DESIGN AND ANALYSIS OF ALGORITHMS – CSCI 612

NAME: AADITHYA GOPALAKRISHNA BHARADWAJ

ZID: Z1862641

ASSIGNEMENT NUMBER, SEMESTER: Bonus Assignment, SPRING 2019

==========

Q.) You are expected to write a study report regarding recursion. Start with an introduction to the concept and then provide many examples to

1. **Show how recursion is implemented;**
2. **Show when recursive implementation is better than non-recursive implementation;**
3. **Show and explain when it is not good to use recursion;**
4. **Draw memory call stack to show how recursion works in memory;**
5. **Explain tail recursion.**

Answers.)

Recursive function can be defined as a routine that references itself directly or indirectly.

Eg:
```
int Recursion (x)
{
        if (x==0)
        return;
        recursion (x-1);
}
```

Using recursive algorithm, certain problems can be solved quite easily. Towers of Hanoi (TOH) is one such programming exercise. Moreover, every recursive program can be written using iterative methods. Mathematically recursion helps to solve few puzzles easily.

For example: In a party of N people, each person will shake her/his hand with each other person only once. On total how many hand-shakes would happen?

Recursively it can be solved in the following manner: Considering N^{th} person, he/she has to shake-hand with (N-1) persons. Now the problem reduced to small instance of (N-1) persons. Assuming T_N as total shake-hands, it can be formulated recursively.

$T_N = (N-1) + T_{N-1}$ [$T_1 = 0$, i.e. the last person has already shook-hand with every one].

Solving it recursively yields an arithmetic series, which can be evaluated to N(N-1)/2.

Usually recursive programs results in poor time complexities. An example is Fibonacci series. The time complexity of calculating N^{th} Fibonacci number using recursion is approximately 1.6^n. It means the same computer takes almost 60% more time for next Fibonacci number. Recursive Fibonacci algorithm has overlapped subproblems. There are other techniques like *dynamic programming* to improve such overlapped algorithms.

However, few algorithms, (e.g. merge sort, quick sort, etc…) results in optimal time complexity using recursion.

Base Case: One critical requirement of recursive functions is termination point or base case. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behavior.

Algorithm for TOH:

How it should work:

If we have 2 disks –

- ➔ First, we move the smaller (top) disk to aux peg.
- ➔ Then, we move the larger (bottom) disk to destination peg.
- ➔ And finally, we move the smaller disk from aux to destination peg.

For more than 2 disks:

- ➔ We divide the stack of disks in two parts.
- ➔ The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.
- ➔ Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it.

**Note: If we have only one disk, then it can easily be moved from source to destination peg.*

So now using recursive technique following is the algorithm:

Step 1: *Mark three towers with name, **source**, **destination** and **temp** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.*

Step 2: *Move n-1 disks from source to temp.*

Step 3: *Move n^{th} disk from source to destination.*

Step 4: *Move n- disks from temp to destination.*

Recursive approach is good if you are dealing with trees like binary trees. For traversing and inserting values in nodes Recursive method is best approach when compared to iterative approach(non-recursive). Recursion is big plus in case of printing Binary search tree and also for inserting, searching. If input is non-trivial that means not smaller we have to use recursion. The basic idea behind recursion is solve problem by breaking it into smaller ones. This will be useful when dealing with larger implementations, example: - BST operations: Binary search tree operations need so many insertions into deeper nodes.

For simple trivial programs like Fibonacci **we don't need recursion.** Because if we use recursion in Fibonacci it will take O(2^n) steps but if we use iteration it will take O(n) linear time.

And also, there are some problems with recursion like infinite recursion if we do not use correct base case. It is difficult to implement recursive algorithms.

Hence it is better to use Recursive technique when mathematical approach is required whereas in programming mostly iteration method is best.

Recursive functions use something called "the call stack." When a program calls a function, that function goes on top of the call stack. This similar to a stack of books. You add things one at a time. Then, when you are ready to take something off, you always take off the top item. Example: call stack in action with the factorial function: factorial (5) is written as 5! and it is defined like this: 5! = 5 * 4 * 3 * 2 * 1. Here is a recursive function to calculate the factorial of a number:

```
function fact(x)
{
        if (x == 1)
        {
                return 1;
        }
        else
        {
                return x * fact(x-1);
        }
}
```

Now let's see what happens if you call fact (3) The illustration bellow shows how the stack changes, line by line. The topmost box in the stack tells you what call to fact you're currently on.

Code	**Call Stack**	
fact (3)	FACT: x = 5	First call to fact x is 3
if x == 1	FACT: x = 3	
else	FACT: x = 3	

DESIGN AND ANALYSIS OF ALGORITHMS – CSCI 612

A Recursive Call! → return x * fact (x-1)

FACT	
x	2
FACT	
x	3

Now we are in the second call to fact: x is 2

if x == 1

FACT	
x	2
FACT	
x	3

The topmost function call is the call we are currently in

---

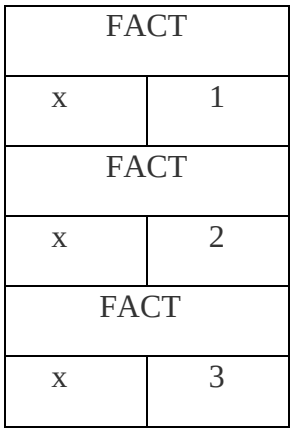
else

FACT	
x	2
FACT	
x	3

Note: Both the function calls have a variable named x and the value of x is different in both

---

return x * fact (x-1)

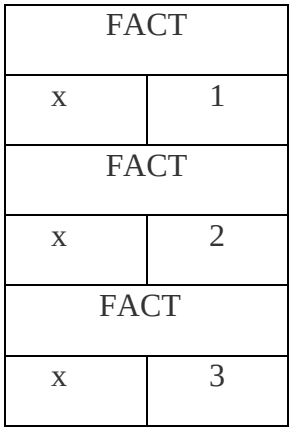


You can't access this call x

From this call and vice versa

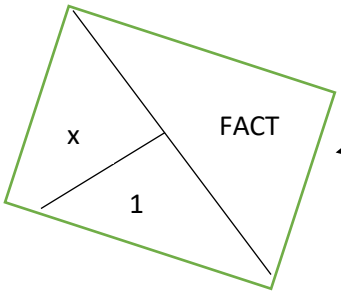
---

if x == 1



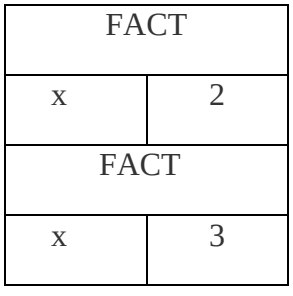
---

return



This is the first box to get popped off the stack, which means it's the first call we return from

We made **three** calls to the fact, but still we have not finished a single call till now!



Returns 1

Notice how each call to fact has its own copy of x. This is very important to making recursion work. You can't access a different function's copy of x.

Tail Recursion:

A recursive function is tail recursive when recursive call is the last thing executed by the function. In traditional recursion, the typical model is that you perform your recursive calls first, and then you take the return value of the recursive call and calculate the result. In this manner, you don't get the result of your calculation until you have returned from every recursive call. In tail recursion, you perform your calculations first, and then you execute the recursive call, passing the results of your current step to the next recursive step. This results in the last statement being in the form of (return (recursive-function params)). Basically, the return value of any given recursive step is the same as the return value of the next recursive call

The tail recursive functions considered better than non-tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

For example, the below code is tail recursive (since the final instruction is a recursive call)

```
int f (int x, int y)
{
        if (y == 0)
        {
                return x;
        }
        return f (x*y, y-1);
}
```