4 Recursion: tail recursion

A previous version of this course was taught in the first rather than the second term and so it was useful to deal with tail recursion. This material has now been removed, but can be see here.

The factorial also provides a convenient example for discussing tail recursion. One problem with recursion is that it can be wasteful of stack memory, roughly speaking the memory the program uses to run. If you call `factorial(10)` the program will write information related to `factorial(10)`, `factorial(9)` and so on down to `factorial(1)` onto the stack before `factorial(1)` returns and the open functions get rolled, being deleted from memory. In short, a copy of `factorial(9)` needs to be stored while it waits for `factorial(8)` to return the value 40320 to it, so that it can multiply that by 9 and return the result, 362880, on to `factorial(10)`. This isn't really a problem here, the factorial function will reach values well beyond the capacity of int long long before the memory use on the stack becomes a problem; however, in other circumstances it might be a problem. The solution is tail recursion.

An algorithm is tail recursive if the return value of the function does not have to be modified before it is returned. The factorial function in Table **??** is not tail recursive because factorial(n-1) is multiplied by n before it is returned. However, the version in Table **??** is tail recursive, it manages this by passing around another variable called big_n that holds the part of the factorial that is calculated so far. Now, when `factorial(10)`, for example, is called it will call `factorial(10,1)`, since $10 > 2$ this will call `factorial(9,10)`; the only thing remaining for `factorial(10,1)` to do is to return the value of `factorial(9,10)` when it is done. This means, with a bit of cunning, the function does not have to remain written on the stack, the compiler just has to know that whatever `factorial(9,10)` returns should be sent to whatever called `factorial(10)`; modern compilers are capable of this cunning so tail recursive algorithms make more efficient use of stack memory.

```
int factorial(int n)
{
    if(n<2)
        return 1;

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

Table 1: The recursive function for calculating $n! = n(n-1)\dots 1$. If $n < 2$ it returns 1, giving a terminating condition, it also means $0! = 1$ which is a normal mathematical convention, otherwise it calls factorial(n-1). If you trying using this function, note that for even modest values of n, n! is too big to fit into int.

```c
int factorial(int n)
{
   return (n<2) ? 1 : n*factorial(n-1);
}
```

Table 2: A fancier version of the factorial program which uses the ternary operator described in Table **??**.

```c
if (a)
   ans=b;
else
   ans=c;
```

Table 3: The ternary operator ans = a ? b : c evaluates a and then does either sets ans=b or ans=c depending on whether a is true of false. Thus ans=a ? b : c is equivalent to the code above. Ternary operators are often faster to execute than the corresponding if statement.

```c
int factorial_r(int n, int big_n)
{
  if(n<2)
    return big_n;

  return factorial_r(n-1,n*big_n);
}

int factorial(int n)
{
  return factorial_r(n,1);
}
```

Table 4: The tail recursive function for calculating $n! = n(n-1)\dots 1$. If $n < 2$ it returns big_n, otherwise it calls factorial(n-1,n*big_n). Since nothing happens to the factorial(n-1,n*big_n) before it is itself returned, this is an example of tail recursion.

```
factorial_r(10,1)→factorial_r(9,10)→factorial_r(8,90)→
factorial_r(7,720)→factorial_r(6,5040)→factorial_r(5,30240)→
factorial_r(4,151200)→factorial_r(3,604800)→factorial_r(2,1814400)→
factorial_r(1,3628800)
```

Table 5: The calling sequence of the tail recursive factorial program.