




# INTRODUCTION TO RECURSION

# Prank - the Google way

 recursion  


All Books Images Videos News More Settings Tools



About 12,600,000 results (0.55 seconds)

Did you mean: *recursion*

**Recursion** occurs when a thing is defined in terms of itself or of its type. **Recursion** is used in a variety of disciplines ranging from linguistics to logic. The most common application of **recursion** is in mathematics and computer science, where a function being defined is applied within its own definition.

[Recursion - Wikipedia](https://en.wikipedia.org/wiki/Recursion)  
<https://en.wikipedia.org/wiki/Recursion>



 About this result  Feedback

# Recursion is Everywhere!

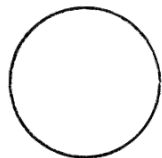
- At first, recursion may seem like a counter-intuitive way of problem solving, it is everywhere.
- It is a top down approach.
- A fun answer from Quora says -
  - Recursion is optimistic and lazy. He likes to solve the problems that are either very small or the problems that are almost done.
  - Iteration is well planned and hard working . He likes big problems.
- ***Bored?***

***Read this sentence again from the beginning.***

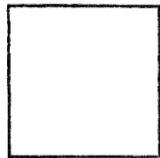
***Stuck? xD***

# Fractals

# Euclidean geometry does not explain everything



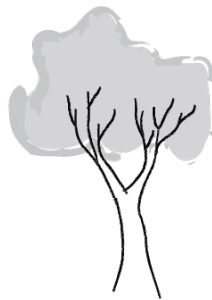
circle



square



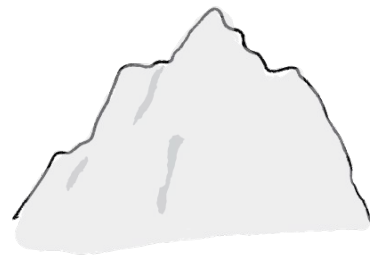
line



tree



lightning bolt

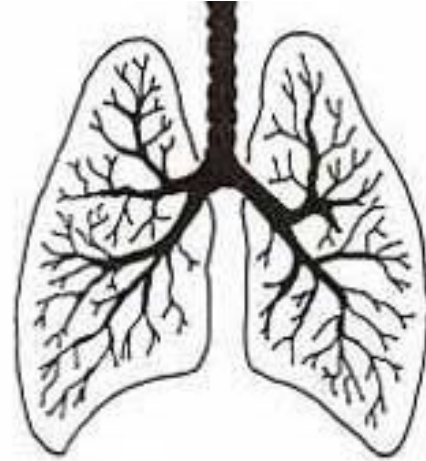
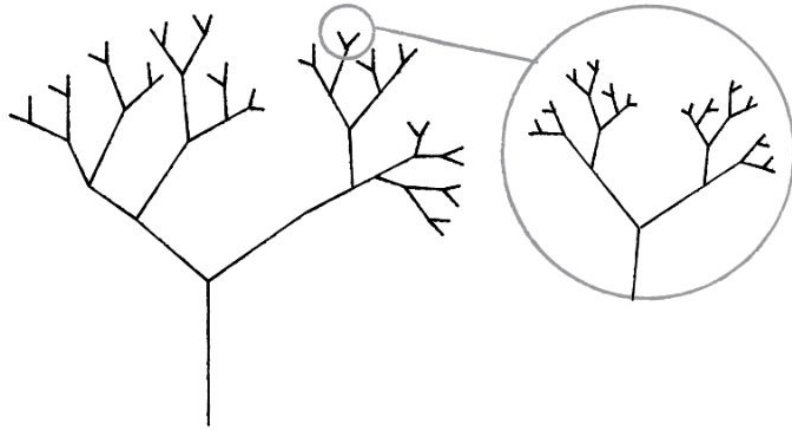


mountain

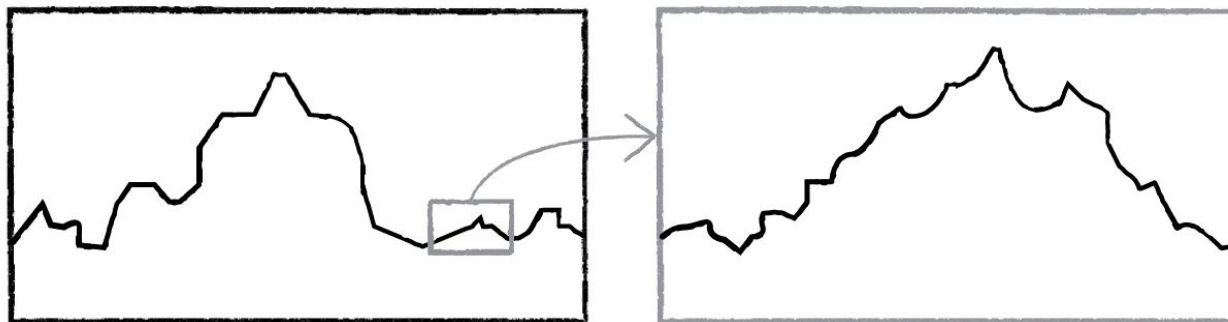
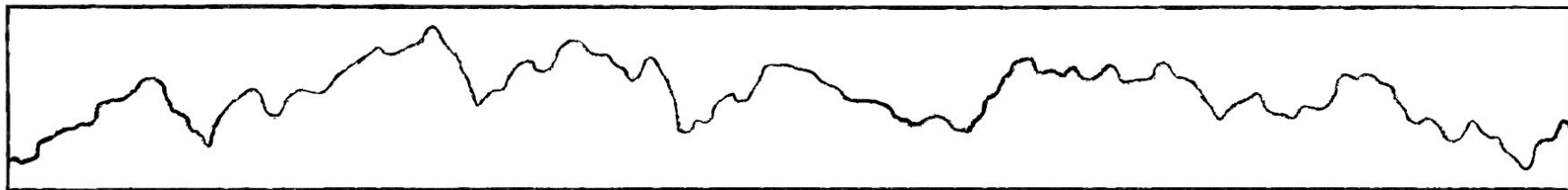
# For example

- Your blood vessels
- Coastline
- Tree structure
- Shape of a cauliflower

# Repeating structures emerge naturally



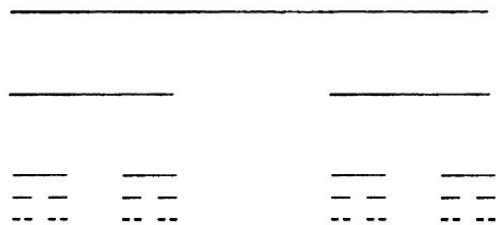
# Apple stock





# Cantor set

1. Start with a line.
2. Erase the middle third of that line
3. Repeat step 2 for the remaining lines,  
again and again and again.



In mathematics, the Cantor set is a set of points lying on a single line segment that has a number of remarkable and deep properties.

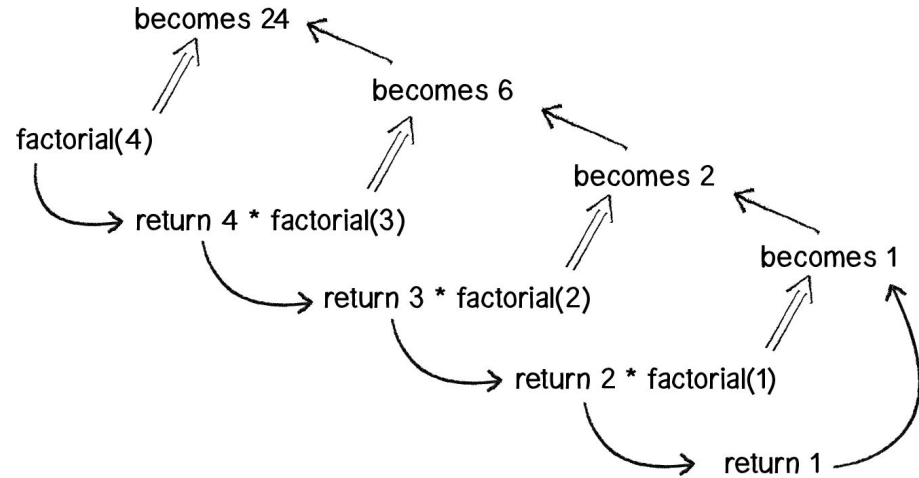
# Shockingly ...



A column capital from the Ancient Egyptian site of the island of Philae carries a pattern which resembles the Cantor set.

# Recursion is a great tool for generating fractals

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



# Recursion

- Involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially
- Function calls itself
- Recursion allows us to write elegant solutions to problems
- The Three Laws of Recursion

1. A recursive algorithm must have a base case.

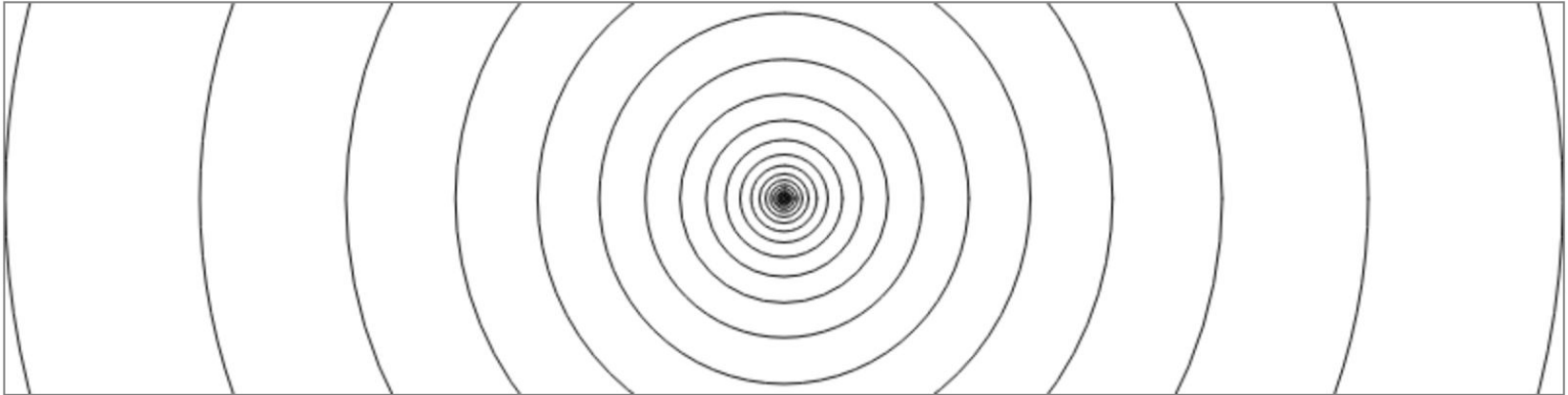
2. A recursive algorithm must change its state and move toward the base case.

3. A recursive algorithm must call itself, recursively

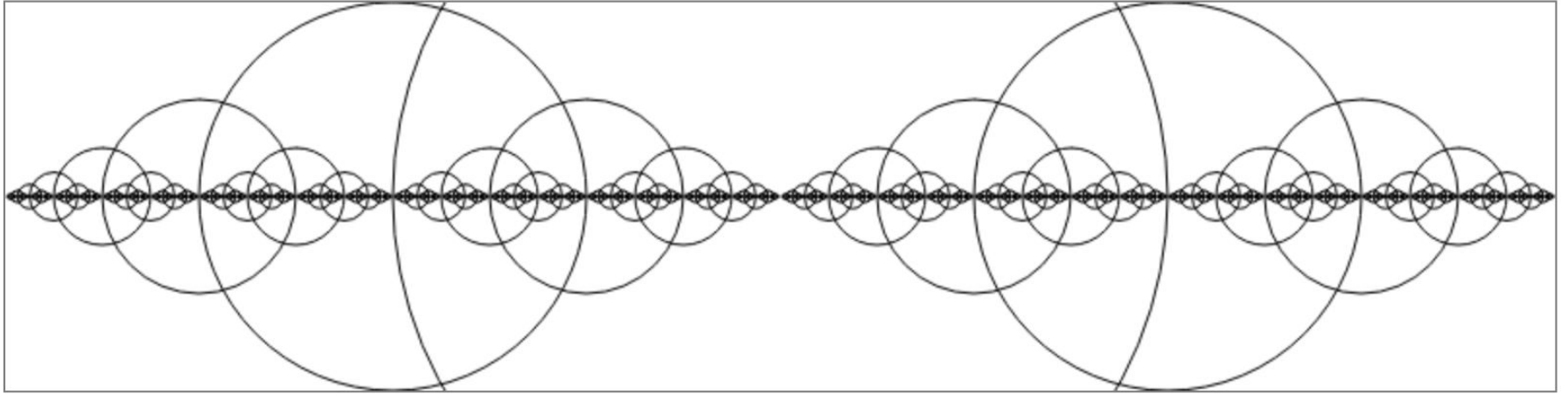
# The recursive circle example (iterates and stops)

```
void drawCircle(int x, int y, float radius) {  
    ellipse(x, y, radius, radius);  
    if(radius > 2) {  
        radius *= 0.75f;  
        drawCircle(x, y, radius);  
    }  
}
```

The drawCircle() function is calling itself recursively.



How do you think you can get the following?

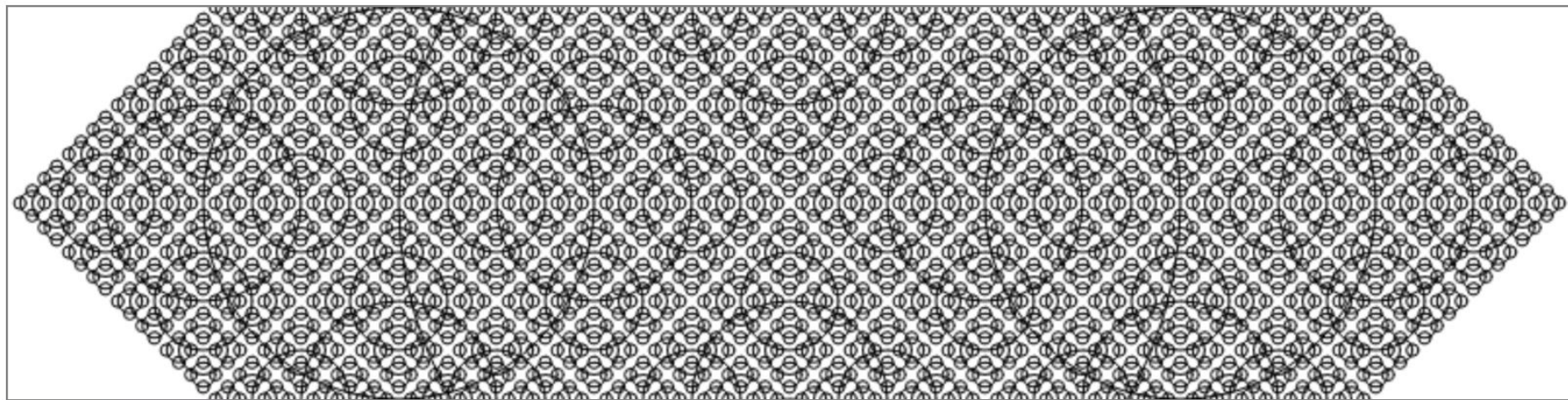


# Here is the solution

```
void setup() {  
  size(640,360);  
}  
  
void draw() {  
  background(255);  
  drawCircle(width/2,height/2,200);  
}  
  
void drawCircle(float x, float y, float radius) {  
  stroke(0);  
  noFill();  
  ellipse(x, y, radius, radius);  
  if(radius > 2) {  
    drawCircle(x + radius/2, y, radius/2);  
    drawCircle(x - radius/2, y, radius/2);  
  }  
}
```

drawCircle() calls itself twice, creating a branching effect. For every circle, a smaller circle is drawn to the left and the right.

For 4 circles

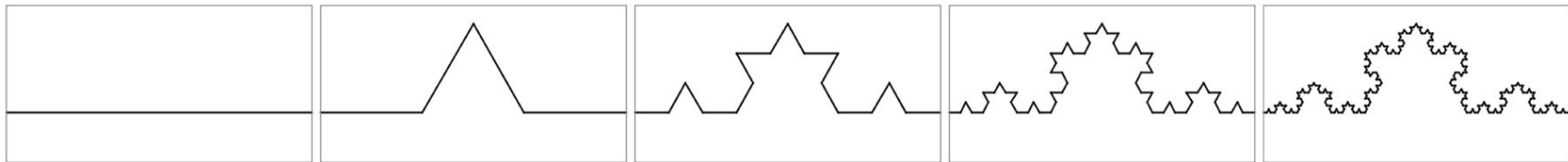




# Not surprising ...

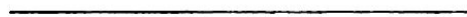
```
void drawCircle(float x, float y, float radius) {  
    ellipse(x, y, radius, radius);  
    if(radius > 8) {  
        drawCircle(x + radius/2, y, radius/2);  
        drawCircle(x - radius/2, y, radius/2);  
        drawCircle(x, y + radius/2, radius/2);  
        drawCircle(x, y - radius/2, radius/2);  
    }  
}
```

Now how do you think you can get this?



# Background

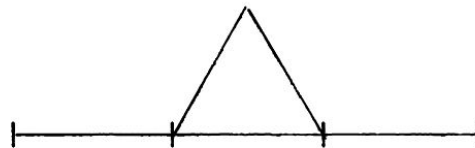
1. Start with a line.



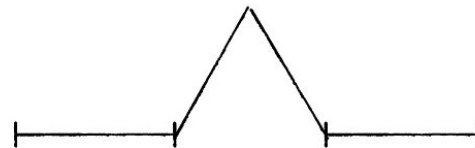
2. Divide the line into three equal parts.



3. Draw an equilateral triangle (a triangle where all the sides are equal) using the middle segment as its base.



4. Erase the base of the equilateral triangle (the middle segment from step 2).



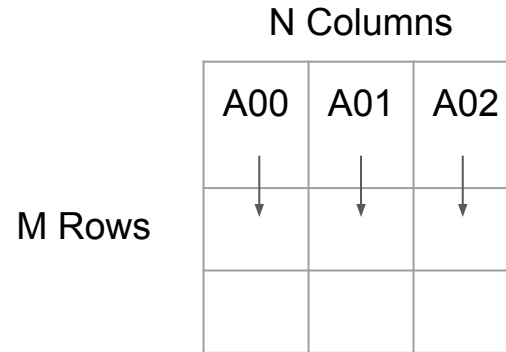
5. Repeat steps 2 through 4 for the remaining lines again and again and again.

# The 'Monster' Curve

The Koch curve and other fractal patterns are often called “mathematical monsters.” This is due to an odd paradox that emerges when you apply the recursive definition an infinite number of times. If the length of the original starting line is one, the first iteration of the Koch curve will yield a line of length four-thirds (each segment is one-third the length of the starting line). Do it again and you get a length of sixteen-ninths. **As you iterate towards infinity, the length of the Koch curve approaches infinity.** Yet it fits in the tiny finite space provided right here on this paper (or screen)!

# Problem - Counting Students

- Consider a classroom where we need a headcount of the students.
- Instead of counting one by one, we decide to delegate this task to the students sitting in the first row
- He/she divides the task into smaller parts and delegates it to the other students, sitting behind them.



# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):
```

```
    if X == NULL
```

```
        return
```

```
    else if check_behind(X) == FALSE           //checks for last student in row
```

```
        return 1
```

```
    else
```

```
        return count_behind(behind of X) + 1 //recursive call
```

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

N Columns

A00	A01	A02
A10	A11	A12
A20	A21	A22

M Rows

# Counting Students Algorithm

Iterate over the list of all students in the first row

for each student X:

    count\_behind(X):

        if X == NULL

            return

        else if check\_behind(X) == FALSE

//checks for last student in row

            return 1

    else

        return count\_behind(behind of X) + 1

//recursive call

N Columns

	A00	A01	A02
M Rows ↓	A10	A11	A12
	A20	A21	A22



# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

		N Columns		
M Rows		A00	A01	A02
		A10	A11	A12
		A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

`count_behind(X):`

    if X == NULL

        return

    else if `check_behind(X) == FALSE`

//checks for last student in row

        return 1

    else

        return `count_behind(behind of X) + 1`

//recursive call

		N Columns		
M Rows		A00	A01	A02
		A10	A11	A12
		A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return 1 + 1 //recursive call
```

N Columns

M Rows ↑

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return 2 + 1 //recursive call
```

		N Columns		
M Rows		A00	A01	A02
		A10	A11	A12
		A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

Value at A00 = 3

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

N Columns

A00	A01	A02
A10	A11	A12
A20	A21	A22

M Rows

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows ↓

A00	A01	A02
A10	A11	A12
A20	A21	A22



# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
//checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

	N Columns		
M Rows	A00	A01	A02
	A10	A11	A12
	A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

	N Columns		
M Rows	A00	A01	A02
	A10	A11	A12
	A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return 1 + 1 //recursive call
```

N Columns

M Rows ↑

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return 2 + 1 //recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

Value at A00 = 3  
Value at A01 = 3

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows ↓

A00	A01	A02
A10	A11	A12
A20	A21	A22



# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

		N Columns		
M Rows		A00	A01	A02
		A10	A11	A12
		A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

		N Columns		
M Rows		A00	A01	A02
		A10	A11	A12
		A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
count_behind(X):  
    if X == NULL  
        return  
    else if check_behind(X) == FALSE  
        //checks for last student in row  
        return 1  
    else  
        return count_behind(behind of X) + 1  
//recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return 1 + 1 //recursive call
```

N Columns

M Rows ↑

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return 2 + 1 //recursive call
```

N Columns

M Rows

A00	A01	A02
A10	A11	A12
A20	A21	A22

# Counting Students Algorithm

Iterate over the list of all students in the first row  
for each student X:

```
    count_behind(X):  
        if X == NULL  
            return  
        else if check_behind(X) == FALSE  
            //checks for last student in row  
            return 1  
        else  
            return count_behind(behind of X) + 1  
//recursive call
```

		N Columns		
M Rows		A00	A01	A02
		A10	A11	A12
		A20	A21	A22
Value at A00 = 3				
Value at A01 = 3				
Value at A02 = 3				

# Counting Students Algorithm

- Now, the values at the first row can be summed up to get the total.
- Value at A00 = 3  
Value at A01 = 3  
Value at A02 = 3
- Total =  $A00 + A01 + A02 = 9$

# Example 1 - Fibonacci Series

- $F(n) = F(n-1) + F(n-2)$
- One of the most basic examples.
- Solution to the factorial of  $n$  depends on the factorial  $n-1$  and  $n-2$



## Example 2 - Factorial

- $n! = n * n-1 * n-2 * \dots * 1$
- $n! = n * (n-1)!$
- $F(n) = n * F(n-1)$
- The function can be expressed in terms of itself

# Recurrences and Running Time

# Recurrences

- Recurrences arise when an algorithm contains recursive calls to itself
- Running time is represented by an equation or inequality that describes a function in terms of its value on smaller inputs.
- What is the actual running time of the algorithm, i.e.,  $T(n) = ?$
- Need to solve the recurrence
- Find an explicit formula of the expression.
- Bound the recurrence by an expression that involves  $n$ .

# Binary Search

# Binary Search

- A search technique to find an element in a sorted array of numbers
- Also known as **half-interval search**, **logarithmic search**, or **binary chop**
- After each unsuccessful intermediate search, half of the array is discarded

# Binary Search Algorithm

**Find if the element 'x' is in the sorted array A[lo...hi]**

```
BINARY-SEARCH (A, lo, hi, x)
  if (lo > hi)
    return FALSE
  mid = (lo+hi)/2
  if x = A[mid]
    return TRUE
  if ( x < A[mid] )
    BINARY-SEARCH (A, lo, mid-1, x)
  if ( x > A[mid] )
    BINARY-SEARCH (A, mid+1, hi, x)
```

# Analysis of Binary Search

**BINARY-SEARCH (A, lo, hi, x)**

if (lo > hi)

return FALSE //constant time:  $c_1$

mid = (lo+hi)/2 //constant time:  $c_2$

if x = A[mid]

return TRUE //constant time:  $c_3$

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

A=

1	2	3	4	5	7	9	11
---	---	---	---	---	---	---	----

**lo= 1, hi= 8, x=6**

**BINARY-SEARCH (A, 1, 8, 6)**

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time:  $c_1$

mid = (lo+hi)/2 //constant time:  $c_2$

if x = A[mid]

return TRUE //constant time:  $c_3$

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

A=

1	2	3	4	5	7	9	11
---	---	---	---	---	---	---	----

lo= 1, hi= 8, x=6

if (1 > 8) //No!



# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

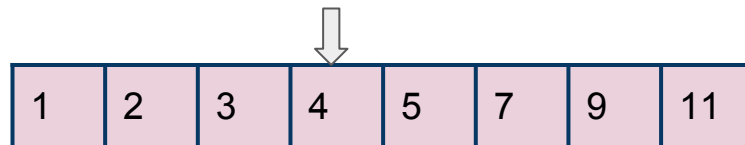
if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2

A=



1	2	3	4	5	7	9	11
---	---	---	---	---	---	---	----

lo= 1, hi= 8, x=6

mid = (1+8)/2  
=4

**Note:** Single statement (independent of n) takes constant time c2

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

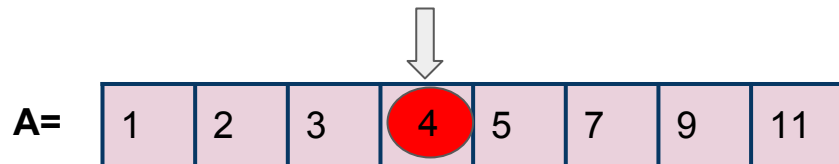
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



lo= 1, hi= 8, x=6

If 6=a[4] //No!

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

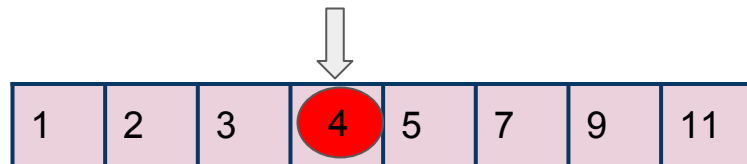
if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2

A=



lo= 1, hi= 8, x=6

If (6 < a[4]) //No!

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

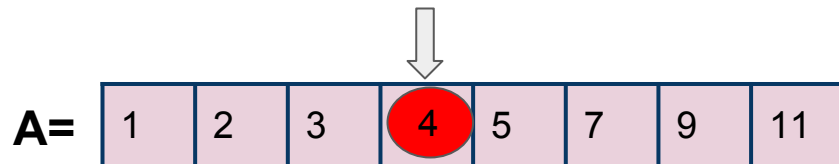
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$



**lo= 1, hi= 8, x=6**

**If (6 > a[4]) //Yes!**

# Analysis of Binary Search

```
BINARY-SEARCH (A, lo, hi, x)
```

```
  if (lo > hi)
```

```
    return FALSE    //constant time: c1
```

```
  mid = (lo+hi)/2    //constant time: c2
```

```
  if x = A[mid]
```

```
    return TRUE     //constant time: c3
```

```
  if ( x < A[mid] )
```

```
    BINARY-SEARCH (A, lo, mid-1, x) //same  
    problem of size  $n/2$ 
```

```
  if ( x > A[mid] )
```

```
    BINARY-SEARCH (A, mid+1, hi, x) //same  
    problem of size  $n/2$ 
```

**A=**

5	7	9	11
---	---	---	----

**lo= 5, hi= 8, x=6**

**BINARY-SEARCH (A, 4+1, 8, 6)**

**Recursive call:**

**BINARY-SEARCH (A, 5, 8, 6)**

**Note:** The number of elements is now 4  
So, the problem is reduced to  $n/2$

# Analysis of Binary Search

**BINARY-SEARCH (A, lo, hi, x)**

if (lo > hi)

    return FALSE     //constant time: c1

mid = (lo+hi)/2     //constant time: c2

if x = A[mid]

    return TRUE     //constant time: c3

if ( x < A[mid] )

    BINARY-SEARCH (A, lo, mid-1, x)     //same  
    problem of size  $n/2$

if ( x > A[mid] )

    BINARY-SEARCH (A, mid+1, hi, x)     //same  
    problem of size  $n/2$

**A=**

5	7	9	11
---	---	---	----

**lo= 5, hi= 8, x=6**

**BINARY-SEARCH (A, 5, 8, 6)**

The algorithm starts executing again, this time for the half elements of the array

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

**A =**

5	7	9	11
---	---	---	----

**lo = 5, hi = 8, x = 6**

if (5 > 8) //No!

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

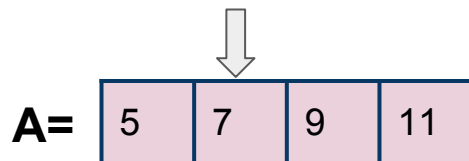
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



lo= 5, hi= 8, x=6

mid = (5+8)/2  
=6



# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

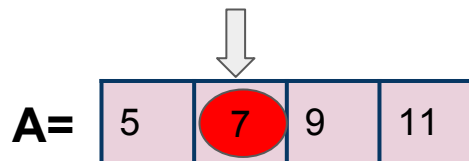
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



lo= 5, hi= 8, x=6

If 6=A[6] //No!

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

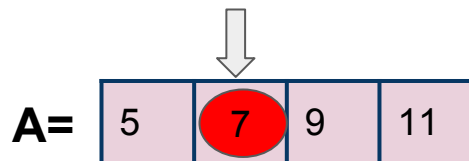
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



**lo= 5, hi= 8, x=6**

**If (6<A[6]) //Yes!**

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

A= 5

lo= 5, hi= 5, x=6

BINARY-SEARCH (A, 5, 6-1, 6)

**Recursive call:**

BINARY-SEARCH (A, 5, 5, 6)

**Note:** The number of elements is now 1  
So, the problem is reduced by half

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

A= 

5
---

lo= 5, hi= 5, x=6

If (5>5) //No!

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

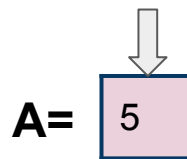
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



lo= 5, hi= 5, x=6

mid =(5+5)/2  
=5

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

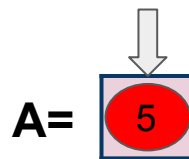
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



**lo= 5, hi= 5, x=6**

**If 6=A[5] //No!**

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

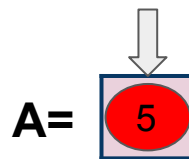
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



lo= 5, hi= 5, x=6

If (6<A[5]) //No!

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

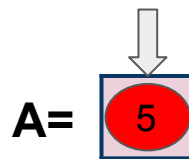
return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2



lo= 5, hi= 5, x=6

If (6>A[5]) //Yes!



# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

A= 

5	7
---	---

lo= 6, hi= 5, x=6

BINARY-SEARCH (A, 5+1, 5, 6)

**Recursive call:**

BINARY-SEARCH (A, 6, 5, 6)

# Analysis of Binary Search

**BINARY-SEARCH (A, lo, hi, x)**

if (lo > hi)

    return FALSE     //constant time: c1

mid = (lo+hi)/2     //constant time: c2

if x = A[mid]

    return TRUE     //constant time: c3

if ( x < A[mid] )

    BINARY-SEARCH (A, lo, mid-1, x)     //same  
    problem of size n/2

if ( x > A[mid] )

    BINARY-SEARCH (A, mid+1, hi, x)     //same  
    problem of size n/2

**A=**

5	7
---	---

**lo= 6, hi= 5, x=6**

**BINARY-SEARCH (A, 6, 5, 6)**

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

A= 

5	7
---	---

lo= 6, hi= 5, x=6

If (6 > 5) //Yes!

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size  $n/2$

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size  $n/2$

A= 

5	7
---	---

lo= 6, hi= 5, x=6

return FALSE

Takes constant time c1

# Analysis of Binary Search

BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE //constant time: c1

mid = (lo+hi)/2 //constant time: c2

if x = A[mid]

return TRUE //constant time: c3

if ( x < A[mid] )

BINARY-SEARCH (A, lo, mid-1, x) //same  
problem of size n/2

if ( x > A[mid] )

BINARY-SEARCH (A, mid+1, hi, x) //same  
problem of size n/2

$$T(n) = c + T(n/2)$$

A= 

5	7
---	---

lo= 6, hi= 5, x=6

return FALSE

Takes constant time c1

So, the total complexity can be presented by the following recurrence relation:

$$T(n) = c + T(n/2)$$

# Formative Assessment!

# Merge Sort

# Merge Sort

- This is Divide-and-Conquer type technique of sorting elements
- First list is divided into the smallest unit (1 element)
- Then comparison of each element with the adjacent list to sort and merge the two adjacent lists.
- Finally all the elements are sorted and merged.



# Merge Sort Algorithm

```
MergeSort(A, left, right)
```

```
    if (left < right)
```

```
        mid = floor((left + right) / 2)
```

```
        MergeSort(A, left, mid)
```

```
        MergeSort(A, mid+1, right)
```

```
        Merge(A, left, mid, right)
```

Merge() takes two sorted subarrays and merges them into a single sorted subarray of A

# Analysis of Merge Sort

```
MergeSort(A, left, right)
```

```
    if (left < right)                                //constant time:  $c_1$ 
```

```
        mid = floor((left + right) / 2)              //constant time:  $c_2$ 
```

```
        MergeSort(A, left, mid)                      //same problem of size  $n/2$ 
```

```
        MergeSort(A, mid+1, right)                   //same problem of size  $n/2$ 
```

```
        Merge(A, left, mid, right)                   //time proportional to size of  $n$ 
```

$$T(n) = 2 * T(n/2) + nc$$

# Tower of Hanoi

# Problem Definition

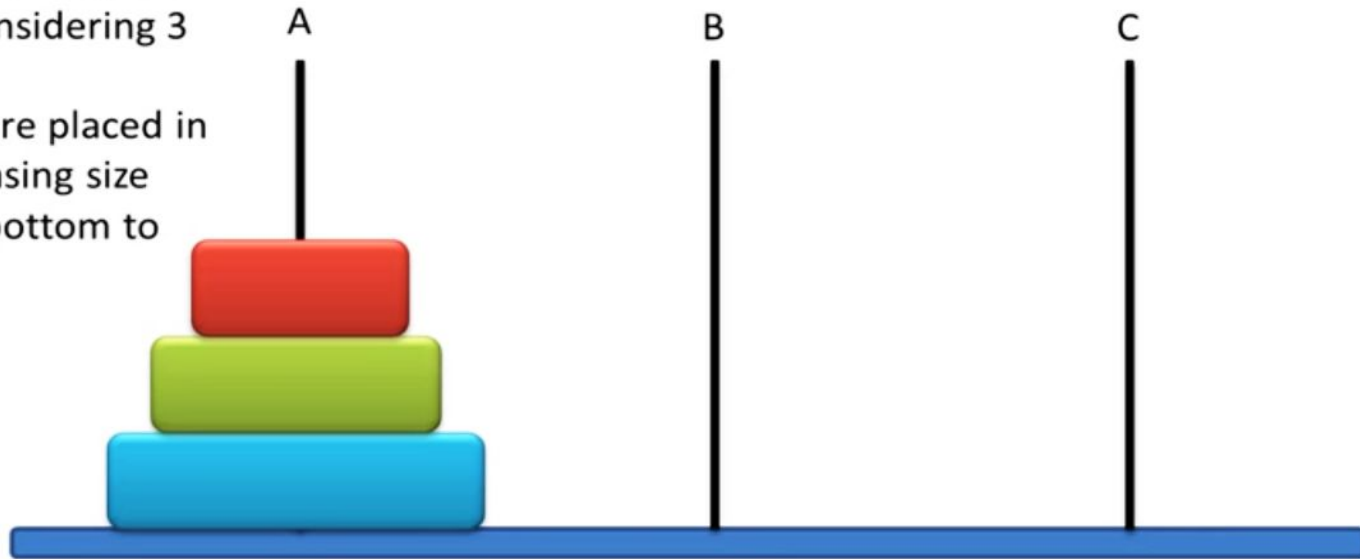
- Tower of Hanoi- a very famous game
- 3 pegs and N number of disks placed one over the other in decreasing order
- **Objective:** Move the disks one by one from the first peg to the last peg.
- **Condition-** Can't place a bigger disk on top of a smaller disk

# Example:

In this example we are considering 3 disks.

They are placed in decreasing size from bottom to top.

The 3 pegs are labeled A, B and C.



Our objective is to move the disks from peg A to peg C in such a way that they are in the same order: RED then GREEN then BLUE from top to bottom as they are in peg A.

# The tower of Brahma puzzle

There is a story about an Indian **temple in Kashi Vishwanath** which contains a large room with **three time-worn posts** in it, surrounded by **64 golden disks**. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks in accordance with the immutable rules of Brahma since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, **when the last move of the puzzle is completed, the world will end.**

# How to solve?

We can solve this problem by following 3 simple steps recursively:

We will use a general notation:

$T(N, Beg, Aux, End)$

- T*** - denotes our procedure
- N***- denotes the number of disks
- Beg***- the initial peg
- Aux***- is the auxiliary peg
- End***- is the final peg

# Steps

**Step 1:**  $T(N-1, Beg, End, Aux)$

**Step 2:**  $T(1, Beg, Aux, End)$

**Step 3:**  $T(N-1, Aux, Beg, End)$

Step 1 says: Move top  $(N-1)$  disks from *Beg* to *Aux* Peg

Step 2 says: Move 1 disk from *Beg* to *End* Peg

Step 3 says: Move top  $(N-1)$  disks from *Aux* to *End* Peg

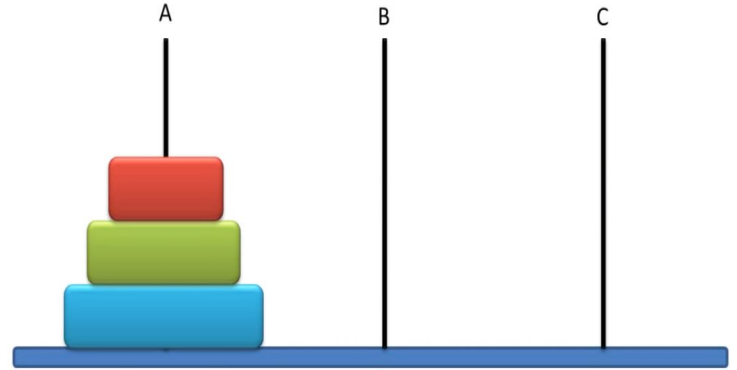


# Solution:

We have 3 disks Red, Green and Blue all placed in peg A.

So,  $N = 3$  (Number of disks)

Therefore, we will start with  
 $T(3, A, B, C)$



# Solution:

We will follow the three steps recursively to find the moves:

Step 1:  $T(N-1, Beg, End, Aux)$

Step 2:  $T(1, Beg, Aux, End)$

Step 3:  $T(N-1, Aux, Beg, End)$

At the starting,

**$N=3, Beg=A, Aux=B, End=C$**

So, we will apply the three steps on this:

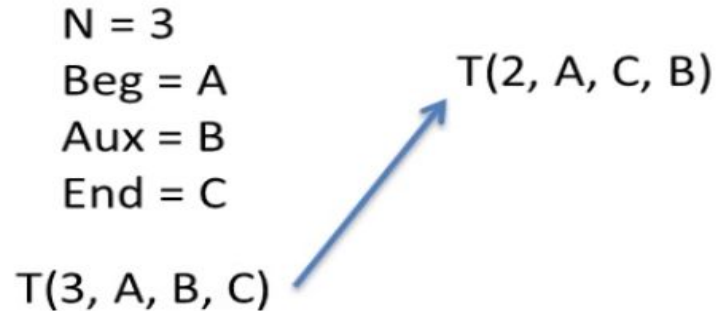
**$T(3, A, B, C)$**

# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$

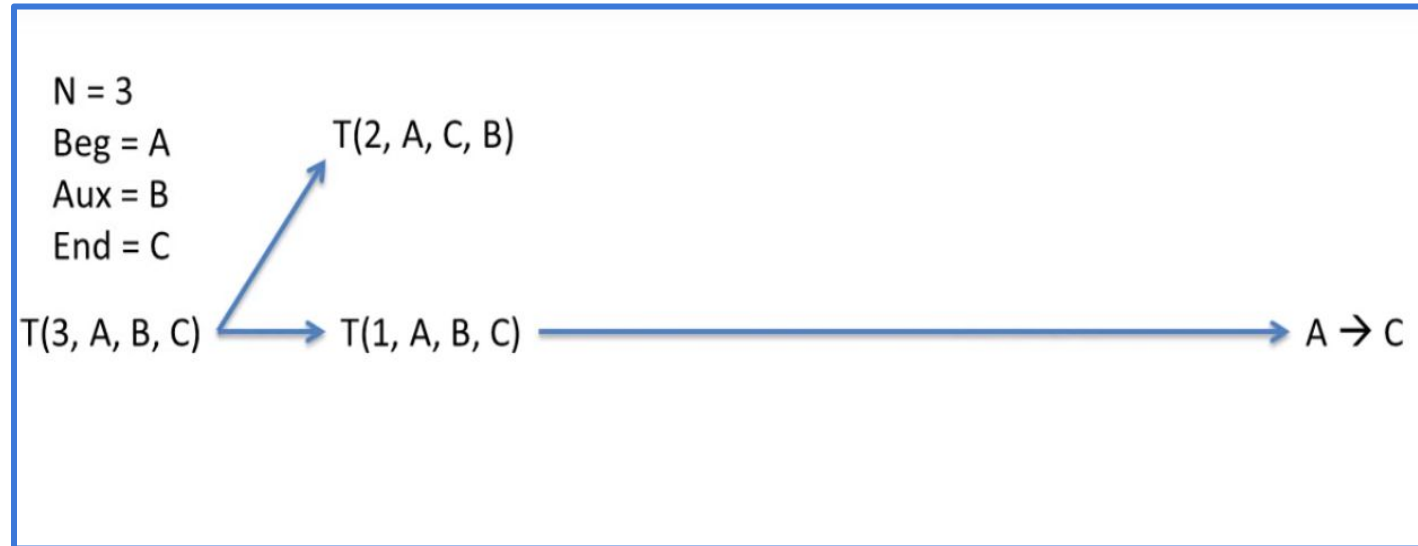


# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$

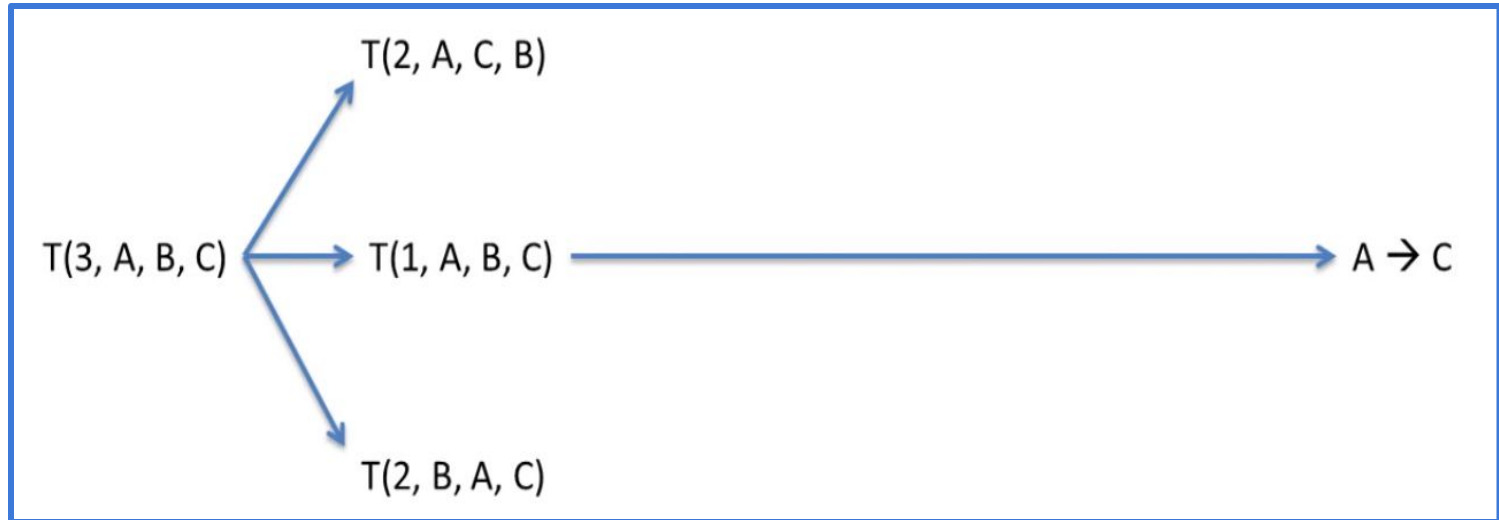


# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$



# Solution:

Step 1:  $T(N-1, Beg, End, Aux)$

Step 2:  $T(1, Beg, Aux, End)$

Step 3:  $T(N-1, Aux, Beg, End)$

Now, we will apply the three steps on this:

**$T(2, A, C, B)$**

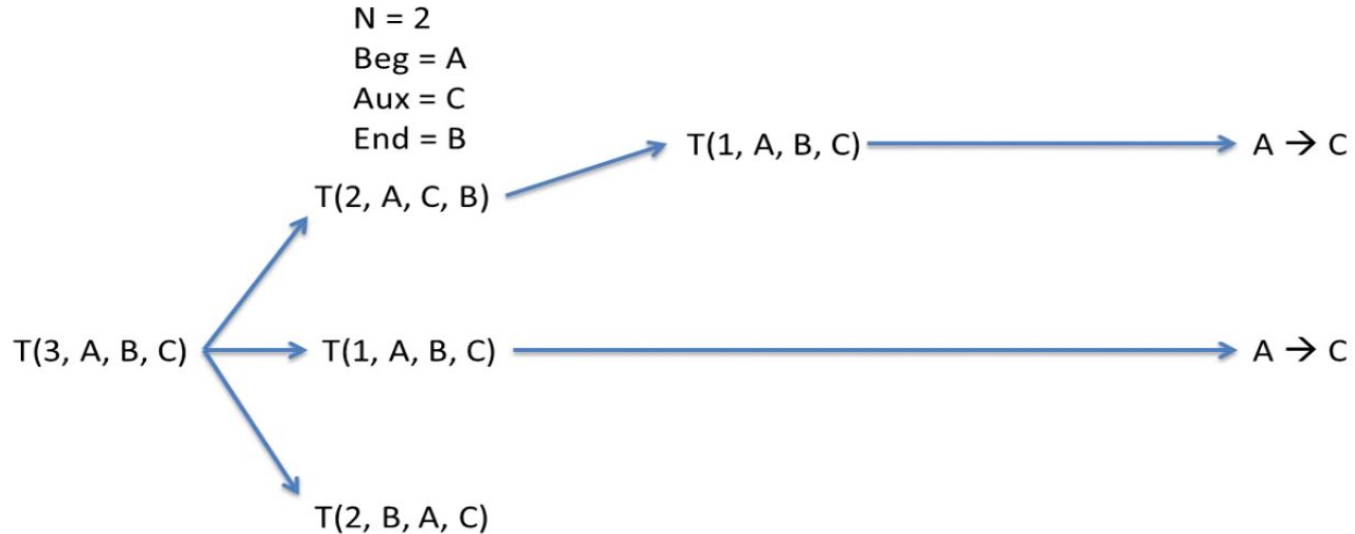
**$N=2, Beg=A, Aux=C, End=B$**

# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$

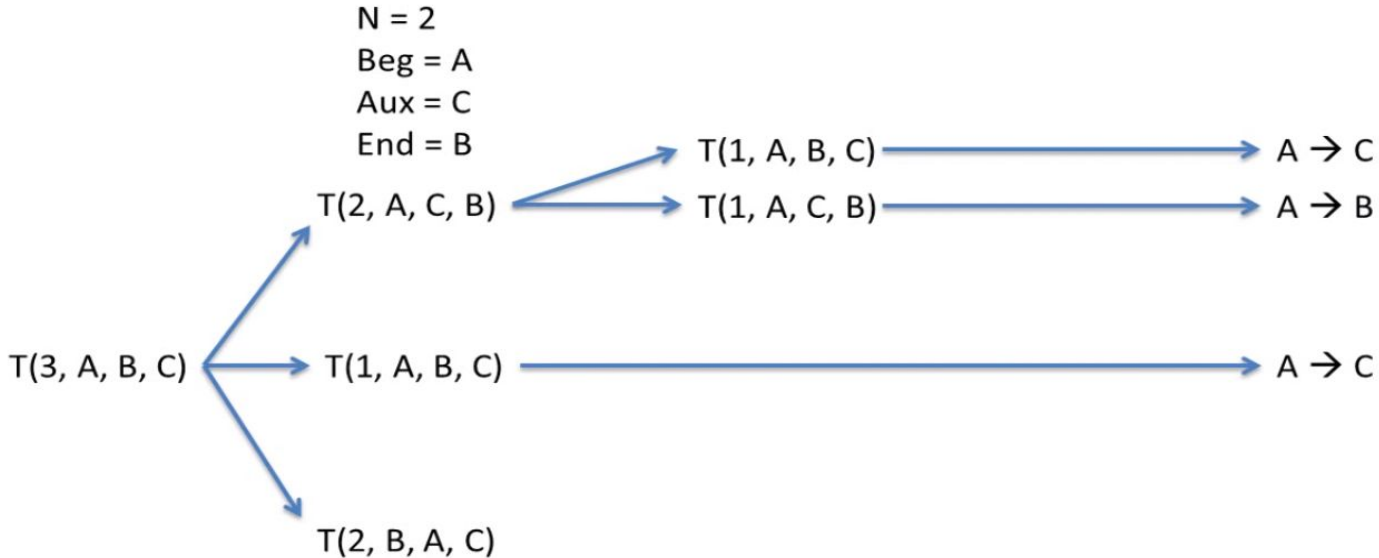


# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$



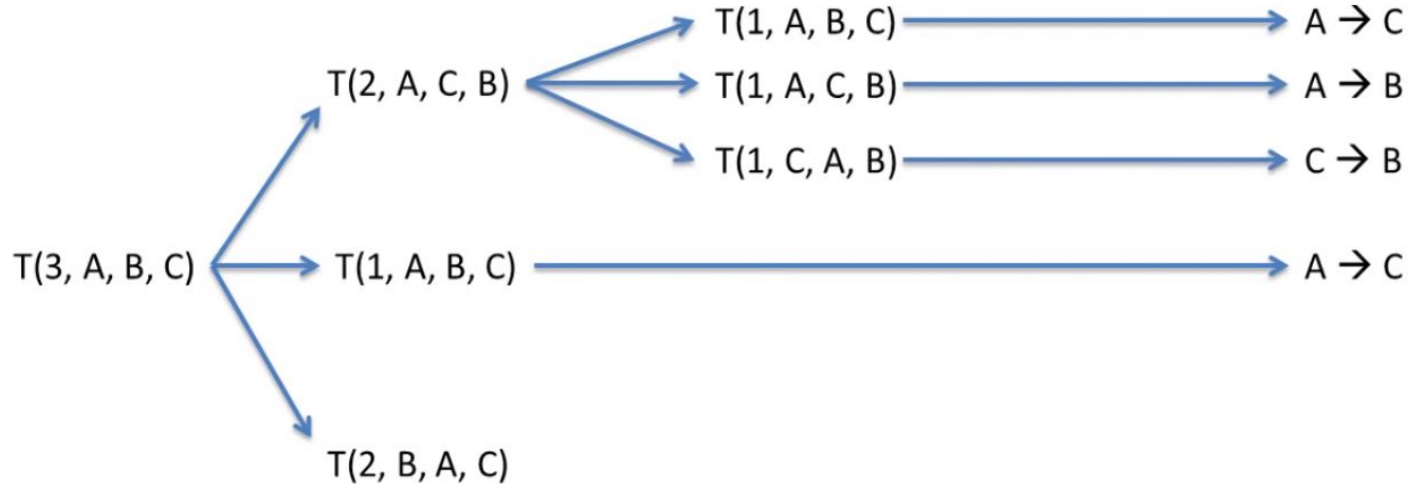


# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$



# Solution:

Step 1:  $T(N-1, Beg, End, Aux)$

Step 2:  $T(1, Beg, Aux, End)$

Step 3:  $T(N-1, Aux, Beg, End)$

Now, we will apply the three steps on this:

**$T(2, B, A, C)$**

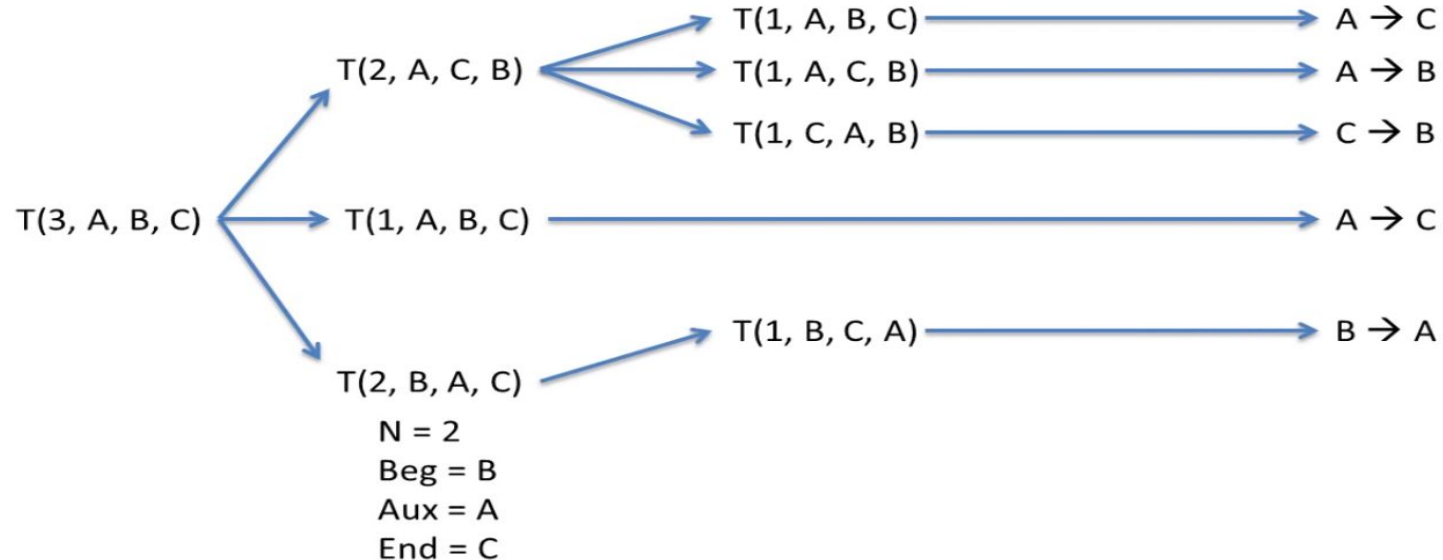
**$N=2, Beg=B, Aux=A, End=C$**

# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$

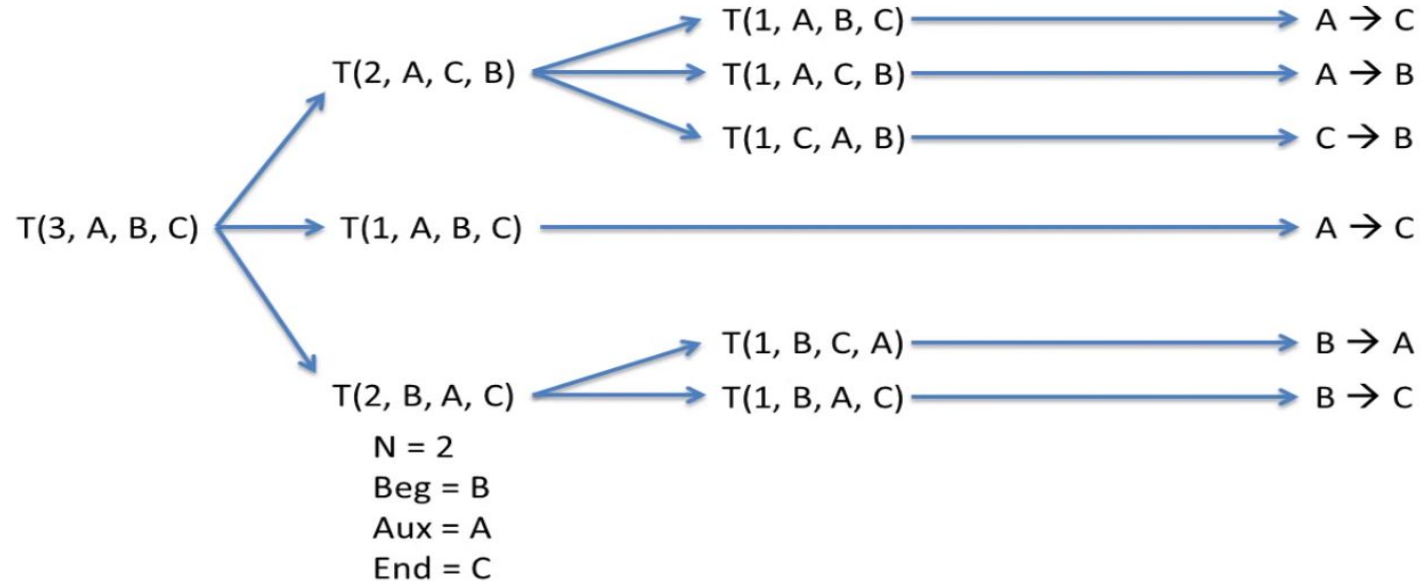


# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$

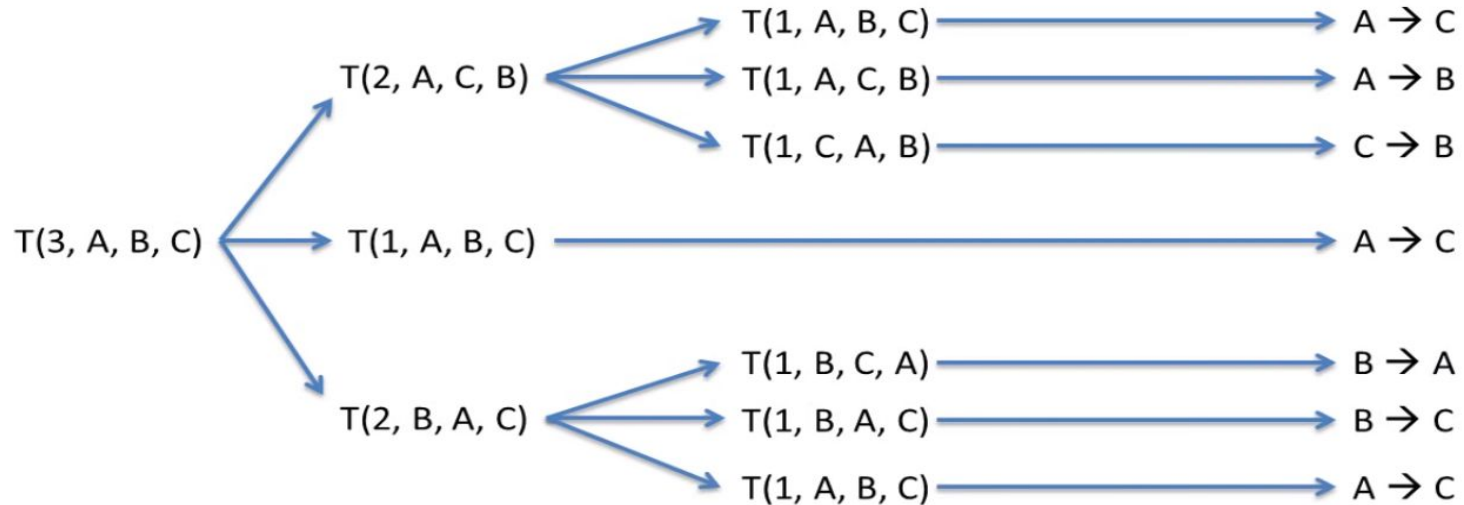


# Solution:

Step 1:  $T(N-1, \text{Beg}, \text{End}, \text{Aux})$

Step 2:  $T(1, \text{Beg}, \text{Aux}, \text{End})$

Step 3:  $T(N-1, \text{Aux}, \text{Beg}, \text{End})$



# Solution:

So, we have the moves! By taking these moves, we can reach to the final solution.

Moves

A → C

A → B

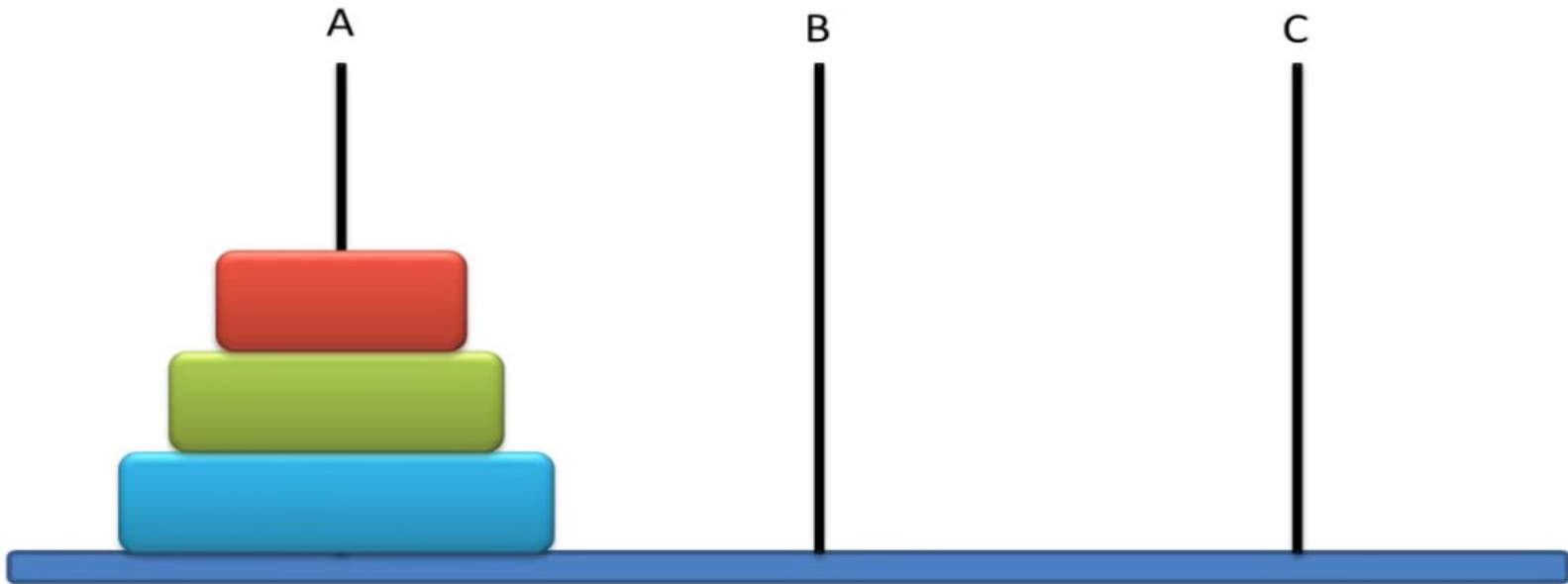
C → B

A → C

B → A

B → C

A → C



# Algorithm: Tower of Hanoi

```
Hanoi(N, Beg, End, Aux)
```

```
    if N == 1
```

```
        move disk from Beg to End
```

```
    else
```

```
        Hanoi(N - 1, Beg, Aux, End)
```

```
        move disk from Beg to End
```

```
        Hanoi(N - 1, Aux, End, Beg)
```

# Analysis of Tower of Hanoi

Hanoi(N, Beg, End, Aux)

```
if N == 1
    move disk from Beg to End    //constant time c1
else
    Hanoi(N - 1, Beg, Aux, End)  //same problem of size n-1
    move disk from Beg to End
    Hanoi(N - 1, Aux, End, Beg)  //same problem of size n-1
```

$$T(n) = 2T(n-1) + c$$



# Formative Assessment!