Introduction to Computer and Programming

Lecture 6

Yue Zhang
Westlake University

August 1, 2023

WestlakeNLP

Chapter 6.

Functions and Modules

WestlakeNLP

Lambda Expressions

$$f(x) = x^2$$

```
f = lambda x: x*x
```

keyword: lambda; ***parameter***: x; ***return value expression***: x*x

WestlakeNLP

Lambda Expressions

$$f(x, y) = x + 2y$$

```
f = lambda x,y: x + 2*y
```

parameter list: x,y; ***return value expression***: x+2*y

WestlakeNLP

Lambda Expressions

$$f(x, y) = x + 2y$$

```
f = lambda x,y: x + 2*y
```

parameter list: x,y; ***return value expression***: x+2*y

```
>>> f = lambda x,y:x+2*y
>>> f(1,2)
5
>>> f(3,4)
11
```

arguments: 1,2; 3,4

arguments fill the **parameters**

WestlakeNLP

Functions

- Function Definition

  f = lambda x,y: x+2*y

- Function Call

  f(1,2) equvialent as: 1+2*2

WestlakeNLP

Functions

Function objects

```
>>> f = lambda x,y:x+2*y
>>> type(f)
<class 'function'>
>>> g = f           # assignment
>>> g(1,2)
5
```

- Functions are objects, and calls are operators. just as $+$, $-$, $*$, $/$, %, $>$, $<$, [] (get slice).

WestlakeNLP

Functions

Default arguments

```
>>> import math
>>> math.log(256)
5.545177444479562
>>> math.log(256,2)
8.0
```

- By default the base is *e*.

Functions

Default arguments

```
>>> f=lambda x=1,y=2:x+2*y
>>> f(1,2)
5
>>> f()
5
>>> f(1)
5
```

- In expression "x=1, y=2", 1 and 2 are default values.

WestlakeNLP

Functions

Keyword argument

```
>>> f=lambda x=1,y=2:x+2*y
>>> f(y=1)
3
```

- Here we specify y only.

Functions

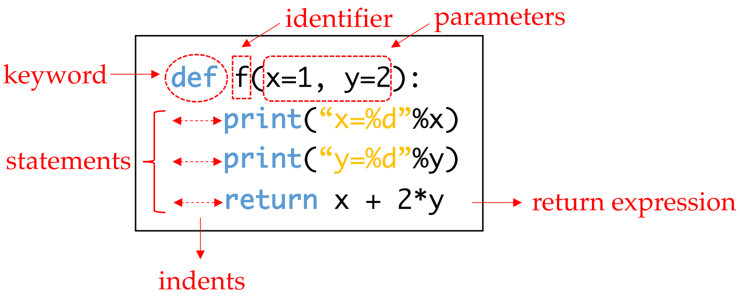
Functions that contain statements can be used for more complex tasks.

Multiple Statements

```
>>> def f(x=1,y=2):
...     print("x=%d"%x)
...     print("y=%d"%y)
...     return x+2*y
...
>>> type(f)
<class 'function'>
```

```
>>> f()        # <--- 1+2*2
x=1
y=2
5
>>> f(3)       # <--- 3+2*2
x=3
y=2
7
>>> f(y=4)     # <--- 1+2*4
x=1
y=4
9
>>> f(5,6)     # <--- 5+2*6
x=5
y=6
17
```

Functions that contain statements can be used for more complex tasks.



Functions

Functions without return statements

```
>>> def h():
...     print("Hello.")
...
>>> h()
Hello.
>>> a = h()
Hello.
>>> a
>>>
```

What is the return value of *a*? Nothing!

```
>>> type(a)
<class 'NoneType'>
>>> str(a)
'None'
```

WestlakeNLP

Functions

Why do we need functions?

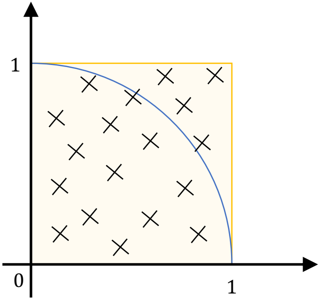
- Functions make code more modularized and easier to maintain.
- Functions can be put into modules(python files) and imported.

Functions

Functions offer modularity.

mcpi.py

```
import random
import math
N = int(input("n="))
M = 0
i = 0
while i < N:
    x = random.random()
    y = random.random()
    if x*x + y*y < 1:
        M += 1
    i += 1
pi = 4 * (M/N)
print("Approxmate Value: %f"%pi)
print("Error: %f"%(math.pi-pi))
```



Functions

Functions offer modularity.

mcpi.py

```
import random
import math
N = int(input("n="))
M = 0
i = 0
while i < N:
    x = random.random()
    y = random.random()
    if x*x + y*y < 1:
        M += 1
    i += 1
pi = 4 * (M/N)
print("Approxmate Value: %f"%pi)
print("Error: %f"%(math.pi-pi))
```

Function calls make the main loop more easier to understand and debug. ⟶

mcpi_function.py

```
import random
import math
N = int(input("n="))
M = 0
# functions
def sample_point():
    x = random.random()
    y = random.random()
    return (x, y)   # tuple as return type

def point_in_circle(x, y):
    if x*x + y*y < 1:
        return True
    else:
        return False

# iteration
for i in range(N):       # for loop
    x, y = sample_point()
    if point_in_circle(x, y):
        M += 1
pi = 4 * (M/N)
print("Approxmate Value: %f"%pi)
print("Error: %f"%(math.pi-pi))
```

Modules

Putting functions into modules.

mcpi_function.py

```
import random
import math
N = int(input("n="))
M = 0
# functions
def sample_point():
    x = random.random()
    y = random.random()
    return (x, y)   # tuple as return type

def point_in_circle(x, y):
    if x*x + y*y < 1:
        return True
    else:
        return False

# iteration
for i in range(N):        # for loop
    x, y = sample_point()
    if point_in_circle(x, y):
        M += 1
pi = 4 * (M/N)
print("Approxmate Value: %f"%pi)
print("Error: %f"%(math.pi-pi))
```

circle.py

```
import random
def sample_point():
    x = random.random()
    y = random.random()
    return (x, y)   # tuple as return type

def point_in_circle(x, y):
    if x*x + y*y < 1:
        return True
    else:
        return False
```

mcpi_module.py

```
import circle
import math
N = int(input("n="))
M = 0
# iteration
for i in range(N):
    x, y = circle.sample_point()
    if circle.point_in_circle(x, y):
        M += 1
pi = 4 * (M/N)
print("Approxmate Value: %f"%pi)
print("Error: %f"%(math.pi-pi))
```

Modules

Where does Python look for module files?

- There is one environment variable in the OS, called PYTHON_PATH.
- Python looks for PYTHON_PATH and the current working folder, for modules to import.
- You can modify PYTHON_PATH to include folders that contain your modules.

Modules

Modules can contain variables in addition to functions.
e.g., math.pi, math.e

circle.py

```python
import random

radius = 1.0

def sample_point():
    x = random.random()*radius
    y = random.random()*radius
    return (x, y)    # tuple

def point_in_circle(x, y):
    if x*x + y*y < radius:
        return True
    else:
        return False
```

```
>>> import circle
>>> circle.radius
1.0
>>> circle.point_in_circle(1, 2)
False
>>> circle.point_in_circle(0.2, 0.2)
True
```

WestlakeNLP

Modules

importing variables directly from modules

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import sqrt
>>> sqrt(25)
5.0
```

WestlakeNLP

Modules

giving alternative names to imported modules

```
>>> import math as m
>>> m.e
2.718281828459045
>>> m.log(100)
4.605170185988092
```

Modules

The __builtins__ module

- All the gloablly available functions in Python are defined in the __builtins__ module.
  e.g., str, float, len, sum, ···

```
>>> t=(1,2,3)
>>> sum(t)
6
>>> __builtins__.sum(t)
6
>>> __builtins__.str(t)
'(1, 2, 3)'
```

WestlakeNLP

Flow of Execution

Module import and function call involve **flow of execution**

- module import:
  executes all the statements in the imported module;
- function call:
  executes all the statements in the called function.

Flow of Execution

circle.py

```python
import random
radius = 1.0
def sample_point():
    x = random.random()*radius
    y = random.random()*radius
    return (x, y)   # tuple
def point_in_circle(x, y):
    print("Executing point_in_circle().")
    if x*x + y*y < radius:
        return True
    else:
        return False
print("Executing circle.py.")
```

```
>>> import circle
Executing circle.py.
>>> circle.point_in_circle(1,1)
Executing point_in_circle().
False
```

WestlakeNLP

Flow of Execution

circle.py

```python
import random
radius = 1.0
def sample_point():
    x = random.random()*radius
    y = random.random()*radius
    return (x, y)    # tuple
def point_in_circle(x, y):
    print("Executing point_in_circle().")
    if x*x + y*y < radius:
        return True
    else:
        return False
print("Executing circle.py.")
```

```
>>> import circle
Executing circle.py.
>>> circle.point_in_circle(1,1)
Executing point_in_circle().
False
```

- When importing circle

  - one print statement is executed.

  - the assignment radius=1.0 and the function definitions, def sample_point and def point_in_circle are executed.

  - when def point_in_circle is defined, the print statement inside it is **not** executed.

- When calling circle.point_in_circle(1,1)

  - one print statement is executed.

WestlakeNLP

Namespace

Inside a module import execution, or inside a function call, there is a **local** namespace.

circle.py

```
import random
radius = 1.0
def sample_point():
    x = random.random()*radius
    y = random.random()*radius
    return (x, y)   # tuple
def point_in_circle(x, y):
    print("Executing point_in_circle().")
    if x*x + y*y < radius:
        return True
    else:
        return False
print("Executing circle.py.")
```

```
>>> import circle
>>> radius
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'radius' is not defined
>>> circle.radius
1.0
```

WestlakeNLP

Namespace

Inside a module import execution, or inside a function call, there is a **local** namespace.

```
>>> x=1
>>> def f(a):
...     y = a+x
...     return y*y
...
>>> f(1)
4
>>> f(2)
9
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

WestlakeNLP

Namespace

Inside a module import execution, or inside a function call, there is a **local** namespace.

Hence, namespaces are associated with the point of execution.

Namespace

- After importing, identifiers in a module's local namespace can be accessed as a module object's attributes.

circle.py

```
import random
radius = 1.0
def sample_point():
    x = random.random()*radius
    y = random.random()*radius
    return (x, y)   # tuple
def point_in_circle(x, y):
    print("Executing point_in_circle().")
    if x*x + y*y < radius:
        return True
    else:
        return False
print("Executing circle.py.")
```

```
>>> import circle
Executing circle.py.
>>> circle.point_in_circle(1,1)
Executing point_in_circle().
False
```

WestlakeNLP

Namespace

How do I know the current namespace?

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '
    __name__', '__package__', '__spec__']
```

How do I know what is in a module?

```
>>> import math as m
>>> dir(m)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '
    __spec__',  'acos', 'acosh', 'asin', 'asinh', ...]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '
    __name__', '__package__', '__spec__', 'm']
```

↑ *m* in current namespace

Namespace

Local namespaces are preferred to global namespaces, and then __builtins__.

```
>>> a=1
>>> def f():
...     a=2
...     print(a)
...
>>> def g():
...     a=3
...     print(a)
...
>>> f()
2
>>> g()
3
>>> print(a)
1
```

Namespace

How can I change a global variable during function call?

```
>>> a=1
>>> def f():
...     a=2
...     print(a)
...
>>> f()
2
>>> a
1
```

```
>>> a=1
>>> def f():
...     global a
...     a=2
...     print(a)
...
>>> f()
2
>>> a
2
```

The global statement specifies the namespace of a variable.

WestlakeNLP

Namespace

How can I access a built-in function if I have a function with the same name?

```
>>> def sum(a,b=0,c=0):
...     return a+b+c
...
>>> sum(1,2)
3
>>> sum(1,3,5)
9
>>> t=(1,3,5,7,9)
>>> sum(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum
TypeError: can only concatenate tuple (not "int") to tuple
>>> __builtins__.sum(t)
25
```

WestlakeNLP

Recursive Function Calls

Calls by a function to itself

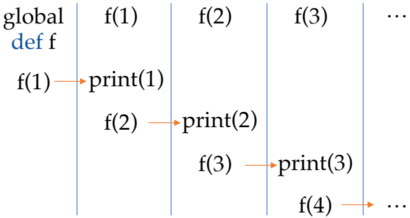
```
>>> def f(x):
...     print(x)
...     f(x+1)
...
>>> f(1)
1
2
3
4
5
6
^C
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

Recursive Function Calls

Calls by a function to itself

```
>>> def f(x):
...     print(x)
...     f(x+1)
...
>>> f(1)
1
2
3
4
5
6
^C
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

What happened?
The flow of execution



Note: each function call has a local namespace, which contains a private version of x!

Recursive Function Calls

Calls by a function to itself

```
>>> def f(x):
...     print(x)
...     f(x+1)
...
>>> f(1)
1
2
3
4
5
6
^C
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

How to fix it?
Add a stopping criteria.

```
>>> def f(x):
...     if x>5:
...         return
...     print(x)
...     f(x+1)
...
>>> f(1)
1
2
3
4
5
```

WestlakeNLP

Recursive Function Calls

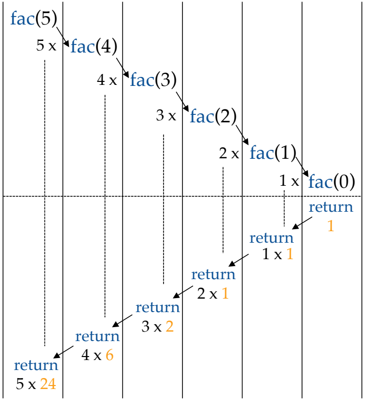
Factorial

fac_rec.py

```python
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



WestlakeNLP

Recursive Function Calls

Factorial $n! = \prod_{i=1}^{n} i$

Recursive Function Calls

Factorial	$n! = \prod_{i=1}^{n} i$
Iteration	$n! = n \times (n-1)!$

WestlakeNLP

Recursive Function Calls

Factorial $n! = \prod_{i=1}^{n} i$

Iteration $n! = n \times (n-1)!$

fac.py

```
# initialization
n = int(input("n="))
s = 1
# loop
i = 1
while i <= n:
    s *= i
    i += 1
# output
print("The factorial of %d is %d"%(n,s))
# (n,s) is tuple parameters
```

WestlakeNLP

Recursive Function Calls

Factorial $n! = \prod_{i=1}^{n} i$

Iteration $n! = n \times (n-1)!$

fac.py

```
# initialization
n = int(input("n="))
s = 1
# loop
i = 1
while i <= n:
    s *= i
    i += 1
# output
print("The factorial of %d is %d"%(n,s))
# (n,s) is tuple parameters
```

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

WestlakeNLP

Recursive Function Calls

Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

WestlakeNLP

Recursive Function Calls

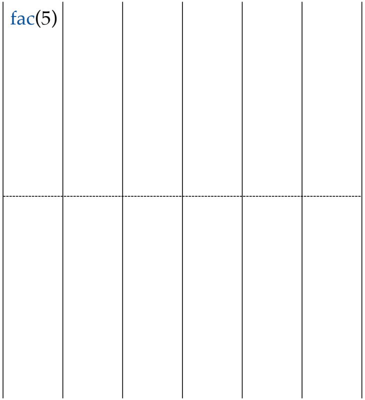
Factorial

fac_rec.py

```python
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues-MacBook-Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues-MacBook-Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



Recursive Function Calls

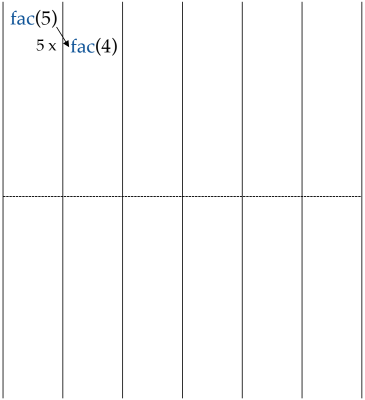
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



Recursive Function Calls

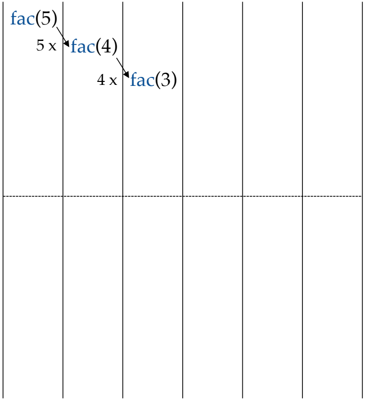
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



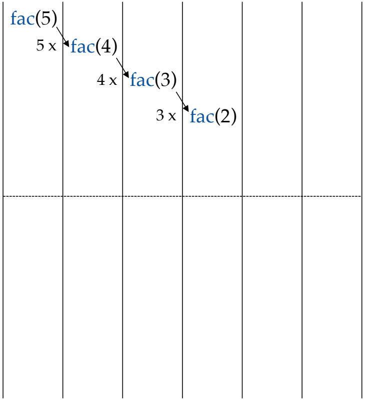
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



Recursive Function Calls

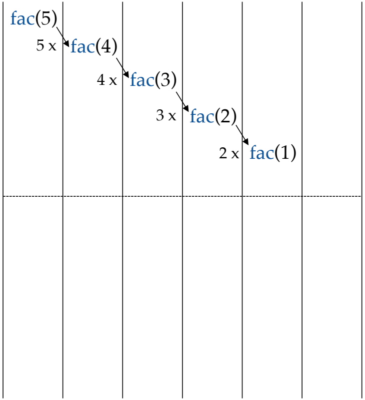
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



WestlakeNLP

Recursive Function Calls

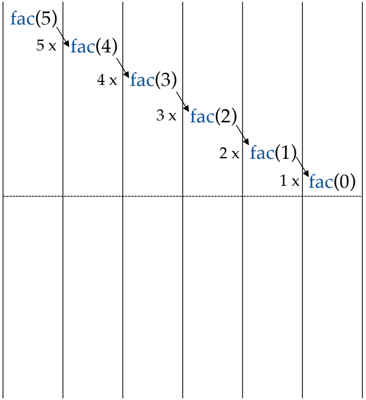
Factorial

fac_rec.py

```python
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



Recursive Function Calls

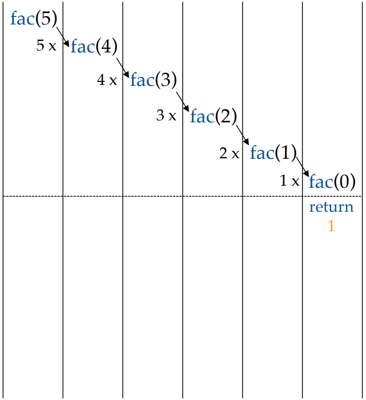
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



WestlakeNLP

Recursive Function Calls

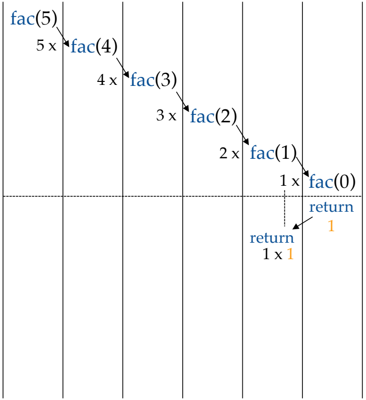
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



WestlakeNLP

Recursive Function Calls

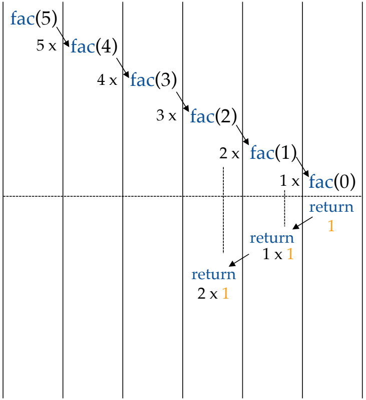
Factorial

fac_rec.py

```python
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues_MacBook_Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues_MacBook_Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



WestlakeNLP

Recursive Function Calls

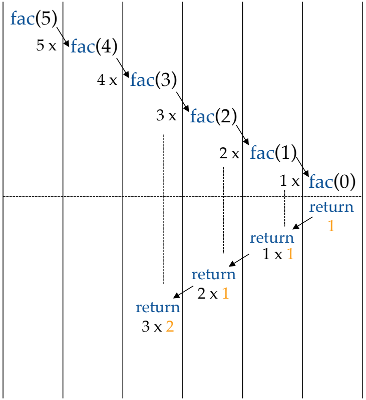
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues_MacBook_Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues_MacBook_Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



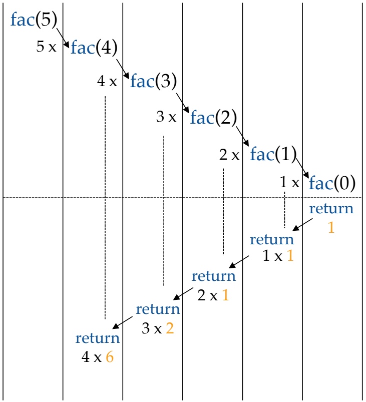
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



Recursive Function Calls

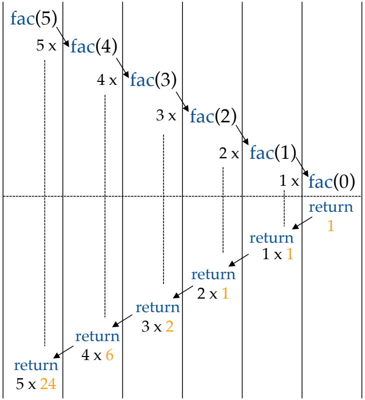
Factorial

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

```
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=10
The factorial of 10 is 3628800
Yues␣MacBook␣Pro:code$ python fac_rec.py
n=5
The factorial of 5 is 120
```

The execution sequence



WestlakeNLP

Recursive Function Calls

Iterative Solution V.S. Recursive Solution

fac.py

```
# initialization
n = int(input("n="))
s = 1
# loop
i = 1
while i <= n:
    s *= i
    i += 1
# output
print("The factorial of %d is %d"%(n,s))
# (n,s) is tuple parameters
```

fac_rec.py

```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
# input
n = int(input("n="))
# output
print("The factorial of %d is %d"%(n,fac(n)))
# (n,s) is tuple parameters
```

- Both based on incremental calculation $n! = n \times (n-1)!$
- Iterative solution starts from the first case.
- Recursive solution starts from the boundary case.
- Making use of function calls in the incremental equation.

Recursive Function Calls

- Recursive solution can be more readable.
- Must pay attention to the boundary case.

Fibonacci — call twice

$$f_0 = 1, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}$$

fib_iter.py

```
# initialization
n = int(input("Input the index of n="))
x1 = 1               # f_{i-2}
x2 = 1               # f_{i-1}
# iteration
i = 2
while i <= n:
    x = x1 + x2      # f_i
    x2, x1= x1, x    # tuple assignment
    i += 1
print(n,"- fibonacci:", x)
```

Recursive Function Calls

Fibonacci — call twice

$$f_0 = 1, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}$$

fib_iter.py

```
# initialization
n = int(input("Input the index of n="))
x1 = 1              # f_{i-2}
x2 = 1              # f_{i-1}
# iteration
i = 2
while i <= n:
    x = x1 + x2     # f_i
    x2, x1= x1, x   # tuple assignment
    i += 1
print(n,"- fibonacci:", x)
```

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

WestlakeNLP

Recursive Function Calls

Fibonacci — call twice

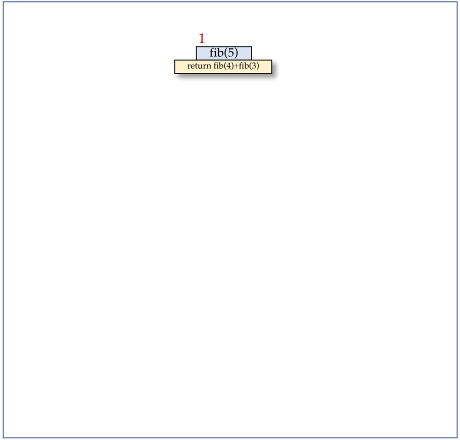
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

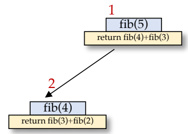
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

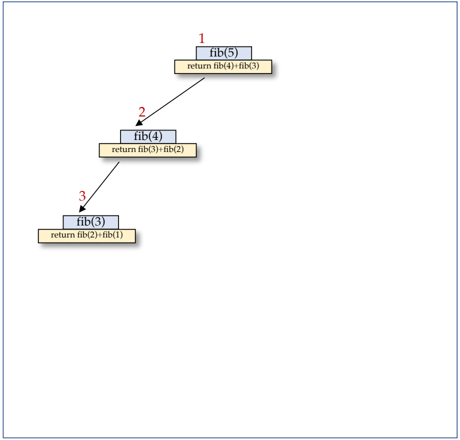
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Fibonacci — call twice

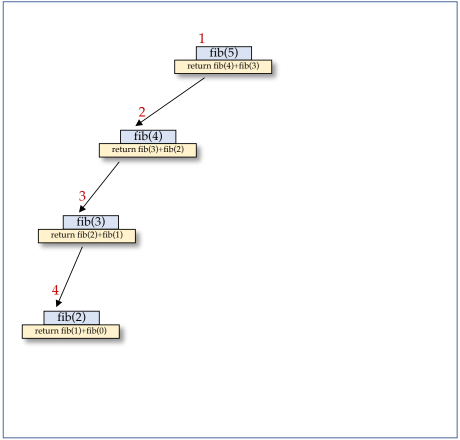
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

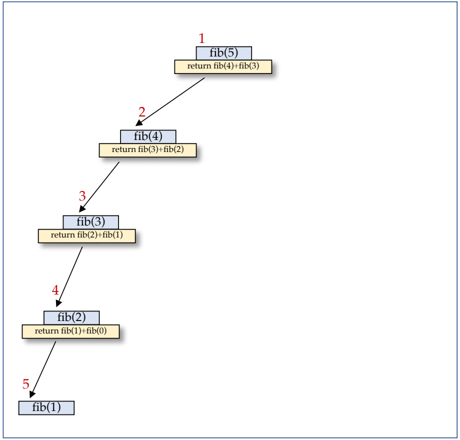
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

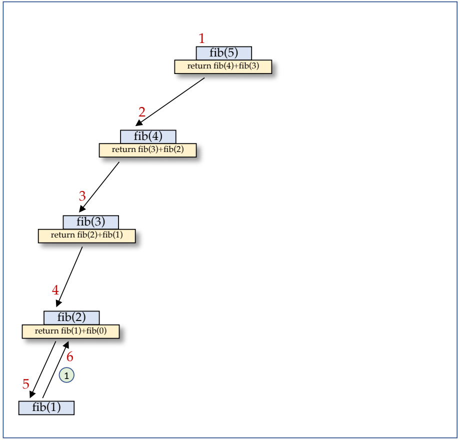
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

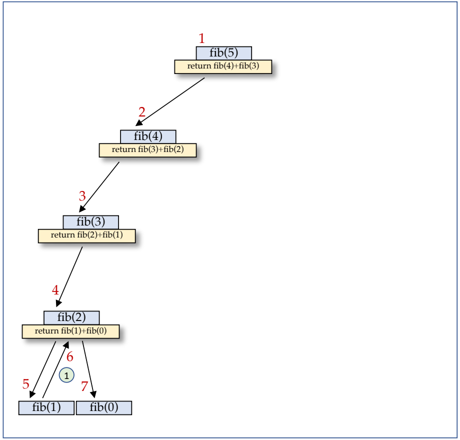
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

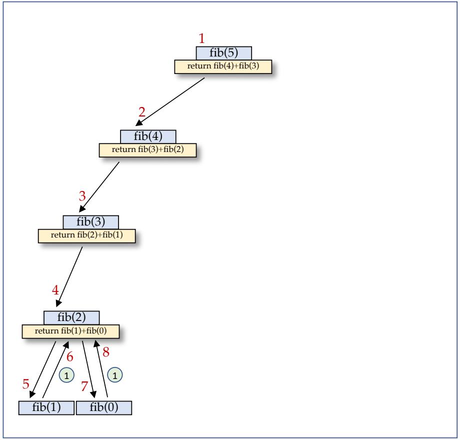
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

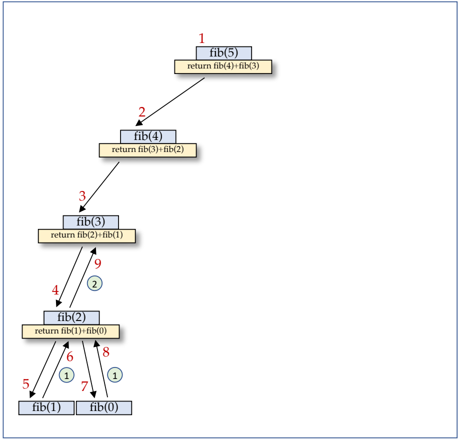
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

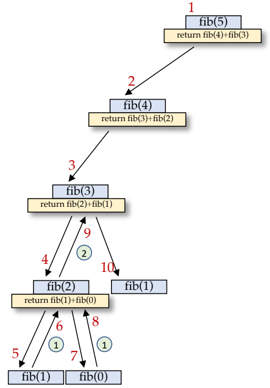
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

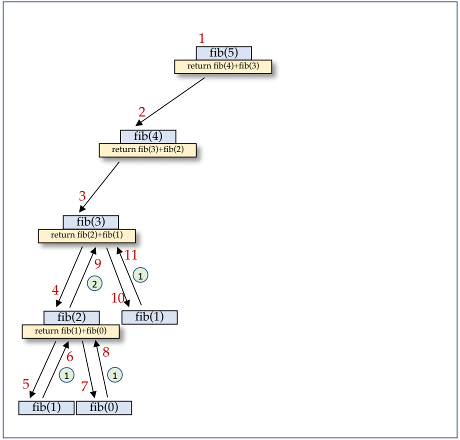
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

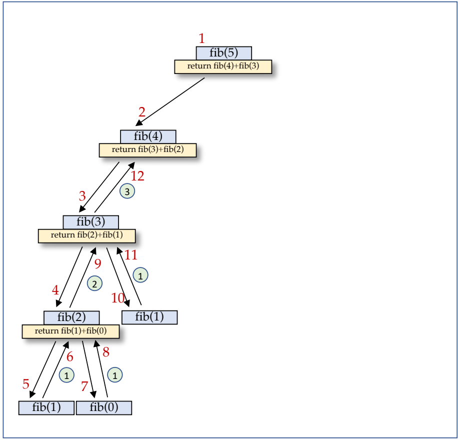
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

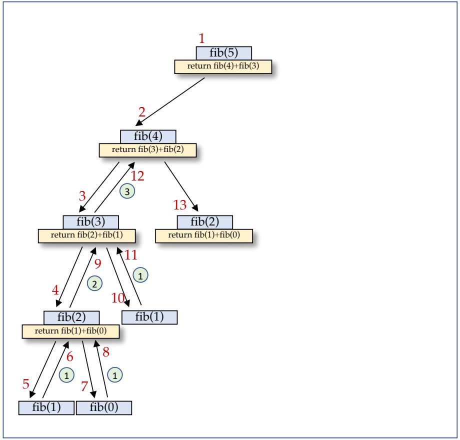
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

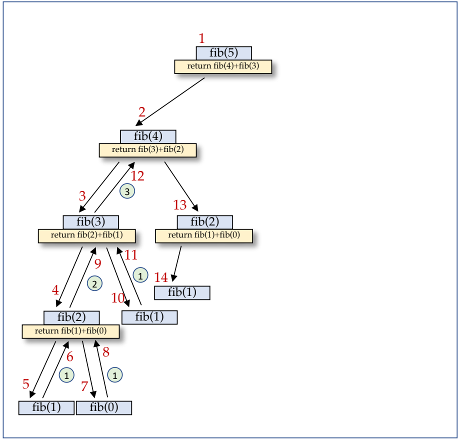
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

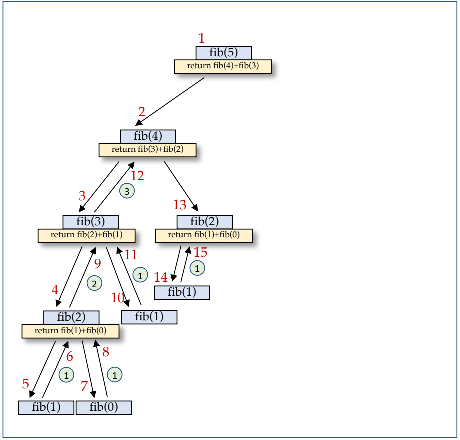
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

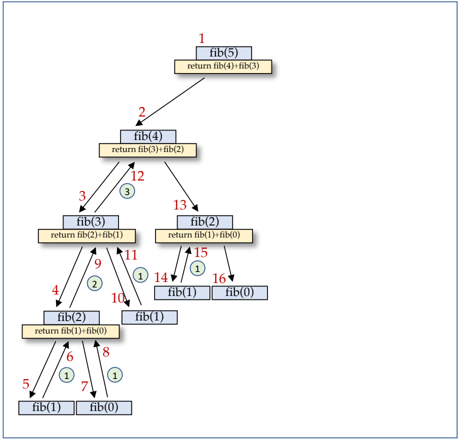
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

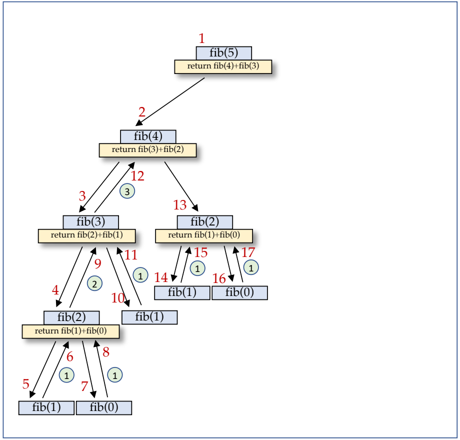
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

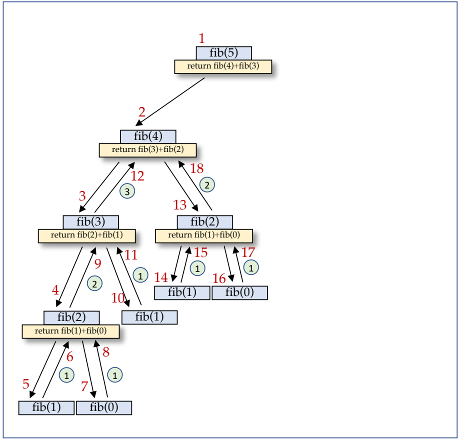
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

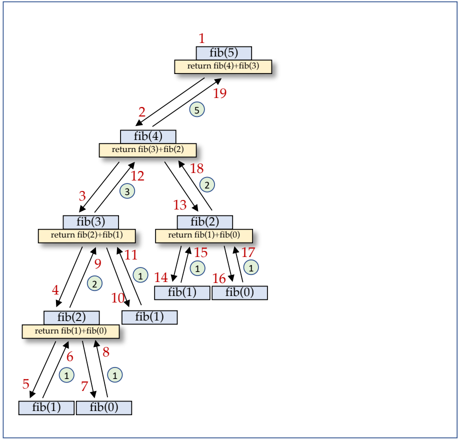
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Fibonacci — call twice

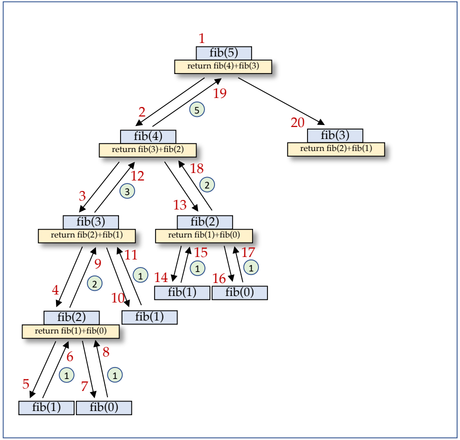
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

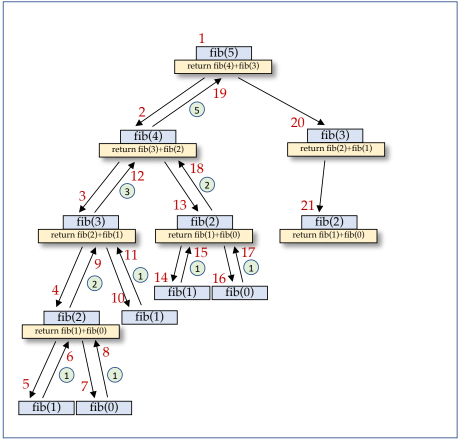
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

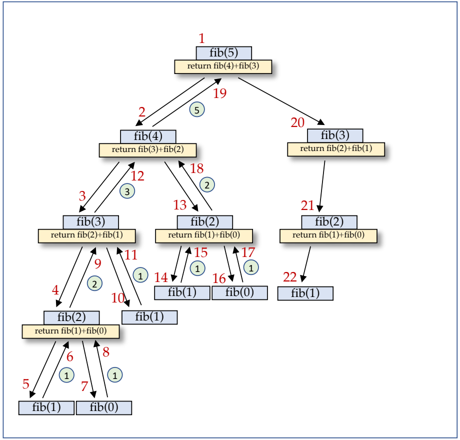
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues MacBook Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

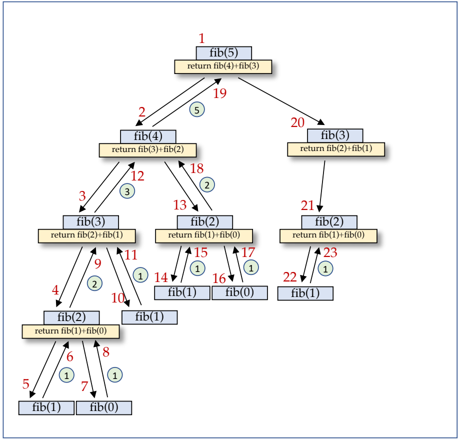
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

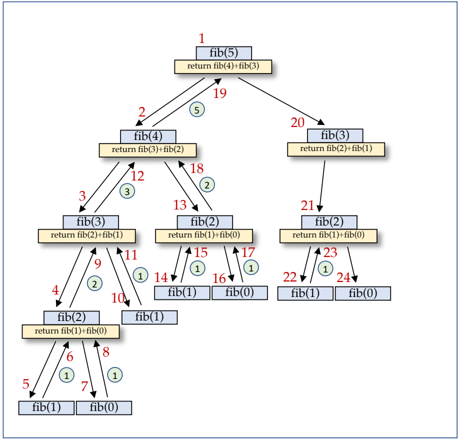
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Fibonacci — call twice

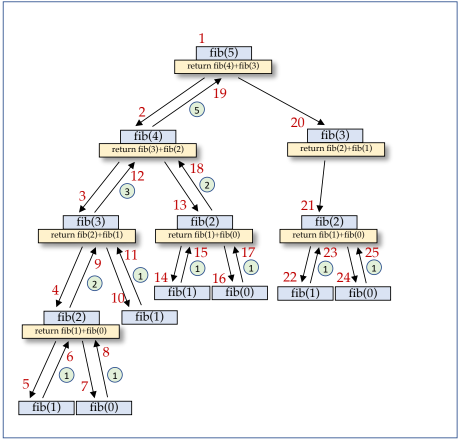
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

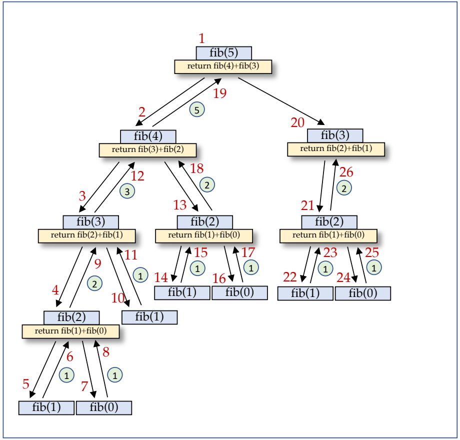
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

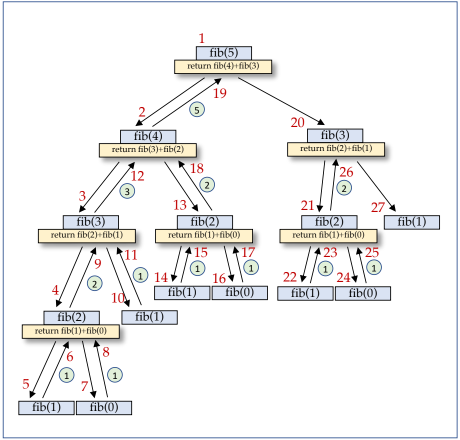
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues-MacBook-Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

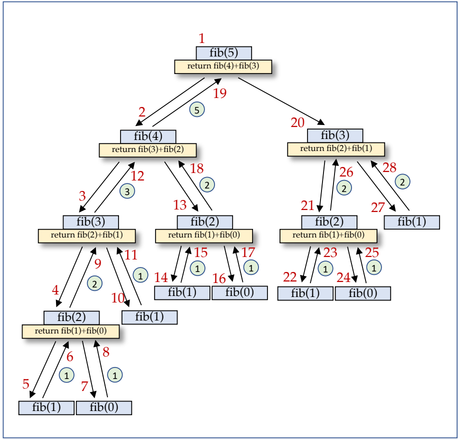
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Fibonacci — call twice

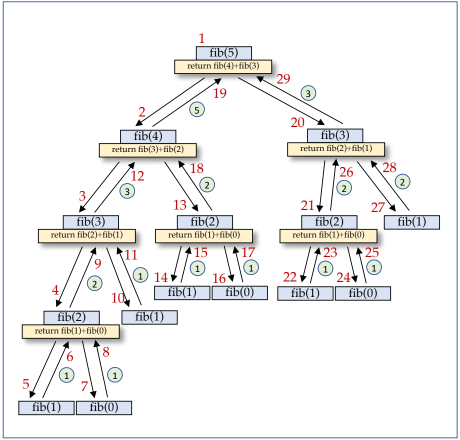
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Fibonacci — call twice

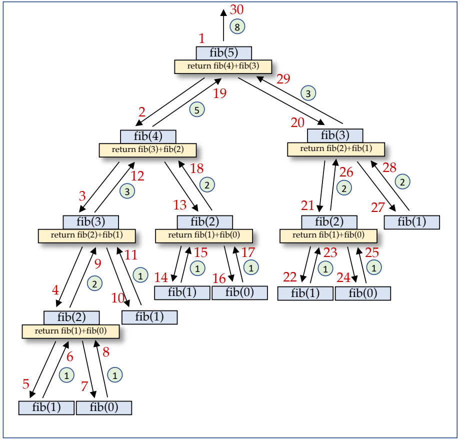
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

The execution sequence



Recursive Function Calls

Fibonacci — call twice

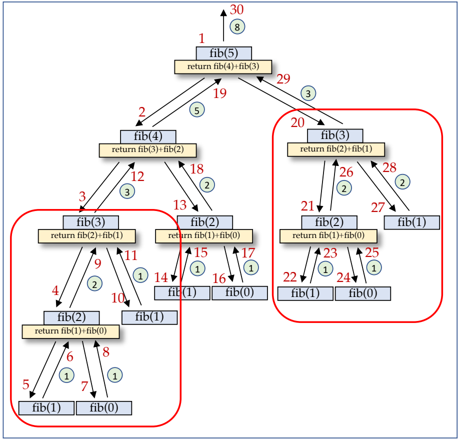
$$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$$

fib_rec.py

```python
def fib(n):
    # boundary case
    if n==0 or n==1:
        return 1
    # recursion
    return fib(n-1) + fib(n-2)
# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

```
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=5
5 - fibonacci: 8
Yues␣MacBook␣Pro:code$ python fib_rec.py
Input the index of n=10
10 - fibonacci: 89
```

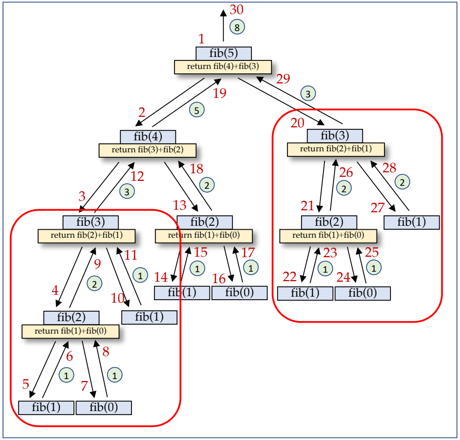
The execution sequence

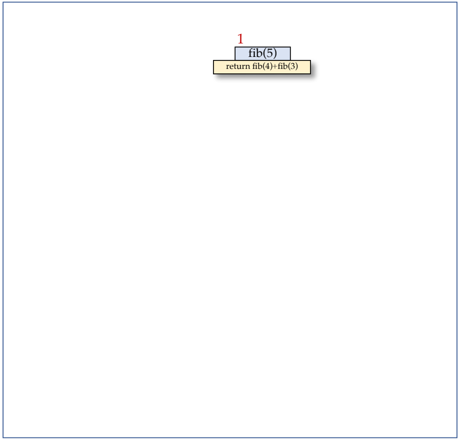


Did you find the waste of computation?

Recursive Function Calls

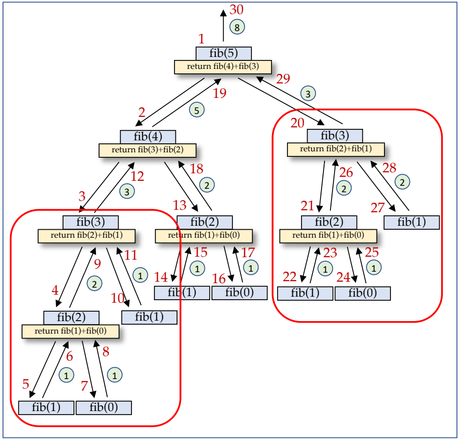
Caching – save computed results

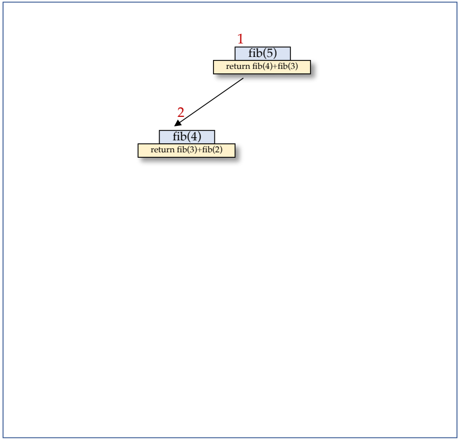




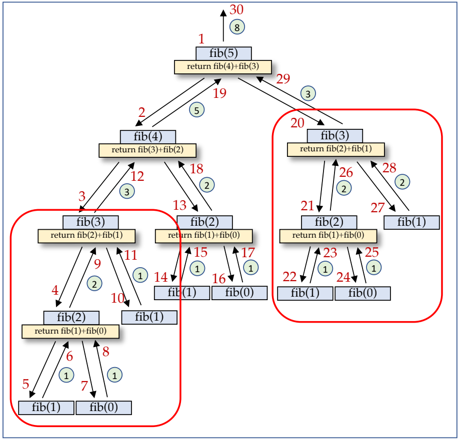
Recursive Function Calls

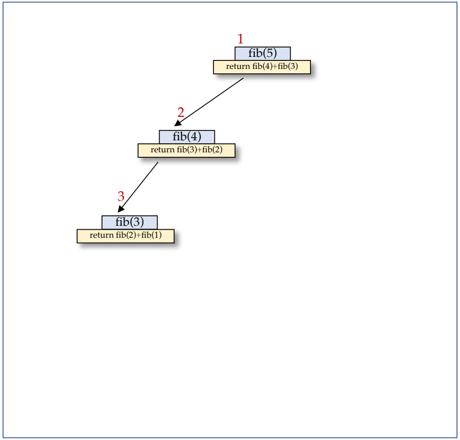
Caching – save computed results





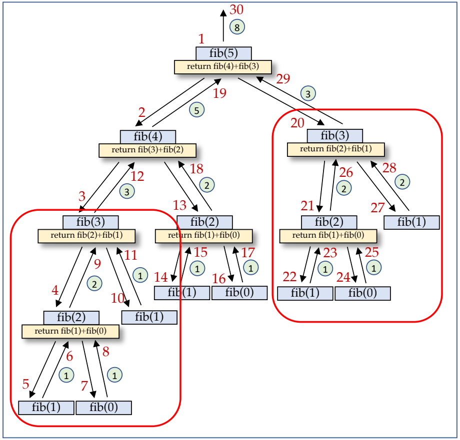
Caching – save computed results

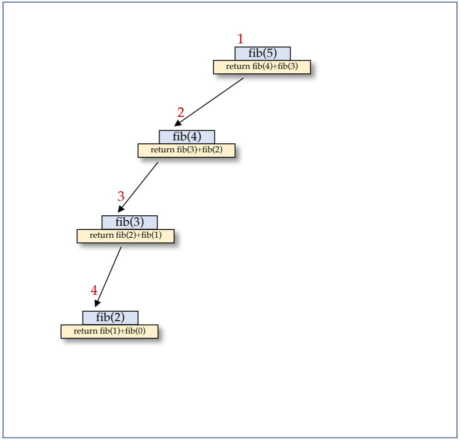




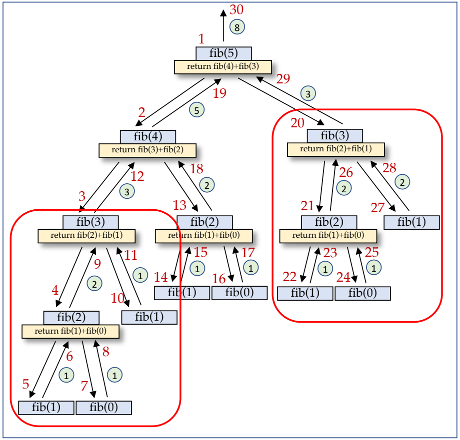
Recursive Function Calls

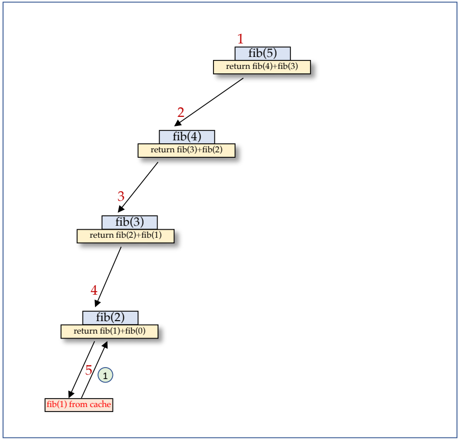
Caching – save computed results



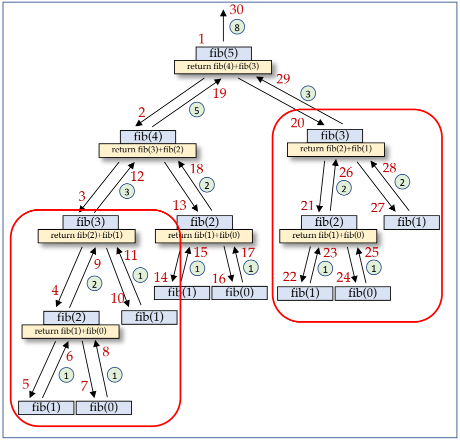


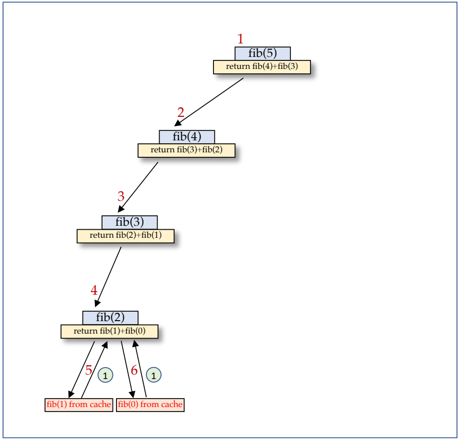
Caching – save computed results





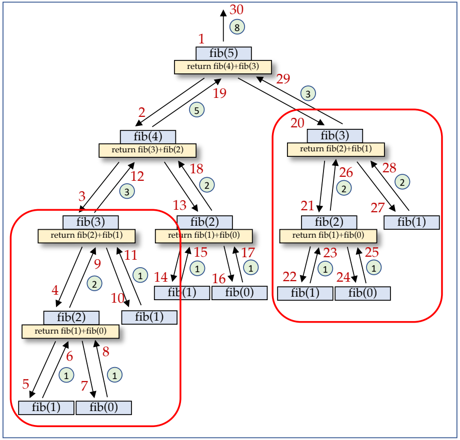
Caching – save computed results

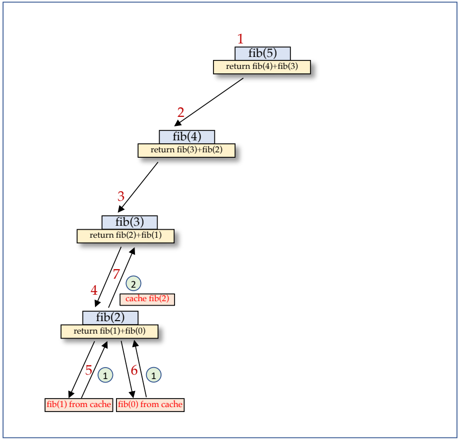




Recursive Function Calls

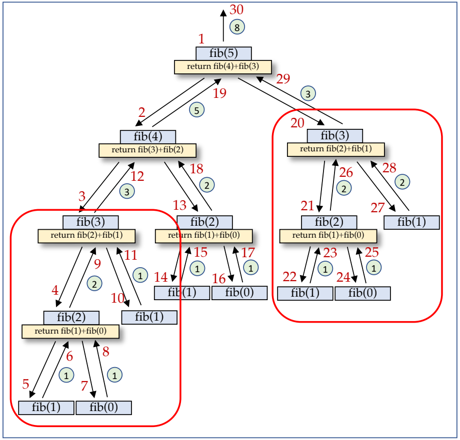
Caching – save computed results

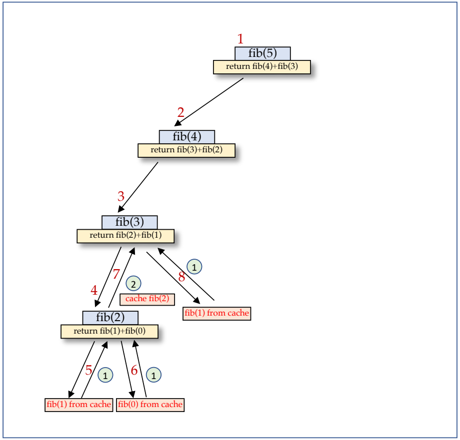




Recursive Function Calls

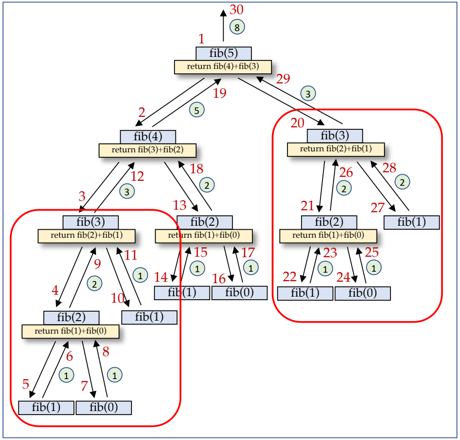
Caching – save computed results

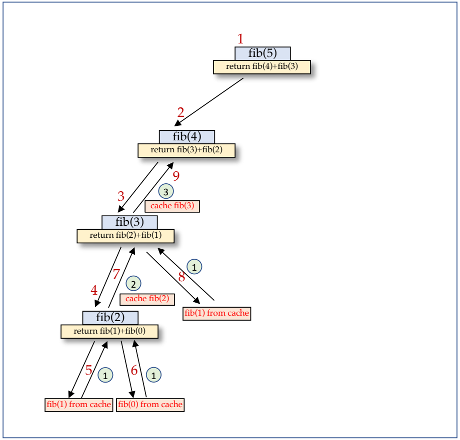




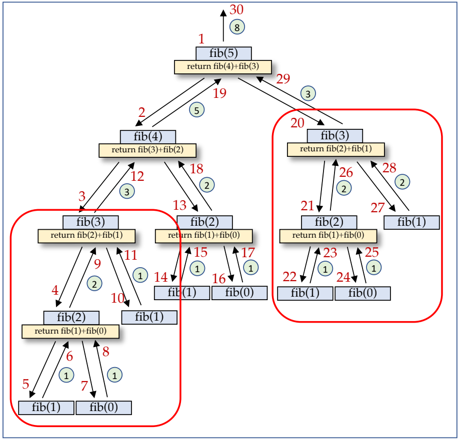
Recursive Function Calls

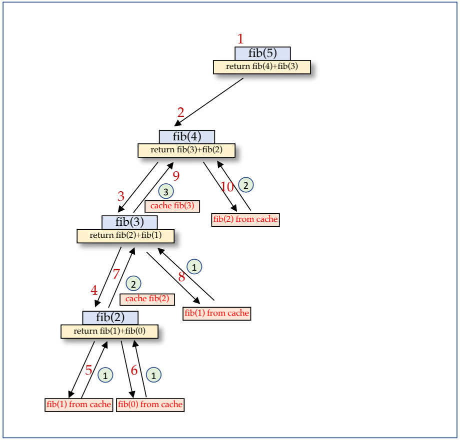
Caching – save computed results





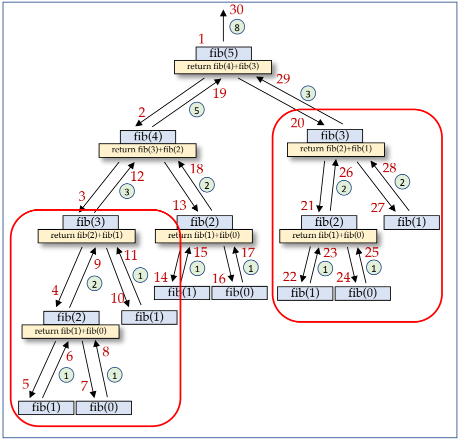
Caching – save computed results

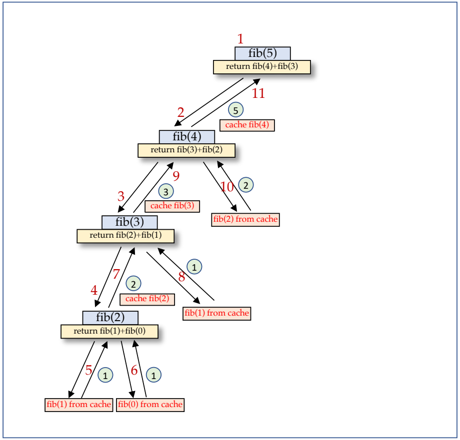




Recursive Function Calls

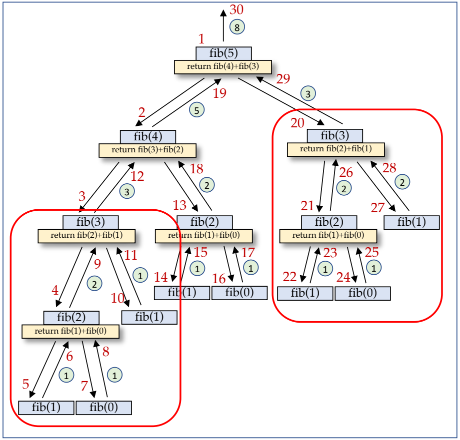
Caching – save computed results

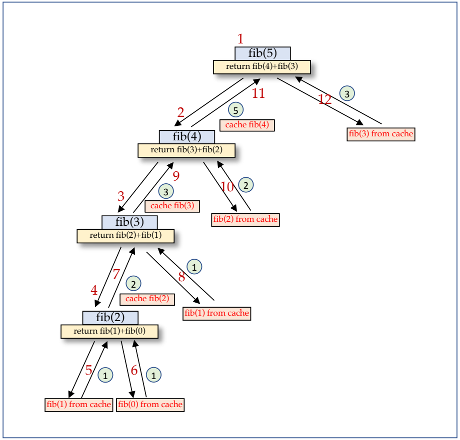




Recursive Function Calls

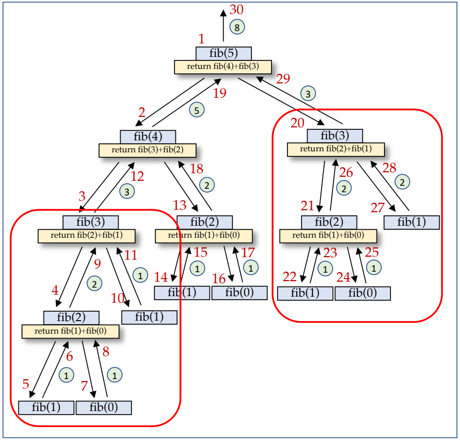
Caching – save computed results

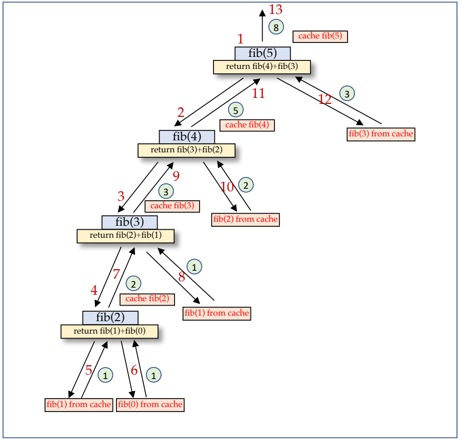




Recursive Function Calls

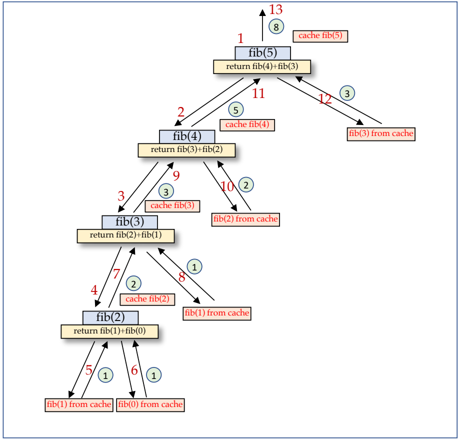
Caching – save computed results





Recursive Function Calls

Caching – save computed results



fib_rec_cache.py

```python
cache = (1,1)
def fib(n):
    global cache
    if n < len(cache):
        return cache[n]

    f_n_1 = fib(n-1)
    assert n == len(cache)

    f = f_n_1 + cache[n-2]
    cache += (f, )
    print(cache)
    return f

# input
n = int(input("Input the index of n="))
#output
print(str(n),"- fibonacci:", fib(n))
```

Why assert n == len(cache)?

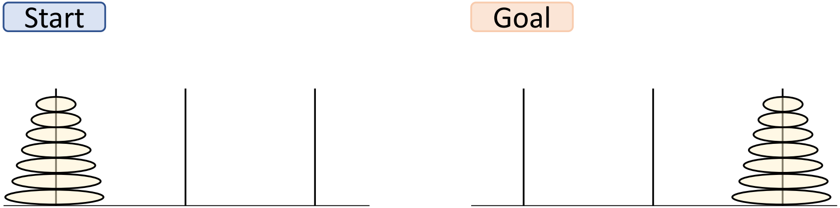
Recursive Function Calls

- Three different ways to compute f_n:
  - Iterative
  - Recursive
  - Recursive with cache
- Each is one *algorithm*.
- Algorithms study how to automatically compute something.
- Different algorithms to the same problem can vary in speed, memory cost, etc.

WestlakeNLP

Recursive Function Calls

Tower of Hanoi



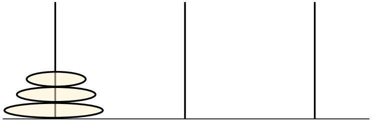
- Rules
  - Each time a disk can be moved from one rod to another.
  - Only the top-disk on a rod can be moved.
  - A disk cannot be placed on a smaller disk.

WestlakeNLP

Recursive Function Calls

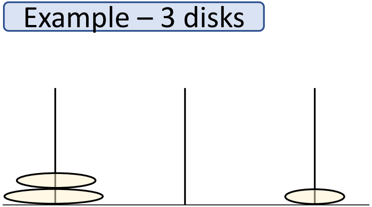
Tower of Hanoi

Example – 3 disks



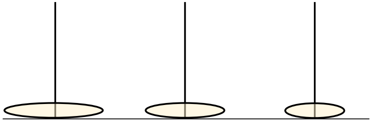
WestlakeNLP

Tower of Hanoi



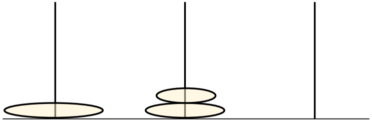
Tower of Hanoi

Example – 3 disks



Tower of Hanoi

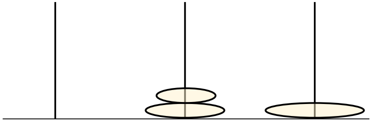
Example – 3 disks



Recursive Function Calls

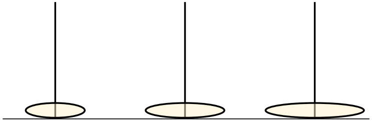
Tower of Hanoi

Example – 3 disks



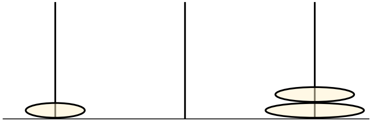
Tower of Hanoi

Example – 3 disks



Tower of Hanoi

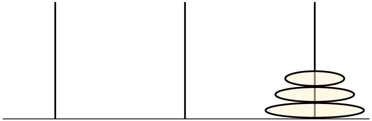
Example – 3 disks



Recursive Function Calls

Tower of Hanoi

Example – 3 disks



WestlakeNLP

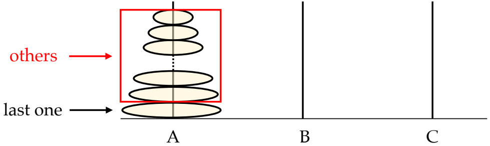
Recursive Function Calls

Tower of Hanoi

- What kind of recursive rules did you find?

Tower of Hanoi

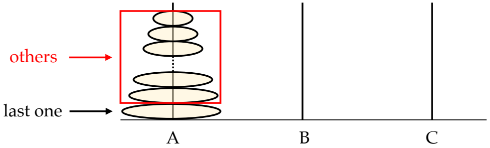
- What kind of recursive rules did you find?
  1. move the others from A to B
  2. move the last one from A to C
  3. move the others from B to C



Recursive Function Calls

Tower of Hanoi

- What kind of recursive rules did you find?
  1. move the others from A to B
  2. move the last one from A to C
  3. move the others from B to C

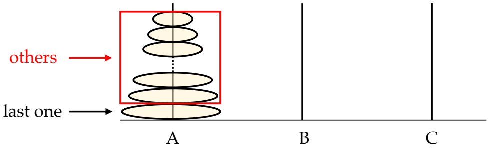


- When we move the others, we can safely ignore the last one, since every disk is smaller than the last disk.

WestlakeNLP

Recursive Function Calls

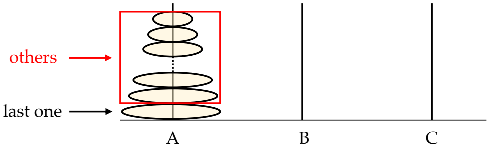
Tower of Hanoi



- Formal specification of recursion to solve $hanoi(n, A \to C)$
  1. solve $hanoi(n-1, A \to B)$
  2. the last one move $A \to C$
  3. solve $hanoi(n-1, B \to C)$

Recursive Function Calls

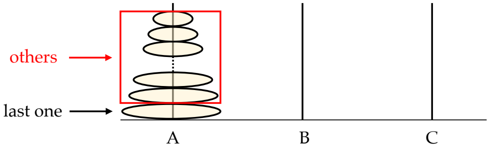
Tower of Hanoi



- Formal specification of recursion to solve $hanoi(n, A \rightarrow C)$
  1. solve $hanoi(n-1, A \rightarrow B)$
  2. the last one move $A \rightarrow C$
  3. solve $hanoi(n-1, B \rightarrow C)$
- Boundary case? – n==1 must be the smallest, move directly.

Recursive Function Calls

Tower of Hanoi



- Formal specification of recursion to solve $hanoi(n, A \rightarrow C)$
  1. solve $hanoi(n-1, A \rightarrow B)$
  2. the last one move $A \rightarrow C$
  3. solve $hanoi(n-1, B \rightarrow C)$
- Boundary case? – n==1 must be the smallest, move directly.
- no need to maintain the disk states, but only print moves in order.

Recursive Function Calls

Tower of Hanoi

hanoi.py

```
def hanoi(n, source="A", target="C", other="B"):
    if n == 1:
        print("Move the top disk(#%d) from %s to %s"%(n, source, target))
    else:
        hanoi(n-1, source, other, target)
        print("Move the top disk(#%d) from %s to %s"%(n, source, target))
        hanoi(n-1, other, target, source)
n = int(input("n="))
hanoi(n)
```

WestlakeNLP

Recursive Function Calls

Tower of Hanoi

```
Yues␣MacBook␣Pro:code$ python hanoi.py
n=3
Move the top disk(#1) from A to C
Move the top disk(#2) from A to B
Move the top disk(#1) from C to B
Move the top disk(#3) from A to C
Move the top disk(#1) from B to A
Move the top disk(#2) from B to C
Move the top disk(#1) from A to C
```

WestlakeNLP

Recursive Function Calls

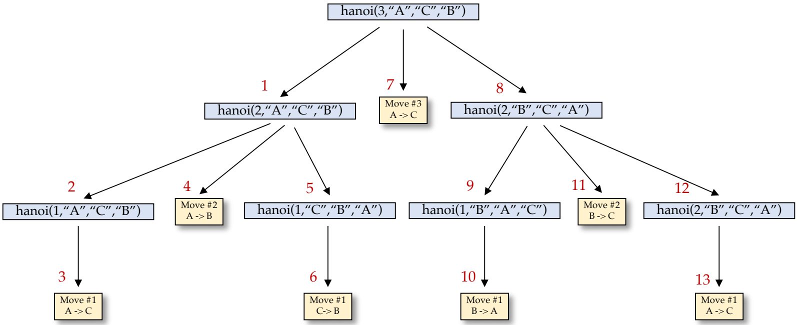
Tower of Hanoi

```
Yues␣MacBook␣Pro:code$ python hanoi.py
n=4
Move the top disk(#1) from A to B
Move the top disk(#2) from A to C
Move the top disk(#1) from B to C
Move the top disk(#3) from A to B
Move the top disk(#1) from C to A
Move the top disk(#2) from C to B
Move the top disk(#1) from A to B
Move the top disk(#4) from A to C
Move the top disk(#1) from B to C
Move the top disk(#2) from B to A
Move the top disk(#1) from C to A
Move the top disk(#3) from B to C
Move the top disk(#1) from A to B
Move the top disk(#2) from A to C
Move the top disk(#1) from B to C
```

WestlakeNLP

Recursive Function Calls

Tower of Hanoi



WestlakeNLP

This week check-off:
Function Exercises

WestlakeNLP