Table of Contents

1 Summery

Lecture 5 and lecture 6 were carried out on the study of the recursion algorithm, were lecture 5 were introduce to the concept of recursion and lecture 6 have extended on the concept of recursion.

1.1 Review of Last Lecture

In the previous lecture we have covered the following points：

- **Defined and explained the recursion and its operations** i.e. a function which can call itself; to for the purpose of decomposing into smaller sub tasks.
- **The Base case**; which is a stopping case enplaned to avoid the function to entering an infinite loops i.e. call itself forever.
- **Python and backstop**; where python in-built mechanism that is able to rise Recursion Error and break out the function if the recursion function entering an infinity loops.
- **Call stack**; number of time which a recursion function called itself.
- **Stack overflow;** call stack reaches the stack bound.

1.2 Outline

The few key concept which introduced within lecture 6 were the following:

- Recursion algorithm and iteration algorithm;
- Introduce of Tail recursion and it advantages;
- The possible methods which able to convert the recursion algorithm into iterative algorithm.
- Methods introduced to examining the complexity of recursion algorithm.

Other than the material covered within the lecture. There are few extra material which listed within this lecture note review, such as few interesting

example of recursion in practice and the work out solution of pass exam paper question

2 Today's Lecture:

As defined in the previews lecture, recursion algorithm able to break down a **large task into smaller simple sub task,** the **solution** from those **sub task combined together to produce a solution to the original task**. Note: there are three important feature of recursion algorithm:

1. A recursive algorithm **must have a base case.**
2. A recursive algorithm **must change its state and move toward the base case.**
3. A recursive algorithm **must call itself, recursively**.

2.1 Recursion and Iterative Function

Recursion and iteration able to perform same kinds of problem. Recursion and iteration shares a lot of similarities.

1. Both recursion and iteration solves problem by **executes a piece of code repeatedly**,
2. Solve a large problem by executes smaller task one at a time, and sum up the solution from smaller task to form a solution for the large problem.
3. Both possible of entering **infinite recursion / loop** if the code are improperly coded .i.e recursion: never reaches it base case, iterative: will never break out the loop if condition never becomes false.

So, what is difference?

1. **Stop points**; recursion executes the same piece of code by a function to call it itself repeatedly until it **reaches base case**, whereas irritation is a repeatedly executes a code until the controlling condition becomes false.

2. **Stack**; recursion uses of stack to store new local variable from each function called. Whereas iteration not uses of stack.
3. **Performance wise**, recursion are less efficient than iteration because for every return value from a function call a new stack is created; therefore recursion uses much more memories.
4. **Code size;** recursion reduces the size of code and allows us to write elegant whereas iteration can results in a much large code sets,
5. **Understand;** recursion are illusive i.e difficult to understand, whereas iterative can be visualise and understandable. [1]

Iterative and Recursive Algorithm

To Calculate Factorial: $n! = n \times (n-1)!$, $1! = 1$

Algorithm – Non-Tail Recursive Factorial (n)

```
1
2  def factorial(n):
3      if n == 1:  #base case
4          return 1
5      else:
6          return n * factorial(n-1)   #no of recursive call
```

Algorithm - While Loop -Iterative Factorial (n)

```
1  def factorial (n):
2      if (n <= 1):    # condition controling
3          return 1
4
5      i = 1
6      product = 1
7      while (i <= n): # condition controling
8          product = product * i
9          i = i + 1
10
11     return product
```

Take home message:

The recursive function is easy to write, but they do not perform well as compared to iteration whereas, the iteration is hard to write but their performance is good as compared to recursion

2.2 Tail Recursion

As stated by Dr Andrew Hines, "*one of the biggest drawback of non-tail recursion is a potential of stack over flow and wastage of RAM*". i.e new stack is created to store intermediate value for every function call , the intermediate value are used to calculate the final result when the recursion call returns. As stated in lecture 5, most of the programming language i.e python will set a maximum stack, once this limit is exceeded, there will be RecursionError.

Tail recursion are introduce to reduce the require recursive stack and eliminates the stack overflow and improves the recursive performance.

**Note: python doesn't support tail-call optimization.

2.2.1 So, what is Different between Tail and Non-Tail Recursion?

Non-Tail Recursion

A function is non tail recursive if there is **some processing done** after the **function returns**. This is because the result of non-recursive calculation is the sum of the return value of the every recursive call. I.e. you will only get your result of your calculation until all value are returned from every recursive call and each recursive call will result in a formation of new stack to store the intermediate value .*Note: It require recursive in then recursion call returns*

Tail Recursion

A function call is said to be **tail recursive if there is nothing to do after the function returns except return its value.** Tail recursion can perform better than non-tail recursion by perform current calculations first then passing the results from the current calculations to the next recursive and so on. Instead

of creating a new stack frame for the results of the new function (arguments, local variables, etc.), **an accumulator are used as a temporary storage** to optimized the function by **reduce the number of stack** i.e there is no need to store the current stack once you entering the next recursive step.

Note: Only require recursive in. **NO call returns**

Recursive and Tail Recursive Algorithm

Non-Tail Recursion

Example 1: non tail recursion

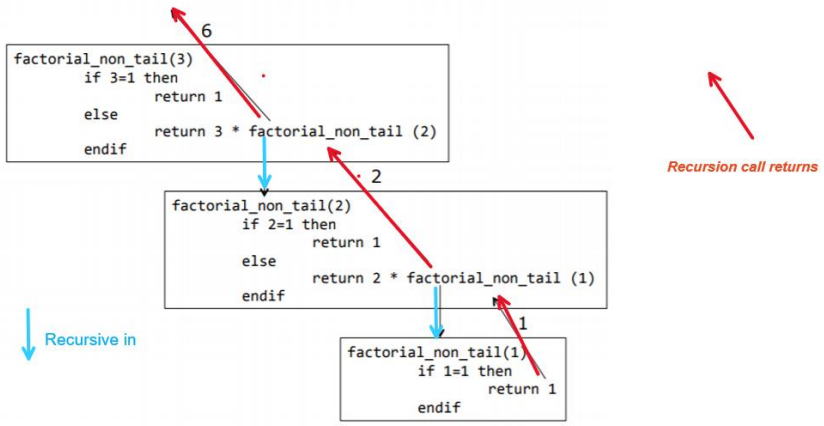
This is how we implemented factorial yesterday

Algorithm *factorial_non_tail*(*n*)

Input: n, a natural number
Output: the nth factorial number

```
1: if n = 1 then
2:     return 1
3: else
4:     return n* factorial_non_tail(n − 1)    # note n*rec. call
5: endif
```

We are multiplying the return value by n



Tail Recursion

Example 2: Tail Recursion

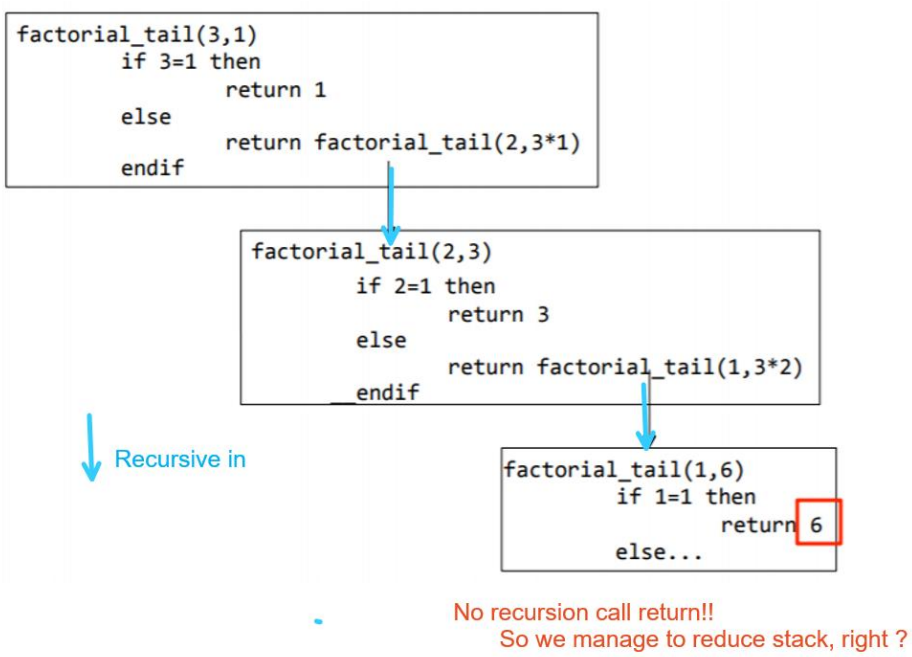
Note the differences in the base call and recursion call

Algorithm *factorial_tail*(*n*, *accumulator*)

Input: n and accumulator, two natural numbers
Output: the n^{th} factorial number

```
1: if n = 1 then
2:     return accumulator     # Note: new accumulator variable
3: else
4:     return factorial_tail(n-1, n*accumulator)     # Note n * gone
5: endif
```

We are using an accumulator for the total and finish at the tail

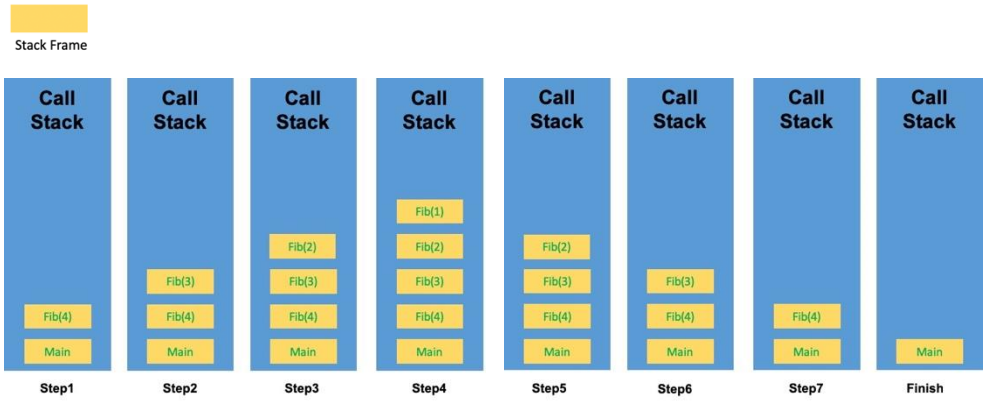


2.2.2 Let's take a look about space complexity of Tail and Non-Tail Recursion.

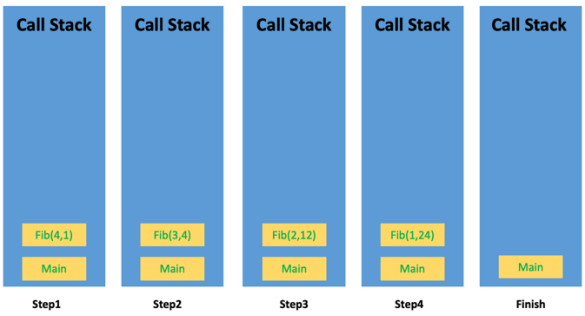
Each program has a call stack, when method is called the state of system is stored in call frame, which includes local variables and state information. Let take finding factorial of a number as example again:

When we'd find 4th factorial number using **non-tail recursion**:

The process is implemented in stack as bellow:



And tail recursion function is implemented in stack as bellow:



So, we can see tail recursion eliminates the O(n) stack space required to store function arguments in memory when performing recursive calls; the "n" here refers to the number of recursive calls done before hitting the base case. It turns that O(n) stack space into O(1) stack

space by reusing the stack frame for each recursive call rather than creating a new one for each call. This is a great space optimization.

Take home message

- *Tail recursion is usually more efficient (although more difficult to write) than non-tail recursion, because stack is minimize.*
- *No recursion call returns in tail recursion.*

2.3 Turning a recursive algorithm into an iterative algorithm

2.3.1 Why do we border to take the hassle i.e. turning recursive into iterative?

Recursive is a powerful method to for writing an elegant piece of code, but it is often less efficient with compared to iterative algorithm regardless non-tail or tail recursion. Also executing a large recursive algorithm would possible to running into stack overflow. i.e RecursionError .Taking the hassle could be the difference between getting an answer and getting an Error message,

As stated in the lecture note “It is often possible to write the same algorithm using recursive or iterative functions”. So, is there any method we can use to transfer recursive into iterative? The answer is YES, and the example of step are listed as the following.

The turning process [2]**:**

Step1. Study the original function;

Step2. Convert none tail recursive into tail recursive i.e. all recursive calls into tail calls.

Step3. Introduce a one-shot loop around the function body. I.e. while True: body; Break

Step4. Convert tail calls into continues statements.

Step5. Tidy up.

Example of Tuning Recursive into Iterative; Factorial

Step 1. Study the original function;

```
def factorial(n):
    if n < 2:
        return 1
    return n * factorial(n - 1)
```

Step 2. Convert none tail recursive into tail recursive i.e. all recursive calls into tail calls

```
def factorial1a(n, acc=1):
    if n < 2:
        return 1 * acc
    return factorial1a(n - 1, acc * n)
```

Step 3. Introduce a one-shot loop around the function body. I.e. while True: body; Break

```
def factorial1b(n, acc=1):
    while True:
        if n < 2:
            return 1 * acc
        return factorial1b(n - 1, acc * n)
        break
```

Step 4. Replace all recursive tail calls f(x=x1, y=y1, ...) with (x, y, ...) = (x1, y1, ...); continue. Be sure to update all arguments in the assignment. [2]

```
def factorial1c(n, acc=1):
    while True:
        if n < 2:
            return 1 * acc
        (n, acc) = (n - 1, acc * n)
        continue
        break
```

Step 5. Tidy the code and make it more idiomatic. [2]

```
def factorial1d(n, acc=1):
    while n > 1:
        (n, acc) = (n - 1, acc * n)
    return acc
```

2.3.2 Recursion → Dynamic Iterative Algorithm

Other than the above turning methods; there is an alternative turning methods called dynamic programming. This is an idea recursively divide a complex problem into a number of simpler sub-problems; store the answer to each of these sub problems into extra structures (e.g., arrays) to store the and, ultimately, use the stored answers to solve the original problem.

Pseudo code of: Recursion → Dynamic Iterative Algorithm

Algorithm *factorial_dynamic*(*n*)

Input: n, a natural number
Output: the n^{th} factorial number

```
1: array ← array of size n
2: array[0] ← 1
3: for i from 2 till n do
4:     array[i-1] ← array[i-2] * i
5: endfor
6: return array[n-1]
```

2.4 Complexity of Recursive Function

The time complexity of an algorithm is basically how many operations are necessary to compute it. While for nonrecursive algorithms the reasoning is quite straight-forward, for recursive algorithms, it is a little tricker, but still easy to follow. [3]

Let's take a simple example, the recursive algorithm to compute the factorial of a natural number n below.

```
def factorial(n):
    if n < 2:
        return 1
    return n * factorial(n - 1)
```

First, we compute how many operations there are in a basic call, in this example, it will be 3 operations:

1. One for if condition (is n equal to 0?)
2. one for the product of n by the output of the next call of Factorial()
3. one to subtract 1 from n in the parameter of the next call of Factorial()

Therefore, calling T(n) the time complexity of the factorial recursive program above, it will be expressed recursively as:

$$T(n) = 3 + T(n-1)$$

Deriving it to a general case, and considering that T(0) = 1, the time complexity becomes an expression of order n.

$$T(n) = 1 + 3n \longrightarrow O(n)$$

Remember that the Big O means: what really matters in this equation is. For this particular case, it means that the time complexity to compute the factorial of n is a function of n. A complexity like this one is said to be linear since it is directly proportional to the parameter n. [3]

2.5 What is missing?

There are many way to analysis of an algorithm such as complexity analysis i.e the big O notation (as discuss on the previous chapter).Recursive algorithm can be tricky to understand and often illusive , so when working with a recursive algorithm it is necessary to carry out the analysis on correctness and termination.

Correctness: the algorithm does what it claims i.e pass an input which produce the expected output.

Termination: It can be problematic if a given recursive never terminate and this will results in eating up most of the resource within the computer. This is evaluating whether a given program will definitely terminate at some point (without human interferes) i.e will the given recursive function finish after it hits the base case?

3 Conclusion

This lecture have briefly run through the concepts of recursion, the similarity and difference between recursion and iterative and concluded that the recursive algorithm require more computation and memory than the iterative algorithm. The reason for which recursive algorithm are less efficient than the iterative algorithm is because new stack frame are needed per every recursive call, however tail recursion able to eliminate the uses of stack frame and improves the performance by uses of accumulator. Also the way of examining the recursive function by estimate the number of activation and basic operations in each calls.

3.1 Recommendation

Recursion can be tricky to handing the following hint are provided within the lecture while working with recursive algorithms:

Termination condition: improperly designed recursive function have a risk of entering infinite recursive. Therefore, it is necessary to test base case to ensure every possible chain of recursive call are progress towards a base case and will eventually stop recursion when it hits the base case.

Structures: Ensure the original the algorithm able to recursive into a sub problem (break and solves the problem into simpler form) and the sub-problem should have the same general structure as the original i.e. to ensure the recursive call are structured in order to accomplish the computation you want.

Parameters: uses of extra parameters and methods overloading could makes interface cleaner.

Correctness: run a correctness test to ensure the algorithm are design as what you were expected i.e return the expected results

At lot of time the Iterative algorithm can handle what recursive algorithm do therefore, “You can still live a happy life without understand recursive” !!

4 Extracurricular - Example of Recursion

4.1 Euclid's algorithm

The greatest common divisor (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the gcd(102, 68) = 34.

We can efficiently compute the gcd using the following property, which holds for positive integers p and q: [4]

If p > q, the gcd of p and q is the same as the gcd of q and p % q.

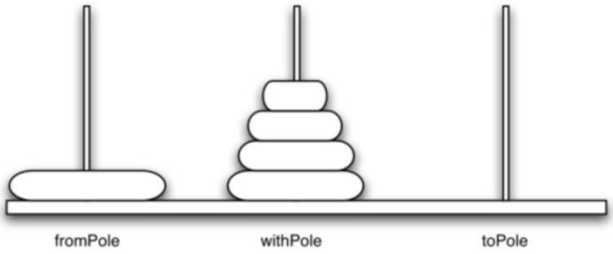
```
gcd(1440, 408)
   gcd(408, 216)
      gcd(216, 192)
         gcd(192, 24)
            gcd(24, 0)
               return 24
            return 24
         return 24
      return 24
   return 24
```

[4]

The static method gcd() belowis a compact recursive function whose reduction step is based on this property.

```
def gcd(x, y):
    if(y == 0):
        return x
    return gcd(y, x % y)
```

4.2 Towers of Hanoi



[5]

Towers of Hanoi. recursive solution to towers of Hanoi In the towers of Hanoi problem, we have three poles and n discs that fit onto the poles. The discs differ in size and are initially stacked on one of the poles, in order from largest (disc n) at the bottom to smallest (disc 1) at the top. The task is to move all n discs to another pole, while obeying the following rules: [5]

Move only one disc at a time.

Never place a larger disc on a smaller one.

Recursion provides just the plan that we need:

1. Move a tower of height-1 to an intermediate pole, using the final pole.

2. Move the remaining disk to the final pole.

3. Move the tower of height-1 from the intermediate pole to the final pole using the original pole.

```
def moveTower(height, fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height - 1, fromPole, withPole, toPole)
        moveDisk(fromPole, toPole)
        moveTower(height - 1, withPole, toPole, fromPole)


def moveDisk(fp, tp):
    print("moving disk from", fp, "to", tp)
```

4.3 Recursive tree

Fractals have the property of self-similarity. One type of figure that fulfills this requirement, are Recursion Trees. The idea is as follows: you draw the stem of the tree, then the stem splits into two smaller branches, each of these two branches again splits into two smaller branches, etc... until infinity. [4]

[4]

To code such a recursion tree, we need of course a recursive function. This function will draw one branch, and call itself again to draw two new branches, and since it has called itself again, these two called versions will again call itself again, etc... The parameters are changed each time to draw the branch at the correct position, with the correct angle and size. There's also a stop condition, it'll stop after n recursions, otherwise the calculation would never be finished and it'd end up in an infinite loop. In the nth step of recursion, 2^n branches have to be drawn. [4]

Let's translate this idea to some Python code. Figure 1 below shows how we can use our turtle(Library for drawing in Python) to generate a fractal tree. Let's look at the code a bit more closely. You will see that on lines 7 and 9 we are making a recursive call. On line 7 we make the recursive call right after the turtle turns to the right by 20 degrees; this is the right tree mentioned above. Then in line 9 the turtle makes another recursive call, but this time after turning left by 40 degrees. The reason the turtle must turn left by 40 degrees is that it needs to undo the original 20 degree turn to the right and then do an additional 20 degree turn to the left in order to draw the left tree.

Also notice that each time we make a recursive call to tree we subtract some amount from the branchLen parameter; this is to make sure that the recursive trees get smaller and smaller. [4]

```
1  import turtle
2
3  def tree(branchLen,t):
4      if branchLen > 5:
5          t.forward(branchLen)
6          t.right(20)
7          tree(branchLen-15,t)
8          t.left(40)
9          tree(branchLen-15,t)
10         t.right(20)
11         t.backward(branchLen)
12
13 def main():
14     t = turtle.Turtle()
15     myWin = turtle.Screen()
16     t.left(90)
17     t.up()
18     t.backward(100)
19     t.down()
20     t.color("green")
21     tree(70,t)
22     myWin.exitonclick()
23
24 main()
```

Figure 1

4.4 Recursion from Exam Paper

2015-16

(b) (20 marks)

(i) How many recursive calls are involved in the execution of mystery_function(25,4) below? **(5 marks)**

```
def mystery_function(i, j):
    if i == 1:
        return j
    if (i%2) ==1:
        return j + mystery_function(i/2, j*2)
    else:
        return mystery_function(i/2, j*2)
```

(ii) What is the output of mystery_function() for:

- (10, 1),
- (10, 2),
- (10, 4),
- (10, 10)? **(4 marks)**

(iii) What does mystery_function() compute? **(1 mark)**

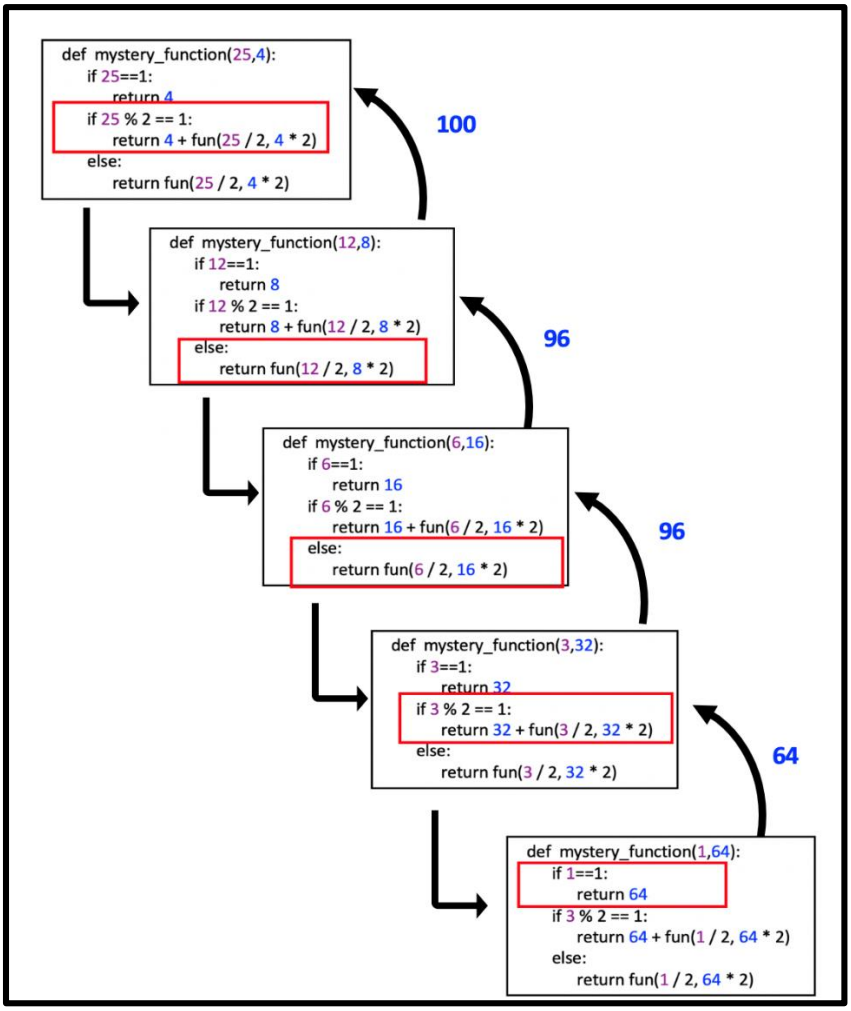
(iv) Briefly define *tail recursion* in your own words. **(5 marks)**

(v) Is mystery_function() a tail recursive function? **(5 marks)**

(i) How many recursive calls are involved in the execution of mystery_function(25,4) below?

```
def mystery_function(i, j):
    if i == 1:
        return j
    if (i%2) ==1:
        return j + mystery_function(i/2, j*2)
    else:
        return mystery_function(i/2, j*2)
```

Ans: As diagram below, there are totally 5 recursion calls.



(ii) What is the output of mystery_function() for:

mystery_function(10,1) : Output:10

mystery_function(10,2) : Output:20

mystery_function(10,4) : Output:40

mystery_function(10,10) : Output:100

(iii) What does mystery_function() compute?

Ans:

mystery_function(25,4)

output:100

(iv) Briefly define tail recursion in your own words.

Ans: Tail recursion is a special kind of recursion where the recursive call is the very last thing in the function. It's a function that does not do anything at all after the recursion call.

(v) Is mystery_function() a tail recursive function?

Ans: No, in line 5, there is a computation after the return of recursive call.

```
if (i%2) ==1:
    return j + mystery_function(i/2, j*2)
```

5 References

[1] TechDifference, "Difference Between Recursion and Iteration," Tech Differences, 30 May 2016. [Online]. Available: https://techdifferences.com/difference-between-recursion-and-iteration-2.html. [Accessed 10 Feb 2019].

[2] T. Moertel, "Tricks of the trade: Recursion to Iteration, Part 1: The Simple Method, secret features, and accumulators," Hakyll , 11 May 2013. [Online]. Available: http://blog.moertel.com/posts/2013-05-11-recursive-to-iterative.html. [Accessed 10 Feb 2019].

[3] P. L. Fernandes, "Analysis of Recursive Algorithms," Study.com, [Online]. Available: https://study.com/academy/lesson/analysis-of-recursive-algorithms.html. [Accessed 10 Feb 2019].

[4] Robert Sedgewick, Kevin Wayne, "Recursion," princeton.edu, 02 August 2016.. [Online]. Available: https://introcs.cs.princeton.edu/java/23recursion/. [Accessed 10 Feb 2019].

[5] Brad Miller and David Ranum, , "Tower of Hanoi," Luther College, [Online]. Available: http://interactivepython.org/runestone/static/pythonds/Recursion/TowerofHanoi.html. [Accessed 10 Feb 2019].