13 Recursion

13.1 Two examples

A routine which calls itself is called *recursive*. For example, this program uses a recursive version of `gcd()`:

```
#include <stdio.h>
#include <stdlib.h>

int gcd ( int m, int n )
{
  if ( n == 0 )
  { return m;}
  else
  { return gcd ( n, m%n ); }
}

int main ( )
{
  int m,n;
  while ( scanf("%d %d", &m, &n) == 2 )
  {
    printf("gcd(%d,%d) is %d\n", m,n,gcd(abs(m), abs(n)));
                //                          ^^^^^^^^^^^^^
  }
                // In the 2/11/21 lecture, the sign correction
                // was omitted, initially.
}
```

It works. The `gcd()` function is recursive and based directly on the fact that `gcd(m,n) = gcd(n,m%n)`. The previous, iterative, version works for the same reason, but it's not so evident.

Another function was shown which adds array elements recursively. Again there was an error initially, though the data didn't show it.

```
#include <stdio.h>
        // The version on slides in the 11th lecture
        // wrongly included a[n].  Changes:
        // if (n<=0), not <0, and add a[n-1] to the total,
        // not a[n].

int total ( int n, int a[] )
{
  if ( n <= 0 )
  { return 0; }
  else
```

```
  { return total ( n-1, a ) + a[n-1]; }
}

int main ( )
{
  int array[1000];
  int count, x;
  count =0;
  while ( count < 1000 && scanf( "%d", &x ) == 1 )
  { array[count] = x;
    ++ count;
  }
  printf("%d numbers total %d\n",
        count, total(count, array));
}
```

Again, this works. It is based on the fact that, assuming $n > 0$,

$$\sum_{i<n} a_i = (\sum_{i<n-1} a_i) + a_{n-1}$$

13.2 How does recursion work?

It works because of the *runtime memory stack* organisation. This will be discussed in another lecture.

13.3 Is this useful?

Sometimes it is, not always. Recursion can give some short, simple, programs. Some examples follow below, and some of them are striking and some of them are unnecessary.

13.4 Printing a decimal number

This one isn't useful, but it is interesting. The idea is that to print `n`, first print `n/10` and then print the last digit, `n % 10`.

```
% cat rec-decimal-print.c
#include <stdio.h>
#include <stdlib.h>

int rec_print( int n ) // if n==0, prints nothing
{
  if ( n > 0 )
  {
    rec_print ( n/10 );
    printf("%d", n%10);
```

```
  }
}

int main ( int argc, char * argv[] )
{
  int n = atoi ( argv[1] );
  if ( n == 0 )  // rec_print does nothing with 0
    printf("0\n");
  else
  {
    rec_print(n);
    printf("\n");
  }
}
% gcc rec-decimal-print.c
% a.out 314159265
314159265
% a.out 0
0
%
```

13.5 Russian Peasant multiplication

This multiplies mn by a process based on the binary representation of n.

```
% cat peasant.c
#include <stdio.h>

int product ( int m, int n ) // assumed nonnegative
{
  if ( n == 0 )
    return 0;
  else
  {
    int p = product ( m, n/2 );
    if ( n%2 == 0 )
    { return p + p; }
    else
    { return p + p + m; }
  }
}

int main ( int argc, char * argv[] )
{
  int m = atoi ( argv[1] ), n = atoi ( argv[2] );
```

```
  printf("%d x %d is %d\n", m, n, product(m,n));
}

% gcc peasant.c
% a.out 9 0
9 x 0 is 0
% a.out 99 9
99 x 9 is 891
% a.out 9 99
9 x 99 is 891
% a.out 12345 678
12345 x 678 is 8369910
%
```

This method of multiplying numbers is not of any practical value, but the idea could be adapted to calculate A^n where A is a square matrix, and that would be efficient.

13.6 Prime factorisation

This is based on the fact that if n is not prime, and m is its smallest diviser ≥ 2, then m is prime and you next factorise n/m.

```
% cat factors.c
#include <stdio.h>
#include <stdlib.h>

int divides ( int m, int n )
{
  return n == 0 || (m != 0  &&  n % m  == 0 );
}

void factorise ( int n )
{
  int m;
  int seems_prime = 1;
  for ( m = 2; seems_prime && m<n; ++m )
  {
    if ( divides ( m,n ) )
    {
      seems_prime = 0;
      printf(" %d", m );
      factorise ( n/m );
    }
  }
  if ( seems_prime )
```

```
    printf(" %d", n);
}

int main ( int argc, char * argv [] )
{
  int n = atoi ( argv[1] );
  printf("factors of %d: ", n );
  factorise ( n );
  printf("\n");
}
% gcc factors.c
% a.out 99
factors of 99:  3 3 11
% a.out 123
factors of 123:  3 41
% a.out 127
factors of 127:  127
% a.out 1073741823
factors of 1073741823:  3 3 7 11 31 151 331
%
```