Recursion

The programs we have discussed so far have been primarily iterative and procedural. Code calls other methods in a hierarchical manner. For some problems, it is very useful to have the **methods call themselves**. This is called a recursive method or function. Recursion is used in great detail in many upper division courses, so it will be very beneficial if you can learn the basics here in CS201! We'll go through several examples, because you have to think somewhat differently to solve a problem recursively.

Recursion works in a similar fashion to the mathematical process of induction. First, start by being able to solve the simplest case of a problem, or the base case. Sometimes this is referred to as the termination criteria or stopping condition. The next step is if you are given a more complex problem, split it up so that it becomes a **smaller version of the same problem** to solve. To solve it, you will use the same function. That is, you must call the original function while the same function is still executing. The difference will be that the new function call will take input representing smaller and smaller problems, until finally the base case is reached and the smallest problem is solved. After the smallest problem is solved, the pieces are put back together until the large problem is solved.

Let's start by looking at some problems that are nicely solved using recursion. First, let's look at generating The Fibonacci series.

The Fibonacci series begins with 0 and 1 and has the property that subsequent Fibonacci numbers are the sum of the previous two Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, etc.

There are many properties in nature that correspond to this number sequence. The ratio of Fibonacci numbers converges on 1.618, which is the "golden ratio" found to be aesthetically pleasing. Rabbits have been observed to multiply in accordance with the Fibonacci sequence. The sequence also describes spirals that are seen in seashells and pine cones, and the numbers pop up in many other facets of nature.

Let's look at how Fib was defined:

Fib(0) = 0
Fib(1) = 1
Fib(2) = Fib(0) + Fib(1) = 1
Fib(3) = Fib(2) + Fib(1) = 1 + 1 = 2
…
Fib(n) = Fib(n-1) + Fib(n-2)   where n >=2

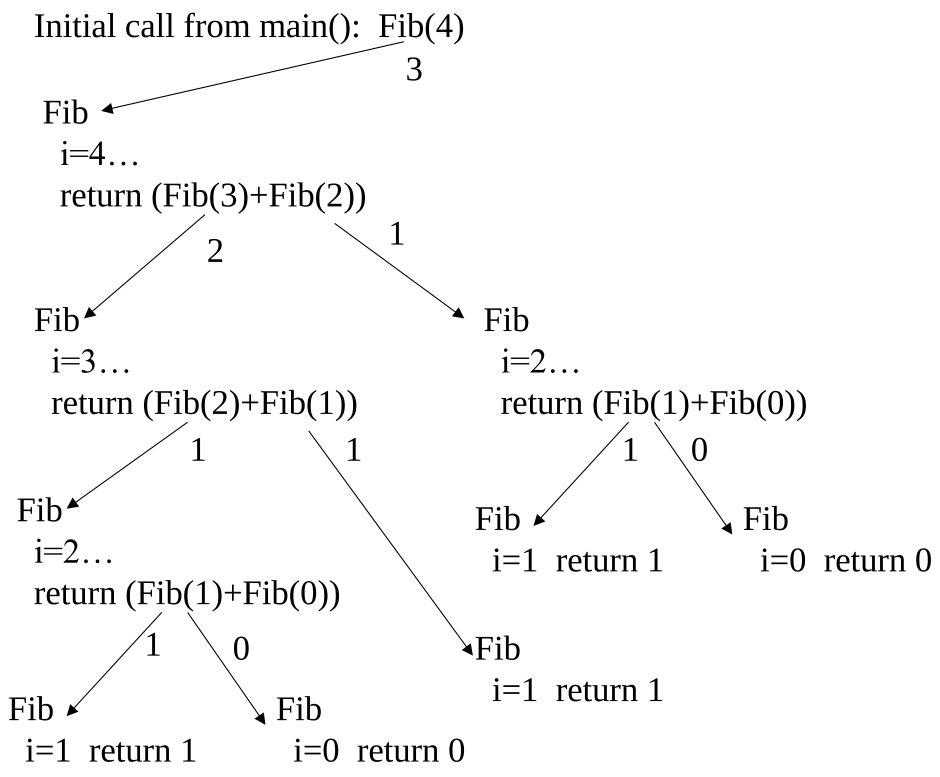
Now let's write a program to find the *ith* Fibonacci number, where *i* is some number entered by the user. First let's solve the base case , which will be the termination criteria for the recursion. This is simply to return 0 or 1 if i==0 or i==1.:

```
public class RecursionTest
{
      // Find the ith Fibonacci number and return it
      public static int Fib(int i)
      {
            if ((i==0) || (i==1)) {
                  return i;
            }
      }

      public static void main(String[] args)
      {
            // Print out the 4th Fibonacci number
            System.out.println(Fib(4));
      }
}
```

We're almost there!   This program will correctly return the 0th or the 1st Fibonacci number. But what about bigger numbers? We can just use the definition of Fib: Fib(n)=Fib(n-1) + Fib(n-2).

```
      public static int Fib(int i)
      {
            if ((i==0) || (i==1)) {
                  return i;
            }
            else {
                  return (Fib(i-1) + Fib(i-2));
            }
      }
```

Let's see what this looks like in terms of method calls if we give it Fib(4):



As we make the recursive calls, we are splitting the problem up into smaller versions of the same problem. In this case, smaller versions mean that we are computing a smaller Fibonacci number. The recursive calls end when we reach the base case, returning 0 or 1. Finally we end up adding together all the numbers as the recursive functions exit.

Let’s look at another problem, solving Factorial. Factorial(n) is expressed mathematically as n!.

n! = n * (n-1) * (n-2) * (n-3) * …. 1

So, 5! = 5*4*3*2*1

We can also express factorial recursively, by observing that:

n! = n * (n-1)!

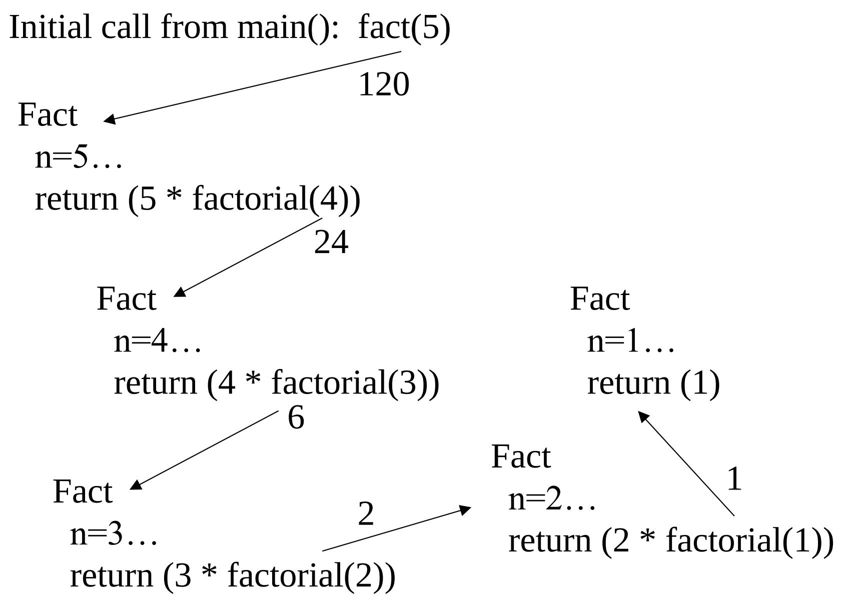
What is the base case or termination condition for the recursion? In this case, a simple base case is to know what 1! is.

1! = 1.

We can write the factorial method as:

```
// Return n!
public static long factorial(long n)
{
      if (n==1) return 1;
      return (n*(factorial(n-1));
}
```

Let's trace through factorial called from main with factorial(5):



Initial call from main(): fact(5)
120
Fact
n=5…
return (5 * factorial(4))
24
Fact
n=4…
return (4 * factorial(3))
6
Fact
n=3…
return (3 * factorial(2))
2
Fact
n=2…
return (2 * factorial(1))
1
Fact
n=1…
return (1)

As the recursion exits, we multiply the numbers together to return the actual factorial value.

When the recursive call is at the end of the function leaving no more statements on the stack to execute, then this is called end or tail recursion. In general, tail-recursive programs can be easily converted to an iterative, rather than recursive solution. Let's look at an iterative solution to the Fibonacci problem:

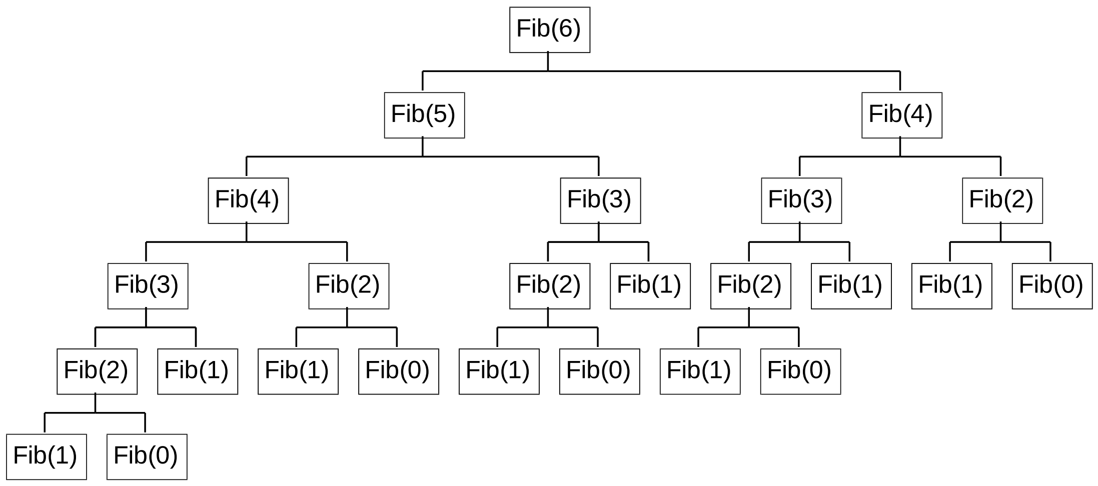
```
public static int IterativeFib(int n)
{
      int last, next_to_last, answer=0;

      if ((n==1) || (n==0)) return n;
      last=1;
      next_to_last=0;
      for (int i=2; i<=n; i++)
      {
            answer=last + next_to_last;
            next_to_last = last;
            last = answer;
      }
      return answer;
}
```

Let’s trace through this for IterativeFib(4):

```
Initially:
        Last=1, Next_To_last=1
I=2
        Answer = 1+0 = 1
        Next_To_Last = 1
        Last = 1
I=3
        Answer = 1+1 = 2
        Next_To_Last = 1
        Last = 2
I=4
        Answer = 2 + 1 = 3
        Next_To_Last = 2
        Last = 3
Exit, returning 3
```

Why is this more efficient than the recursive version of Fib? First, the iterative version doesn’t have the overhead of making method calls and pushing variables onto the stack that the recursive version does. Even more importantly, the iterative version runs in linear time to n. That is, it needs to do some factor times n computations.

In contrast, the recursive version makes on the order of Fibonacci Number of comparisons. This is exponential as n grows. For Fib, this is because we re-compute a large number of repeat Fibonacci numbers. For example, consider Fib(6):



We’re recomputing Fib(4) here twice, Fib(3) 3 times, Fib(2) 5 times, etc.! ! This is a huge amount of computation!

It turns out that the number of calls we need to make is around 2^n in recursive Fibonacci. Consider a value of n that is 30. This means we would have to make around 2^{30} or about a billion function calls to compute the 30th fibonacci number! Obviously, trying to compute the 1000th fibonacci number would not be feasible using the recursive routine. However, computing the 30th fibonacci using the iterative solution would only require some factor of 30 computations!

From this little example it looks like iteration is better than recursion. For the Fibonacci method, it most certainly is. However, there are many other methods where it is much more elegant and much easier (and just as efficient) to write a recursive solution. We'll see some of these a bit later.

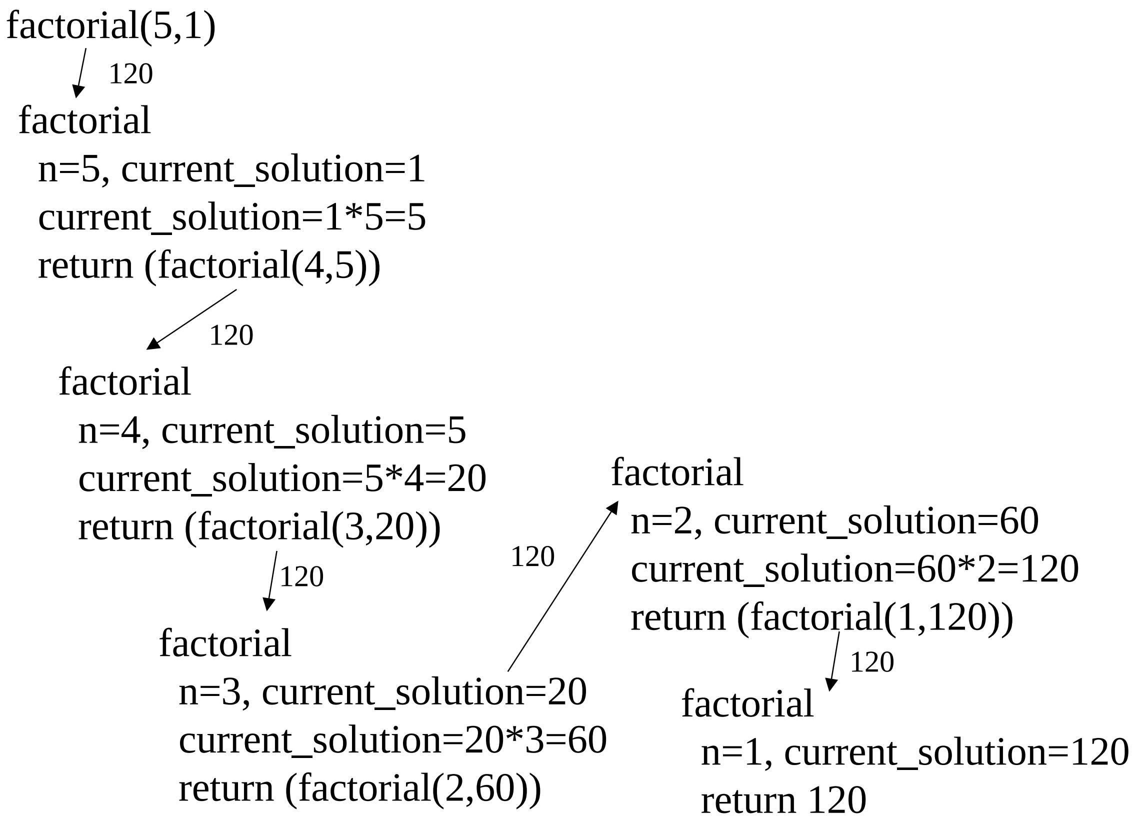
Exercise: Write a recursive method "exponent" that computes x raised to the yth power:
e.g. exponent(2,5) should return 2^5 or 32.

So far we've been building up our answer as the recursive method calls end, piecing together each return value to make the final answer. Let's write the factorial program recursively, but instead of building the final value as the recursive calls end, we will build the value up and return it when we've reached the end of the recursion. In this case, we'll pass a parameter to each recursive call that contains the current solution.

The factorial call will be invoked via: factorial(number, current-solution), where we will initialize our current-solution to be some initial base number. For factorial, 1 is a good initial base number since we'll be multiplying values up.

```
public static long factorial(long n, long current_solution)
{
      if (n==1) return current_solution;
      current_solution = n * current_solution;
      return (factorial(n-1, current_solution));
}
```

Let's look at what happens when this is invoked to compute the factorial of 5. We start with a current_solution of 1, and invoke the function as: factorial(5,1);



In this case, we compute the factorial as we make the recursive calls. When we finally reach the end of the recursion , where n =1, we simply return back the value we've computed. In this example, the final factorial is returned from each method call.

So far, we've been looking at tail-recursive methods. Let's look at a method that doesn't use tail recursion. See if you can figure out what the following code would do:

```
public static void Mystery()
{
      Scanner keyboard = new Scanner(System.in);
      String s;
      try {
            s = keyboard.nextLine();
      }
      catch (Exception e) {
            System.out.println(e);
            System.exit(0);
      }
      if (!s.equals(".")) Mystery();
      System.out.println(s);
}

public static void main(String[] args)
{
      Mystery();
}
```

When we run this code, let's say we give it the input of:

```
neat
foo
bar
zot
.
```

```
Mystery()
    s='neat'
    Mystery()
        s='foo'
        Mystery()
            s='bar'
            Mystery()
                s='zot'
                Mystery()
                    s='.'
                    no recursive call!
                    Print out "."
                    Return
                Print out 'zot'
                Return
            Print out 'bar'
            Return
        Print out 'foo'
        Return
    Print out 'neat'
    Return
```

If we look at the order in which text was printed, we get the input data in reverse. So what the mystery program does is reverse each line of input you give, ending with a period. This behavior is made possible because in each recursive call, we get a new local variable for s which is set to whatever the user types in. Thanks to the stack, as the recursive calls exit, the computer remembers where we left off in the execution along with the contents of the variable s at the time the recursive call was made.

Binary Search

Recursion gives us a particularly nice way to search for data quickly in a sorted array. Binary search is much quicker than simple linear search, where we scan through each item in the array one by one until we find (or don't find) what we're looking for.

The binary search algorithm eliminates one-half of the elements in the array being searched after each comparison. The algorithm starts at the middle element of the array and compares it with the search key. If they are equal, the search key is found and the array information is returned. Otherwise, the problem is reduced to searching one half of the array. If the search key is less than the middle element, the first half is searched; otherwise the second half is searched. The process continues until the search key is equal to the middle element or the subarray consists of one element that is not equal to the search key (ie. The key is not found).

Here is pseudocode for a recursive solution to the binary search algorithm.

```
int BinarySearch(int array A, int SearchKey, int StartIndex, int EndIndex)
    if StartIndex>EndIndex
        return Not-Found
    midIndex = (StartIndex + EndIndex) / 2
    if A[midIndex] == SearchKey
        return midIndex
    else if A[midIndex] > SearchKey
        // Search first half
        return BinarySearch(A, SearchKey, StartIndex, midIndex-1)
    else
        // Search second half
        return BinarySearch(A, SearchKey, midIndex+1, EndIndex)
```

Initially, the routine requires the leftmost and rightmost bounds of the array to be passed in (e.g. 0 to array length -1).

This routine requires $\log_2$ comparisons to search through n records, a tremendous increase in performance over linear search for large values of n. (e.g, n=1048576, $\log_2(n)=20$).

Here is an example of binary search to find the index of an element in a array of integers:

```
public static int BinarySearch(int[] a, int target,
            int startIndex, int endIndex)
{
      int mid;

      if (startIndex > endIndex) return -1;
      mid = (startIndex + endIndex) / 2;
      if (a[mid] == target) return mid;
      else if (a[mid] > target)
         return BinarySearch(a, target, startIndex, mid-1);
      else
         return BinarySearch(a, target, mid+1, endIndex);
}


public static void main(String[] args)
{
      int[] a = {13,16,31,53,97};
      System.out.println(BinarySearch(a,97,0,a.length - 1));
}
```

The array a initially contains the following:

a[0]=13
a[1]=16
a[2]=31
a[3]=53
a[4]=97

then let's trace through what happens when we invoke the method to find the index containing target value 97:

Initial call:

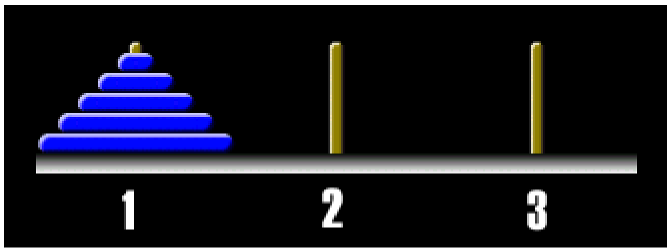
```
    BinarySearch(a, 97, 0, a.length-1)
    BinarySearch (a, 97, 0, 4)
        mid = 4 /2  = 2
        a[2] < 97 so
            return BinarySearch (a, 97, 2+1, 4)
                BinarySearch (a, 97, 3, 4)
                    mid = (3+4) / 2 = 3
                    a[3] < 97 so
                     return BinarySearch(a, 97, 3+1, 4)
                        BinarySearch (a, 97, 4, 4)
                            mid = (4+4) / 2 = 4
                            a[4] == 97
                            return 4
```

In this case, we made three comparisons, less than the five comparisons if we made a linear search. Note that there is a larger amount of overhead in this scheme than the linear search, however. For very small arrays, it will be faster to do linear search than binary search. As the array size increases however, binary search quickly outperforms linear search.

Towers of Hanoi

As a final example, let's look at a problem that really shows the power of recursion. Even though the problem is fairly complex, through the elegance and power of recursion our solution will be very short. The problem we will examine is the game of the Towers of Hanoi.

In the Towers of Hanoi, the game is played with three pegs and a pile of disks of different sizes. We start with all the disks stacked on one peg:



The object of the game is to move the entire stack from peg 1 to peg 3 while obeying the following rules:

1. Only one disk can be moved at a time
2. A larger disk can never go on top of a smaller disk

According to legend, there was a religious temple with 64 such golden disks. The end of the game was supposed to signal the end of the world. In reality, the game was likely invented by a French mathematician. Nevertheless, let's write a computer solution to help facilitate the end of the world!

Let's start with the base case. A stack with a height of 1 is trivial. All we need to do is move it from peg 1 to peg 3.

A stack with a height of 2 is also pretty trivial. All we need to do is move the first disk out of the way onto our temporary peg, which is peg 2. So we move from peg 1 to peg 2. Then move the second disk from peg 1 to peg 3. Finally move the disk on peg 2 to peg 3.

A stack with a height of 3 gets more interesting. Rather than figure out a specific set of moves to solve our problem, let's say that we move the top two disks to peg 2. We can do this using the same procedure we outlined above for a stack with a height of 2, only we are making the final destination as peg 2, not peg 3. In this case, we'd be using peg 3 as the temporary peg. Once the top two disks are in order on peg 2, we move the large disk on peg 1 to peg 3. Then we repeat the process of moving the stack from peg 2 onto peg 3. The end result is all the disks in order on peg 3.

A stack with a height of 4 can be solved using the same idea. Rather than figure out a specific set of moves, let's use the solution we generated for a stack with a height of 3. Move the top three disks from peg 1 to peg 2. Then move the large disk from peg 1 to peg 3. Finally move the stack of three disks on peg 2 to peg 3.

A stack with a height of 5 or any higher number can be solved the same way!

Let’s summarize our solution given N pegs to move:

```
If N==1, move directly from the source to destination peg
Otherwise:
    Move N-1 pegs from the source to the middle peg
    Move 1 disk from the source to the destination peg
    Move N-1 pegs from the middle peg to the destination peg
```

Here is code to solve the Towers of Hanoi problem:

```
public static void Hanoi(int height, int sourcePeg, int tempPeg,
                         int destPeg)
{
      if (height==1) {
            System.out.println("Move disk from peg " +
                  sourcePeg + " to " + destPeg);
      }
      else {
            Hanoi(height-1, sourcePeg, destPeg, tempPeg);
            System.out.println("Move disk from peg " +
                  sourcePeg + " to " + destPeg);
            Hanoi(height-1, tempPeg, sourcePeg, destPeg);
      }
}

public static void main(String[] args)
{
      Hanoi(4,1,2,3);
}
```

Here is the output of this program:

```
Move disk from peg 1 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 1
Move disk from peg 3 to 2
Move disk from peg 1 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 3
Move disk from peg 2 to 1
Move disk from peg 3 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 3
```

If you verify this, it correctly moves all the disks from peg 1 to peg 3.

Let’s trace through this for a slightly smaller problem, using Hanoi with a height of 3:
Hanoi(3, 1, 2, 3);

```
Hanoi(h=3, s=1, t=2, d=3)
        Hanoi(h=2, s=1, t=3, d=2)
                Hanoi(h=1, s=1, t=2, d=3)
                        Move disk from peg 1 to 3
                        Return
                Move disk from peg 1 to 2
                Hanoi(h=1, s=3, t=1, d=2)
                        Move disk from peg 3 to 2
                        Return
                Return
        Move disk from peg 1 to 3
        Hanoi(h=2, s=2, t=1, d=3)
                Hanoi(h=1, s=2, t=3, d=1)
                        Move disk from peg 2 to 1
                        Return
                Move disk from peg 2 to 3
                Hanoi(h=1, s=1, t=2, d=3)
                        Move disk from peg 1 to 3
                        Return
                Return
        Return
```