



Recursion

Overview

Recursion: a definition in terms of itself.

Recursion in algorithms:

- Natural approach to **some** (not all) problems
- A *recursive algorithm* uses itself to solve one or more smaller identical problems

Recursive Methods Must Eventually Terminate

*A recursive method must have
at least one base, or stopping, case.*

- A base case does not execute a recursive call
 - stops the recursion
- Each successive call to itself must be a "smaller version of itself"
 - an argument that describes a smaller problem
 - a base case is eventually reached

Key Components of a Recursive Algorithm Design

1. What is a smaller ***identical*** problem(s)?
 - Decomposition
2. How are the answers to smaller problems combined to form the answer to the larger problem?
 - Composition
3. Which is the smallest problem that can be solved easily (without further decomposition)?
 - Base/stopping case


Factorial ($N!$)

- $N! = (N-1)! * N$ [for $N > 1$]
- $1! = 1$
- $3!$
 - $= 2! * 3$
 - $= (1! * 2) * 3$
 - $= 1 * 2 * 3$
- Recursive design:
 - Decomposition: $(N-1)!$
 - Composition: $* N$
 - Base case: $1!$

factorial Method


```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; // composition
    else // base case
        fact = 1;

    return fact;
}
```



```
int factorial(int n)
{
    int fact;
    if (n > 1)
        fact = factorial(n-1) * n;
    else
        fact = 1;
    return fact;
}
```

***Go, change
the world***



```
int factorial(int 3)
```

```
{
```

```
    int fact;
```

```
    if (n > 1)
```

```
        fact = factorial(2) * 3;
```

```
    else
```

```
        fact = 1;
```

```
    return fact;
```

```
}
```



```
int factorial(int 2)
```

```
{
```

```
    int fact;
```

```
    if (n > 1)
```

```
        fact = factorial(1) * 2;
```


```
    else
```

```
        fact = 1;
```

```
    return fact;
```

```
}
```


***Go, change
the world***



```
int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

*Go, change
the world*

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



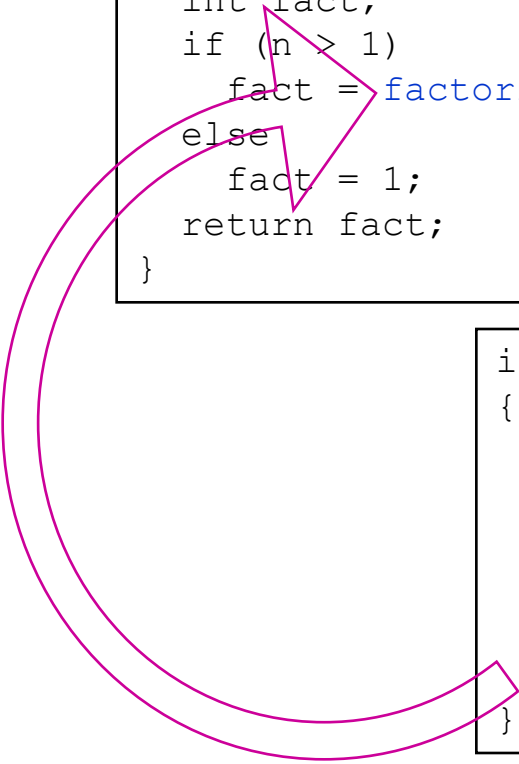
```
int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return fact;
}
```

*Go, change
the world*

```
int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

```
int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return 1;
}
```

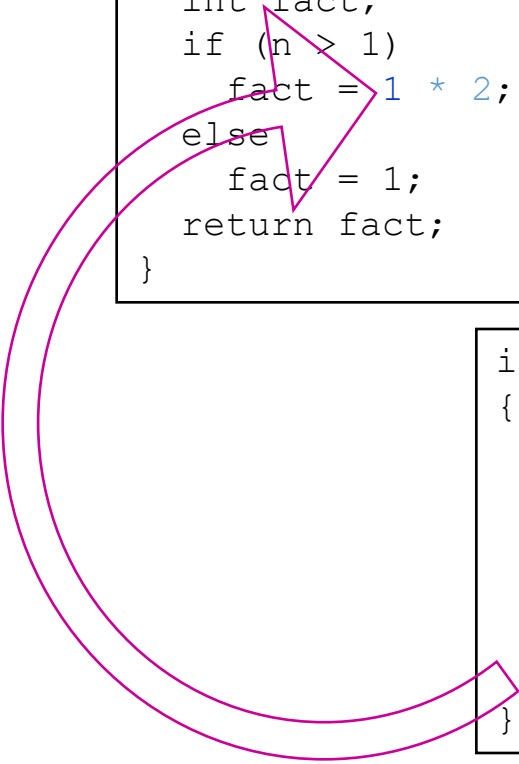


*Go, change
the world*

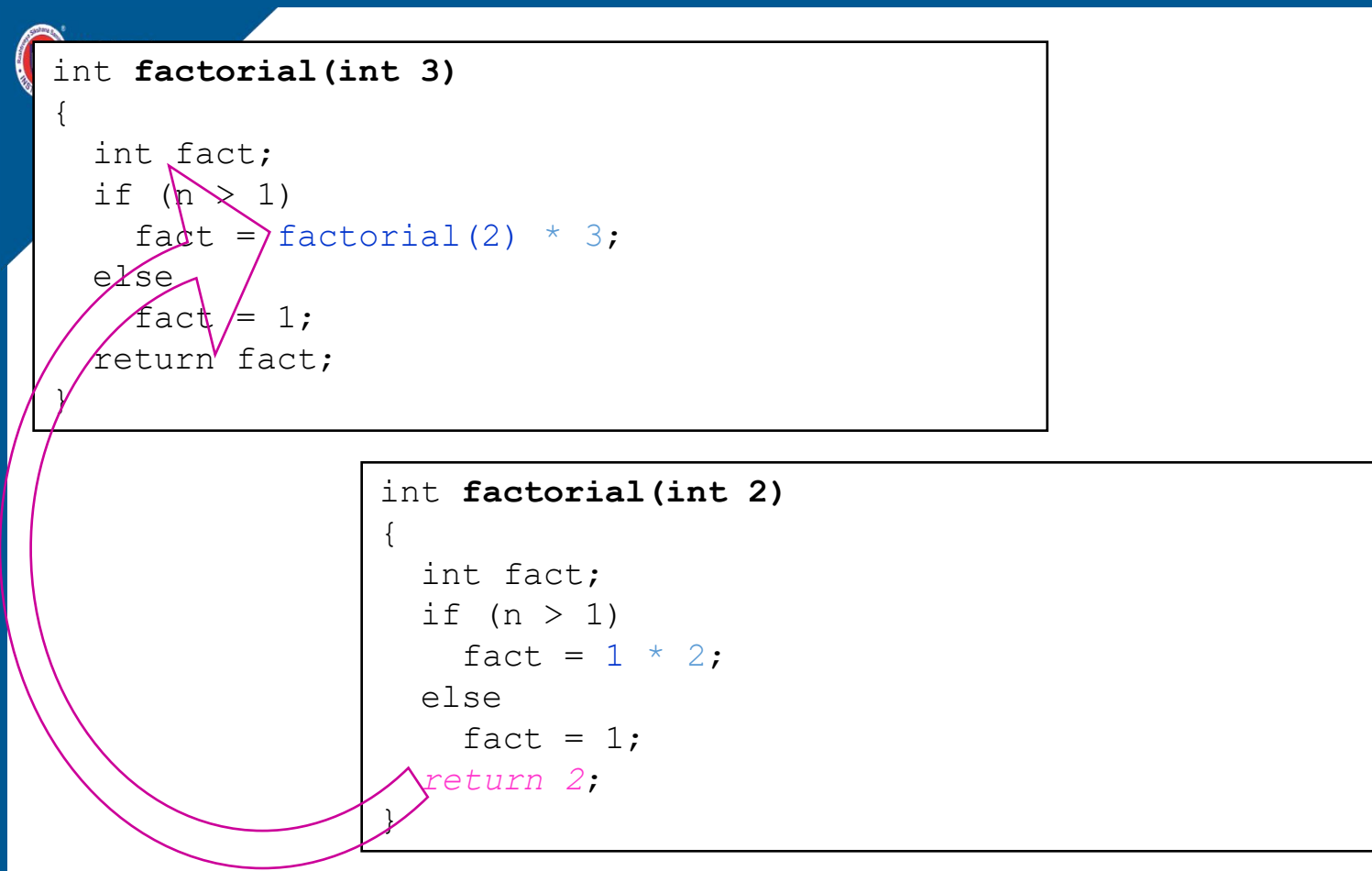
```
int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return fact;
}
```

```
int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return 1;
}
```



*Go, change
the world*

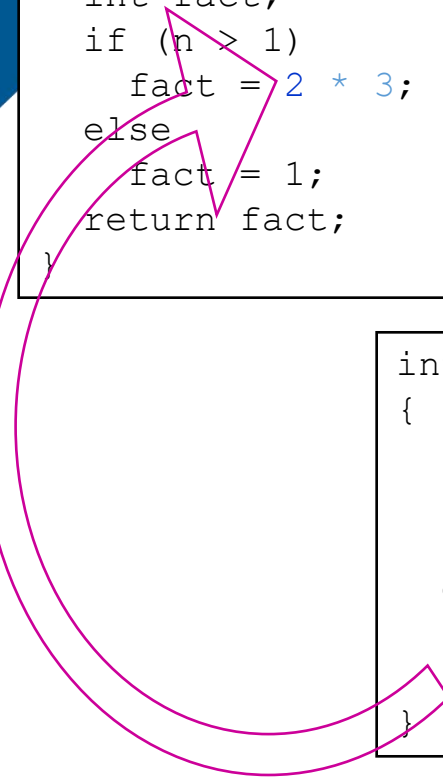


```
int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```


```
int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return 2;
}
```

*Go, change
the world*

```
int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return fact;
}
```



```
int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return 2;
}
```

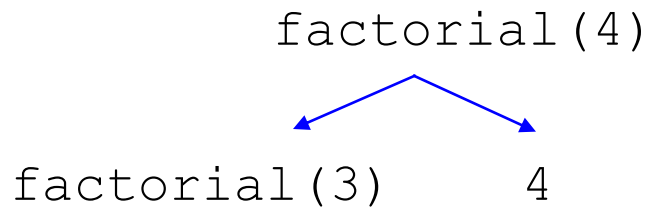


```
int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return 6;
}
```

***Go, change
the world***

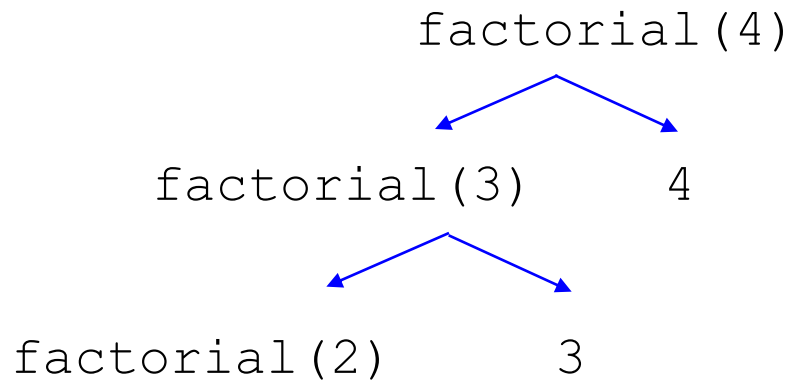
Execution Trace (decomposition)

```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



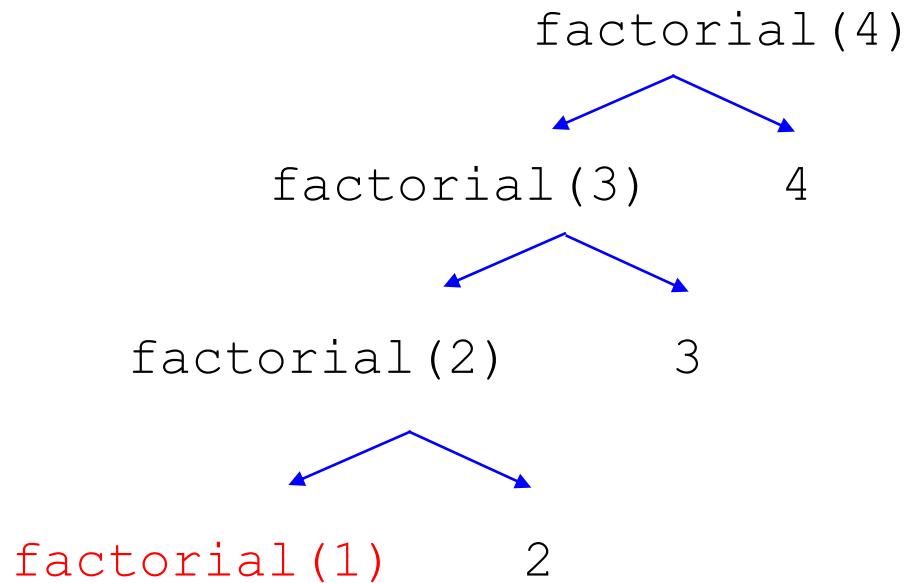
Execution Trace (decomposition)

```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



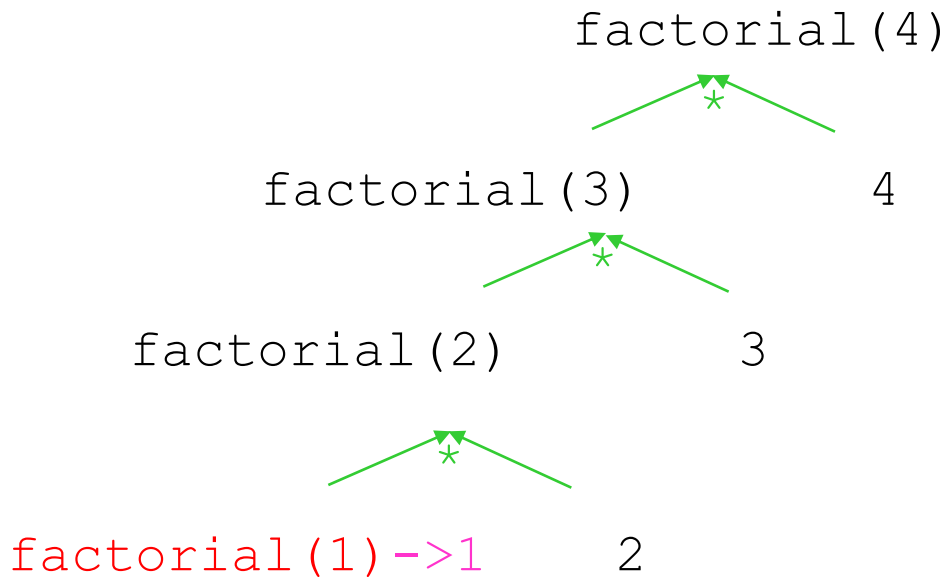
Execution Trace (decomposition)

```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



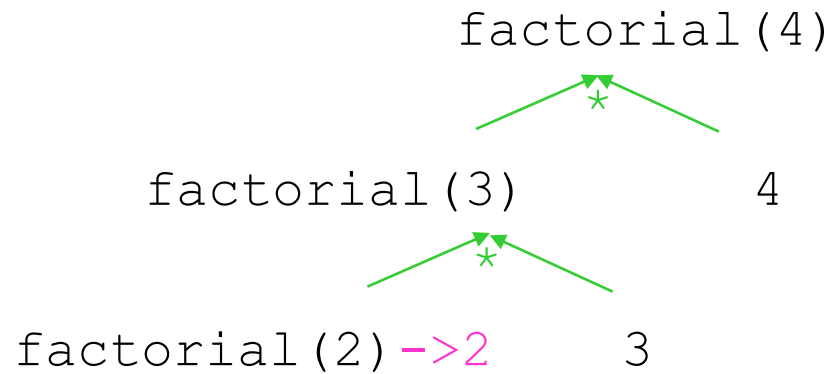
Execution Trace (composition)

```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



Execution Trace (composition)


```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



Execution Trace (composition)

```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```

factorial(4)



factorial(3) \rightarrow 6 4

Execution Trace (composition)

```
int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```

factorial(4) -> 24

Improved factorial Method

```
public static int factorial(int n)
{
    int fact=1;    // base case value

    if (n > 1)     // recursive case (decomposition)
        fact = factorial(n - 1) * n; // composition
    // else do nothing; base case

    return fact;
}
```

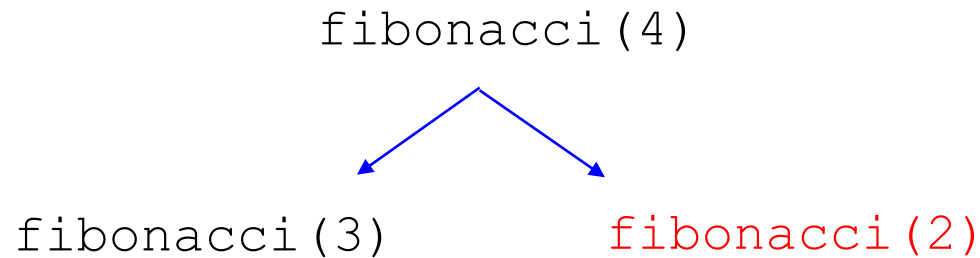
Fibonacci Numbers

- The N th Fibonacci number is the sum of the previous two Fibonacci numbers
- 0, 1, 1, 2, 3, 5, 8, 13, ...
- Recursive Design:
 - Decomposition & Composition
 - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$
 - Base case:
 - $\text{fibonacci}(1) = 0$
 - $\text{fibonacci}(2) = 1$

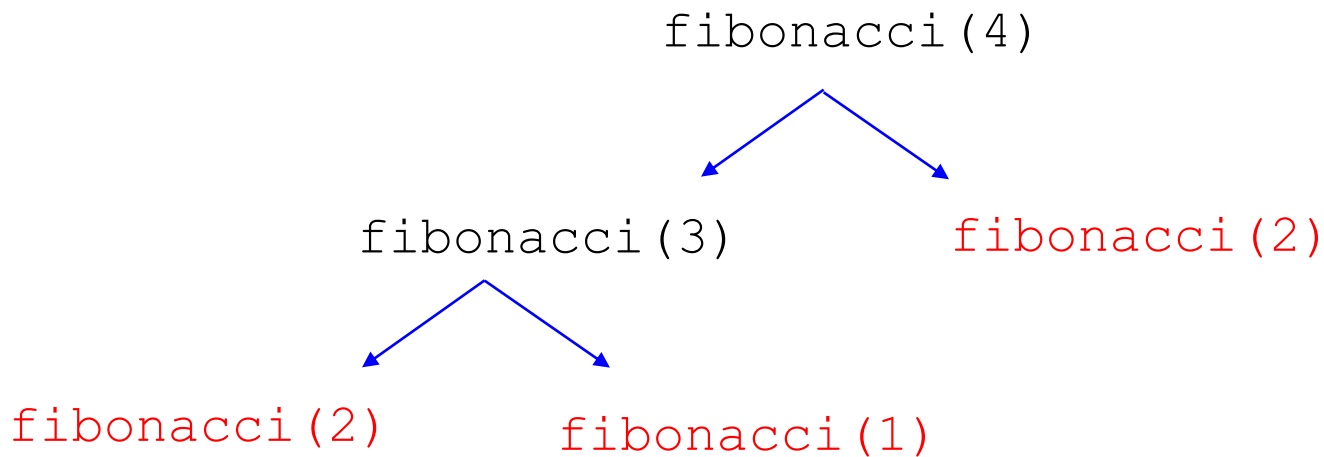
fibonacci Method

```
int fibonacci(int n)
{
    int fib;
    if (n == 1)
        return 0;
    if (n == 2)
        return 1;
    fib = fibonacci(n-1) + fibonacci(n-2);
    return fib;
}
```

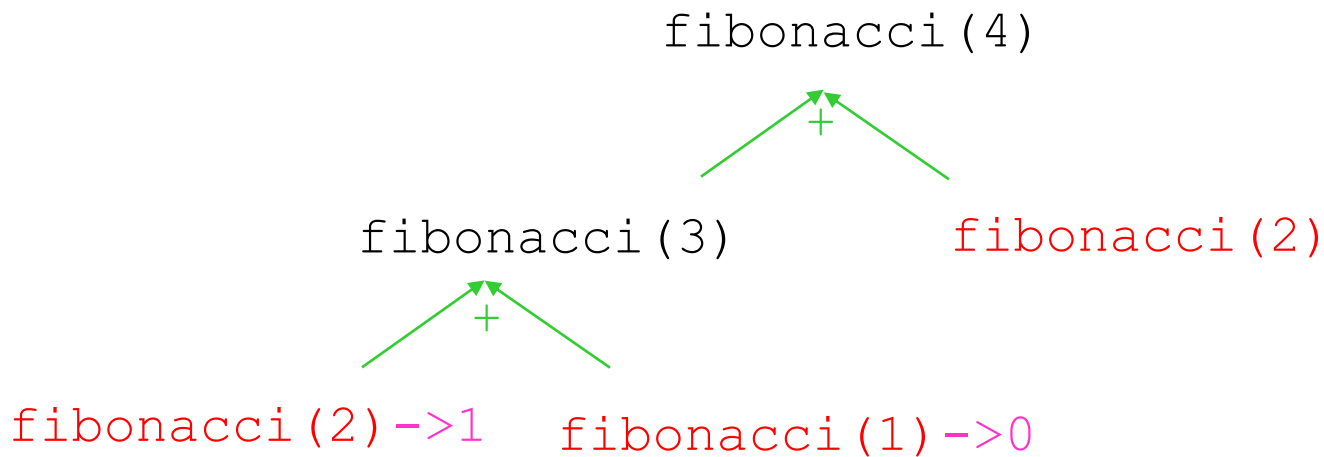

Execution Trace (decomposition)



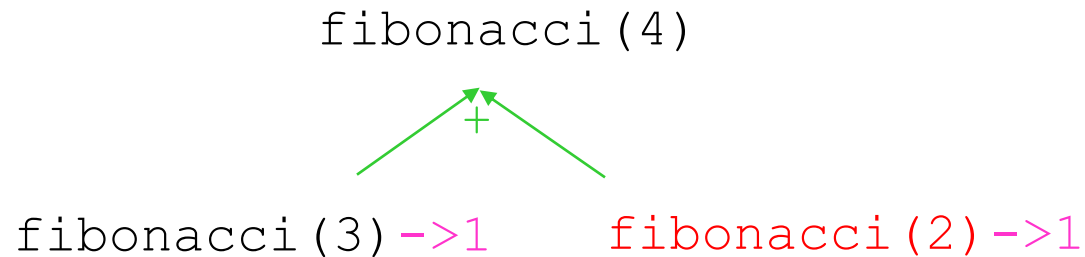
Execution Trace (decomposition)



Execution Trace (**composition**)



Execution Trace (**composition**)





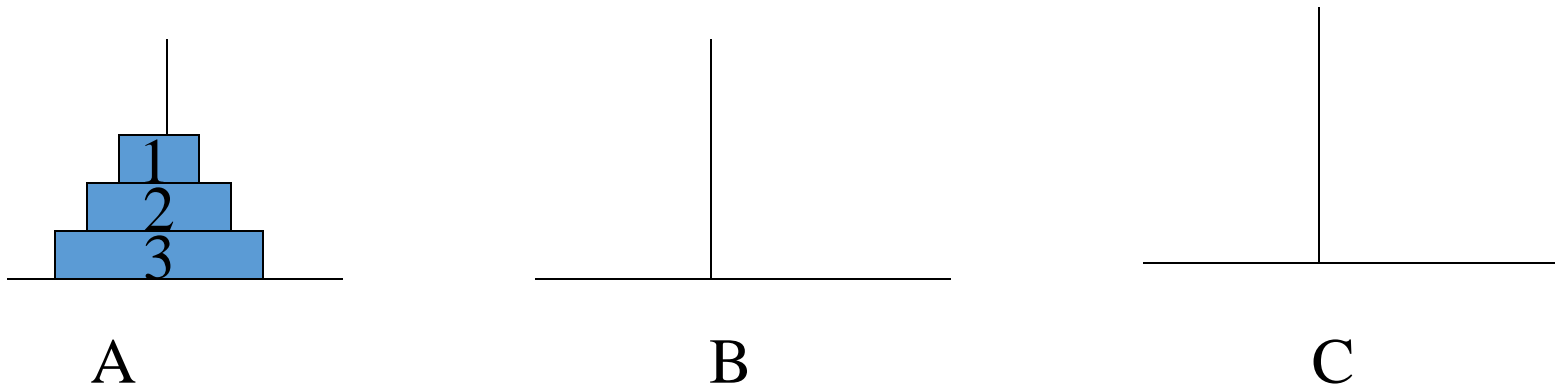
Execution Trace (**composition**)

`fibonacci (4) -> 2`

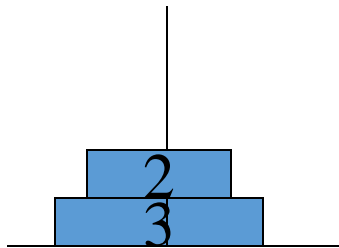
Example: Towers of Hanoi puzzle

*Go, change
the world*

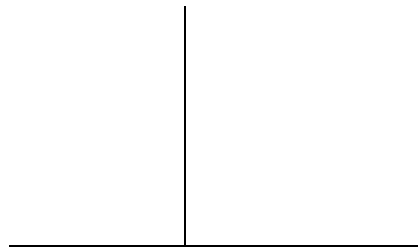
In this puzzle, the player begins with n disks of decreasing diameter placed one on top of the other on one of three pegs of the game board. The player must move the disks from peg to peg, using each of the three pegs, until the entire tower is moved from the starting peg to one of the others. The only rule governing the movement of the disks is that in each move a disk of larger diameter must never be placed on top of one of smaller diameter



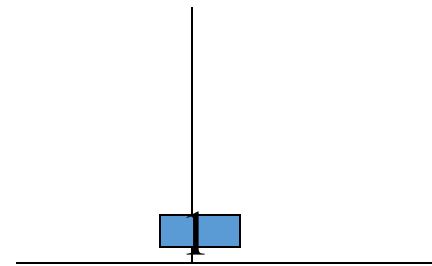
Towers of Hanoi Puzzle



A



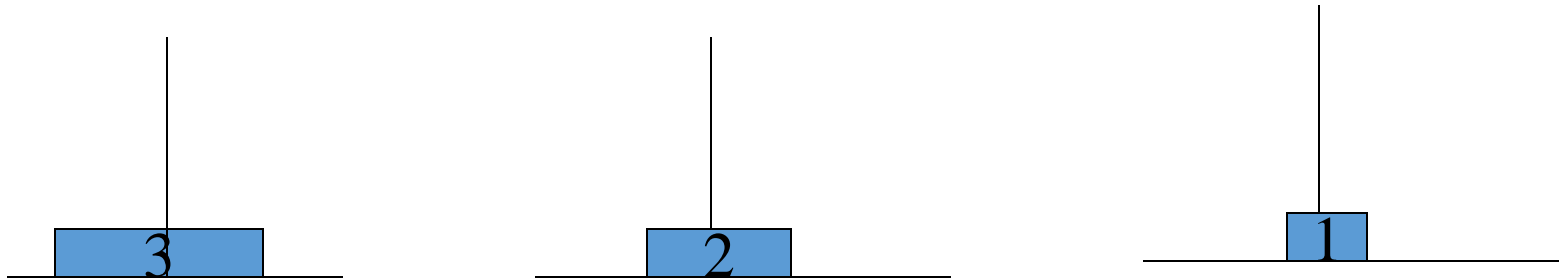
B



C

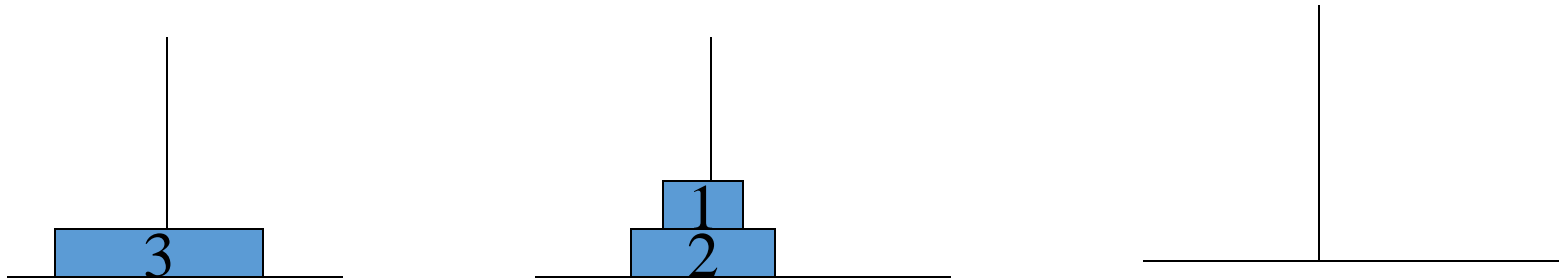


Towers of Hanoi Puzzle

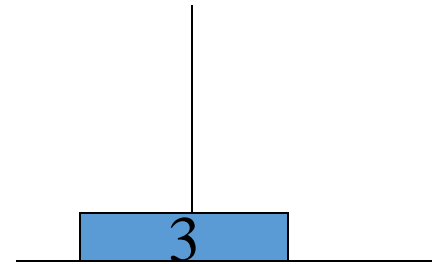
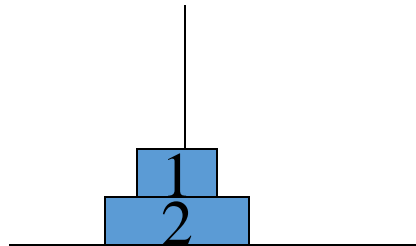
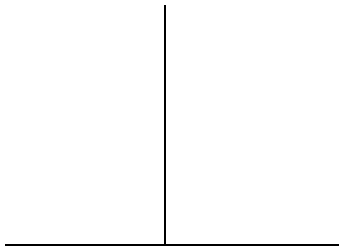




Towers of Hanoi Puzzle



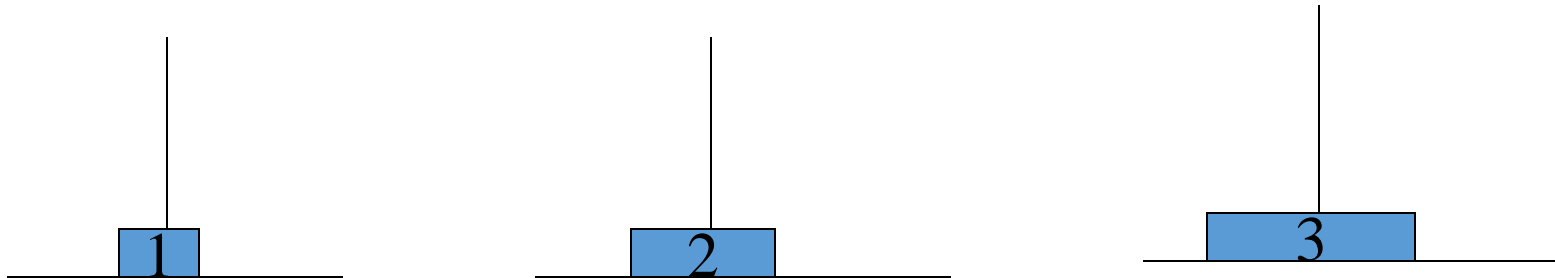
Towers of Hanoi Puzzle



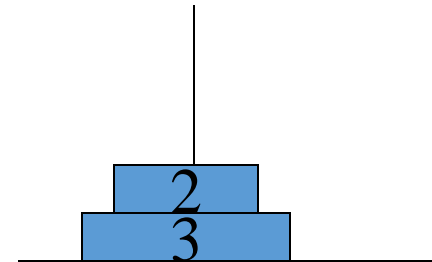
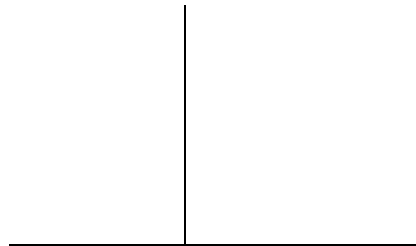
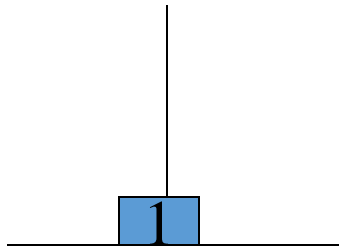


*Go, change
the world*

Towers of Hanoi Puzzle

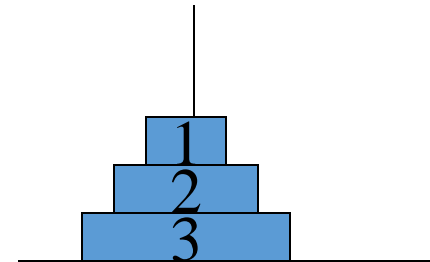
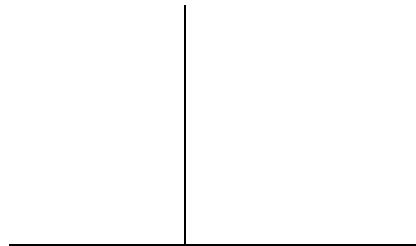
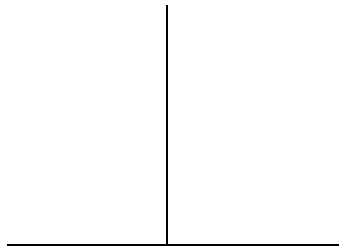


Towers of Hanoi Puzzle



*Go, change
the world*

Towers of Hanoi Puzzle



Towers of Hanoi Puzzle

A solution to the problem of moving a tower of size n from the source peg to the destination peg using a spare peg is found by moving a tower of size $n - 1$ from the source peg to the spare peg, moving the bottom disk from the source peg to the destination peg, and finally moving a tower of size $n - 1$ from the spare peg to the destination peg.

Towers of Hanoi Puzzle

Source -A Destination-C Spare /Temporary-B

Steps :

If($n > 0$)

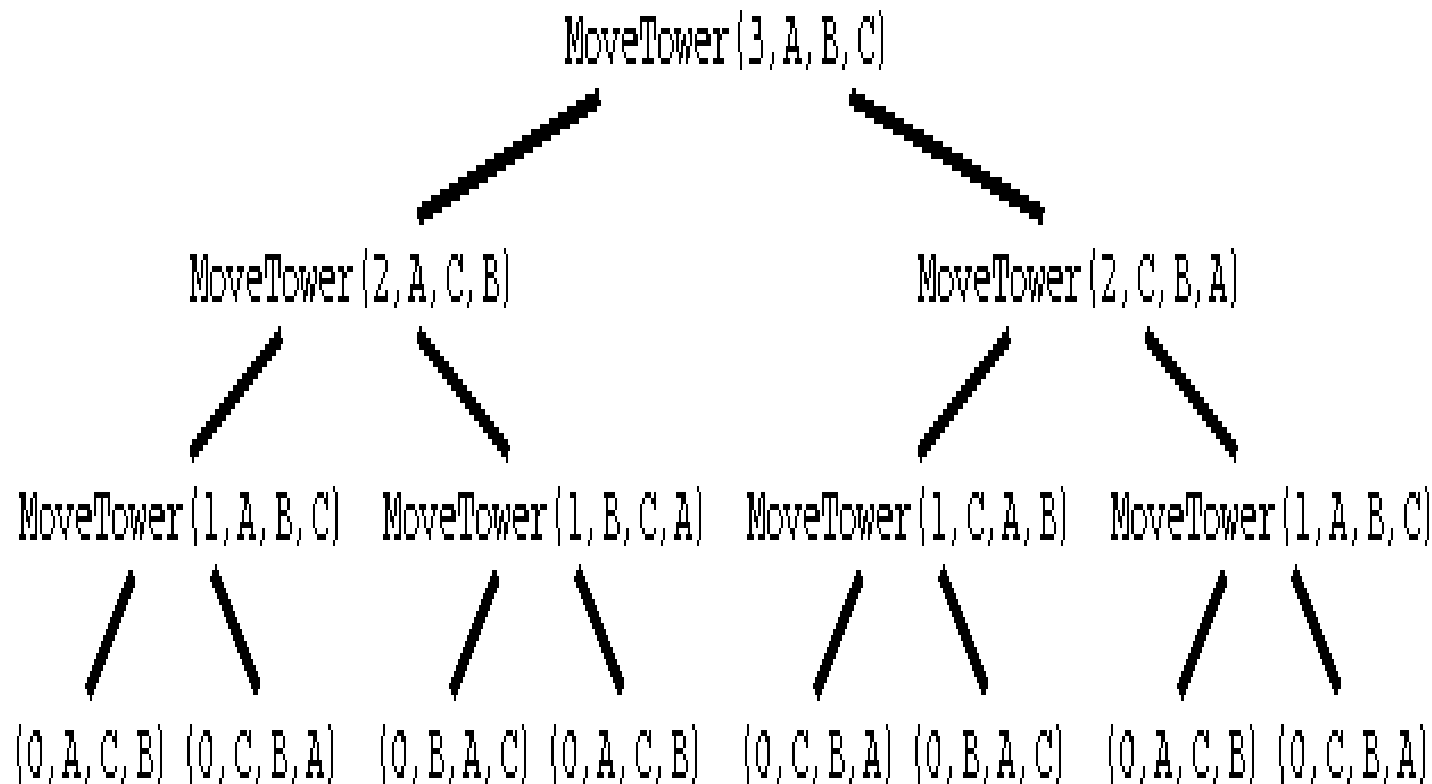
1. Move $n-1$ discs from A to B using C as spare/temporary.
(Recursive Step)
2. Transfer n^{th} ring from A to C using B as spare.
3. Move the $n-1$ discs from B to c using A as spare. (Recursive Step)

Towers of Hanoi Puzzle

```
void ToH(int n, char src , char dest, char temp)
{
    if(n >0)
    {
        ToH(n-1 , src, temp, dest);
        printf("\nMove disk %d  from %c  to %c",n,src,dest);
        ToH(n-1, temp, dest, src);
    }
}

int main()
{
    ToH(3,'A','C','B');
    return 0;
}
```


Towers of Hanoi Puzzle



Towers of Hanoi Puzzle

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

GCD (m, n)

```
int GCD(int m, int n)
{
    int ans;
    if (n == 0)
        return m;
    ans = GCD(n, m%n);
    return ans;
}
```

GCD (m, n)

■ Trace gcd(539, 84)

gcd(539, 84)



gcd(84, 35)



gcd(35, 14)



gcd(14, 7)



gcd(7, 0)



7

Sum of Natural Numbers

```
int Sum(int n)
{
    int s;
    if (n == 1)
        return 1;
    s= Sum(n - 1) + n;
    return s;
}
```

Binary Search

```
int binarySearch(int arr[], int left, int right, int x)
{
    int mid;
    if (right >= left)
    {
        mid = (left+right)/ 2;
        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, left, mid - 1, x);

        return binarySearch(arr, mid + 1, right, x);
    }
    return -1;
}
```

Recursive Display for an array

Sum of digits of n

Remember:

Key to Successful Recursion

- if-else statement (or some other branching statement)
- Some branches: recursive call
 - "smaller" arguments or solve "smaller" versions of the same task (*decomposition*)
 - Combine the results (*composition*) [if necessary]
- Other branches: no recursive calls
 - *stopping* cases or *base* cases

Warning: Infinite Recursion May Cause a Stack Overflow Error

- Infinite Recursion
 - Problem not getting smaller (no/bad decomposition)
 - Base case exists, but not reachable (bad base case and/or decomposition)
 - No base case
- *Stack*: keeps track of recursive calls
 - Method begins: add data onto the stack
 - Method ends: remove data from the stack
- Recursion never stops; stack eventually runs out of space
 - **Stack overflow error**

Number of Zeros in a Number

- Example: 2030 has 2 zeros
- If n has two or more digits
 - the **number of zeros** is the **number of zeros** in n with the last digit removed
 - plus an additional 1 if the last digit is zero
- Examples:
 - number of zeros in 20030 is number of zeros in 2003 plus 1
 - number of zeros in 20031 is number of zeros in 2003 plus 0

recursive

numberOfZeros Recursive Design

- numberOfZeros in the number N
- K = number of digits in N
- Decomposition:
 - numberOfZeros in the first $K - 1$ digits
 - Last digit
- Composition:
 - Add:
 - numberOfZeros in the first $K - 1$ digits
 - 1 if the last digit is zero
- Base case:
 - N has one digit ($K = 1$)

numberOfZeros method

```
int numberOfZeros(int n)
{
    int zeroCount;
    if (n==0)
        zeroCount = 1;
    else if (n < 10)    // and not 0
        zeroCount = 0;    // 0 for no zeros
    else if (n%10 == 0)
        zeroCount = numberOfZeros(n/10) + 1;
    else    // n%10 != 0
        zeroCount = numberOfZeros(n/10);
    return zeroCount;
}
```

**Which is
(are) the
base
case(s)?
Why?**

**Decompo
sition,
Why?**

**Compositi
on, why?**

Recursive Versus Iterative Methods

*All recursive algorithms/methods
can be rewritten without recursion.*

- *Iterative* methods use loops instead of recursion
- Iterative methods generally run faster and use less memory--less overhead in keeping track of method calls