i = 0; i < array.size(); i++)
    {
        if (array[i] < min)
            min = array[i];
    }
    return min;
}

```

## Timer.h

```

#ifndef __TIMER_H_
#define __TIMER_H_

#include <ctime>

namespace _TIMER_
{
    class timer
    {
    private:
        int running;
        double accumulated_time;
        clock_t start_time;

        timer(const timer& t);
        void operator=(const timer& t);

    public:
        timer() : running(0), accumulated_time(0) { }
        ~timer() { }

        void start(const char* timer_name = 0);
        void restart(const char* timer_name = 0);
        double stop(const char* timer_name = 0);
        double elapsed_time();
        double check(const char* timer_name = 0);
    };
};

#endif

```

## Timer.cpp

```

using namespace std;

#include <ctime>
#include <iostream>
#include <iomanip>
#include "Timer.h"

using namespace _TIMER_;

double timer::elapsed_time()
{
    clock_t curr_time = clock();
    return double(curr_time - start_time) / CLOCKS_PER_SEC;
}

void timer::start(const char* timer_name)
{
    if (running)    return;

    running = 1;
    start_time = clock();
}

double timer::check(const char* timer_name)
{
    if (running)
        return accumulated_time + elapsed_time();

    return accumulated_time;
}

void timer::restart(const char* timer_name)
{
    accumulated_time = 0;
    running = 1;
    start_time = clock();
}

double timer::stop(const char* timer_name)
{
    if (running)
        accumulated_time += elapsed_time();
    running = 0;
    return accumulated_time;
}

```