

CMSC 132: Object-Oriented Programming II

Recursion

Recursion

- ▶ When one function calls itself directly or indirectly
- ▶ a method where the solution to a problem depends on solutions to smaller instances of the same problem.

Factorial

Definition:

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

$$5! = ?$$

$$5! = 5 * 4!$$

$$4 = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1$$

$$5 * 24 = 120$$

$$4 * 6$$

$$3 * 2$$

$$2 * 1$$

Greatest Common Divisor

- ▶ gcd: Find largest integer d that evenly divides into p and q **Euclid's algorithm. [300 BCE]**

$$\text{gcd}(p, q) = \begin{cases} p & \text{if } q = 0 \\ \text{gcd}(q, p \% q) & \text{otherwise} \end{cases}$$

← base case

← reduction step,
converges to base case

$$\begin{aligned} \text{gcd}(4032, 1272) &= \text{gcd}(1272, 216) \\ &= \text{gcd}(216, 192) \\ &= \text{gcd}(192, 24) \\ &= \text{gcd}(24, 0) \\ &= 24. \end{aligned}$$

$$4032 = 3 \times 1272 + 216$$

Greatest Common Divisor

- ▶ gcd: Find largest integer d that evenly divides into p and q **Euclid's algorithm. [300 BCE]**

$$\text{gcd}(p, q) = \begin{cases} p & \text{if } q = 0 \\ \text{gcd}(q, p \% q) & \text{otherwise} \end{cases}$$

← base case

← reduction step,
converges to base case

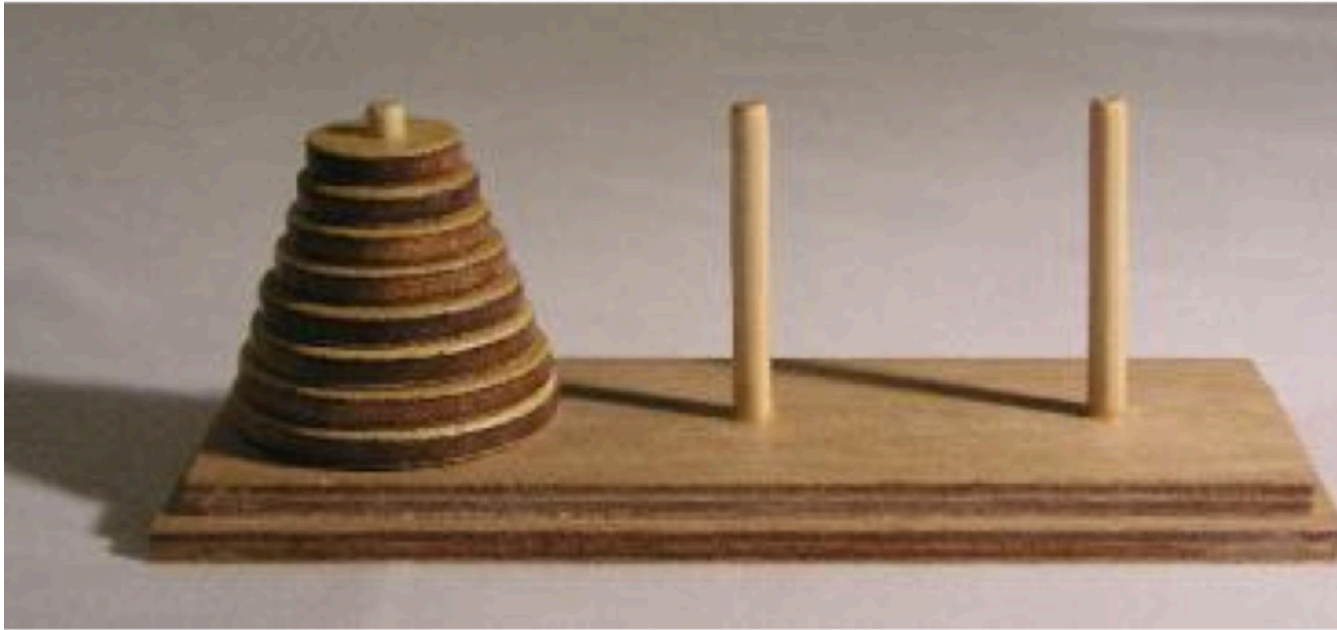
Java implementation.

```
public static int gcd(int p, int q) {  
    if (q == 0) return p;  
    else return gcd(q, p % q);  
}
```

← base case

← reduction step

Towers of Hanoi



<http://en.wikipedia.org/wiki/Image:Hanoikle.in.jpg>

Towers of Hanoi

- ▶ Move all the discs from the leftmost peg to the rightmost one.
 - Only one disc may be moved at a time.
 - A disc can be placed either on empty peg or on top of a larger disc



start

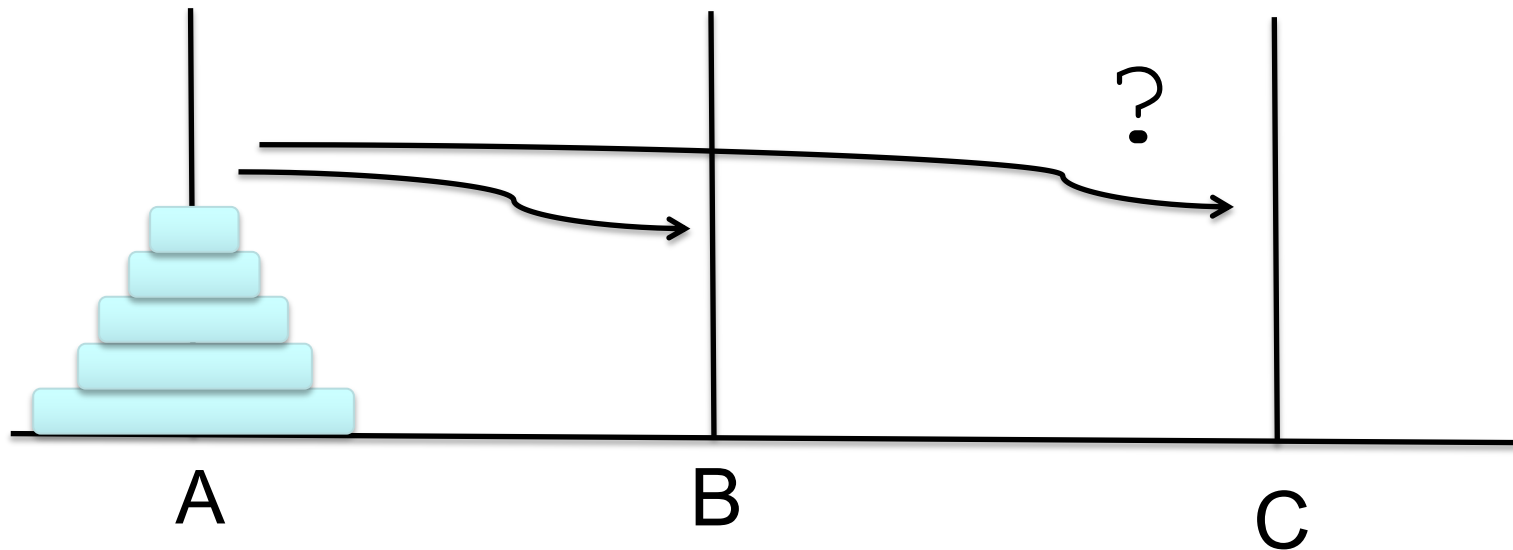


finish

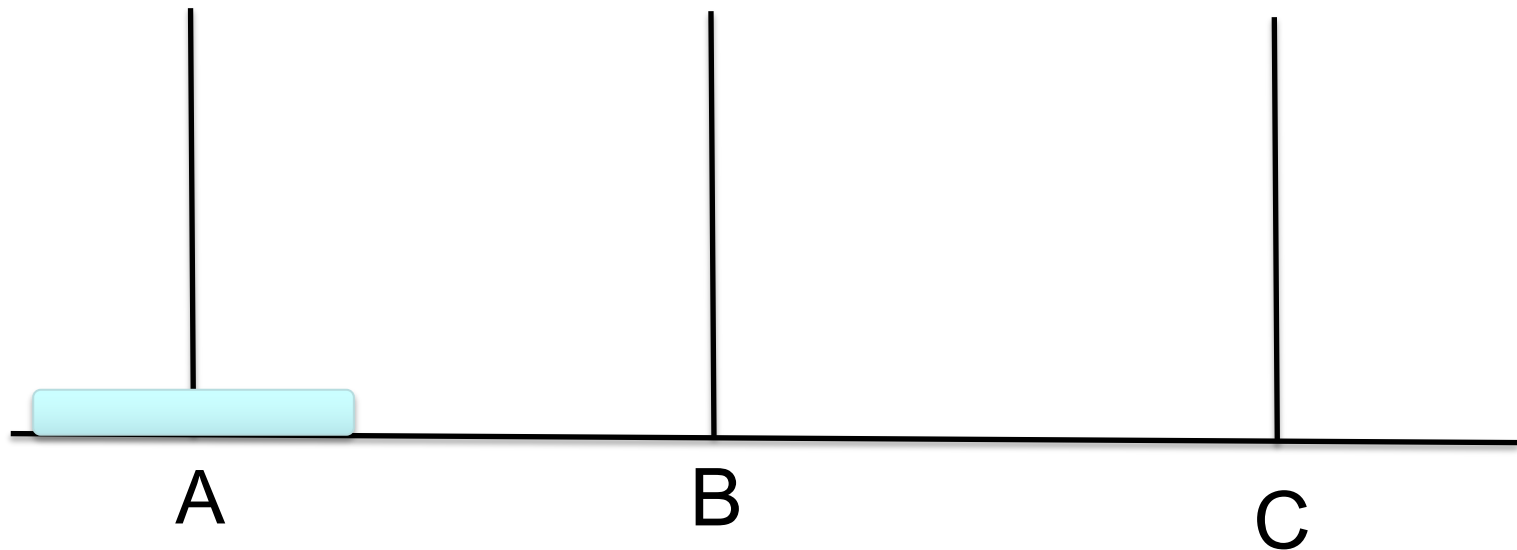
Towers of Hanoi Legend

- ▶ Q. Is world going to end (according to legend)?
 - 64 golden discs on 3 diamond pegs.
 - World ends when certain group of monks accomplish task.
- ▶ Q. Will computer algorithms help

Let's move disks

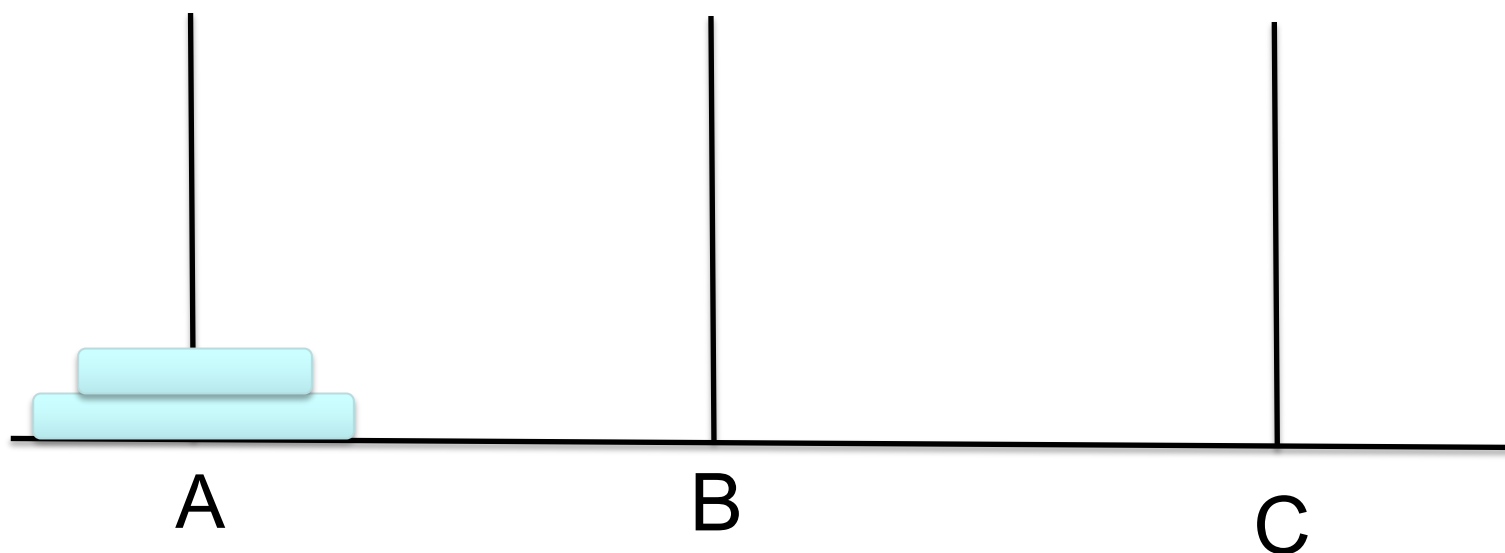


Move one disk



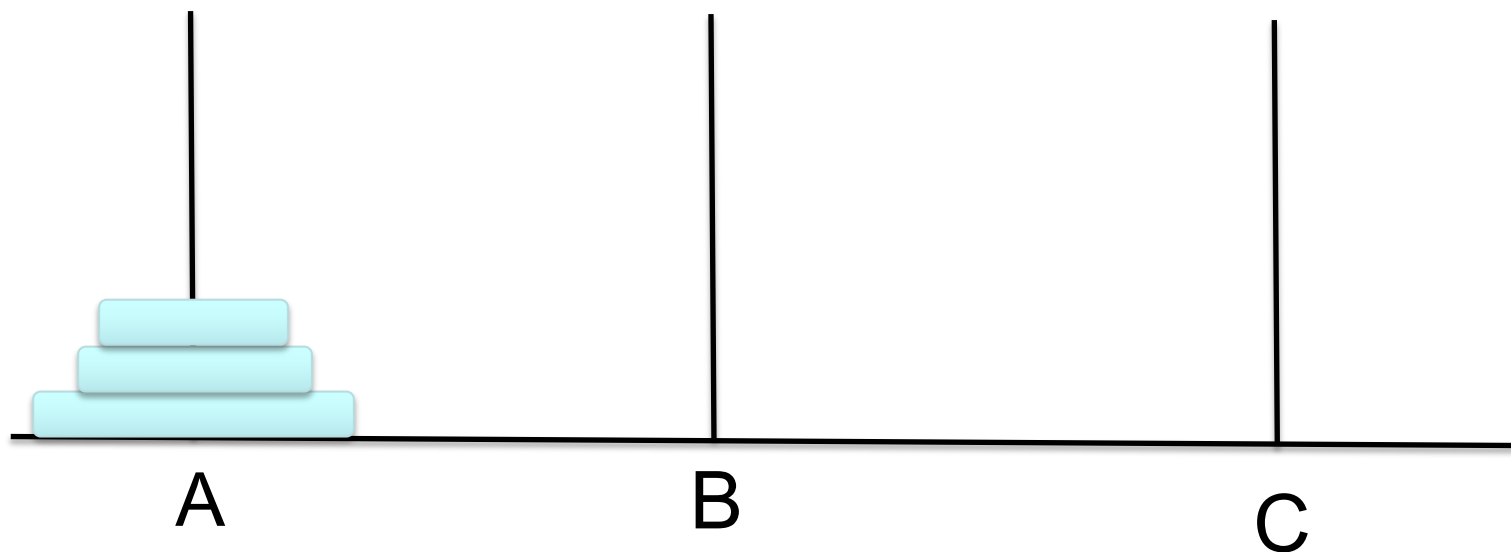
```
void move1();
```

Move two disks



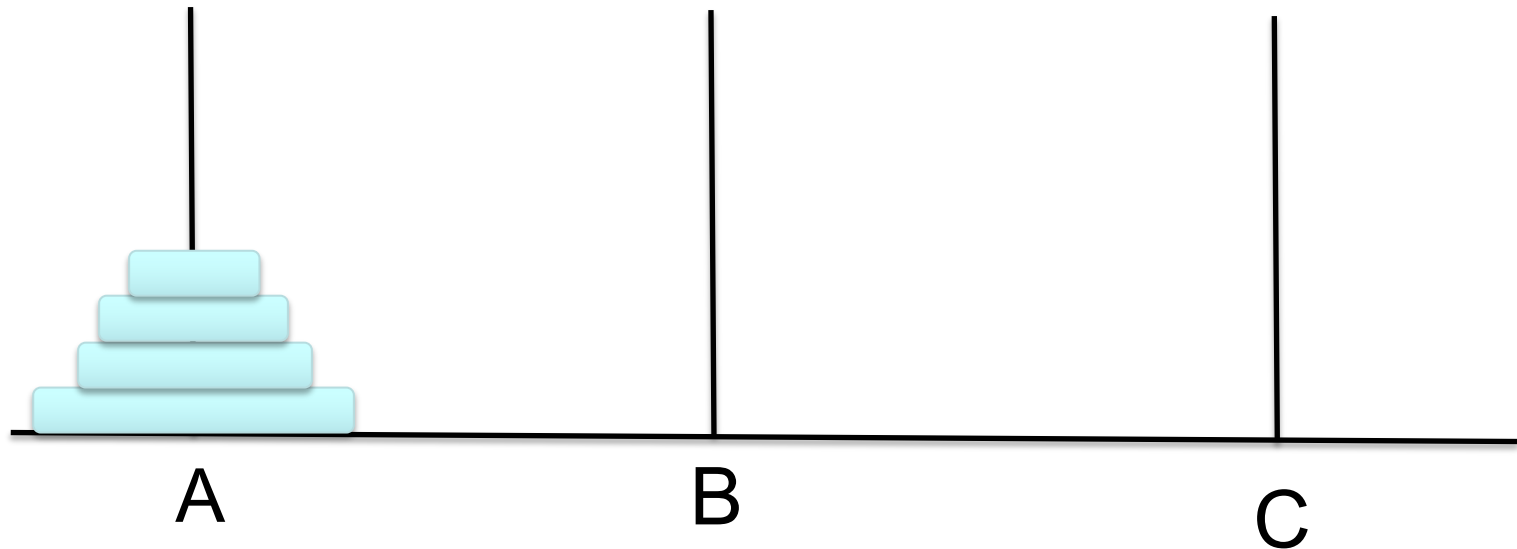
```
void move2 () {  
    move1 ();  
    move1 ();  
    move1 ();  
}
```

Move three disks



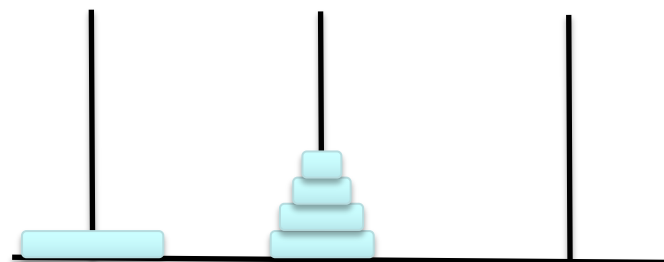
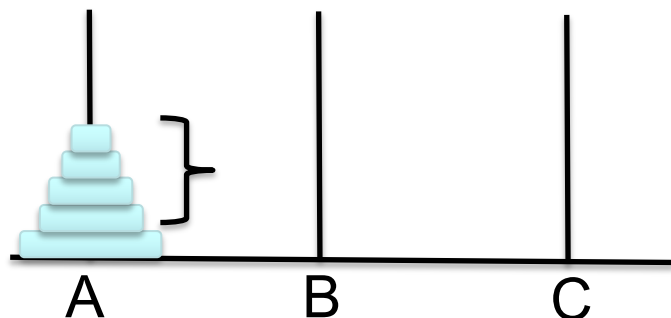
```
void move3 () {  
    move2 ();  
    move1 ();  
    move2 ();  
}
```

Move four disks

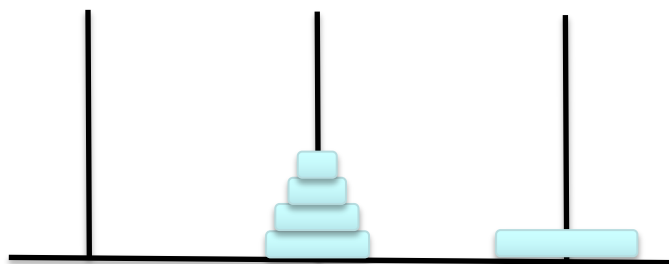


```
void move4 () {  
    move3 ();  
    move1 ();  
    move3 ();  
}
```

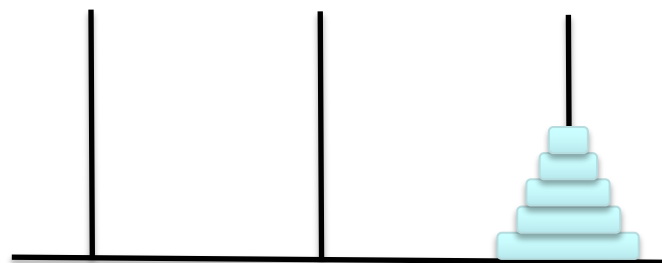
Move n disks



1. Move $n-1$ disks
2. from A \rightarrow B



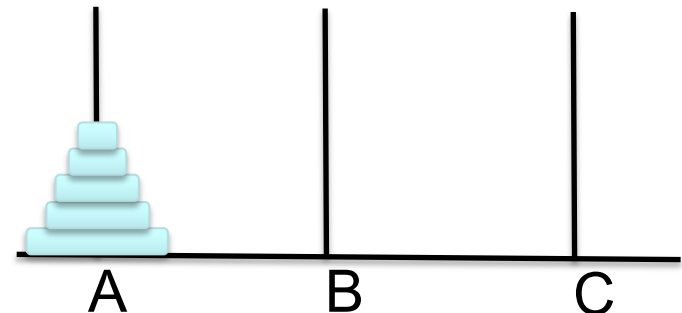
2. Move the largest disk to C



3. Move $n-1$ disks from B \rightarrow C

Towers of Hanoi: Recursive Solution

```
public class TowersOfHanoi {
    public static void solve(int n, String A, String B, String C) {
        if (n == 1) {
            System.out.println(A + " -> " + C);
        } else {
            solve(n - 1, A, C, B);
            System.out.println(A + " -> " + C);
            solve(n - 1, B, A, C);
        }
    }
    public static void main(String[] args) {
        int discs = 3;
        solve(discs, "A", "B", "C");
    }
}
```



Remarkable properties of recursive solution

- ▶ Takes 2^n steps to solve n disc problem.
- ▶ Takes 585 billion years for $n = 64$ (at rate of 1 disc per second).
- ▶ Reassuring fact: any solution takes at least this long

Quiz 1

It takes _____ steps to solve 8 disk problem.

- A. 64
- B. 128
- C. 256
- D. 512

Quiz 1

It takes _____ steps to solve 8 disk problem.

- A. 64
- B. 128
- C. 256**
- D. 512

Quiz 2:

What is the output of `fun(2)` ?

```
int fun(int n) {  
    if (n == 4)  
        return n;  
    else  
        return 2*fun(n+1);  
}
```

- A. 4
- B. 8
- C. 16
- D. Runtime Error

Quiz 2:

What is the output of `fun(2)` ?

```
int fun(int n) {  
    if (n == 4)  
        return n;  
    else  
        return 2*fun(n+1);  
}
```

- A. 4
- B. 8
- C. 16**
- D. Runtime Error

Quiz 3:

What is the output of fun(25)?

```
void fun(int n) {  
    if (n == 0)  
        return;  
    print(n%2);  
    fun(n/2);  
}
```

- A. 11001
- B. 10011
- C. 11111
- D. 00000

Quiz 3:

What is the output of fun(25)?

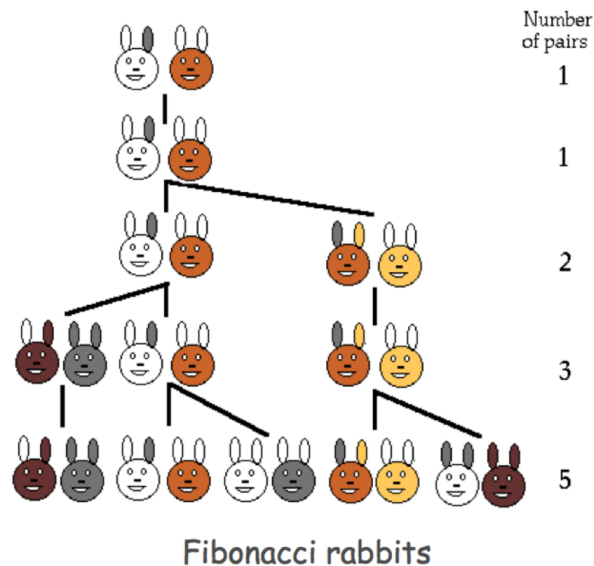
```
void fun(int n) {  
    if (n == 0)  
        return;  
    print(n%2);  
    fun(n/2);  
}
```

- A. 11001
- B. 10011**
- C. 11111
- D. 00000

Fibonacci Number

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

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



L. P. Fibonacci
(1170 - 1250)

Fibonacci Number

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

A natural for recursion?

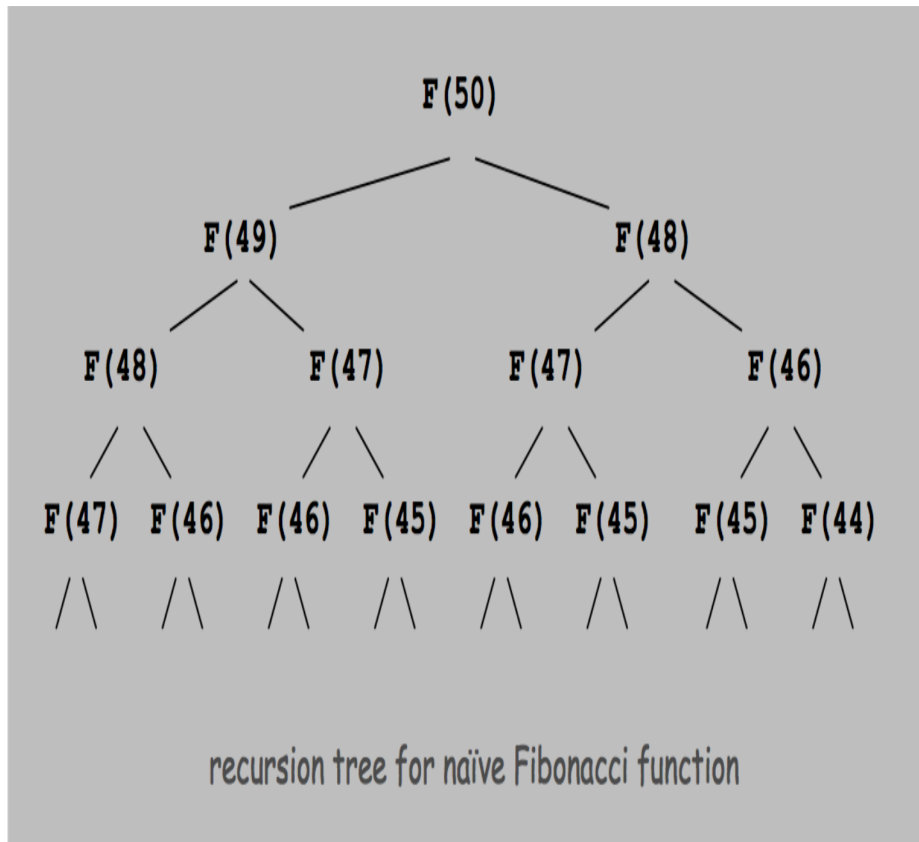
```
public static long F(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return F(n-1) + F(n-2);  
}
```

spectacularly inefficient code

Observation. It takes a really long time to compute $F(50)$.

Inefficient Recursion

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34



$F(50)$ is called once.

$F(49)$ is called once.

$F(48)$ is called 2 times.

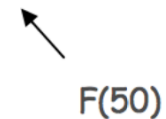
$F(47)$ is called 3 times.

$F(46)$ is called 5 times.

$F(45)$ is called 8 times.

...

$F(1)$ is called 12,586,269,025 times.

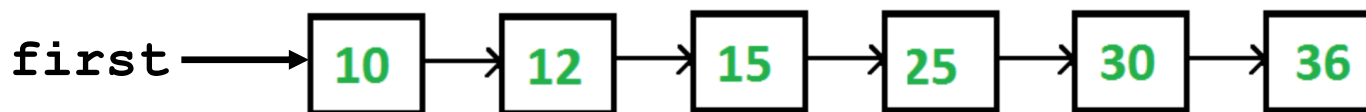


Memoized Version of the Fibonacci

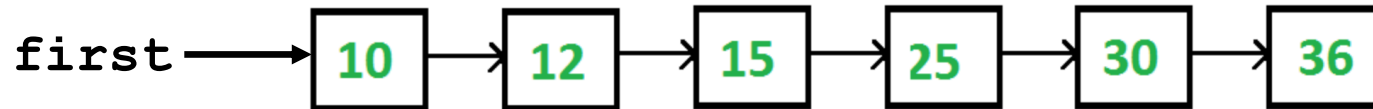
```
public Map<Integer, Integer> fibo;
public int fib(int n){
    int f1=0,f2=0;
    if( (n == 1) || (n == 2)) return 1;
    else{
        if(fibo.containsKey(n-1)) f1 = fibo.get(n-1);
        else{
            f1 = fib(n-1);
            fibo.put(n-1,f1);
        }
        if(fibo.containsKey(n-2)) f2 = fibo.get(n-2);
        else{
            f2 = fib(n-2);
            fibo.put(n-2,f2);
        }
        return f2+f1;
    }
}
```

Recursive Print a Linked List

```
public void print() {  
    print(first);  
    System.out.println("");  
}  
  
private void print(Node r) {  
    if(r == null) return;  
    System.out.print(r.data+", ");  
    print(r.next);  
}
```



Insert a Node into a Sorted List



```
public Node insert(Node r, E item) {
    if(r == null) return new Node(item);
    if(r.data.compareTo(item) < 0) {
        r.next = insert(r.next, item);
        return r;
    } else {
        Node t = new Node(item);
        t.next = r;
        return t;
    }
}
```

Contains Method

Iterative:

```
public boolean contains(E item) {
    Node<E> current = first;
    while(current != null) {
        if(current.data.equals(item)) { return true; }
        current = current.next;
    }
    return false;
}
```

Recursive:

```
public boolean contains_rec(Node r, E item) {
    if(r == null) return false;
    return (r.data.equals(item))
        || contains_rec(r.next, item);
}
```

Merge 2 Sorted Lists

```
public Node merge(Node list1, Node list2) {  
    if (list1 == null) return list2;  
    if (list2 == null) return list1;  
    if (list1.data < list2.data) {  
        list1.next = merge(list1.next, list2);  
        return list1;  
    } else {  
        list2.next = merge(list2.next, list1);  
        return list2;  
    }  
}
```

List1: 1→2→4→9

List2: 1→3→4→7→8→10

After merge

1→1→2→3→4→4→7→8→9→10