CSCI 241 - Notes
Recursion

Matt Warner

1 Overview

Recursion is a general programming technique used to solve problems with a "divide and conquer" strategy

Note:-

most computer programming langauges support recursion by allowing a function or member function to call itself within the program text.

Example of recursion

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

A recursive function call will always by conditional. There must tbe at least one base case for which the function produces a result trivially without a recursive call. For example:

```
int factorial(int n){
  if (n == 1 ){   // Base case - no recursion
    return 1;
  }
  else
  {
    return n * factorial(n-1);
  }
}
```

A function with no base cases leads to "infinite recursion" (similar to an infinite loop)

In addition to the base cases, a recursive function will have one or more recursive cases. The job of a recursive case can be seen as breaking down complex inputs into simpler ones.

In a properly designed recursive function, with each recursive call, the input problem must be simplified in such a way that eventually the base case must be reached. For example:

```
int factorial(int n) {
  if (n == 1) {
    return 1;
  }
  else
    return n * factorial(n - 1); // n - 1 gets us closer to 1
}
```

Recursion requires automatic storage - each new call to the function creates its own copies of the local variables and function parameters on the program stack.

Recursion is never a required programming technique. Recursion can always be replaced by either

1. A loop
2. A loop and a stack (that takes the place of the program call stack used by recursion)

In the first case, a non-recursive algorithm will usually be superior, if only in terms of memory usage. But in the second case, we might choose a recursive algorith if it is shorter or eaiser to code (which is often true).

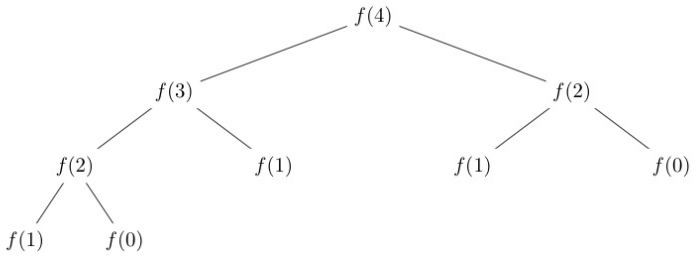
2 Recursive Trees

A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

For instance, consider the following recursive function:

```
int fib(int n) {
  if (n < 2)
    return n;
  return fib(n - 1) + fib(n - 2);
}
```

Here is an example recursive tree for **fib(4)**, note the repeated computations:



3 Tail Recursion

Tail recursion is defined as a recursive function in which the recursive call is the last statement that is executed by the function. So basically, nothing is left to execute after the recursive call.

```
  void print(int n) {

    if (n < 0) {
      return;
    }
    // The last executed statement is recursive call
    print(n - 1);
}
```

3.1 Need for Tail Recursion

The tail recursive functions are considered better than non-tail recursive functions as tail-recursion can be optimized by the compiler.

Compilers usually execute recursive procedures by using a *stack*. This stack consits of all the pertinent information, including the parameter values for each recursive call.

When a procedure is called, its information is *pushed* onto a stack, and when the function terminates the information is *popped* out of the stack. Thus for the non-tail recursive functions, the *stack depth* (maximum amount of stack space used at any time during compilation) is more.

Can a non-tail-recursive function be written as tail-recursive to optimize it?

Consider the following function to calculate the factorial of n.

```
unsigned int fac(usigned int n) {
  if (n <= 0)
    return 1;
  return n * fac(n - 1);
}
```

Although this looks like tail-recursion, if we take a closer look we can see that the value returned by **fac(n - 1)** is used in **fac(n)**. So the call to **fac(n-1)** is not the last thing done by **fac(n)**. The above function can be written as a tail-recursive function. The idea is to use one more argument and accumulate the factorial value in the second argument. When n reaches 0, return the accumulated value.

```
unsigned factTR(unsigned int n, unsigned int a) {
  if (n <= 1)
    return a;
  return factTR(n - 1, n * a);
}

// Wrapper over factTR
unsigned int fact(unsigned int n) { return facttr(n,1); }
```