5 Recursion

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

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

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.

Our task here is to calculate the algorithmic complexity of recursive algorithms. 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 4. Here, leaving out rounding effects and so on, in the worst case

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

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.

Some more recursion examples

Leaving out the algorithms for the minute, lets examine some other examples of solving recursion relations. First lets consider

$$T(n) = T(n-1) + 3 \tag{12}$$

with $T(0) = 1$. Now

$$T(n) = T(n-1) + 3 = T(n-2) + 3 + 3 = T(n-3) + 3 \cdot 3 = \dots \tag{13}$$

It would be easy to solve this directly by telescoping, but lets use an ansatz, since there is clearly a 3 for each iteration we'll try $T(n) = 3n + A$, substituting in

$$3n + A = 3n - 3 + A + 3$$

```
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 $n! = n(n-1)\dots 1$. 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.

```
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 either sets ans=b or ans=c depending on whether a is true or 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 search(int a[],int n, int val)
2  {
3      return find_r(a,val,0,n−1);
4  }
5
6  int find_r(int a[], 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,val,mid+1,high);
19
20     return find_r(a,val,low,mid−1);
21  }
```

Table 4: 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`.

so the equation holds for all A, alternatively substituting a more general ansatz of the form $T(n) = Bn + A$ would give you

$$Bn + A = Bn - B + A + 3$$

which holds for all A and $B = 3$. Either way $T(n) = 3n + A$, now the initial condition is $T(0) = 1$ but setting $n = 0$ gives $T(0) = A$ so $A = 1$ and the solution is $T(n) = 3n + 1$. If instead of $T(0) = 1$ we had $T(1) = 1$ the question is exactly the same except the $T(1) = 1$ so taking $T(n) = 3n + A$ again $T(1) = 3 + A = 1$ and hence $A = -2$ and the solution is $T(n) = 3n - 2$.

Now lets look at a harder one:

$$T(n) = 2T(n - 1) + 3 \tag{14}$$

with $T(0) = 1$. Now

$$T(n) = 2T(n - 1) + 3 = 4T(n - 2) + 2 \cdot 3 = 2^3 T(n - 3) + (4 + 2 + 1) \cdots 3 = \cdots \tag{15}$$

Doing this directly by telescoping requires skill because you need to know that $2^{n-1} + 2^{n-2} + \ldots + 1 = 2^n - 1$, but you could guess, based on the 2^3 that

$$T(n) = 2^n A + B \tag{16}$$

and hope for the best, substituting in gives

$$2^n A + B = 2^n A + 2B + 3$$

so this is a solution when $B = -3$, hence

$$T(n) = A2^n - 3$$

and the initial condition gives $T(0) = A - 3 = 1$ so $A = 4$ and

$$T(n) = 2^{n+2} - 3$$

Here is another tricky example, consider $T(n) = T(n - 1) + 3n$ with $T(1) = 1$. Now telescoping gives bits that look like $3n + 3(n - 1) + 3(n - 2) + \ldots$, in other words, you seem to get an n in every iteration of the telescope. Since there are n iterations of the telescope you might guess

$$T(n) = An^2 + Bn + C \tag{17}$$

so substituting that in gives

$$An^2 + Bn + C = A(n - 1)^2 + B(n - 1) + C + 3n = An^2 - 2An + A + Bn - B + C + 3n \tag{18}$$

or, after cancelling

$$-2An + A - B + 3n = 0 \tag{19}$$

so $A = 3/2$ and $B = A$. Thus

$$T(n) = \frac{3}{2}n^2 - \frac{3}{2}n + C \tag{20}$$

and the initial condition means $C = 1$.

An important example with logs

Now lets consider the example

$$T(n) = cn + 2T(n/2) \tag{21}$$

This will be relevant to algorithms we look at in the course and comes from a divide-and-conquer approach to sorting. Now, telescoping

$$
\begin{align}
T(n) &= cn + 2T(n/2) \tag{22} \\
&= cn + 2c(n/2) + 4T(n/4) \tag{23} \\
&= 2cn + 4c(n/4) + 8T(n/8) \tag{24} \\
&= 3cn + 8c(n/8) + 16T(n/16) \tag{25} \\
&= \ldots \tag{26} \\
&= rcn + 2^r T(n/2^r) \tag{27}
\end{align}
$$

which carries on until $n/2^r = 1$, or $r = \log n$; with a bit of care this gives the solution; here we will use the ansatz

$$T(n) = An \log n \tag{28}$$

so

$$2T(n/2) = 2A\frac{n}{2}\log\frac{n}{2} = An\log n - An \tag{29}$$

and so this solves the approximate equation when $A = c$. In fact by substitution you can see that

$$T(n) = An \log n + Cn \tag{30}$$

solves the equation for any value of C, if we were solving this equation exactly then the initial value would be used to fix C; in this course though we are only interested in the large n behaviour, so this isn't important.