



Olympiads School

Recursion

Bruce Nan

What is recursion?

- Recursion is a programming technique in which a **call** to a method appears in that method's **body**.

Basic Form :

```
void recurse ()  
{  
    recurse ();           //Function calls itself  
}
```

```
int main ()  
{  
    recurse ();           //Sets off the recursion  
}
```

4 Fundamental Rules :

1. Base Case : Always have at least one case that can be solved without recursion.
 2. Make Progress : Any recursive call must progress towards a base case.
 3. Always Believe : Always assume the recursive call works.
 4. Compound Interest Rule : Never duplicate work by solving the same instance of a problem in separate recursive calls.
-

A simple example

- Mathematical functions are sometimes defined in a less standard form.

$$f(x) = 2f(x-1) + x^2$$

$$f(0) = 0$$

```
int F(int x)
{
    if (x == 0)                /*base case*/
        return 0;
    else
        return 2*F(x-1) + x*x; /*recursive call*/
}
```

- In recursion, an important point is that recursive calls will keep on being made until a base case is reached.

Wrong Example

What's wrong with the following example?

```
int Bad (int N)
{
    if (N==0)
        return 0;
    else
        return Bad (N/3 +1) + N -1;
}
```

Recursive Functions – Exponentiation

- Exponentiation is a mathematical operation written as x^n involving two numbers the base x and the exponent n . When n is a positive integer exponentiation corresponds to repeated multiplication.

$$\text{power}(x, n) = \underbrace{x \cdot x \cdot x \cdot x \cdots x}_n$$

Q: Find a recursive definition for $\text{power}(x, n)$?

Recursive Functions – Exponentiation

INITIALIZATION: $\text{power}(x, 1) = x$

RECURSION: $\text{power}(x, n) = x \cdot x^{n-1}$

To compute the value of a recursive function e.g. 5^4 , one plugs into the recursive definition obtaining expressions involving lower and lower values of the function until arriving at the base case.

Coding Practice

- Use recursion to implement $\text{power}(x,n)$
- Use recursion to implement
 $\text{sum}(n) = 1 + 2 + 3 + \dots + n$

Recursive Functions

Factorial

- A simple example of a recursively defined function is the ***factorial*** function:

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots (n-2) \cdot (n-1) \cdot n$$

i.e., the product of the first n positive numbers (by convention, the product of nothing is 1, so that $0! = 1$).

Q: Find a recursive definition for $n!$

Recursive Functions

Factorial

A:INITIALIZATION: $0! = 1$

RECURSION: $n! = n \cdot (n-1)!$

To compute the value of a recursive function, e.g. $5!$, one plugs into the recursive definition obtaining expressions involving lower and lower values of the function, until arriving at the base case.

EG: $5! =$

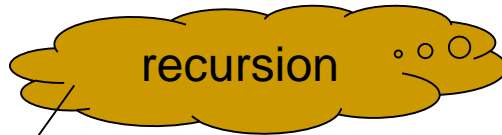
Recursive Functions

Factorial

A:INITIALIZATION: $0! = 1$

RECURSION: $n! = n \cdot (n-1)!$

To compute the value of a recursive function, e.g. $5!$, one plugs into the recursive definition obtaining expressions involving lower and lower values of the function, until arriving at the base case.



EG: $5! = 5 \cdot 4!$

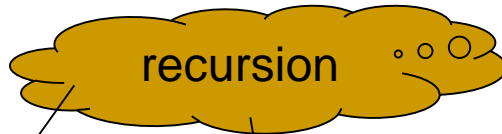
Recursive Functions

Factorial

A:INITIALIZATION: $0! = 1$

RECURSION: $n! = n \cdot (n-1)!$

To compute the value of a recursive function, e.g. $5!$, one plugs into the recursive definition obtaining expressions involving lower and lower values of the function, until arriving at the base case.



EG: $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3!$

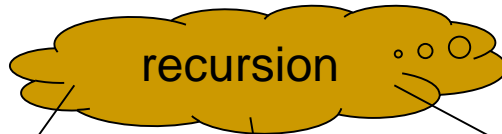
Recursive Functions

Factorial

A:INITIALIZATION: $0! = 1$

RECURSION: $n! = n \cdot (n-1)!$

To compute the value of a recursive function, e.g. $5!$, one plugs into the recursive definition obtaining expressions involving lower and lower values of the function, until arriving at the base case.



EG: $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2!$

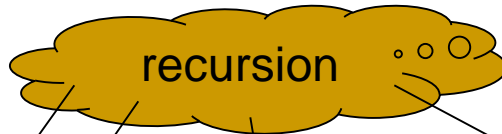
Recursive Functions

Factorial

A:INITIALIZATION: $0! = 1$

RECURSION: $n! = n \cdot (n-1)!$

To compute the value of a recursive function, e.g. $5!$, one plugs into the recursive definition obtaining expressions involving lower and lower values of the function, until arriving at the base case.



EG: $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2!$
 $= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1!$

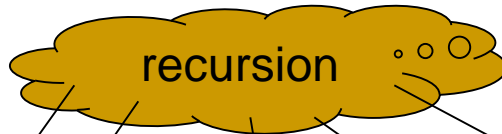
Recursive Functions

Factorial

A:INITIALIZATION: $0! = 1$

RECURSION: $n! = n \cdot (n-1)!$

To compute the value of a recursive function, e.g. $5!$, one plugs into the recursive definition obtaining expressions involving lower and lower values of the function, until arriving at the base case.



$$\begin{aligned} \text{EG: } 5! &= 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! \end{aligned}$$

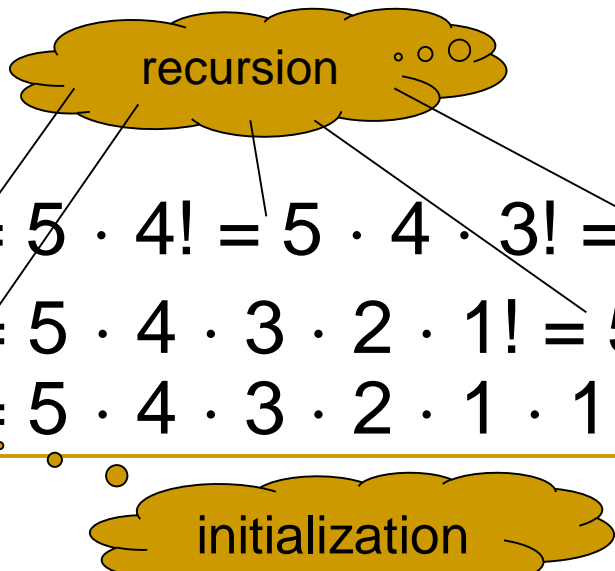
Recursive Functions

Factorial

A:INITIALIZATION: $0! = 1$

RECURSION: $n! = n \cdot (n-1)!$

To compute the value of a recursive function, e.g. $5!$, one plugs into the recursive definition obtaining expressions involving lower and lower values of the function, until arriving at the base case.

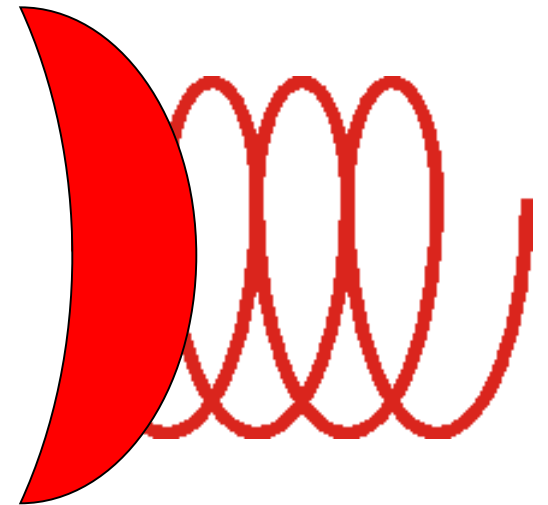


EG: $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2!$
 $= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0!$
 $= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$

Recursive Algorithms Computer Implementation

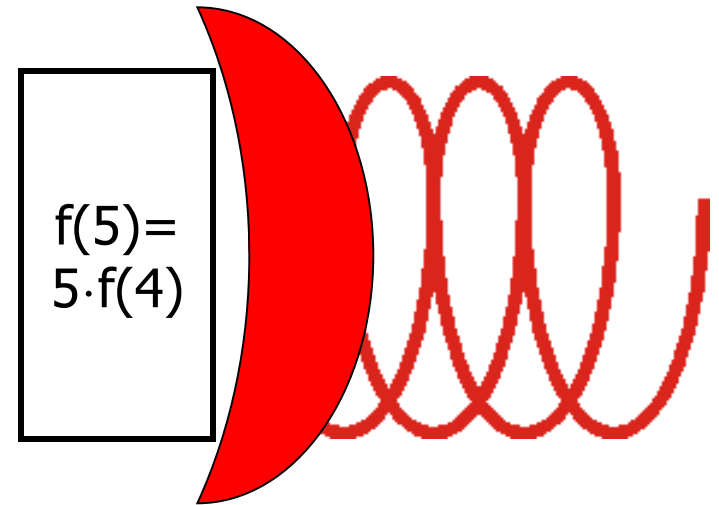
```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

Compute 5!



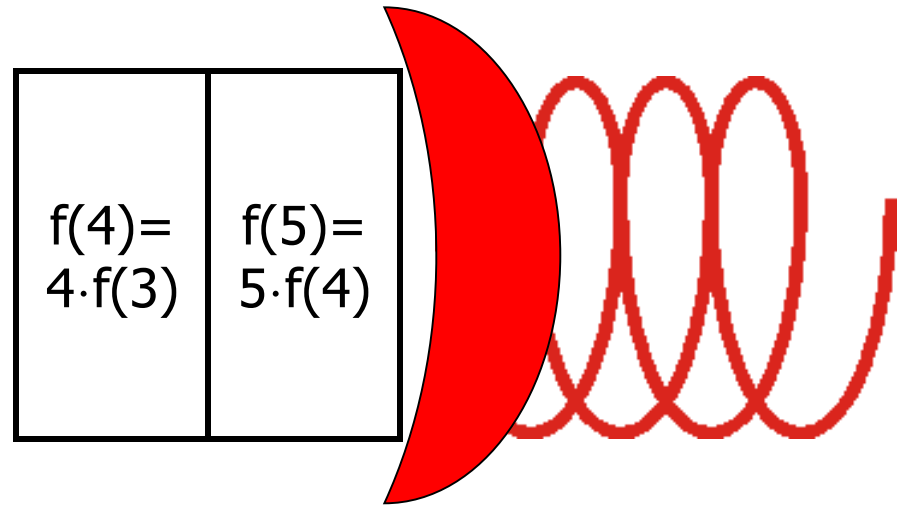
Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



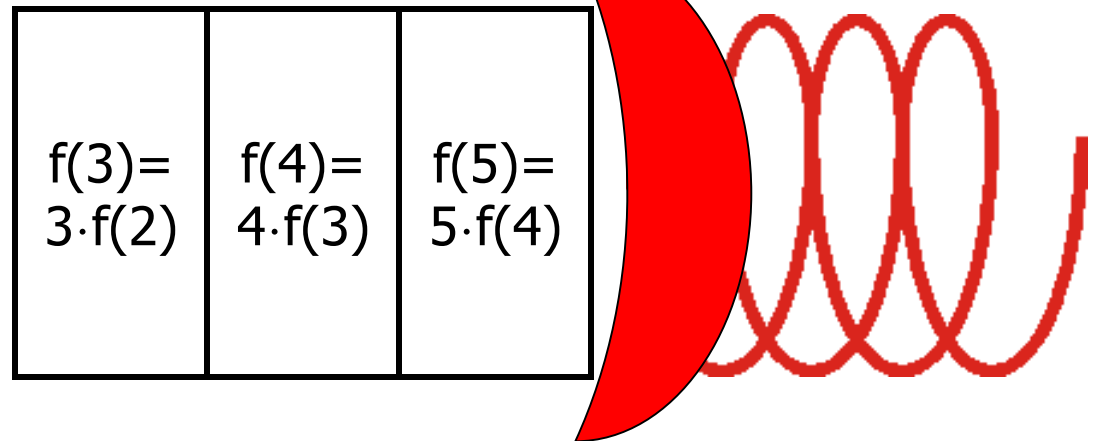
Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



Recursive Algorithms Computer Implementation

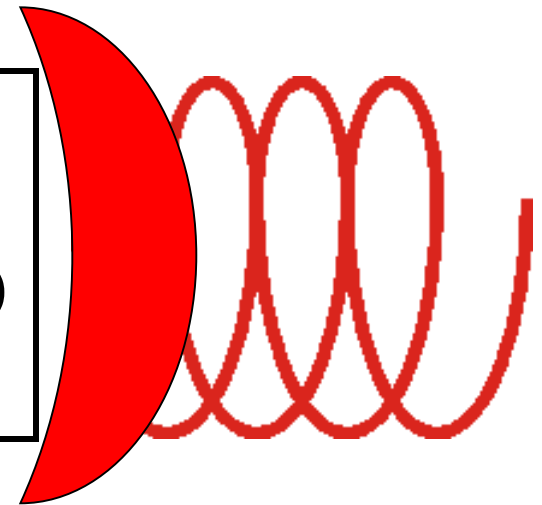
```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

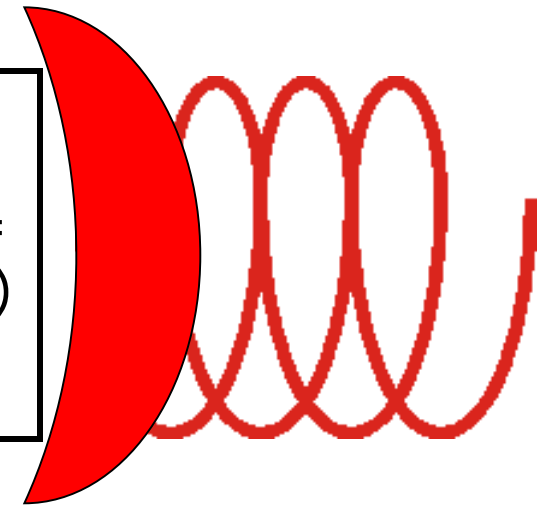
$f(2)=2 \cdot f(1)$	$f(3)=3 \cdot f(2)$	$f(4)=4 \cdot f(3)$	$f(5)=5 \cdot f(4)$
---------------------	---------------------	---------------------	---------------------



Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

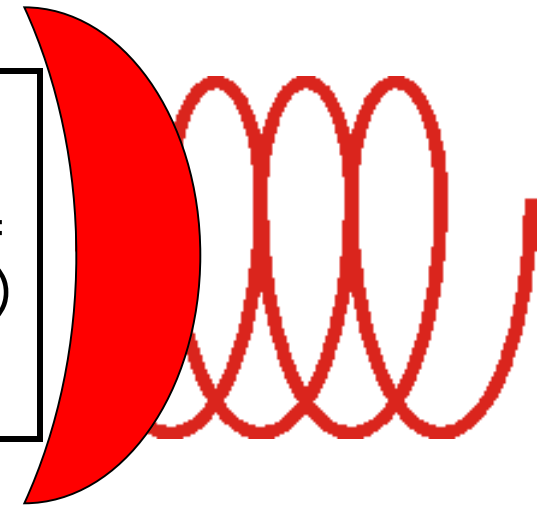
$f(1)=1 \cdot f(0)$	$f(2)=2 \cdot f(1)$	$f(3)=3 \cdot f(2)$	$f(4)=4 \cdot f(3)$	$f(5)=5 \cdot f(4)$
---------------------	---------------------	---------------------	---------------------	---------------------



Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

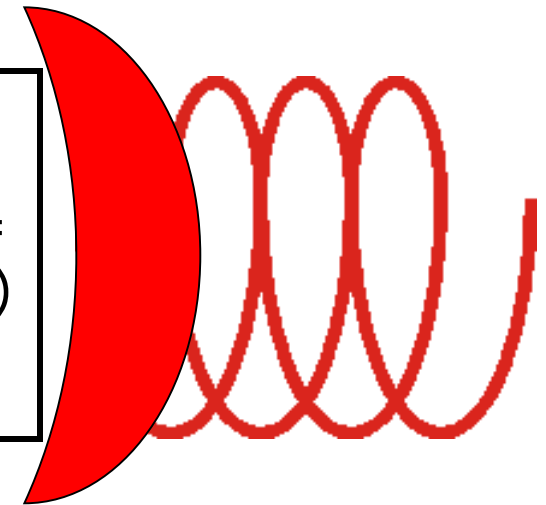
$f(0)=$ $1 \rightarrow$	$f(1)=$ $1 \cdot f(0)$	$f(2)=$ $2 \cdot f(1)$	$f(3)=$ $3 \cdot f(2)$	$f(4)=$ $4 \cdot f(3)$	$f(5)=$ $5 \cdot f(4)$
----------------------------	---------------------------	---------------------------	---------------------------	---------------------------	---------------------------



Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

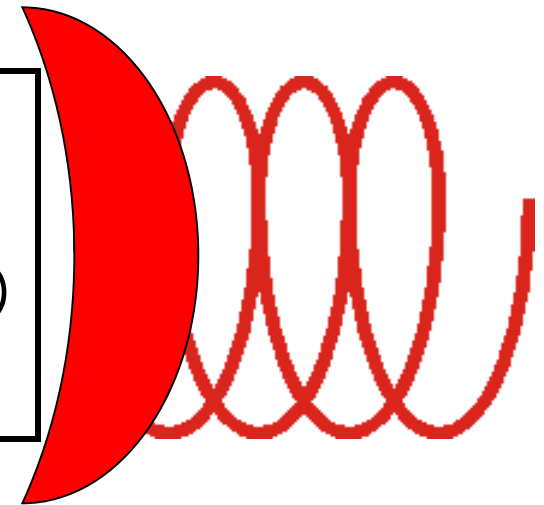
$1 \cdot 1 =$ $1 \rightarrow$	$f(2) =$ $2 \cdot f(1)$	$f(3) =$ $3 \cdot f(2)$	$f(4) =$ $4 \cdot f(3)$	$f(5) =$ $5 \cdot f(4)$
----------------------------------	----------------------------	----------------------------	----------------------------	----------------------------



Recursive Algorithms Computer Implementation

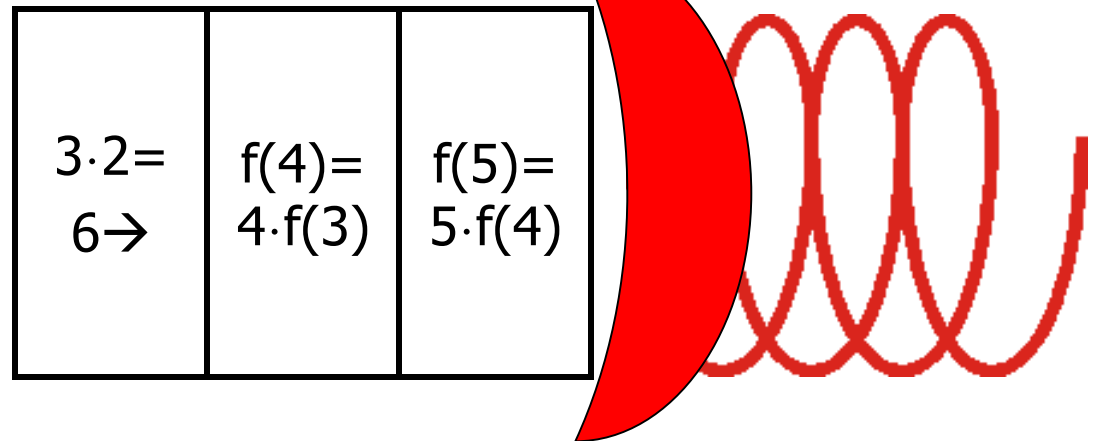
```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

$2 \cdot 1 =$ $2 \rightarrow$	$f(3) =$ $3 \cdot f(2)$	$f(4) =$ $4 \cdot f(3)$	$f(5) =$ $5 \cdot f(4)$
----------------------------------	----------------------------	----------------------------	----------------------------



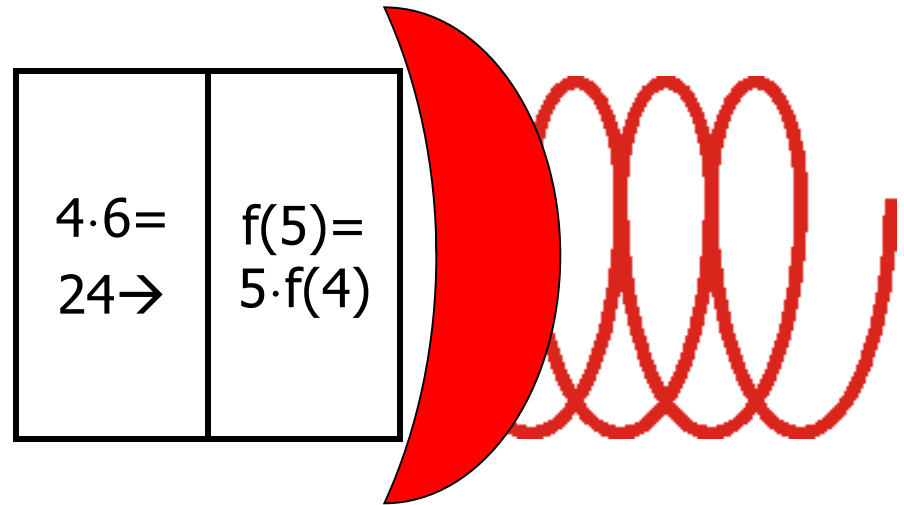
Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



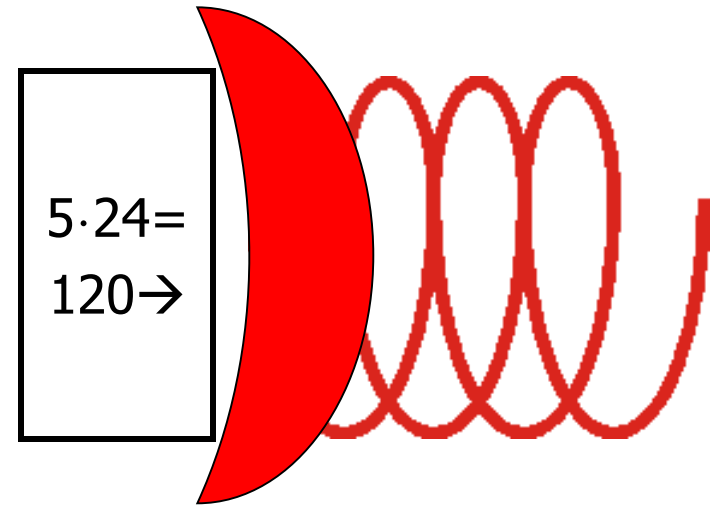
Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



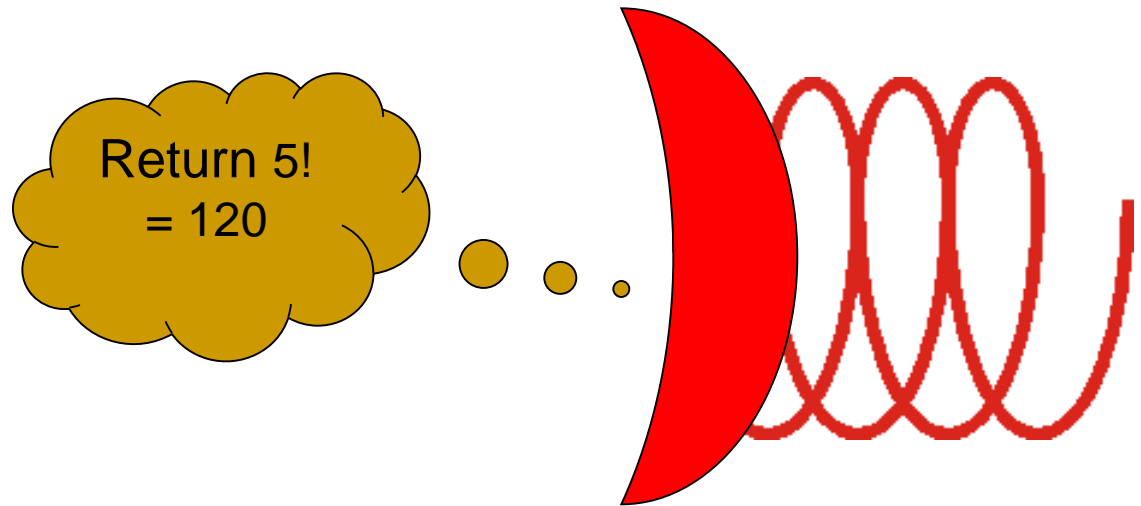
Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



Recursive Algorithms Computer Implementation

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



Practice

- Q: Find a recursive definition for the gcd function. Write a program to implement it.
- *Hint* : Euclid's algorithm.
gcd (m, n)
rem = m%n;
m = n;
n = rem;
until n <= 0
m contains the value of greatest common divisor

Recursive Algorithms

gcd

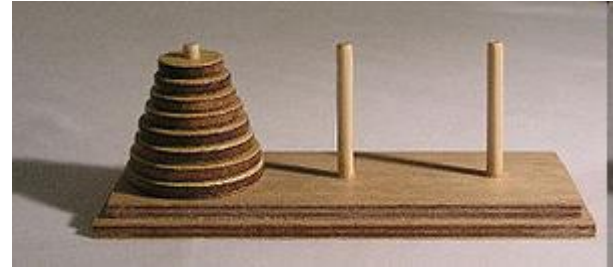
```
integer gcd (positive integer  $x$ , nonneg-integer  $y$ )  
{  
    if ( $y == 0$ ) return  $x$   
    return gcd( $y, x \% y$ )  
}
```

Fibonacci Problem

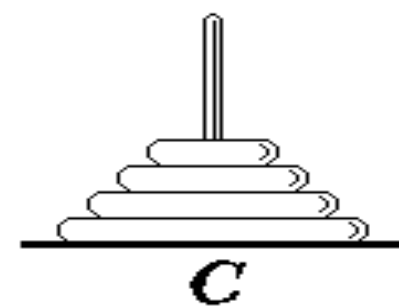
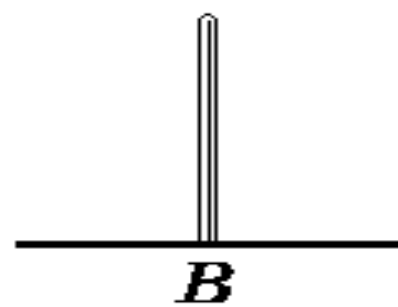
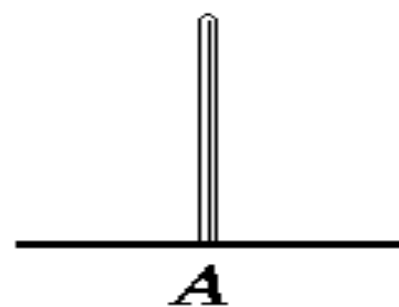
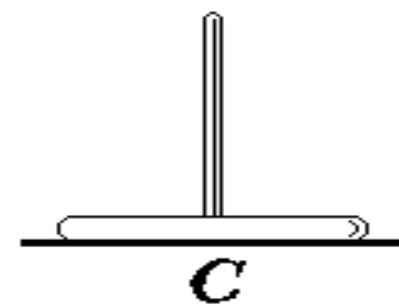
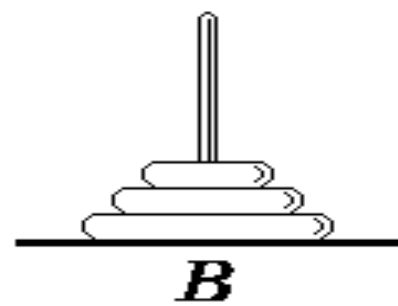
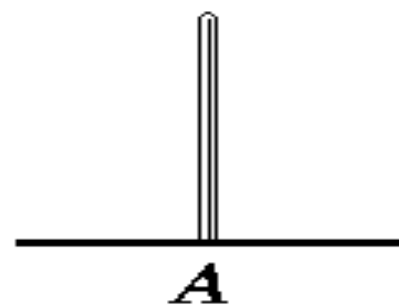
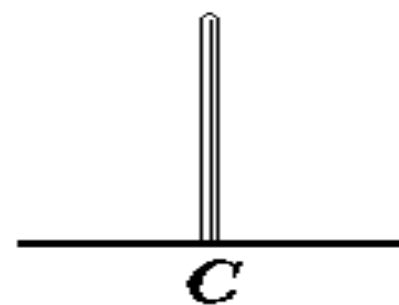
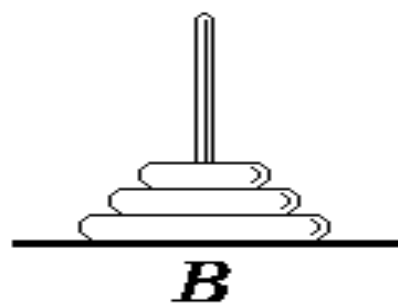
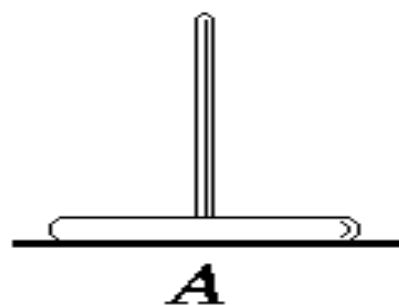
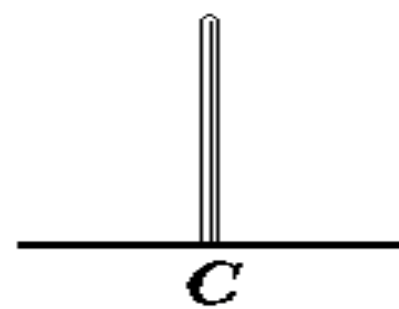
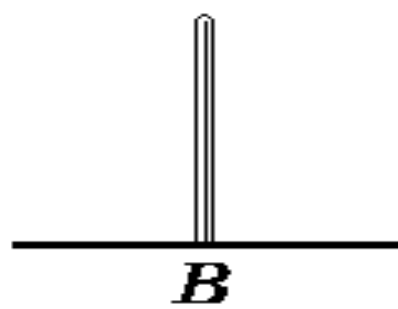
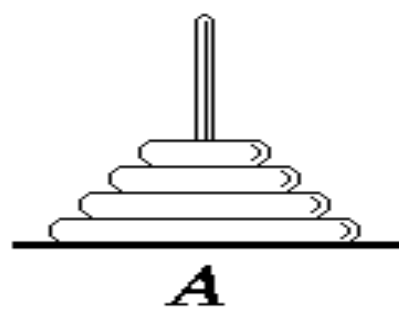


- In the year 1202, Fibonacci became interested in the reproduction of rabbits. He created an imaginary set of ideal conditions under which rabbits could breed, and posed the question, "How many pairs of rabbits will there be a year from now?" The ideal set of conditions was as follows:
 1. You begin with one male rabbit and one female rabbit. These rabbits have just been born.
 2. A rabbit will reach sexual maturity after one month.
 3. The gestation period of a rabbit is one month.
 4. Once it has reached sexual maturity, a female rabbit will give birth every month.
 5. A female rabbit will always give birth to one male rabbit and one female rabbit.
 6. Rabbits never die.
-

Tower of Hanoi



- The **Tower of Hanoi** is a mathematical game or puzzle.
- It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.
- The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:
 1. Only one disk may be moved at a time.
 2. Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
 3. No disk may be placed on top of a smaller disk.



```
void Hanoi (int n, char A, char B, char C)
{
    if (n==1)
    {
        move (A, C);
    }
    else
    {
        Hanoi (n-1, A, C, B);
        move (A, C);
        Hanoi(n-1, B, A, C);
    }
}
```

Four Notices

1. Base cases. You must always have some base cases, which can be solved without recursion.
 2. Making progress. For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.
 3. Design rule. Assume that all the recursive calls work.
 4. Compound interest rule. Never duplicate work by solving the same instance of a problem in separate recursive calls.
-

Thank You
