COMP20230: Data Structures & Algorithms **2018 – 2019 Semester 2**

Lecture 6: More Recursion

Lecturer: Dr. Andrew Hince *Scribe: Meihan Xue, Yitong Sun*

6.1 Recursion Review

6.1.1 The Recursion and Base Case

"Recursion is a way of decomposing problems into smaller, simpler sub-tasks that are similar to the original."
Recursion is the function which calls itself. There are two points need to be noticed: (1) when using recursion, we should be aware that recursion calls itself in the function; (2) requires a base case (a case simple enough to solve without recursion) and a stop condition to end the recursion.

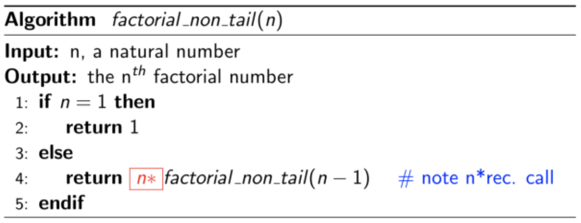
6.1.2 Call stack

Call stack is a stack data structure. It stores information about the active subroutines of a computer program. For recursion, every call has its own memory address, owns values for parameter and variables.

6.2 Tail Recursion and Non-tail Recursion

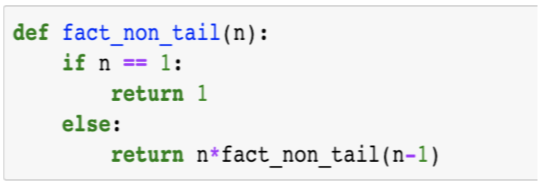
6.2.1 Non-tail recursion

If we want to calculate the factorial of 5, in theory it could be written by 5*4! then 4! = 4*3! and so on until 1!=1. According to this idea, we could write the code like blowing picture.



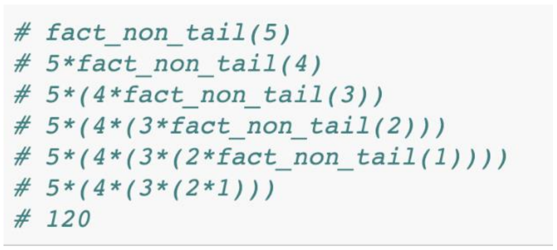
```
Algorithm factorial_non_tail(n)
Input: n, a natural number
Output: the n^th factorial number
1: if $n = 1$ then
2:     return 1
3: else
4:     return $\boxed{n*}$ factorial_non_tail($n-1$)     # note n*rec. call
5: endif
```

Non-tail Recursive Functions Pseudocode



```python
def fact_non_tail(n):
    if n == 1:
        return 1
    else:
        return n*fact_non_tail(n-1)
```

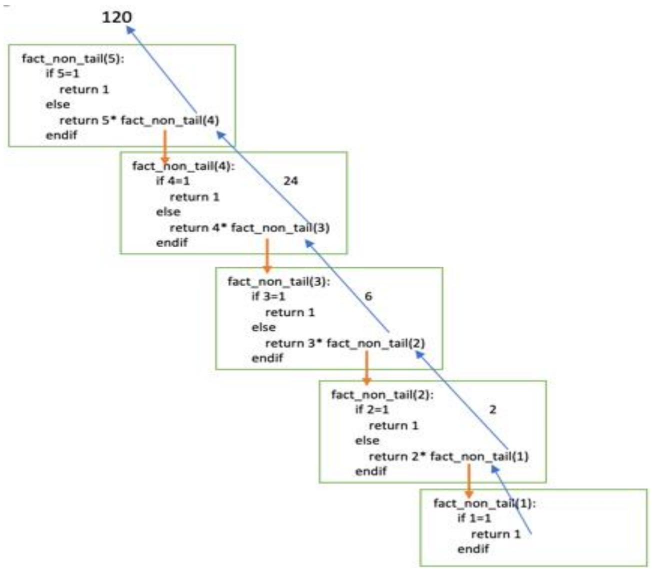
Python Code and Sample Run



```
# fact_non_tail(5)
# 5*fact_non_tail(4)
# 5*(4*fact_non_tail(3))
# 5*(4*(3*fact_non_tail(2)))
# 5*(4*(3*(2*fact_non_tail(1))))
# 5*(4*(3*(2*1)))
# 120
```

Executing Fact

Below is a diagram to simulate it and understand it more directly:



```
fact_non_tail(5):
  if 5=1
    return 1
  else
    return 5* fact_non_tail(4)
  endif

fact_non_tail(4):
  if 4=1
    return 1
  else
    return 4* fact_non_tail(3)
  endif

fact_non_tail(3):
  if 3=1
    return 1
  else
    return 3* fact_non_tail(2)
  endif

fact_non_tail(2):
  if 2=1
    return 1
  else
    return 2* fact_non_tail(1)
  endif

fact_non_tail(1):
  if 1=1
    return 1
  endif
```

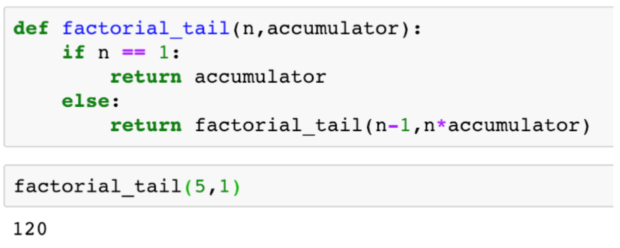
Process Simulating

From the pictures we could notice that in the function, it could call itself. Non-tail recursion is easy to understand and logical.

However, when the non-tail recursion is used the overflow should be considered. When I used the code to calculate the factorial of 3000, there will be an error.

6.2.2 Tail recursion

A tail recursion is a recursive function where the function calls itself at the end (tail) of the function in which no computation is done after the return of recursive call. In other words, non-tail recursion always returns an evaluation expression. For instance, if we want to calculate the factorial (the name of the function is fact(n)), normally it returns n*fact (n-1). However, the tail-recursion function returns tail fact (n-1, accumulate).
Tail recursion is a kind of programming skill. After the function returns, there is nothing to do except its value.

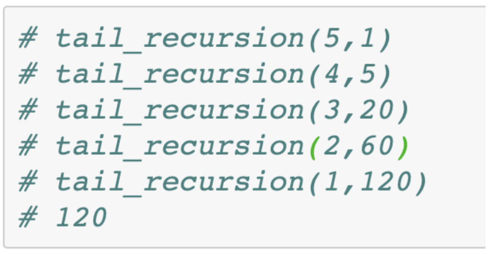


```python
def factorial_tail(n,accumulator):
    if n == 1:
        return accumulator
    else:
        return factorial_tail(n-1,n*accumulator)
```

```python
factorial_tail(5,1)
```

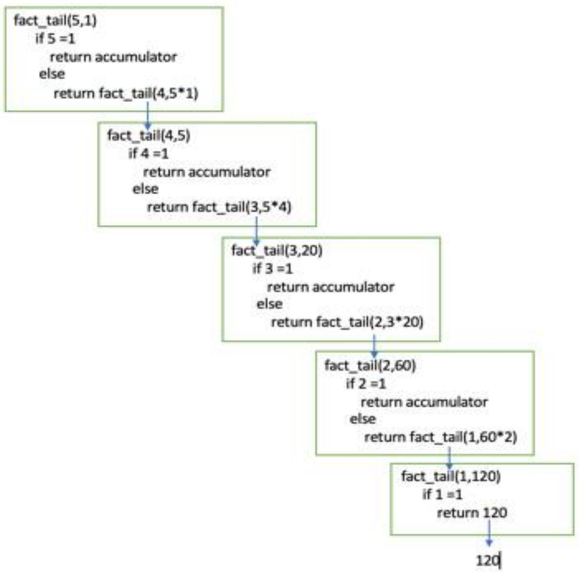
```
120
```

Python Code and Sample Run



```python
# tail_recursion(5,1)
# tail_recursion(4,5)
# tail_recursion(3,20)
# tail_recursion(2,60)
# tail_recursion(1,120)
# 120
```

Executing Fact

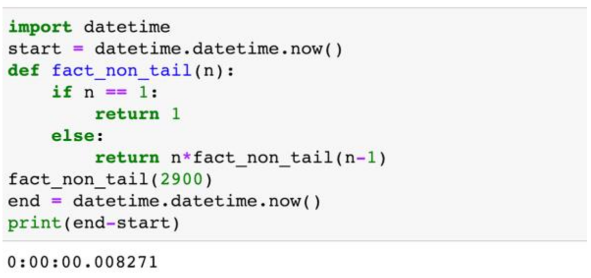


Process Simulating

Comparing with the non-tail recursion, a number ‘n’ is needed to control the recursion (from n to 1), and when the n==1 is false, the function does not return a math expression, instead it uses two numbers to call the function. Form the process simulating we could see directly that when the n is 1, we do not go back anymore, the function could return the result directly.

6.3 Evaluation

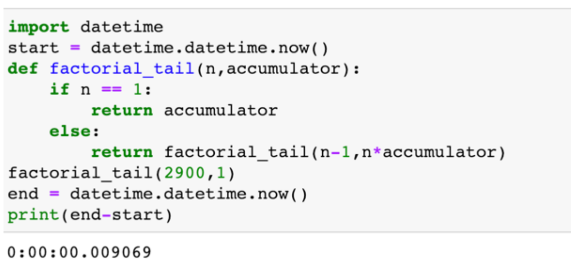
The idea of tail recursion is removing the result from lower layer to upper layer again. Then the upper layer could continue the calculation and get the result. If we look at the example carefully, we can see that the result of each recursion is stored in the second parameter “accumulate”. In the last calculation, only one value will be returned. Because of the recursion’s principle, it still has to be returned to the upper layer, but no calculation any more. The advantage is that the memory allocated each time will not be expanded.
When we think about the running time of these two recursions, there is no big difference.



```
import datetime
start = datetime.datetime.now()
def fact_non_tail(n):
    if n == 1:
        return 1
    else:
        return n*fact_non_tail(n-1)
fact_non_tail(2900)
end = datetime.datetime.now()
print(end-start)
```

```
0:00:00.008271
```

Non-tail Recursion



```
import datetime
start = datetime.datetime.now()
def factorial_tail(n,accumulator):
    if n == 1:
        return accumulator
    else:
        return factorial_tail(n-1,n*accumulator)
factorial_tail(2900,1)
end = datetime.datetime.now()
print(end-start)
```

```
0:00:00.009069
```

Tail Recursion

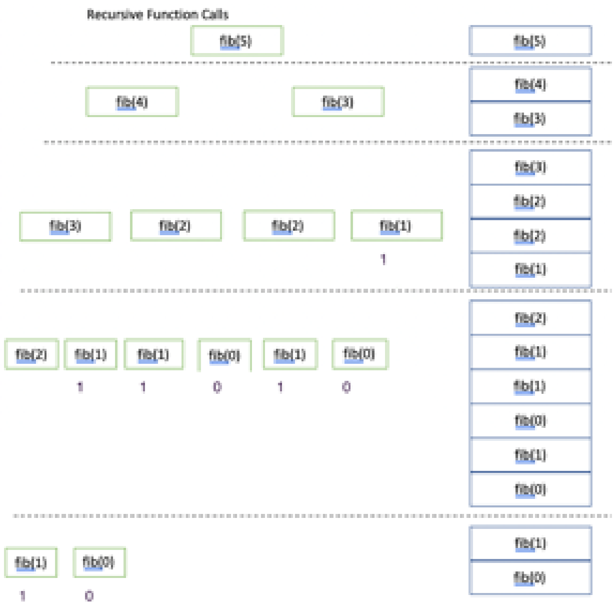
Almost all computer programs rely on the call stack. We look at the difference between tail recursion and non-tail recursion from the perspective of the call stack.

- Non-tail recursion

  Here we use the Fibonacci sequence as the example to explain how tail and non-tail recursion call stack.

```
# def fib(n):
#     if n = 0 or n = 1:
#         return n
#     else return fib(n-1)+fib(n-2)
```

Pseudocode



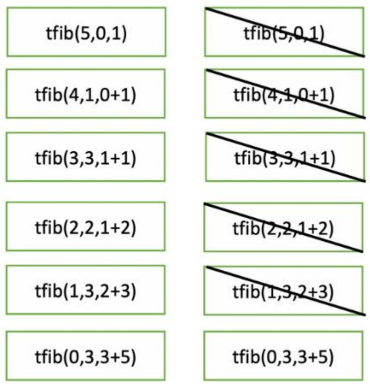
Call Stack and Data Structure

We can see from the flowchart that when we return fib(4) and fib(3), there are two new addresses added to the stack. This procedure repeats till fib(1) and fib(0) appears. From a graphically view, it is a tree-like structure.
That is, in a recursive program each level of recursion call must add an address to the call stack, so if the program has infinite recursion (or just too many recursive levels), the call stack will generate a stack overflow.

- Tail recursion

```
# def tfib(n,a,b):
#     if n = 0:
#         return a
#     else:
#         return tfib(n-1,b,a+b)
# a,b are the first two numbers of Fibonacci,which are 0 and 1.
```

Pseudocode



Call Stack

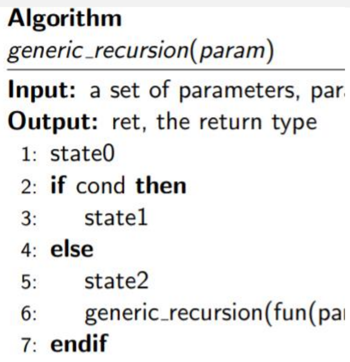
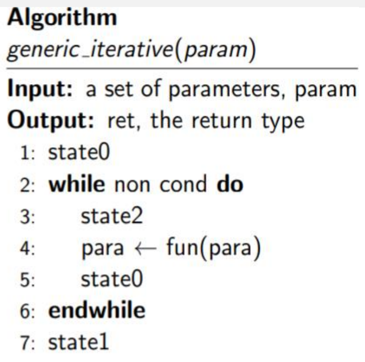
According to the picture above, when we want to know the 5th number of Fibonacci sequence we can use input (5,0,1) to call the function, where 0 and 1 are the first two numbers of Fibonacci sequence. Because 5=0 is false and it will return function call tfib(4,1,0+1) . At this

time, tfib(5,0,1) is executed already and as a result the last function call no longer in the stack frame.
Theoretically, overflow will not happen in the tail recursion, but Python does not optimize the tail loop, so there will still be overflows.

6.4 Recursion and Iterative

6.4.1 Comparison between recursive algorithm and iterative algorithm

Recursion is a process which always applied to a function. The iteration is applied to set of instructions which will be executed repeatedly. Based on the variation of situations, some cases are naturally recursive and some are suitable with iteration.

COMPARISION BASIS	RECURSION	ITERATION
NATURE	Recursion statement calls the function itself.	Repeatably execution of a block of instructions.
FORMAT	 **Algorithm** *generic_recursion(param)* **Input:** a set of parameters, par **Output:** ret, the return type 1: state0 2: **if** cond **then** 3: state1 4: **else** 5: state2 6: generic_recursion(fun(pa 7: **endif**	 **Algorithm** *generic_iterative(param)* **Input:** a set of parameters, param **Output:** ret, the return type 1: state0 2: **while** non cond **do** 3: state2 4: para ← fun(para) 5: state0 6: **endwhile** 7: state1
TERMINATION	A conditional statement is included in the body of the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached
INFINITE REPETITION	Both can have an infinite repetition if the condition is not satisfied or always satisfied. Infinite recursion can crash the kernel, and infinite loop occupies the CPU cycles repeatedly.	
STACK	Every time the function is called, the stack is used to store the local variable. Stack overflow is a problem when process a large input size.	Does not use stack to store variable.
SPEED	Slow in execution.	Fast in execution.
CODE	Recursion reduces the size of the code.	Iteration algorithm often has a bigger size of code.

6.4.2 Recursion optimization

The mechanism of recursion function is easy to understand, and the code is relatively easy to write. However, because of the problematic issues leading by big input size, which is stack overflow, it is better to have iterative algorithms when we are not sure about the input size. And in most cases, it is convertible between the two forms of functions.

Before we convert a recursive function to iterative form, it is necessary to come up with a tail recursion firstly and then use the transforming formula below. And we often call this procedure as recursion optimization.

```
Algorithm
generic_recursion(param)
Input: a set of parameters, param
Output: ret, the return type
state0
if cond then
    state1
else
    state2
    generic_recursion(fun(para))
endif
```

```
Algorithm
generic_iterative(param)
Input: a set of parameters, param
Output: ret, the return type
state0
while non cond do
    state2
    para ← fun(para)
    state0
endwhile
state1
```

Conversion Formula

Alternatively, the overall procedure can be separated into the following steps:

i. Use the not condition as new loop condition;
ii. Take state2 (if exists) as a loop instruction;
iii. Reassign of the recursion parameters;
iv. If there was a state0, put it in the loop and then end loop;
v. Return the state1.

```python
def fact_recursion(n, r = 1):
    if n <= 0:
        return 1 * r
    else:
        return fact_recursion(n - 1, r * n)
```

➡

```python
def fact_iterative(n, r = 1):
    while n > 0:
        r = r * n
        n = n - 1
    return r
```

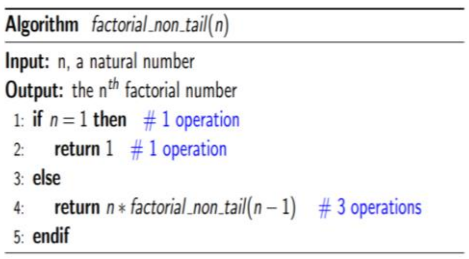
Conversion Example

6.5 Complexity Analysis

To analyze the complexity if a recursive algorithm, we cannot apply the same mechanism used by iterative algorithms. There are two things we have to consider specifically.

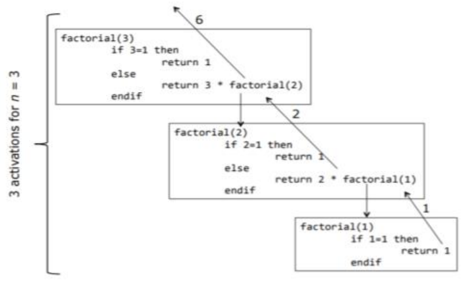
- Number of basic operations in each activation of the recursion
- Number of activations of one recursive function call.

Activations are the amount of stack frame used by the initial function call of a recursive algorithm. Take the factorial function as an example, for input n=3, there should be three activations.



```
Algorithm  factorial_non_tail(n)
Input: n, a natural number
Output: the n^th factorial number
1: if n = 1 then   # 1 operation
2:     return 1   # 1 operation
3: else
4:     return n * factorial_non_tail(n − 1)   # 3 operations
5: endif
```

Basic Operations



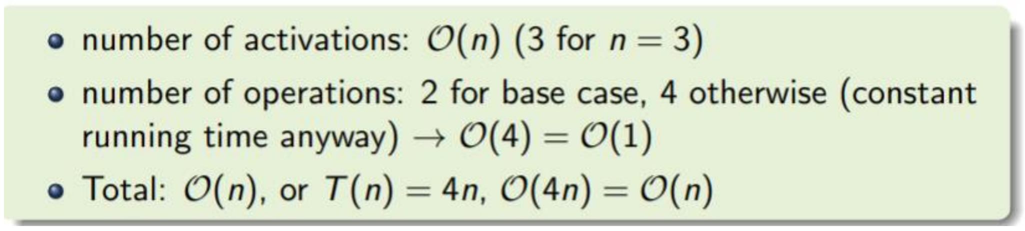
```
factorial(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial(2)
        endif

factorial(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial(1)
        endif

factorial(1)
        if 1=1 then
                return 1
        endif
```

Activations for n=3

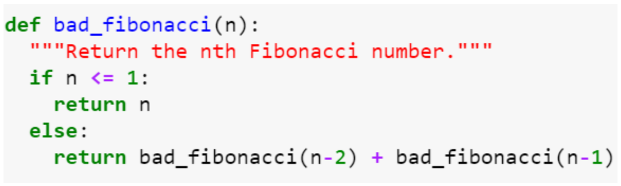
The time complexity of its activation is O(n). Two basic operations should be executed in the base case, where n equals one, and 4 operations for other conditions. Therefore, the complexity in terms of operations is constant O(1). As a result, the time complexity with input n is the cross product of n and constant, which is O(n).



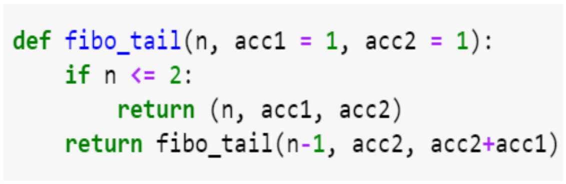
- number of activations: $\mathcal{O}(n)$ (3 for $n = 3$)
- number of operations: 2 for base case, 4 otherwise (constant running time anyway) $\rightarrow \mathcal{O}(4) = \mathcal{O}(1)$
- Total: $\mathcal{O}(n)$, or $T(n) = 4n$, $\mathcal{O}(4n) = \mathcal{O}(n)$

Computation Procedure of Time Complexity

Compare with the time complexity of tail recursive function in terms of factorial product, which is O(1), the time complexity decreased obviously after converting a recursion algorithm into tail recursive form. Similarly, this rule is also proved by the algorithms regarding calculating Fibonacci numbers.



```python
def bad_fibonacci(n):
  """Return the nth Fibonacci number."""
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n-1)
```



```python
def fibo_tail(n, acc1 = 1, acc2 = 1):
    if n <= 2:
        return (n, acc1, acc2)
    return fibo_tail(n-1, acc2, acc2+acc1)
```

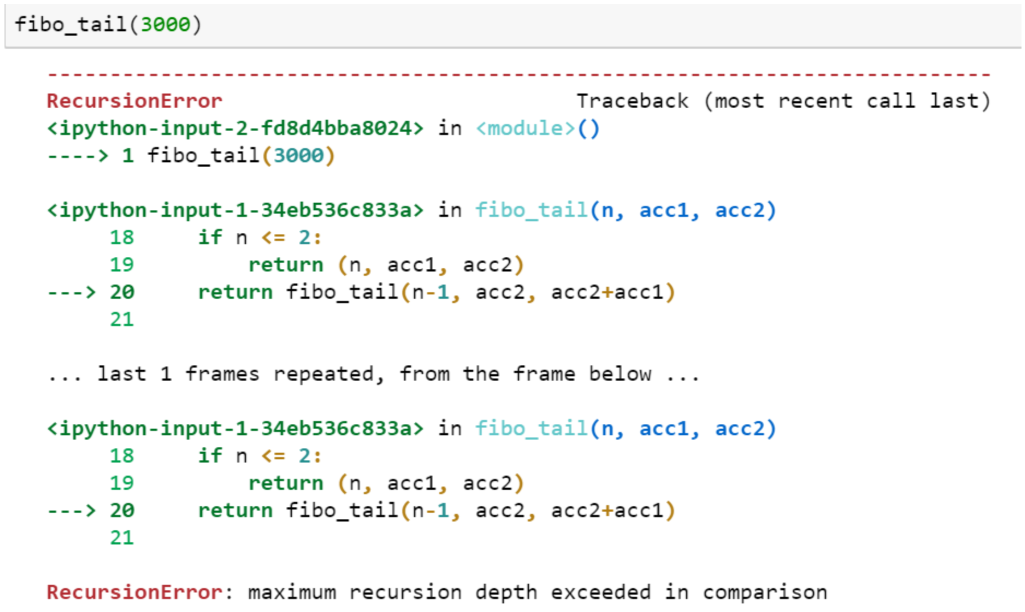
$$O((\frac{1+\sqrt{5}}{2})^n) \approx O(1.618^n)$$

O(n)

6.6 What's Missing

6.6.1 Problems in application of tail recursion

The tail recursive algorithms occupy only one stack space, therefore, there is no stack overflow problem. However, in some cases, take Python programming as an example, when we have input n=3000 the stack frame is full. The reason is that python does not support tail recursion optimization, so running the code will still hit the recursion limit.



```
fibo_tail(3000)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-2-fd8d4bba8024> in <module>()
----> 1 fibo_tail(3000)

<ipython-input-1-34eb536c833a> in fibo_tail(n, acc1, acc2)
     18     if n <= 2:
     19         return (n, acc1, acc2)
---> 20     return fibo_tail(n-1, acc2, acc2+acc1)
     21 

... last 1 frames repeated, from the frame below ...

<ipython-input-1-34eb536c833a> in fibo_tail(n, acc1, acc2)
     18     if n <= 2:
     19         return (n, acc1, acc2)
---> 20     return fibo_tail(n-1, acc2, acc2+acc1)
     21 

RecursionError: maximum recursion depth exceeded in comparison
```

Recursion Error with Large Input

Although there are some open source python decorators optimized the tail recursion algorithms, we should still keep caution in using recursion functions.

6.6.2 Recursion optimization and complexity

Another question is that, does tail recursion function always has lower level complexity? Tail recursion algorithms behave better in terms of space complexity definitely. However, is there any rule regarding the time complexity while doing such conversion?

From the two cases above, it seems true that after convert the original algorithm into tail recursion the time complexity decreased significantly. But according to an article by Bhaskar (2017), recursion and tail recursion structures can be same in computability, which depends on the structure of algorithms. And because of some nature measure of complexity, general recursion algorithms can be more efficient than tail recursion regarding some special function.

Reference:

Bhaskar, S. 2017, "A Difference in Complexity Between Recursion and Tail Recursion", Theory of Computing Systems, vol. 60, no. 2, pp. 299-313.