Ⅰ. Purpose of the experiment

For this assignment, you will explore both iteration and recursion to solve the Tower of Hanoi --a mathematical puzzle that was popularized by the French mathematician Edouard Lucas in 1883 and is consider a classic computer science problem.

Ⅱ. Experimental content

Assignment: You will design and code two separate programs: one which uses iteration and another which uses recursion to solve the Tower of Hanoi puzzle.

Tower of Hanoi: The game consists of three poles (or “rods”) and a number of disks of different sizes which can slide onto any poles. The objective of the puzzle is to move the entire stack to another pole, obeying the following simple rules:

- You can only move one disk at a time.
- A disk cannot be put on top of a smaller disk.

Requirements:

1. Design, in pseudo code, Tower of Hanoi algorithms
2. In the programming language of your choice:
   a. Present a programmatic solution to the Tower of Hanoi using iteration
   b. Present a programmatic solution to the Tower of Hanoi using recursion
3. Provide a written report which:
   a. Introduces the assignment and the problem to be solved
   b. Briefly discusses the steps you took and decisions you made to start and solve the problem
   c. Discusses: did the pseudo code help you design the code? Briefly explain your answer. Which version of the program works “best”; discuss your answer.
   d. Concludes the report: wrap up the report (summary of assignment and your work)

Deliverables: You will submit a single Word or PDF document which includes:

a. The written discussion elements found in #3 (in the above section)
b. Your pseudo code
c. Video of each program/version (the working URL to the video should be listed in the report document; at the end of the report):
   i. compiling
   ii. Showing the programs run from start to finish with the output of the solution for each version
d. List (working URLs are sufficient) all resources used to complete this work (you are expected to explore at least two sources)
e. Text of your code copied at the end of the report

Expectations:

1. Compiling errors: Your solutions *must compile*
2. Readability. Your code should meet basic readability principles:
    1. Separate each component/part with white space.
    2. Align everything in a meaningful way.
3. Comments: you must include enough comments to ensure that the code is described in sufficient detail such that anyone else looking at the code can easily understand the design and the purpose of the code.
4. All code and work associated with this assignment is your own work/written by you.

III. Experimental steps

Step1:

Find out the way to solve the Hanota problem, and make a deep understanding and analysis of it. In this step, through summary analysis and online search, I got the basic steps to solve the Hanota problem. Assuming the number of disks is n, it can be summarized in the following steps

1. Move n-1 discs to the auxiliary pole
2. Move the nth disc, the last (largest) disc, to the target pole
3. Move the previous n-1 discs from the auxiliary pole to the target pole

In order to move the first n-1 discs from the source pole to the auxiliary pole, we can consider it as an independent Hanoi Tower problem, that is, the number of discs becomes n-1, the target pole becomes the auxiliary pole. By analogy, we can regard each global movement as a new Hanoi Tower problem according to the number of discs that need to be moved, as well as the source pole and the target pole at this time. Until the number of discs to be moved is 1, we will directly complete the movement

Step2:

According to the su1mmarized solutions, they are combined with recursion and iteration respectively.

For recursion：

1. The idea of recursion is to call itself continuously within a function until it encounters a termination condition.Therefore, we can regard the above analogy process as a recursive process and write the analogy as a recursive function.

2. We write the moving disc as a function body. Each call requires the current number of discs, source pole, target pole and auxiliary pole.In the function body, that is, during the moving process, our termination condition is that the number of discs is 0. Under other conditions, we move n-1 discs from the current pole to the auxiliary pole, move the remaining disc from the current pole to the target pole, and then move n-1 pole from the auxiliary pole to the target pole. Each move is a function call to realize recursion. Each time we move a dics, we print the mobile information.

For iteration:

In the general iteration process, the current result is directly used as the variable of the next cycle. In the Hanoi Tower problem, each movement is divided into three steps, and the second step can only be continued after the first step is completed. The first step can be decomposed continuously, so it is more complex. Therefore, we consider using a data structure to store the state of Hanoi Tower during each movement, so as to achieve continuous decomposition of the movement and ensure that the results of this time are used as next parameters in the program.

Through the comparative analysis of each data structure and the online search for relevant information, I finally chose to use stack. It is more in line with our logic to press and snap the stack. This is also an important reason why I choose Java as the program language. Its data structure is convenient to use.

When programming, pay attention to the characteristics of "first in, last out" of the stack. Each time we move, we push the three parts obtained from the decomposition into the stack, and pop the next step that should be executed in the next loop. That is, the current move to be executed is the moving statu poped from stack. The decomposition of this move will be pushed into stack for the next move. When the number of disks in the pop-up moving information is 1, the moving information is printed.

Step3:

Write pseudo code according to the recursive iterative process summarized.

Step4: Use java to program according to pseudo code.

Ⅳ. Experimental results and analysis

Pseudo code:

For recursion process:

```
// Use each move as the recursive function subject
// Input: Current number of discs, source, destination, spare pole
procedure moveDiscs(count: integer, source: int, destination: int, spare: int);
    begin
        if count > 0
          then moveDiscs(count - 1, source, spare, destination)
               print(source + "-->" + destination)
               moveDiscs(count - 1, spare, destination, spare)
    End
```

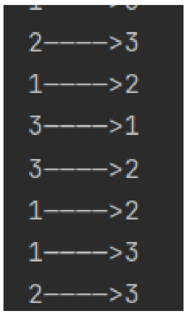
For iteration process:

```
// Create a class to record the information of Hanoi
class HanoiState:
    count, source, destination, spare: int

// Use pop-stack and push-stack to complete iteration
// Input: Current number of discs, source, destination, spare pole
procedure iterateHanoi(source: int, destination: int, spare: int, count: int);
    begin
        var hanoi<-new HanoiState(count, source, destination, spare);
        var stack<-new Stack<HanoiState>;
        stack.push(hanoi);
        var HanoiTemp: HanoiState;
        var countTemp, sourceTemp, destinationTemp, spareTemp:int;
        while stack.isEmpty()!=0
            do
                HanoiTemp<-stack.pop();
                countTemp<-Hanoi.count;
                sourceTemp<-Hanoi.source;
                destinationTemp<-Hanoi.destination;
                spareTemp<-Hanoi.spare;
                if countTemp == 1
                    then print(sourceTemp + "————>" + destinationTemp);
                else
                    then
                    stack.push(new HanoiState(countTemp - 1, spareTemp, destinationTemp, sourceTemp));
                    stack.push(new HanoiState(1, sourceTemp, destinationTemp, spareTemp));
                    stack.push(new HanoiState(countTemp - 1, sourceTemp, spareTemp, destinationTemp));
            end
    end
```

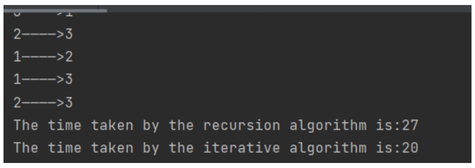
Test when discs are 12:

Part of the process:



```
2----->3
1----->2
3----->1
3----->2
1----->2
1----->3
2----->3
```

Graph 1 12_process



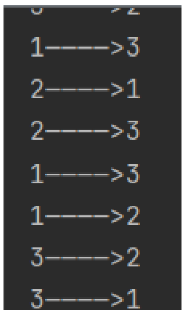
```
2———>3
1———>2
1———>3
2———>3
The time taken by the recursion algorithm is:27
The time taken by the iterative algorithm is:20
```

Time costed by recursion and iteration:

Graph 2 12_time

Test when discs are 15:

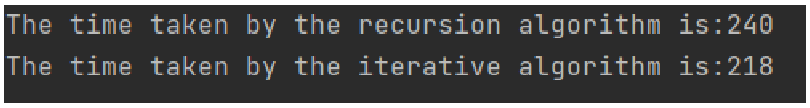
Part of the process:



```
1---->3
2---->1
2---->3
1---->3
1---->2
3---->2
3---->1
```

Graph 3 15_process

Time costed by recursion and iteration:

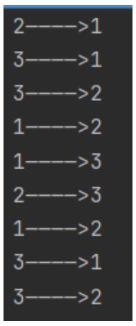


```
The time taken by the recursion algorithm is:240
The time taken by the iterative algorithm is:218
```

Graph 4 15_time

Test when discs are 20:

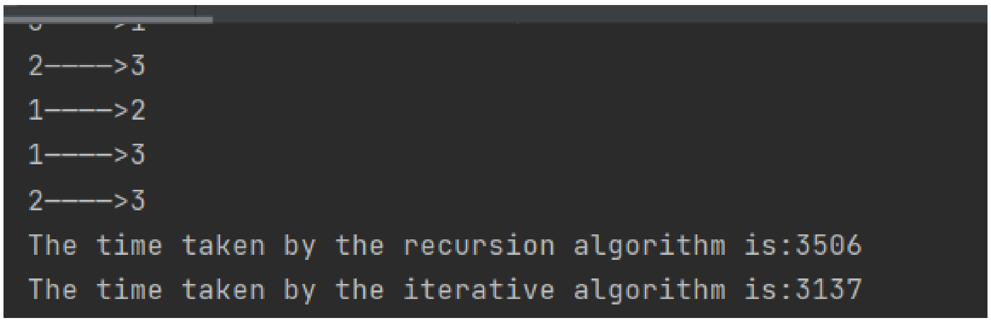
Part of the process:



```
2---->1
3---->1
3---->2
1---->2
1---->3
2---->3
1---->2
3---->1
3---->2
```

Graph 5 20_process

Time costed by recursion and iteration:



```
2---->3
1---->2
1---->3
2---->3
The time taken by the recursion algorithm is:3506
The time taken by the iterative algorithm is:3137
```

Graph 6 20_time

Analysis:

First, both iteration and recursion can solve the Hanoi Tower problem.

Secondly, comparing the two, we can find that the time efficiency is not different, and the iteration is slightly more efficient. But in general, the efficiency of iteration should be much higher than recursion, especially when the number is large. In my opinion, the main reason for this situation is that when comparing iteration with recursion, the recursion is prone to lead to stack overflow and time-consuming due to the continuous calling of functions. However, in this code, our iterative function borrows the stack. During the iteration, it continuously creates new objects and pushes them onto the stack. This leads to problems such as stack overflow in our iterative function, and leads to the loss of the original advantages of iteration. Therefore, in this experiment, there is little difference in time complexity and space complexity between the two. The idea of recursion is more consistent with the logic of solving Hanoi Tower problem, and the code is more concise and easy to be understood by readers. So I think recursion is more suitable for solving Hanoi Tower.

Tiredly, pseudo code helps me sort out the logic better and make my programming more efficient.

V. Experimental experience and gains

1. Deeply explored and mastered the solution to the Hanoi Tower problem.
2. Have a deep understanding of the implementation process of iteration and recursion, as well as their advantages and disadvantages.
3. Learned how to implement iteration by borrowing stack.
4. Explored reason of high complexity of recursive space.
5. Understand the memory allocation of heap and stack.

VI. Core Code

RecursiveHanoi:

```
/**
  * A programmatic solution to the Tower of Hanoi using recursion
  * @author H9
  * @version 1.0
  */
public class RecursiveHanoi
{

    /**
      * Constructs a Hanoi game.
      * @param count The number of discs to use
      */
    public RecursiveHanoi(int count)
    {
        // Move the number of discs from pole1 to peg pole3
        // using pole2 as a temporary storage location.
        moveDiscs(count, 1, 3, 2);
    }

    /**
     * Displays a disc move.
     * @param count The number of discs to move
     * @param source The pole to move from
     * @param destination The pole to move to
     * @param spare The temporary pole
     */
    private void moveDiscs(int count, int source,int destination, int spare){
        // Recursion process
        if(count > 0)
        {
            moveDiscs(count - 1, source, spare, destination);
            System.out.println(source + "-->" + destination);
            moveDiscs(count - 1, spare, destination, source);
        }
    }
}
```

IterativeHanoi:

```
import java.util.Stack;

/**
 * A programmatic solution to the Tower of Hanoi using iteration
 * @author H9
 * @version 1.0
 */
public class IterativeHanoi {

    /**
     * Constructs a Hanoi game.
     * @param count The number of discs to use
     */
    public IterativeHanoi(int count)
    {
        // Move the number of discs from pole1 to peg pole3
        // using pole2 as a temporary storage location.
        iterateHanoi(1, 3, 2, count);
    }

    /**
     * Moves discs with iteration
     * @param count The number of discs to move
     * @param source The pole to move from
     * @param destination The pole to move to
     * @param spare The temporary pole
     */
    public void iterateHanoi(int source, int destination, int spare, int count) {
        // Creates initial Hanoi
        HanoiState hanoi = new HanoiState(count, source, destination, spare);

        // Uses stack to implement iteration
        Stack<HanoiState> stack = new Stack<>();
        stack.push(hanoi);

        // Temp variables to record current moving information
        int countTemp, sourceTemp, destinationTemp, spareTemp;
        HanoiState HanoiTemp;
        while (!(stack.isEmpty())) {
            HanoiTemp = stack.pop();
            countTemp = HanoiTemp.count;
            sourceTemp = HanoiTemp.source;
            destinationTemp = HanoiTemp.destination;
            spareTemp = HanoiTemp.spare;

            // When there has only 1 disc on source pole, move it to destination directly
            if (countTemp == 1) {
                System.out.println(sourceTemp + "————>" + destinationTemp);
            } else {
                // Because the character of stack is first-in-last-out
                // so the moving order is on the contrary
                // move n-1 discs from spare to destination
                stack.push(new  HanoiState(countTemp  -  1,  spareTemp,  destinationTemp,
sourceTemp));
                // move the last one disc from source to destination directly
                stack.push(new HanoiState(1, sourceTemp, destinationTemp, spareTemp));
                // move n-1 discs from source to spare
                stack.push(new  HanoiState(countTemp  -  1,  sourceTemp,  spareTemp,
destinationTemp));
            }
        }
    }

    /**
     * Record the state of Hanoi
     */
    static class HanoiState {
```

```
		// number of discs
		int count;
		// The pole to move from
		int source;
		// The temporary pole
		int spare;
		// The pole to move to
		int destination;

		public HanoiState(int count, int source, int destination, int spare) {
			this.count = count;
			this.source = source;
			this.destination = destination;
			this.spare = spare;
		}
	}
}
```

HanoiTest:

```
/**
 * Test RecursionHanoi and IterationHanoi
 * @author H9
 * @version 1.0
 */
public class HanoiTest{
	/**
	 * Uses two way to play Hanoi
	 * @param args A reference to a string array containing command-line arguments
	 */
	public static void main(String[] args){
		//Runtime r = Runtime.getRuntime();
		//long t1 = r.totalMemory()/1024;
		//long f1 = r.freeMemory()/1024;
		long start_time1 = System.currentTimeMillis();
		System.out.println("Recursion:");
		new RecursiveHanoi(25);
		long end_start1 = System.currentTimeMillis();
		//long t2 = r.totalMemory()/1024;
		//long f2 = r.freeMemory()/1024;


		//Runtime r2 = Runtime.getRuntime();
		//long t12 = r2.totalMemory()/1024;
		//long f12 = r2.freeMemory()/1024;
		long start_time2 = System.currentTimeMillis();
		System.out.println("Iteration:");
		new IterativeHanoi(25);
		long end_start2 = System.currentTimeMillis();
		//long t22 = r.totalMemory()/1024;
		//long f22 = r.freeMemory()/1024;

		System.out.println("The time taken by the recursion algorithm is:" + (end_start1 - start_time1));
		System.out.println("The time taken by the iterative algorithm is:" + (end_start2 - start_time2));
		//System.out.println((t2-f2)-(t1-f1));
		//System.out.println((t22-f22)-(t12-f12));

	}
}
```

VII. Reference

1. https://statanalytica.com/blog/recursion-vs-iteration/

2. https://stackoverflow.com/questions/19794739/what-is-the-difference-between-iteration-and-recursion

3. https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/

4. https://blog.csdn.net/weixin_43779268/article/details/109581789?spm=1001.2014.3001.5506

5. https://blog.csdn.net/liliangpin/article/details/125437935

6. http://t.zoukankan.com/sanyouge-p-7209511.html