

Assignment 3

ES22E Software Development

0013679 – TJ Kennaugh

Group 5:

Mr TJ Kennaugh

Mr TMP Veeran

Mr S Kamarzaman

Mr EJ White

School of Engineering, University of Warwick

25/01/02

Specification

- The program should simulate a snakes and ladders game based on the board given.
- The dice will be in the form of a random/pseudo-random number generator.
- The game will finish when play passes 100; it does not need to land on 100.
- The program should determine:
 1. The minimum number of throws of the dice that are needed and a typical path in this circumstance.
 2. The most common number of throws.
 3. The average number of throws.
 4. The most popular square other than 100.
 5. The least popular square on which to land and the probability of doing so.
 6. The mean real time to simulate one game.
- The code must run on Windows NT (Microsoft C++/C) and UNIX (gcc).
- The random number generator must be tested to confirm suitability.
- At least 10,000 games must be simulated to provide meaningful results.

Design and Implementation

The first section to consider is the random number generator. C has a built in random number generator in the math.h library called rand(). For easy of use this is included in the tools.h library and is imported through this. A random long can be generated from this and it can be scaled to 0-5 by finding the remainder when divided by 6 incrementing this will give a number between 1-6. Enclosed in a function this will give:

```
short dice(void) {
    long num;
    num = rand();
    num %= 6;
    return (short)(++num);
}
```

The function to seed this random generator must be placed outside this function to prevent it being seeded every time, therefore giving the same number. The seed function is given below:

```
srand( (unsigned)(time(NULL)) );
```

This will be placed in the main function.

A function is written to test the dice by keeping a tally of how many times each number occurs:

```
void TestDice(long numRolls) {
    long i;
    long countdice[6];
    for(i=0;i<6;i++) countdice[i] = 0;
    printf("----- Dice Test -----\n");
    printf("Performing %i rolls...",numRolls);
    long roll;
```

```

    long numones = 0;
    double probone;
    for(i=0;i<numRolls;i++) {
        roll = dice();
        countdice[roll-1] += 1;
    }
    probone = (double)numones/(double)numRolls;
    printf("Done.\nProbabilities. Should be about 0.16666\n",probone);
    for(i=0;i<6;i++) {
        printf("%i ",countdice[i]);
    }

    printf("\n-- End of Dice Test --\n");
}

```

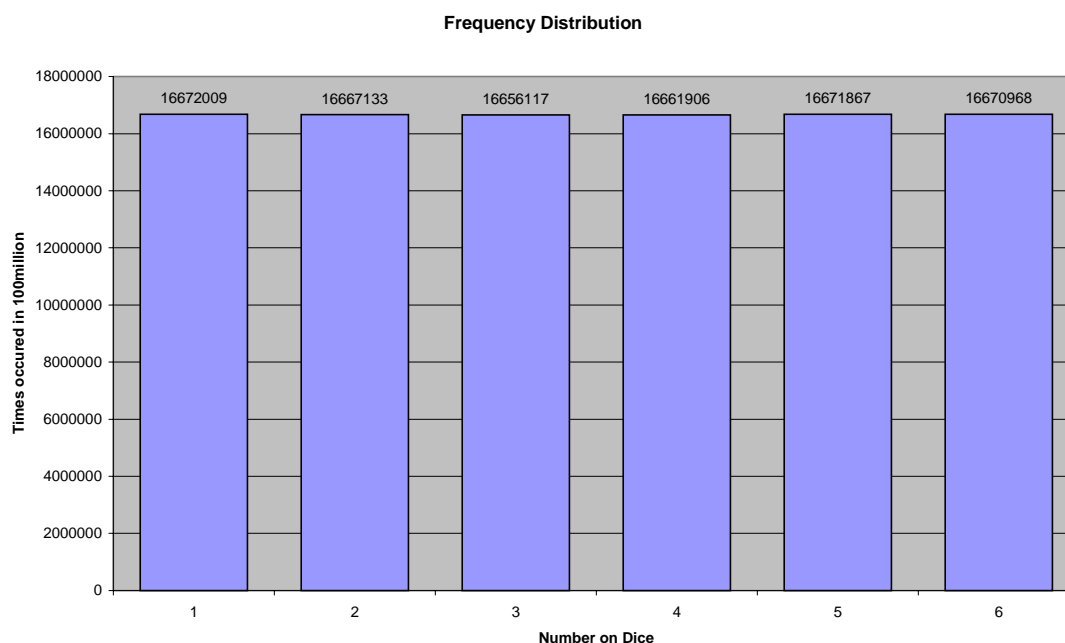
A sample output from this function when called with the statement `TestDice(100000000);` is:

```

----- Dice Test -----
Performing 100000000 rolls...Done.
Probabilities. Should be about 0.16666
16672009 16667133 16656117 16661906 16671867 16670968
-- End of Dice Test --

```

A frequency distribution can be produced:



This shows that the distribution is sufficiently uniform for an application such as this. The dice test was run a few more times and it was noted that the most/least popular number varied each time.

With a random number generator the main simulation algorithm can be developed. There are many possible ways of implementing the game. If statements are the most simplistic, but will be slow. Implementations using arrays of positions are also possible but again this will be slow and complicated. The simplest way found consists of an array representing the whole board (100+1) squares. The array is populated with zeros except when there is a snake or ladder when the location the snake or ladder leads to is given. The position pointer can then step through the array according to the dice rolls. When a non-zero number is found the counter will be set to the new position. The simulation will run until the counter moves past 100. It would also be possible to put the next position in the array, eliminating the if statement, but the array

is easier to understand and change with zeros. An initial program is written to test the algorithm by running a single game. This is shown below:

```
#include "tools.h"

int dice(void);
const char rules[] = {0,
38,0,0,14,0,0,0,0,31,0,
0,0,0,0,6,0,0,0,0,
42,0,0,0,0,0,84,0,0,
0,0,0,0,44,0,0,0,0,
0,0,0,0,0,26,0,11,0,
67,0,0,0,0,53,0,0,0,0,
0,19,0,60,0,0,0,0,0,0,
91,0,0,0,0,0,0,0,100,
0,0,0,0,0,0,24,0,0,0,
0,0,93,0,75,0,0,78,0,0,0,0,0,0,0 };

int main(void) {
    int roll;
    int pos = 0;
    srand( (unsigned)(time(NULL)) );
    while (pos < 100) {
        roll = dice();
        printf("%i!",roll);
        if ( (roll+pos) <= 100 ) {pos += roll;}
        printf("%i, ",pos);
        if (rules[pos] != 0) { pos = rules[pos]; }
    }
    printf("Finished\n");
    return 0;
}

int dice(void) {
    long num;
    num = rand();
    num %= 6;
    return ++num;
}
```

A typical output from this program is shown below:

```
1!1, 3!41, 6!47, 1!27, 1!28, 6!90, 4!94, 6!100, Finished
```

or:

```
6!6, 3!9, 2!33, 5!38, 5!43, 6!49, 5!16, 2!8, 5!13, 5!18, 1!19, 4!23, 5!28, 1!85, 5!90,
6!96, 6!96, 2!98, 2!80, Finished
```

The output shows the roll of the dice followed by ‘!’ followed by the position after that throw. By comparing the output with the board diagram the operation of the game can be confirmed. The algorithm is shown to be very efficient. A constant array of chars is used since the range of values can only be 0-100. The array is global since it will not be worth creating/destroying it every time a game is simulated.

The algorithm can now be implemented so that statistics can be collected. The simulation of one game can be enclosed in one function to simplify the layout. This function will need to return the number of rolls the game takes. Other data will be needed to be collected, but this can be dealt with later. The basic function is shown below:

```
long CalculateGame(void) {
    short pos = 0; // Current position
    long numRolls = 0; // Number of rolls
    while (pos < 100) { // While counter is on board
        pos += dice(); // Roll dice to find new position
        //if snake or ladder defined in array set position
        if (rules[pos] != 0) { pos = rules[pos];}
        numRolls++; //count rolls
    }
    return numRolls; // Return number of rolls for game
}
```

A simple for loop can be used in the main function to simulate a given number of games. The number of games can be defined as NUM_GAMES at the top of the program. The minimum can be found within the for loop and the most common can be found by keeping a tally of how many rolls each game takes in an array. The for loop is shown below:

```

long i; // Loop index
long game; // Stores number of rolls after each game
long min,mode;
long max = 0;
long sum = 0;
double average
long countRolls[ROLL_COUNT]
for (i=0;i<ROLL_COUNT;i++) countRolls[i] = 0; //Zeros array
for (i=0;i<NUM_GAMES;i++) {
    game = CalculateGame(); // Calls calculate game which returns number of rolls
    if (game < min) min = game; //Finds minimum number of rolls
    sum += game; // Calculates total number of rolls (used for average)
    if (game < 200) countRolls[game] += 1; //Keeps tally of roll frequency
}

```

Another for loop can then be used to process the array and give the most common number of throws and find the average:

```

for (i=0;i<ROLL_COUNT;i++) { // Find most common no of throws
    if (countRolls[i] > max) {
        max = countRolls[i];
        mode = i;
    }
}
average = (double)sum/NUM_GAMES; //Calculate average

```

And print the results:

```

printf("Average number of throws is %g.\n",average);
printf("Minimum number of throws is %i.\n",min);
printf("Most Common number of throws is %i.\n",mode);

```

The most and least popular square must now be determined.

A suitable way of doing this is to keep an array of 100 elements and increment the required element each time the corresponding square is landed on. A global array is declared to allow multi-function access:

```

long squareCount[SQUARE_COUNT];
. . . . .
for (i=0;i<SQUARE_COUNT;i++) squareCount[i] = 0; //Zeros array

```

The code to keep the square count is placed in the calculategame() function:

```

while (pos < 100) { // While counter is on board
    pos += dice(); // Roll dice to find new position
    squareCount[pos] += 1; // Increment required square count
    //if snake or ladder defined in array set position and increment square count
    if (rules[pos] != 0) { pos = rules[pos]; squareCount[pos] += 1;}
    numRolls++; //count rolls
}

```

Another for loop is used to find and note to most and least popular square (and the total – used for the probability):

```

for (i=1;i<100;i++) { //Find most and least popular square
    if (squareCount[i] > maxs) {
        maxs = squareCount[i];
        popularsquare = i;
    }
    if (squareCount[i] < mins) {
        mins = squareCount[i];
        unpopularsquare = i;
    }
    squaresum += squareCount[i]; //Used for probability
}

```

The probability is calculated and the results displayed:

```

probleast = (double)(squareCount[unpopularsquare])/((double)(squaresum));
. . . . .
printf("Most popular square is %i.\nLeast popular is %i with a probability of
landing of %g.\n",popularsquare,unpopularsquare,probleast);

```

The average time must now be determined.

Since the accuracy of the system clock is about 50us the best way of calculating the average time would be to simply take the time for simulating all the games and divide it by the number of games. The code for this is shown below:

```
double tv1, tv2;
double etime;
tv1 = clock(); //start clock
//For loop simulating games
tv2 = clock(); //stop clock
// Calculate average time in us
etime = (double)(tv2 - tv1)/(double)(CLOCKS_PER_SEC)/(double)NUM_GAMES*1e6;
//rest of code
printf("Average time per game is %gus.\n", etime); // Print average time
```

A path is required for the minimum number of throws. The easiest way of keeping a path is to store the path for every game simulated. An array is declared to keep the path:

```
short moves[MOVE_COUNT];
```

The moves are stored in this array during the calculategame() function:

```
while (pos < 100) { // While counter is on board
    pos += dice(); // Roll dice to find new position
    squareCount[pos] += 1; // Increment required square count
    //if snake or ladder defined in array set position and increment square count
    if (rules[pos] != 0) { pos = rules[pos]; squareCount[pos] += 1;}
    // Stores moves for first 100
    if (numRolls < 100 ) { moves[numRolls] = pos; }
    numRolls++; //count rolls
}
```

To save the trouble of storing this in the main simulation loop a path is found by iterating using the following code:

```
while (game != min) game = CalculateGame(); //Iterate to find require moves
```

The path must then be printed, the following function does this when called:

```
void PrintMoves(int num_moves) {
    printf("0,"); // Start with 0
    long i = 0; // Array index
    short nextmove = 0;
    while (i < num_moves) {
        printf("%i,", moves[i]);
        nextmove = moves[++i];
    }
    printf("\b.\n");
}
```

The revised output statement is shown below:

```
printf("Minimum number of throws is %i.\nTypical Path:\n", min);
while (game != min) game = CalculateGame(); //Iterate to find require moves
PrintMoves(min); //Print moves
```

The separate sections are combined to give the full program.

Areas of responsibility:

Simulation Algorithm - Mr TJ Kennaugh

Dice & Testing - Mr EJ White

Most/Least Common number of throws - Mr TMP Veeran

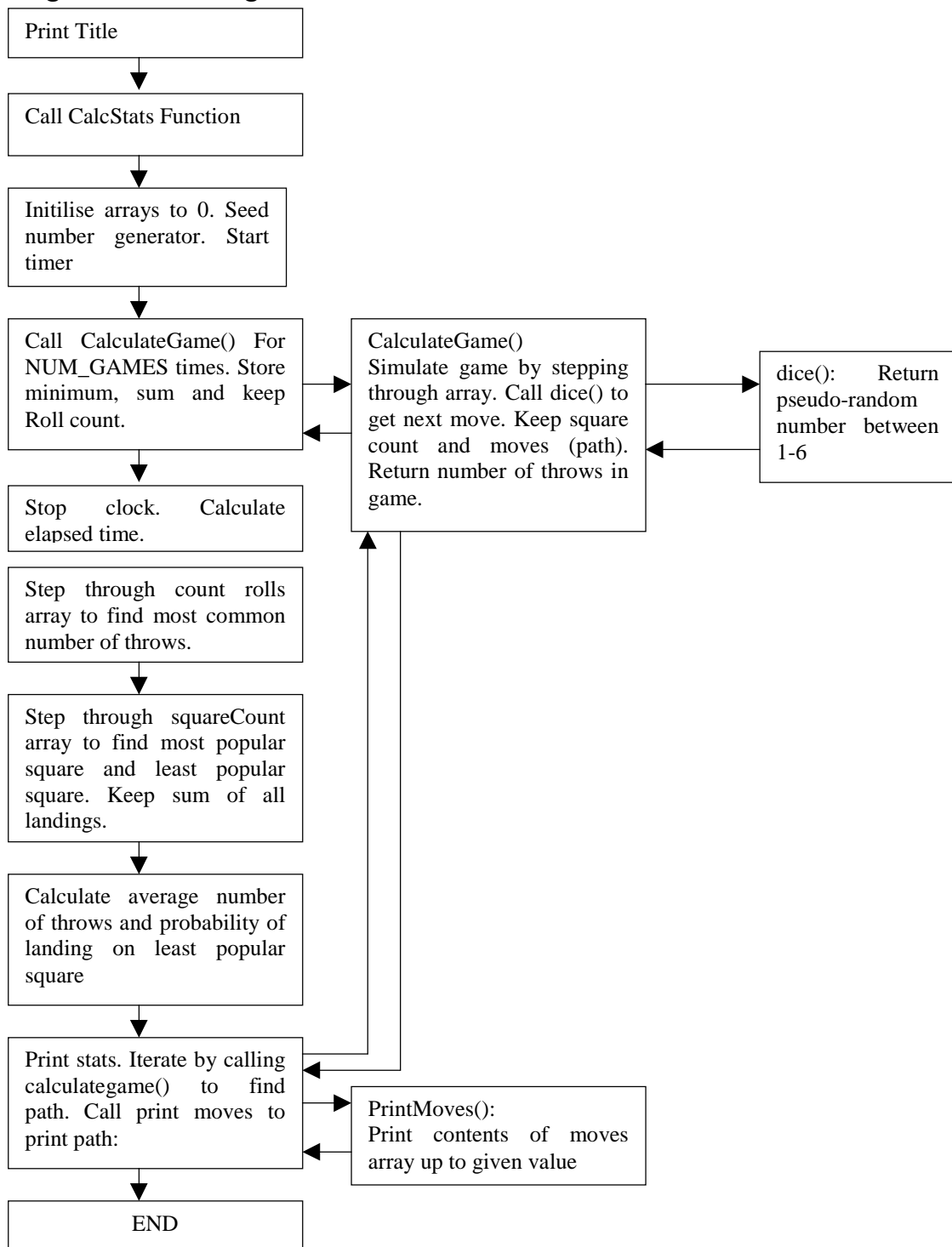
Most/Least Popular Square - Mr TJ Kennaugh

Timer - Mr S Kamarzaman

Move Logging - Mr S Kamarzaman

Debugging/Testing – Joint (Search & Correct)

Program Block Diagram:



Program Listing

```

// Snakes and Ladders Simulator
// Group 5
// Version 1.0
#include "tools.h"

#define NUM_GAMES 1000000 //Number of games to simulate
#define ROLL_COUNT 200 //Size of roll count array
#define SQUARE_COUNT 107 //Size of square count array
#define MOVE_COUNT 120 //Size of move count array

//Function declarations
short dice(void);
long CalculateGame(void);
void CalcStats(void);
void TestDice(long numRolls);
void PrintMoves(int num_moves);

//Global array of constants defining layout of snakes and ladders
const char rules[] = {0,
38,0,0,14,0,0,0,0,31,0,
0,0,0,0,6,0,0,0,0,
42,0,0,0,0,0,0,84,0,0,
0,0,0,0,44,0,0,0,0,
0,0,0,0,0,26,0,11,0,
67,0,0,0,0,53,0,0,0,0,
0,19,0,60,0,0,0,0,0,0,
91,0,0,0,0,0,0,0,100,
0,0,0,0,0,24,0,0,0,
0,0,73,0,75,0,0,78,0,0,0,0,0,0,0,0,0 };

short moves[MOVE_COUNT]; // Array for keeping path taken
long squareCount[SQUARE_COUNT]; //Array for keeping square landings

long main(void) {
    // Prints Title
    printf("\nAssignment 3 - Snakes and Ladders Simulator\n");
    printf("-----\n\n");

    CalcStats(); // Calls main processing routine

// TestDice(100000000);

    printf("\nNormal Program Termination\n");
    return 0;
}

void CalcStats(void) {

    long i; // Loop index
    long game; // Stores number of rolls after each game
    long min,mode; // Minimum/most common square
    long max = 0; // Stores maximum number of throws
    long sum = 0; // Stores minimum number of throws
    long countRolls[ROLL_COUNT]; // Stores frequency of number of rolls (for most
common)
    long mins = NUM_GAMES; //Used for calculating least popular square
    long maxs = 0; //Used for calculating most popular
square
    long popularsquare = 0;// Stores most popular square
    long unpopularsquare = INT_MAX;// Stores least popular square
    long squaresum = 0; // Total number of landings
    double average,probleast;// Average number of throws/probability on landing on
least popular
    double tv1, tv2; //Stores start and end time
    double etime; // Elapsed time
    srand( (unsigned)(time(NULL)) ); // Seeds random number generator using time
    printf("Simulating %i games.\nCalculating ...",NUM_GAMES); // Prints message
    for (i=0;i<ROLL_COUNT;i++) countRolls[i] = 0; //Zeros array
    for (i=0;i<SQUARE_COUNT;i++) squareCount[i] = 0; //Zeros array
    min = 200; // Initialises minimum number of throws to high number
    tv1 = clock(); //Starts clock
    for (i=0;i<NUM_GAMES;i++) {
        game = CalculateGame(); // Calls calculat game which returns number of rolls

```



```

        if (game < min) min = game; //Finds minimum number of rolls
        sum += game; // Calculates total number of rolls (used for average)
        if (game < 200) countRolls[game] += 1; //Keeps tally of roll frequency
    }
    tv2 = clock(); //Stops clock
    etime = (double)(tv2 - tv1)/(double)(CLOCKS_PER_SEC)/(double)NUM_GAMES*1e6; //
Calculate average time in us
    printf(" Finished\n");
    for (i=0;i<ROLL_COUNT;i++) { // Find most common no of throws
        if (countRolls[i] > max) {
            max = countRolls[i];
            mode = i;
        }
    }
    for (i=1;i<100;i++) { //Find most and least popular square
        if (squareCount[i] > maxs) {
            maxs = squareCount[i];
            popularsquare = i;
        }
        if (squareCount[i] < mins) {
            mins = squareCount[i];
            unpopularsquare = i;
        }
        squaresum += squareCount[i]; //Used for probability
    }

    average = (double)sum/NUM_GAMES; //Calculate average
    propleast = (double)(squareCount[unpopularsquare])/(double)(squaresum); //Calculate
probability
    //Print out stats
    printf("Most popular square is %i.\nLeast popular is %i with a probability of
landing of %g.\n",popularsquare,unpopularsquare,propleast);
    printf("Average number of throws is %g.\n",average);
    printf("Average time per game is %gus.\n",etime);
    printf("Minimum number of throws is %i.\nTypical Path:\n",min);
    while (game != min) game = CalculateGame(); //Iterate to find require moves
    PrintMoves(min); //Print moves
    printf("Most Common number of throws is %i.\nTypical Path:\n",mode);
    while (game != mode) game = CalculateGame();//Iterate to find require moves
    PrintMoves(mode); //Print moves
}

long CalculateGame(void) {
    short pos = 0; // Current position
    long numRolls = 0; // Number of rolls
    while (pos < 100) { // While counter is on board
        pos += dice(); // Roll dice to find new position
        squareCount[pos] += 1; // Increment required square count
        //if snake or ladder defined in array set position and increment square count
        if (rules[pos] != 0) { pos = rules[pos]; squareCount[pos] += 1;}
        // Stores moves for first 100
        if (numRolls < 100 ) { moves[numRolls] = pos; }
        numRolls++; //count rolls
    }
    return numRolls; // Return number of rols for game
}

// Dice
short dice(void) {
    long num;
    num = rand(); // Get next random number
    num %= 6; // integer between 0-5
    return (short)(++num); // return number between 1-6
}

void PrintMoves(int num_moves) { // Prints path in moves array (ie last simulated
game)
    printf("0,"); // Start with 0
    long i = 0; // Array index
    short nextmove = 0;
    while (i<num_moves) { // -1 marks end of moves
        printf("%i,",moves[i]);
    }
}

```

```
        nextmove = moves[++i];
    }
    printf("\b.\n");
}

void TestDice(long numRolls) {
    long i;
    long countdice[6];
    for(i=0;i<6;i++) countdice[i] = 0;
    printf("----- Dice Test -----\n");
    printf("Performing %i rolls...",numRolls);
    long roll;
    long numones = 0;
    double probone;
    for(i=0;i<numRolls;i++) {
        roll = dice();
        countdice[roll-1] += 1;
    }
    probone = (double)numones/(double)numRolls;
    printf("Done.\nProbabilities. Should be about 0.16666\n",probone);
    for(i=0;i<6;i++) {
        printf("%i ",countdice[i]);
    }

    printf("\n-- End of Dice Test --\n");
}
```

Testing/Debugging

For dice testing see ‘Design and Implementation’.

Most functions were tested by printing results at each step. This was especially important for the `calculategame()` function. During development the main game simulation algorithm was simulated with a separate program. This is shown below:

```
#include "tools.h"

int dice(void);
char rules[] = {0,
38,0,0,14,0,0,0,0,31,0,
0,0,0,0,6,0,0,0,0,
42,0,0,0,0,0,0,84,0,0,
0,0,0,0,44,0,0,0,0,
0,0,0,0,0,26,0,11,0,
67,0,0,0,0,53,0,0,0,0,
0,19,0,60,0,0,0,0,0,0,
91,0,0,0,0,0,0,0,100,
0,0,0,0,0,24,0,0,0,
0,0,73,0,75,0,0,78,0,0,0,0,0,0,0,0 };

int main(void) {
    int roll;
    int pos = 0;
    srand( (unsigned)(time(NULL)) );
    while (pos < 100) {
        roll = dice();
        printf("%i!",roll);
        pos += roll;
        printf("%i, ",pos);
        if (rules[pos] != 0) { pos = rules[pos]; }
    }
    printf("Finished\n");
    return 0;
}

int dice(void) {
    long num;
    num = rand();
    num %= 6;
    return ++num;
}
```

This program simulates one game only. It prints out the dice roll followed by ‘!’ and the position after that roll until simulation is complete. Typical output from this program is shown below:

```
primrose > lst.exe
4!4,5!19,5!24,1!25,4!29,6!35,4!39,4!43,2!45,1!46,6!52,2!54,2!56,2!55,6!61,1!62,4
!23,3!26,1!27,4!31,5!36,1!45,1!46,5!51,6!73,3!76,2!78,3!81,3!84,1!85,3!88,4!92,5
!97,6!103,Finished
primrose > lst.exe
5!5,1!6,2!8,3!11,6!17,2!19,2!21,4!46,2!48,5!53,2!55,1!56,1!54,5!59,3!62,5!24,6!3
0,3!33,4!37,2!39,3!42,3!45,6!51,1!68,6!74,2!76,5!81,5!86,1!87,2!26,3!29,3!32,2!3
4,4!38,5!43,1!44,6!50,1!51,2!69,2!71,2!93,4!77,6!83,3!86,1!87,2!26,1!27,3!30,6!3
6,3!47,6!32,4!36,6!50,4!54,5!59,6!65,2!67,6!73,2!75,2!77,6!83,5!88,3!91,1!92,5!9
7,5!102,Finished
primrose >
```

From the output it is possible to compare the path taken with the board layout, therefore confirming the operation of the simulator is correct.

Other checks involved looking for unlikely results. It was found that sometimes time sometimes was producing a negative number, obviously a wrong answer, and occasionally an access violation was produced. It was difficult to isolate the problem(s) causing this although it seemed likely this was a pointer problem, likely to be caused by array indexes. A constant value for the seed was found that produced an access violation. Print statements were used to find out where the problem was by printing out single characters to represent stages in the program. The problem was

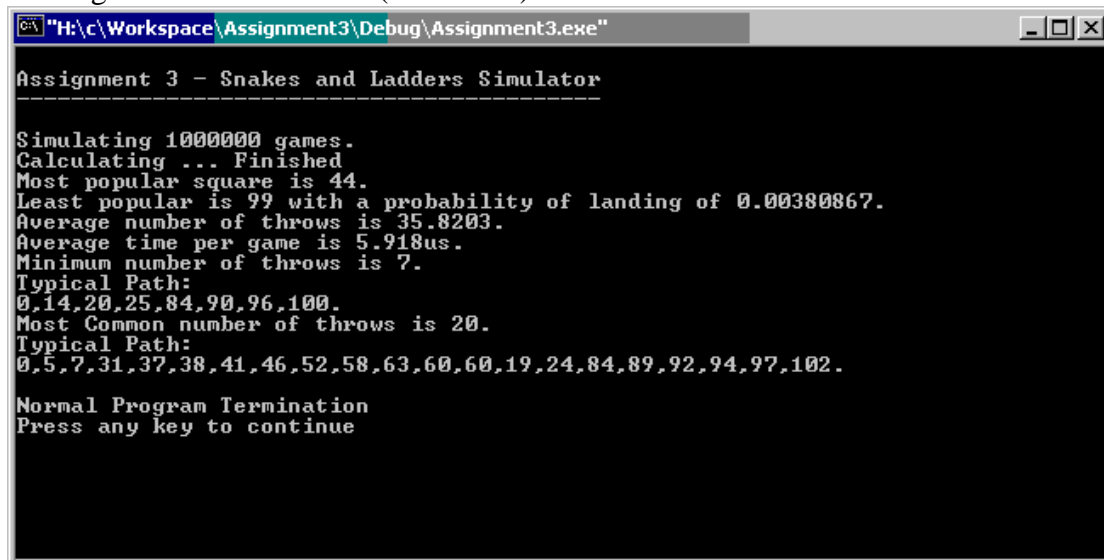
found to be part of the move logging method. The array was sometimes being set to an incorrect location, resulting in the time being overwritten or an access violation. The method was altered to prevent this and both problems were corrected. Because a/multiple path(s) can be printed for any number of throws using the iterative method at the end of the program the program can be tested in location. For instance for 10 rolls it is possible to have the following paths (as well as others):

```
0,38,40,41,44,48,67,70,91,75,100.  
0,2,8,13,18,24,84,88,94,99,103.  
0,5,11,17,22,24,84,88,91,96,100.  
0,14,20,26,27,84,88,92,94,75,100.
```

Results were also compared with other groups to assess similarity of results.

Analysis of Results

The program was written in Microsoft Visual C++/C and the initial development and testing was performed in this environment. An example output from the program running under Windows NT (CADLAB) is shown below:



```
"H:\c\Workspace\Assignment3\Debug\Assignment3.exe"  
Assignment 3 - Snakes and Ladders Simulator  
-----  
Simulating 1000000 games.  
Calculating ... Finished  
Most popular square is 44.  
Least popular is 99 with a probability of landing of 0.00380867.  
Average number of throws is 35.8203.  
Average time per game is 5.918us.  
Minimum number of throws is 7.  
Typical Path:  
0,14,20,25,84,90,96,100.  
Most Common number of throws is 20.  
Typical Path:  
0,5,7,31,37,38,41,46,52,58,63,60,60,19,24,84,89,92,94,97,102.  
Normal Program Termination  
Press any key to continue
```

The same program was run on the ITS machines (also NT) to compare the results:

```

Command Prompt
0,38,43,26,31,33,39,45,67,73,76,82,83,24,84,86,88,94,97,100.
Normal Program Termination
H:\C>"Assignment3"
Assignment 3 - Snakes and Ladders Simulator
-----
Simulating 1000000 games.
Calculating ... Finished
Most popular square is 44.
Least popular is 99 with a probability of landing of 0.00380183.
Average number of throws is 35.8404.
Average time per game is 8.902us.
Minimum number of throws is 7.
Typical Path:
0,38,42,46,67,69,74,100.
Most Common number of throws is 20.
Typical Path:
0,6,7,31,35,44,46,11,17,18,24,84,89,75,79,84,89,92,73,74,100.
Normal Program Termination
H:\C>_

```

It can be seen that the same results are produced except the time for one game is different. This would be expected and the CADLAB PCs are faster than the ITS PCs. The program must also be run and compiled on Unix. The source code ("Assignment3.cpp") was uploaded to the 'primrose' Unix machine into the 'c' directory. The code is then compiled with gcc and run as shown below:

```

C:\WINNT\System32\telnet.exe
Assignment2.c          a.out
Assignment3.c         getlimits.c
primrose > gcc Assignment3.cpp
primrose > mv a.out Assignment3
primrose > Assignment3
Assignment 3 - Snakes and Ladders Simulator
-----
Simulating 1000000 games.
Calculating ... Finished
Most popular square is 44.
Least popular is 99 with a probability of landing of 0.00380321.
Average number of throws is 35.7951.
Average time per game is 30.89us.
Minimum number of throws is 7.
Typical Path:
0,38,42,48,67,72,78,100.
Most Common number of throws is 20.
Typical Path:
0,14,6,7,31,35,39,42,26,30,34,39,43,48,67,70,76,82,88,94,100.
Normal Program Termination
30.91u 0.02s 0:31.27 98.9%
primrose > _

```

The code is a lot slower on Unix taking over three times as long as the NT machines. This is likely to be since primrose is a multi-user computer and there is not as much processing time available.

Other than the time difference the results given by the three different programs are the same showing the program works on NT and Unix machines and will therefore probably work on any machine with an ISO c compiler.

One million games were simulated, which should give good, reliable results. However this took 31 seconds on Unix so it may have been reasonable to use a smaller number. 10,000 should still give statistically reliable results and would take a fraction of the time.

To test reliability and adjustability the program is run with fewer games simulated (100,000):

```

Telnet - primrose
Connect Edit Terminal Help
0,38,39,43,26,84,89,94,96,97,100.

Normal Program Termination
primrose > gcc Assignment3.cpp
aprimrose > a.out

Assignment 3 - Snakes and Ladders Simulator
-----

Simulating 1000000 games.
Calculating ... Finished
Most popular square is 44.
Least popular is 99 with a probability of landing of 0.00383997.
Average number of throws is 35.9867.
Average time per game is 30.9us.
Minimum number of throws is 7.
Typical Path:
0,38,42,46,67,72,75,100.
Most Common number of throws is 20.
Typical Path:
0,6,12,13,17,22,25,26,27,29,31,37,41,44,46,26,84,90,96,99,103.

Normal Program Termination
primrose > █

```

Again, this gives the same results.

The typical path is not always the same since multiple paths are possible and the choice of which path to print is ultimately controlled by the random number generator. Similarly the probability and average number of throws vary slightly, although they are fairly constant.

The program is shown to work as required in the specification. The process of the simulation algorithm has been verified and shown to follow the rules laid down on the board. It calculates all the required information and has been tested with up to one million games on Unix and 10 million on Windows NT. The time to simulate a single game has been shown to be a tiny 6us on a fast PC up to a still small 30us on a slower Unix machine. The results have been shown to be consistent across platforms.

Bibliography

Lecture Notes.

A. Fischer, D. Eggert, S. Ross, *Applied C: An introduction and more*, McGraw-Hill, 2001.

Appendix 1: Assignment 3 Sheet (Attached)

Contains board definition.

The University of Warwick is permitted to keep a copy of this submission, including the program listing, and use it for any educational purposes in the future without time limit.