C - Programming

Recursion

Recursion – Introduction

- A powerful approach to solve computational problems is by solving smaller versions of the original problem - the recursive problem solving
- In mathematics also, we define functions in terms of same (but smaller versions) function - called Inductive definition
- For example, the inductive and non-inductive definition of factorial function is given below

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n\text{-}1)!*n & \text{if } n > 0 \end{cases} \quad (\textit{recursive} \text{ solution})$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n\text{-}1)*n & \text{if } n > 0 \end{cases} \quad (\textit{closed form} \text{ solution})$$

Recursion – Introduction

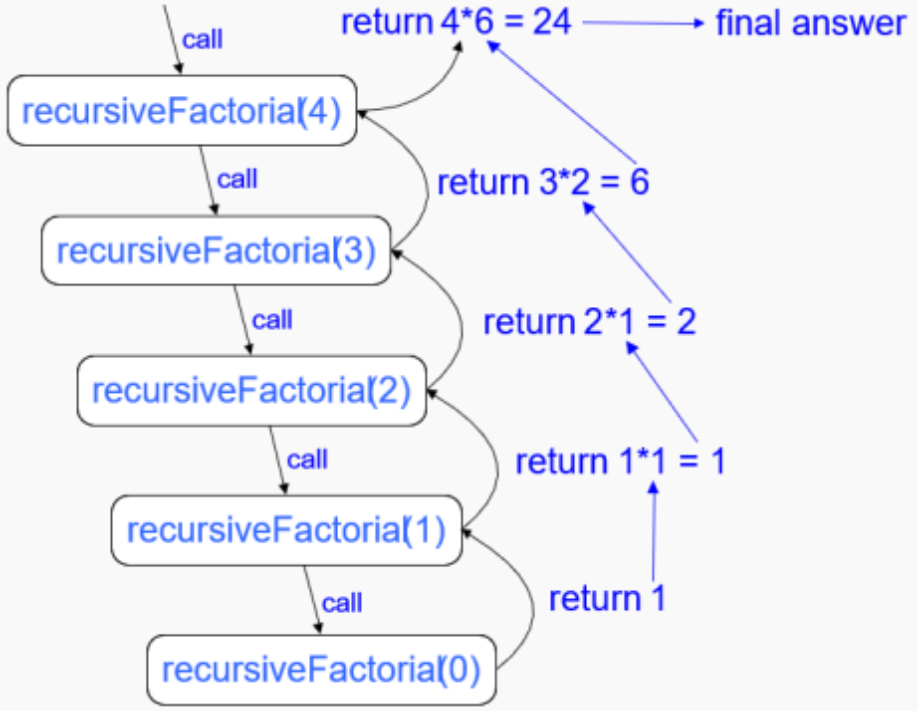
- Programming languages support recursive problem solving using recursive functions (or recursion simply)
- A recursive function is a function that calls itself.
- As an example, the recursive solution of the factorial function in C language is,

```
int fact(int n)
{
    if (n==0) return 1;
    else return n * fact(n-1);
}
```

How does a recursion work?

Tracing the recursion

- Every recursive call is a function call in which the body of the function has to be executed (with the new value of the parameters)
- Recursion visualizers
  https://visualgo.net/en/recursion
  https://www.cs.usfca.edu/ galles/visualization/RecFact.html



Characteristic prooerties of recursion

- Base case - Every recursive function must have at least one base case where the recursion stops. For example, the statement 1 in the factorial code is the base case.

- Recursive case - Every recursive function must have at least one recursive case. For example, the statement 2 in the factorial code is the recursive case.

- Progress – Every recursive call is made with an input that is closer to the base case. (to ensure that the recursion eventually reach the base case and terminates)

Problems

Identify the base case(s) and recursive case(s) in the following recursions.

```
int f2(int n){
if (n==0) return 1;
else if (n==1) return 1;
else return f2(n-n/2);
}

int f3(int n){
if (n==0||n==1) return 1;
else return f3(n-1) + f3(n-2);
}

int f4(int n){
if (n>0) return f4(n/2);
}
```

Problems – Solution

Identify the base case(s) and recursive case(s) in the following recursions.

```
int f2(int n){
if (n==0) return 1;
else if (n==1) return 1;
else return f2(n-n/2);
}
```

The function f2 has two base cases n==0 and n==1, It a has a single recursive case

```
int f3(int n){
if (n==0||n==1) return 1;
else return f3(n-1) + f3(n-2);
}
```

The function f3 has two base cases n==0 and n==1, It a has a single recursive case

```
int f4(int n){
if (n>0) return f4(n/2);
}
```

The base case is not explicitly given, but the recursion terminates when $n <= 0$.

Problems

Identify the errors in the following recursions.

```
int f5(int n){
if (n==1) return 1;
else return f5(a-2);
}

int f6(int n){
return f6(n-1);
if (n==1) return 0;
}
```

Problems – Solution

Identify the errrors in the following recursions.

```
int f5(int n){
if (n==1) return 1;
else return f5(a-2);
}
```

Termination not guaranteed, for example if n is an even number.

```
int f6(int n){
return f6(n-1);
if (n==1) return 0;
}
```

infinite recursion since the base case is unreachable.

Problems

Write a recursive function for finding nth Fibonacci number.

Solution

Write a recursive function for finding nth Fibonacci number.

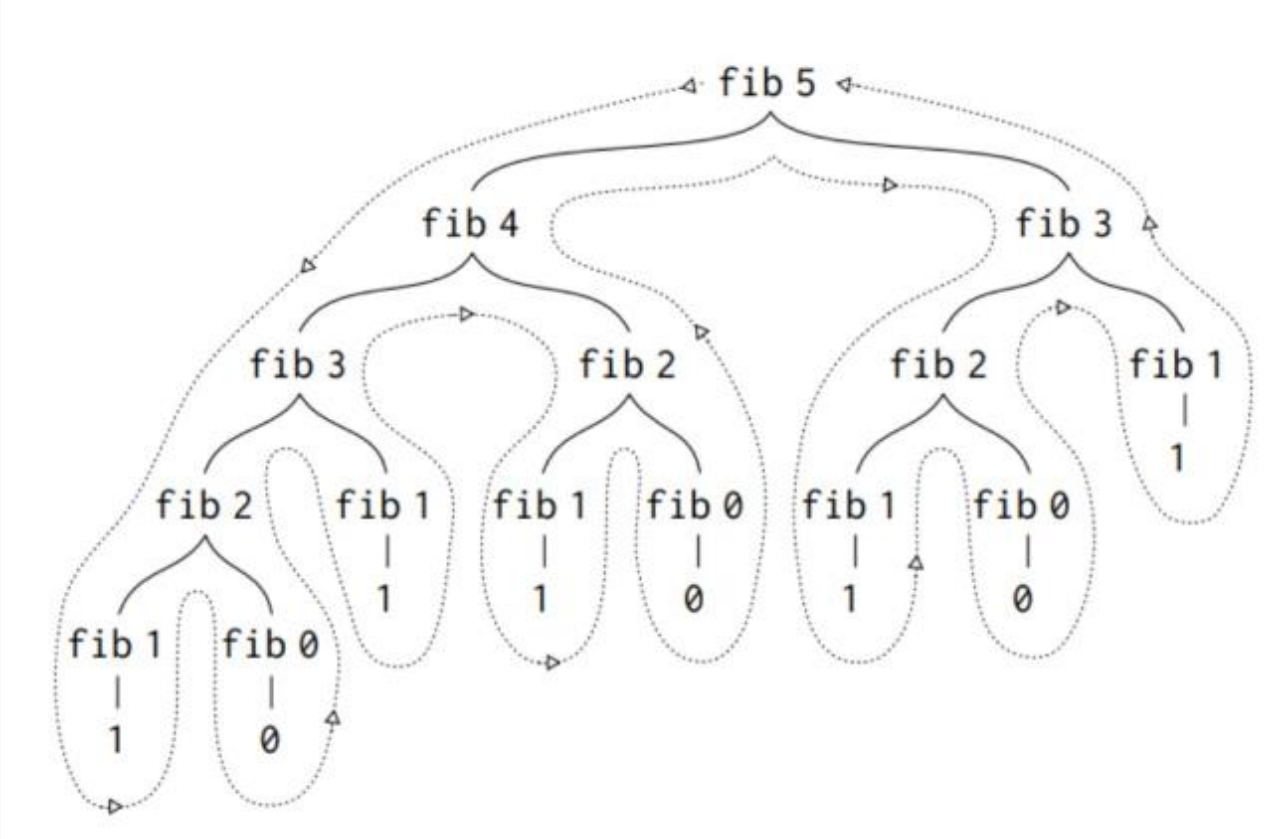
```
int fib(int n){
    if (n==0) return 0;
    else if (n==1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

Problem

Trace the execution of the fib function, for $n = 5$.

Solution

Trace the execution of the fib function, for $n = 5$.



The dotted line shows the order in which the fib function recurses.

Function call mechanism

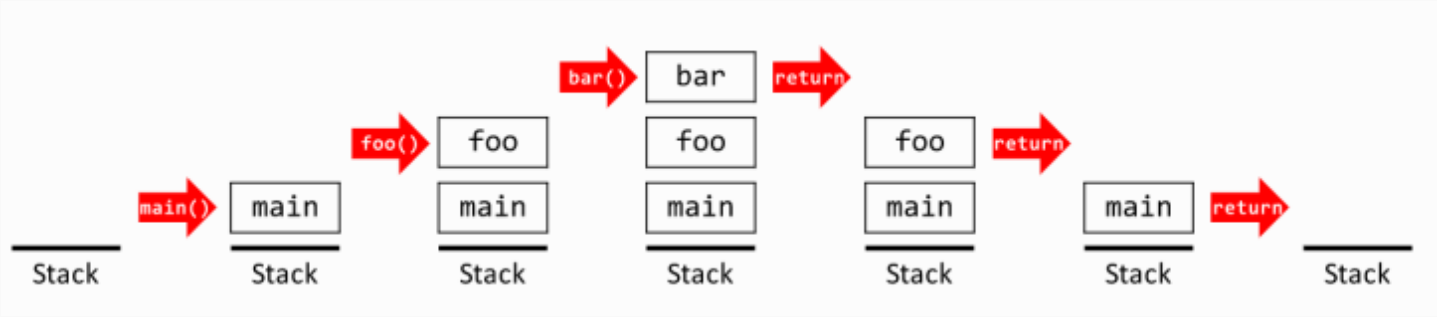
- Activation record of a function contains all the information necessary to keep track of an active function call.
- This includes the details like parameters of the function, return value of the function, local variables of the function, and the line in the function that is currently being executed etc..
- As the progam calls more and more functions, the activation records are piled-up to form the Activation stack.

Function call mechanism

- For example, when the program starts execution main() function is called - So an activation record of main() function is created and added to the activation stack.
- When the main() function calls another function, say foo(), an activation record for foo() is created and it is added at the top of activation stack.
- Similarly when the function foo() calls another function say bar(), an activation record for bar() is created and is added to the top of the stack.
- The function foo() becomes inactive only after bar() terminates and main() becomes inactive only after foo() terminates.

Function call mechanism

This function call mechanism is depicted in the following figure. Note that the red arrow shows the function call and return.



[1]

- The activation record is temporary - it is allocated when the function is called (becomes active) and it is deallocated when the function returns (becomes inactive).
- The importance of the call stack is that each called function always finds the information it needs to return to its caller at the top of the call stack

[1]Image: https://eecs280staff.github.io/notes/02_ProceduralAbstraction_Testing.html