Recursion

Ja-Hee Kim

Agenda

01 Introduction

Iteration vs recursion, Factorial, general rule

02 Time complexity

Factorial, power function, recurrent relation

03 Avoiding recursion

Tail recursion, iteration, stack, memorizing intermediate result

04 Examples

Tower of Hanoi, binary search, indirect recursion

Introduction of Recursion

Iteration vs. Recursion

- How a computer repeats:
  - Iteration
    - for-statement
    - while-statement
  - Recursion

Definition of Recursion

- Recursion is a problem-solving process that breaks a problem into identical but smaller problems.
- Solves problem using itself
  - based on the general problem solving technique of breaking down a task into subtasks
  - Mathematical induction 수학적 귀납법

- The proof consists of two steps:
  - The base case: prove that the statement holds for the first natural number n. Usually, n = 0 or n = 1, rarely, n = −1
  - The inductive step: prove that, if the statement holds for some natural number n, then the statement holds for n + 1.

Example: Factorial function

- $n! = 1 \times 2 \times 3 \times \cdots \times n$
- $n! = \begin{cases} 1, & \text{if } n = 0, 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$

Base case

Inductive step

- Implementation

```
public long factorial(int n){
        if(n<0) {
                System.err.println("Number should be positive");
                System.exit(-1);
        }
        if(n<=1) return 1;
        else return n * factorial(n-1);
}
```

n	n!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Tracing a Recursive Method

```
public long factorial(int n){
    if(n<0) {
        System.err.println("Number should be positive");
        System.exit(-1);
    }
    if(n<=1) return 1;
    else return n * factorial(n-1);
}
```

```
public long factorial(int n){
    if(n<0) {
        System.err.println("Number should be positive");
        System.exit(-1);
    }
    if(n<=1) return 1;
    else return n * factorial(n-1);
}
```

```
public long factorial(int n){
    if(n<0) {
        System.err.println("Number should be positive");
        System.exit(-1);
    }
    if(n<=1) return 1;
    else return n * factorial(n-1);
}
```

```
public static void main(String[] args) {
    Factorial f = new Factorial();
    System.out.println(f.factorial(3));
}
```

Execution of factorial(3)

factorial(1):
n: 1
return point in main

factorial(2):
n: 2
return point in main

factorial(3):
n: 3
return point in main

main

Stack of activation records

Stack overflow

RecursionStack.java

```java
package tsp;

public class RecursionStack {

    public static void main(String[] args) {
        System.out.println(factorial(3));
    }
    public static int factorial(int n) {
        return n*factorial(n-1);
    }

}

```

Console

```
<terminated> RecursionStack [Java Application] C:₩Program Files₩Java₩jdk1.8.0_14
Exception in thread "main" java.lang.StackOverflowError
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
	at tsp.RecursionStack.factorial(RecursionStack.java:9)
```

Generic rules

- Infinite Recursion
  - If every recursive call had produced another recursive call, then a call to that method would, in theory, run forever. In practice, such a method runs until the computer runs out of resources, and the program terminates abnormally.

- The general outline of a successful recursive method definition is as follows:

  - One or more cases that include one or more recursive calls to the method being defined
    - These recursive calls should solve "smaller" versions of the task performed by the method being defined

Example of Rules

- No base

```
public static int factorial(int n) {
        return n*factorial(n-1);
}
```

- No convergence

```
public static int factorial(int n) {
        if (n == 1)
                return 1;
        else return n*(n-1)*factorial(n-2);
}
```

Time complexity of Recursion

Time efficiency of factorial

- Revised code

```
public long factorial(int n){
        System.out.println(n);
        if(n<=1) return 1;
        else return n * factorial(n-1);
}
```

- Recurrence relation of time function of factorial method.

$$t(n) = \begin{cases} 1, & \text{if } n = 1 \\ 1 + t(n-1) & \text{if } n > 1 \end{cases}$$

where t(n) is the time requirement of factorial(n)

Recurrence relation

- Recurrence relation

$$t(n) = \begin{cases} 1, & \text{if } n = 1 \\ 1 + t(n-1) & \text{if } n > 1 \end{cases}$$

- If n=4

$$t(4) = 1 + t(3) \quad = 1 + 2 = 4$$
$$t(3) = 1 + t(2) \quad = 1 + 2 = 3$$
$$t(2) = 1 + t(1) \quad = 1 + 1 = 2$$

- Closed form

  $t(n) = 1 + t(n-1)$ for $n > 1 \longrightarrow t(n) = 1 + n - 1 = n$

  proof by induction:

  if n=1: t(1) = 1 is the same to the result of the recurrence relation

  assume that if n=k, the close form is the correct, that is, t(n)=k,

  if n = k+1, t(n)=1+t(n-1) = 1+k=n.

  This means the this closed form satisfies the recurrence relation.

- Time complexity of a factorial function is O(n)

Time efficiency of computing x^n

- Recursive definition of x^n

$$x^n = \begin{cases} (x^{\frac{n}{2}})^2 & n \text{ is even and positive} \\ x(x^{\frac{n-1}{2}})^2 & n \text{ is odd and positive} \\ x^0 & n = 0 \end{cases}$$

- Recurrence relation of t(n) of x^n

$$t(n) = \begin{cases} t(n) = 1 + t(\frac{n}{2}) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

  - t(0)=0, t(1)=1, t(2)=2, t(4)=3, t(8)=4, t(16)=5

- Closed form

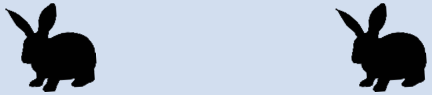
$$t(n) = 1 + \lfloor \log_2 n \rfloor$$

- Time complexity: O(log *n*)

Avoiding Recursion

Fibonacci function

- $F_n = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2}. & \text{otherwise} \end{cases}$

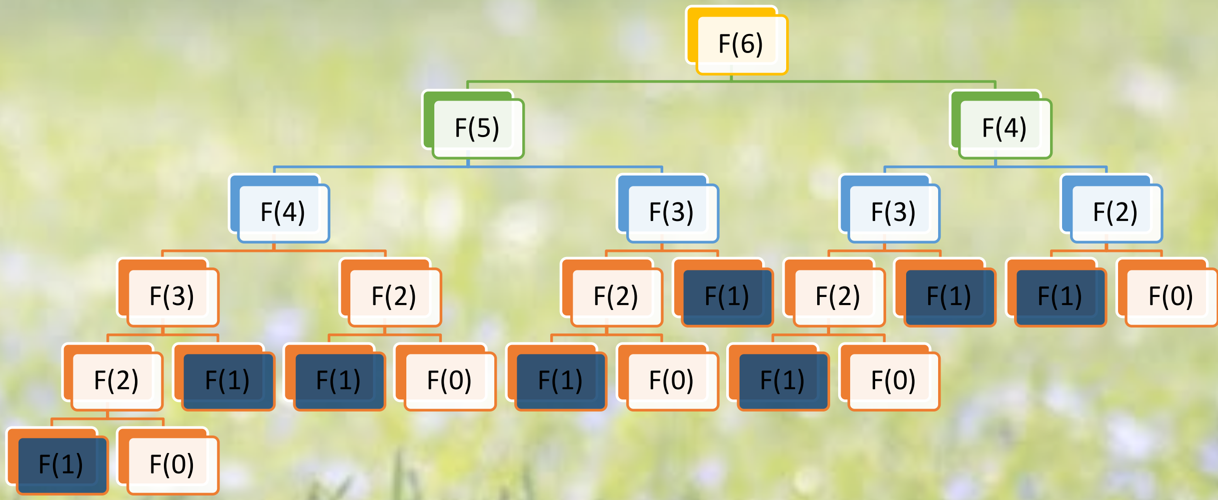
n	Number of pairs after the end of the month
0	
1	
2	
3	
4	

How many does it call f() for 6

```java
public class Fibonacci {
    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        System.out.println("Recursive Fibonacci(6) is "+ f.recursive(6));
    }
    public long recursive(int n) {
        if(n<0) {
            System.err.println("Number should be positive");
            System.exit(-1);
        }
        if(n<2) return (long)1;
        return recursive(n-1)+recursive(n-2);
    }
}
```

Problems Javadoc Declaration Console

```
<terminated> Fibonacci [Java Application] C:₩Program Files₩Java₩jdk-14.0.2₩bin₩javaw.exe  (2020. 10. 10. 오후 11:35:00 – 오후 11:35:00)
Recursive Fibonacci(6) is 13
```



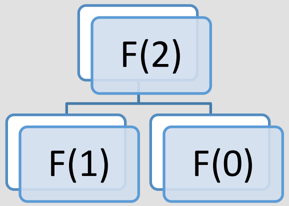
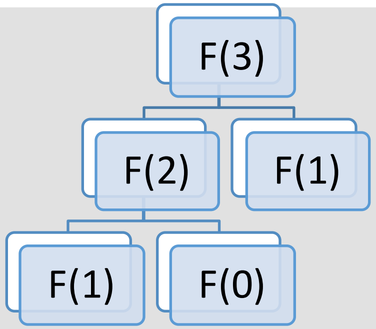
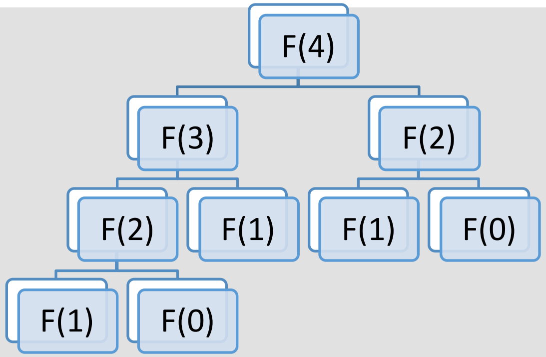
Time efficiency of Fibonacci function

- Fibonacci number

$$F_n = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2}. & \text{otherwise} \end{cases}$$

- Recurrence relation of time requirement for calculating Fibonacci number

$$\mathrm{t}(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ 1 + t(n-1) + t(n-2). & \mathrm{n} \geq 2 \end{cases}$$

n=2	n=3	n=4
		
$T(2)=1+t(1)+t(0)=3=1+F_2 > F_2$	$T(3)=1+t(2)+t(1)=5=1+F_3 > F_3$	$T(4)=1+t(3)+t(2)=9=1+F_4 > F_4$

Proof of the time complexity of Fibonacci function

- Growth function

$$t(n) = 1 + F_{n-1} + F_{n-2} = 1 + F_n > F_n$$
$$\therefore \Omega(F_n)$$

- Fibonacci number

$$F_n = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}} > \frac{(\frac{1+\sqrt{5}}{2})^n - 1}{\sqrt{5}}$$

Because, $\left|\frac{1-\sqrt{5}}{2}\right| < 1$

Therefore, $\Omega(F_n) = \Omega((\frac{1+\sqrt{5}}{2})^n) = \Omega(a^n)$, exponential algorithm

Solution1: tail recursion

- Tail recursion: The last action performed by a recursive method is a recursive call.
- Tail recursion of a Fibonacci number

```
if(n<2) return (long)1;
return recursive(n-1)+recursive(n-2);
```

```
long prePreFibo = recursive(n-1);
long preFibo = recursive(n-2);
long currentFibo = preFibo+prePreFibo;
return currentFibo;
```

```
currentFibo = preFibo+prePreFibo;
prePreFibo = preFibo;
preFibo = currentFibo;
return tailRecursion(n-1, preFibo, prePreFibo);
```

```
public long tailRecursion(int n, long preFibo, long prePreFibo) {
    long currentFibo;
    if (n < 2) return n*preFibo;
    return tailRecursion(n-1, preFibo+prePreFibo, preFibo);
}
```

$$t(n) = \begin{cases} 1 & n = 1 \\ 1 + t(n-1) & n > 1 \end{cases}$$

Solution2: iteration

1. Study the function.
2. Convert all recursive calls into tail calls. (If you can't, stop. Try another method.)
3. Introduce a one-shot loop around the function body.
4. Convert tail calls into continue statements.
5. Tidy up.

Ref: http://blog.moertel.com/posts/2013-05-11-recursive-to-iterative.html

Tail recursion

```
public long tailRecursion(int n, long preFibo, long prePreFibo) {
    long currentFibo;
    if (n < 2) return n*preFibo;
    currentFibo = preFibo+prePreFibo;
    prePreFibo = preFibo;
    preFibo = currentFibo;
    return tailRecursion(n-1, preFibo, prePreFibo);
}
```

iteration

```
public long iteration(int n) {
    long currentFibo=1;
    long preFibo=1,prePreFibo=1;
    for(int i=n; i > 1 ; i--) {
        currentFibo = preFibo+prePreFibo;
        prePreFibo = preFibo;
        preFibo = currentFibo;
    }
    return currentFibo;
}
```

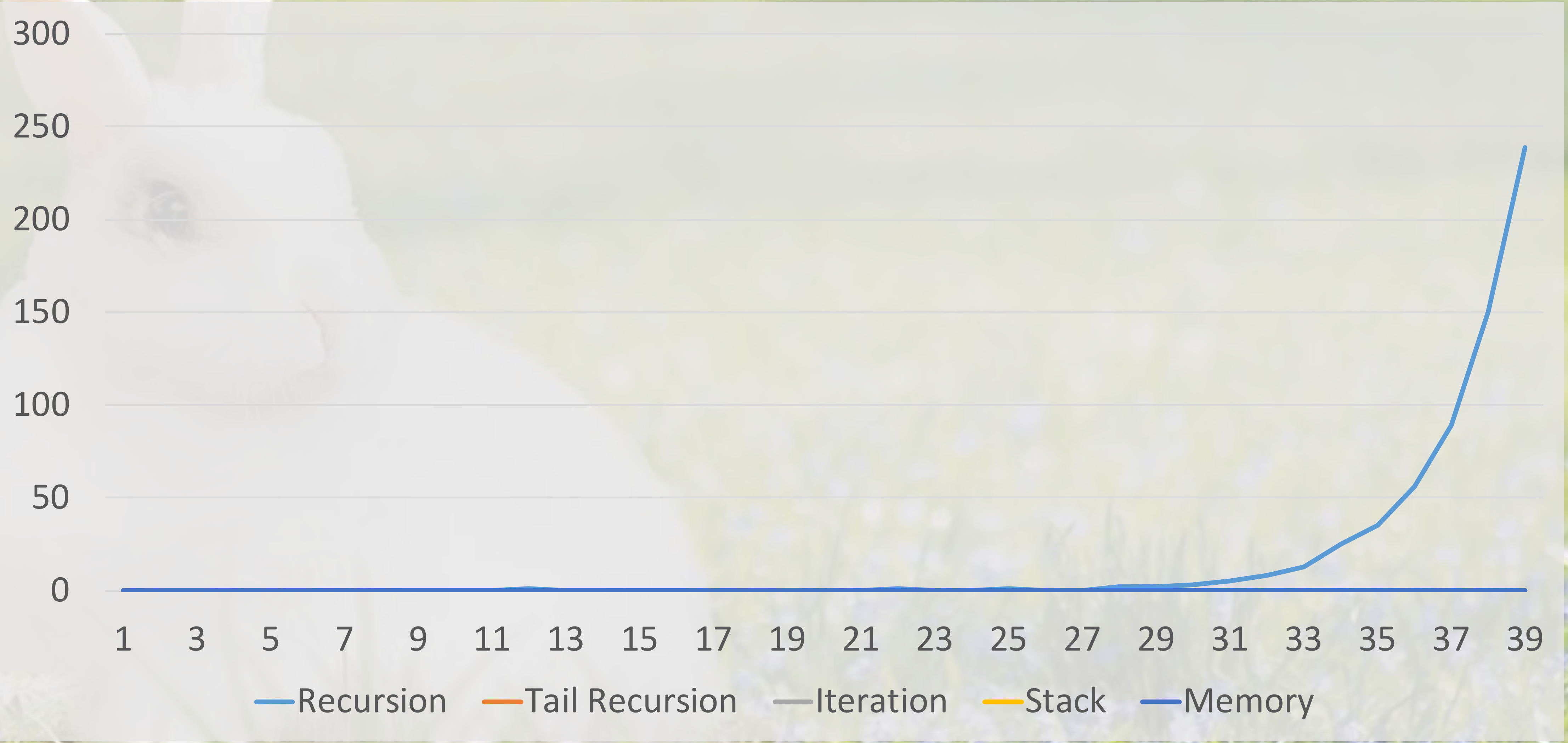
Solution 3: using a stack

```
public long usingStack(int n) {
    ArrayDeque<Record> programStack = new ArrayDeque<>(100);
    programStack.push(new Record(n, 1, 1));
    long currentFibo = n;
    while(!programStack.isEmpty()) {
        Record topRecord = programStack.pop();
        currentFibo = topRecord.n;
        long preFibo = topRecord.pre;
        long prePreFibo = topRecord.prePre;
        if(currentFibo < 3)
            currentFibo =preFibo+prePreFibo;
        else
            programStack.push(new Record(currentFibo-1, preFibo+prePreFibo, preFibo));
    }
    return currentFibo;
}
private class Record{
    private long n;
    private long pre, prePre;
    public Record(long n, long pre, long prePre) {
        this.n = n;
        this.pre = pre;
        this.prePre = prePre;
    }
}
```

Solution 4: memorize the result

```
private long[] fibonacci;
private int num=2;
private static final int MAX=1010;
public Fibonacci() {
    fibonacci = new long[MAX];
    fibonacci[0]=fibonacci[1]=1;
}
public long memorize(int n) {
    if(n<num) return fibonacci[n];
    else if(n==num) {
        fibonacci[n]=fibonacci[n-1]+fibonacci[n-2];
        num++;
        return fibonacci[n];
    }
    else return memorize(n-1)+memorize(n-2);
}
```

Comparison of time complexity



Other Examples

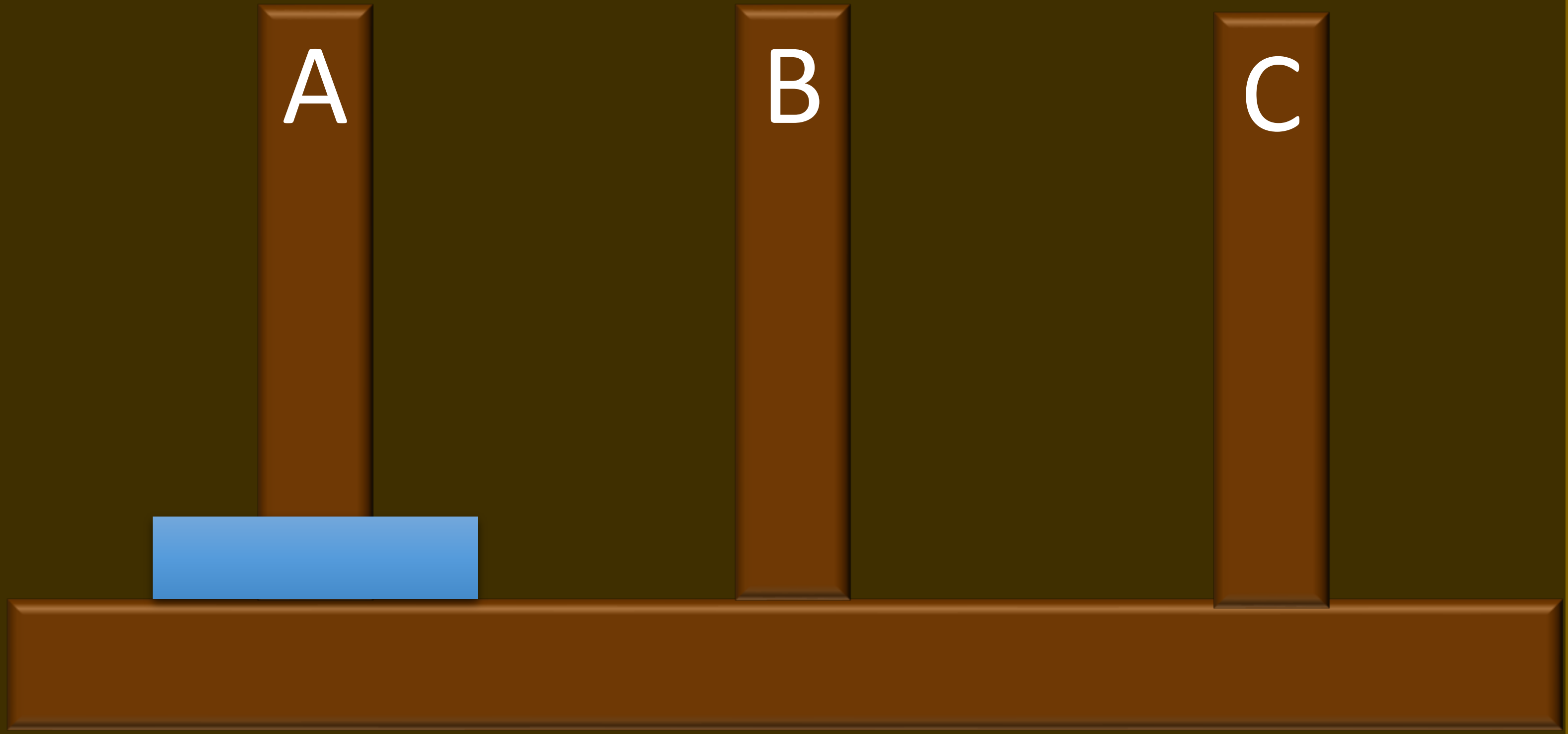
Tower of Hanoi

- Consists of
  - A board
  - Three vertical pegs
  - A progression of disks of increasing diameter
- Rule
  - Only one disk can be moved at a time.
  - No disk may be placed on top of a smaller disk.
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
- Visualization

  http://towersofhanoi.info/Animate.aspx

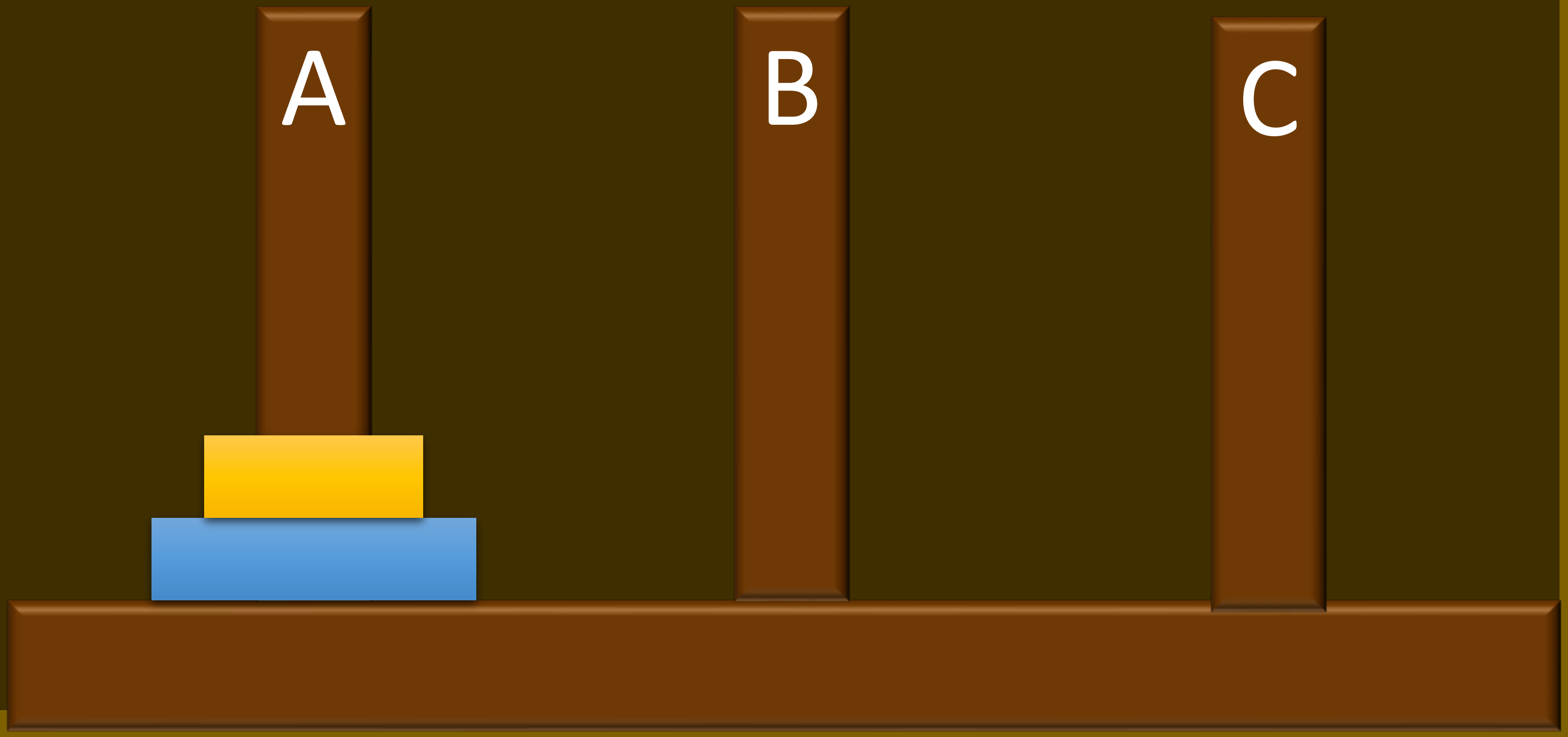
Example of tower of Hanoi

- Base case: one disk
    - Just move from the source peg to the target peg



Example of tower of Hanoi

- 2 disks



Solution of tower of Hanoi

- Base case: If the number of disks(n) is 1
  - Just move from the source to the target
- Induction step: n>1
  - Move n-1 disks from the source to another peg
  - Move the largest disk from the source to the target
  - Move n-1 disks from another peg to the target

Implementation

```java
public class Hanoi {
    public static void main(String[] args) {
        Hanoi tower = new Hanoi();
        tower.hanoi(3, "A", "C", "B");
    }
    /** solve the tower of Hanoi puzzle
     *
     * @param num the number of disks
     * @param source The source peg. You should move all disks from the source peg to target peg
     * @param target The target peg. You should move all disks from the source peg to target peg
     * @param spare The spare peg. You can use this peg for storing an unnecessary disks
     */
    public void hanoi(int num, String source, String target, String spare) {
        // base case
        if(num == 1)
            System.out.println("Move one disk from " + source + " to "+ target);
        else {
            hanoi(num-1, source, spare, target);
            System.out.println("Move one disk from " + source + " to "+ target);
            hanoi(num-1, spare, target, source);
        }
    }
}
```

Problems Javadoc Declaration Console

```
<terminated> Hanoi [Java Application] C:₩Program Files₩Java₩jdk-14.0.2₩bin₩javaw.exe (2020. 10. 11. 오후 11:37:06 – 오후 11:37:06)
Move one disk from A to C
Move one disk from A to B
Move one disk from C to B
Move one disk from A to C
Move one disk from B to A
Move one disk from B to C
Move one disk from A to C
```

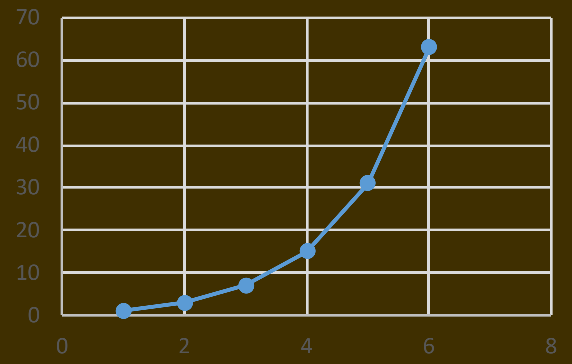
- n=1
  - 1 move
- n=2
  - 3 moves
- n=3
  - 7 moves

move(n)

$=move(n-1)+1+move(n-1)$

$= 2move(n-1)+1$

$= 2^n-1$



Efficiency of the algorithm

- Proof by induction that $m(n)=2^n-1$

$$\begin{aligned} m(k+1) &= 2 \times m(k) + 1 \\ &= 2 \times \left(2^k - 1\right) + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

where m(k) is the number of moves for k disks

- This cannot be solved less than the exponential time.

  Proof

  Assume that there is less moves for n disks, we denote the move as M(n).

  If n = 1, M(1)=m(1).

  Let's assume that M(n-1)=m(n-1). The largest disk is isolated on one peg and n - 1 dis ks are on another. The only way to move the largest disk is just to move the source to th e target.

  $M(n) \geq 2\ M(n-1) + 1 \geq 2\ m(n-1) + 1 = m(n)$

Example: binary search

- Binary search uses a recursive method to search an array to find a specified value. The array must be a sorted array:

  a[0]≤a[1]≤a[2]≤. . . ≤ a[finalIndex]

  - If the value is found, its index is returned.
  - If the value is not found, -1 is returned.

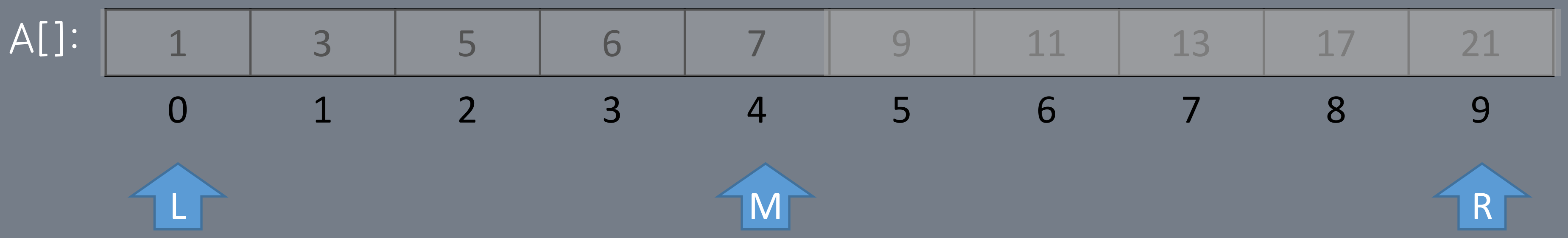
https://www.cs.usfca.edu/~galles/visualization/Search.html

Solution

- Given an array A of n elements with values or records $A_0, A_1, \ldots, A_{n-1}$, sorted such that $A_0 \leq A_1 \leq \ldots \leq A_{n-1}$, and target value X,

  1. Init: L=0, R=n-1
  2. if $L>R$, return -1,
  3. M = (L+R)/2
  4. L=m+1 and call itself, if $A_m<X$
  5. R=m-1 and call itself, if $A_m>X$
  6. if A_m=T, Return m,

  Example: X=13



Implementation and time complexity

```java
public class MyList {
    public static void main(String[] args) {
        int[] a = {1, 3, 5, 6, 7, 9, 11, 13, 17, 21};
        System.out.println(binarySearch(13, a, 0, a.length-1));
    }
    public static int binarySearch(int x, int[] a, int l, int r) {
        if(l>r) return -1;
        int m=(l+r)/2;
        if(a[m]<x) return binarySearch(x, a, m+1, r);
        else if(a[m]>x) return binarySearch(x, a, l, m-1);
        else return m;
    }
}
```

Problems Javadoc Declaration Console

```
<terminated> MyList [Java Application] C:₩Program Files₩Java₩jdk-14.0.2₩bin₩javaw.exe  (2020. 10. 12. 오전 3:17:41 – 오전 3:17
7
```

$$t(n) < \begin{cases} 1 & n = 0 \\ 1 + \left\lfloor t(\frac{n}{2}) \right\rfloor & n > 0 \end{cases}$$

$$\therefore O(\log n)$$

proof: https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/

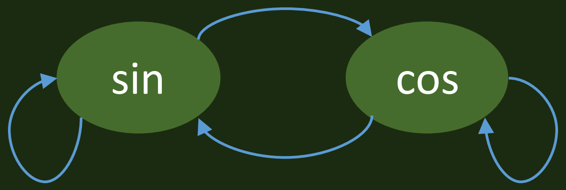
If the method cannot find the data, it i s the worst case.

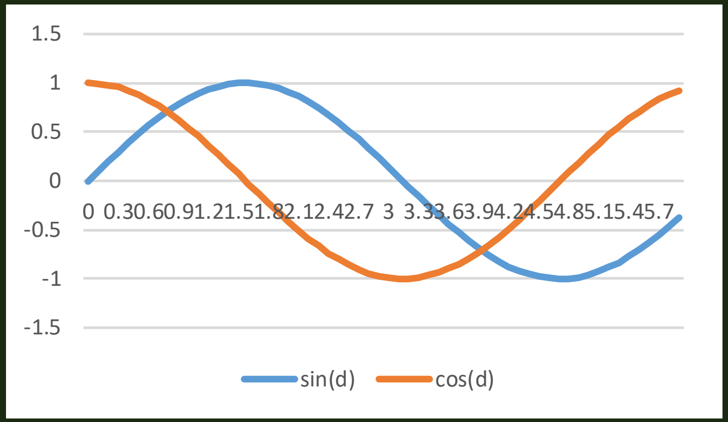
Example: Trigonometric function

- Indirect recursion

  when Method A calls a different method, this in turn calls the original calling Method A.

- Relation between trigonometric function
  - $Sin(x) = \begin{cases} x, \text{ for small } x \\ \sin(x/2)\cos(x/2) \end{cases}$
  - $Cos(x) = \begin{cases} 1, \text{ for small } x \\ \cos(x/2)^2 \sin(x/2)^2 \end{cases}$






```java
/**
 * Calculating the sine function
 * @param x : degree
 * @param num : the number of recursion
 * @return sin(x)
 */
public double sin(double x, int num) {
    if(num==0) return x;
    else return 2*sin(x/2, num-1)*cos(x/2, num-1);
}

/**
 * Calculating the cosine function
 * @param x : degree
 * @param num : the number of recursion
 * @return cos(x)
 */
public double cos(double x, int num) {
    if(num==0) return 1;
    else {
        double cos = cos(x/2, num-1);
        double sin = sin(x/2, num-1);
        return cos*cos-sin*sin;
    }
}
```

Thank you