4 Recursion

A process is recursive if it repeats itself in a self-similar way. An easy example is the factorial

$$n! = n(n-1)(n-2)\dots 1 \tag{1}$$

where n is multiplied by $n-1$ and then by $n-2$ and so on down to one; or, put another way

$$n! = n(n-1)! \tag{2}$$

and $1! = 1$.

In computer science recursion refers to an approach where a problem is solved by breaking it up into smaller similar problems. In practise this often means solving a problem using a function that calls itself. An easy example of recursion is given by the factorial; see Table 1 or, for a fancier version Table 2. You will have learned that recursion is often a good way to implement algorithms on computers, that many algorithms can be designed recursively and that this is usually the best way to program them. Being comfortable with recursion is one of the signs of, and benefits of, learning to program properly! We will also see that working out the big-oh complexity for many recursive algorithms can be straight-forward using a set of formula known as the master equation.

A recursive function consists of two parts, a 'recursive case', what happens normally, here return n*factorial(n-1), and a 'terminating condition' or 'base case', a set of cases that the algorithm will always arrive at and can deal with without calling itself. These are important to avoid infinite recursion.

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 1 is not tail recursive because factorial(n-1) is multiplied by n before it is returned. However, the version in Table 4 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.

Our task here is to calculate the algorithmic complexity of recursive algorithms. For simplicity we will not worry about tail recursion, though in practise, calculating the complexity for algorithms with tail recursion is no harder.

The trick is to work out a recursive formula for $T(n)$, the run time. Consider the factorial example; here

$$T(n) = c + T(n-1) \tag{3}$$

where c is a constant representing the computational time required for factorial(n) itself, the if statement, the multiplication and so on. Now we can expand it out

$$T(n) = c + T(n-1) = c + [c + T(n-2)] = 2c + T(n-2) \tag{4}$$

and this keeps going

$$T(n) = 2c + T(n-2) = 3c + T(n-3) = 4c + T(n-4) = \ldots = (n-1)c + T(1) \tag{5}$$

so factorial is $O(n)$.

Another approach is to use an ansatz, that is to guess the answer. For

$$T(n) = c + T(n-1) \tag{6}$$

we might guess from experience that this has a solution of the form

$$T(n) = An + B \tag{7}$$

for some A and B we haven't specified yet. In fact $T(n) = An + B$ represents a whole family of possible solutions corresponding to different As and Bs, we just need to show that family contains an actual solution. We do this by substitution. If we substitute into the recursion relation we get

$$An + B = c + A(n-1) + B = An + B + c - A \tag{8}$$

which holds provided $A = c$. In otherwords we can make an educated guess as to the form of the solution and then show by substitution that there is a solution of this form.

We have already seen another recursive algorithm, although we didn't write it as one: binary search. A recursive version of binary search is given in Table 6. Here, leaving out rounding effects and so on, in the worst case

$$T(n) = c + T(n/2) \tag{9}$$

```
1 int factorial(int n)
2 {
3     if(n<2)
4         return 1;
5
6     return n*factorial(n-1);
7 }
```

Table 1: The recursive function for calculating $n! = n(n-1)\ldots 1$. 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.

```
1  int factorial(int n)
2  {
3     return (n<2) ? 1 : n*factorial(n-1);
4  }
```

Table 2: A fancier version of the factorial program which uses the ternary operator described in Table 3.

```
1  if (a)
2     ans=b;
3  else
4     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.

```
1  int factorial_r(int n, int big_n)
2  {
3    if(n<2)
4      return big_n;
5
6    return factorial_r(n-1,n*big_n);
7  }
8
9  int factorial(int n)
10 {
11   return factorial_r(n,1);
12 }
```

Table 4: The tail recursive function for calculating $n! = n(n-1)\ldots 1$. 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.

```
1  int search(int a[],int n, int val)
2  {
3    return find_r(a,n,val,0,n-1);
4  }
5
6  int find_r(int a[],int n, int val,int low,int high)
7  {
8
9    if(high<low)
10     return -1;
11
12   int mid=(high+low)/2;
13
14   if(a[mid]==val)
15     return mid;
16
17   if(val>a[mid])
18     return find_r(a,n,val,mid+1,high);
19
20   return find_r(a,n,val,low,mid-1);
21 }
```

Table 6: A recursive implementation of binary search. There are two halting conditions, val is found, or high¡low, meaning that val isn't an element of a. Note that, though each call works with a smaller and smaller number of elements, for convenience the same array is used each time. This function is implemented in `binary_search_recursive`.

which is solved by

$$T(n) = c\log_2(n) \tag{10}$$

because

$$T(n/2) = c\log_2(n/2) = c\log_2(n) - c\log_2(2) = c\log_2(n) - c \tag{11}$$

and so, substituting back into the equation, this is the solution. Here, again, working out the run time requires that you know how to solve the recursion relation. Our approach here is to do what we have been doing, we guess, based on the examples we've studied, and then show we are correct by substituting back in.