PH20018 Programming Skills: Coursework 2

Contents

[Discussion of approach 1](#_Toc39789576)

[Question 1 3](#_Toc39789577)

[Q1 i & ii results 3](#_Toc39789578)

[100 measurements of L=10 4](#_Toc39789579)

[Question 2 8](#_Toc39789580)

[Question 3 9](#_Toc39789581)

[Testing 12](#_Toc39789582)

[General 12](#_Toc39789583)

[Q1.c 13](#_Toc39789584)

[Q2.c 14](#_Toc39789585)

[Q3.c 16](#_Toc39789586)

[C Code 19](#_Toc39789587)

[Q1.c 19](#_Toc39789588)

[Q2.c 24](#_Toc39789589)

[Q3.c 28](#_Toc39789590)

# Discussion of approach

To design the program the Skelton programming method was implemented. The skeleton programming method uses a series of headers for each section of code. These headers dictate what each part of the program must accomplish. Each header of code is then developed separately with pseudo inputs. For all 3 problems, the following headers were used:

1. Assigning variables
2. Setting up file to write to
3. Loops for main part of code
   1. Resetting variables
   2. Random number
   3. Move by random number
   4. Check every site is visited and print result
4. END

The benefit of skeleton programming vs flowcharts and pseudocode is that it is much faster to code and much more reliable. Using this method, the first task is to figure out the general headers for each section of the code. Then decide which parts are the most important and start coding those first. When writing the code for a section, pseudo inputs are employed. This prevents errors carrying forward from previous parts of the code. As the overall program gets bigger, compilation time extends. programming sections of the code at a time, prevents long compilation times when debugging.

#include <stdio.h>

int move (int value, int \*position, int L)

{

\*position += value;

if (\*position > L) //if the position goes past L, it moves it to 1

{

\*position -= L;

}

if (\*position < 1) // if it moves below 1, it moves it to L

{

\*position += L;

}

}

int main ()

{

int **L** = 10; //pseudo inputs

int **position** = 1;

printf ("start position: %d\n", position);

move (**-1**, &position, L); //should move to L

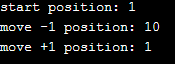
printf ("move -1 position: %d\n", position);

move (**1**, &position, L); //should move back to 1

printf ("move +1 position: %d\n", position);

}

OUTPUT:



Above is an example of the Move function for Q1 and how it was coded. L, positions and the move value are the pseudo inputs (shown in purple). 1 is the starting position and L = 10. The first move is “-1”, so the function should return L as the new position. Then the second move is “+1”, so it should return 1 as the final position. This result is confirmed by print statements.

Once this function is made, the next section is coded independently of the move function. Once all sections are complete, the code is assembled. This leads to fewer bugs and errors and prevents a bug from carrying over to other parts of the code. This is a problem that can occur in top down programming. When using flow charts, sometimes there will be a flaw in logic. This leads to the wrong output, even if coded correctly. This method can prevent this as sections are coded independently. They form building blocks that remain useful even if the overall order is incorrect. Creating functions simplifies the code and helps readability. This method also makes it easier to understand where the bugs are coming from. The code follows a logical order and allows for cleaner code. The biggest benefit over flowcharts and pseudocode, is that coding starts sooner. Bugs are impossible to avoid, the sooner coding begins, the sooner debugging begins.

# Question 1

Q1 i & ii results

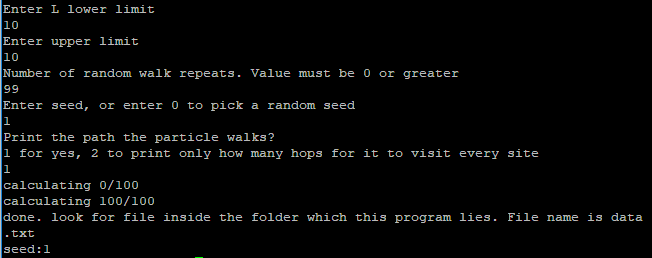
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Seed | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 10 | 10 | 10 |
| Attempt | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| Hop Number | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 10 | 10 | 10 | 10 | 10 | 10 | 2 | 2 | 2 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 2 | 2 | 2 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 4 | 3 | 3 | 3 | 1 | 1 | 1 | 9 | 9 | 9 | 1 | 1 | 1 |
| 5 | 4 | 4 | 4 | 2 | 2 | 2 | 10 | 10 | 10 | 10 | 10 | 10 |
| 6 | 5 | 5 | 5 | 1 | 1 | 1 | 9 | 9 | 9 | 1 | 1 | 1 |
| 7 | 4 | 4 | 4 | 10 | 10 | 10 | 8 | 8 | 8 | 2 | 2 | 2 |
| 8 | 3 | 3 | 3 | 1 | 1 | 1 | 7 | 7 | 7 | 3 | 3 | 3 |
| 9 | 4 | 4 | 4 | 2 | 2 | 2 | 8 | 8 | 8 | 4 | 4 | 4 |
| 10 | 5 | 5 | 5 | 3 | 3 | 3 | 7 | 7 | 7 | 5 | 5 | 5 |
| 11 | 4 | 4 | 4 | 2 | 2 | 2 | 6 | 6 | 6 | 6 | 6 | 6 |
| 12 | 5 | 5 | 5 | 3 | 3 | 3 | 5 | 5 | 5 | 7 | 7 | 7 |
| 13 | 4 | 4 | 4 | 4 | 4 | 4 | 6 | 6 | 6 | 6 | 6 | 6 |
| 14 | 5 | 5 | 5 | 5 | 5 | 5 | 7 | 7 | 7 | 7 | 7 | 7 |
| 15 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 16 | 5 | 5 | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 17 | 4 | 4 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 18 | 3 | 3 | 3 | 7 | 7 | 7 | 7 | 7 | 7 | 5 | 5 | 5 |
| 19 | 2 | 2 | 2 | 8 | 8 | 8 | 8 | 8 | 8 | 6 | 6 | 6 |
| 20 | 1 | 1 | 1 | 7 | 7 | 7 | 9 | 9 | 9 | 5 | 5 | 5 |

**Figure 1.** Seed against hops. Shows that changing seed changes result. But repeating seed gives same result.

L = 10 sites, 20 hops were made. The seed was set to 1,2,3 and 10. The program was rerun with these same seeds 3 times (figure 1).

1. When the seed was changed, the output differed. The results for seed 1, 2, 3 and 10 were not the same.
2. If the seed is repeated, the same results are gathered as shown by the 3 attempts.

100 measurements of L=10

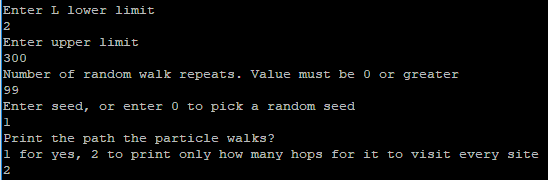


**Figure 2.** input to get data for Q1ii. Produces txt file with all the data.

**Figure 3.** seed against number of hops to reach every site.

The program was edited, so it creates a file and prints out the seed number followed by the number of hops for it to visit every site. The data.txt was copied onto an excel. The average came out to be 44.8.

The program was then modified to print the values onto a text file which could be copied to excel. Data was collected for L = 10 to 300. This was done for 100 seeds and for each L site the average was taken. Figure 5 shows how the L sites have a square proportionality to hops to visit every site. There is also has a constant of 0.4511.



**Figure 4.** Values entered to obtain the data shown in figure 5.

**Figure 5.** graphs show “hops to visit every site (averaged for 100 seeds)” against “the number of sites”. 2nd graph is a log version of the graph above it.

The graph has a relationship to the square because as the number of sites increases, the chances of reaching the furthest site decreases. The probability of the particle moving left, or right is both 50%. The most probable position for the particle after 2 consecutive hops, is back where it started which has a 50 percent probability. 25 percent for it to move 2 to the left, and 25 percent to move 2 to the right. The probability to visit each site can be related to pascals triangle. This would represent 121 on pascals triangle as stopping on the sites shown in black would be impossible in 2 moves only (as shown below) But the particle has still visited 1 of the black sites in every scenario(below).

The “furthest” place from the start would be half of L away. This can be shown by plotting all the sites on a circle. Sites “2”, and “L” would be either side of site “1”. On the opposite side of the circle would be half of “L”. As the number of sites(L) increases, it is less likely to visit sites far away. The probability for the particle to travel n consecutive hops forwards after n moves is (1/2)n. So as L increases, the number of hops increases exponentially.

# Question 2

**Figure 6.** Shows hops (averaged for 100 seeds) against number of total sites(N=L3)(values entered to get results; L lower:2, L upper: 50, Repeats 99, seed:1)

This is a very interesting relationship. It is almost linear. For the 1d walk to visit the same number of sites in a 3d walk, the number of hops it takes is exponentially larger. For example, to visit 216 sites in a 1d walk it takes on average 22853 hops. For a 3d walk it is only 1688 hops on average. This can be reasoned by the fact that there are multiple ways to reach every site in a 3d walk, but only 2 ways in 1d. The constant is also a lot larger for the 3d walk. However, the power is smaller.

# Question 3

**Figure 7. Top graph.** There are 50 lines on this graph. In each line the Number of clusters (A) (from values 1-50, hence 50 lines) remain constant, and sites(B) is varied (from 2-50, x axis). The points were repeated for 100 seeds and averaged. The y axis shows the average number of hops.

**Bottom Graph.** The second graph consists of 49 lines and is when the sites(B) remain constant and clusters(A) is varied.

From these graphs, a trend was spotted. Each line took the form of the equation y=αxβ. For graph β was constant but α varied. α was plotted on a graph. For both instances for when A is constant, and B is constant (figure 8).

**Figure 8.** α plotted for the graph in figure 7. This can be worked out using 10^(INDEX(LINEST(LOG(Yvalues), LOG(Xvalues)),2)) on excel

One thing to note is that A has a larger power then B. This indicates A influences how many hops it takes more. This makes sense, as when the particle enters a cluster it is almost trapped in that cluster. The only way for it to get out is if it lands on either “B=1” or “B=B” And then travels up or down to the next cluster. This means traveling cluster to cluster is slow. When the particle reaches a cluster, it is likely to visit every site in that cluster before moving to the next cluster. The movement of the particle can be thought of as a dice roll. To move site to site inside a cluster the dice is rolled and then the particle moves to the site on the dice. For it to move up or down a cluster, the particle must roll a 1 or “B” and then roll the same value again. The probability to visit every site<then to get 1 or B twice in a row.

Using these graphs (figure 7 and 8), the following equations were formed.

(equation 1 (from red graphs))

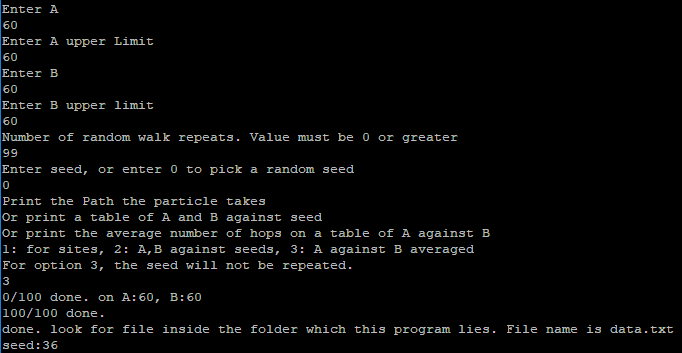
(equation 2(from blue graphs))

These 2 equations should be equivalent. However due to randomness, the accuracy is low. The constant however is accurate. The general equation for N is

These equations can be used to predict how many hops it will take to visit every site at A clusters, B sites per a cluster. Using these equations, a prediction of the average number of hops for a given A and B value is made (figure 9).

Using equation 1, A=60, B=60, gives

|  |  |
| --- | --- |
| Avg | B |
| A | 60 |
| 60 | 3328066.17 |

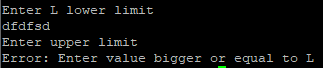
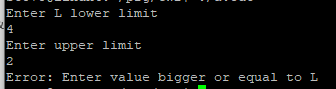


**Figure 9.** inputs(right) and the result(left) from the program.

The equation can predict the number of hops to the same order of magnitude, but the true value is inaccurate due to randomness.

# Testing

General

****  
**Figure 10.**  error handing. If non valid unit entered, the program exits.

Error messages (figure 10) appears for all inputs options if invalid inputs entered by user. With custom error messages for each input. This prevents the code from breaking with invalid inputs. This input check was put in for all pieces of code.

While coding the program, online compilers were used due to their ease of access. When the Linux.bath compiler was used once the program was finished, some bugs occurred. One issue being that “pointer++;” does not work correctly sometimes. This was fixed by using “pointer=pointer +1;”.

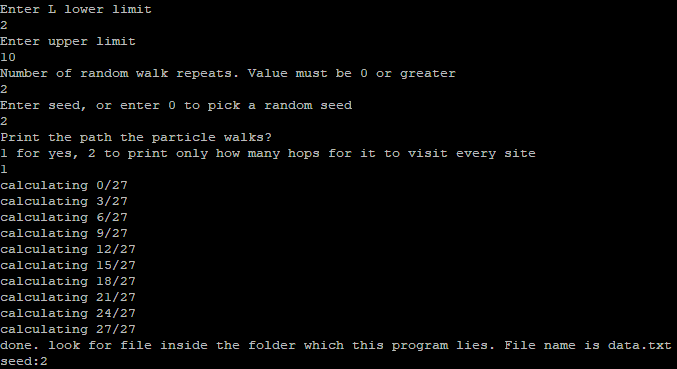
When making the program, seeds were repeated during initial gather of results. This was soon realized to be a problem. If a seed is chosen and is fixed, and then the number of sites is varied, an ordered graph is printed in which the number of hops to visit every site does not decrease as the number of sites is varied. This is because the seed repeats the path to visit the previous path and so there is no chance for the particle to visit every site in few hops. To fix this, seeds where not repeated in the results.

For Q1 and Q2 certain criteria must be met in order to show that the code is working correctly

1. The particle must only move 1 step at a time
2. The particle must move randomly
3. When reaching the lower limit, the particle must move to the highest value of that axis.
4. When reaching upper limit, the particle must move to the lowest value of that axis.
5. The particle must visit every site.
6. The last site should only be visited once, last.

Q1.c

The program was made to give the user the option to output the path the particle takes. This can be used to check if the program is running correctly.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| L | HopsTovisit | sites |  | | | | | | | | | | | |  | | | | | | | | |  | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | 1 | 1 | 2 |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |
| 3 | 5 | **1** | **3** | **1** | **3** | **1** | 2 |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |
| 4 | 10 | **1** | **4** | **1** | **4** | **1** | 2 | **1** | **4** | **1** | 2 | 3 |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |
| 5 | 13 | **1** | **5** | **1** | **5** | **1** | 2 | **1** | **5** | **1** | 2 | 3 | 2 | 3 | | 4 |  |  |  |  |  |  |  | |  |  |  |  |  |  |
| 6 | 14 | **1** | **6** | **1** | **6** | **1** | 2 | **1** | **6** | **1** | 2 | 3 | 2 | 3 | | 4 | 5 |  |  |  |  |  |  | |  |  |  |  |  |  |
| 7 | 15 | **1** | **7** | **1** | **7** | **1** | 2 | **1** | **7** | **1** | 2 | 3 | 2 | 3 | | 4 | 5 | 6 |  |  |  |  |  | |  |  |  |  |  |  |
| 8 | 16 | **1** | **8** | **1** | **8** | **1** | 2 | **1** | **8** | **1** | 2 | 3 | 2 | 3 | | 4 | 5 | 6 | 7 |  |  |  |  | |  |  |  |  |  |  |
| 9 | 19 | **1** | **9** | **1** | **9** | **1** | 2 | **1** | **9** | **1** | 2 | 3 | 2 | 3 | | 4 | 5 | 6 | 7 | 6 | 7 | 8 |  | |  |  |  |  |  |  |
| 10 | 26 | **1** | **10** | **1** | **10** | 1 | 2 | **1** | **10** | **1** | 2 | 3 | 2 | 3 | | 4 | 5 | 6 | 7 | 6 | 7 | 8 | 7 | | 6 | 5 | 6 | 7 | 8 | 9 |

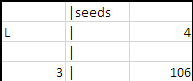
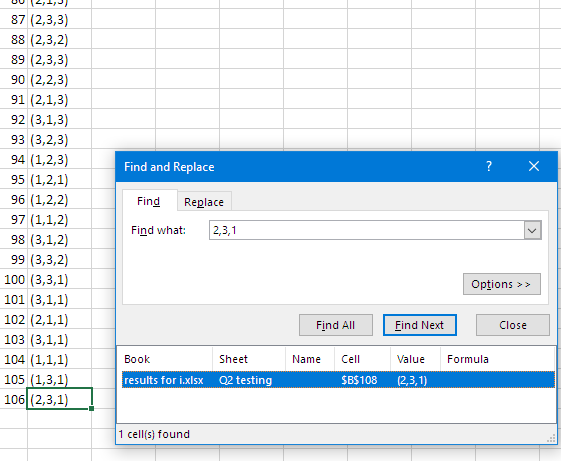
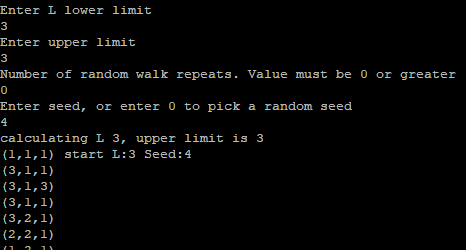
**Figure 11.** Top shows the input. Bottom is the result. Bold shows hops for site 1, from and to, site L working correctly. The last site visited is squared and has not been visited before.

From inspection (figure 11), shows that the last site the particle visits, has not been visited before. Furthermore, the particle moves back and forwards which shows randomness. Lastly, if the particle moves “-1”, from 1, it goes to L. And if it moves “+1” from L, it returns to site 1. This is shown in bold in figure 11.

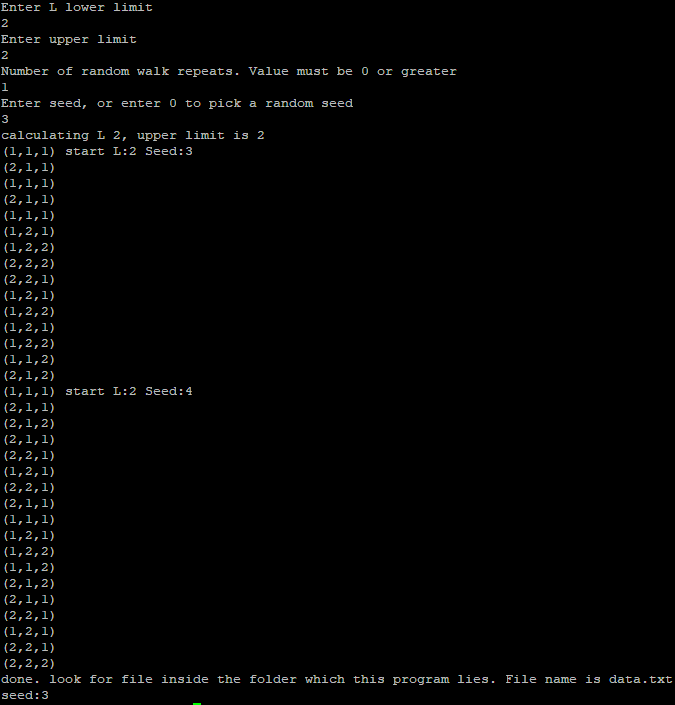
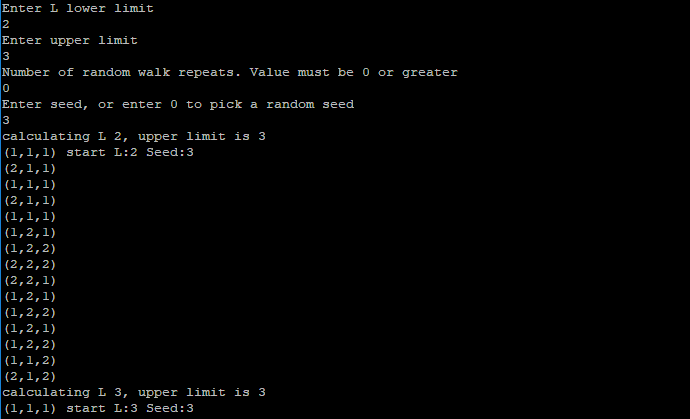
1. ✔The particle must only move 1 step at a time
2. ✔The particle must move randomly
3. ✔When reaching the lower limit, the particle must move to the highest value of that axis.
4. ✔When reaching upper limit, the particle must move to the lowest value of that axis.
5. ✔The particle must visit every site.
6. ✔The last site should only be visited once, last.

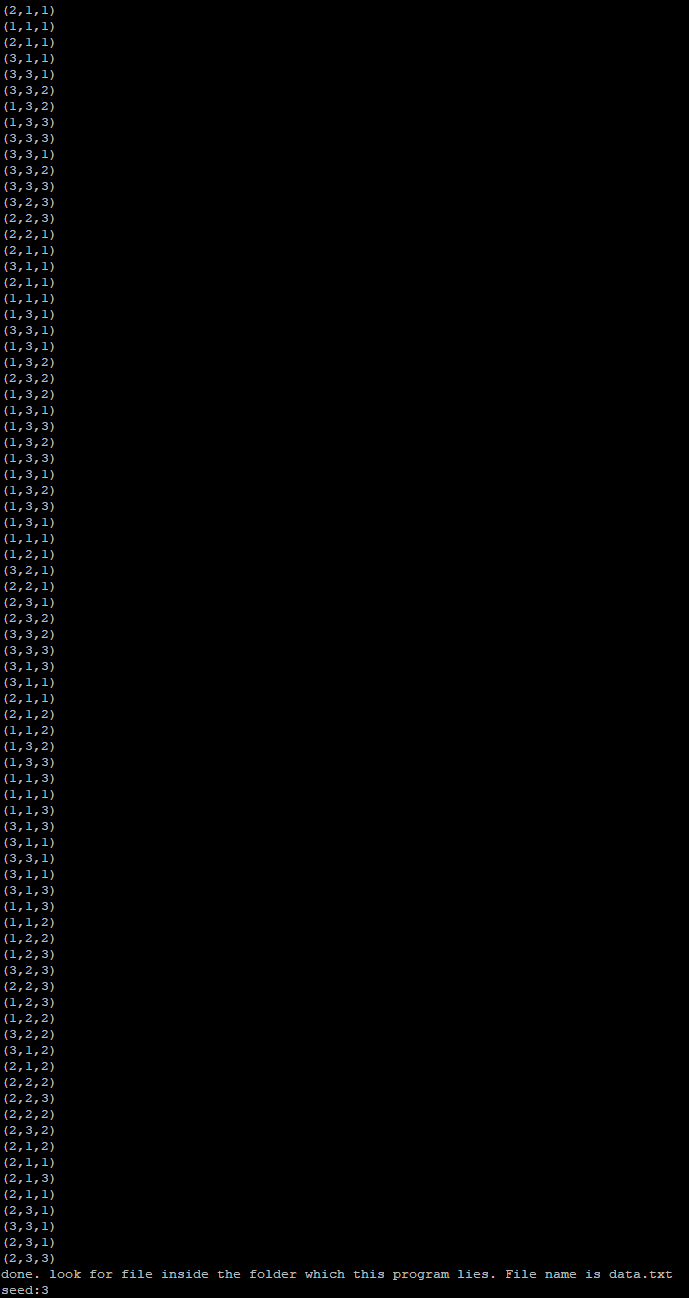
Q2.c

To test this program, 2 print statements were inserted into the program. The first print statement prints the particles starting position. The second print statement prints the particles position after the particle has moved (these were commented out after). From inspection, the particle is only moving 1 space at a time. It also moves to L, if it moves “-1” from site 1, and back to site 1 if it moves “+1” from L. Using excel, the search function (ctrl + f) is used to check if the last site has only been visited once. Excel shows that the last site visited has not been visited before. Therefore, the part of the code that checks if the particle has visited a site before is working correctly.



**Figure 12.** L=3. Using Find function on excel shows that the site is only visited once. And the count is correct for the number of hops.





1. ✔The particle must only move 1 step at a time
2. ✔The particle must move randomly
3. ✔When reaching the lower limit, the particle must move to the highest value of that axis.
4. ✔When reaching upper limit, the particle must move to the lowest value of that axis.
5. ✔The particle must visit every site.
6. ✔The last site should only be visited once, last.

**Figure 13.**  check that it resets values currently when looping. Right screenshot shows that if L is fixed, and seeds vary, it still runs correctly. The Left shows that if L is varied from 2 to 3 and the seed is fixed it still runs correctly.

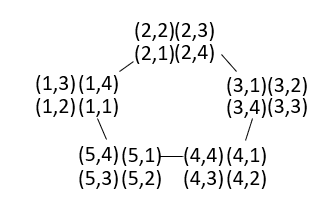
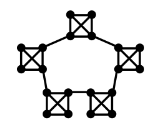
The Last site only occurs once in each iteration (figure 13). Every possible site is also visited. The results show that the program works correctly.

Q3.c

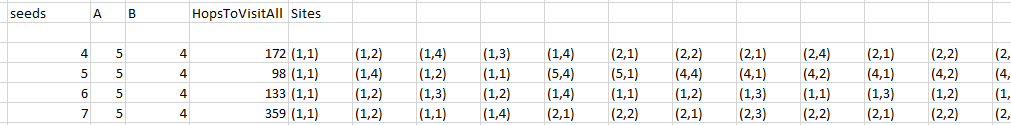
Each point would be stored in a coordinate style in an array (Figure 14). The particle can only move from 1 cluster to another when it lies in position 1 or L. From B=1 it can move down 1 cluster. And from B=L it can move up one cluster. When it reaches the last cluster, it loops back to the first cluster and vice versa. In this program there are 3 printing options. The first printing options allows to check if it is moving the particle correctly by printing the path the particle walks. This is used to check the second print option which prints a table which shows the number of hops to visit every site against seed. Then the second print option is used to check if the 3rd option is working. The third option is that the average of a given number of iterations are shown on a table of “A” against B.

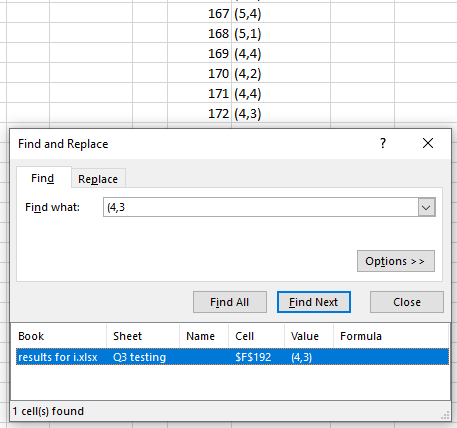
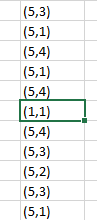
For Q3 certain criteria must be met in order for the code to be considered working correctly.

1. The particle can move to any value within in its current cluster.
2. The particle can only move up a cluster if it lies on B
3. The particle can only move down a cluster if it lies on B=1
4. When the cluster goes down from 1,1 it must go to A, B
5. When it moves up from A, B it must move to 1,1
6. The particle must visit every site.
7. The last site the particle visit must not of been visited before.



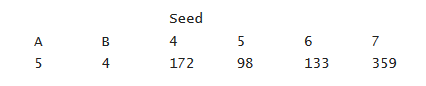
**Figure 14.** Shows how the position would be stored. (A, B). “A” would be stored in position[0] in the array, and B stored in position [1] in the array.



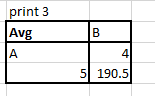


**Figure 15.** Top shows how the data is printed. For the bottom images, the data collected has been transposed to be vertical. The inputs used were: A:5, B:4, Repeats:3, Seed 4.

From figure 15 bottom right(red), it is shown that the last value only appears once. From the bottom left(purple) it is shown that the position goes from 5,4 to 1,1 and vice versa. From the bottom right and top(green) it is shown that the number of hops, 172 is correct.



**Figure 16.** print option 2 has same results as above.



**Figure 17.** print option 3. Which is the average of the results shown in figure 14 top (dark blue).

Figure 16 and 17 use the same inputs as figure 15 but with print option set as 2 and 3, respectively. This shows the program is working correctly and the values outputted are consisent.

1. ✔The particle can move to any value within in its current cluster.
2. ✔The particle can only move up a cluster if it lies on B
3. ✔The particle can only move down a cluster if it lies on B=1
4. ✔When the cluster goes down from 1,1 it must go to A, B
5. ✔When it moves up from A, B it must move to 1,1
6. ✔The particle must visit every site.
7. ✔The last site the particle visit must not of been visited before.

When getting the final data for the program, initially, ordered data was found. This can be seen below. This result was gathered because seeds were being reused in the same order, 1-100. When A is varied and B is constant, the particle takes the same path to complete all the iterations. E.g. A=11 contains the same path to complete A = 10 when the seed is repeated. (figure 18) This means it is impossible for A=11 to average less hops then A = 10. This means none of the lines in the graph in figure 18 can overlap. Therefore, the data is ordered and not random.



**Figure 18.** Initial data for Q3 and how if B is a constant, the particle takes the same path. A cannot get a lower “hops to visit all sites” then A-1 for the same seed.

# C Code

## Q1.c

#include <stdio.h>

#include <stdlib.h>

#include<time.h>

//F1. Move

// when called, it moves position by value

int move(int value, int \*position, int L)

{

\*position += value;

if (\*position > L) //if the position goes past L, it moves it to 1

{

\*position -= L;

}

if (\*position < 1) // if it moves below 1, it moves it to L

{

\*position += L;

}

}

//F2. Check

// checks all positions have been visited. if they have it returns 1.

int check(int \*pointer, int L) { // the pointer that is entered is the going to be visited[0]

int visitedall = 0; //used to keep track if it has visited it all

for (int i = 0; i < L; i++) { //when the atom visits a site, the visited[site] =1.

visitedall += \*pointer; //if all sites have been visited then the total sum of value in array will be ==L

pointer++;

}

if (visitedall == L)

return 1;

else

return 0;

}

//F3. Random Number generator(given)

// declaration of a function that returns a random integer

// in the range 0...(max-1)

// note random() is a built-in C function

// the % operator divides by max and takes the remainder

int randomInt(int max)

{

return (random() % max);

}

void

main()

{

//0. Variables

int upperLimit; //upper L limit

int printSites; //options for printing data

int seedLowerLimit;

int seedUpperLimit;

int seedcounter = 0;

int L; //Number of sites

int position; //current position

int count; //keeps track of how many hops

int hasItVisitedAllSites; //used as a true/false statement

int history[20000]; //saves order of visit.

FILE\*fp; //file that it prints into

//1. Assign Variables

//Setting up L upper and lower limit. if invalid value entered, it exits.

printf("Enter L lower limit\n");

scanf("%d", &L);

if (2 > L) {

printf("Error: Enter valid lower limit\n");

exit(0);

}

printf("Enter upper limit \n");

scanf("%d", &upperLimit);

if (upperLimit < L) {

printf("Error: Enter value bigger or equal to L\n");

exit(0);

}

//Setting up seed upper and lower limit. if invalid value entered, it exits.

int repeats;

printf("Number of random walk repeats. Value must be 0 or greater\n");

scanf("%d", &repeats);

if (0 > repeats) {

printf("Error: repeats must be 0 or greater\n");

exit(0);

}

printf("Enter seed, or enter 0 to pick a random seed\n");

scanf("%d", &seedLowerLimit);

if (0 > seedLowerLimit) {

printf("Error: seed must be greater or equal to 0\n");

exit(0);

}

if (seedLowerLimit ==0) {

srand(time(NULL));

seedLowerLimit = (rand() % 99) + 1;

}

seedcounter = seedLowerLimit;

seedUpperLimit = seedLowerLimit + repeats;

//choice between 2 types of data output.

printf("Print the path the particle walks?\n");

printf("1 for yes, 2 to print only how many hops for it to visit every site\n");

scanf("%d", &printSites);

if (1 > printSites || printSites > 2) {

printf("Error: Enter 1 or 2\n");

exit(0);

}

//2. Setting up file and printing headers

//data.txt made to write to

fp = fopen("data.txt", "w");

//option 1, printing headers to file

if (printSites == 1) {

fprintf(fp, "L\tseeds\tHopsTovisit\tPath");

}

//option 2, printing headers to file

if (printSites == 2) {

fprintf(fp, "Seed:%d\n", seedcounter);

fprintf(fp, "L\tHops to visit every site");

}

//setting up loading bar for user

int loading;

int loadingtotal = (upperLimit+1-L)\*(seedUpperLimit+1-seedLowerLimit);

//3. Main Loops

for (L; L <= upperLimit; L++) {

//loading bar

printf("calculating %d/%d\n", loading, loadingtotal);

//option 2, 1st column printing to file which is L value

if (printSites == 2) {

fprintf(fp, "\n%d", L);

}

for (int seed = seedLowerLimit; seed <= seedUpperLimit; seed++) {

//setting up seed

//for print 1, to prevent particle walking same path when seed is repeated

//a seed counter makes sure seed isnt repeated.

if (printSites == 2) {

srandom(seedcounter);

seedcounter++;

}

else

{

srandom(seed);

}

//loading

loading++;

//option 1, printing first 2 columns to file. These are L and seed values.

if (printSites == 1) {

fprintf(fp, "\n%d\t%d", L, seed);

}

//3.1 Resetting Variables

position = 1;

count = 0;

hasItVisitedAllSites = 0; //sets it as "false" to start with. used in while loop as a condition.

history[count] = position; //start position stored in history[0]

int visited[L]; //visited array will be used to check if it has visited a certain site before.

int \*visitedPointer = visited; // C doesn't allow an array as an input variable for a function, so pointer used instead

//assign all values in visited to 0.(Not done by default)

for (int j = 0; j < L; j++) {

visited[j] = 0;

}

visited[position - 1] = 1; //The start position has already been visited

//3.2 Random number and move

//after all sites have been visited, check fucntion change hasItVisitedAllsites to 1.

while (hasItVisitedAllSites == 0)

{

//random number generated(0 or 1). moves -1 if number generated is 0, and +1 if 1.

if (randomInt(2) == 1)

{

move(1, &position, L); //move function called.

}

else

{

move(-1, &position, L); //same as above except moves -1.

}

//3.3 Check and print

count++; //after each move, counter increases. This is number of hops

visited[position - 1] = 1; //adds "TRUE" to current position in visited array.

hasItVisitedAllSites = check(visitedPointer, L); //check returns 1 if all sites visited.

if (hasItVisitedAllSites == 1) {

fprintf(fp, "\t%d", count); //after all sites have been visited. the number of hops is printed.

}

//option 1

if (printSites == 1) {

history[count] = position; //stores current position in array named history

}

}

//option 1

//print all sites that have been visited in order

if (printSites == 1) {

for (int k = 0; k < count + 1; k++) {

fprintf(fp, "\t%d", history[k]);

}

}

}

}

fclose(fp); //closes data.txt

printf("calculating %d/%d\n", loading, loadingtotal);

printf("done. look for file inside the folder which this program lies. File name is data.txt\n");

printf("seed:%d\n", seedLowerLimit);

}

## Q2.c

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

// F1. Move

// when called, it moves position by given value

int move(int \*positionPointer, int value, int L)

{

\*positionPointer += value;

if (\*positionPointer > L) //if the position goes past L, it moves to 1

{

\*positionPointer -= L;

}

if (\*positionPointer < 1) // if position moves below 1, it moves to L

{

\*positionPointer += L;

}

}

// F2. Random Number generator(given)

// declaration of a function that returns a random integer

// in the range 0...(max-1)

// note random() is a built-in C function

// the % operator divides by max and takes the remainder

int randomInt(int max)

{

return (random() % max);

}

void

main()

{

//0. Variables

int L; //number of sites in each axis

int upperLimit; //upper L limit

int seedLowerLimit;

int seedUpperLimit;

int seedcounter=0;

int count; //keeps track of how many hop

int visitedall; //used to keep track if it has visited it all

FILE\*fp; //file that it prints into

int hasItVisitedAllSites; //used as a true/false statement

int x = 0; //xyz stored in position. for clarity

int y = 1;

int z = 2;

int position[3]; //current position

// 1. Assigning Variables

//Setting up L upper and lower limit. if invalid value entered, it exits.

printf("Enter L lower limit\n");

scanf("%d", &L);

if (2 > L) {

printf("Error: Enter valid lower limit\n");

exit(0);

}

printf("Enter upper limit \n");

scanf("%d", &upperLimit);

if (upperLimit < L) {

printf("Error: Enter value bigger or equal to L\n");

exit(0);

}

//Setting up seed upper and lower limit. if invalid value entered, it exits.

int repeats;

printf("Number of random walk repeats. Value must be 0 or greater\n");

scanf("%d", &repeats);

if (0 > repeats) {

printf("Error: repeats must be 0 or greater\n");

exit(0);

}

printf("Enter seed, or enter 0 to pick a random seed\n");

scanf("%d", &seedLowerLimit);

if (0 > seedLowerLimit) {

printf("Error: seed must be greater or equal to 0\n");

exit(0);

}

if (0 == seedLowerLimit) {

srand(time(NULL));

seedLowerLimit = (rand() % 99) + 1;

}

seedcounter = seedLowerLimit;

seedUpperLimit = seedLowerLimit + repeats;

// 2.File and print headers

// makes a data.txt file to write to

fp = fopen("data.txt", "w");

// headings written to file

fprintf(fp, "Seed:%d \nL\tHops to visit every site",seedcounter);

// prints seed number on top

// 3. Loops

for (L; L <= upperLimit; L++) {

//loading bar for user

printf("calculating L %d, upper limit is %d\n", L, upperLimit);

//first column

fprintf(fp, "\n%d\t", L);

//number of sites that are available

int totalNoSites = pow(L, 3);

for (int seed = seedLowerLimit; seed <= seedUpperLimit; seed++) {

// 3.1 resetting variables

srandom(seedcounter);

// start position

position[x] = 1; //x

position[y] = 1; //y

position[z] = 1; //z

// printf("(%d,%d,%d) start L:%d Seed:%d\n", position[x], position[y], position[z],L,seedcounter); // use to see if code is working correctly.

//there is another print statement below for checking as well(around line 195)

seedcounter++;

visitedall = 1; //start position has already been visited.

count = 0; //hops so far

hasItVisitedAllSites = 0; //sets it as "false". This is a check for the while function to allow it to exit loop.

int visited[L][L][L];

for (int j = 0; j < L; j++) { //sets all array value to 0. prevents errors. Not done by default.

for (int k = 0; k < L; k++) {

for (int t = 0; t < L; t++) {

visited[j][k][t] = 0;

}

}

}

visited[position[x] - 1][position[y] - 1][position[z] - 1] = 1; // start site has been visited

// 3.2 Random number and Move

while (hasItVisitedAllSites == 0)

{

//3.3 Moving the position

switch (randomInt(6)) {

case 0:

{

move(position + x, 1, L);

break;

}

case 1:

{

move(position + x, -1, L);

break;

}

case 2:

{

move(position + y, 1, L);

break;

}

case 3:

{

move(position + y, -1, L);

break;

}

case 4:

{

move(position + z, 1, L); //position + z forwards memory location(pointer) of position[z] to function

break;

}

case 5:

{

move(position + z, -1, L);

break;

}

}

count++; //number of hops

//printf("(%d,%d,%d)\n", position[x], position[y], position[z]); //check used to see if program works.

//3.3 Check

//if site not registered as visited, add 1 to visitedall and changes visited site to true(1)

if (visited[position[x] - 1][position[y] - 1][position[z] - 1] == 0) {

visitedall++;

visited[position[x] - 1][position[y] - 1][position[z] - 1] = 1;

if (visitedall == totalNoSites) {

hasItVisitedAllSites = 1; //this allows it to exit while loop

fprintf(fp, "%d\t", count); //prints count to file

}

}

}

}

}

fclose(fp); //closes data.txt

printf("done. look for file inside the folder which this program lies. File name is data.txt\n");

printf("seed:%d\n", seedLowerLimit);

}

## Q3.c

#include <stdio.h>

#include <stdlib.h>

#include<time.h>

// F1. Random number generator

// declaration of a function that returns a random integer

// in the range 0...(max-1)

// note random() is a built-in C function

// the % operator divides by max and takes the remainder

int randomInt(int max)

{

return (random() % max);

}

// F2. Move Function

// position modelled as a 2d array where position[0] is A(possible cluster positions are 1 to A)

// position[1] is B(possible B positions are 1 to B). The particle can walk to any other B position inside its current cluster.

// But to go to another cluster(A) it must first be at 1 or B. It can only move 1 cluster up(from B) or down(from 1).

int move(int\* positionPointerA, int value, int A, int B)

{

int\* positionPointerB = positionPointerA + 1;

//if current position is 1, 0 will represent a change in cluster.

//all the other positions cannot get 0, so 1 is added to the value.

if (\*positionPointerB != 1) {

value++;

}

//if the value is equal to the position, 1 is added to the value. prevents it hopping to same site.

if (value >= \*positionPointerB) {

value++;

}

\*positionPointerB = value; //the particle has now moved to value

// if position is at B+1, then it has to move 1 cluster up.

if (\*positionPointerB == (B + 1)) {

\*positionPointerB = 1;

\*positionPointerA = \*positionPointerA + 1;

//if has moved past the last cluster, it will move back to first cluster

if (\*positionPointerA == (A + 1)) {

\*positionPointerA = 1;

}

}

// if it has moved to zero, it will move down a cluster

if (\*positionPointerB == 0) {

\*positionPointerB = B;

\*positionPointerA = \*positionPointerA - 1;

//if it has moved down cluster 1, it will loop back to cluster A.

if (\*positionPointerA == 0) {

\*positionPointerA = A;

}

}

}

void

main()

{

int A; //clusters

int B; //sites in each cluster

int ALimit;

int BLimit;

int Blower;

int printSites; //used for data print style

int seedUpperLimit; //seed for random number generator

int seedLowerLimit;

int randomseed;

int history[200000][2]; //used to print walk

double Average = 0.0; //average hops to visit every sites

int position[2];

int possibleJumpPos;

int vistedAll = 1;

int hasItVisitedAll = 0;

int count = 0;

long double total = 0.0;

//1. assigning Variables

//setting up A

printf("Enter A\n");

scanf("%d", &A);

if (1 > A) {

printf("Error: 1 is the minimum value for A\n");

exit(0);

}

printf("Enter A upper Limit\n");

scanf("%d", &ALimit);

if (A > ALimit) {

printf("Error: Enter value bigger or equal to A\n");

exit(0);

}

// setting up B

printf("Enter B\n");

scanf("%d", &Blower);

if (2 > Blower) {

printf("Error: 2 is the minimum value for B\n");

exit(0);

}

printf("Enter B upper limit\n");

scanf("%d", &BLimit);

if (B > BLimit) {

printf("Error: Enter value bigger or equal to B\n");

exit(0);

}

//setting up seed

int repeats;

printf("Number of random walk repeats. Value must be 0 or greater\n");

scanf("%d", &repeats);

if (0 > repeats) {

printf("Error: repeats must be 0 or greater\n");

exit(0);

}

printf("Enter seed, or enter 0 to pick a random seed\n");

scanf("%d", &seedLowerLimit);

if (0 > seedLowerLimit) {

printf("Error: seed must be greater or equal to 1.\n 0 for random seed.\n");

exit(0);

}

if (seedLowerLimit == 0) {

srand(time(NULL));

seedLowerLimit = (rand()% 99) + 1;

}

randomseed = seedLowerLimit;

seedUpperLimit = seedLowerLimit + repeats;

//choice between 3 types of data output.

printf("Print the Path the particle takes\n");

printf("Or print a table of A and B against seed\n");

printf("Or print the average number of hops on a table of A against B\n");

printf("1: for sites, 2: A,B against seeds, 3: A against B averaged\n");

printf("For option 3, the seed will not be repeated.\n");

scanf("%d", &printSites);

if (1 > printSites || printSites > 3) {

printf("Error: Enter 1 or 2 or 3\n");

exit(0);

}

// 2. Setting up File

FILE\* fp;

fp = fopen("data.txt", "w");

if (printSites == 1) {

fprintf(fp, "seeds\tA\tB\tHopsToVisitAll\tSites\n");

}

if (printSites == 2) {

fprintf(fp, "\t\tSeed\nA\tB");

for (int i = seedLowerLimit; i <= seedUpperLimit; i++) {

fprintf(fp, "\t%d", i);

}

}

if (printSites == 3) {

fprintf(fp, "Seed:%d\n", seedLowerLimit);

fprintf(fp, "Avg\tB\nA");

for (int i = Blower; i <= BLimit; i++) {

fprintf(fp, "\t%d", i);

}

}

//loading setup and visited check setup

int loading = 0;

int AlowerLimit = A;

int visited[ALimit][BLimit];

//3. Loops for calculating and printing

for (A; A <= ALimit; A++) {

// option 3 1st column

if (printSites == 3) {

fprintf(fp, "\n%d", A);

}

for (B = Blower; B <= BLimit; B++) {

//loading

printf("%d/%d done. on A:%d, B:%d\n", loading, (ALimit - AlowerLimit + 1) \* (BLimit - Blower + 1) \* (repeats + 1), A, B);

//option 3 resetting average

if (printSites == 3) {

Average = 0.0;

}

//option 2 1st column

if (printSites == 2) {

fprintf(fp, "\n%d\t%d", A, B);

}

for (int seed = seedLowerLimit; seed <= seedUpperLimit; seed++) {

int totalNoSites = A \* B;

//3a) resetting variables

//option 3, seed is not repeated as it skews data

if (printSites == 3) {

srandom(randomseed);

randomseed++;

}

else {

srandom(seed);

}

position[0] = 1; //A

position[1] = 1; //B

loading++;

for (int i = 0; i < A; i++) {

for (int j = 0; j < B; j++) {

visited[i][j] = 0;

}

}

visited[position[0] - 1][position[1] - 1] = 1;

vistedAll = 1;

hasItVisitedAll = 0;

count = 0;

history[count][0] = position[0];

history[count][1] = position[1];

//option 1 first 3 column

if (printSites == 1) {

fprintf(fp, "\n%d\t%d\t%d", seed, A, B);

}

//3b. random number

while (hasItVisitedAll == 0) {

//calculating the number of possible sites it can jump to.

//position 1 and B can jump to other clusters which has 1 more possibility, when compared to other B sites

if (position[1] == 1 || position[1] == B) {

possibleJumpPos = B;

}

else

{

possibleJumpPos = B - 1;

}

//3c. call move function

move(position, randomInt(possibleJumpPos), A, B);

count++; //1 hop done

//option 1, saves the position in history

if (printSites == 1) {

history[count][0] = position[0];

history[count][1] = position[1];

}

// 3d. check and print

// if not visited site yet, add one and check if it has visited all sites.

if (visited[position[0] - 1][position[1] - 1] == 0) {

vistedAll++;

visited[position[0] - 1][position[1] - 1] = 1;

//option 1,2 if visited all sites, print hops on data.txt.

if (vistedAll == totalNoSites) {

hasItVisitedAll = 1;

if (printSites != 3) {

fprintf(fp, "\t%d", count);

}

if (printSites == 3) {

Average += count;

}

}

}

}

//option 1, print all the sites it visited

if (printSites == 1) {

for (int j = 0; j <= count; j++) {

fprintf(fp, "\t(%d,%d)", history[j][0], history[j][1]);

}

}

}

//option 3, find average and print that

if (printSites == 3) {

total = Average / (repeats + 1);

fprintf(fp, "\t%Lf", total);

}

}

}

fclose(fp); //closes data.txt

printf("%d/%d done.\n", loading, (ALimit - AlowerLimit + 1) \* (BLimit - Blower + 1) \* (repeats + 1));

printf("done. look for file inside the folder which this program lies. File name is data.txt\n");

printf("seed:%d\n", seedLowerLimit);

//loading

}