Recursion (I): Linear Recursion, Tail Recursion

CSCD 300 – Data Structures

Eastern Washington University

Outline

1. Recursion
2. Linear recursion
3. Tail recursion

Recursion

In particular, we discuss recursive functions, which

- directly or indirectly call themselves inside of the function body.
- have some certain exit conditions, upon which the function will not endlessly call themselves recursively.

We will demonstrate these characteristics by going through example recursive functions.

Example 1: the factorial function

Definition

The factorial of a non-negative integer n, denoted as $n!$, is defined as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n & \text{if } n \geq 1 \end{cases}$$

The recursive structure in the definition of factorial.

Obviously, the factorial can be rewritten recursively:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{if } n \geq 1 \end{cases}$$

Suppose we have the following function that computes n!

```
int factorial(int n) //n >= 0
```

We can easily implement `factorial` as a recursive function by using the recursive structure in the definition of factorial.

```
int factorial_recursion(int n)  //assume n >= 0. Time cost: T(n)
{
   if(n == 0)                   // the exit condition
      return 1;                 // time cost: c

   return factorial_recursion(n-1) * n; // recursive call
                                        // time cost: T(n-1)
}
```

Time cost

$$
\begin{aligned}
T(n) &= c + T(n-1) = c + c + T(n-2) = c + c + \ldots + c + T(1) \\
&= \underbrace{c + c + \ldots + c}_{n \text{ terms}} = cn = O(n)
\end{aligned}
$$

System overhead caused by recursions

Whenever a function call happens during the run of a Java program, the system needs to use extra memory space to store the state (for example: all the caller's local variables and their values) of the caller before the execution can go into the subroutine being called.

Thus, each recursive function call also introduces extra memory space usage to store the recursive caller's state. That means, if the recursion goes very deep, this extra space overhead can be significant. It can even be worse that overly-deep recursions can crash a system [1], because a system normally has a limit on the depth of function calls. Such space overhead needed in the state store–restore transitions could also slows down the run of the program.

[1] You can try the code at Page 5 with a large n to crash your system.

So ...

We often want to avoid recursion if the alternative implementation is also efficient and is not much more complicated.

Recursion based

```
 //assume n >= 0
int factorial_recursion(int n)
{
   if(n == 0)
      return 1;

   return factorial_recursion(n-1) * n;
}
```

Time cost: $O(n)$

$\Longrightarrow$

Iteration based

```
//assume n >= 0
int factorial_iteration(int n)
{
   if(n == 0) return 1;

   int result = 1;
   for(int i = 1; i <= n; i++)
      result = result * i;
   return result;
}
```

Time cost: $O(n)$

Then why do we need recursive functions ? [2]

There can be many different reasons. For example,

- The solution to a given problem shows its recursive structure, and thus it's natural to implement it recursively.
- The recursive solution is easier to code, avoiding many complex case analysis.
- In the case where the recursive solution is not too bad, meaning the system overhead is not too high.
- The recursion-based code can be more readable.
- ...

[2] In CSCD320, we will see more examples where recursion can be either a good or a bad idea.

Linear recursion

Linear recursion is a recursive function in which there is at most one recursive call.

`factorial_recursion()` that we have discussed is one example of linear recursion.

Tail recursion

A tail recursion is a recursive function, such that

1. the recursive function a linear recursion.
2. inside of the function body, the last operation is the recursive call.

Note

The tail recursion requires that the recursive call must be the last operation, not just part of the last statement in the function body.

Negative example

`factorial_recursion()` is not a tail recursion, because its last operation is a multiplication after the recursive call returns.

Let's look at an example of tail recursion ...

Example 2: reverse an array

```
/* reverse the elements in A[i...j] */
ReverseArray_recursion(A, i, j)              //Time cost: T(j-i+1)
{
   if(i<j)
      swap(A[i], A[j])                        //Time cost: c
      ReverseArray_recursion(A, i+1, j-1)     //Time cost: T(j-i-1)
}
```

A function call `ReverseArray_recursion(A,0,n-1)` will reverse the whole array A, where n is the size of the array A.

Time cost of `ReverseArray_recursion(A,0,n-1)`

$$\begin{aligned} T(n) &= c + T(n-2) = c + c + T(n-4) = c + c + \ldots + c + T(1) \\ &= \underbrace{c + c + \ldots + c}_{n/2 \text{ terms}} = cn/2 = O(n) \end{aligned}$$

Tail recursion $\Longrightarrow$ Iteration

A tail recursion can ALWAYS be transformed to be an iteration-based implementation, so the system overhead caused by the recursion can be avoided.

For example,

Recursion based

```
ReverseArray_recursion(A, i, j)
{
   if(i<j)
      swap(A[i], A[j])
      ReverseArray_recursion(A,i+1,j-1)
}
```

Iteration based

```
ReverseArray_iteration(A, i, j)
{
   while(i<j)
      swap(A[i],A[j]);
      i ++;
      j --;
}
```