SE 2310
Data Structures and Algorithms

Lecture 3

Recursion

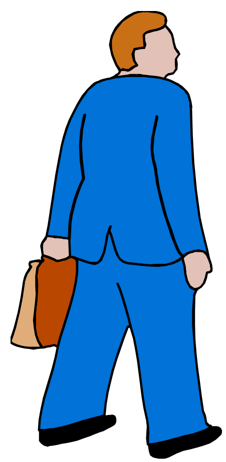
Introduction to Recursion

The Handshake Problem

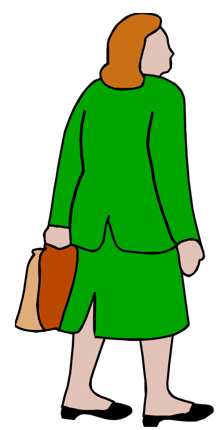
There are n people in a room. If each person shakes hands once with every other person. What is the total number of $h(n)$ of handshakes?

$h(n) = (n-1)+(n-2)+\ldots+1$

$h(4) = 3+2+1= 6$



$h(3)=2+1=3$



$h(2) = 1$

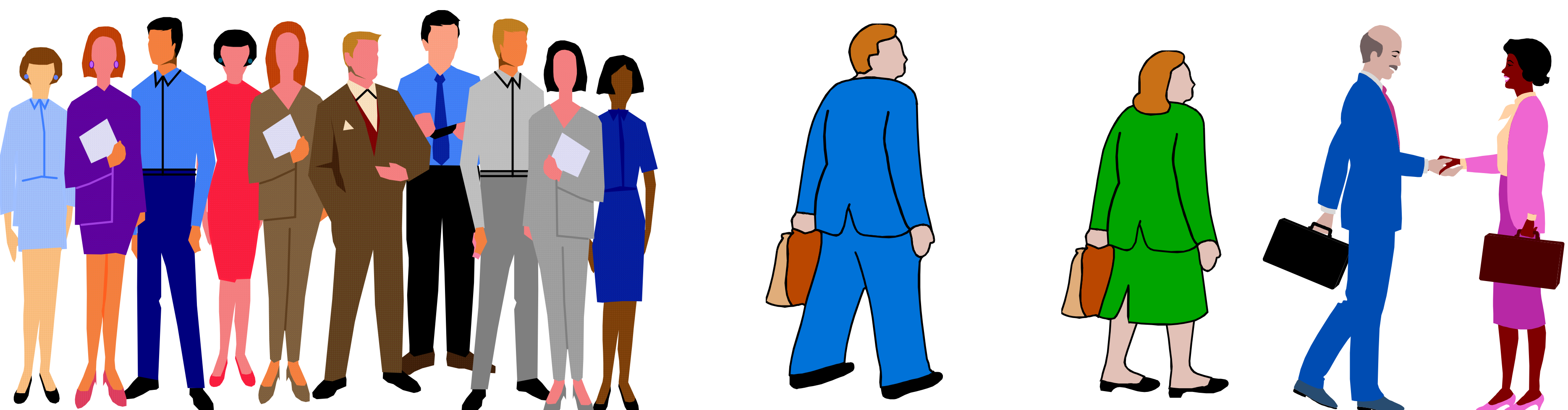
$h(n)$: Sum of integers from 1 to $n-1 = n(n-1) / 2$

Recursion

Recursion is an effective programming technique in which a function calls itself.

Many computational problems can be solved easily using recursion if you *think recursively*.

h(n) = h(n-1) + n-1 …. h(4) = h(3)+ 3=6 h(3)=h(2)+2=3 h(2) = 1



h(n): Sum of integers from 1 to n-1 = n(n-1) / 2

Base Case in Recursion

Each successive recursive call should bring the solution closer to a situation in which the answer is known.

A case for which the answer is known and can be expressed without recursion is called a base case.

Each recursive algorithm must have at least one base case, as well as the general recursive case.

Recursion – Some Examples

- Factorials
- Fibonacci numbers
- Triangular numbers
- Towers of Hanoi

Demonstrating Recursion with Factorials

$$n! = F(n)\ , \quad 0!=1!=1$$
$$= n(n\text{-}1)(n\text{-}2)\ldots(3)(2)(1)$$
$$n! = n\,F(n\text{-}1)$$

EX: 5!=5*4*3*2*1

Pseudocode

Factorial(n)

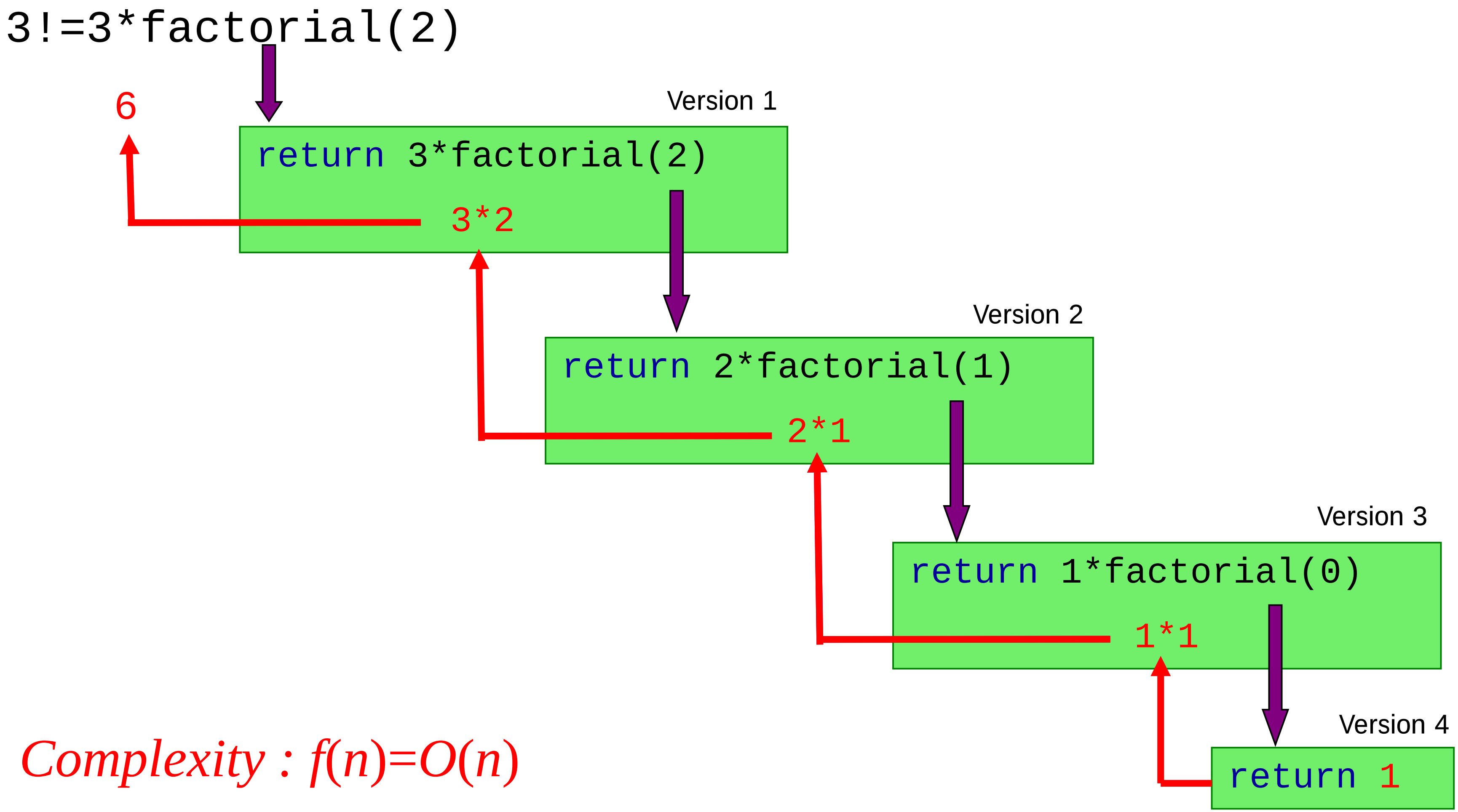
if n<1
    then return 1
else
    return n ***** Factorial(n-1)

C++

```
factorial(int n)
{
   if(n<1)
     return 1;
   else
     return n*factorial(n-1);
}
```

How Recursion Works?

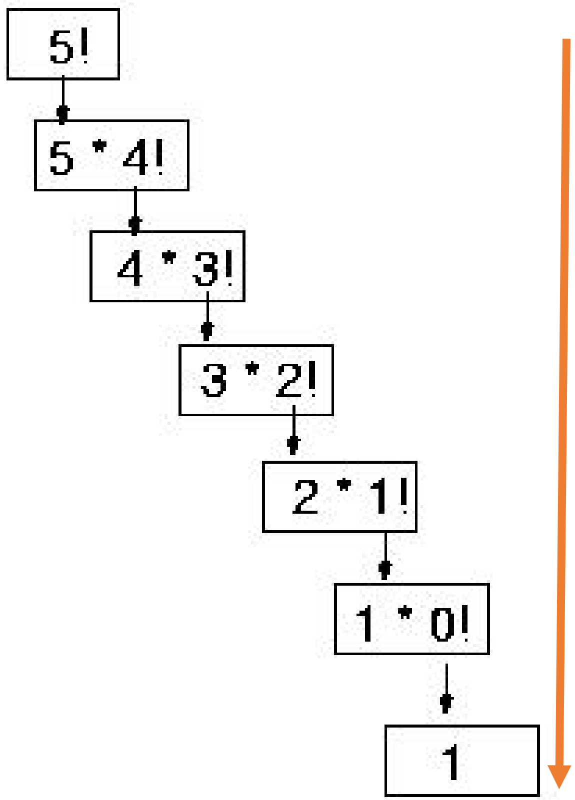
Example1 : Find 3!
There will be no computation until n=0.
(We assume that base case is: 0!=1)

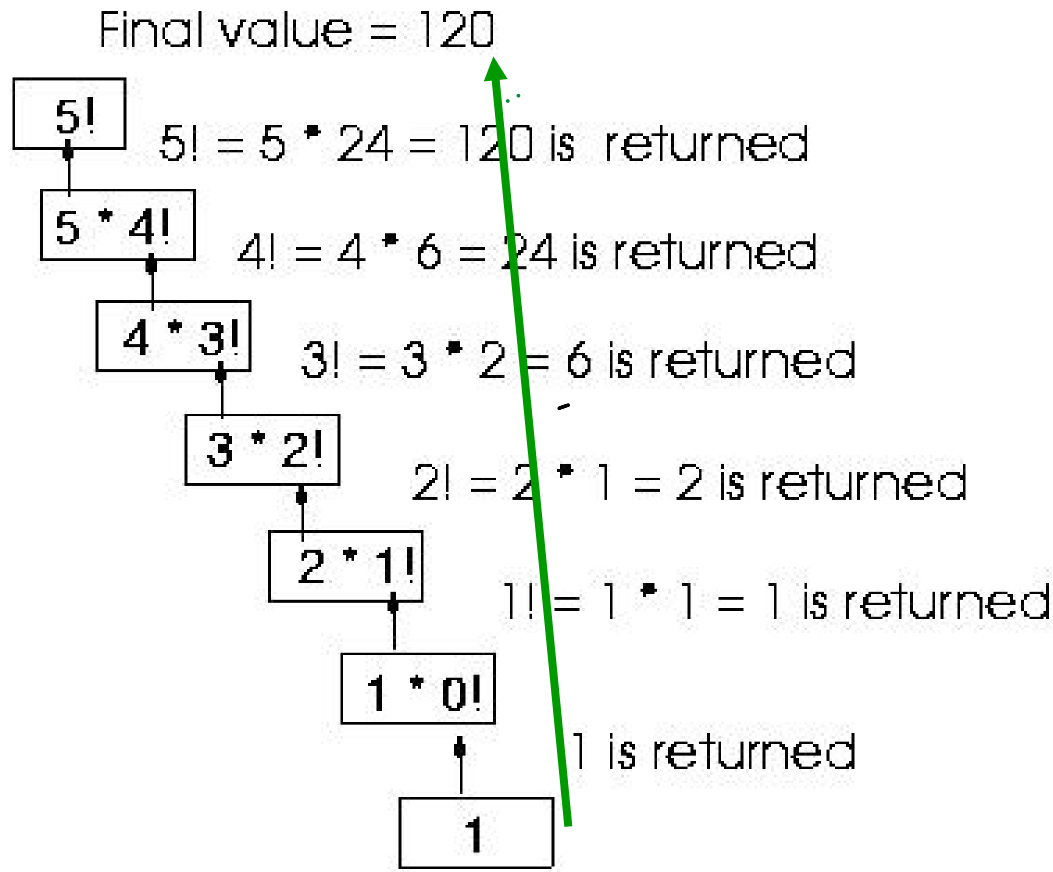


Complexity : f(n)=O(n)

How Recursion Works?

Example 2: n=5





No computation until base case!

Computations proceed upward

Recursions with Fibonacci Numbers

Example: Calculating $F_0=1, F_1=1, F_2=2, F_3=3, F_4=5, \ldots\ldots$ *Fibonacci* numbers. $\rightarrow F_i=F_{i-1}+F_{i-2}, i>1.$

Ex: 5=3+2, 8=5+3,....

Pseudocode

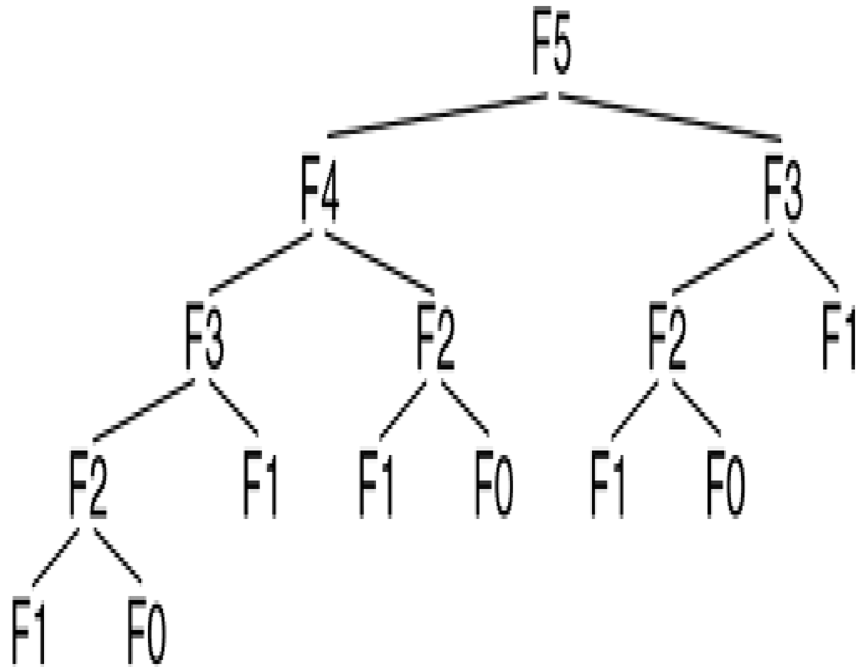
```
Fib(n)

  if(n≤=1)
    return 1
  else
    return Fib(n-1)+Fib(n-2)
```

C++

```
int Fib(int n)
{
  if(n<=1)
    return 1;
  else
    return Fib(n-1)+Fib(n-2);
}
```

Recursion Tree of Recursive Fib() Function



Complexity of Fibonacci

$T(n)$: Total run time
For $n = 1, 2$
$T(1) = 1$
$T(2) = 1$
For $n > 2$
$T(n) = T(n-1) + T(n-2)$

Complexity : $f(n)=O(2^n)$ Exponential time! Why?

Because there are too many unnecessary recursive calls.

Iterative Version of Fibonacci Function

Here we don't use recursion, instead we use a simple loop:

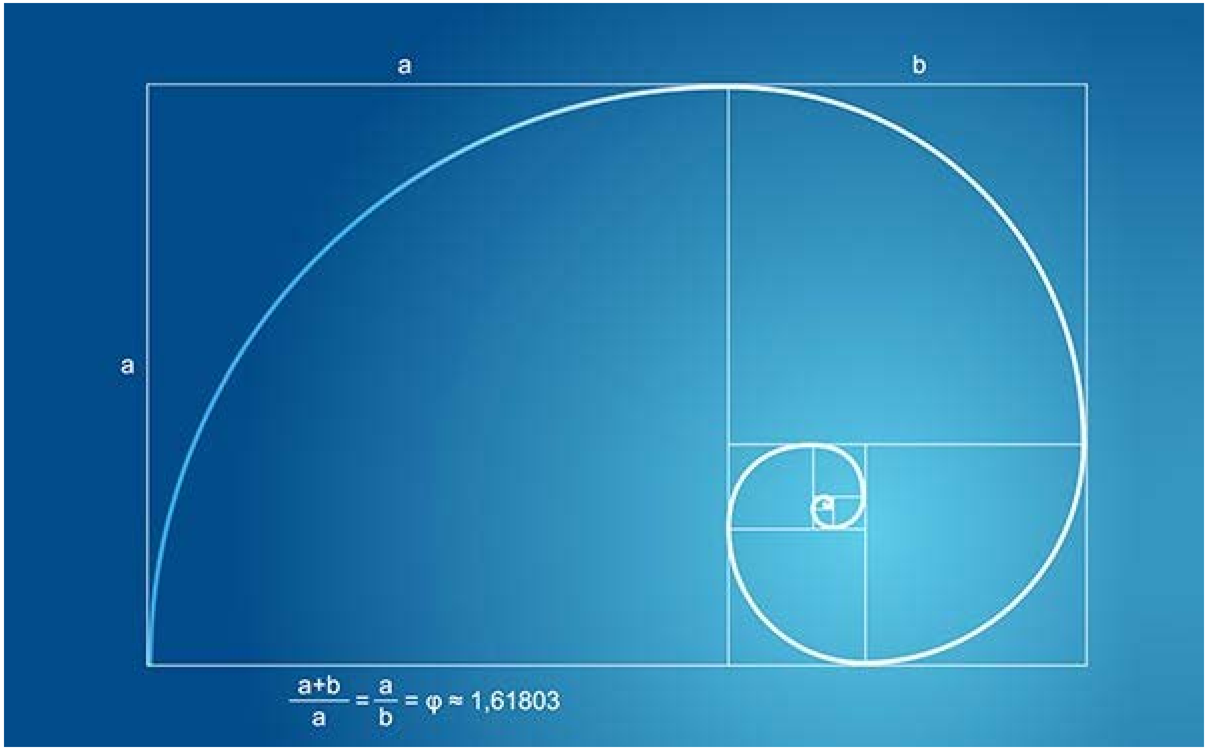
```
int fib(int n)
{
        int F[n+1];
        F[1] = F[2] = 1;
        for (int i = 3;  i <= n;  i++)
                F[i] = F[i-1] + F[i-2];
        return F[n]
}
```

Time Complexity: O(n).

The Golden Ratio

- The ratio of two consecutive Fibonacci Numbers approaches to 1.618
- This is called the Golden (or divine) Ratio.

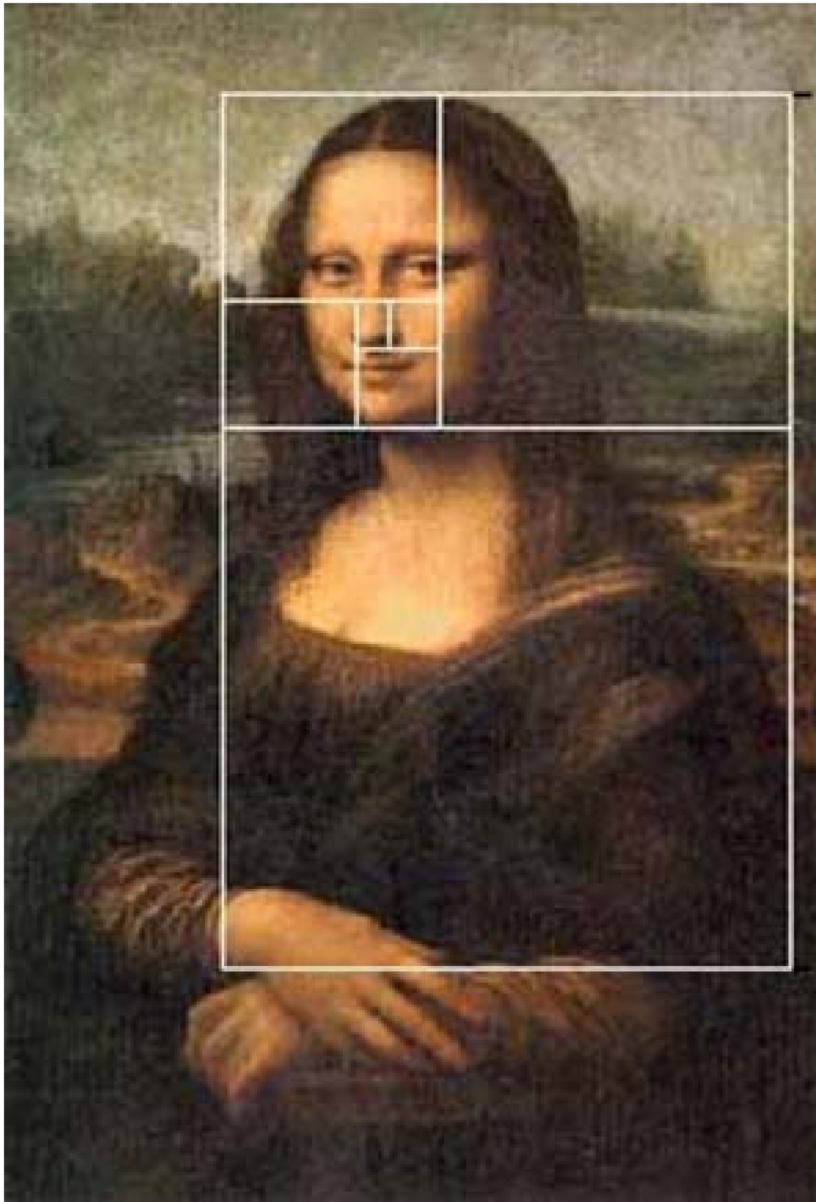
21/15 = 1.61518,…, 233/144 = 1.618055… → 1.618

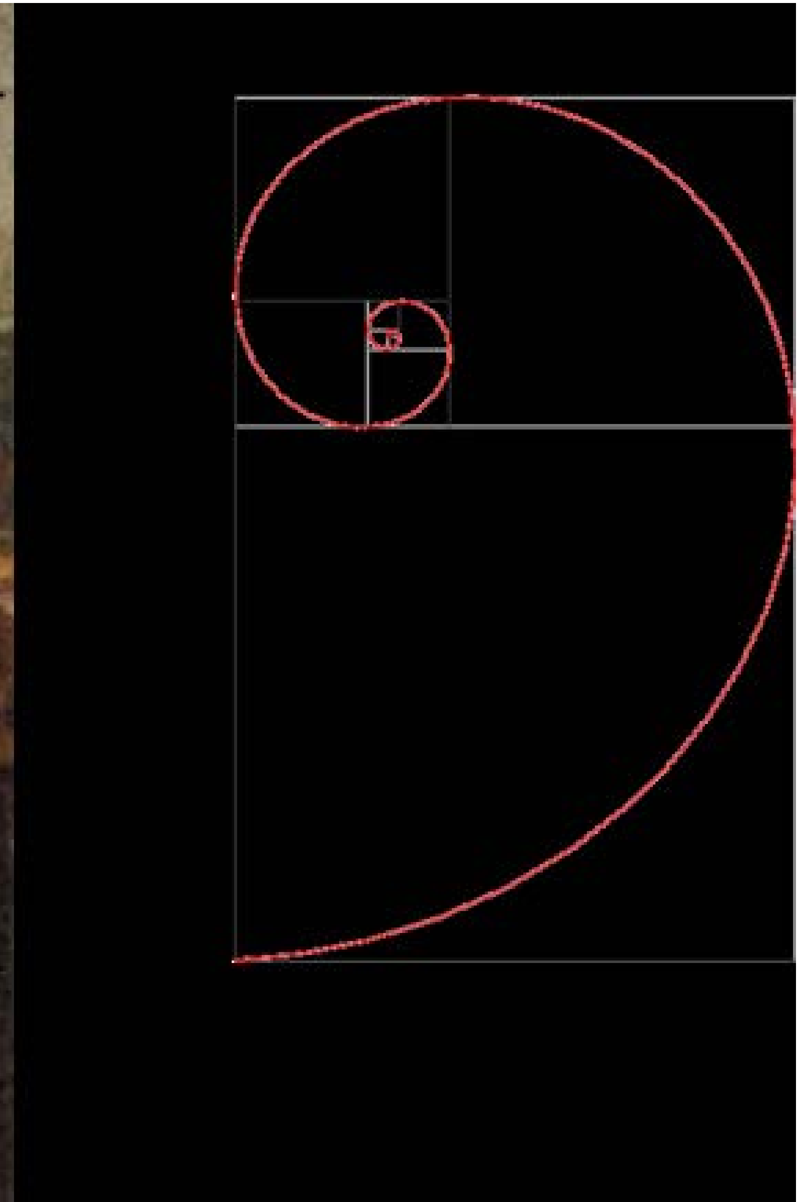


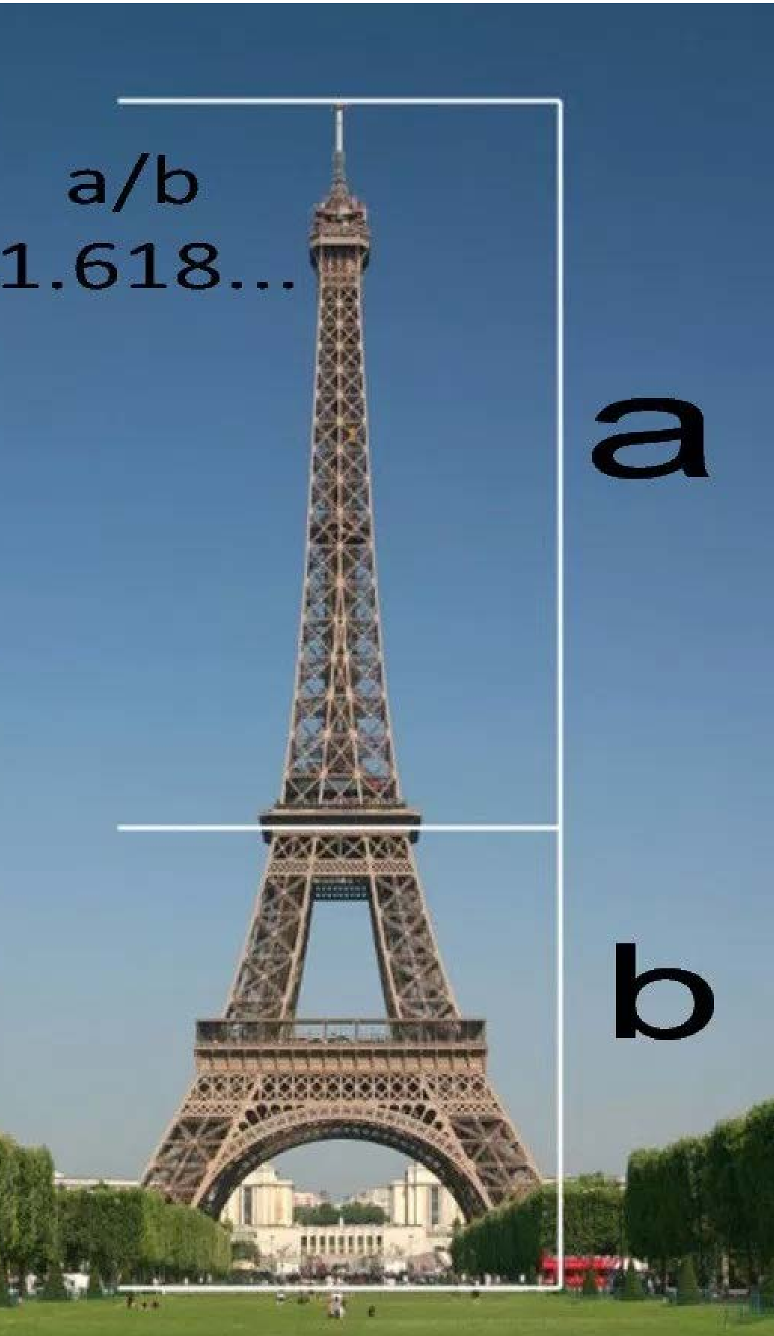
Notice the ratio:
(a+b)/a = a/b
=1.618

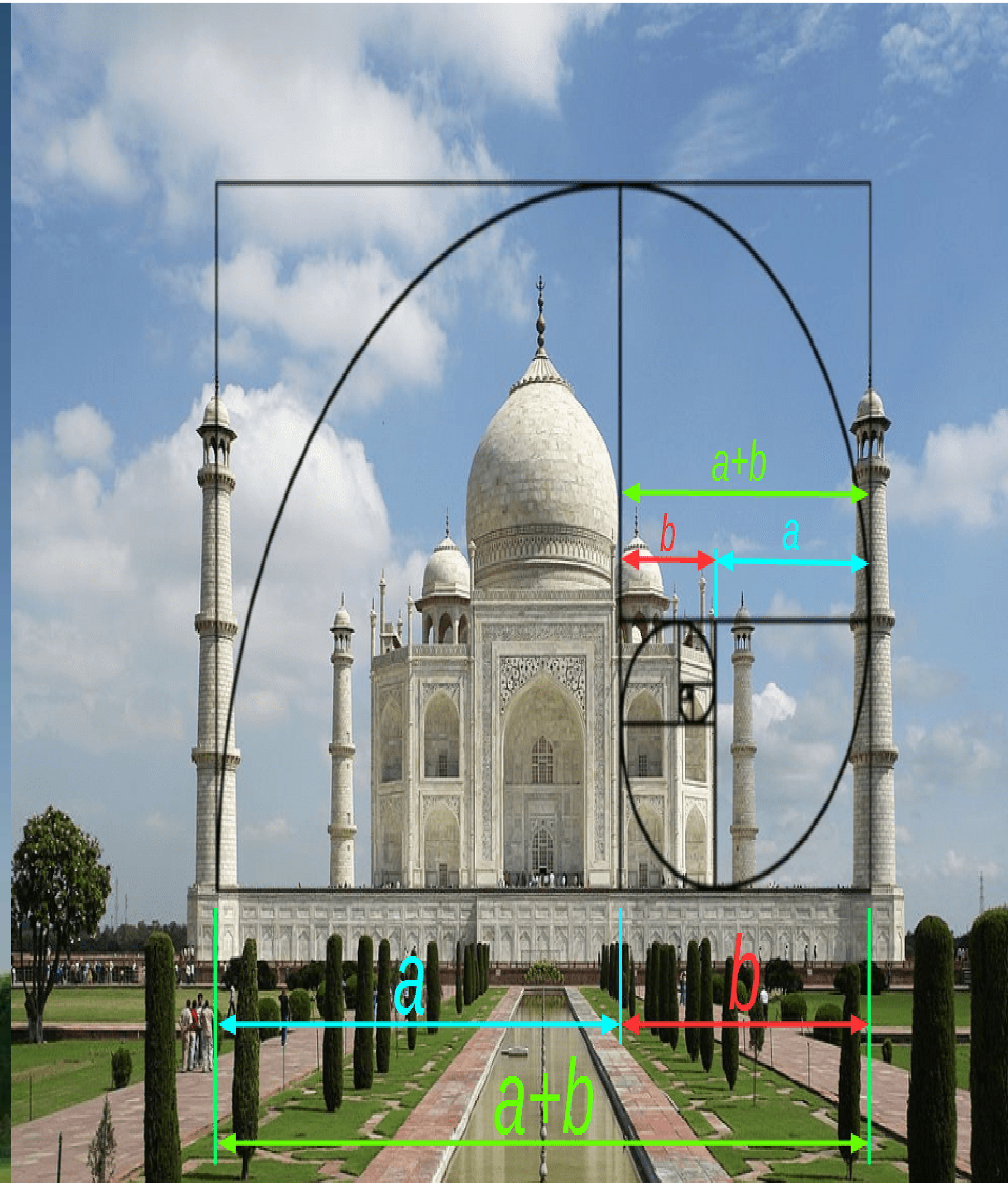
The Golden Ratio is considered as an indication of “beauty”.

Leonardo Da Vinci Used the Golden Ratio






a/b
1.618...
a
b


a+b
b
a
a
b
a+b

Example: Triangular Numbers

Consider the series of numbers: 1, 3, 6, 10, 15, 21, 28,...

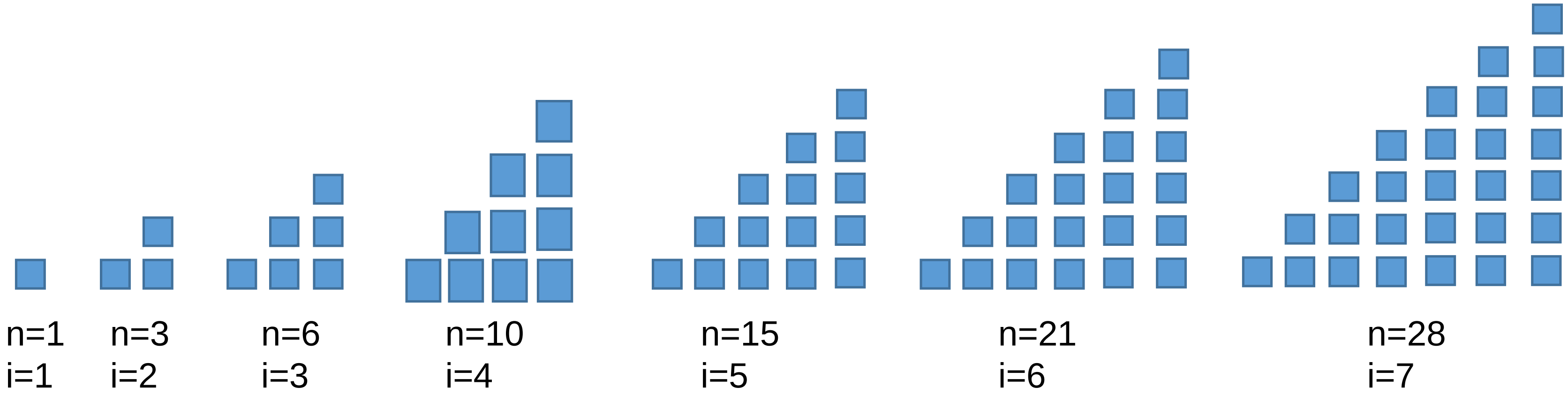
Is there a relationship between the numbers?

How can we get the i^{th} value of n using i?

Example: Triangular Numbers

Consider the series of numbers: 1, 3, 6, 10, 15, 21, 28,...

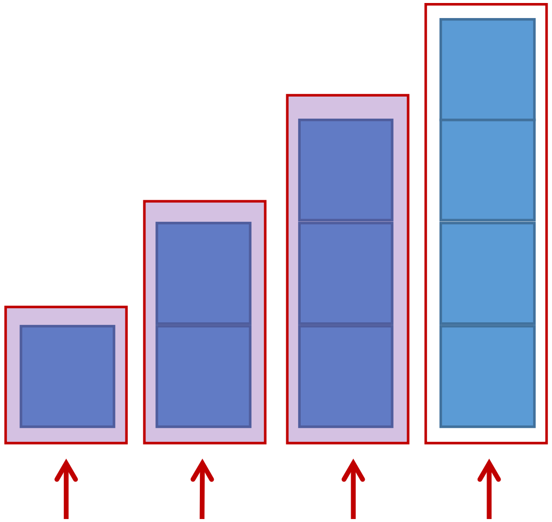
Is there a relationship between the numbers?



How can we get the i^{th} value of n using i?

Triangular Numbers

What is the value of the i^{th} term (Total) in the series?



i 1 2 3 4

Total # boxes 1 3 6 10 : 3=2+1, 6=3+3...

Triangular Numbers: Iterative Solution

The following function TotNum() uses iterative technique to find the total value (n) of the i^{th} term in the series.

Pseudocode

```
TotNum(i )

Total ← 0
while(i > 0)

   Total ← Total + i
   i ← i -1

return TotNum
```

C++

```
int TotNum(int i)
{
    int TotNum = 0;
    while(i > 0)
    {
        total = total + i;
        --i;
    }
    return total;
}
```

Recursive Solution

Another way of finding the total value of the i^{th} term in the series: Recursion. The total value of the $\boldsymbol{i}^{th}$ term in the series is obtained by adding $\boldsymbol{i}$ to the previous total.

The value of the i^{th} term can be obtained as the sum of only two things

1. The last column, which has the value i.
2. The sum of all the previous columns.

Recursive formulation :

TotNum =1 if i=1

TotNum(i) = i + TotNum(i-1) if i>1

Recursive Solution

We can use recursion to solve the problem:

Pseudocode

```
TotNum(i )

if  i=1
    then return 1
else
    return (i + TotNum(i-1))
```

C++

```
int TotNum(int i)
{
if(i==1)
     return 1;
else
   return (i + TotNum(i-1));
}
```

Complexity: $f(n) = O(n)$

Recursion: Reversing an Array

ReverseArray(*A, i, j*):

Input: An array *A* and nonnegative integer indices *i* and *j*

Output: The reversal of the elements in *A* starting at i, i ending at *j*

if $i < j$ then

Swap ($A[i]$, $A[j]$) //Exchange arguments

ReverseArray(A, $i + 1$, $j - 1$)

return

Reversing an Array Iterative Version

IterativeReverseArray(A, i, j)

while $i < j$ do

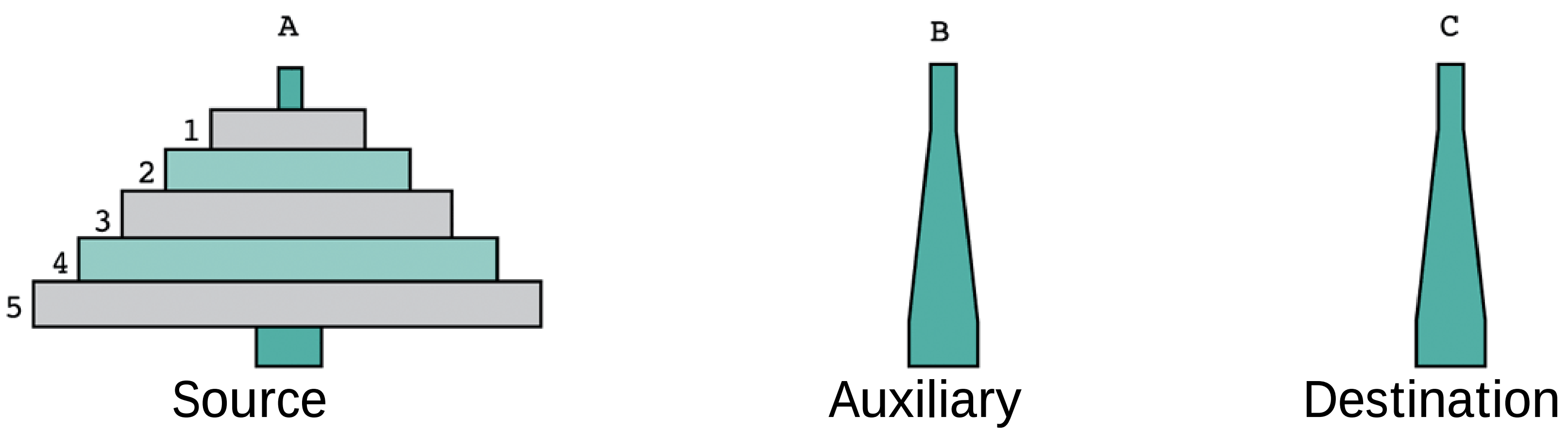
  Swap ($A[i]$, $A[j]$)

  $i = i + 1$

  $j = j - 1$

Recursion: Towers of Hanoi

- The towers of Hanoi problem involves moving a number of disks (in different sizes) from one tower (or "peg") to another.
- Initially the game has a few discs arranged in increasing order of size in one of three towers.
  - The constraint is that a larger disk can never be placed on top of a smaller disk.
  - Only one disk can be moved at each time



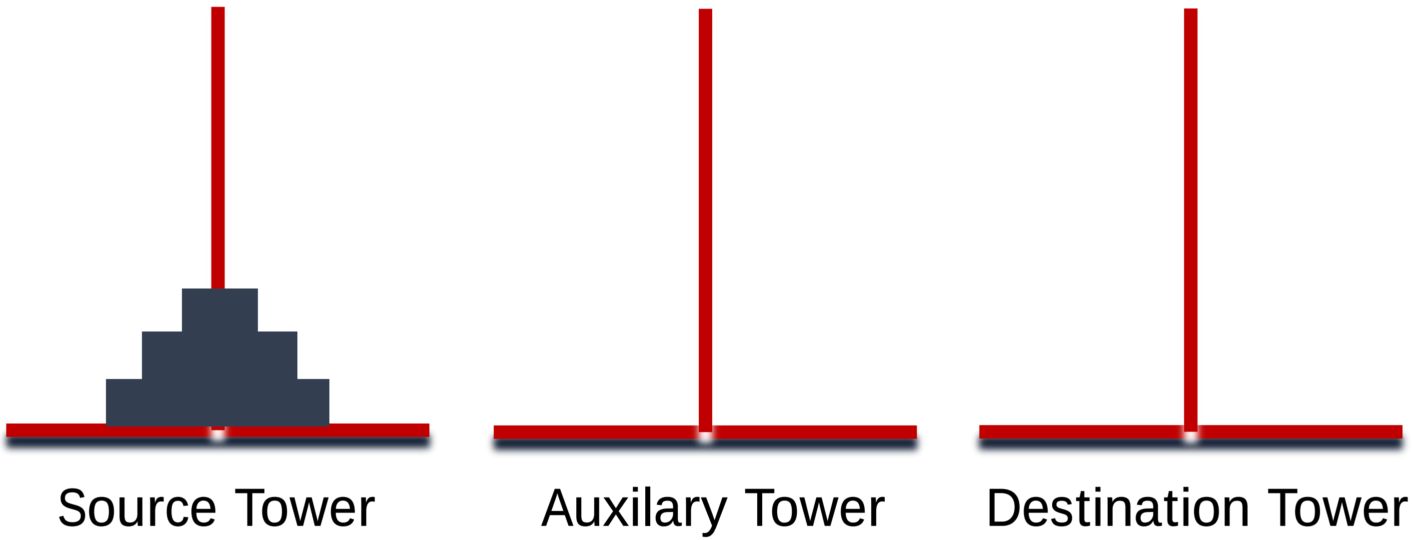
Demonstrating Recursion with Towers of Hanoi

Informal algorithm

1) Move the top n-1 disks from Source to Auxiliary tower,
2) Move the n^{th} disk from Source to Destination tower,
3) Move the n-1 disks from Auxiliary tower to Destination tower.

Where is recursion?

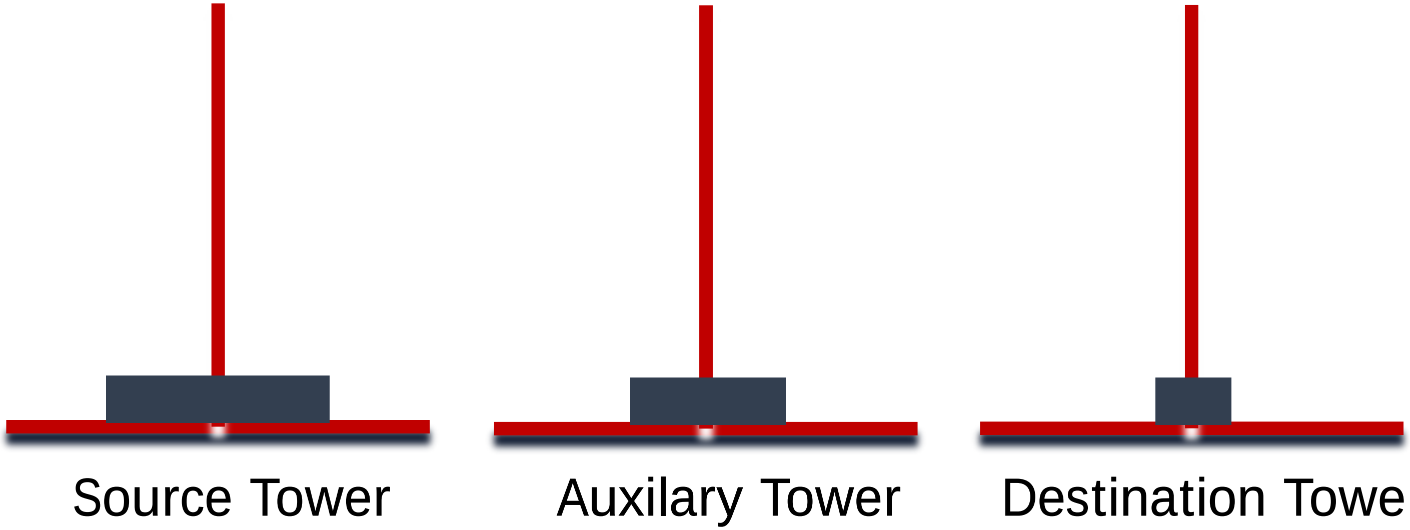
Towers of Hanoi with Recursion: n=3



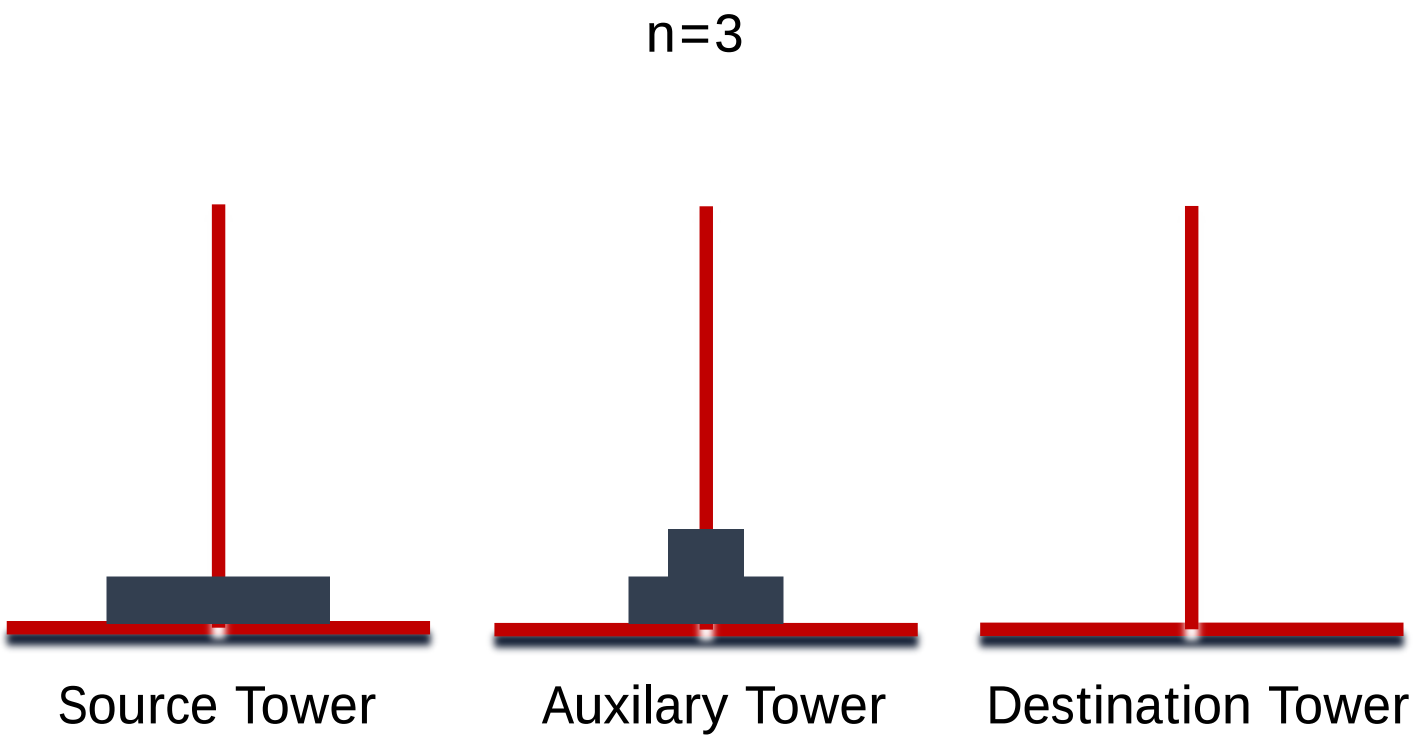
Towers of Hanoi with Recursion

Towers of Hanoi with Recursion

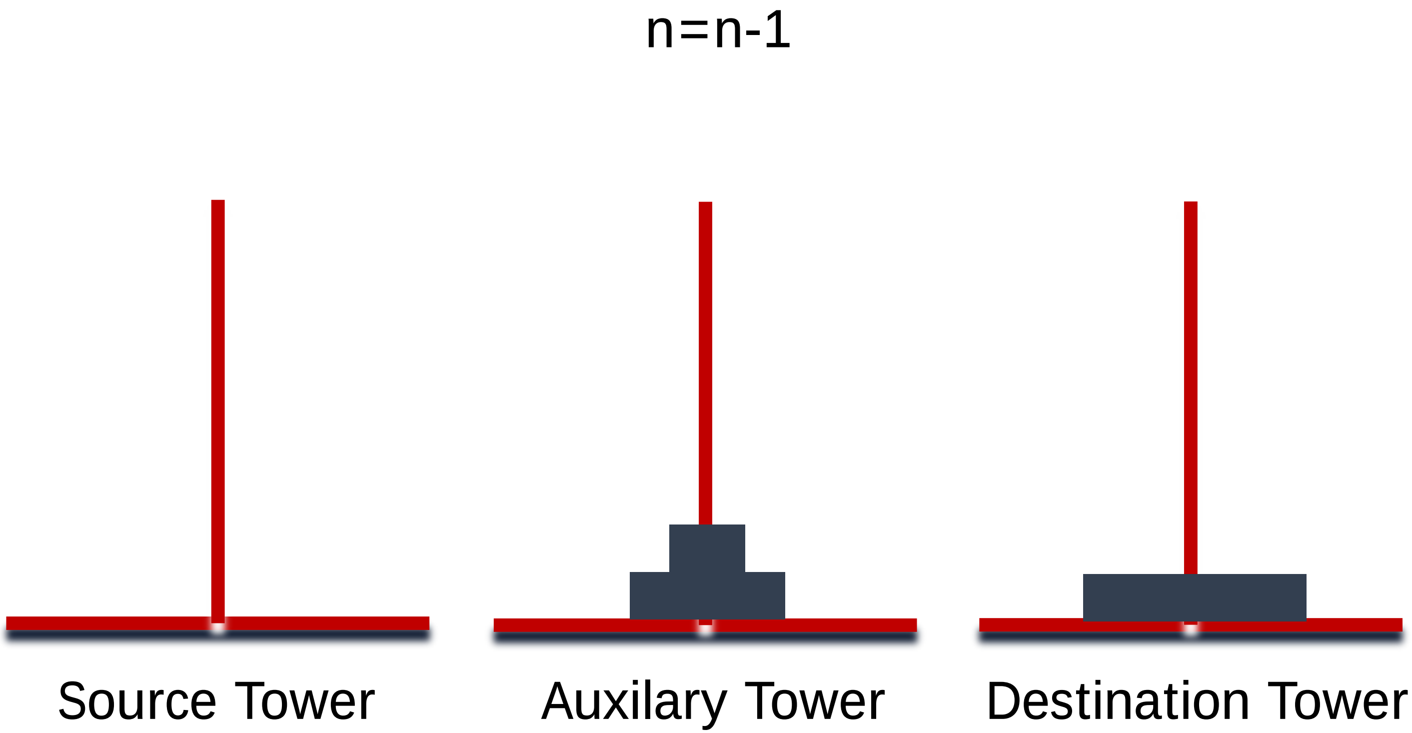
n=3



Towers of Hanoi with Recursion

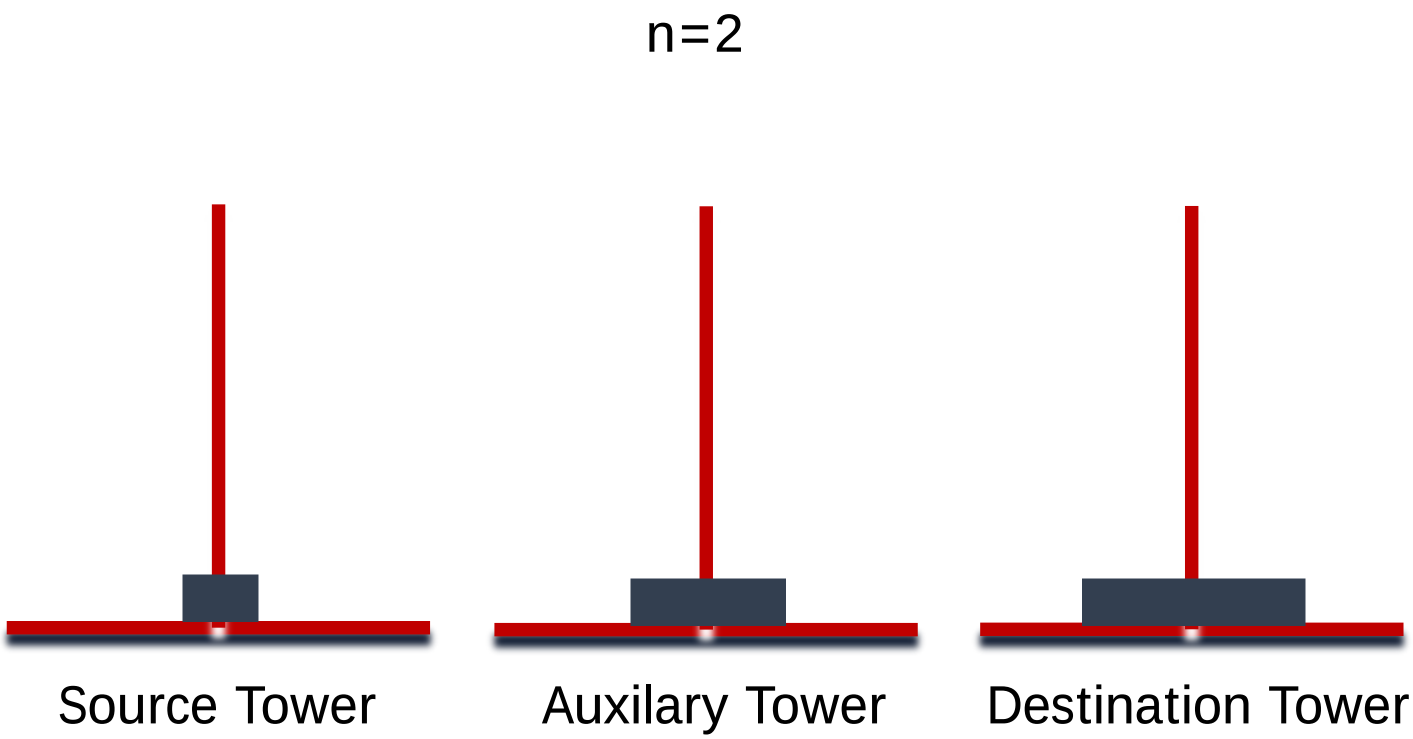


Towers of Hanoi with Recursion



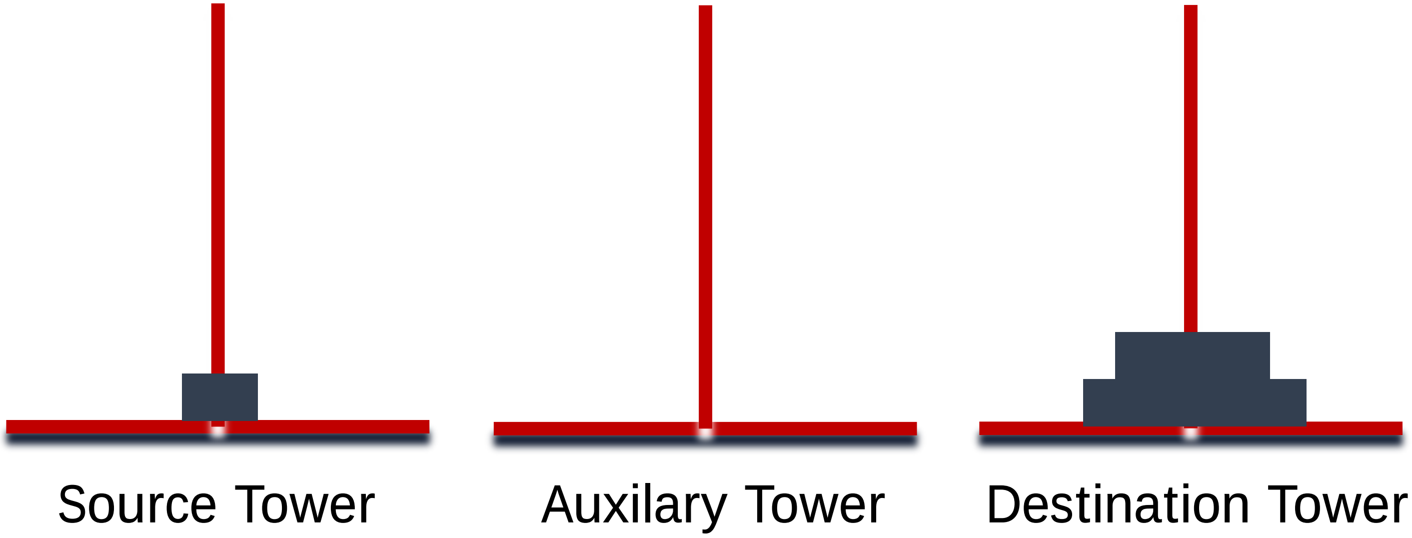
Now, we continue recursively, with n=n-1=2, using source as auxiliary.

Towers of Hanoi with Recursion

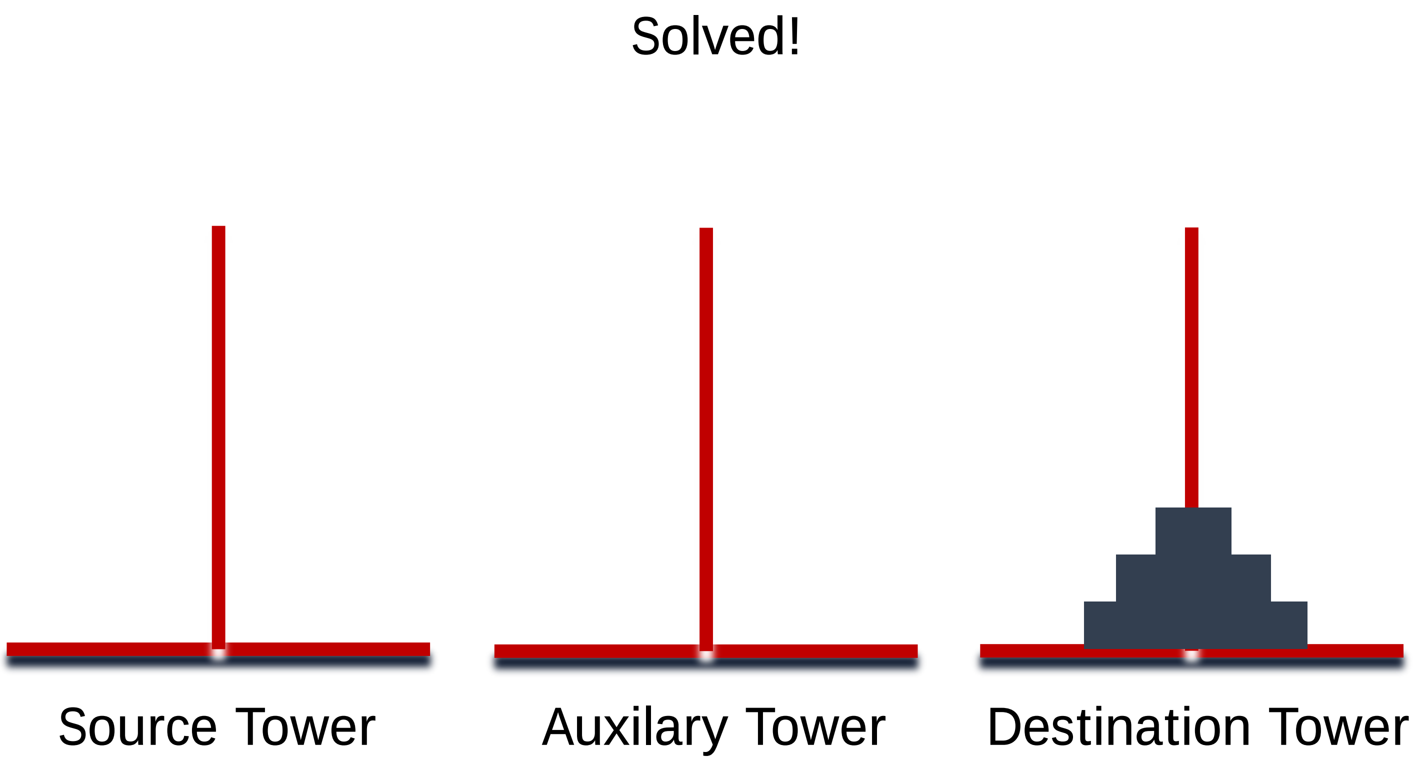


Towers of Hanoi with Recursion

n=1: Base case



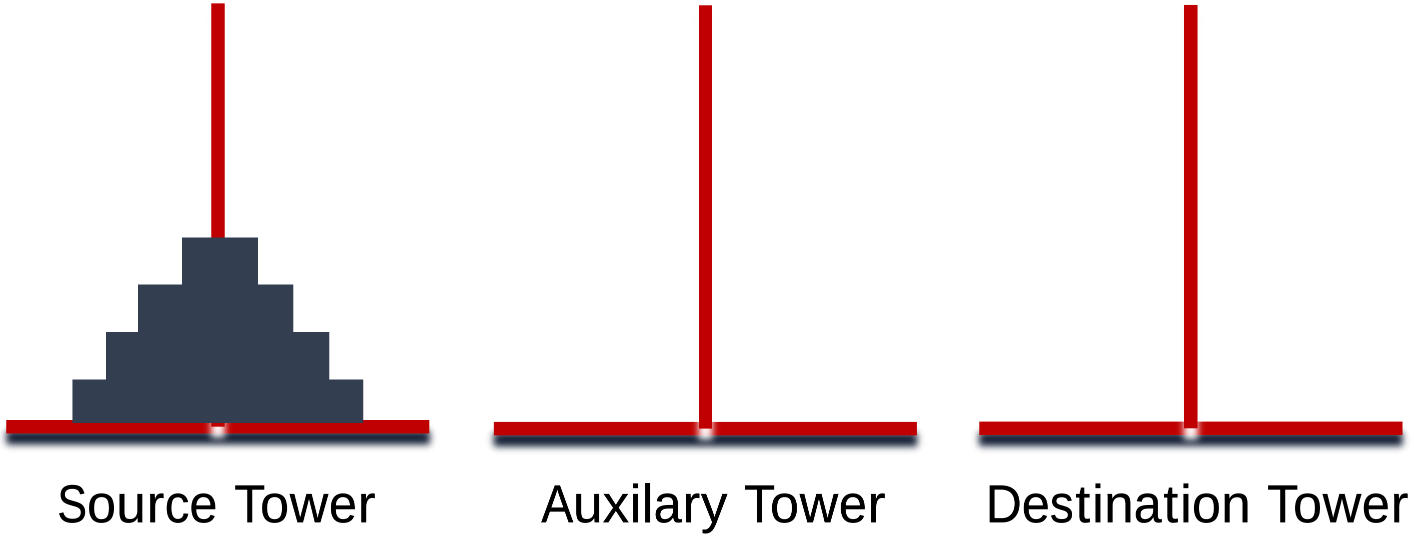
Towers of Hanoi with Recursion



How many moves did we perform?

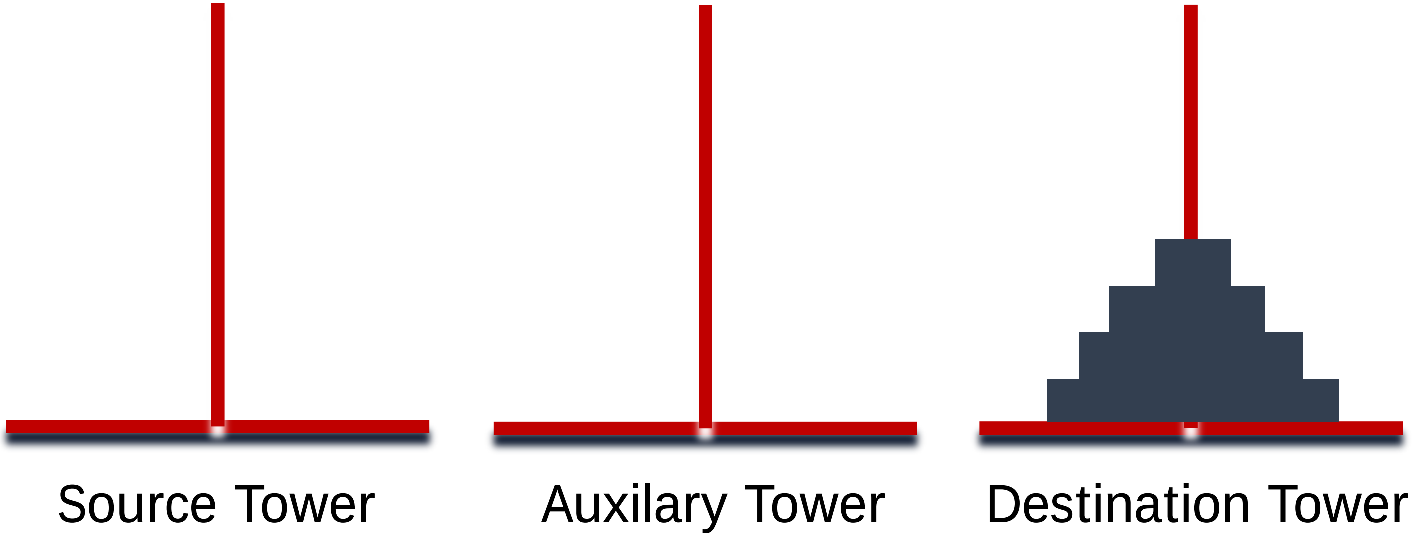
Towers of Hanoi with Recursion

Try with n=4



Towers of Hanoi with Recursion

Try with n=4: Final state.



How many moves are needed?

Towers of Hanoi: Recursive Algorithm

The pseudocode:

```
TOH(n, source, destination, auxilary)    //Move n disks from source to
                                          destination using auxilary tower)
  if n=0 return                             //Base case
  TOH(n-1, source, auxilary, destination)   //Recursive call: n-1 disks
  MOVE(source, destination);                // Move bottom disk to destination
  TOH(n-1, auxilary, destination, source)   //Recursive call: n-1 disks.
```

//MOVE is an external function.

Towers of Hanoi: C++ Code

The C++ code // Pole is a suitable data type.

```
void TOH(int n, Pole source, Pole destination, Pole auxilary) {
        if (n == 0) return;   // Base case
        TOH(n-1, source, auxilary, destination);
         // Recursive call: n-1 disks
         move(source, destination);
         // Move bottom disk to destination
         TOH(n-1, auxilary, destination, source);
         // Recursive call: n-1 disks
}
```

Time Complexity of Towers of Hanoi

- The Towers of Hanoi problem with 3 pegs and n disks takes $T(n) = 2^n - 1$

N=3→ 7

N=4→15

N=5→31

………..

N=64→ 2^{64}

$>10^{19}$

In the real life, for N=64, allowing one move per second would require more than 500 billion years to complete the task!

Properties of Recursive Solutions

The function calls itself with a new and smaller value of the parameter.

When it calls itself, it does so to solve a smaller problem.

There's some version of the problem that is simple enough that the routine can solve it, and return, without calling itself: Base case

Properties of Recursive Solutions

- The base case is the smallest problem that the routine solves and the value is returned to the calling method. (Terminal condition)

- Calling a method involves certain overhead:
  - Transferring the control to the beginning of the method and
  - Storing the information of the return point.

- Memory is used to store all the intermediate arguments and return values internally.

- The overheads increase cost and might cause problems if there is a large amount of data, leading to overflows.

Efficiency of Recursion

Recursion is usually applied because it simplifies a problem conceptually, not because it is inherently more efficient.

- In general, recursive methods may be less efficient than their iterative versions: Remember the complexity of recursive Fibonacci.

- However, recursion can simplify the solution of a problem, often resulting in shorter source code.

Tips for Using Recursion

1. Make sure the recursion stops
2. Use safety counters to prevent infinite recursion

The recursive routine must be able to change the value of safetyCounter*, so in Visual Basic it's a* ByRef *parameter.*

```
Public Sub RecursiveProc( ByRef safetyCounter As Integer )
   If ( safetyCounter > SAFETY_LIMIT ) Then
      Exit Sub
   End If
   safetyCounter = safetyCounter + 1
   ...
   RecursiveProc( safetyCounter )
End Sub
```

3. Limit recursion to one routine

Tips for Using Recursion

4. Keep an eye on the stack
5. Don't use recursion for factorials or Fibonacci numbers

Java Example of an Inappropriate Solution: Using Recursion to Compute a Factorial

```java
int Factorial( int number ) {
   if ( number == 1 ) {
      return 1;
   }
   else {
      return number * Factorial( number - 1 );
   }
}
```

Java Example of an Appropriate Solution: Using Iteration to Compute a Factorial

```java
int Factorial( int number ) {
   int intermediateResult = 1;
   for ( int factor = 2; factor <= number; factor++ ) {
      intermediateResult = intermediateResult * factor;
   }
   return intermediateResult;
}
```