Algorithm (Part 2) – Recursion

Solve a problem by solving smaller sub-problems of itself

Let’s divide and conquer…

Contents

- **Recursion: what and how?**
  - **Termination condition**
  - **Runtime error: stack overflow**
- Simple example: Fibonacci sequence
  - call itself more than once
  - Speedup through Memorization
- General procedure: programming with recursion
- Three more examples
  - Path Search (1): Lattice Path
  - Path Search (2): Number Pyramid
  - The Classic: Tower of Hanoi

Recursion

- What is recursion?
  - A function may call itself
  - A great way to express **math functions**, since quite a number of math functions are defined using recursion
  - E.g.,

```
Factorial n!

if n is 0
    n! = 1
else
    n! = n * (n-1)!
```

```
Power of $x^n$

if n is 0
    $x^n$ = 1
else
    $x^n$ = x * $x^{n-1}$
```

Recursion – Factorial

- Transforming from description to C program code:

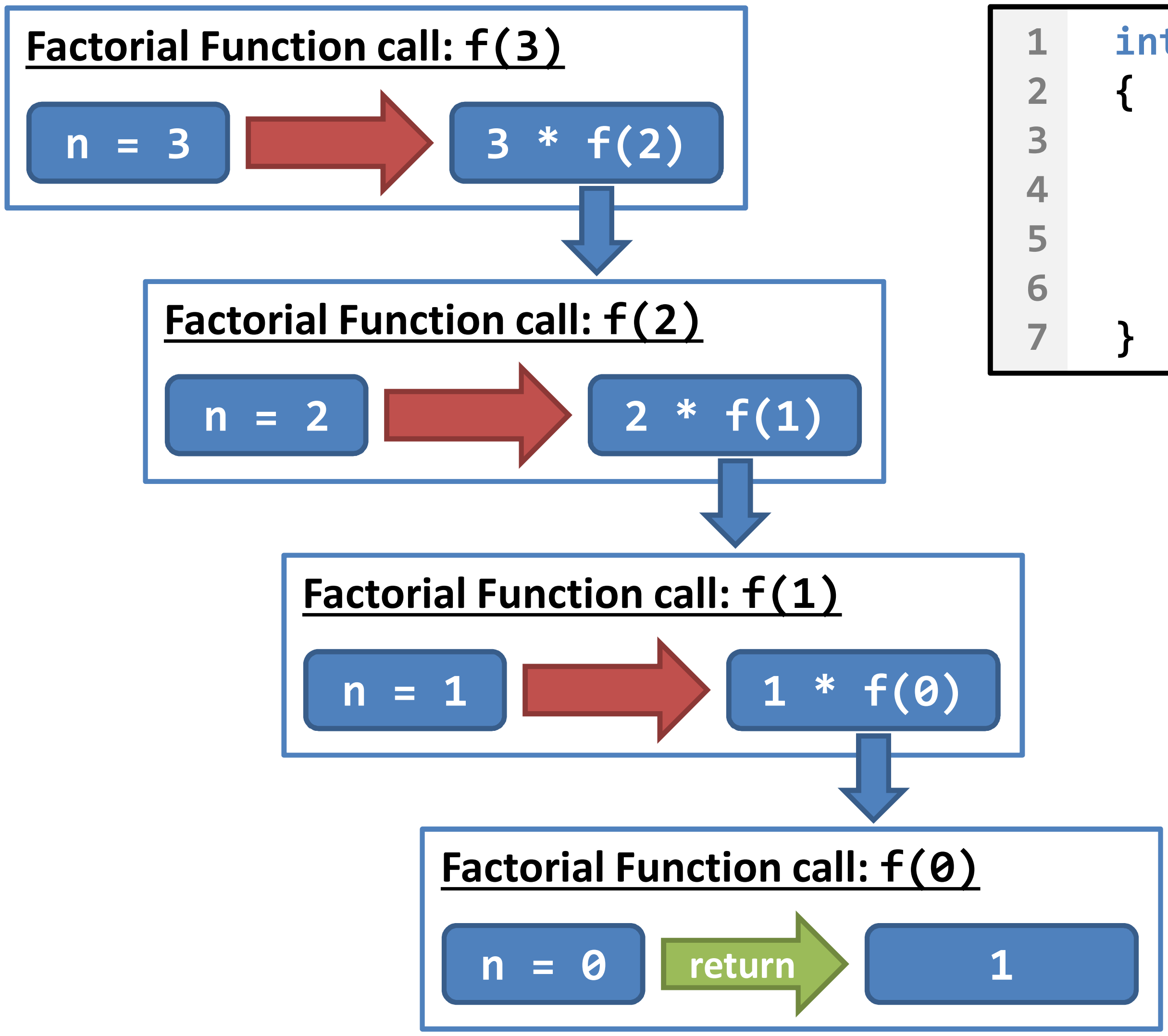
```
Factorial n!
if n is 0:
    n! = 1
else
    n! = n * (n - 1)!
```

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

  – See? Can you find them one-to-one corresponding!!

- Let's see how it works.

Recursion – Factorial example: 3!

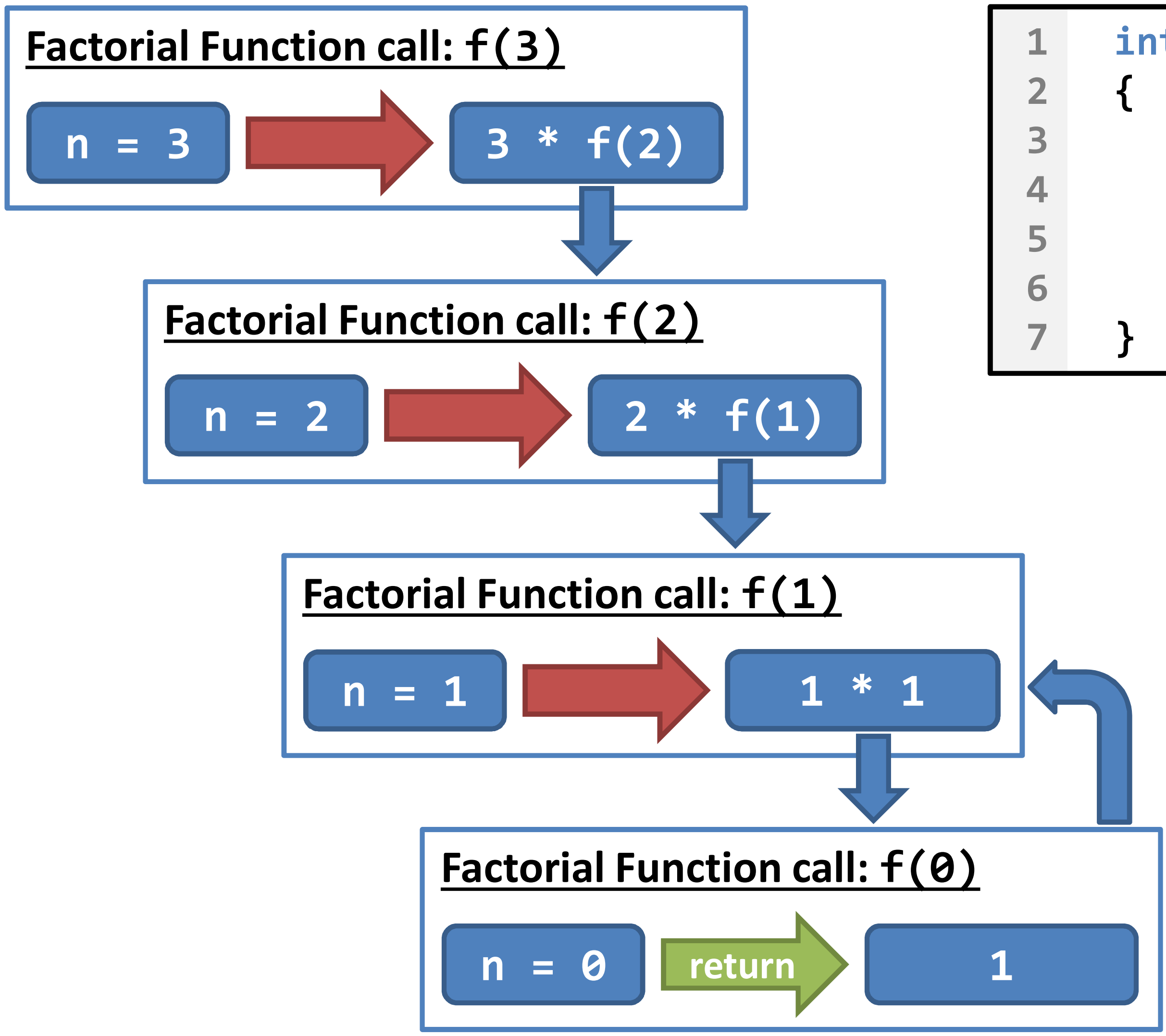


Factorial Function call: f(3)
n = 3
3 * f(2)
Factorial Function call: f(2)
n = 2
2 * f(1)
Factorial Function call: f(1)
n = 1
1 * f(0)
Factorial Function call: f(0)
n = 0
return
1

```
int f( int n )
{
    if ( n == 0 )
        return 1 ;
    else
        return n * f( n - 1 );
}
```

Functions are called in a level-by-level manner

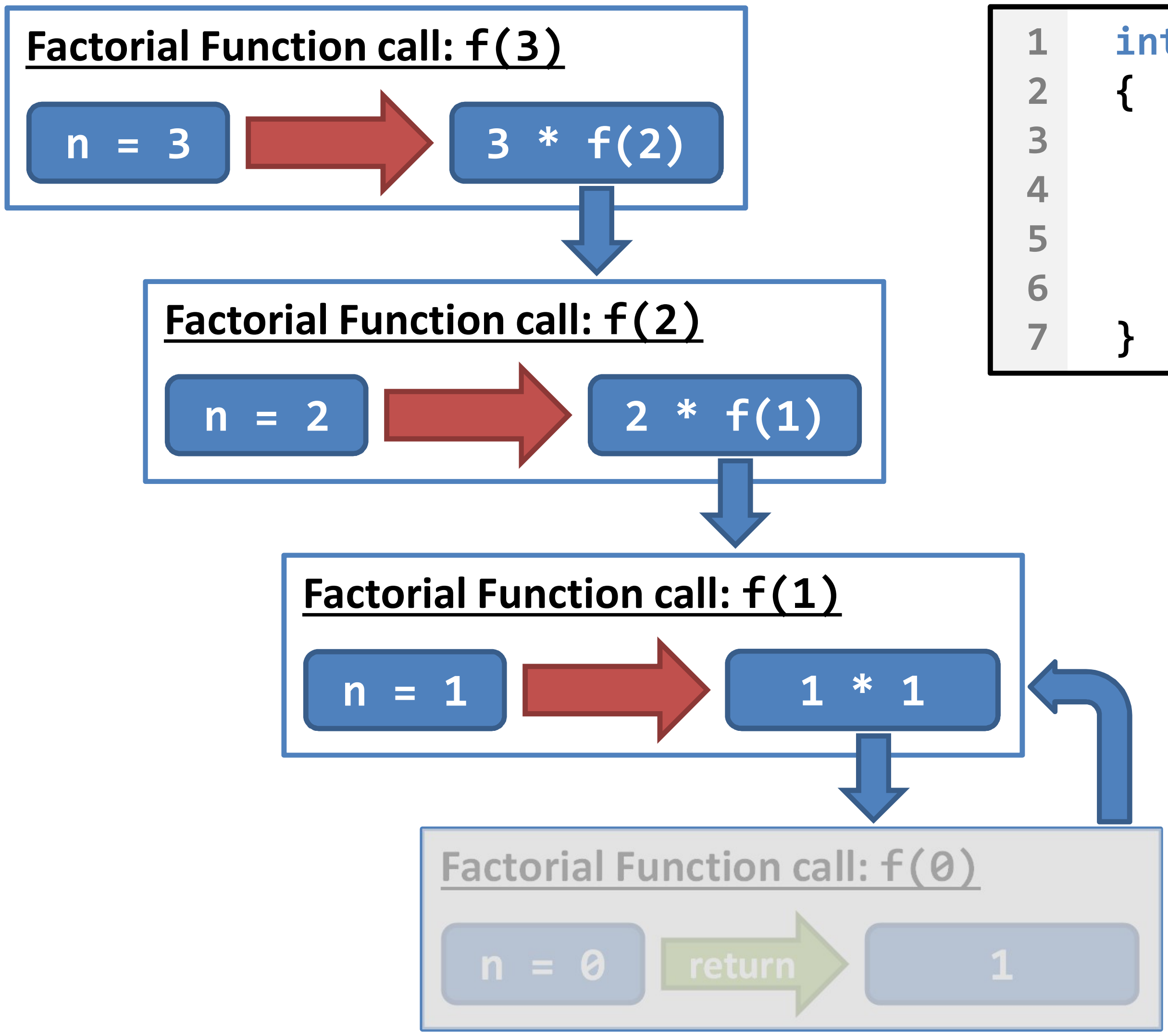
Recursion – Factorial example: 3!


Factorial Function call: f(3)
n = 3
3 * f(2)
Factorial Function call: f(2)
n = 2
2 * f(1)
Factorial Function call: f(1)
n = 1
1 * 1
Factorial Function call: f(0)
n = 0
return
1

```
int f( int n )
{
    if ( n == 0 )
        return 1 ;
    else
        return n * f( n - 1 );
}
```

The deepest layer, f(0), returns 1, and **replaces f(0) with 1** on the caller.

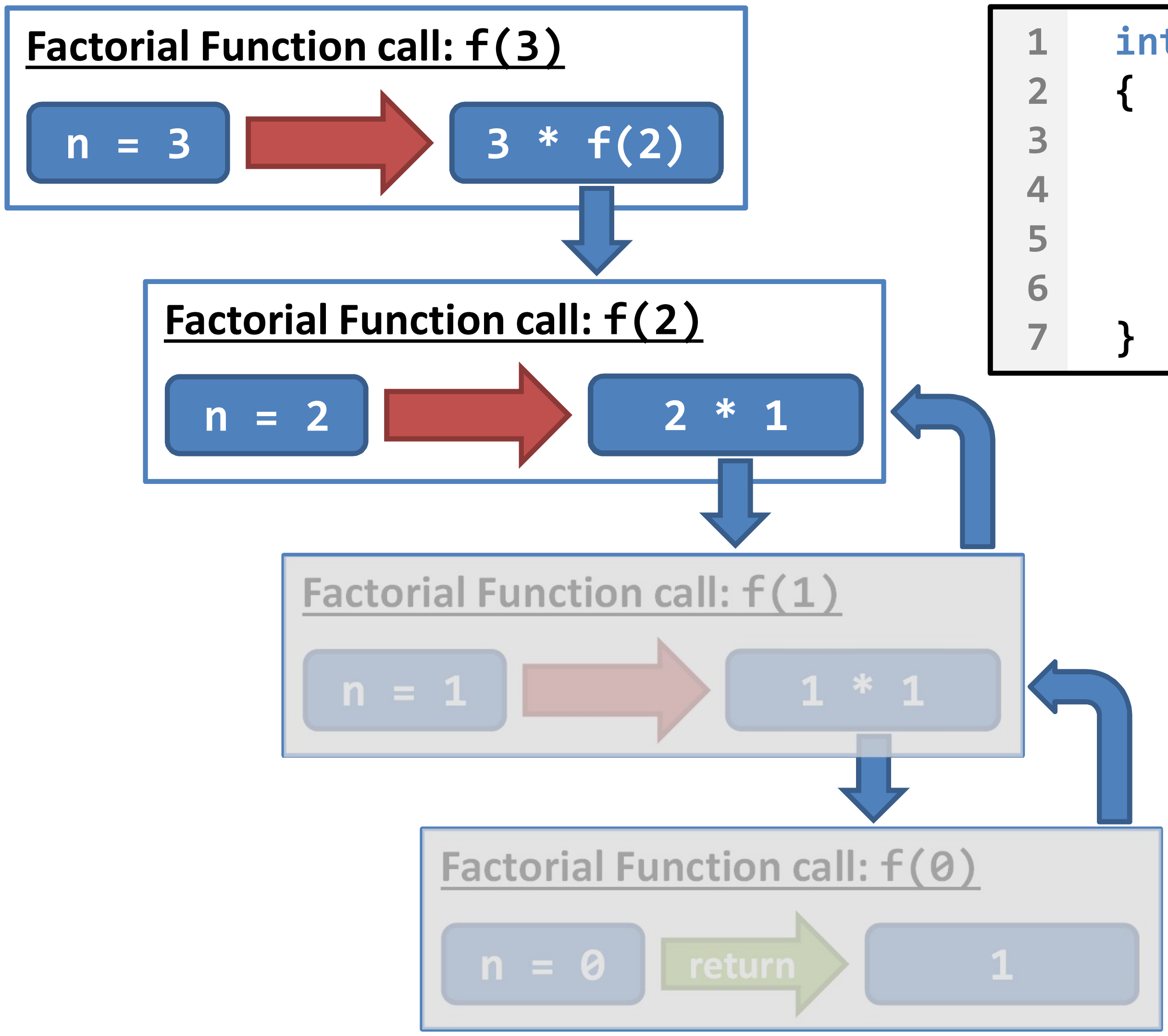
Recursion – Factorial example: 3!


Factorial Function call: f(3)
n = 3
3 * f(2)
Factorial Function call: f(2)
n = 2
2 * f(1)
Factorial Function call: f(1)
n = 1
1 * 1
Factorial Function call: f(0)
n = 0
return
1

```
int f( int n )
{
    if ( n == 0 )
        return 1 ;
    else
        return n * f( n - 1 );
}
```

f(0) is gone....

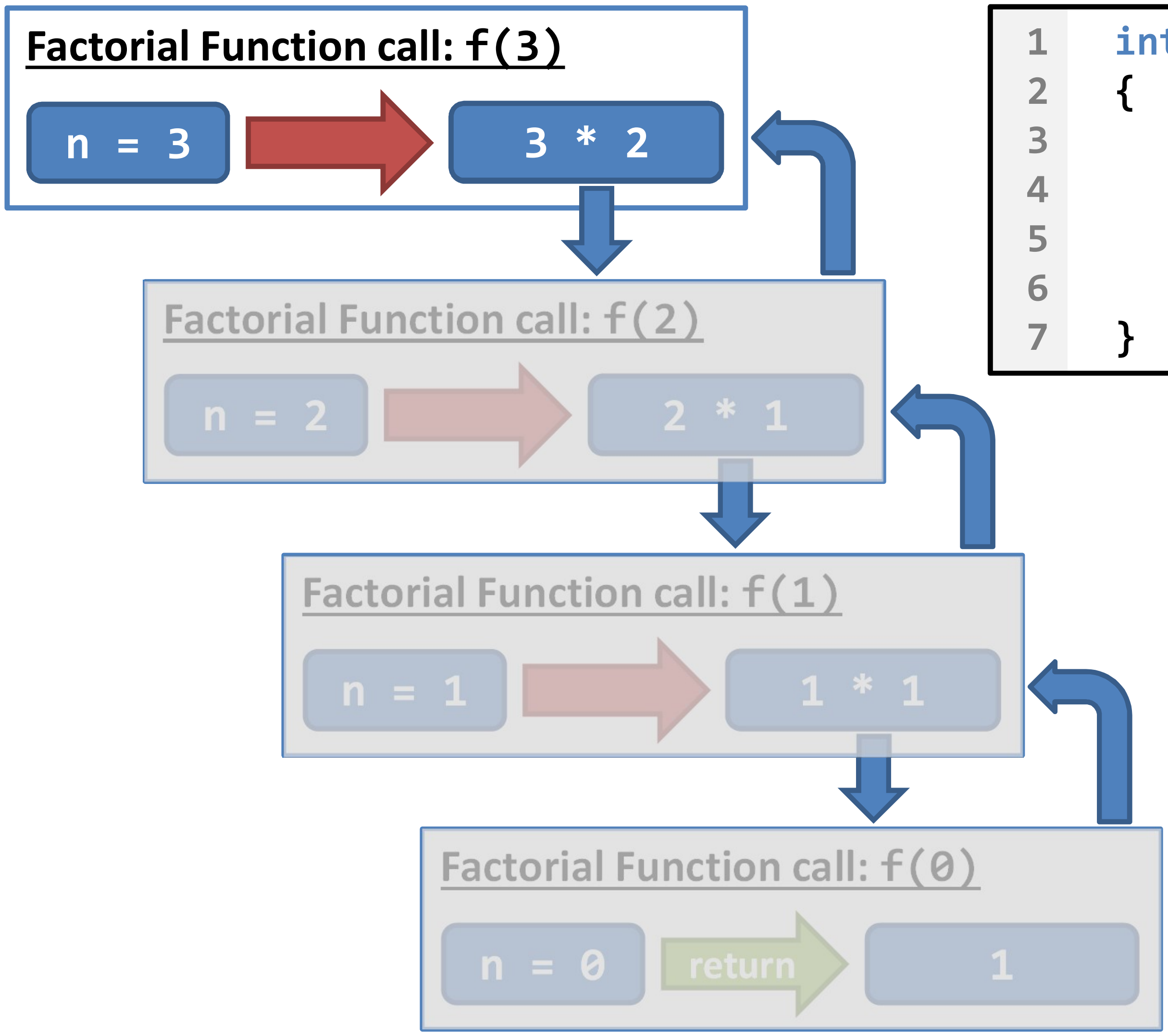
Recursion – Factorial example: 3!


Factorial Function call: f(3)
n = 3
3 * f(2)
Factorial Function call: f(2)
n = 2
2 * 1
Factorial Function call: f(1)
n = 1
1 * 1
Factorial Function call: f(0)
n = 0
return
1

```
int f( int n )
{
    if ( n == 0 )
        return 1 ;
    else
        return n * f( n - 1 );
}
```

f(1) returns 1, and **replaces f(1) with 1** on the caller side.

Recursion – Factorial example: 3!

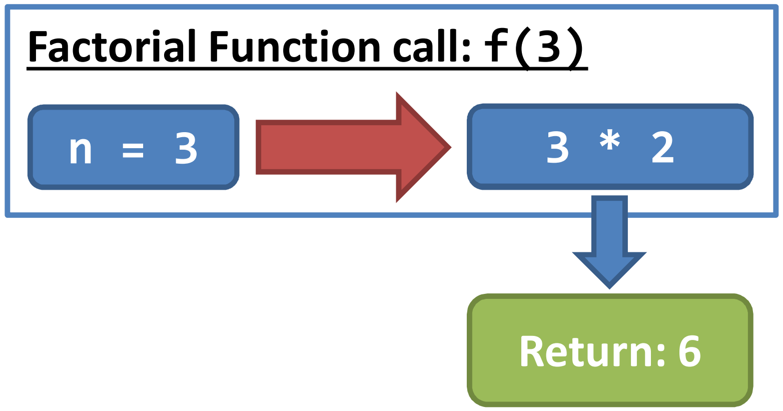


Factorial Function call: f(3)
n = 3
3 * 2
Factorial Function call: f(2)
n = 2
2 * 1
Factorial Function call: f(1)
n = 1
1 * 1
Factorial Function call: f(0)
n = 0
return
1

```
int f( int n )
{
    if ( n == 0 )
        return 1 ;
    else
        return n * f( n - 1 );
}
```

f(2) returns 2, and **replaces f(2) with 2** on the caller side.

Recursion – Factorial example: 3!



```
int f( int n )
{
    if ( n == 0 )
        return 1 ;
    else
        return n * f( n - 1 );
}
```

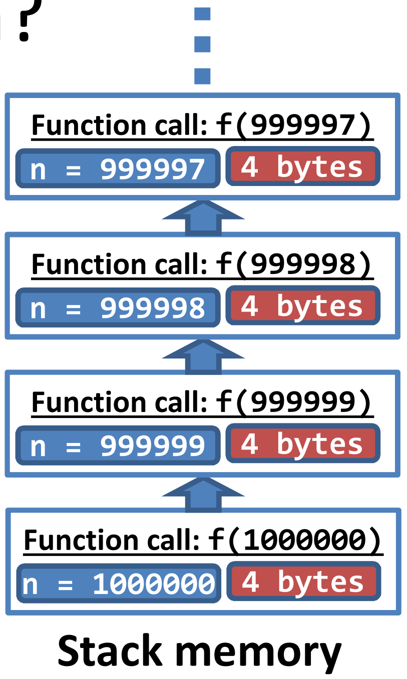
- So, we finally get `f(3) = 6`.
- Though it seems to involve a lot of work, the computer can run the steps in a very fast manner.
- Self Exercise: can you write the code for power?
  - `double power( double x , int n )`?

Recursion – Stack overflow

- One important component of a recursive function is the **termination condition**, i.e., lines 3-4 in the previous code.
- What happens if you miss the condition?

```
int f( int n ) {
    return n * f( n - 1 ) ;
}
```

  - A recursion cannot run indefinitely.
  - Then, the computer would generate a runtime error called **stack overflow**
  - Note: n should not be exceptionally large



Note: 1) recursion and stack: https://www.youtube.com/watch?v=Mv9NEXX1VHc#t=337
2) may increase a program's reserved stack size: diff. development software diff. ways

Contents

- Recursion: what and how?
  - Termination condition
  - Runtime error: stack overflow
- **Simple example: Fibonacci sequence**
  - **call itself more than once**
  - **Speedup through Memorization**
- General procedure: programming with recursion
- Three more examples
  - Path Search (1): Lattice Path
  - Path Search (2): Number Pyramid
  - The Classic: Tower of Hanoi

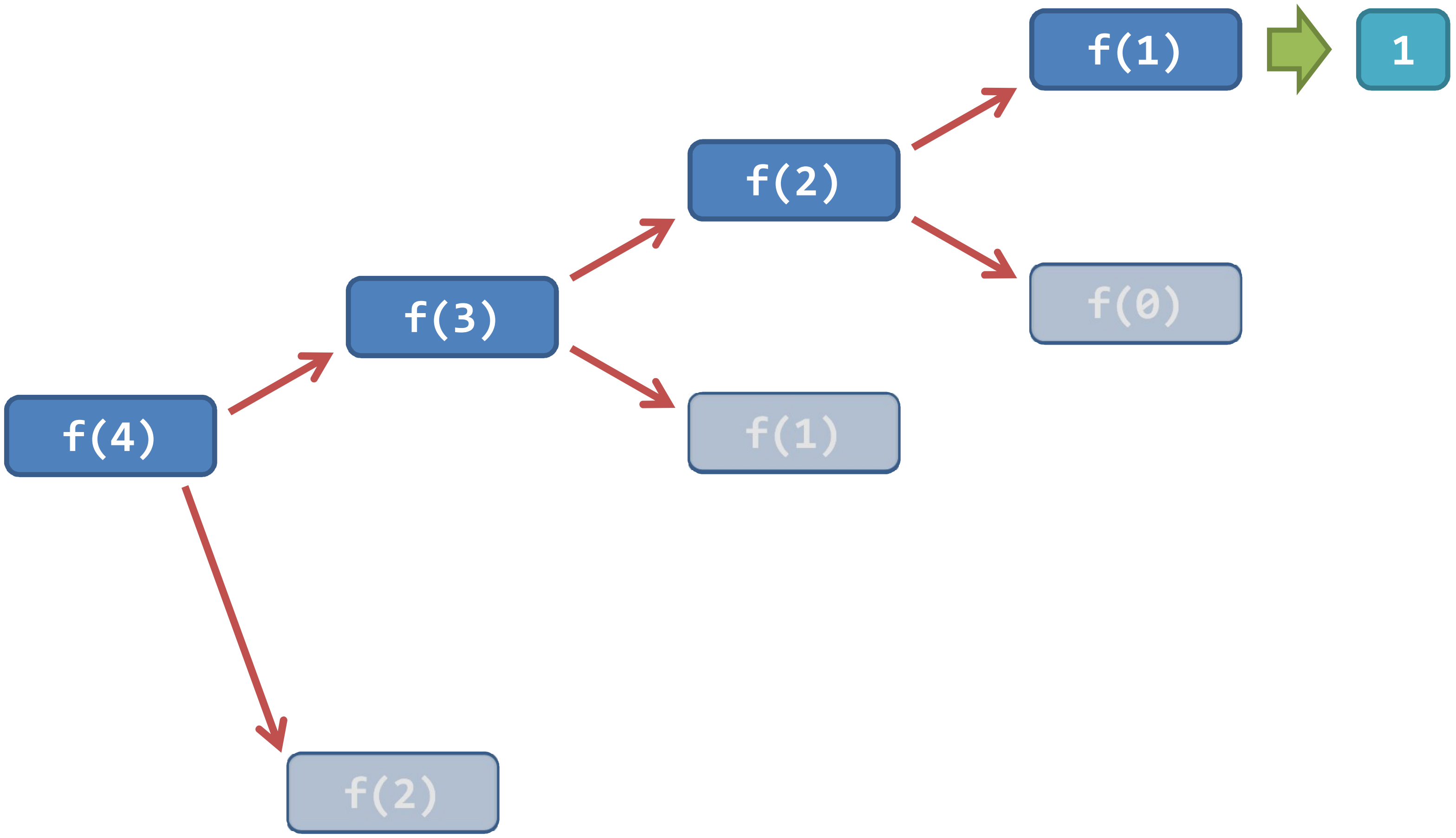
Another Famous Recursion Example

- The Fibonacci number introduction video: http://www.ted.com/talks/arthur_benjamin_the_magic_of_fibonacci_numbers
- Fibonacci number is a sequence of number:
  - 0, 1, 1, 2, 3, 5, 8, 13, ……
  - The i-th number (i>2) **is a sum of the previous two**.

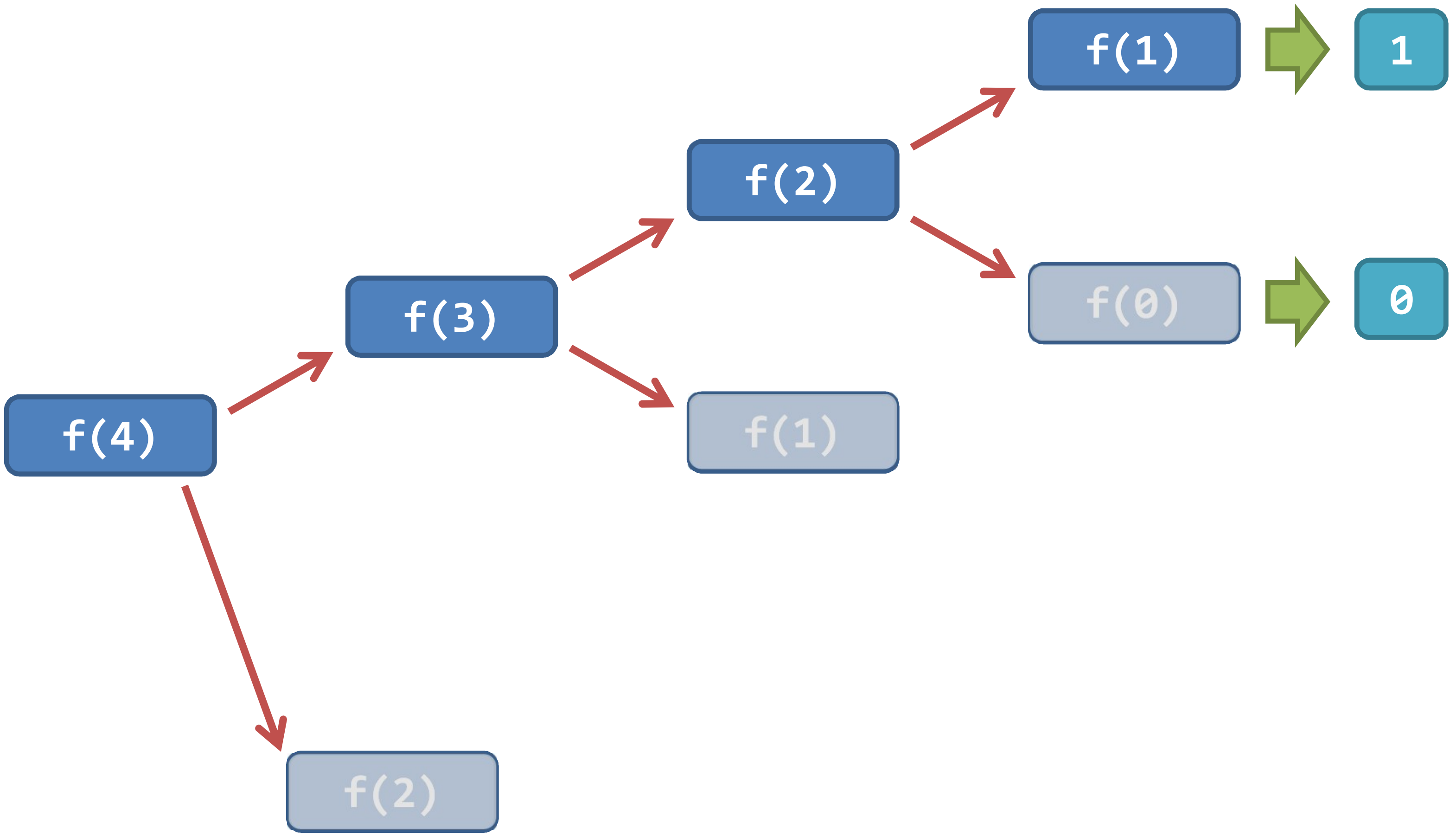
```
      n-th Fibonacci number: f(n)

if n is 0:  return 0 ;
if n is 1:  return 1 ;
if n is > 1:
    return f( n – 1 ) + f( n – 2 ) ;
```

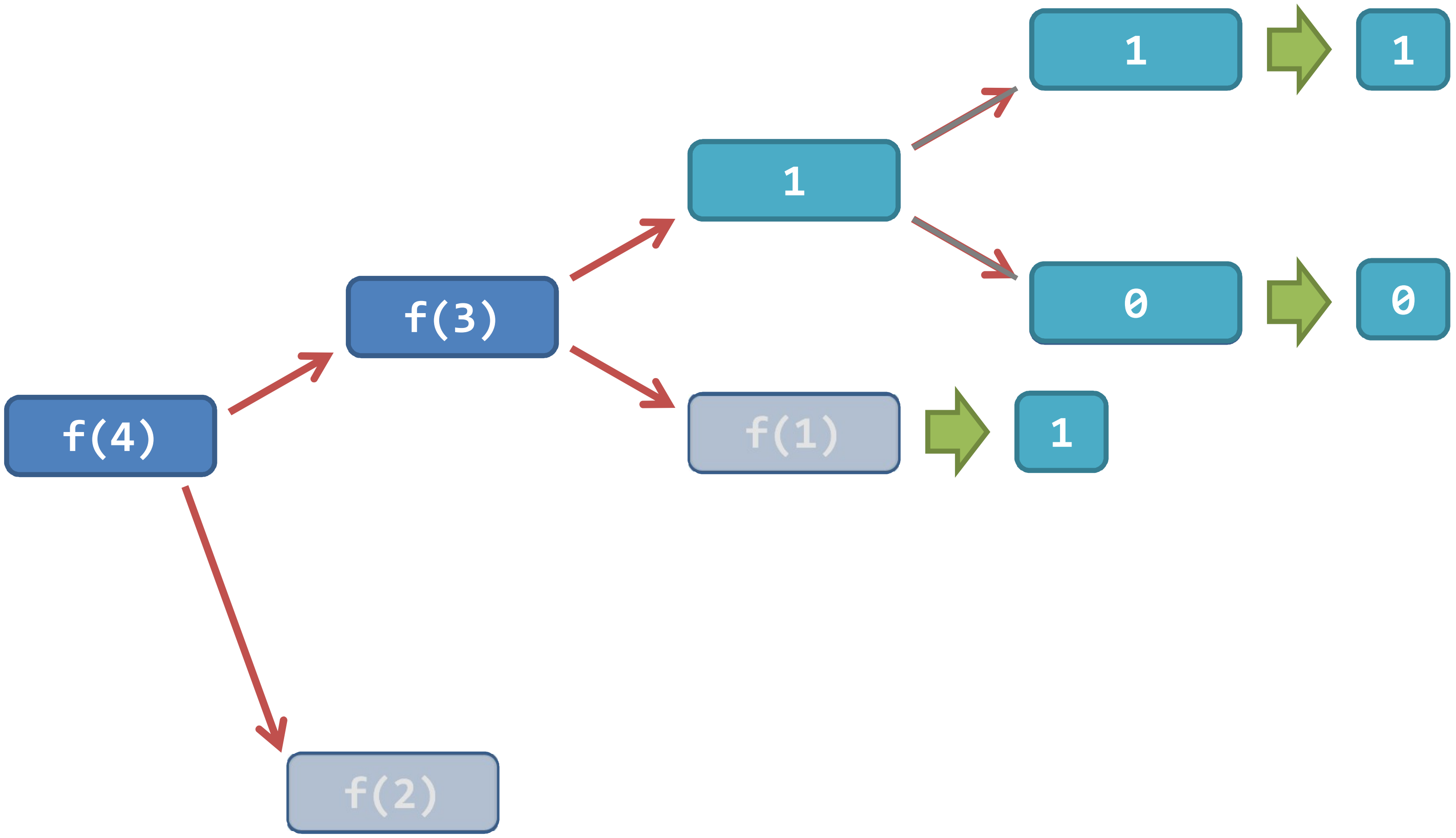
Fibonacci number: illustration



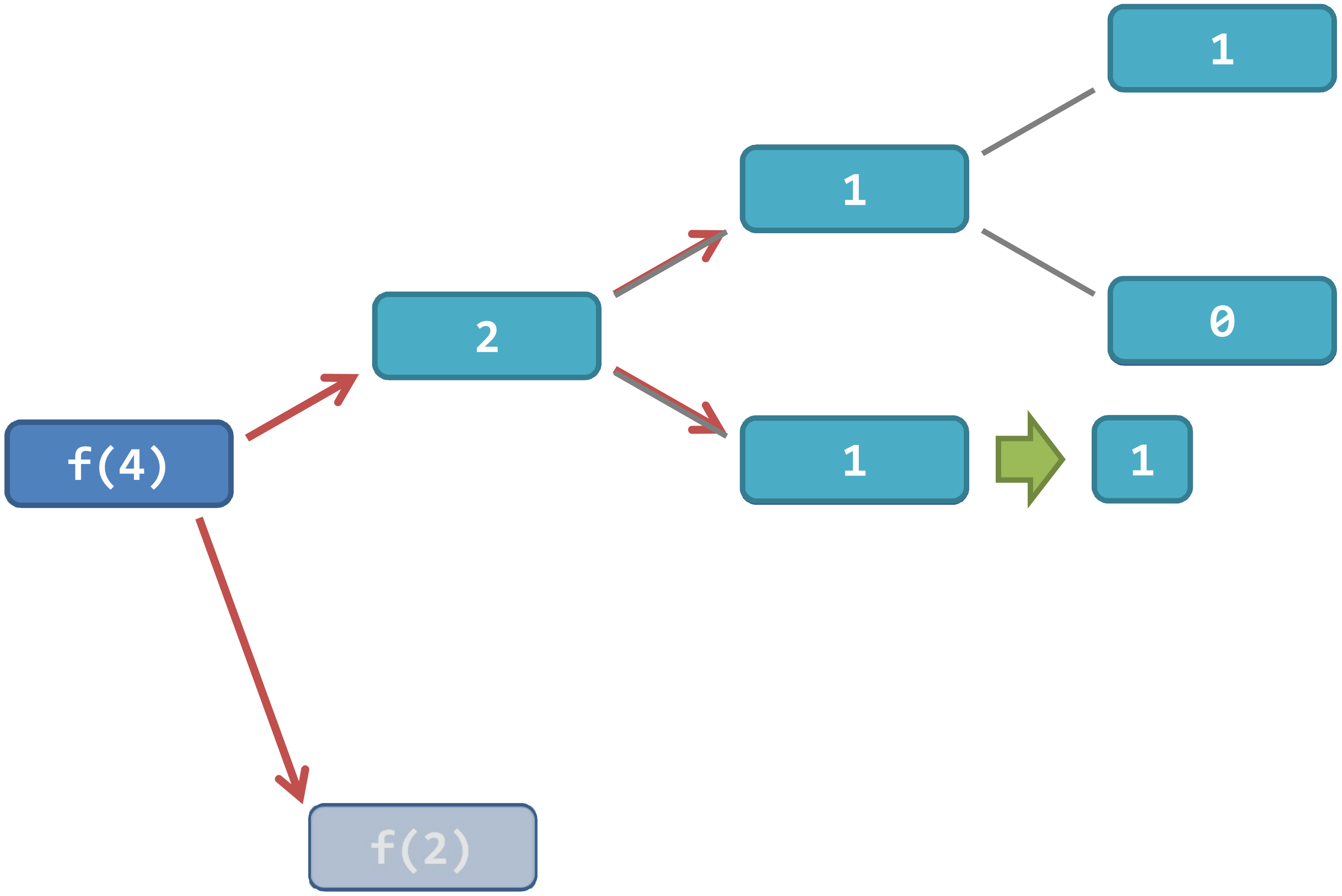
Fibonacci number: illustration



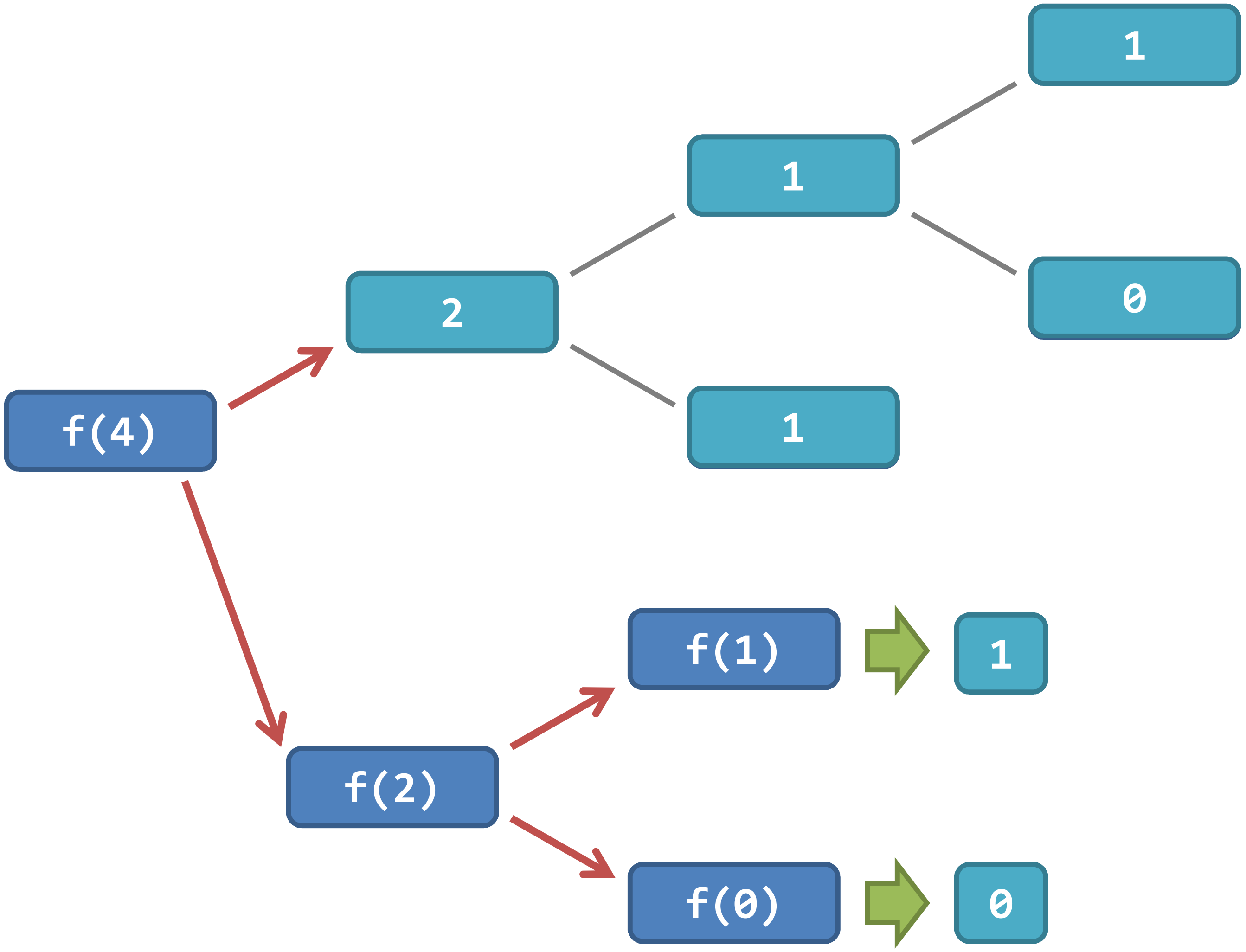
Fibonacci number: illustration



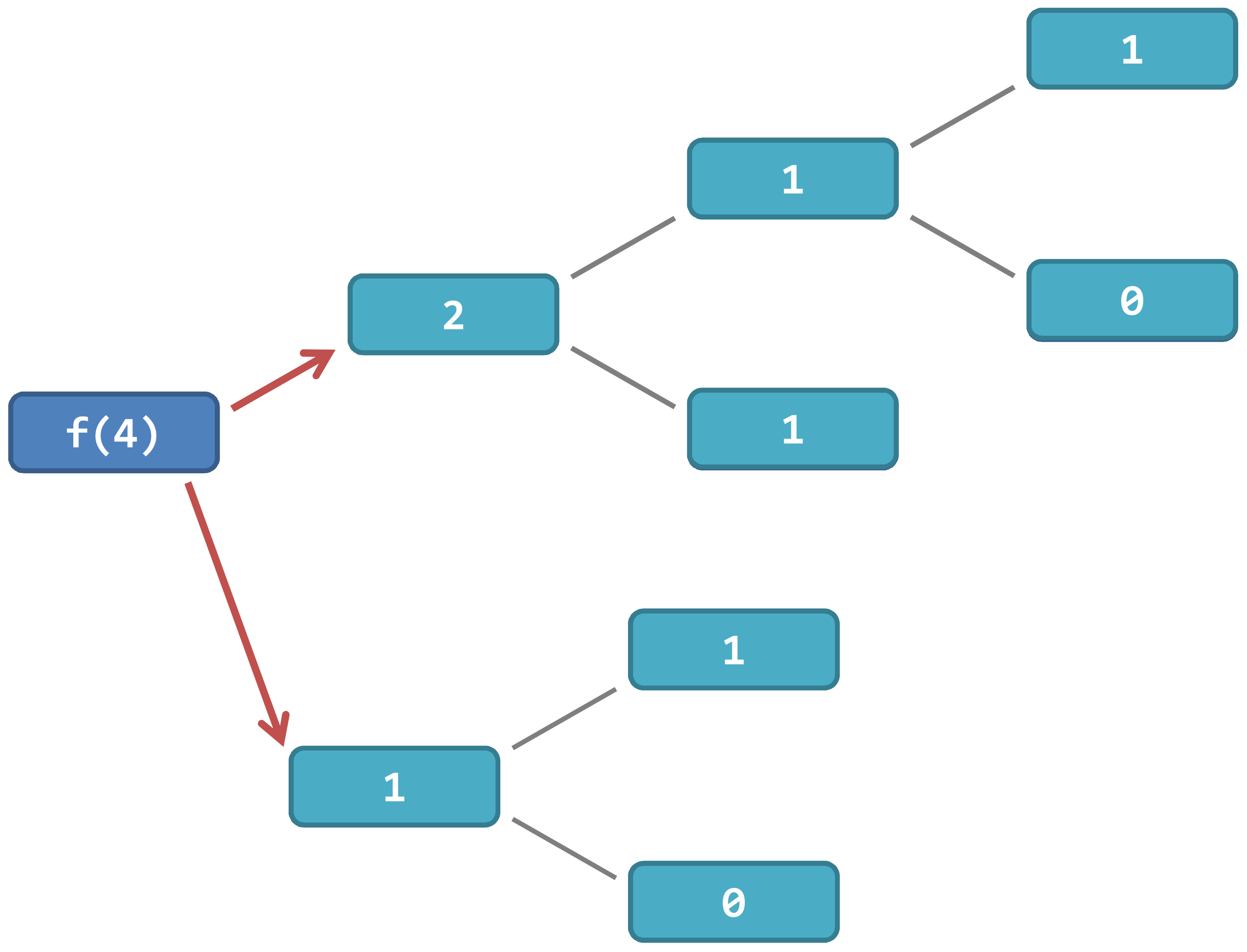
Fibonacci number: illustration



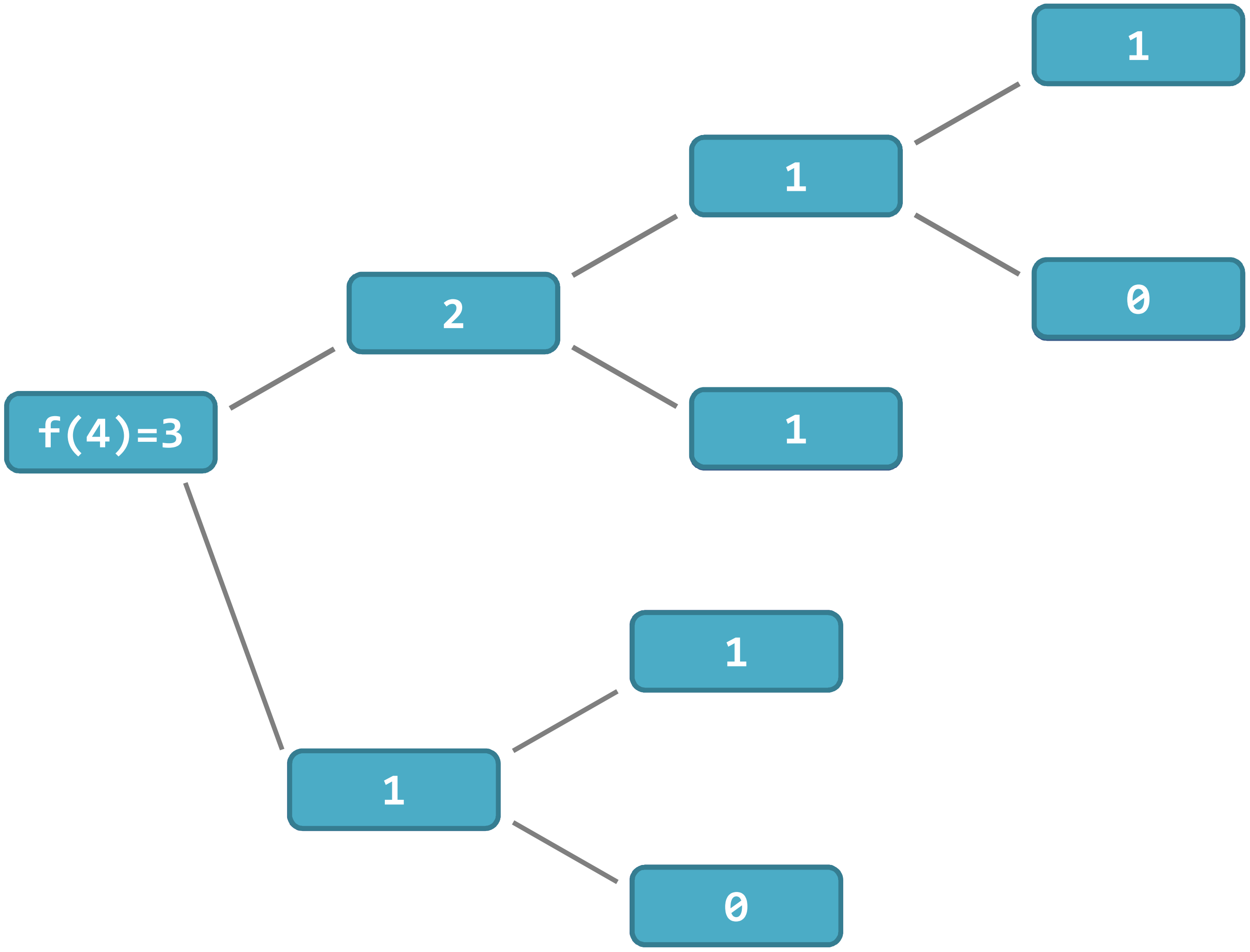
Fibonacci number: illustration



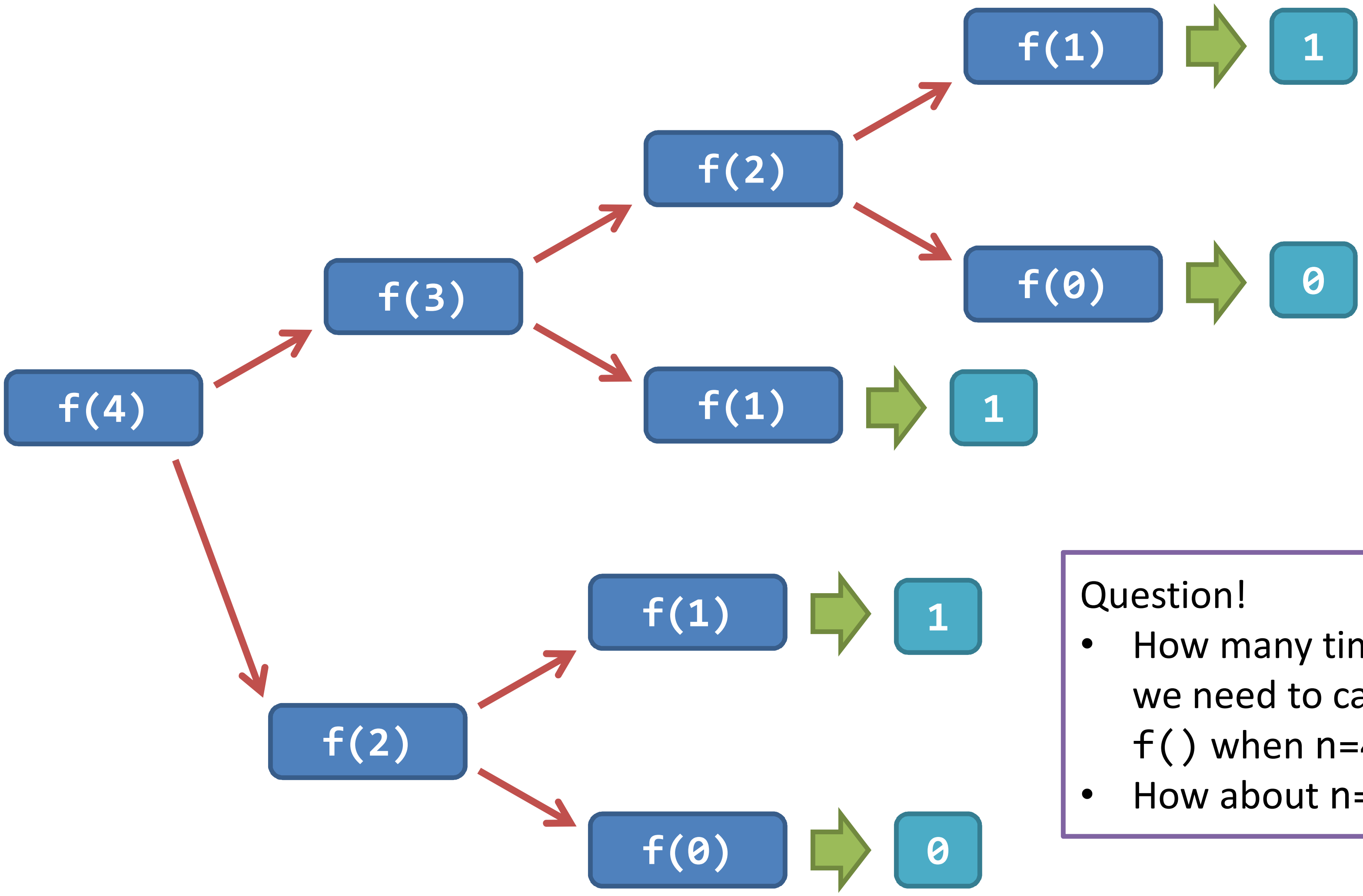
Fibonacci number: illustration



Fibonacci number: illustration



f(4) call graph



Question!
- How many times we need to call `f()` when n=4?
- How about n=5?

f(5) call graph

$$\begin{tikzcd}
 & & & & f(1) \\
 & & & f(2) \arrow[ur] \arrow[dr] & \\
 & & f(3) \arrow[ur] \arrow[dr] & & f(0) \\
 & f(4) \arrow[ur] \arrow[dr] & & f(1) & \\
f(5) \arrow[ur] \arrow[dddr] & & f(2) & & \\
 & & & f(1) & \\
 & & f(2) \arrow[ur] \arrow[dr] & & \\
 & f(3) \arrow[ur] \arrow[dr] & & f(0) & \\
 & & f(1) & &
\end{tikzcd}$$

Any Speedup?

- According to the previous page, there are lots of duplicated computation, can we do better?

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

Speedup through Memorization

- Use a global array to store the computed results

```
int speedup[ 10000 ] ;  // all initialized to -1

int fibonacci( int n )
{
   if ( speedup[ n ] == -1 )  // not computed yet
   {
      // memorize the newly computed result
      if ( n == 0 )
         speedup[ n ] = 0 ;
      else
      if ( n == 1 )
         speedup[ n ] = 1 ;
      else
         speedup[ n ] = fibonacci( n – 1 ) + fibonacci( n – 2 );
   }
   return speedup[ n ] ;
}
```

But… how to avoid global variables?

Speedup through Memorization

- Make it a function parameter and initialize it outside

```
// You should initialize "speedup" all to -1 (e.g., in main())

int fibonacci( int n , int speedup[] )
{
    if ( speedup[ n ] == -1 )  // not computed yet
    {
        // memorize the newly computed result
        if ( n == 0 )
            speedup[ n ] = 0 ;
        else
        if ( n == 1 )
            speedup[ n ] = 1 ;
        else
            speedup[ n ] = fibonacci( n – 1 , speedup )
                         + fibonacci( n – 2 , speedup );
    }
    return speedup[ n ] ;
}
```

Short Summary: Iteration VS Recursion

- It was proved by Alonzo Church and Alan Turing (in the **Church-Turing Thesis**) that:
  - Every recursive algorithm can be transformed into an iteration, and vice versa.
  - Their proof did not provide any procedure to do the transformation.

E.g., Computing by iteration

```
int fibonacci( int n )
{
   int values[ 10000 ] ;

   values[ 0 ] = 0 ;
   values[ 1 ] = 1 ;

   for ( i = 2 ; i <= n ; i ++ )
      values[ i ] = values[ i - 1 ] + values[ i - 2 ] ;

   return values[ n ] ;
}
```

Contents

- Recursion: what and how?
  – Termination condition
  – Runtime error: stack overflow
- Simple example: Fibonacci sequence
  – call itself more than once
  – Speedup through Memorization
- **General procedure: programming with recursion**
- Three more examples
  – Path Search (1): Lattice Path
  – Path Search (2): Number Pyramid
  – The Classic: Tower of Hanoi

General procedure

- Calling a recursion function is like asking a question
- Pay attention to three basic but important things:
  - **Understand the problem & Identify sub-problems**
    - Solve a "bigger" problem by solving "same smaller" problems.
    - Any relationship between bigger and smaller problems?
      - Recurrence relationship!!! Find out the "PATTERN"!!!
    - What is/are the parameters? Any data passing between?
  - **Design function prototype** for "**data passing" & "return"**
  - When to **Terminate**?

Note: try smaller problems with "printf" to debug; only after your program works, accelerate it by memorization

Contents

- Recursion: what and how?
  - Termination condition
  - Runtime error: stack overflow
- Simple example: Fibonacci sequence
  - call itself more than once
  - Speedup through Memorization
- General procedure: programming with recursion
- **Three more examples**
  - **Path Search (1): Lattice Path**
  - **Path Search (2): Number Pyramid**
  - **The Classic: Tower of Hanoi**

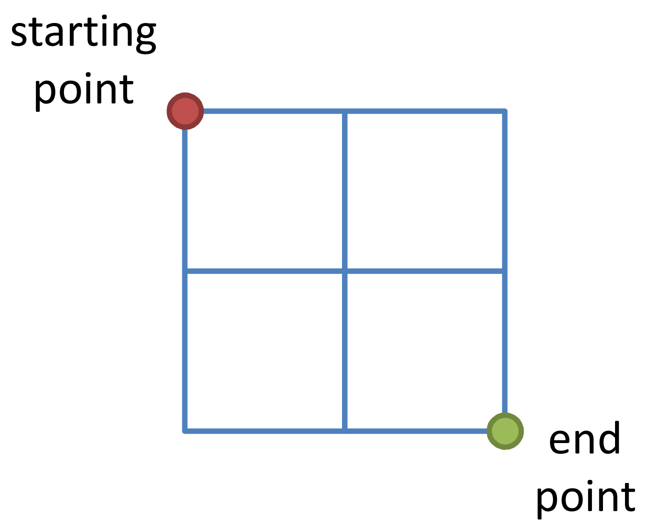
Does it look like a maze?

Map of Manhattan, New York

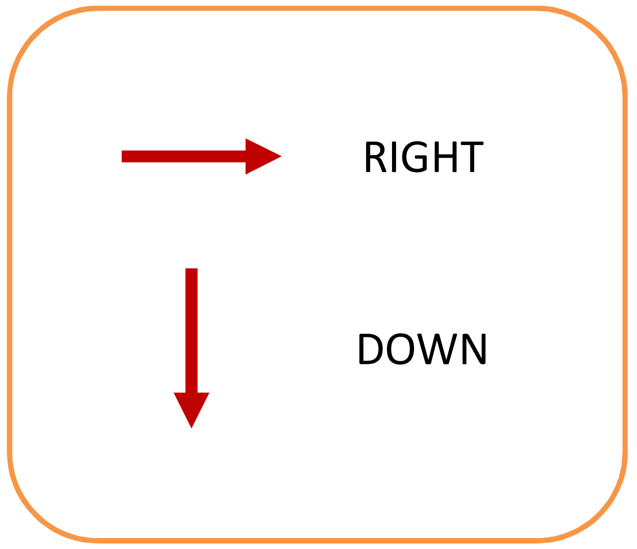
By the way, do you know what is Manhattan (L1) Distance?

Path finding – Lattice path

- In such a maze, how many distinct (different) paths we can travel from one point to another?



2 x 2 cells (or lattice)

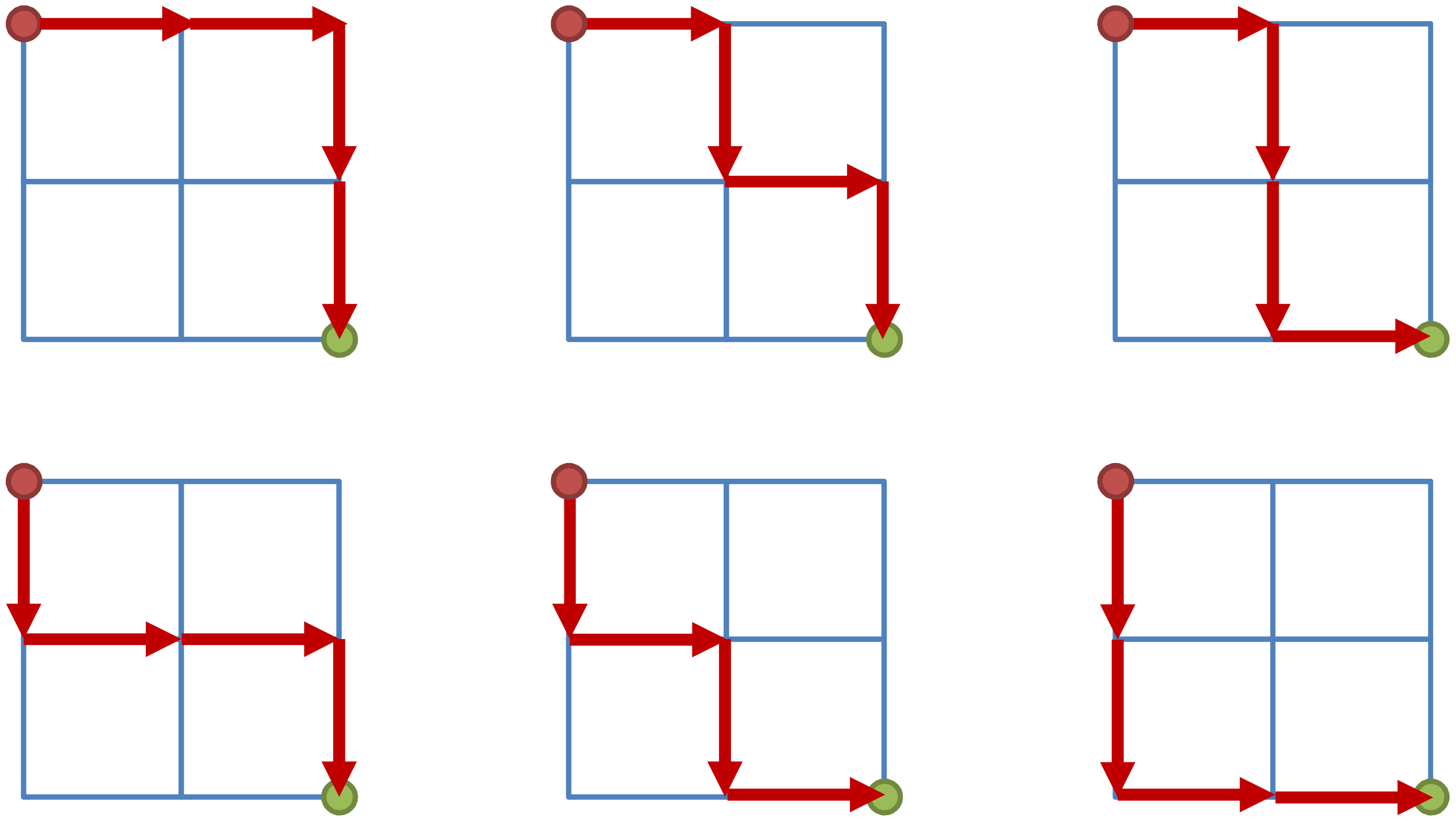


Allowed Movement

Note: Calling a recursion function is like asking a question

Path finding – Lattice path

- In a 2 x 2 lattice: 6 paths

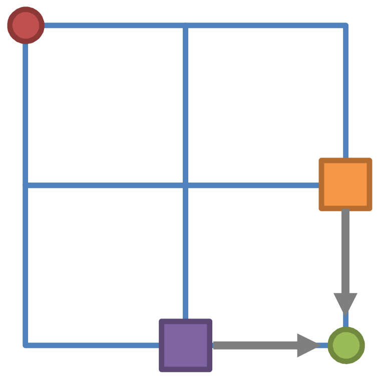


Path finding – Lattice path

- So, 2 x 2 is nice; we can verify by human effort.
- How about a 10 x 10 lattice?
  - There are 184,756 distinct paths.
  - We can't count using our fingers (and toes)!
- Either of the following approaches work:
  - Derive a close-form solution (ENGG 2430)
    - E.g., mathematical formulae
  - Solve the problem by programming (ESTR 1002)

Lattice path: The insight!

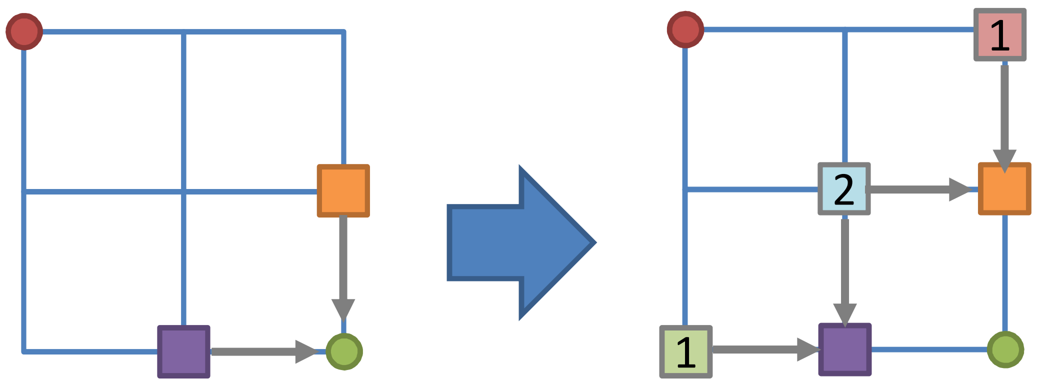
- Let's check this out:



	# of paths reach 🟧	3
+)	# of paths reach 🟪	3
	# of paths reach 🟢	6

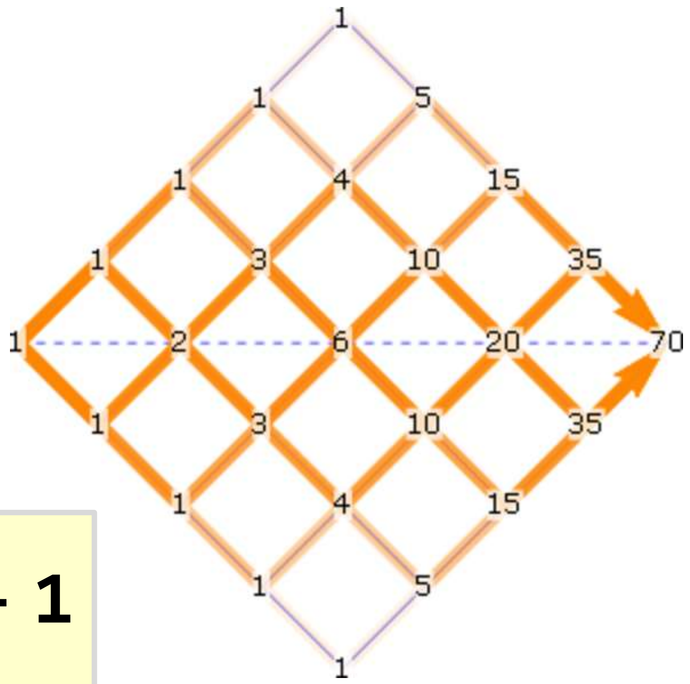
Lattice path: The insight!

- Can we find such a relationship at other points?



🟥 + 🟦 $\Rightarrow$ 2 + 1

🟦 + 🟩 $\Rightarrow$ 2 + 1

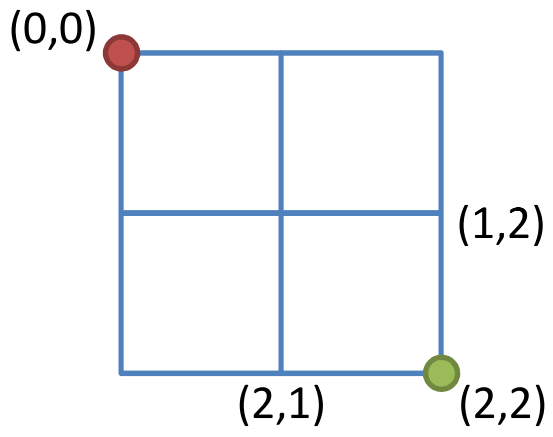


Lattice path: The sub-problem!

- Let's model the problem:
  - Define a coordinate system over the lattice
  - **`(0,0)`** at the starting point
  - **`(i,j)`** at the lattic point at i-th row, j-th column
  - Let function **`f(i,j)`** be the number of distinct paths from **`(0,0)`** to **`(i,j)`**

```
f( i , j ) = f( i-1 ,  j  )
           + f(  i  , j-1 )
```

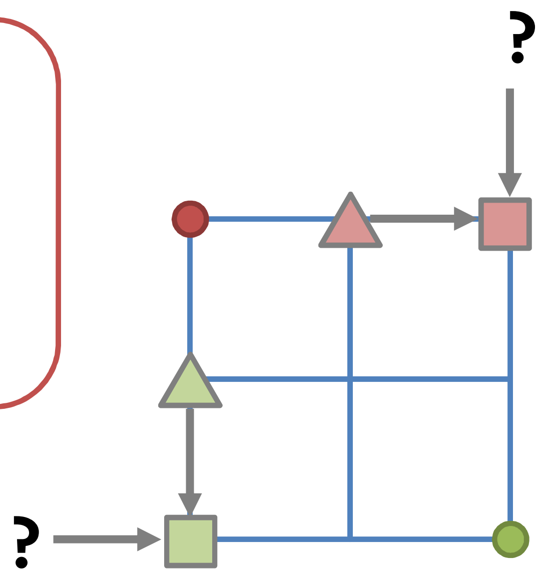
<u>**A recurrence relationship!**</u>



Lattice path: The boundary!

- How about `f(i,0)` and `f(0,j)`?
  - They are at the boundaries
  - The previous recurrence relationship cannot be used.

```
f( i , 0 ) = f( i-1 , 0   )
                 &
f( 0 , j ) = f( 0   , j-1 )
```



Lattice path: The base case!

- How about `f(0,0)`?
  - It means…

    "**`the number of paths from (0,0) to (0,0)`**"

  - Of course, it is **one**!

Lattice path: The solution!

$$f(i,j) = \begin{cases} 1 & , i = j = 0 \\ f(i-1,0) & , j = 0 \\ f(0,j-1) & , i = 0 \\ f(i-1,j) + f(i,j-1) & , \text{otherwise} \end{cases}$$

Lattice path: The program!

- Around 10 lines of code only!
  - Do you believe that it works?
  - Let's have a demo: start with **`path( 2 , 2 )`**.

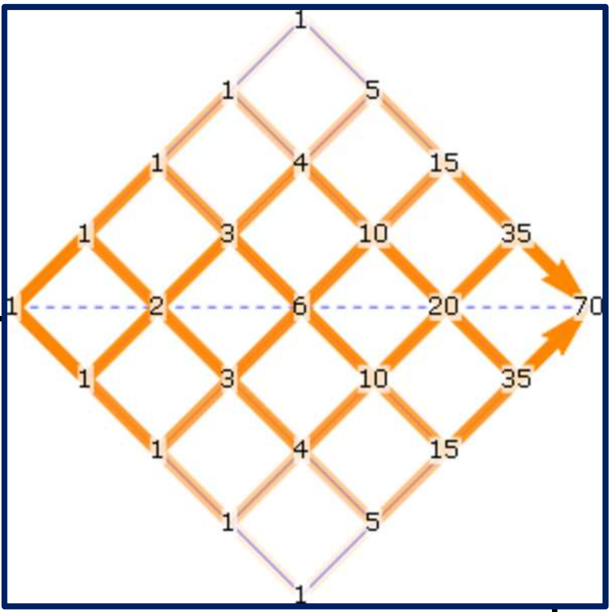
```
int path( int i , int j )
{
    if ( i == 0 && j == 0 )
        return 1 ;
    else
    if ( i == 0 && j  > 0 )
        return path( 0 , j - 1 );
    else
    if ( i  > 0 && j == 0 )
        return path( i – 1 , 0 );
    else
        return path( i – 1 , j ) + path( i , j - 1 );
}
```

Can we speed up?
Any idea?

Lattice path: The program Ver. 2!

- Speed up!
  - Return 1 at first row/column

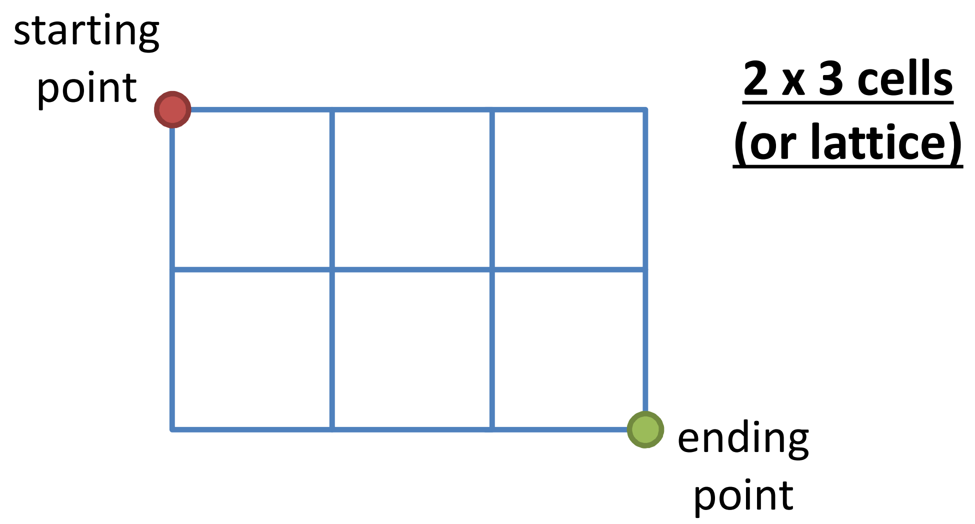
```
int path( int i , int j )
{
    if ( i == 0 || j == 0 )
        return 1 ;
    else
        return path( i - 1 , j ) + path( i , j - 1 );
}
```



Lattice path: Variation #1

#1: Rectangular lattice

- Any difference?
- NO!



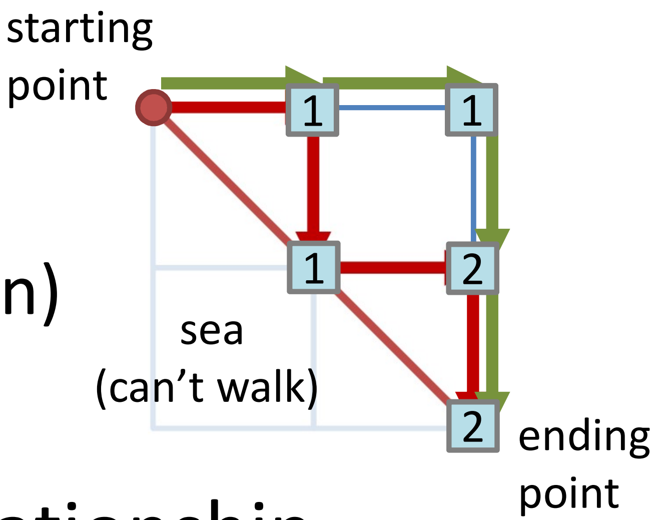
Lattice path: Variation #2

#2 : Cannot go across (but may touch) the diagonal!

- Any differences from the original problem?
- Yes!

E.g., <u>the number of paths is 2</u>

(instead of 6 in the original version)



- Hint: change the recurrence relationship
  - When **`i == j`**, what should you do?

You will try this in lab 8 today ☺

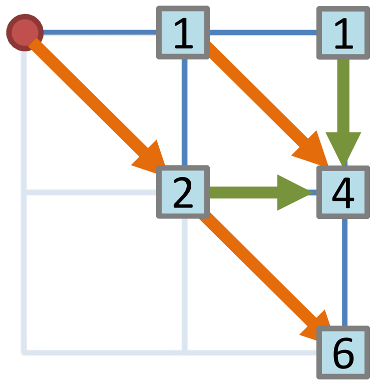
Lattice path: Variation #3

#3: Variation #2 + **new movement**!

- The new movement is a **DIAGONAL** move!
- What change(s) we need?

E.g., the number of paths is 6

(instead of 2 from Variation #2)



- Hint: in the "**`otherwise`**" case of the recurrence relationship, add one more item!

Lattice path: Speedup?

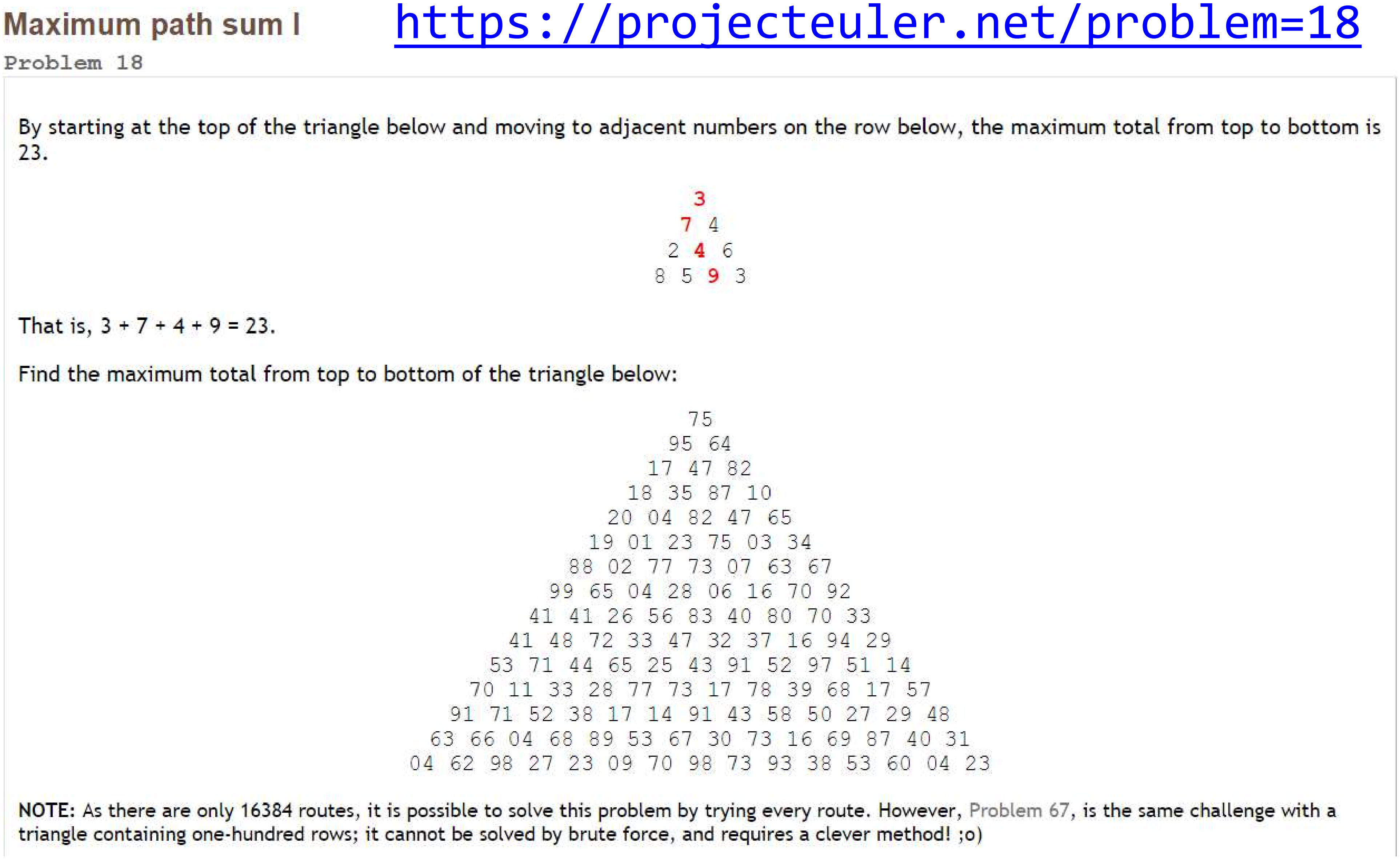
- Class discussion:
  - Speedup:
    - Ask yourself: How can we speed up this program?
    - For example, say by memorization?
  - How about 3D Lattice?

Outline

- Next, we will show you how to solve a problem by identifying its **sub-problems**.
- Pay attention to three basic but important things:
  - **Model the problem with sub-problems**
  - **Design function prototype** for **data passing and return**
  - When to **Terminate**?
- Three Examples:
  - Path Search (1): Lattice Path
  - **Path Search (2): Number Pyramid**
  - The Classic: Tower of Hanoi

Question from Project Euler

https://projecteuler.net/problem=18



Maximum path sum I

Problem 18

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

```
   3
  7 4
 2 4 6
8 5 9 3
```

That is, 3 + 7 + 4 + 9 = 23.

Find the maximum total from top to bottom of the triangle below:

```
                            75
                          95 64
                        17 47 82
                      18 35 87 10
                    20 04 82 47 65
                  19 01 23 75 03 34
                88 02 77 73 07 63 67
              99 65 04 28 06 16 70 92
            41 41 26 56 83 40 80 70 33
          41 48 72 33 47 32 37 16 94 29
        53 71 44 65 25 43 91 52 97 51 14
      70 11 33 28 77 73 17 78 39 68 17 57
    91 71 52 38 17 14 91 43 58 50 27 29 48
  63 66 04 68 89 53 67 30 73 16 69 87 40 31
04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

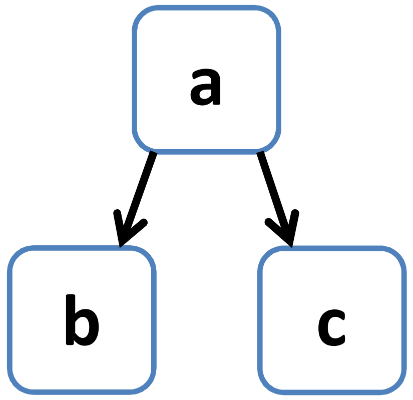
NOTE: As there are only 16384 routes, it is possible to solve this problem by trying every route. However, Problem 67, is the same challenge with a triangle containing one-hundred rows; it cannot be solved by brute force, and requires a clever method! ;o)

Number Pyramid: Insight

- Again, we can first locate its sub-problems and tackle it using recursion!

- Let's look at a simple case: 2 layers only

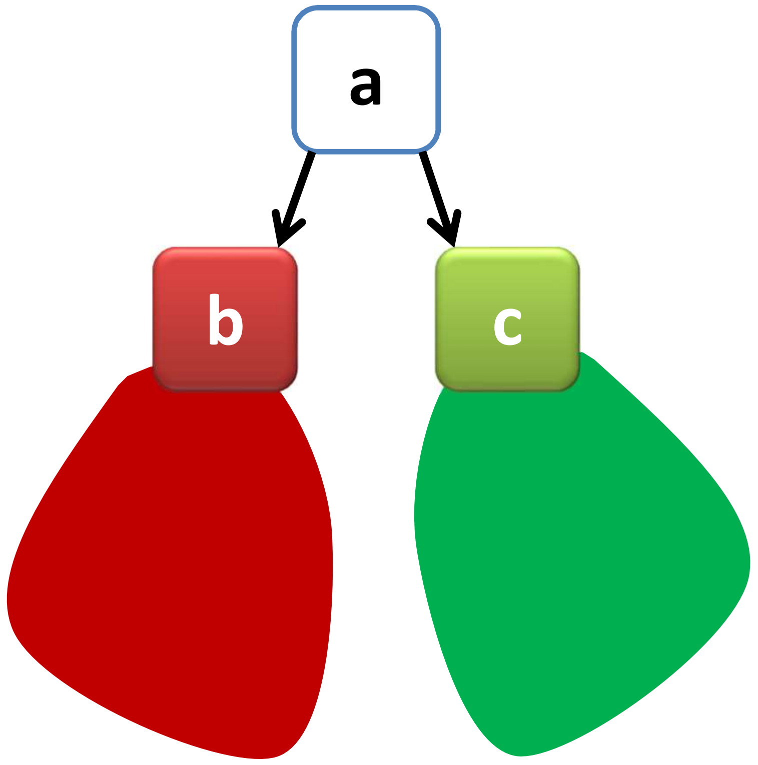
```
Answer = a + max( b , c )
```



Note: Calling a recursion function is like asking a question

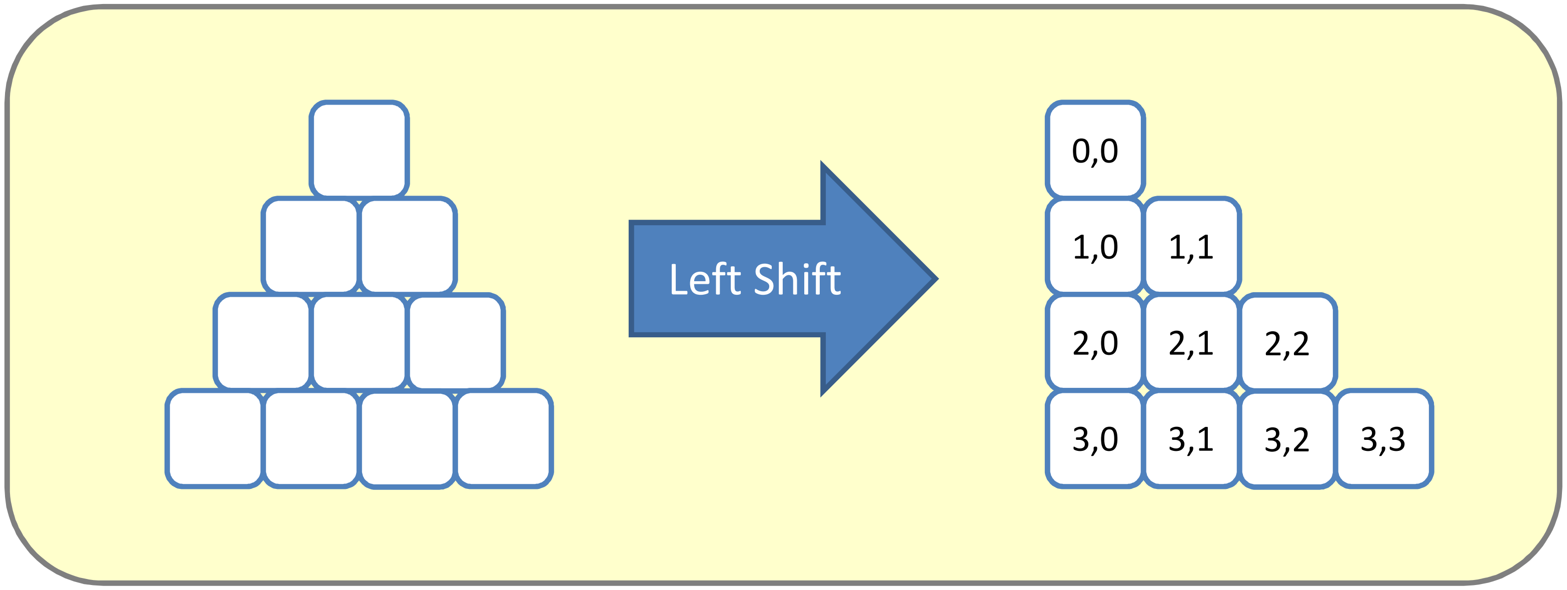
Number Pyramid: Insight

- What if b and c represent the max path sum from the left and right branches of a, respectively?



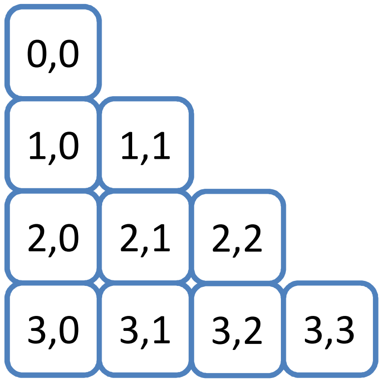
Number Pyramid: Coordinates?

- We can re-order the pyramid to define a simple-to-implement coordinates system, allowing us to store data using a 2D array.



Number Pyramid: Recurrence

- Let **`a(i,j)`** be the value at location **`(i,j)`**, which is at i-th row, j-th column.
- Let **`f(i,j)`** be the maximum path sum from **`(0,0)`** from **`(i,j)`**.
- So, the general recurrence formula is:



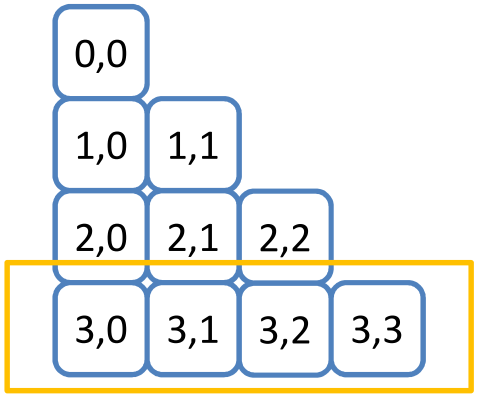
```
f(i,j) = a(i,j)
       + max( f(i+1,j) , f(i+1,j+1) )
```

Number Pyramid: Termination cases

- Let **n** be the number of rows in the pyramid.
- Then, what is **`f(n-1,0)`**, …, **`f(n-1,n-1)`**?
  - They are located on the bottom-most row
- Therefore:

  `f(n-1,i) = a(n-1,i)`

  where **`0 <= i <= n-1`**



Number Pyramid: Solution

$$
f(i,j) = \begin{cases} a(n-1,j), & \text{when } i = n-1 \text{ (bottom)} \\ a(i,j) + \max(f(i+1,j),\ f(i+1,j+1)), & \text{otherwise} \end{cases}
$$

Start the recursion with f(0 , 0)

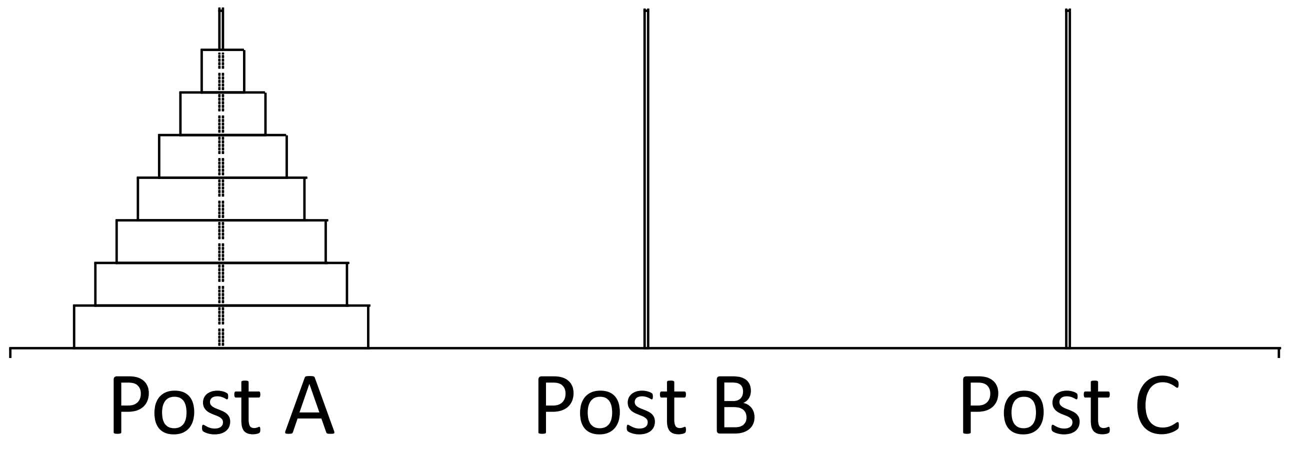
Outline

- Next, we will show you how to solve a problem by identifying its **sub-problems**.

- Pay attention to three basic but important things:
  - **Model the problem with sub-problems**
  - **Design function prototype** for **data passing and return**
  - When to **Terminate**?

- Three Examples:
  - Path Search (1): Lattice Path
  - Path Search (2): Number Pyramid
  - **The Classic: Tower of Hanoi**

What is the Tower of Hanoi



Goal: Move all the disks from Post A to Post C

Rules:

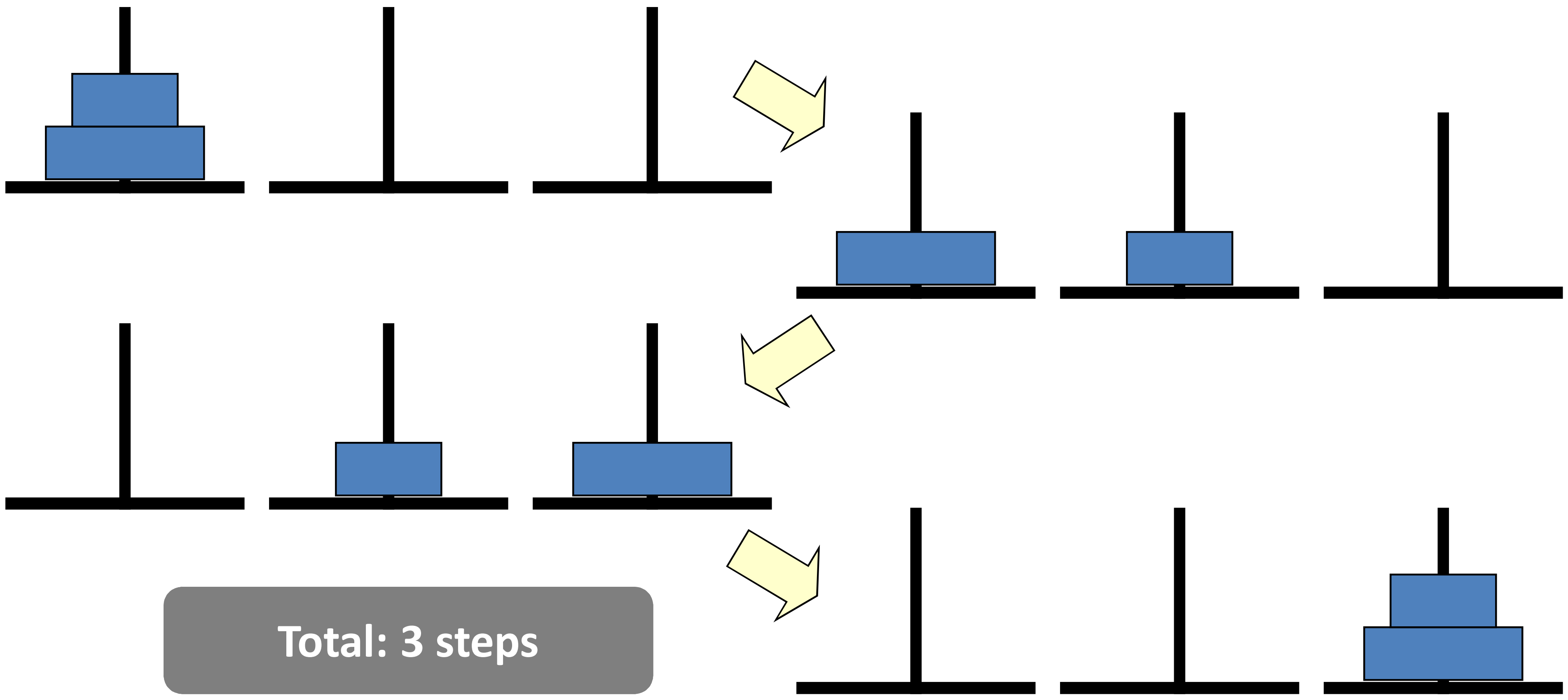
(1) move one disk at a time;
(2) place a disk on the top of another post; and
(3) a bigger disk cannot be placed above a smaller disk.

Program Output:

A series of steps to guide the user how to move the disks

Tower of Hanoi – 2 disks only

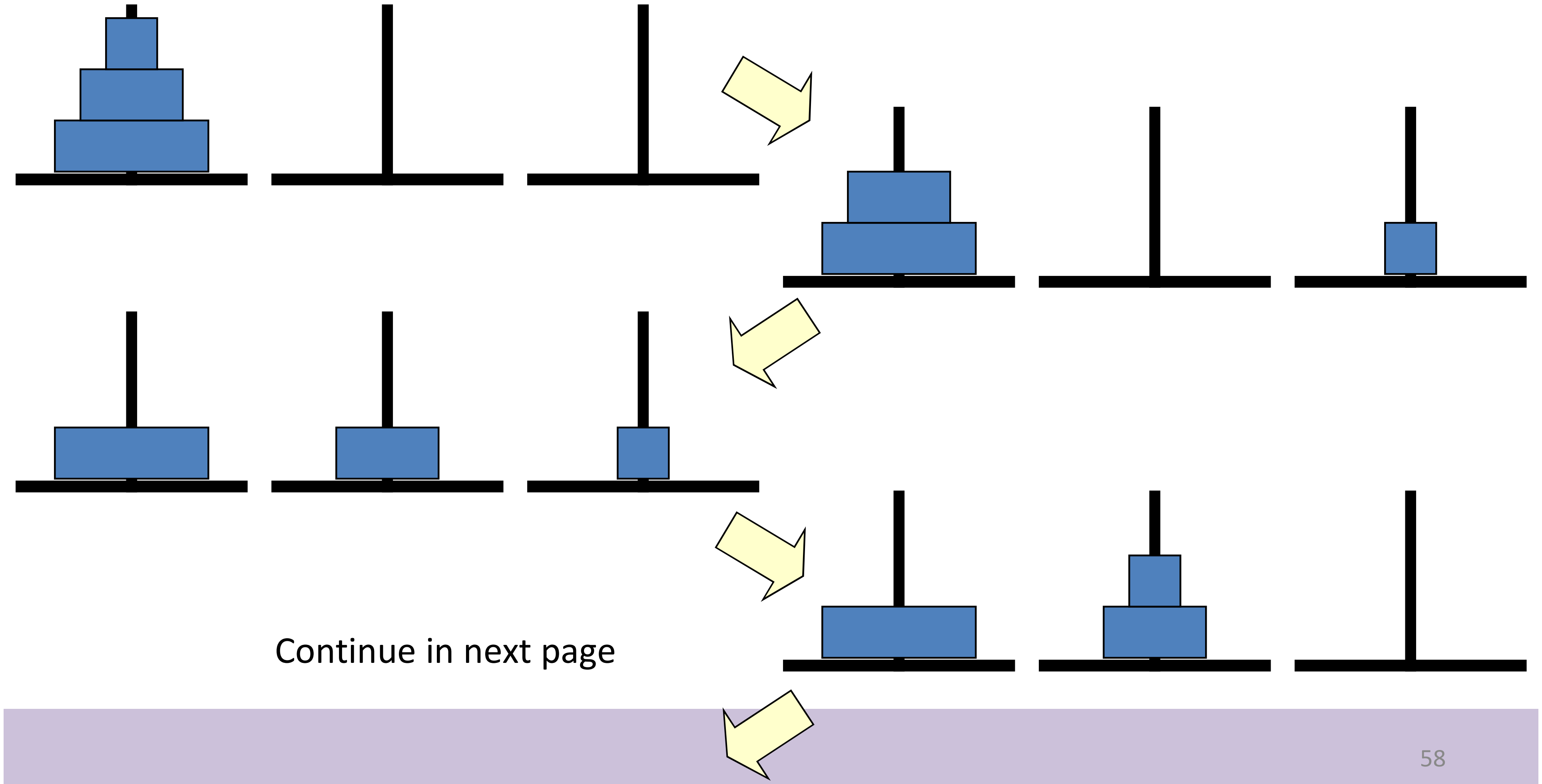
It is easy if we only have 2 disks.



Total: 3 steps

Tower of Hanoi – 3 disks (Part 1 of 2)

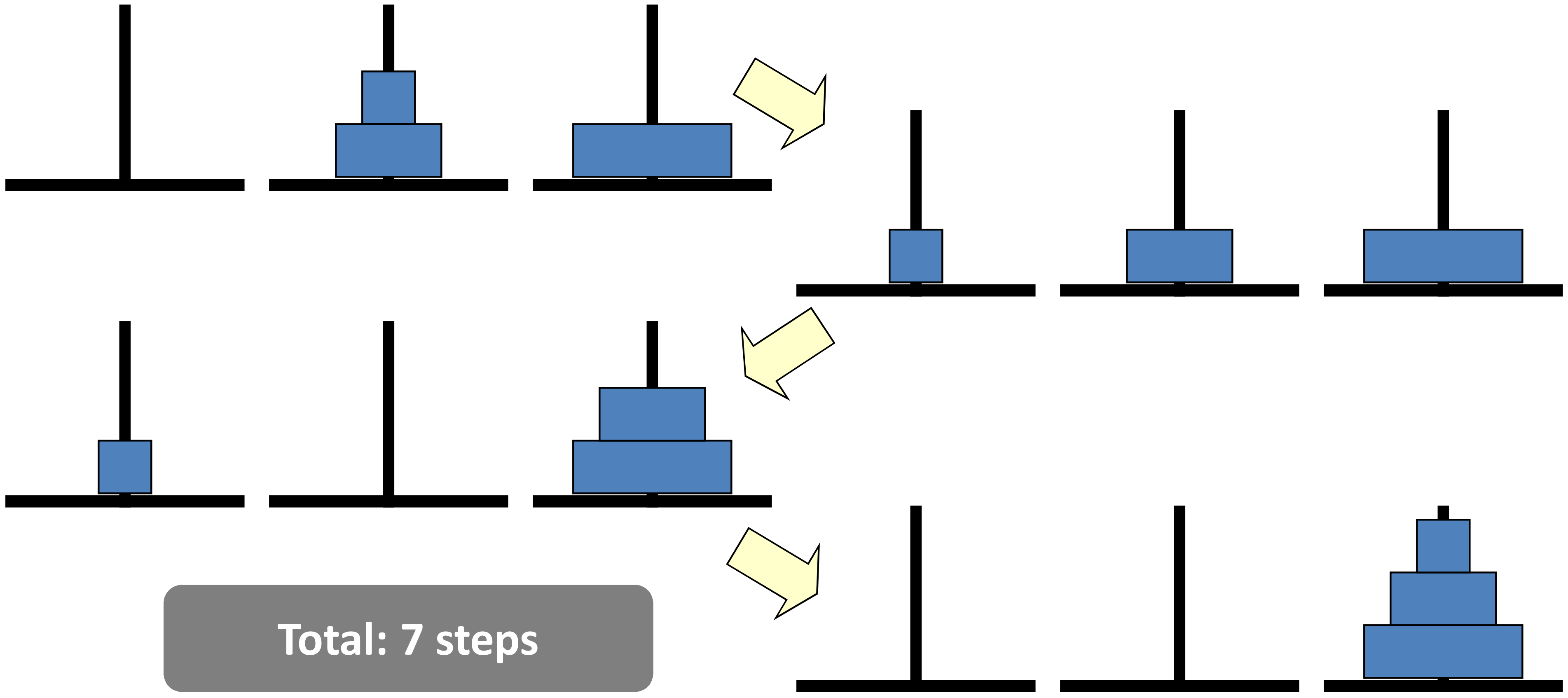
It is not difficult if we only have 3 disks.



Continue in next page

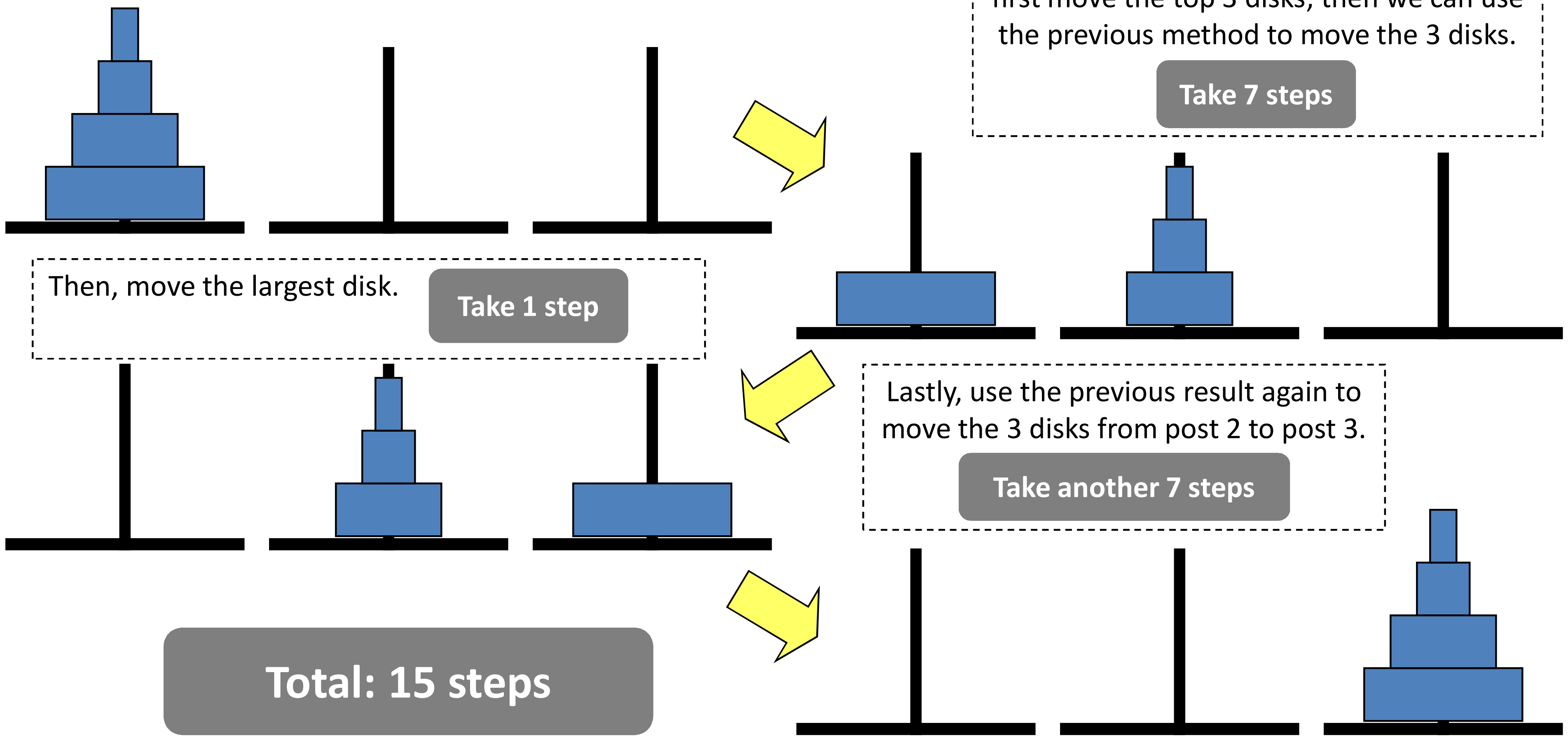
Tower of Hanoi – 3 disks (Part 2 of 2)

It is not difficult if we only have 3 disks.



Total: 7 steps

Tower of Hanoi – 4 disks = 15 steps



Before moving the largest disk, we have to first move the top 3 disks; then we can use the previous method to move the 3 disks.

Take 7 steps

Then, move the largest disk.

Take 1 step

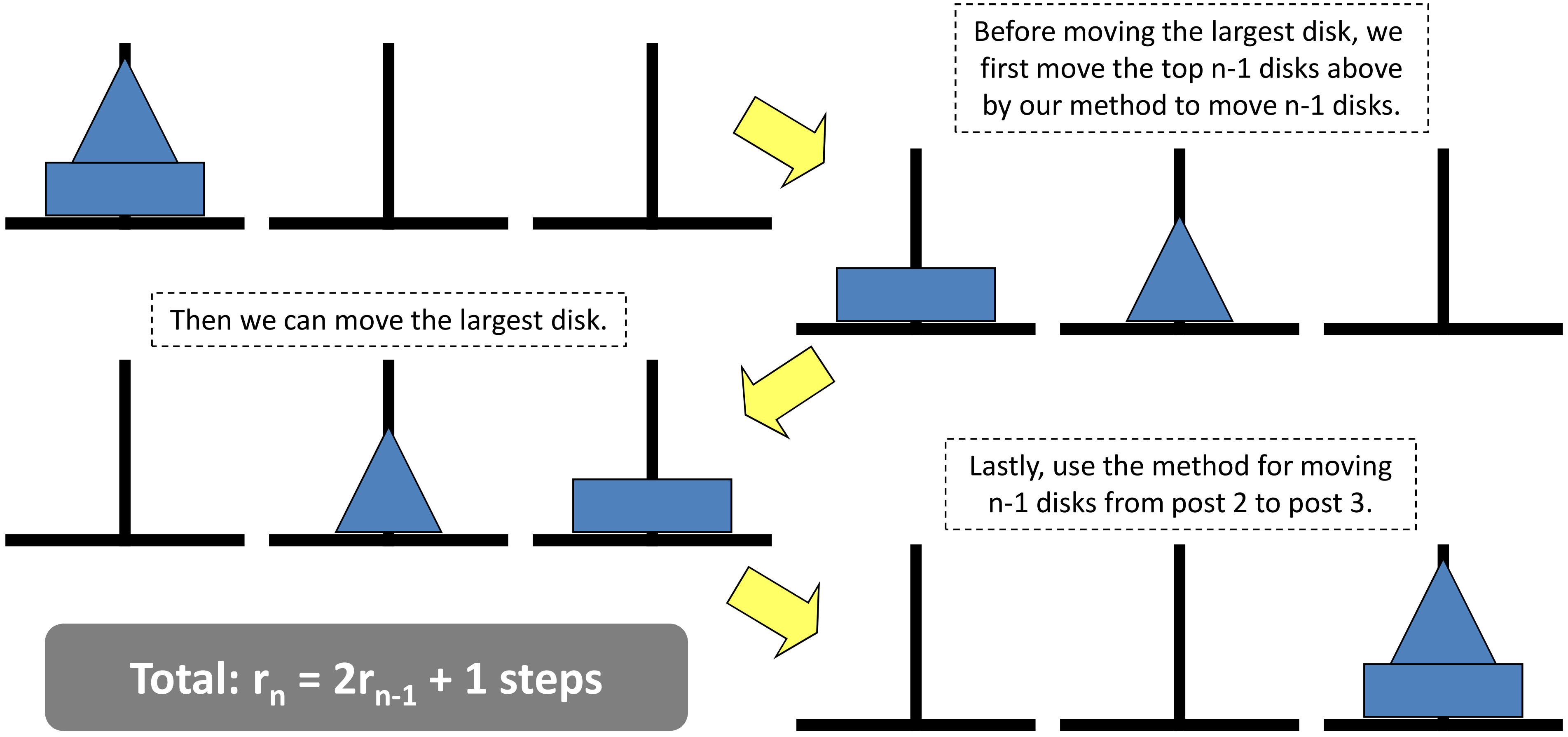
Lastly, use the previous result again to move the 3 disks from post 2 to post 3.

Take another 7 steps

Total: 15 steps

Tower of Hanoi: General Case

This recursion is true for any n. Let r_n be the minimum # of steps to move n disks.



Before moving the largest disk, we first move the top n-1 disks above by our method to move n-1 disks.

Then we can move the largest disk.

Lastly, use the method for moving n-1 disks from post 2 to post 3.

Total: $r_n = 2r_{n-1} + 1$ steps

Generally speaking...

$a_1 = 1$, $a_k = 2a_{k-1} + 1$

$a_2 = 2a_1 + 1 = 3$

$a_3 = 2a_2 + 1 = 2(\ 2a_1 + 1\ ) + 1 = 4a_1 + 3 = 7$

$a_4 = 2a_3 + 1 = 2(\ 4a_1 + 3\ ) + 1 = 8a_1 + 7 = 15$

$a_5 = 2a_4 + 1 = 2(\ 8a_1 + 7\ ) + 1 = 16a_1 + 15 = 31$

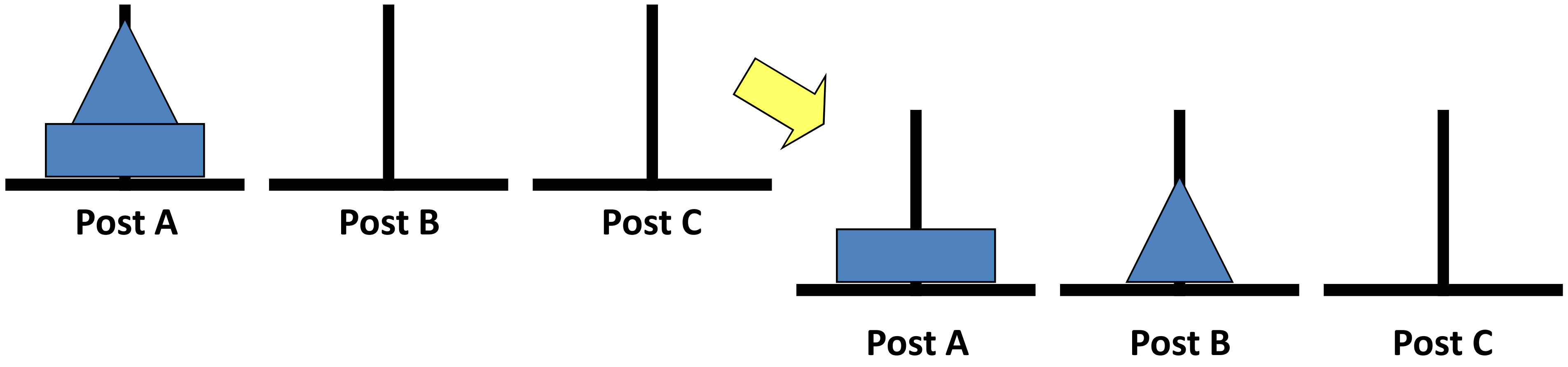
$a_6 = 2a_5 + 1 = 2(\ 16a_1 + 15\ ) + 1 = 32a_1 + 31 = 63$

Guess the pattern is $a_k = 2^k - 1$.

You can verify by induction.

Tower of Hanoi: General Case

- So, what are the sub-problems?



<u>Solving problem of size n: P(n, A, C), move n disk from Post 1 to Post 3</u>

<u>Sub-problem #1</u>: P(n-1, A, B), move n-1 disks from Post A to Post B.

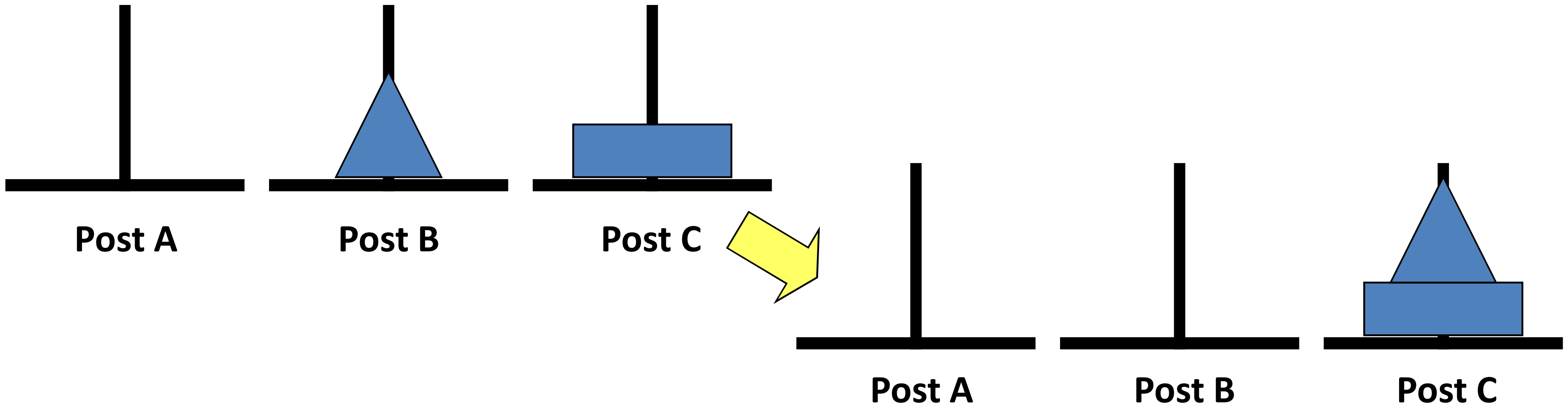
Tower of Hanoi: General Case

Solving problem of size n: P(n, A, C): move n disk from Post A to Post C

Sub-problem #1: P(n-1, A, B), move top n-1 disks from Post A to Post B

Easy step: move the largest disk from Post A to Post C

Sub-problem #2: P(n-1, B, C), move top n-1 disks from Post B to Post C



Final Solution!

```
void tower( int n , char origin , char dest , char buffer )
{
   if ( n == 0 )
      return ;
   tower( n - 1 , origin , buffer , dest   );
   printf( "Move Disk %d from %c to %c\n" , n , origin , dest );
   tower( n - 1 , buffer , dest   , origin );
}

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

Think and trace carefully the **execution order** of each line of code in the recursion!!!

Example: n=3

```c
void tower( int n , char origin , char dest , char buffer )
{
   if ( n == 0 )
      return ;
   tower( n - 1 , origin , buffer , dest   );
   printf( "Move Disk %d from %c to %c\n" , n , origin , dest );
   tower( n - 1 , buffer , dest   , origin );
}

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

output:

```
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
```

3rd level ...

2nd level tower(2 , 'A' , 'B' , 'C');

3rd level ...

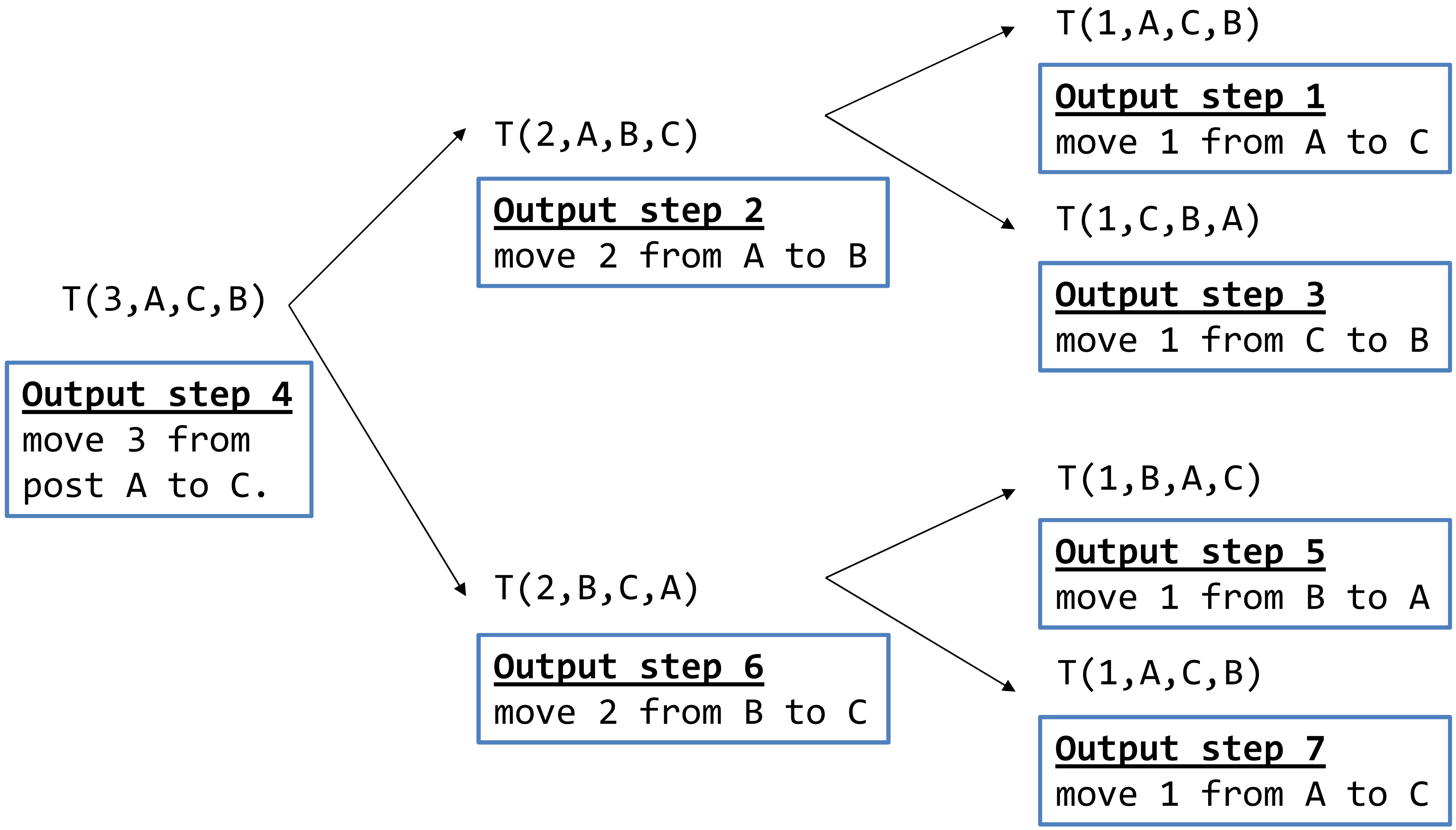
1st level tower(3 , 'A' , 'C' , 'B');

3rd level ...

2nd level tower(2 , 'B' , 'C' , 'A');

3rd level ...

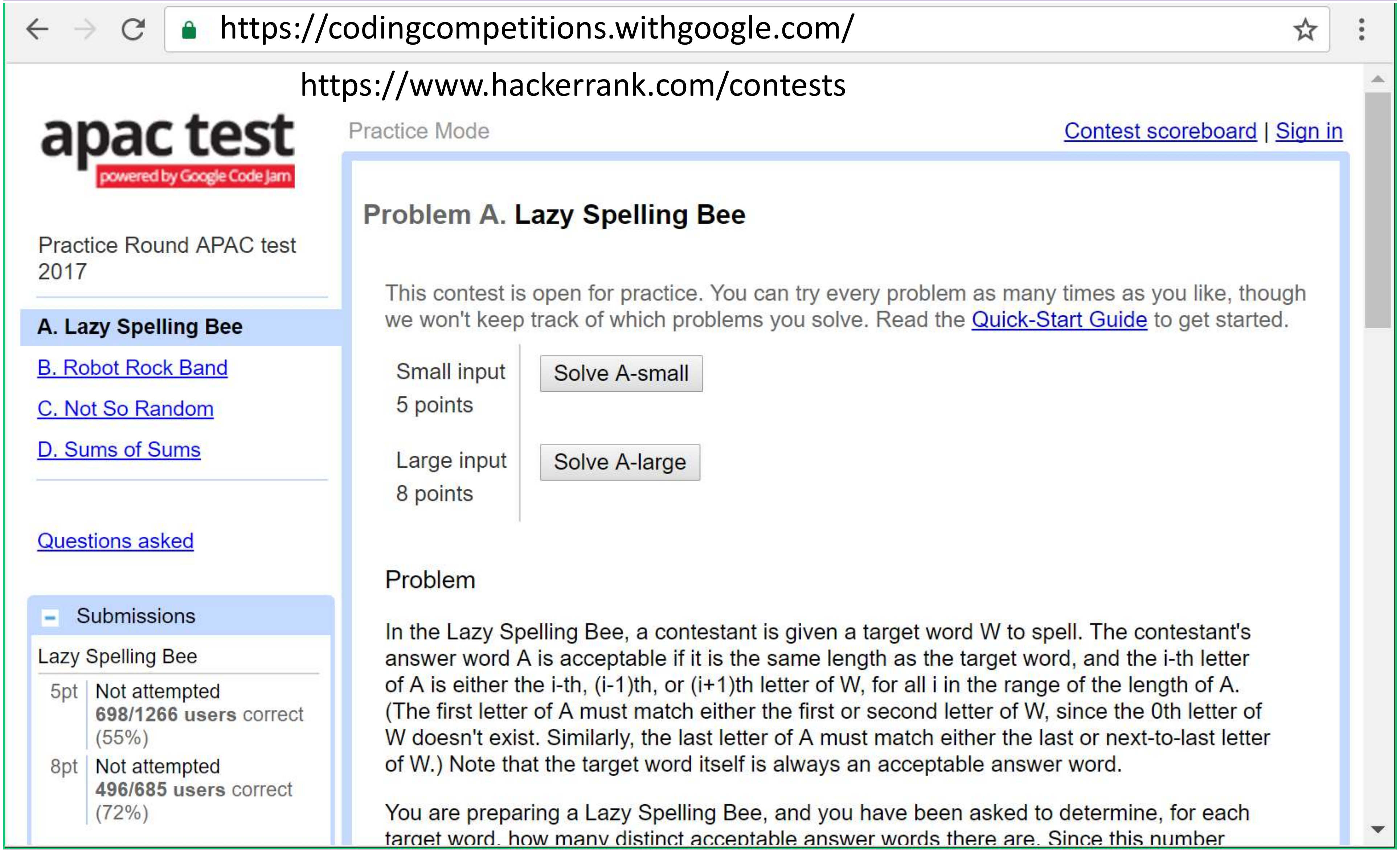
Example: n=3


T(3,A,C,B)
Output step 4
move 3 from post A to C.
T(2,A,B,C)
Output step 2
move 2 from A to B
T(2,B,C,A)
Output step 6
move 2 from B to C
T(1,A,C,B)
Output step 1
move 1 from A to C
T(1,C,B,A)
Output step 3
move 1 from C to B
T(1,B,A,C)
Output step 5
move 1 from B to A
T(1,A,C,B)
Output step 7
move 1 from A to C

Summary

- Recursion is a function that may call itself
  - Calling a recursion function is like asking a question
- Key idea: Solve a problem by solving smaller sub-problems of itself
- Important but basic things (decide before coding):
  - Model the problem with sub-problems
  - Function prototype: data passing and return
  - Termination Condition
- Speed up technique:
  - Memorization to avoid redundant re-computation
- Any limitation? If we call "`factorial(100000)`", then…

Lastly… Want more problems?



https://codingcompetitions.withgoogle.com/

https://www.hackerrank.com/contests

Practice Mode

Contest scoreboard | Sign in

Practice Round APAC test 2017

A. Lazy Spelling Bee

B. Robot Rock Band

C. Not So Random

D. Sums of Sums

Questions asked

Submissions

Lazy Spelling Bee

5pt Not attempted **698/1266 users** correct (55%)

8pt Not attempted **496/685 users** correct (72%)

Problem A. **Lazy Spelling Bee**

This contest is open for practice. You can try every problem as many times as you like, though we won't keep track of which problems you solve. Read the Quick-Start Guide to get started.

Small input 5 points	Solve A-small
Large input 8 points	Solve A-large

Problem

In the Lazy Spelling Bee, a contestant is given a target word W to spell. The contestant's answer word A is acceptable if it is the same length as the target word, and the i-th letter of A is either the i-th, (i-1)th, or (i+1)th letter of W, for all i in the range of the length of A. (The first letter of A must match either the first or second letter of W, since the 0th letter of W doesn't exist. Similarly, the last letter of A must match either the last or next-to-last letter of W.) Note that the target word itself is always an acceptable answer word.

You are preparing a Lazy Spelling Bee, and you have been asked to determine, for each target word, how many distinct acceptable answer words there are. Since this number