Simulating the Pick-up Stones game: A dynamic approach

Thomas Fisher
Department of Computer Science and Electrical Engineering,
University of Maryland Baltimore Country, tom.fisher@umbc.edu

Abstract

The classic pick-up stones game, where some variable number of stones is on a table and two players take turns removing some number of stones from a set of legal moves, has been played for years. The game can be fundamental in the development of logical thinking and strategy. Each game can vary by the number of stones, the legal moves, playing asymmetrically (where each player has a different set of legal moves) and by factoring in rules that prevent players from repeating the opponents previous move.

The pick-up stones game can be simulated using a computer with an implemented dynamic programming algorithm in linear time with respect to the number of stones. Most variations of the game can be simulated in a linear (or almost linear) time, with the only disparity being the constant term in the asymptotic complexity. This constant term is effected by the type of game and mainly by the size and selection of legal moves. With slight variations, the overall analysis of the simulation of the game can be achieved in linear-time algorithms.

When analysis is performed checking against other criteria (besides the increase of the number of stones), such as an increase in the size of the set and the number of stones, the experimental run-time drastically becomes worse, but theoretical analysis suggest the run-time will eventually become linear, with the slope of the line being comparatively large.

Keywords – dynamic programming, pick-up stones, asymptotic complexity, analysis, simulation, two-player games, gaming theory.

1 Introduction

Two-player games are the most studied type in game theory. The classic game of two players picking up a variable number of stones until no stones or no legal moves remains is a classic example.

An Example: Given two players, say Kevin and Amy, a set of legal moves, say {1, 2, 3} and the number of stones, say 12, with Kevin going first, each player will remove stones from the table. Kevin may remove 3 and now only 9 stones remain. Amy removes 3 stones and leaves 6. Kevin again removes 1, so only 5 stones remain. Amy removes 1 stone and then Kevin picks up 3 stones. At this point, only one stone remains, Amy picks it up and now there are no legal moves left for Kevin since no stones are remaining, so Amy wins!

Variations of this game exist as well, with rules that limit the possible moves by prohibiting previous moves and giving each player their own set of legal moves. This paper and the research conducted outlines the steps to design a strategy using dynamic programming to simulate various implementations of the pick-up stones game. Several dynamic algorithms are implemented in the process and experimental timing is conducted to demonstrate the sample run-time matches that of the asymptotic analysis.

2 The Classic (Warm-up) Game

2.1 Game Description

Two players are playing with some n number of stones on a table; the legal moves are contained in a set S. With the assumption that both players play optimally (after all they do want to win), who will win? If n=0, the second player will obviously win, since player one has no legal move and there are no stones left to remove.

2.2 Method to simulate this game

This game can easily be simulated using a dynamic algorithm and an array with n+1 elements. The algorithm's implementation will be recursive and can be demonstrate with the following recurrence:

$$f(n) = \begin{cases} 2 & \text{if } n=0 \\ 1 & \text{if } \max\limits_{1\le i\le m}\{f(n-S_i)\} = 2 \\ 2 & \text{if } \max\limits_{1\le i\le m}\{f(n-S_i)\} = 1 \end{cases}$$

$$\text{when } n - S_i \ge 0 \text{ and } m = |S|$$

Following the simple convention, a 2 indicates a player two winning position and a 1 indicates a player one winning position. The recurrence can be implemented easily using recursion and an array of size n+1. We include the dynamic programming element, the array, to store previous known values to save on look up time, if we wish to run the algorithm again. We start by checking to see if n=0, if so the winner is a 2. If its not, then we look at each of the previous winning positions based on the current legal moves (i.e. if n=13, and the legal moves are 1, 2, and 3, we will check who is the winner at positions n = 12, 11, and 10). If you can move to a position marked 2 (ie. $\max\{f(n\text{-}S_i)\}$=2), the current n is a player 1 winning position since by moving there would make the current player, player 1, the winner. If no move to a player 2 winning position is available, then the current n is a player 2 winning position. If an array element is empty (stored with a 0 in below) at any given position $n_i \le n$ we make a recursive calls to the function. Pseudo-code for the algorithm is provided below (the following pseudo-code can determine and will return the winner at a position of n-stones.).

```
Who_Wins(Int. Array A, Int. n, Int. Array S, Int. m)
 1:  if n = 0
```

```
 2:  then A[0] = 2
 3:       return 2
 4:  else for i ← 0 to m
 5:       do   if n-S[i] ≥ 0 and A[n-S[i]] = 2
 6:            then A[n] = 1
 7:                 return 1
 8:            else if n-S[i] ≥ 0 and A[n-S[i]] = 0
 9:                 then x ← Who_Wins(A, n-S[i], S, m)
10:                      if x = 2
11:                      then A[n] = 1
12:                           return 1
13:       A[n] = 2
14:       return 2
```

The complexity of this algorithm is quite simple. Lines 1-3, 5-8 and 10-14 are clearly constant. Line 4 introduces a **for** loop that iterates m times. Line 9 is the possible recursive call that will visit every n in the worst case. The algorithm will execute m times at each position and there are n+1 positions giving an overall complexity of O(mn). This appears to be somewhat quadratic, but in practice, $m<<n$ and as $n\rightarrow\infty$, the algorithm will act very linear. It is also obvious that due to the use of the dynamic approach, each element in the array will only be calculated a single time; every other occurrence at a particular element is accessed from the array in a constant time. There are n+1 elements, again leaving an almost linear-time algorithm. The complexity of the memory use for this algorithm is O(m+n) + memory used by the recursive calls. A clever programmer could use array elements of 2-bits to save the winners (this can be done with all algorithms described in this paper, but this point is only mentioned at here). It should also be noted that since the algorithm works recursively, that with certain specific sets S, sorting the set S in a decreasing order may improve the run-time in practice but will have no effect on the complexity. This is because the number of recursive calls is decreasing by some value in S, we may potentially find a 2 quicker if we deduce by the bigger value, making less recursive calls. But as noted, this is only in some special cases of S and in general will not improve the complexity of the algorithm.

2.3 An example exercise

We can use the above algorithm to simulate a game between two players. Given the legal moves S = {1, 5, 8, 10}, we can calculate the winner at any given position (ie. a player 1 or player 2 win) n. Below are the winners for up to n=100. The tables are self-explanatory. Pick an n_i, say 68, look below, and we see a 1. That means with 68 stones on the table, the player going first is going to win, assuming both players play optimally.

n_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
winner	2	1	2	1	2	1	2	1	1	1	1	1	1	2	1	2	1	2	1

n_i	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
winner	2	1	1	1	1	1	1	2	1	2	1	2	1	2	1	1	1	1

n_i	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
winner	1	1	2	1	2	1	2	1	2	1	1	1	1	1	1	2	1	2

n_i	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
winner	1	2	1	2	1	1	1	1	1	1	2	1	2	1	2	1	2	1

n_i	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90
winner	1	1	1	1	1	2	1	2	1	2	1	2	1	1	1	1	1	1

n_i	91	92	93	94	95	96	97	98	99	100								
winner	2	1	2	1	2	1	2	1	1	1								

Table(s) 1: Winning positions for n_i where $0 \leq i \leq 100$ with set $S = \{1, 5, 8, 10\}$

3 The No Repeat Game

3.1 Game Description

The no repeat game is similar to the game described in section 2 with the stipulation that neither player can repeat their opponent's previous move. For example, if Kevin and Amy are playing, and Kevin removes 5 stones from the pile, with the set of legal moves being {1, 3, 5}, then Amy can only remove 1 or 3 stones.

3.2 Safe Position Definition

With the stipulation that you cannot repeat moves, it no longer makes sense to simulate the game for winning positions since you don't know what the optimal play will be in all cases since it depends on the opponent's previous move. Therefore, we define a position n to be **safe**, if whenever there are n stones in the pile, no matter what the previous move was, there is a winning strategy for the second player. If the safe positions can be determined, the optimal move for any given player will be to the safe position. For example, suppose some position k was known to be safe and it was my move, and I could legally move to the k^{th} position and guarantee myself victory since I would be the second player to move when the game is played at the k^{th} position.

3.3 The Repeating Phenomena

A keen observer may notice in the results to section 2.3 that the winning positions for each player seems to repeat. This is not a singular event; in fact calculations of the safe positions repeat as well. This phenomenon arises from the fact that the determination of whether or not a given position k is safe depends on the previous j positions, where j is the maximum element in the set of legal moves (ie. S).

Similar to the algorithm in section 2, the determination of safe positions can be solved using a dynamic algorithm, but with a matrix, rather than a simple array. The rows of the matrix are determined by cardinality of S and the columns by the value of n. By expanding the matrix, it is obvious that positions are dependent upon the previous values and that it does repeat. Given the

set of legal moves, $S = \{1, 2, 3, 4, 5\}$, the safe positions occur at $n = 0, 7, 13, 20, 26, 33, \ldots$ or at 0 mod13 and 7 mod13. That is, a position k is safe if $[k]_{13} \equiv [0]_{13}$ or $[k]_{13} \equiv [7]_{13}$. Appendix A, section A.1 includes a formalize proof of this phenomenon with the above given set. In general, this will always be the cases, and the repeating factor can be found and is dependent upon the largest element in S.

3.4 Determining a Safe Position

Solving whether a position is safe or not is similar to the dynamic algorithm given in section 2. The difference is that we are no longer looking for just one possible move resulting in a win, but rather the case where all possible moves would result in a win. Since a safe position has been defined as a position where the second player is guaranteed victory, we check to make sure no possible moves from the current position result in a player 2 winning position. If it is my turn, and there are 9 stones, and 7 is a known safe position, and 2 is a legal move, 9 cannot be a safe position since moving from 9 to 7 would result in myself winning. But I made the first move from 9, whence it is not a guaranteed player 2 winning position and not safe. The algorithm can be demonstrated with the following recurrence:

$$f(n) = \begin{cases} 2 & \text{if } n=0 \\ 1 & \text{if any } f(n - S_i) = 2 \\ & \quad 1 \le i \le m \\ 2 & \text{otherwise} \end{cases}$$

$$\text{when } n - S_i \ge 0 \text{ and } m = |S|$$

The algorithm that implements the recurrence constructs a matrix with m rows and n+1 columns where $m = |S|$ and n is the number of stones. From some arbitrary position, we check all possible moves, that is m possible moves since we do not know what the previous move was. Within each move, there are *m-1* possible moves, since we must ignore the previous move made. If no moves result in a player 2 win, then that particular element of the matrix is safe. We store known results in the matrix for quick access when many calls are going to be made. When an entire column contains nothing but 2's, that particular n position is safe. Sample pseudo-code is given below.

```
Is_Safe(Int. Matrix (mXn) A, Int. n, Int. S, Int. m)
 1:  for i ← 0 to m
 2:  do   x ← Is_Safe_Aux(A, n, S, m, i)
 3:       if x = 1
 4:       then return 1  (i.e. Not Safe)
 5:  return 2  (i.e. must be Safe)

Is_Safe_Aux(Int. Matrix A, Int. n, Int. S, Int. m, Int. prev)
 6:  if A[prev][n] ≠ 0
 7:  then return A[prev][n] (i.e. it's a stored value)
 8:  if n = 0
```

```
 9:  then A[prev][n] = 2
10:       return 2
11:  for j ← 0 to m
12:  do   if j ≠ prev and n - S[j] ≥ 0
13:       then x ← Is_Safe_Aux(A, n-S[j], S, m, j)
14:            if x = 2
15:            then A[prev][n] = 1
16:                 return 1
17:  A[prev][n] = 2
18:  return 2
```

The analysis of this algorithm follows that of the first algorithm described above in section 2. Most of the code is performed at a constant or O(1) time, the exceptions being lines 2 and 11 with the **for** loops. Each of these loops is based on the size of *S* and are embedded within one-another. Line 13 makes a recursive call, access all n+1 values in the worst case. Keeping in mind we have an mXn matrix of values. In the worse case scenario, each column will be accessed and the embedded for loops will each execute fully at each column resulting in a $O(m^2n)$ run-time. Again we should note that in most cases, $m<<n$ and that as $n\rightarrow\infty$, the algorithm will run in an almost linear-time. It is also obvious that since the algorithm uses a dynamic approach, each element of the matrix will only be calculated a single time and accessed afterwards in a constant fashion, this allows the algorithm to run in a more $O(mn)$ time the more it is exercised. Memory complexity is dependent on the size of the matrix, $O(mn)$, and the recursive calls made.

3.5 A Generalized Form

The question about generalizing the algorithm to account for more than a single previous move, say some k previous moves, where $1 \leq k < m$ (having $k = m$ makes no sense as there would be no legal moves!) arises almost instantly. One may ask, can this be done in an efficient time? The algorithm to do so does not vary much from that described in section 3.4. Instead of a single matrix of size mn we have an array of k matrices, or a three-dimensional array of size kmn. The difference in the algorithm would be to check each of the possible previous moves against each of the k matrices. This can be done by modifying the algorithm in section 5.4 to include km recursive calls rather than the m calls made previously. The outside function, `Is_Safe()` would also have to take into account the k matrices by checking each matrix. By this, we mean that if we want to know if some position n is safe, we would have to check n with all the possible k-previous moves (i.e. km). The analysis of the algorithm follows that previously stated with the k factor contributing to the run-time. The run-time would follow; $O((km)^2n)$ in the worst case running time. As noted before, in practice, $k \leq m << n$, so the algorithm would act in an almost linear fashion with respect to n. Like the previous algorithms, since the algorithm is dynamic we take advantage of storing previous known values, so each element in the 3D array will only be calculate once, and then accessed afterwards in a constant time, leaving a $O(kmn)$ run-time the more the algorithm is accessed. Likewise for the memory use, $O(kmn)$, and the recursive calls made.

4 An Asymmetric Game

4.1 Game Description

The asymmetric game occurs when the players have different sets of valid moves. Because of the different sets, it no longer makes sense to discuss player-1 winning position or a player 2 safe position. From tradition, we have two players, namely Lillian and Rekha, representing Left and Right respectively. Each player has a set of valid moves. We will call these sets, S_L and S_R, respectively for Lillian and Rekha.

4.2 Determining Safe Positions

The algorithm for determining safe positions does not vary much from determining safe positions in section 3.4. The main difference is the use of two arrays, one for each player giving the possible legal moves. The idea is the same, look for <u>all</u> potential moves seeing if they result in a player 1 win, then the current position is considered safe. The only difference is that you check the opposing players' possible moves. The simulated game can be represented by the following recurrence:

$$f(n,S^B)=\begin{cases}2 & \text{if } n=0\\ 1 & \text{if any } f(n-S_i^B,S^A)=2 \\ & \quad 1\le i\le m\\ 2 & \text{otherwise}\end{cases}$$

When S^A is one players given moves

and S^B is the opponents valid moves

and $n-S_i^B\ge 0$ and $m=|S^B|$

If we wish to determine the safe positions for all positions up to some n we can put the algorithm in a loop to check all moves from 0 to n. Pseudo-code for determining all the safe positions up to n is included below.

```
n_Safe(Int. n, Int. Array S_L, Int. m_L, Int. Array S_R, Int m_R)
 1:  Initialize Int. Array A_L(m_L X (n+1)), A_R(m_R X (n+1))
 2:  for i ← 0 to n   (Lillian going first)
 3:  do   x = n_Safe_Aux(n, A_L, S_L, m_L, A_R, S_R, m_R)
 4:       output x
 5:  for i ← 0 to n   (Rekha going first)
 6:  do   x = n_Safe_Aux(n, A_R, S_R, m_R, A_L, S_L, m_L)
 7:       output x
 8:  Destroy A_L and A_R

n_Safe_Aux(n, A_1, S_1, m_1, A_2, S_2, m_2)
 9:  for i ← 0 to m_2
```

```
10:  do   x = Is_Safe(n, A1, S1, m1, A2, S2, m2, i)
11:       if x = 1
12:       then return 1
13:  return 2

Is_Safe(n, A1, S1, m1, A2, S2, m2, move)
14:  if A2[move][n] ≠ 0
15:  then return A2[move][n]
16:  if n = 0
17:  then A2[move][n] = 2
18:       return 2
19:  for i ← 0 to m1
20:  do   if n - S1[i]
21:       then x = Is_Safe(n-S1[i], A2, S2, m2, A1, S1, m1)
22:            if x = 2
23:            then A2[move][n] = 1
24:                 return 1
25:  A2[move][n] = 2
26:  return 2
```

The complexity of this algorithm looks complicated, but like the other algorithms discussed turns out to be very linear in practice. Each of the loops in the `n_Safe` function is independent, running in a O(n) time, we have embedded **for** loops at lines 9 and 19 based on two different variables, the maximum of the two will dominate and we make n+1 recursive calls in the worst case resulting in an overall O($\max\{m_1, m_2\}^2 n$) time. Now we are running this for all 0 to n and it appears the algorithm should be worse, but because of the dynamic approach we are saving the previous results so each n+1 will only be accessed a single time leaving an overall O($\max\{m_1,m_2\}^2 n$) run-time. As with the other algorithms, in practice m_1, $m_2 << n$, therefore making the algorithm act in a linear fashion with respect to n. Again, since the algorithm is dynamic and we use previous results (the for loops in `n_Safe` implement a bottom-up approach) and access them in a constant time. The algorithm populates the two arrays, which is dictated mainly by the size of the arrays, resulting in a O($m_1 n + m_2 n$) or O($(m_1+m_2)n$) run time; again very linear in practice. Complexity of the memory use is O($m_1 n + m_2 n$) for the matrices and any recursive calls made.

4.3 The repeating phenomena returns

In exercising a few rounds of the asymmetric game, it may quickly become apparent that depending on the set of legal moves, the safe positions may repeat or result in a game where the same player is going to win after a certain number of stones is placed on the table. One such game is one where Lillian's legal moves are {2, 5, 9} and Rekha's moves are {3, 4, 8}. Some sample output from an implementation of the algorithms above is included below; the resulting number corresponds to who will win in the optimally played game (i.e. a 2 is a safe position for the second player in the given game).

```
n = 25
S for Lillian = { 2, 5, 9}
S for Rekha   = { 3, 4, 8}

If Lillian goes first.
----------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2

If Rekha goes first.
--------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  1  1  2  2  1  1  1  2  2  2
```

From the sample output above it appears that if 10 or more stones are on the table, and both players play optimally, Lillian is going to win every time. This is a fact, and the proof follows that of the repeating phenomena in the symmetric game. A formalized proof is included in the appendix (A.2).

Similar events happen with different sets of moves, if Rekha's moves were {3, 5, 9} and Lillian has the same moves as the previous example, the safe position results follow a distinct pattern, but no guaranteed winner seems to appear. Sample output is included below.

```
n = 27
S for Lillian = { 2, 5, 9}
S for Rekha   = { 3, 5, 9}

If Lillian goes first.
----------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  1  1  1  1  1  2  1  1  1  1  1  1  2  1  1  1  1  1  1  2  1  1  1  1  1  1  2  2

If Rekha goes first.
--------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  1  2  1  1  2  2  2  1  2  1  1  2  2  2  1  2  1  1  1  2  2  1  1  1  1  2  2  2
```

It is apparent in this above sample that the player who goes first is very important. In fact, as the output shows, if Lillian goes first, only 5 safe positions are found, leaving much hope for Lillian's changes of winning the game. Rekha isn't as lucky, but can win in about 50% of the games when going first since about half of the positions are found to be safe. The patterns continue in this same fashion.

The guaranteed winner phenomenon is not limited to when Lillian and Rekha have the same number of moves as seen in this sample outputs below.

```
n = 27
S for Lillian = { 2, 3, 5, 7, 9}
S for Rekha   = { 2, 3}

If Lillian goes first.
----------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2

If Rekha goes first.
--------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  1  1  1  2  2
```

```
n = 27
S for Lillian = { 2, 4, 5, 6, 7, 9}
S for Rekha   = { 3, 8}

If Lillian goes first.
----------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2

If Rekha goes first.
--------------------
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-------------------------------------------------------------------------------------
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  1  1  2  2  2  1  1  2  2  2
```

In both of the above cases, Lillian appears to be guaranteed victory if some number of stones is present. As mentioned before, this phenomenon arises from the fact that the current position is determined by the moves the opponent can make, specifically the previous s_m moves, where s_m is the largest element in the set of the opponent's legal moves. In the two above cases, it should be apparent that Rekha's only chance of victory is if there are a small number of stones remaining, otherwise Lillian can control the game and guarantee victory.

4.4 Determining if a position is safe

If you wanted to just determine whether or not a particular n is safe, you could use only the `n_Safe_Aux` and `Is_Safe` functions to do so (this is more of a pure recursive technique), just specifying the n. The asymptotic complexity of this algorithm would be no different than that of determining the safe positions up to n. In the worse case, every column in the array would have to be visited and two arrays of the same size as above would be used, resulting in no saving of memory. Like the algorithm in section 4.2, finding if a position was safe or not would be done in a linear time with respect to n since $n >> \max\{m_1, m_2\}$ in practice.

5 Some Experimental Analysis

5.1 Introduction

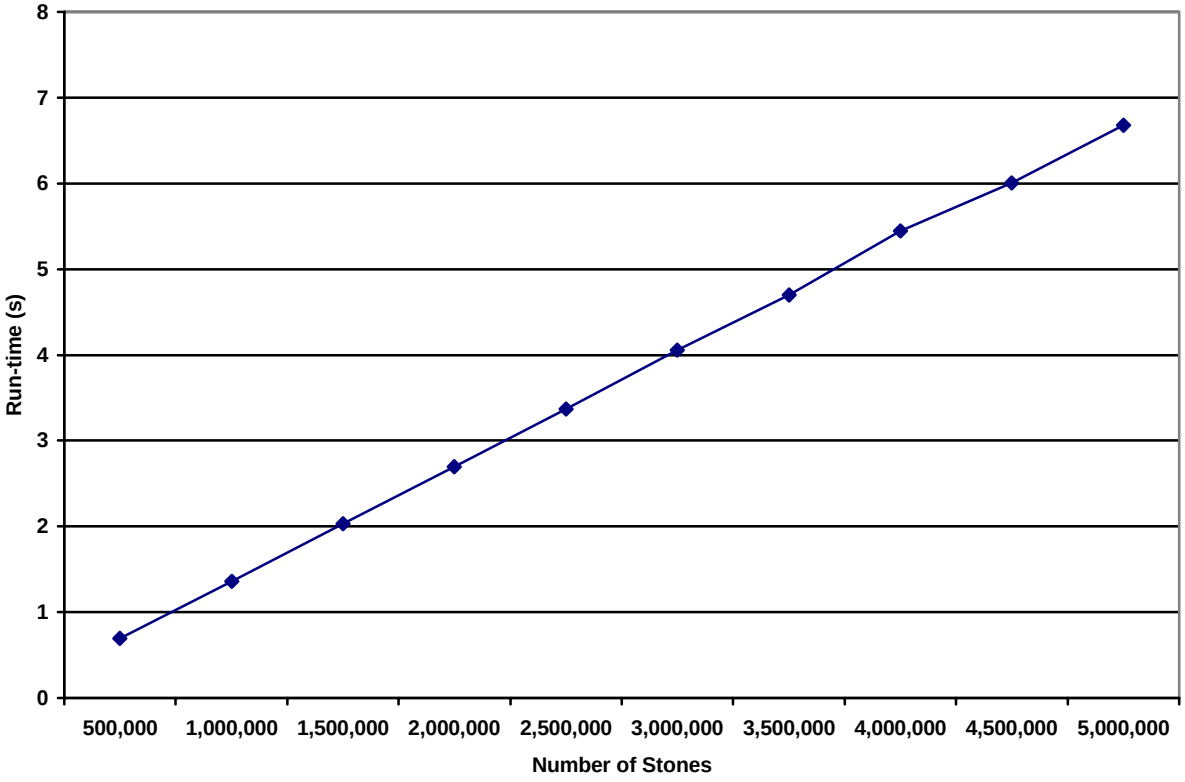
In the following experimental analysis, the algorithm was implemented (see appendix B for source code) to find all the safe positions from 0 to a given n (algorithm given in section 4.2 above) based upon the asymmetric game. The timing of the algorithm was computed by finding the average of 50 executions of the algorithm. So at a given n, the algorithm was executed 50 times (from scratch, i.e. empty matrices) and the average time was taken. Each of the tested n values is in tables in the following sections. All experimental sampling was tested using a timing program (included in Appendix B – Source Code) and was exercised on a PC system with an AMD Athlon 700 MHz with 256 L2 Cache, 256MB RAM, a 200MHz system BUS running Debian GNU/Linux 2.4.21 with a 256 MB Swap space and no major processes running (e.g. nerdy statistical checking software like SETI).

5.2 Asymmetric Game 1

The claim has been repeatedly made in this report that many of these algorithms run in an almost linear time. Some experimental analysis was conducted to test this claim. This is a subset of possible testing that can be conducted. The following uses Lillian's legal moves as $S_L = \{2, 5, 9\}$ and Rekha's legal moves as $S_R = \{3, 4, 8\}$. The difference marks the difference between the current and previous run-time. Linearity of the run-time can be seen in the table and graph below.

Average run-time (out of 50)					
n	run time	difference	n	run time	difference
500,000	0.69160	---	3,000,000	4.05260	0.6808
1,000,000	1.36300	0.6714	3,500,000	4.69620	0.6436
1,500,000	2.03160	0.6686	4,000,000	5.44620	0.7500
2,000,000	2.69800	0.6664	4,500,000	6.00440	0.5582
2,500,000	3.37180	0.6738	5,000,000	6.67840	0.6740

Table 2: Run-times (seconds) and number of stones (n) for Game 1



Graph 1: Number of stones vs. Run-Times (in seconds) for Game 1

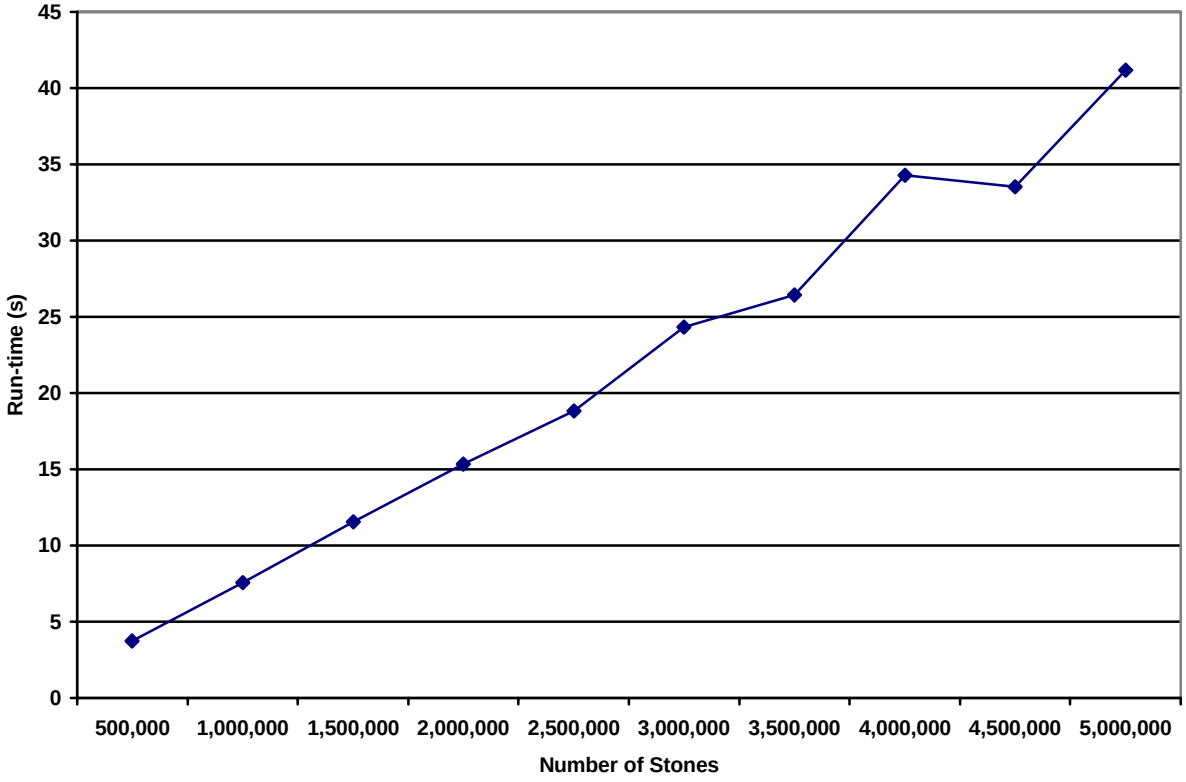
As the graph shows, the run time is very linear. The average difference in run-times between any two points are 0.6652 over every 500,000 stones. Based on this we should be able to predict the outcome of some other number of stones, say n=10,000,000, our prediction is 6.67840 + 10*(0.6652) = 13.3304. The actual run-time (experimental derived like other sample values by a 3-line modification of the source code to specify n=10,000,000) is 13.44680, which produces a relative error of 0.0086563 (0.86%) which implies the linear form continues.

5.3 Asymmetric Game 2

The first example was run with $m_1 = m_2 = 3$, but what happens when the values vary and are greater. Let S_L = {2, 4, 5, 6, 7, 9, 12, 15, 18, 21, 23} and S_R = {3, 4, 5, 6, 7, 9, 11, 13, 19}.

Average run-time (out of 50)					
n	run time	difference	n	run time	difference
500,000	3.74620	---	3,000,000	24.33840	5.49566
1,000,000	7.54400	3.7978	3,500,000	26.43280	2.0944
1,500,000	11.57500	4.031	4,000,000	34.30060	7.8678
2,000,000	15.34500	3.77	4,500,000	33.54440	-1.2438
2,500,000	18.82740	3.4824	5,000,000	41.17520	7.6380

Table 3: Run-times (seconds) and number of stones (n) for Game 2



Graph 2: Number of stones vs. Run-Times (in milliseconds) for Game 2

Again, the graph shows a very linear run-time although a few interesting instances occur for the larger number of stones, but the graph is still overall very linear. The overall average difference is 4.10. If you were to compute the average distance of the first half (where the line visual looks perfectly linear), you get an average distance of 4.12. If we were to use that distance and predict (just like above) the value at n=5,000,000 (prediction is 40.799888), we get a relative error of 0.009115 (0.91%). This suggests the little blips on the graph are strictly produced as part of the overall sampling error. We use the overall average distance to accurately predict the run time at n=10,000,000 (predicted: 82.1752; actual: 82.42500), the result leaves a relative error of 0.00303 (0.03%), again suggesting the run-time will remain linear.

5.4 Comparison of the two example games

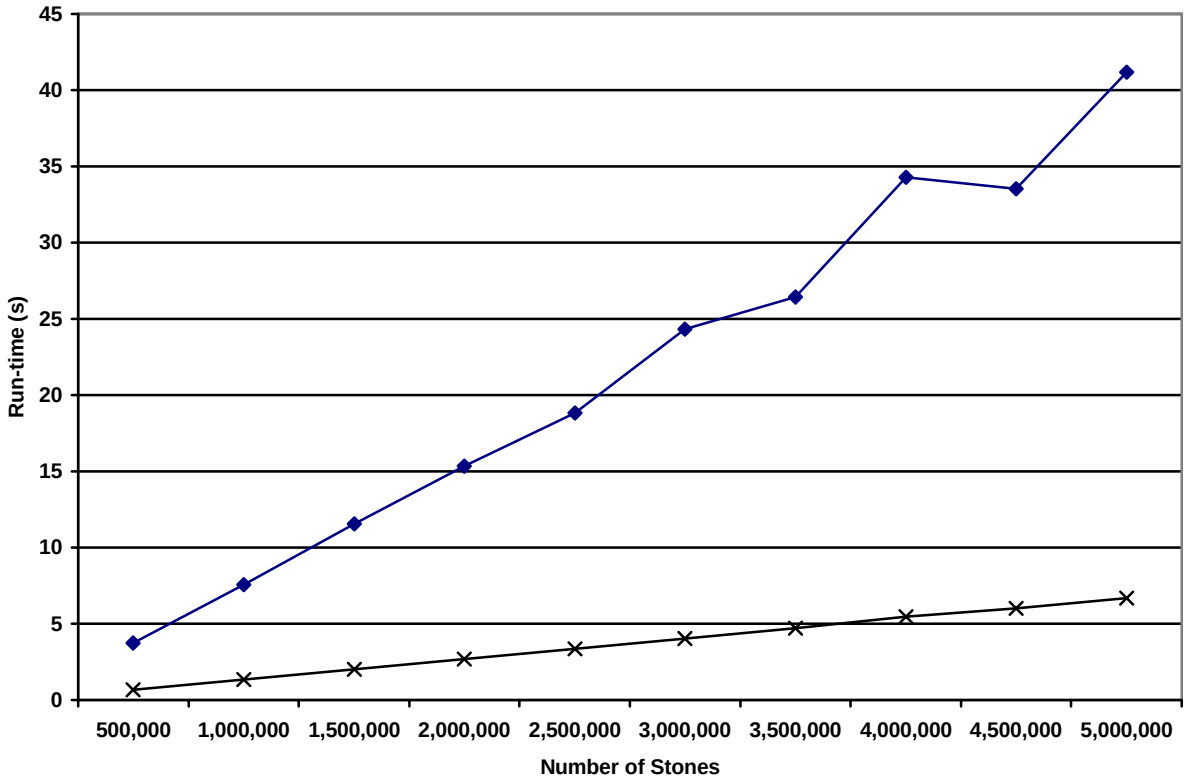
To show the relationship between the two examples above, the following table was constructing computing the comparison of the average at each n for the examples. For instance, in the table below, at n = 500,000, we take the run-time of game 2 at n = 500,000 and divide by the average run-time of game 1 at n = 500,000 to get a ratio of 5.41. The ratio computed is the numerator portion of the divided difference between any two points.

Ratio between Game 1 & 2			
n	ratio	n	ratio
500,000	5.41	3,000,000	6.01

Ratio between Game 1 & 2			
1,000,000	5.53	3,500,000	5.63
1,500,000	5.69	4,000,000	6.29
2,000,000	5.68	4,500,000	5.58
2,500,000	5.58	5,000,000	6.16

Table 4: Ratio between games (demonstrating constant increase)

The ratio stays fairly constant around 5.756 (mean). This suggests that the relationship between the two functions is only in the overall slope of the function, and will remain about the same as n increases. Likewise, we should be able to determine the ratio between our predicted values at n=10,000,000 and the actual. The ratio between our predicted values is 6.16 (7% error compared to average ratio), which is in the interval of the ratios computed from our samples. The ratio for the actual run-times at n=10,000,000 is 6.12, again within our interval above and suggesting the ratio will continue and the graphs will remain linear as n increases.



Graph 3: Comparing the two games

6 Additional Thoughts and Analysis

6.1 The linearity claim

Throughout this paper, the claim that the algorithms run in an almost linear time in practice has been based on the fact that $m << n$ in most cases, where $m = |S|$. What happens when $m \rightarrow n$ or $m > n$?

If $m > n$ than there must be elements in S that are greater than n. By the design of the algorithm, these elements are not used; they are ignored by line 20 of the algorithm in 4.2. Therefore we look at if $m = O(n)$, then some interesting things begin to happen. First, m has a big affect on the asymptotic complexity of the algorithm, causing it to run in coordination with that of a third order polynomial. However, as n and m increase, we predict the dynamic element of the algorithm will help the complexity and the algorithm will eventually 'level off' or more accurately, the derivative of the function will become constant, leaving a linear based algorithm.

6.2 More experimental analysis

A modified version of the application (appendix B.3) for the sampling in section 5 was used to conduct test with $m = O(n) = n/2$ and the non-repeated random values of $s \in S$ to be in the range from [0, n]. The sampling took the average of 50 executions resulting in some interesting results.

n, m_L & m_R	run-time	ratio
500	0.28540	
1500	7.99760	28.02
2500	26.99145	3.37
3500	>320.000	---
4500	---	--

Table 5: Run-times with *m* and *n* (demonstrating non-exponential increase)

The above table demonstrates the run times encountered and the ratio between the previous and current run-times. As the table demonstrates, the run-times appear to becoming exponentially worse. In fact, it is computationally infeasible to populate the table further with the limited resources and time allocated for this research. At n=3500, and therefore m_L and m_R = 1250, the program executed for over 4.5 hours and was still not complete. This suggests an estimate of one-execution of the algorithm to be over 320 seconds. Optimistically if the sampling at n=3500 finished at 320 seconds, and the run-time became linear after that with a slope of 2 (we know it would be much worse), it could take upwards of a day to sample n=5500. This is beyond the scope of this paper, and therefore no additional experimental analysis was conducted. The extremely bad results do suggest the polynomial of order 3 (or exponential) growth as predicted. Given adequate time and computer resources (higher performance machine than I have available), in the course of a day or two, analysis of samples can be conducted and I predict it will show a 'leveling off' of the exponential looking graph to a linear-time as n increases.

Appendix A – Selected Proofs

A.1 Proof to Repeating Phenomena in Section 3.3

Definition: A position n is safe if, whenever there are n stones remaining on the table, regardless of the previous move played, there is a winning strategy for the second player.

Theorem: Given the set of moves $S = \{1, 2, 3, 4, 5\}$, with s being an element of S, the safe positions will be $[0]_{13}$ and $[7]_{13}$, that is, 0mod13 and 7mod13.

Proof: We start by finding the first few safe positions by constructing a matrix with $|S|$ rows and $(n+1)$ columns, labeled 0 to n and the rows labeled 1, 2, ..., 5. The rows represent the previous move while the columns represent the current number of stones remaining. A position is safe in this model if the entire column is populated with 2's. By default, the n=0 column is filled with all 2's, since no moves exist for the first player. At row 1, column 1, we can only move 2, 3, 4, or 5 stones since the previous move was a 1, so (1, 1) gets populated with a 2. At row 2, column 1 though, we can remove 1 stone to n=0, which would be a player 1 victory, so (2, 1) is a 1. We continue for the remaining rows in column 1.

	Stones remaining (n)		
$s \in S$	0	1	2
1	2	2	-
2	2	1	-
3	2	1	-
4	2	1	-
5	2	1	-

We can continue populating the matrix up to n = 7. At the first row of n = 7, we can only move to elements on the matrix that are currently filled with a 1 (shaded), therefore making (1, 7) a 2. Likewise with row 2, 3, 4, and 5 of column 7, therefore making 7 a safe position.

	Stones Remaining (n)							
$s \in S$	0	1	2	3	4	5	6	7
1	2	2	1	1	1	1	1	2
2	2	1	1	1	1	1	1	2
3	2	1	1	2	1	1	2	2
4	2	1	1	1	2	1	1	2
5	2	1	1	1	1	2	1	2

We notice that the results of any given column, in the above case n=7, depends on the previous 5 columns. We can continue to populate the matrix, with 2 and 1's indicating safe positions and non-safe positions, respectively, up to n=26.

	Stones Remaining (n)																			
$s \in S$	0	1	…	7	8	9	10	11	12	13	14	…	20	21	22	23	24	25	26	27
1	2	2	…	2	1	1	1	1	1	2	2	…	2	2	1	1	1	1	2	2
2	2	1	…	2	1	1	1	1	1	2	1	…	2	1	1	1	1	1	2	1
3	2	1	…	2	1	1	1	1	1	2	1	…	2	1	1	1	1	1	2	1
4	2	1	…	2	1	1	1	2	1	2	1	…	2	1	1	1	2	1	2	1
5	2	1	…	2	1	1	1	1	2	2	1	…	2	1	1	1	1	2	2	1

We've already shown that the determination of a given position to be safe or not depends on the previous 5 columns in the matrix. In the above matrix we have found a repeat, columns n=9…13 and n=22…26, likewise it follows that n=27 matches n=14, and n=28 matches n=15, and so on. From the pumping lemma |1| we know this pattern will continue to repeat as we expand the matrix.
∴The winning positions will continue to occur in the pattern at $[0]_{13}$ and $[7]_{13}$. ■

Generalized: This proof can be generalized using more abstract techniques. You are given a finite set of legal moves S, by the well-ordering principal there exists a maximum element in that set. We know that the decision of whether a position n is safe or not is dependent upon the previous x positions, where x is that maximum element. Since S is finite, there is only a finite possibility of permutations in our matrix, if we keep extending n we will eventually pass the number of permutations and there will have to be a repeating sequence, once we find this sequence we can quickly predict whether the next position is safe or not, and the position after and after and after, and so on. This again is a specific case of the pumping lemma |1|. The pattern of safe positions must repeat in all cases. ■

A.2 Proof of repeating in the asymmetric game

Theorem: No matter which player goes first, if 10 or more stones are on the table, Lillian should win the game, given the set of moves, S_L = {2, 5, 9} and S_R = {3, 4, 8} for Lillian and Rekha respectively.

Proof: Recall from A.2 that the pattern repeats in the symmetric game. The asymmetric game is no different and the algorithm works the same except there are two matrices. We must also factor in which player goes first in the game to correctly decide. By constructing the matrices for the two players below in the two different games (based on who goes first) we find a repeat.

Lillian going first

A_L	Stones Remaining (n)															
S_L	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	2	2	2	1	1	1	2	2	1	1	2	2	2	2	2	2
5	2	2	2	-	-	1	2	-	-	-	-	-	-	-	-	-
9	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-

A_R	Stones Remaining (n)															
S_R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Rekha going first

A_L	Stones Remaining (n)															
S_L	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	2	2	2	1	1	1	2	2	1	1	2	2	2	2	2	2
5	2	2	2	-	-	1	2	2	-	-	2	2	2	2	2	2
9	2	2	2	-	-	-	2	2	-	-	2	2	2	2	2	2

A_R	Stones Remaining (n)															
S_R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	-	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1

We populate the matrices following the asymmetric algorithm given in section 4.2. In both games, Rekha's matrix is primarily populated with 1's, indicating non-safe positions. In Lillian's matrix when she goes first, we see a not-to-filled matrix, but what appears to be a row of 2's forming. This is because Lillian would win these games, but is the first player to go. The matrix that gives the most information is Lillian's matrix when Rekha goes first, it is populated with 2's from n=10 and higher, it will in fact repeat based on the same principles and using the pumping lemma |1|. Therefore Lillian will win the game is 10 or more stones are played, when using the given sets. ■

Appendix B – Selected Source Code

B.1 Implementation of the Asymmetric Algorithm (4.2)

Asym.h

```
 1  #ifndef _ASYM_H_
 2  #define _ASYM_H_
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  typedef unsigned char uchar;
 6  int Find_Safe_Positions(const int, const int*, const int, const int*, const int);
 7  uchar Is_Safe_Aux(const int, uchar**, const int*, const int, uchar**, const int*, const int);
 8  uchar Is_Safe(const int, uchar**, const int*, const int, uchar**, const int*, const int, const int);
 9  #endif
```

Asym.c

```
 1  /*
 2   * Thomas Fisher
 3   * CMSC441 - Dr Tad White - Fall 03
 4   *
 5   * C-File with the asymetric algorithm in it.
 6   * This is the implementation of the algorithm described
 7   * in section 4.2 of the report.
 8   *
 9   */
10
11  #include "asym.h"
12
13  int Find_Safe_Positions(const int n, const int Sl[], const int ml, const int Sr[], const int mr)
14  {
15     uchar **Al, **Ar;
16     uchar *x;
17     int i, j;
18
19     x = malloc((n+1)*sizeof(uchar) );
20     if(x==NULL)
21     {
22        perror("No memory for x\n");
23        return 22;
24     }
25     for(i=0; i<=n; i++)
26        x[i] = 0;
27     Al = (uchar**)malloc(ml*sizeof(int) );
28     if(Al==NULL)
29     {
30        perror("Memory for Al error\n");
31        return 432;
32     }
33     for(i=0; i<ml; i++)
34     {
35        Al[i] = (uchar*)malloc((n+1)*sizeof(uchar));
36        if(Al[i]==NULL)
37        {
38           perror("Memory error for Al\n");
39           return 23;
40        }
41     }
42     for(i=0; i<ml; i++)
```

```
43          for(j=0; j<=n; j++)
44             Al[i][j] = 0;
45       Ar = (uchar**)malloc(mr*sizeof(int) );
46       if(Ar==NULL)
47       {
48          perror("Memory Error for Ar\n");
49          return 765;
50       }
51       for(i=0; i<mr; i++)
52       {
53          Ar[i] = (uchar*)malloc((n+1)*sizeof(uchar) );
54          if(Ar[i] == NULL)
55          {
56             perror("Memory error for Ar\n");
57             return 84;
58          }
59       }
60       for(i=0; i<mr; i++)
61          for(j=0; j<=n; j++)
62             Ar[i][j] = 0;
63       for(i=0; i<=n; i++)
64          x[i] = Is_Safe_Aux(i, Al, Sl, ml, Ar, Sr, mr);
65
66       if(n<36)
67       {
68          printf("\nIf Lillian goes first.\n");
69          printf("-----------------------\n");
70          for(i=n; i>=0; i--)
71             printf("%3i",i);
72          printf("\n");
73          for(i=0; i<=n; i++)
74             printf("---");
75          printf("\n");
76          for(i=n; i>=0; i--)
77             printf("%3i", x[i]);
78          printf("\n");
79       }
80
81       /* Reset the memory to check the other way */
82       for(i=0; i<ml; i++)
83          for(j=0; j<=n; j++)
84             Al[i][j] = 0x00;
85       for(i=0; i<mr; i++)
86          for(j=0; j<=n; j++)
87             Ar[i][j] = 0x00;
88
89       /* Do with Rekha going first! */
90       for(i=0; i<=n; i++)
91          x[i] = Is_Safe_Aux(i, Ar, Sr, mr, Al, Sl, ml);
92       if(n<36)
93       {
94          printf("\nIf Rekha goes first.\n");
95          printf("--------------------\n");
96          for(i=n; i>=0; i--)
97             printf("%3i",i);
98          printf("\n");
99          for(i=0; i<=n; i++)
100            printf("---");
101         printf("\n");
102         for(i=n; i>=0; i--)
103            printf("%3i", x[i]);
104         printf("\n");
105      }
106
107      for(i=0; i<ml; i++)
108         free(Al[i]);
109      free(Al);
110      for(i=0; i<mr; i++)
111         free(Ar[i]);
112      free(Ar);
113      free(x);
114
115      return 0x00;
116   }
117
118   uchar Is_Safe_Aux(const int n, uchar** A1, const int *S1, const int m1, uchar** A2, const int *S2, const int m2)
119   {
120      int i;
121      uchar x;
122
123      for(i=0; i<m2; i++)
124      {
125         x=Is_Safe(n, A1, S1, m1, A2, S2, m2, i);
126         if(x==1)
127            return 1;
128         else
129            ;
130      }
131      return 2;
132   }
133
134   uchar Is_Safe(const int n, uchar** A1, const int *S1, const int m1,
135                 uchar** A2, const int *S2, const int m2, const int move)
136   {
137      int i;
```

```
138     uchar x;
139
140     if(A2[move][n]!=0)
141        return A2[move][n];
142     if(n==0)
143     {
144        A2[move][n] = 2;
145        return 2;
146     }
147     for(i=0; i<m1; i++)
148     {
149        if((signed)(n-S1[i]) >= 0)
150        {
151           x = Is_Safe(n-S1[i], A2, S2, m2, A1, S1, m1, i);
152           if(x==2)
153           {
154              A2[move][n] = 1;
155              return 1;
156           }
157        }
158     }
159     A2[move][n] = 2;
160     return 2;
161  }
```

B.2 Main Function for Statistical Testing (called in section 5)

```
 1  /*
 2   * A general program to simulate the asymmetric game.
 3   * This meets requirement 7 of the project, and
 4   * uses the algorithm in 4.2 of the project
 5   *
 6   * Thomas Fisher
 7   * Dec. 03, 2003
 8   *
 9   * CMSC441.  Dr. Tad White.
10   *
11   */
12
13  #include<stdlib.h>
14  #include<stdio.h>
15  #include<time.h>
16
17  #include "asym.h"
18
19  int main(void)
20  {
21     int n, ml, mr, i;
22     int *Sl, *Sr;
23     clock_t start, stop, avg;
24     double timed;
25
26     printf("What is the number of moves for Lillian?\n");
27     scanf("%i", &ml);
28
29     Sl = (int*)malloc(ml*sizeof(int) );
30     if(Sl == NULL)
31     {
32        perror("Memory error for Sl\n");
33        return 99;
34     }
35     for(i=0; i<ml; i++)
36     {
37        printf("Enter Move %i: ", i+1);
38        scanf("%i", &Sl[i]);
39     }
40
41     printf("What is the number of moves for Rekha?\n");
42     scanf("%i", &mr);
43
44     Sr = (int*)malloc(mr*sizeof(int) );
45     if(Sr == NULL)
46     {
47        perror("Memory error for Sr\n");
48        return 23;
49     }
50     for(i=0; i<mr; i++)
51     {
52        printf("Enter move %i: ", i+1);
53        scanf("%i", &Sr[i]);
54     }
55
56  /*   printf("What is the value of \'n\' in this simulation?\n");
57     scanf("%i", &n);*/
58  for(n=500000; n<=5000000; n+=500000)
59  {
60     n = 10000000;
61     start = clock();
62     for(i=0; i<50; i++)
63        Find_Safe_Positions(n, Sl, ml, Sr, mr);
64     stop = clock();
65     avg = (stop - start)/i;
```

```
66     timed = (double)avg/(double)CLOCKS_PER_SEC;
67
68     printf("\nPerformed n=%d %i times with an average time of %0.5lf seconds.\n", n, i, timed);
69  }
70     free(Sr);
71     free(Sl);
72
73     return 0;
74  }
```

B.3 Modified Version for Section 6 experimental analysis

```
 1  /*
 2   * This is a modified version of the part 7 of the project.  This creates the
 3   * size of the Sets of legal moves to be O(n) or specifically n/2.
 4   * Then a random set of values is put into each set.
 5   * Timing analysis is done.
 6   *
 7   * Thomas Fisher
 8   * Dec. 03, 2003
 9   *
10   * CMSC441.  Dr. Tad White.
11   *
12   */
13
14  #include<stdlib.h>
15  #include<stdio.h>
16  #include<time.h>
17
18  #include "asym.h"
19
20  int main(void)
21  {
22     int n, ml, mr, i, x;
23     int *Sl, *Sr, *tmp;
24     clock_t start, stop, avg;
25     double timed;
26
27     srand(time(NULL) );
28
29  for(n=500; n<=10500; n+=1000)
30  {
31     ml = mr = n/2;
32     Sl = malloc(ml*sizeof(int) );
33     if(Sl == NULL)
34        return 34;
35     Sr = malloc(mr*sizeof(int) );
36     if(Sr == NULL)
37        return 28;
38     tmp = malloc((n+1)*sizeof(int) );
39     if(tmp == NULL)
40        return 33;
41     for(i=0; i<=n; i++)
42        tmp[i] = 0;
43     i=0;
44     while(i<ml)
45     {           /* this is to prevent repeated moves */
46        x = (int)rand()%n + 1; /* we don't want to have a move of 0 */
47        if(!tmp[x])
48        {
49           tmp[x] = 1;
50           Sl[i] = x;
51           i++;
52        }
53     }
54     for(i=0; i<=n; i++)
55        tmp[i] = 0;
56     i=0;
57     while(i<mr)
58     {           /* this is to prevent repeated moves */
59        x = (int)rand()%n + 1;
60        if(!tmp[x])
61        {
62           tmp[x] = 1;
63           Sr[i] = x;
64           i++;
65        }
66     }
67     printf("\nn = %d\nml = %d\nmr = %d\nAbout to start\n", n, ml, mr);
68     start = clock();
69     for(i=0; i<50; i++)
70        Find_Safe_Positions(n, Sl, ml, Sr, mr);
71     stop = clock();
72     avg = (stop - start)/i;
73     timed = (double)avg/(double)CLOCKS_PER_SEC;
74
75     printf("\nPerformed n=%d %i times with an average time of %0.5lf seconds.\n", n, i, timed);
76
77     free(tmp);
78     free(Sr);
79     free(Sl);
80  }
```

```
81      return 0;
82  }
```

Appendix C – Acknowledgements

I would like to acknowledge the following people for their assistant in the completion of this research project and consequential paper.

Dr. Tad White – For spending several hours after class every Tuesday and Thursday answering and giving hints to our tedious questions!

Scott Martucci & **Matthew Tinnirella** – For reviewing the assigned project, brainstorming together and reviewing the final technical report

Kevin Arber – For a review of the rough draft of this technical report

Scott Moser – For guidance on how to accurately find the run-time of an executing program

Dr. Charles Toll – For guidance and suggestions on the repeating phenomenon proof (A.1)

Appendix D – References

|1| The Pumping Lemma for Regular Languages, page 126. Introduction to Automata Theory, Languages and Computation, 2nd Edition. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Addison-Wesley Publishing © 2001.