# Function

## Function Abstraction

A function or procedure is a named encapsulation of code that can be reused. It gives better modularity in a program. A set of functions such as `input`, `sqrt()` etc. are supplied as library with the language.
But programming languages provide support for user defined function.

## Example

```
def factorial(n):      # factF1.py
    fact = 1
    for i in range(1,n+1):
        fact = fact * i
    return fact

n = input("Enter a +ve integer: ")
print n, "!=", factorial(n)
```

## Function Definition

- `def`: a function definition begins with this *keyword*.
- *Name*: factorial is the name of the function.
- *Formal parameter(s)*: this refers to actual parameter from the caller.
- *Function body*: `fact = ··· return fact`
- Return value: The function may return a value to the caller - `return fact`

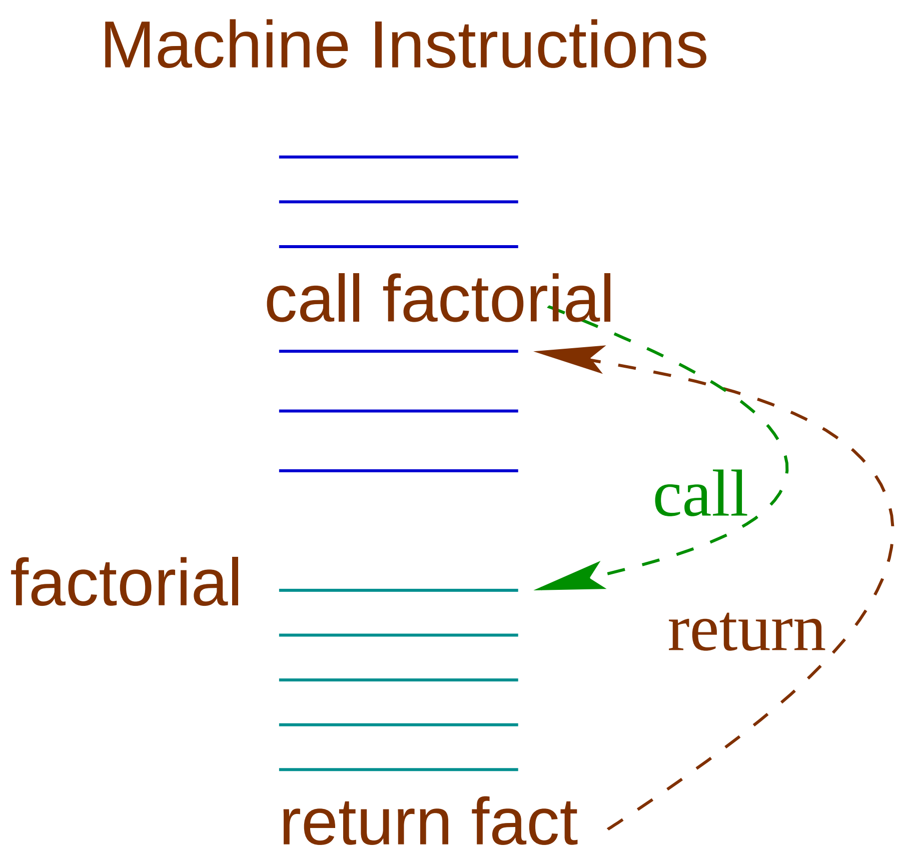
## Function Call/Invocation

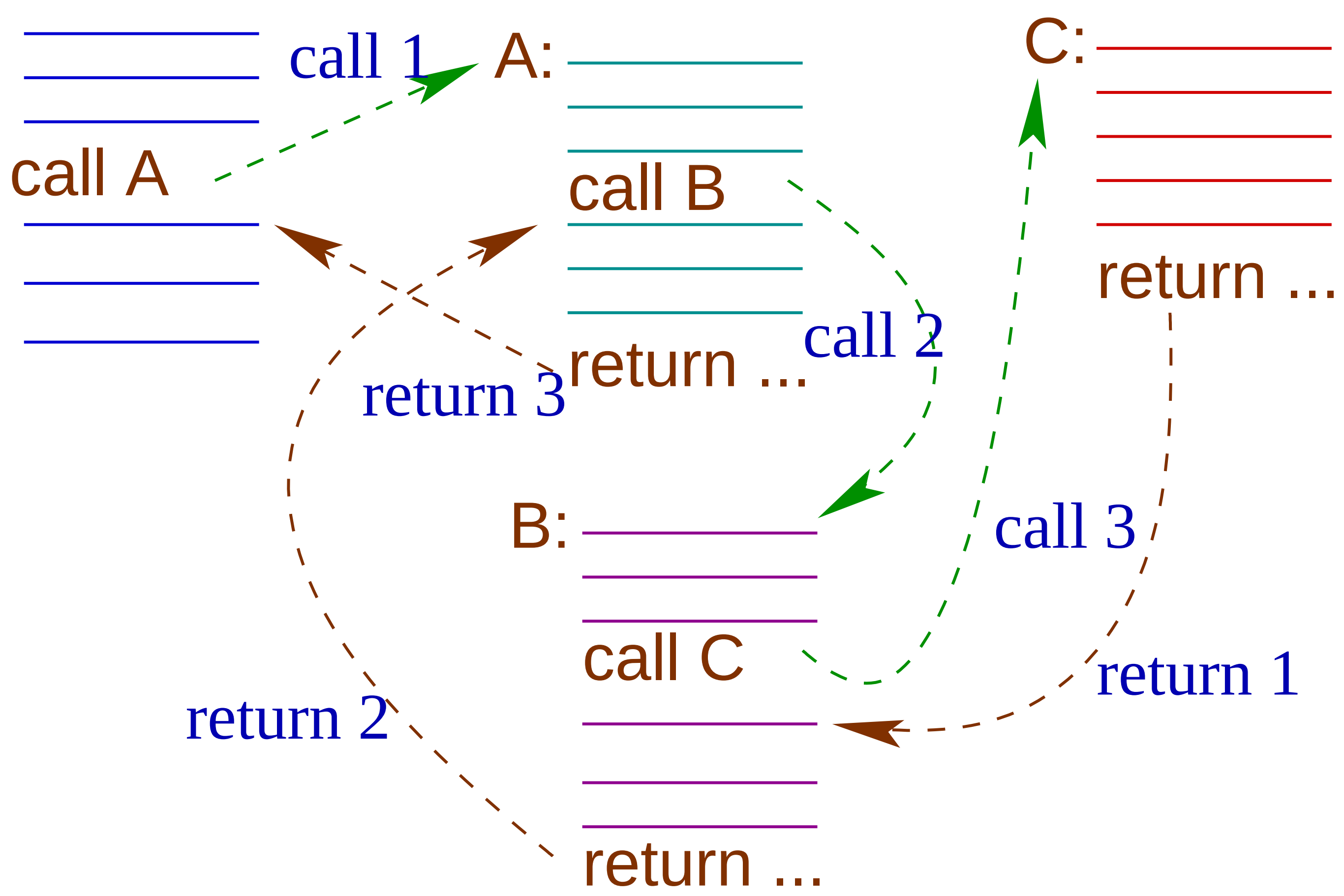
The function is called by passing the *actual parameter* (if there is one). It is `n` in this example. The return value (if there is one) is returned to the caller. The return value is printed in this example.

```
print n, "!=", factorial(n)
```

## Flow of Control

When a function is called, the control of computation is transfered to the beginning of the called function. Once the execution of the callee (the called function) is over, the control is transfered back to the instruction next to the call in the caller.


Machine Instructions
call factorial
call
factorial
return
return fact


call 1
A:
C:
call A
call B
return ...
call 2
return ...
return 3
B:
call 3
call C
return 1
return 2
return ...

## Example 1

Write a Python function that takes an integer and returns the sum of its decimal digits.

```
def sumOfDig(n):
    ..........

n = input("Enter an integer: ")
print "digSum(", n,") = ", sumOfDig(n)
```

## Example 2

Write a Python program that reads a positive integer $n$ and calls the function sumBinDig(n) to compute and return the sum of the binary digits of $n$. The program then prints the value.

## Example 3

Write a Python program that reads two positive integer $s$ and $l$, calls the function lcm(s,l) to compute and return the lcm of $s$ and $l$. The program then prints the value.

## Example 4

Write a Python program that reads a list of integers $l$ and calls the function maxL(l) that returns the largest element of $l$. Then the program prints it.

## Harmonic Series

The following series is called an harmonic series:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots$$

## Example 5

Write a Python program that reads a positive integer $n$ and calls the function sumH(n) to compute the sum of the Harmonic series up to the $n^{th}$ term and returns it. Then the program prints the returned value.

Input: 5

Output: 2.28333333333

The output is a floating-point number.

## Example 6

Write a Python program that reads a positive integer $n$ and also a number x. It calls the function xPowN(x,n) to compute $x^n$ and returns the value. The program prints the returned value.

Input: 5, 2.5

Output: 97.65625

You are not allowed to use `pow(x,y)` or `**n`.

## Example 7

Modify the previous program for any integer $n$.

You are not allowed to `pow(x,y)` or `**n`.

# Inductive Definition and Recursive Function

## Factorial Function

Consider the following definition (recursive/Inductive) of the factorial function.

$$n! = \begin{cases} 1, & \text{if } n = 0, \\ n \times (n-1)!, & \text{if } n > 0. \end{cases}$$

The function is used to define itself. The definition is an equation with a computational counterpart.

**Example**

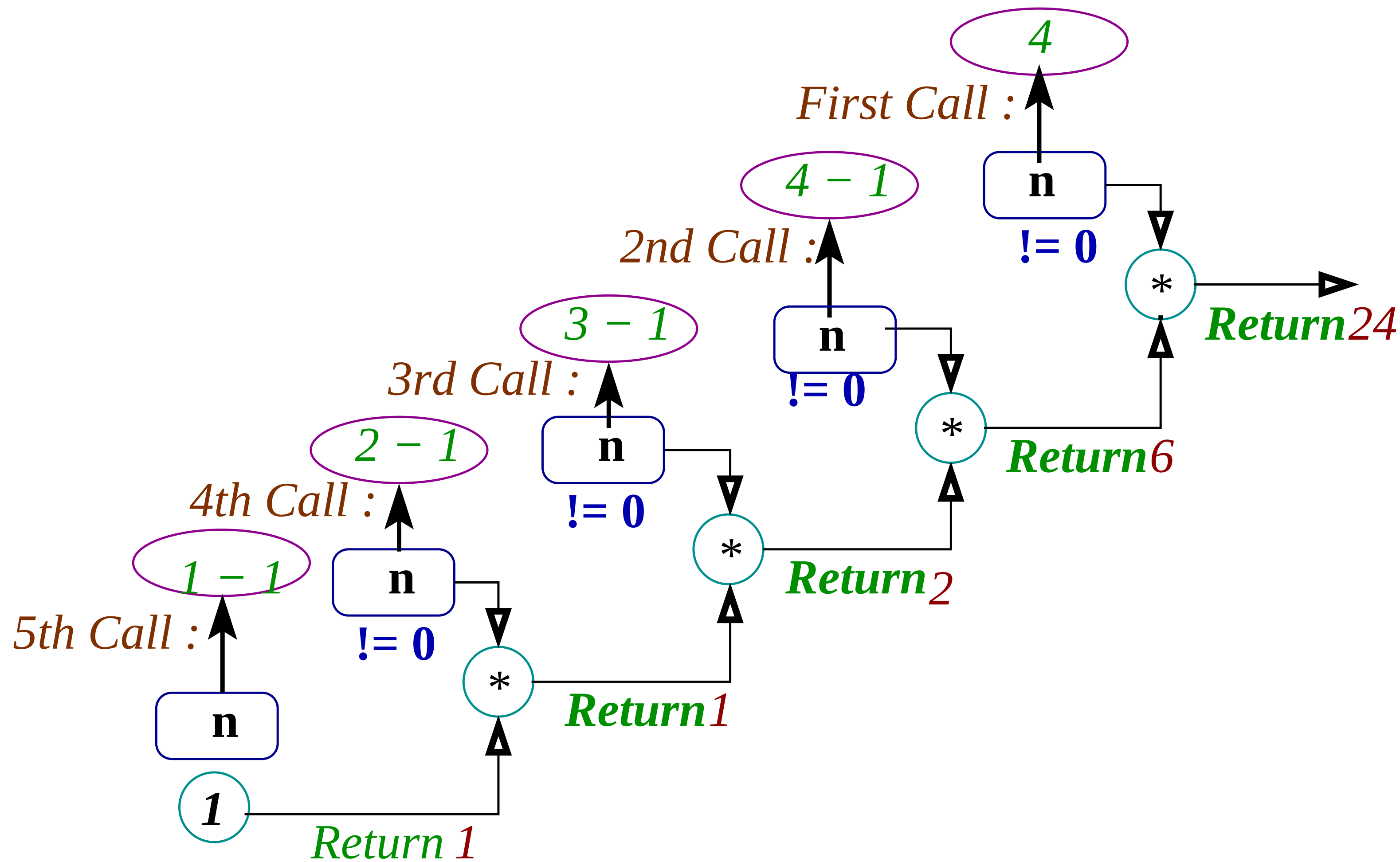
```
def factorial(n):  # factRF1.py
    if n == 0: return 1
    return n*factorial(n-1)
n = input("Enter a +ve integer: ")
print n, "!=", factorial(n)
```

# Computation of 4!

$$\begin{aligned}
4! &= 4 \times 3! \\
&= 4 \times (3 \times 2!) \\
&= 4 \times (3 \times (2 \times 1!)) \\
&= 4 \times (3 \times (2 \times (1 \times 0!))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 4 \times (3 \times (2 \times 1)) \\
&= 4 \times (3 \times 2) \\
&= 4 \times 6 \;=\; 24
\end{aligned}$$

## Note

- There is no value computation in the first four steps. The function is being unfolded.
- The value computation starts only after the basis of the definition is reached.
- Last four steps computes the values.


4
First Call :
n
!= 0
*
Return24
4 − 1
2nd Call :
n
!= 0
*
Return6
3 − 1
3rd Call :
n
!= 0
*
Return2
2 − 1
4th Call :
n
!= 0
*
Return1
1 − 1
5th Call :
n
1
Return 1

## Another Recursive Factorial

```
def factorial(n, fact):  # factRF2.py
    if n == 0: return fact
    return factorial(n-1, fact*n)

n = input("Enter a +ve integer: ")
print n, "!=", factorial(n,1)
```

## Fibonacci Sequence

$$f_n = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ f_{n-1} + f_{n-2}, & \text{if } n > 1. \end{cases}$$

## Iterative Fibonacci Function

```
def fib(n):      # fibIF.py
    if n<=1: return n
    f0, f1 = 0, 1
    for i in range(n+1)[2:]:
        f0, f1 = f1, f0+f1
    return f1

n = input("Enter a positive integer: ")
print "fib(",n,") = ", fib(n)
```

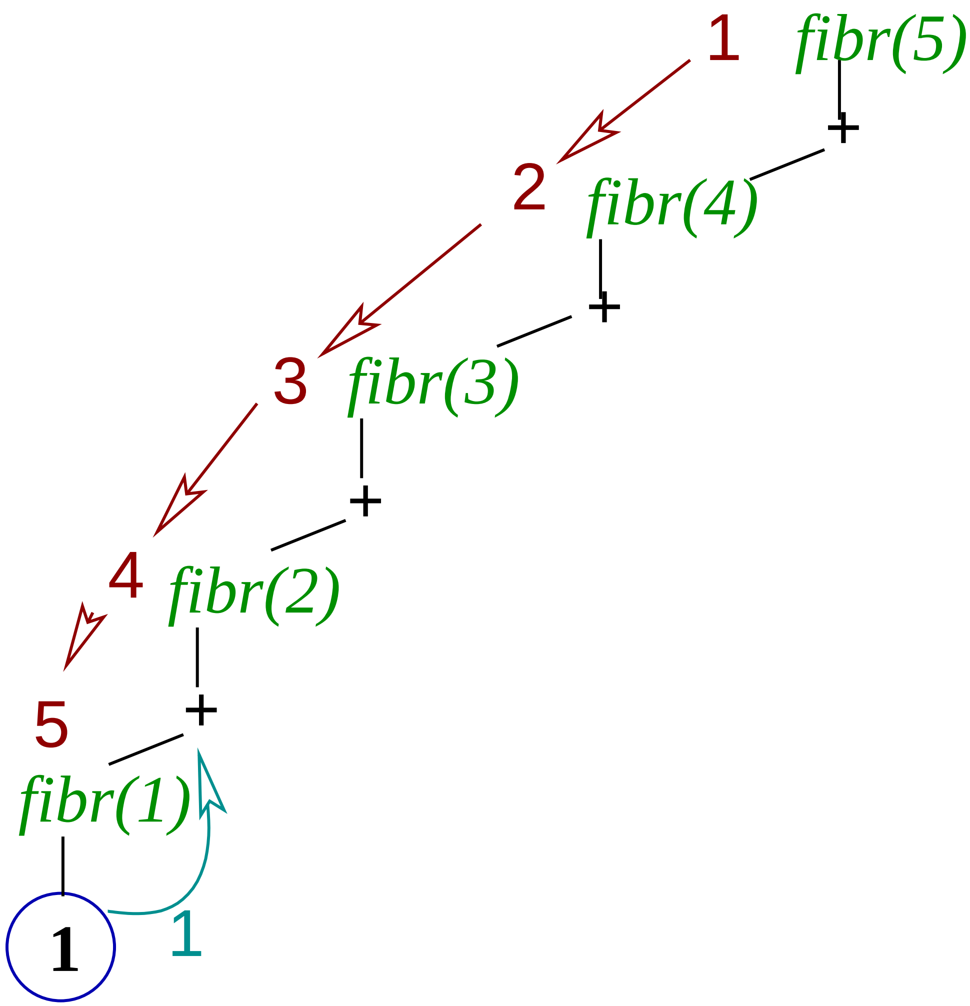
## Recursive Fibonacci Function

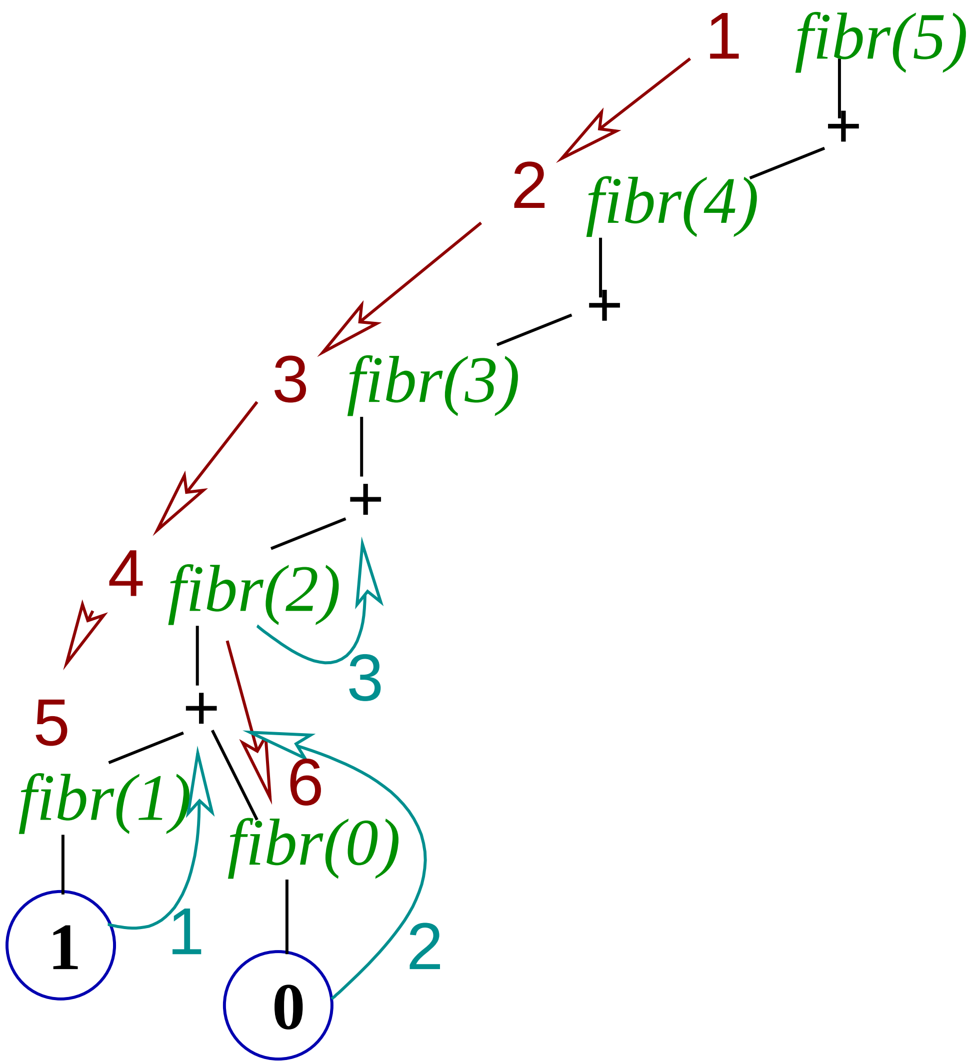
```
def fib(n):      # fibRF.py
    if n<=1: return n
    return fib(n-1) + fib(n-2)

n = input("Enter a positive integer: ")
print "fib(",n,") = ", fib(n)
```

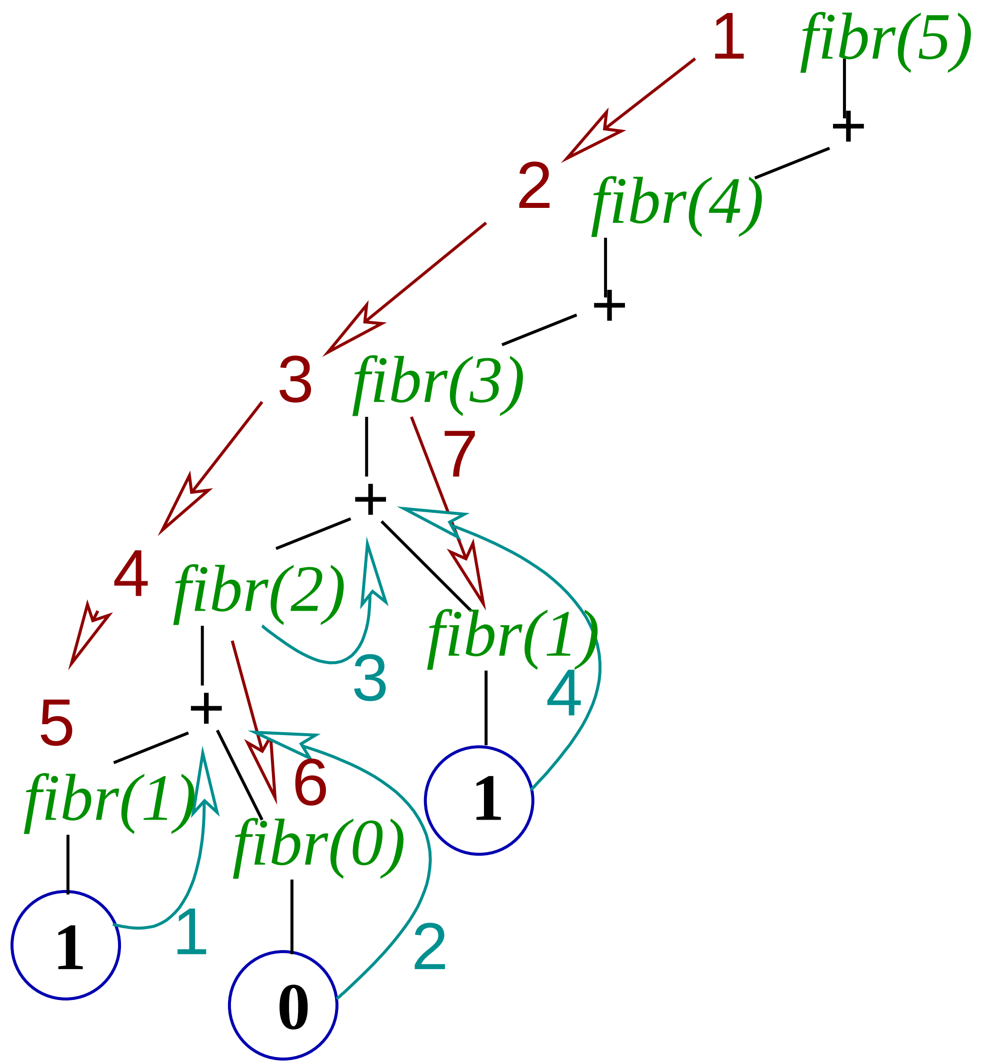
This function is very inefficient.

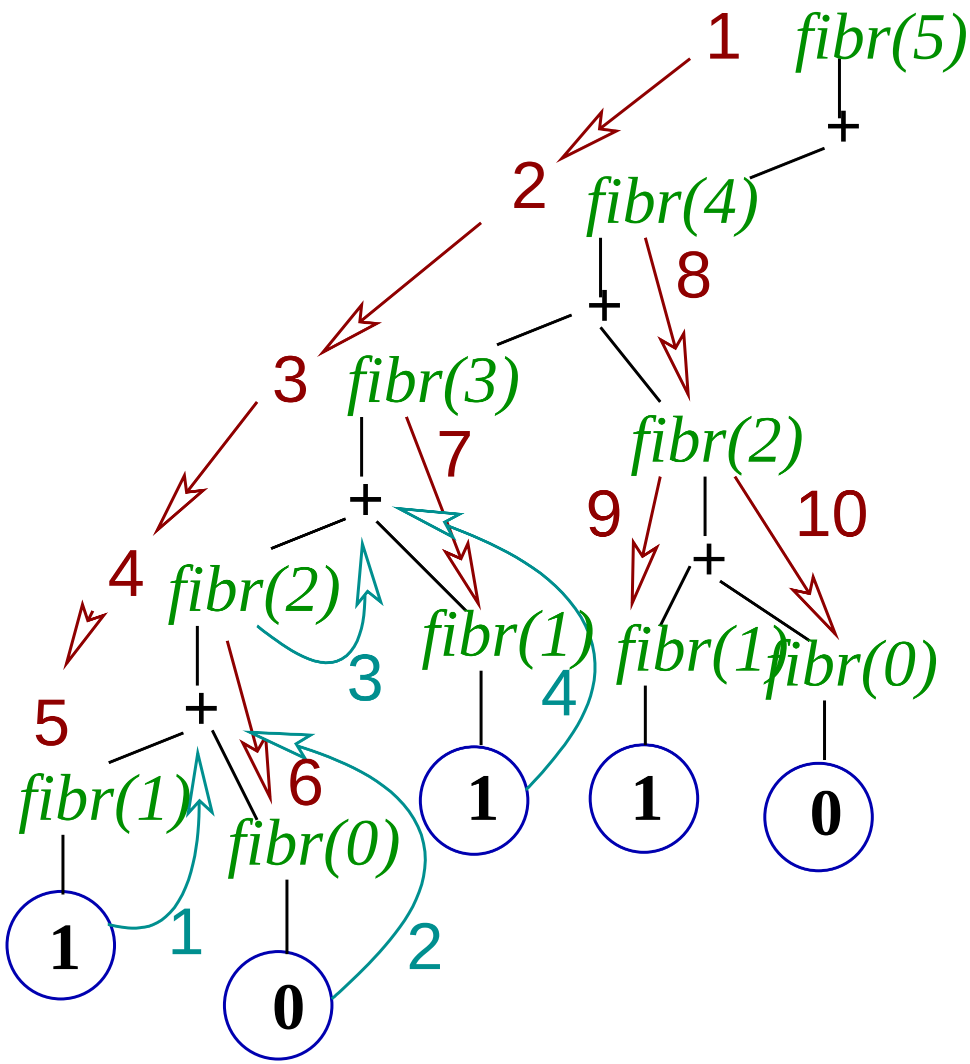
## The Call Tree: $n = 5$

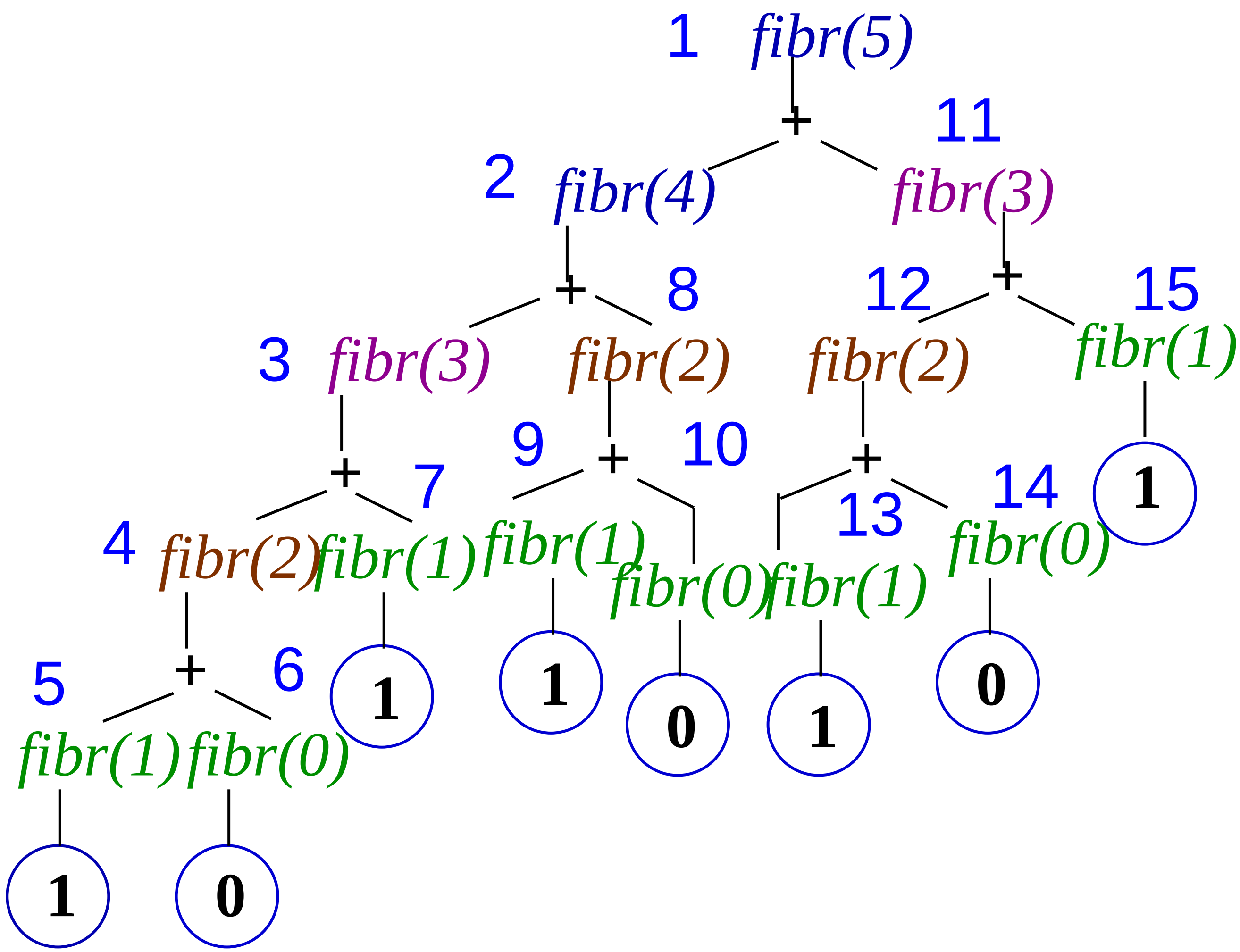
The call sequence for $n = 5$ is as follows.


1
fibr(5)
+
2
fibr(4)
+
3
fibr(3)
+
4
fibr(2)
+
5
fibr(1)
1
1


1
fibr(5)
+
2
fibr(4)
+
3
fibr(3)
+
4
fibr(2)
3
5
+
6
fibr(1)
fibr(0)
1
1
0
2


1 fibr(5)
+
2 fibr(4)
+
3 fibr(3)
7
+
4 fibr(2)
3
fibr(1)
4
5
+
fibr(1)
6
fibr(0)
1
1
1
2
0


1
fibr(5)
+
2
fibr(4)
8
+
3
fibr(3)
7
fibr(2)
9
10
+
4
fibr(2)
3
fibr(1)
4
fibr(1)
fibr(0)
5
+
6
fibr(1)
fibr(0)
1
1
1
0
1
0
2


1 fibr(5)
+
2 fibr(4)
11 fibr(3)
+
8
3 fibr(3)
fibr(2)
12 fibr(2)
+
15 fibr(1)
+
9
10
7
4 fibr(2)
fibr(1)
fibr(1)
fibr(0)
13 fibr(1)
14 fibr(0)
1
+
5 fibr(1)
6 fibr(0)
1
1
0
1
0
1
0

**Note**

Fifteen calls are made and seven additions are performed. This could have been done by only four additions in a iterative program.

| | | | | | | |
|---|---|---|---|---|---|---|
| $n$ | 0 | 1 | 2 | 3 | 4 | 5 |
| $fibr(n)$ | 0 | 1 | 1 | 2 | 3 | 5 |
| $op$ | | | + | + | + | + |

## Efficient Recursive Fibonacci

```
def fib(n, f0, f1):      # fibERF.py
    if n == 0: return f0
    if n == 1: return f1
    return fib(n-1, f1, f0+f1)
n = input("Enter a positive integer: ")
print "fib(",n,") = ", fib(n,0,1)
```

# Computation of $sin(x)$

## Power Series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \cdots$$

Finite number of terms of this infinite series may be used to compute an approximate value of $\sin(x)$, where $x$ is in radian.

$\pi$ radian is $180°$

$$
\begin{aligned}
\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \cdots \\
&= \sum_{i \geq 0} (-1)^i \frac{x^{2i+1}}{(2i+1)!} \\
&= \sum_{i \geq 0} t_i, \text{ where } t_i = (-1)^i \frac{x^{2i+1}}{(2i+1)!}
\end{aligned}
$$

## Inductive Definition of $t_i$

$$t_i = \begin{cases} x & \text{if } i = 0, \\ -t_{i-1}\dfrac{x^2}{2i(2i+1)} & \text{if } i > 0. \end{cases}$$

This is also called recurrence relation or recursive definition.

## Approximation of $\sin(x)$

The sum up to the $n^{th}$ term $(S_n)$ of the series gives an approximate value of $\sin(x)$. The inductive definition of $S_n$ is

$$S_n = \begin{cases} t_0 & \text{if } n = 0, \\ s_{n-1} + t_n & \text{if } n > 0. \end{cases}$$

## From Inductive Definition to Iterative Process

An iterative process of computation can be obtained from the inductive definition.

1. Start from initial values of $t_i$ and $S_i$.
2. Repeat the following two steps.
   (a) Compute the next term, $t_{i+1}$.
   (b) Compute the next approximate value of $\sin(x)$ by computing $S_{i+1}$.

## Termination of Iteration

The process is to be terminated after a finite number of iterations. The termination may be

1. after a fixed number of iterations, or
2. after achieving a pre-specified accuracy.

## Fixed no Iteration

```
# sin1.py : computation of sin x
import math
def mySin(x):
    x = x%(2.0*math.pi)
    term = x
    termSum = term
    for i in range(100)[1:]:
        factor = 2.0*i
        factor = factor*(factor+1.0)
```

```
        factor = -x*x/factor
        term = term * factor
        termSum = termSum + term
    return termSum
a = input("Input angle in degree: ")
x = math.pi*a/180.0
print "sin(", x, ") = ", mySin(x)
```