

# Lecture 5: Recursion

# Course outline

- Part 1 Introduction to Computing and Programming (first 2 weeks):
  - Problem solving: Problem statement, algorithm design, programming, testing, debugging
  - Scalar data types: integers, floating point, Boolean, others (letters, colours)
  - Arithmetic, relational, and logical operators, and expressions
  - Data representation of integers, floating point, Boolean
  - Composite data structures: string, tuple, list, dictionary, array
  - Sample operations on string, tuple, list, dictionary, array
  - Algorithms (written in pseudo code) vs. programs
  - Variables and constants (literals): association of names with data objects
  - A language to write pseudo code
  - Programming languages: compiled vs. interpreted programming languages
  - Python as a programming language
  - Computer organization: processor, volatile and non-volatile memory, I/O

# Course outline (may change a bit)

- Part 2 Algorithm design and Programming in Python (balance 11 weeks):
  - Arithmetic/Logical/Boolean expressions and their evaluations in Python
  - Input/output statements (pseudo code, and in Python)
  - Assignment statement (pseudo code, and in Python)
  - Conditional statements, with sample applications
  - Iterative statements, with sample applications
  - Function sub-programs, arguments and scope of variables
  - Recursion
  - Modules
  - Specific data structures in Python (string, tuple, list, dictionary, array), with sample applications
  - Searching and sorting through arrays or lists
  - Handling exceptions
  - Classes, and object-oriented programming
  - (Time permitting) numerical methods: Newton Raphson, integration, vectors/matrices operations, continuous-time and discrete-event simulation

# Recursion

- Recursion is a powerful concept, and a tool, for developing algorithms or programs
- Often certain definitions are given recursively:

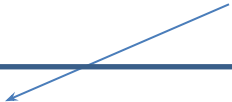
# Recursion

- Example 1: n factorial

For  $n > 0$ ,  $n! = n * (n-1) * (n-2) * \dots * 2 * 1$

OR, equivalently

Called the “base case”



$$\begin{aligned} n! &= 1, \text{ if } n = 1, \\ &= n * (n-1)!, \text{ otherwise} \end{aligned}$$

# Recursion

- Example 1: n factorial

For  $n > 0$ ,  $n! = n * (n-1) * (n-2) * \dots * 2 * 1$

OR, equivalently

Called the “base case”

$$\begin{aligned} n! &= 1, \text{ if } n = 1, \\ &= n * (n-1)!, \text{ otherwise} \end{aligned}$$

- Example calculation of n!

$$1! = 1$$

$$2! = 2$$

$$3! = 6$$

$$4! = 24$$

Etc.

# Recursion

- Example 2: GCD(a, b), where  $a > b > 1$

GCD(a, b) = b, if a mod b = 0,  
= GCD(b, a mod b), otherwise

Called the “base case”



# Recursion

- Example 2: GCD(a, b), where  $a > b > 1$

$$\begin{aligned}\text{GCD}(a, b) &= b, \text{ if } a \bmod b = 0, \\ &= \text{GCD}(b, a \bmod b), \text{ otherwise}\end{aligned}$$

Called the “base case”



- Example calculations:

$$\text{GCD}(15, 6) = \text{GCD}(6, 3) = 3$$

$$\text{GCD}(42, 15) = \text{GCD}(15, 12) = \text{GCD}(12, 3) = 3$$

etc.



# Recursion

- Example 3: Fibonacci numbers (work by an Indian mathematician in 450 BC–200 BC)

$$\begin{aligned} F(n) &= 1, \text{ if } n = 0 \text{ or } n = 1, \\ &= F(n-1) + F(n-2), \text{ otherwise (viz. } n \geq 2) \end{aligned}$$

Called the “base case”

[https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)

Alternate definition  $F(0) = 0$ ,  $F(1) = 1$ ,  $F(2) = 1$ , etc.

# Recursion

- Example 3: Fibonacci numbers

$F(n) = 1$ , if  $n = 0$  or  $n = 1$ ,  
 $= F(n-1) + F(n-2)$ , otherwise (viz.  $n \geq 2$ )

Called the “base case”

- Example calculations:


<b>n</b>	<b>F(n)</b>
0	1
1	1
2	2
3	3
4	5
Etc.	Etc.

[https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)

Alternate definition  $F(0) = 0$ ,  $F(1) = 1$ ,  $F(2) = 1$ , etc.


# Recursion

- Example 3: **Fibonacci numbers**:  
(early work by an Indian mathematician in 450 BC–200 BC)

$F(n) = 1$ , if  $n = 0$  or  $1$ ,  **The “base case”**  
 $= F(n-1) + F(n-2)$ ,  $n \geq 2$

resulting in Fibonacci sequence 

n	F(n)	F(n)/F(n-1)
0	1	
1	1	1.00000
2	2	2.00000
3	3	1.50000
4	5	1.66667
5	8	1.60000
6	13	1.62500
7	21	1.61538
8	34	1.61905
9	55	1.61765
10	89	1.61818
11	144	1.61798
12	233	1.61806
13	377	1.61803
14	610	1.61804
15	987	1.61803
16	1597	1.61803
Etc.	Etc.	etc.

Golden ratio:  
solution to equation  $x^2 = x + 1$  

Also:  
 $F(n) = 1.61803 * F(n-1)$   
when  $n$  is large

# Recursion

- Example 4: Is a given string **S** a **Palindrome**?
- A string **S** is a Palindrome if S reads the same way forward and backwards
- Example Palindromes:
  - '\$'
  - 'anna'
  - 'kayak'

# Recursion

- Example 4: Is a given string  $S$  a **Palindrome**?
- A string  $s$  is a Palindrome if  $S$  reads the same way forward and backwards

where  $c$  is a character

$S$  is a Palindrome, if  $s = ""$ , or  
 $s = 'c'$ , or  
 $s[0] = s[-1]$ , and  $s[1, -1]$  is a Palindrome  
is NOT a Palindrome, otherwise

the "base cases"  
 $\text{len}(s) \leq 1$

# Recursion

- Example 4: Is a given string  $S$  a **Palindrome**?
- A string  $s$  is a Palindrome if  $S$  reads the same way forward and backwards

$S$  is a Palindrome, if  $s = ""$ , or  
 $s = 'c'$ , or  
 $s[0] = s[-1]$ , and  $s[1, -1]$  is a Palindrome  
is NOT a Palindrome, otherwise

where  $c$  is a character

the "base case"  
 $\text{len}(s) \leq 1$

- Example Palindromes:  
"  
'\$'  
'anna'  
'kayak'  
Etc.
- Example NON-Palindromes:  
'IN'  
'e-Rupee'  
Etc.

The following MAY also be considered to be a palindrome:  
'Was it a cat I saw'  
Under what assumptions, though?

# Recursion

- Recursion – based **function program** to compute  $n!$

By definition:

$$\begin{aligned} n! &= 1, \text{ if } n = 1, \\ &= n * (n-1)!, \text{ otherwise} \end{aligned}$$

```
# compute n! using recursion
# Assumes that n > 0; returns n!
def fact(n):
    if n == 1:
        return(1)
    else:
        return(n*fact(n - 1))

for k in range(1, 6):
    print(fact(k))
```

Exercise to be done at home:

Rewrite the program without using recursion.

# Recursion

- Recursion – based **function program**

	Local variables/objects	Global variables/objects
Main	f(.)	
fact(.)	n	fact(.)

By definition:

$$\begin{aligned} n! &= 1, \text{ if } n = 1, \\ &= n * (n-1)!, \text{ otherwise} \end{aligned}$$

```
# compute n! using recursion
# Assumes that n > 0; returns n!
def fact(n):
    if n == 1:
        return 1
    else:
        return (n*fact(n - 1))

for k in range(1, 6):
    print(fact(k))
```



# Recursion

- Recursion – based **function program** to compute  $n!$

By definition:

$n! = 1$ , if  $n = 1$ ,  
=  $n * (n-1)!$ , otherwise

```
# compute n! using recursion
# Assumes that n > 0; returns n!
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n - 1)

for k in range(1, 6):
    print(fact(k))
```

Visualize its execution using:  
<https://tinyurl.com/47v4xyp6>

How is the “base case” handled?  
Could it be that this program will  
never terminate?

What ensures that the program will  
terminate?

# Recursion

- Example 4: Is a given string S a **Palindrome**?
- A string s is a Palindrome if s reads the same way forward and backwards

s is a Palindrome, if s = "", or  
s = 'c', or  
s[0] = s[-1], and s[1, -1] is a Palindrome  
is NOT a Palindrome, otherwise

- Algorithmically:  
if (len[s] <= 1) then s is palindrome  
else if ((s[0] = s[-1]) and (string s[1: -1] is a palindrome)) then s is palindrome  
else s is not a palindrome

The "base case"



# Recursion

- Recursion – based function sub-program to test if given string *s* is a **Palindrome**
- Algorithmically:  
if (**len[s] <= 1**) then *s* is palindrome  
else if (**(s[0] = s[-1]) and (string s[1: -1] is a palindrome)**) then *s* is palindrome  
else *s* is not a palindrome

```
# determine whether a given string is a palindrome
```

```
def isPal(s):
```

```
    """Assumes s is a str only of lower-case English letters.  
    It returns True if s is a palindrome, False otherwise."""
```

```
    if len(s) <= 1:
```

```
        return(True)
```

```
    else:
```

```
        return((s[0] == s[-1]) and isPal(s[1:-1]))
```

```
print(isPal("wasitacatisaw"))
```

<https://tinyurl.com/3n74b3yu>

# Recursion

- Recursion –based algorithm/program to test whether a given string *s* is a **Palindrome**
  - This time it allows strings to be form: 'Was it a cat I saw', or 'dog God'

```
# Palindrome 2
# Determine if S is a palindrome
def isPalindrome(s):
    """Assumes s is a str
    Returns True if s is a palindrome; False otherwise.
    Punctuation marks, blanks, and capitalization are ignored."""
    #
    def toChars(s):
        s = s.lower()
        letters = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                letters = letters + c
        return letters
    #
    def isPal(s):
        print(' isPal called with', s)
        if len(s) <= 1:
            print(' About to return True from base case')
            return True
        else:
            answer = s[0] == s[-1] and isPal(s[1:-1])
            print(' About to return', answer, 'for', s)
            return answer
    return(isPal(toChars(s)))
#
Print(isPalindrome('dog..God'))
```

Built-in "method" for strings, as in  
txt = "Hello my FRIENDS"  
x = txt.lower()  
print(x)

hello my friends

Let us visualize the execution flow using

Intro to Programming Monsoon 2023 <http://pythontutor.com/visualize.html>

# Recursion

- Example: **Fibonacci numbers**:

$$\begin{aligned} F(n) &= 1, \text{ if } n = 0 \text{ or } 1, \\ &= F(n-1) + F(n-2), \text{ } n \geq 2 \end{aligned}$$

resulting in Fibonacci sequence

n	F(n)	F(n)/F(n-1)
0	1	
1	1	1.00000
2	2	2.00000
3	3	1.50000
4	5	1.66667
5	8	1.60000
6	13	1.62500
7	21	1.61538
8	34	1.61905
9	55	1.61765
10	89	1.61818
11	144	1.61798
12	233	1.61806
13	377	1.61803
14	610	1.61804
15	987	1.61803
16	1597	1.61803
Etc.	Etc.	Etc.

[https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)

In some cases  $F(0) = 0$ ,  $F(1) = 1$ ,  $F(2) = 1$ , etc.

Golden ratio:  
solution to equation  $x^2 = x + 1$

Also:  
 $F(n) = 1.61803 * F(n-1)$   
when  $n$  is large

# Recursion

- Recursion – based function sub-program to compute **Fibonacci numbers**
- By definition:

$$F(n) = 1, \text{ if } n = 0 \text{ or } 1,$$
$$= F(n-1) + F(n-2), n \geq 2$$

```
# Fibonacci-1
# compute F(k), for k in [0, 1, 2, 3, 4]
def fib(x):
    """Assumes int x >= 0
       Returns F(x)"""
    if x == 0 or x == 1:
        return(1)
    else:
        return(fib(x-1) + fib(x-2))

for k in range(5):
    print('fib of', k, '=', fib(k))
```

<http://tinyurl.com/yc78yey7>

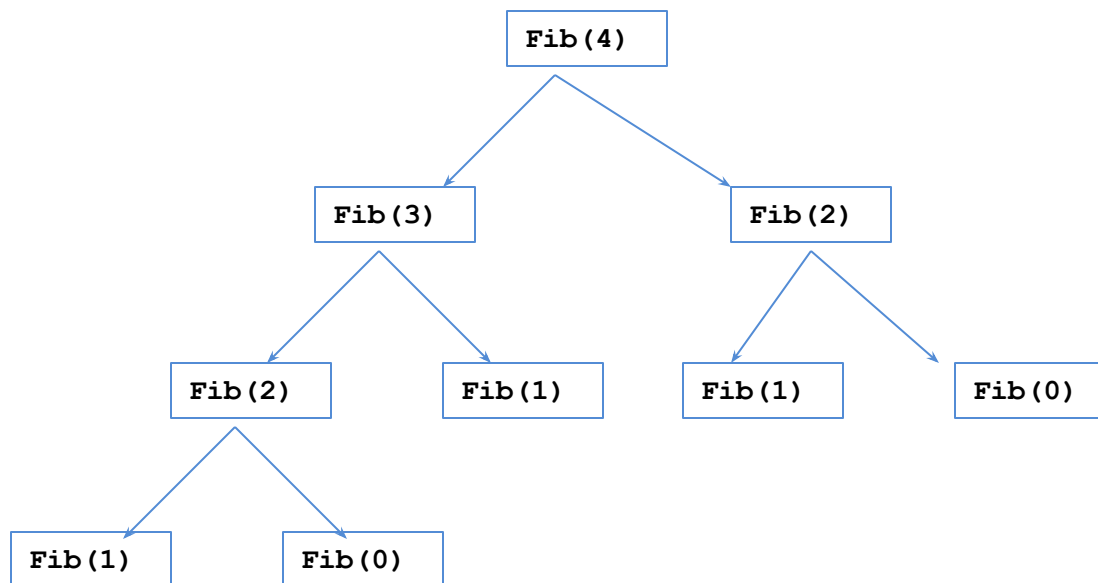
1. How is the “base case” handled?  
2. Could it be that this program will never terminate?

Output:

```
fib of 0 = 1
fib of 1 = 1
fib of 2 = 2
fib of 3 = 3
fib of 4 = 5
```

# Recursion

- How many times is **fib(n)** called for **n=4**?
  - More generally, number of times fib(.) is called, M(n):  
 $M(n) = 1$ , of  $n \leq 1$   
 $= 1 + M(n-1) + M(n-2)$ , otherwise



n	M(n)
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177
11	287
12	465
13	753

# Recursion

- How many times is `fib(n)` called for `n=0` or `1`?

`Fib(1)`

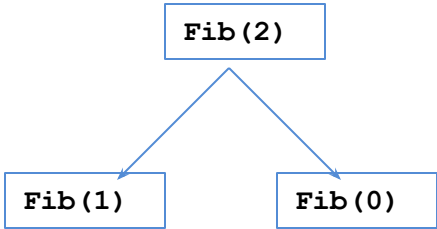
`Fib(0)`

n	M(n)
0	1
1	1



# Recursion

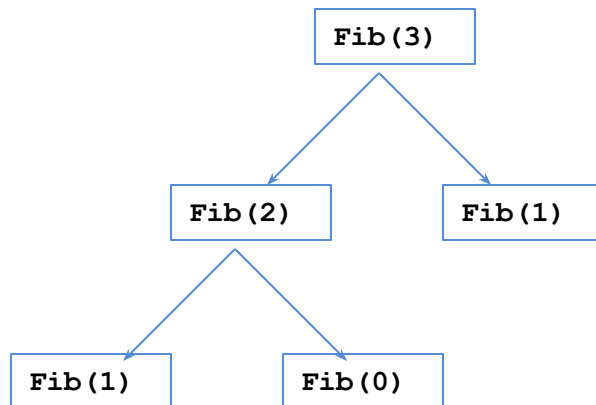
- How many times is `fib(n)` called for `n=2`?



n	M(n)
0	1
1	1
2	3

# Recursion

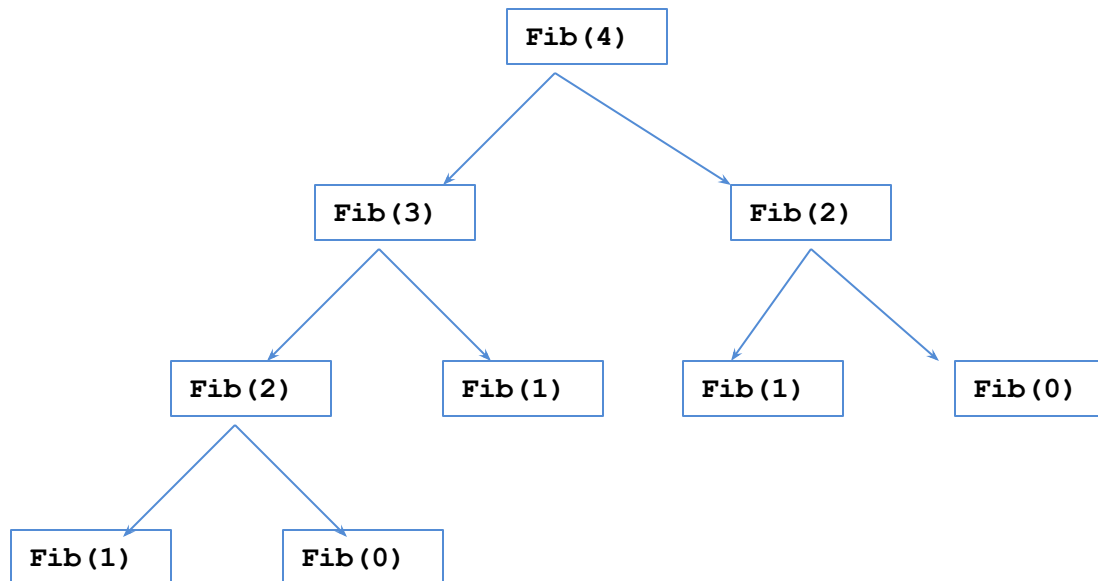
- How many times is **fib**(**n**) called for **n=3**?



n	M(n)
0	1
1	1
2	3
3	5

# Recursion

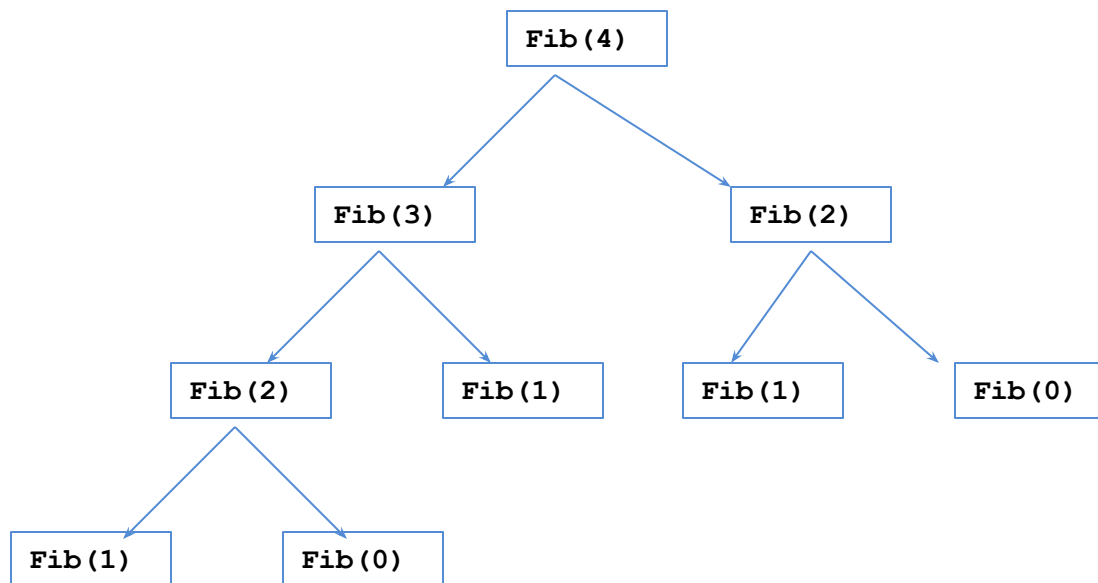
- How many times is **fib**(**n**) called for **n=4**?



n	M(n)
0	1
1	1
2	3
3	5
4	9

# Recursion

- How many times is **fib(n)** called for **n=4**?
  - More generally, number of times fib(.) is called, M(n):  
 $M(n) = 1$ , of  $n \leq 1$   
 $= 1 + M(n-1) + M(n-2)$ , otherwise



n	M(n)
0	1
1	1
2	3
3	5
4	9
Etc.	Etc.

# Recursion

- Global variables vs. local variables
- Counting no. of calls, while computing Fibonacci numbers

```
# Fibonacci-2
# Compute F(k), k >= 0, while counting no. of calls to fib(.)
def fib(x):
    global numFibCalls
    numFibCalls = numFibCalls + 1
    if x == 0 or x == 1:
        return(1)
    else:
        return(fib(x-1) + fib(x-2))

global numFibCalls
for k in range(5):
    numFibCalls = 0
    print('fib of', k, '=', fib(k))
    print("fib called", numFibCalls, "times.")
```

<http://tinyurl.com/czv4wd4j>

# Recursion

- Global variables vs. local variables
- Counting no. of calls, while computing Fibonacci numbers

```
# Fibonacci-2
# Compute F(k), k >= 0, while counting no. of calls to fib(.)
def fib(x):
    global numFibCalls
    numFibCalls = numFibCalls + 1
    if x == 0 or x == 1:
        return(1)
    else:
        return(fib(x-1) + fib(x-2))

global numFibCalls
for k in range(5):
    numFibCalls = 0
    print('fib of', k, '=', fib(k))
    print("fib called", numFibCalls, "times.")
```

Output:

```
fib of 0 = 1
fib called 1 times.
fib of 1 = 1
fib called 1 times.
fib of 2 = 2
fib called 3 times.
fib of 3 = 3
fib called 5 times.
fib of 4 = 5
fib called 9 times.
```

<http://tinyurl.com/czv4wd4j>

# Recursion

- Global variables vs. local variables
- Counting no. of calls, while computing Fibonacci numbers

```
# Fibonacci-2
# Compute F(k), k >= 0, while counting no. of calls to fib(.)
def fib(x):
    global numFibCalls
```

```
        numFibCalls = numFibCalls + 1
        if x == 0 or x == 1:
            return(1)
        else:
            return(fib(x-1)
```

```
global numFibCalls
for k in range(5):
    numFibCalls = 0
    print('fib of', k, '=', fib(k))
    print("fib called", numFibCalls, "times.")
```

Advice: Use global variables sparingly. Since indiscriminate use of 'global' variables is likely to result in incorrect results.

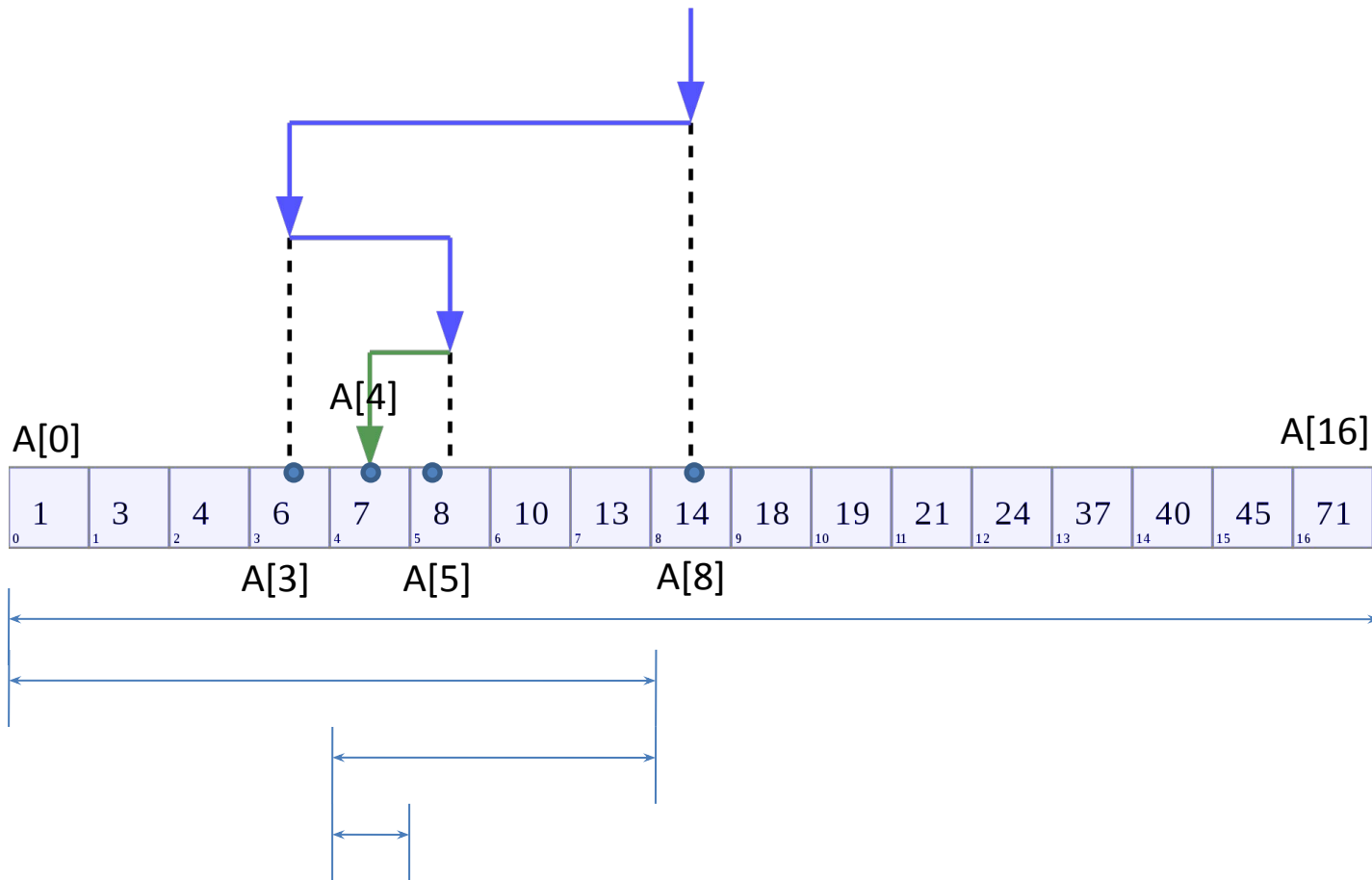
Output:

```
fib of 0 = 1
fib called 1 times.
fib of 1 = 1
fib called 1 times.
fib of 2 = 2
fib called 3 times.
fib of 3 = 3
fib called 5 times.
fib of 4 = 5
fib called 9 times.
```

# Recursion

- Recursion (as a method) to solve some simple problems:
  - Search for a data object using “binary search” in a sorted array (for the present array == vector)

[https://en.wikipedia.org/wiki/Binary\\_search\\_algorithm](https://en.wikipedia.org/wiki/Binary_search_algorithm)





# Recursion

- Recursion (as a method) to solve some simple problems:
  - Search for a data object using “binary search” in a sorted array (for the present array == vector)

# Binary search -2

# Use iteration to locate of data, T, in list A of size n

```
def binary_search(A, n, T):  
    L = 0  
    R = n-1  
    while L <= R:  
        m = int(((L + R) / 2))  
        if A[m] < T:  
            L = m + 1  
        else:  
            if A[m] > T:  
                R = m-1  
            else:  
                return("Located at:" , m)  
    return("Not found")
```

#

n = 17

A = [1, 3, 4, 6, 7, 8, 10, 13, 14, 18, 19, 21, 24, 37, 40, 45, 71]

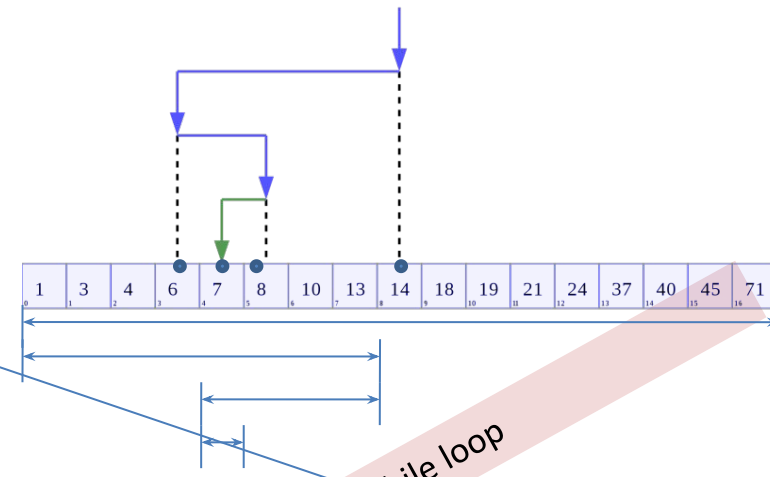
result = binary\_search(A, n, 7)

print(result)

#

result = binary\_search(A, n, 11)

print(result)



# Recursion

- Recursion (**as a method**) to solve some **simple problems**:
  - Search for a data object using “binary search” in a **sorted** array (for the present array == vector)

**# Binary search -2**

**# Use recursion to locate of data, T, in list A indexed L through R**

```
def binary_search(A, L, R, T):  
    if L > R:  
        return(-1)  
    m = int(((L + R) / 2))  
    if T == A[m]:  
        return(m)  
    if T < A[m]:  
        R = m - 1  
    else:  
        L = m + 1  
    return(binary_search(A, L, R, T))
```

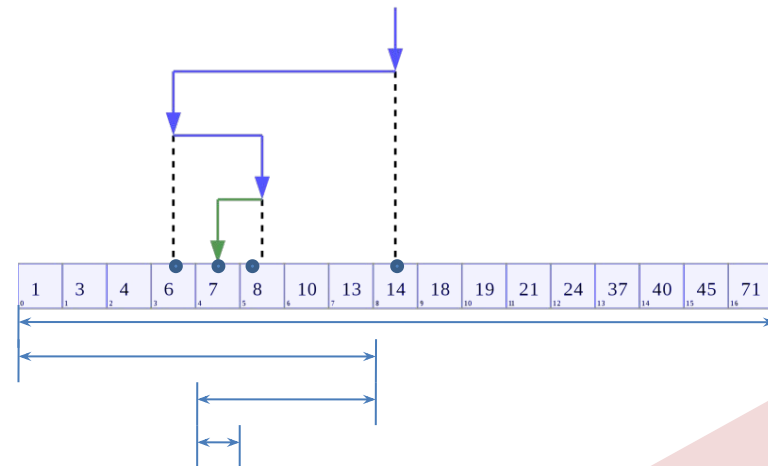
**#**

```
A = [1, 3, 4, 6, 7, 8, 10, 13, 14, 18, 19, 21, 24, 37, 40, 45, 71]
```

```
n = len(A)          # indexed A[0] through A[16]
```

```
print(binary_search(A, 0, 16, 7)) # list, low, high, data
```

```
print(binary_search(A, 0, 16, 9))
```



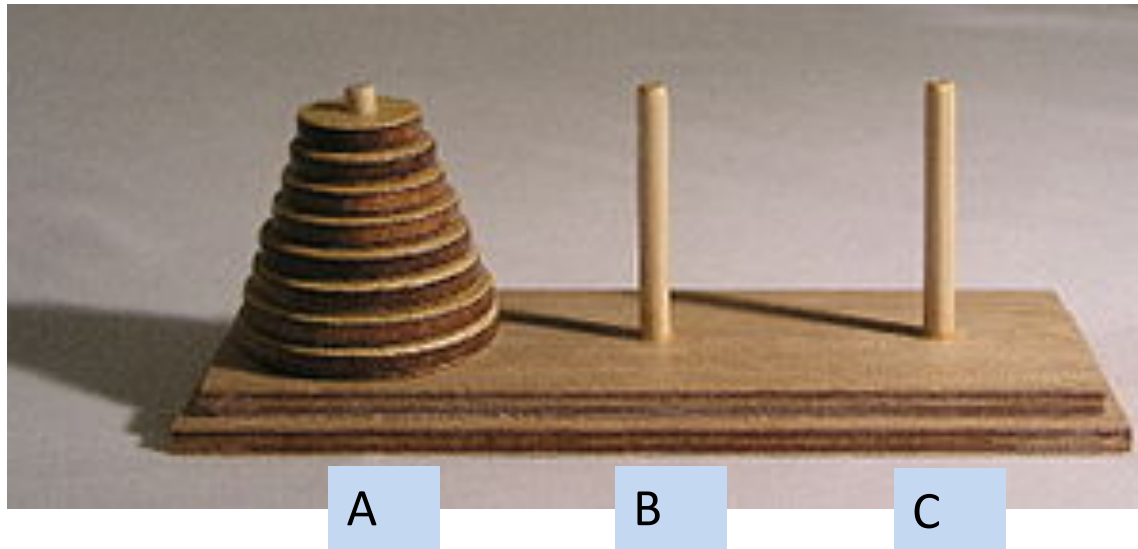
Using recursion

<https://tinyurl.com/2rar95ns>

# Recursion

[https://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](https://en.wikipedia.org/wiki/Tower_of_Hanoi)

- Recursion (as a method) to solve some difficult problems:
  - Tower of Hanoi problem: move N disks from tower A to tower C, possibly using tower B:
    - Only one disk can be moved at a time
    - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
    - No larger disk may be placed on top of a smaller disk



# Recursion

[https://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](https://en.wikipedia.org/wiki/Tower_of_Hanoi)

- Recursion (as a method) to solve some difficult problems:
  - Tower of Hanoi problem: move N disks from tower A to tower C, possibly using tower B:
    - Only one disk can be moved at a time
    - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
    - No larger disk may be placed on top of a smaller disk
- Solutions:
  - For no. of disks = 1 or = 2 it is easy
  - For no. of disks = 3 it is not too difficult
  - For no. of disks = 4 or more, it seems impossible, unless ...

Here is the recursive algorithm:

If  $n = 1$ , “move the disk to Tower C”

Else { “move the top  $n-1$  disk from Tower A to Tower B”;  
“move the remaining 1 disk from Tower A to Tower C”  
“move the  $n-1$  disks from Tower B to Tower C”  
}

# Recursion

If  $n = 1$ , “move the disk to Tower C”  
else {“move the top  $n-1$  disk from Tower A to Tower B”;  
    “move the remaining 1 disk from Tower A to Tower C”  
    “move the  $n-1$  disks from Tower B to Tower C”}

## # Solving the Tower of Hanoi puzzle

```
def TowerOfHanoi(n , src, dest, aux):  
    if n == 1:  
        print("Move disk 1 from src", src,"to dest", dest)  
        return  
    TowerOfHanoi(n-1, src, aux, dest)  
    print("Move disk",n,"from src", src,"to dest", dest)  
    TowerOfHanoi(n-1, aux, dest, src)  
  
#  
n = 4  
#Initial peg = 'A' , Final peg = 'B' , Via peg = 'C'  
TowerOfHanoi(n, 'A', 'B', 'C')
```

# Recursion

If  $n = 1$ , "move the disk to Tower C"  
else {"move the top  $n-1$  disk from Tower A to Tower B";  
    "move the remaining 1 disk from Tower A to Tower C"  
    "move the  $n-1$  disks from Tower B to Tower C"}

## # Solving the Tower of Hanoi puzzle

```
def TowerOfHanoi(n , src, dest, aux):  
    if n == 1:  
        print("Move disk 1 from src", src, "to dest", dest)  
        return  
    TowerOfHanoi(n-1, src, aux, dest)  
    print("Move disk", n, "from src", src, "to dest", dest)  
    TowerOfHanoi(n-1, aux, dest, src)
```

```
#  
n = 4  
#Initial peg = 'A', Final peg = 'B', V: 'C'  
TowerOfHanoi(n, 'A', 'B', 'C')
```

<https://tinyurl.com/mum66zhz>

## Output:

```
Move disk 1 from src A to dest C  
Move disk 2 from src A to dest B  
Move disk 1 from src C to dest B  
Move disk 3 from src A to dest C  
Move disk 1 from src B to dest A  
Move disk 2 from src B to dest C  
Move disk 1 from src A to dest C  
Move disk 4 from src A to dest B  
Move disk 1 from src C to dest B  
Move disk 2 from src C to dest A  
Move disk 1 from src B to dest A  
Move disk 3 from src C to dest B  
Move disk 1 from src A to dest C  
Move disk 2 from src A to dest B  
Move disk 1 from src C to dest B
```

# Recursion

```
If n = 1, "move the disk to Tower C"
else {"move the top n-1 disk from Tower A to Tower B";
      "move the remaining 1 disk from Tower A to Tower C"
      "move the n-1 disks from Tower B to Tower C"}
```

## # Solving the Tower of Hanoi puzzle

```
def TowerOfHanoi(n , src, dest, aux):
    print("new ToH with n = ", n, "src = ", src, "dest = ", dest)
```

```
    if n == 1:
```

```
        # print("Move disk 1 from", src, "to", dest)
        return
```

```
    TowerOfHanoi(n-1, src, aux, dest)
```

```
    # print("Move disk", n, "from", src, "to", dest)
```

```
    TowerOfHanoi(n-1, aux, dest, src)
```

```
#
```

```
n = 4
```

```
#Initial peg = 'A', Final peg = 'C'
```

```
TowerOfHanoi(n, 'A', 'B', 'C')
```

## Sequence of calls to TowerOfHanoi

```
new ToH with n = 4 src = A dest = B
```

```
new ToH with n = 3 src = A dest = C
```

```
new ToH with n = 2 src = A dest = B
```

```
new ToH with n = 1 src = A dest = C
```

```
new ToH with n = 1 src = C dest = B
```

```
new ToH with n = 2 src = B dest = C
```

```
new ToH with n = 1 src = B dest = A
```

```
new ToH with n = 1 src = A dest = C
```

```
new ToH with n = 3 src = C dest = B
```

```
new ToH with n = 2 src = C dest = A
```

```
new ToH with n = 1 src = C dest = B
```

```
new ToH with n = 1 src = B dest = A
```

```
new ToH with n = 2 src = A dest = B
```

```
new ToH with n = 1 src = A dest = C
```

```
new ToH with n = 1 src = C dest = B
```

# Recursion

```
If n = 1, "move the disk to Tower C"  
else {"move the top n-1 disk from Tower A to Tower B";  
      "move the remaining 1 disk from Tower A to Tower C"  
      "move the n-1 disks from Tower B to Tower C"}
```

Number of times ToH is called for n:

n = 1: 1

n = 2: 3

n = 3: 7

n = 4: 15

n = 5: ?

n = 6: ?

Etc.

## Sequence of calls to TowerOfHanoi

```
new ToH with n = 4 src = A dest = B  
new ToH with n = 3 src = A dest = C  
new ToH with n = 2 src = A dest = B  
new ToH with n = 1 src = A dest = C  
new ToH with n = 1 src = C dest = B  
new ToH with n = 2 src = B dest = C  
new ToH with n = 1 src = B dest = A  
new ToH with n = 1 src = A dest = C  
new ToH with n = 3 src = C dest = B  
new ToH with n = 2 src = C dest = A  
new ToH with n = 1 src = C dest = B  
new ToH with n = 1 src = B dest = A  
new ToH with n = 2 src = A dest = B  
new ToH with n = 1 src = A dest = C  
new ToH with n = 1 src = C dest = B
```



# Recursion

If  $n = 1$ , “move the disk to Tower C”  
else {“move the top  $n-1$  disk from Tower A to Tower B”;  
    “move the remaining 1 disk from Tower A to Tower C”  
    “move the  $n-1$  disks from Tower B to Tower C”}

Number of times ToH is called for  $n$ :

$n = 1$ : 1

$n = 2$ : 3

$n = 3$ : 7

$n = 4$ : 15

$n = 5$ : ?

$n = 6$ : ?

Etc.

More generally:

$$P(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2 * P(n-1) + 1 \end{cases}$$

## Sequence of calls to TowerOfHanoi

```
new ToH with n = 4 src = A dest = B
new ToH with n = 3 src = A dest = C
new ToH with n = 2 src = A dest = B
new ToH with n = 1 src = A dest = C
new ToH with n = 1 src = C dest = B
new ToH with n = 2 src = B dest = C
new ToH with n = 1 src = B dest = A
new ToH with n = 1 src = A dest = C
new ToH with n = 3 src = C dest = B
new ToH with n = 2 src = C dest = A
new ToH with n = 1 src = C dest = B
new ToH with n = 1 src = B dest = A
new ToH with n = 2 src = A dest = B
new ToH with n = 1 src = A dest = C
new ToH with n = 1 src = C dest = B
```

# Q&A

- On recursion
- On  $n!$
- On palindromes
- On Fibonacci(k)
- On binary search
- On Tower of Hanoi