Lecture 2

Playing with numbers

Integer operators

- Most programming languages provide operators which allow us to break integers into their individual digits.
- In Mathematics these are called the “div” and “mod” operators and they are only used with integers.
- If n = 123456 then n div 10 = 12345 and n mod 10 = 6
- n div 10 divides n by 10 and “forgets” the remainder
- n mod 10 divides n by 10 and gives us the remainder
- General shape
  - (N div x) * x + (N mod x) = N

Integer operators

- In both Python and C we would write “x div y” as x / y

- In both Python and C we would write “x mod y” as x % y

Integer operators

- These operators can be useful in many situations.
- If n is an even number then n % 2 == 0
- If n is an odd number then n % 2 != 0

```
int leap (int y)
        {
        if (((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0))
        {return 1;}
else if (((y % 4 != 0) || (y % 100 == 0)) && (y % 400 != 0))
        {return 0;}
}
```

We could also have written this in a simpler form. Choose whichever
Is easiest for the reader to understand.

```
int leap (int y)
        {
if (((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0))
        {return 1;}
else
        {return 0;}
}
```

Given 2 integers x and y, we want to check whether x is a factor of y, that is, whether x divides into y a whole number of times.

```
int this_divides_that (int x, int y )
{
if ((y % x) == 0)
  { return 1;}
else if ((y % x) != 0)
  { return 0;}
}
```

Now we can use this to write a function which takes in a positive integer and returns the sum of the divisors of this integer.

```
int sum_of_divisors (int n)
  {
  int sum ;
  int i ;
  sum = 0;
  i = 1 ;
while (i != n)
  {
  if (this_divides_that (i, n))
    { sum = sum + i ;}
  else if (!(this_divides_that (i, n)))
    { sum = sum + 0 ;}
  i = i + 1 ;
  }
return sum ;
}
```

Now it becomes very easy to write a function which decides if a positive integer is a prime number.

```
int is_prime (int n)
  {
  if (1 == (sum_of_divisors(n)))
    { return 1;}
else
  {return 0;}
}
```

It is also very easy to write a function which decides if a positive integer is a perfect number.

```
int is_perfect (int n)
{
if (n == (sum_of_divisors(n)))
  { return 1;}
else
  {return 0;}
}
```

The mathematical integer operators, div and mod (/ and % in C) are also very useful for breaking an integer down into its individual digits.

Count the digits

- Suppose we have an integer and we want to count the number of digits in it.

Count the digits

- If a number is less than 10 it will only have 1 digit
- if it is not less than 10 then cut off the right hand digit, count 1 for that and add it to the number of digits in the remainder of the number

Count the digits; Loop

```
int count_the_digits (int n)
        {
        int count = 0 ;

        while (!( n < 10))
                {
                        count = count + 1 ;
                        n = n / 10 ;
                }

count = count + 1 ;
return count ;
}
```

Count the digits

- There is another way to write this function.

- It is based on solving the problem for smaller and smaller sizes of the problem.

Count the digits

```
int digit_count (int n)
  {
if (n < 10)
  {return 1;}
else
  {return (1 + digit_count( n / 10 ));}
}
```

Count the digits

- Note the way the function calls itself.
- This is called a **Recursive** function.
- Recursive solutions are usually very pretty and compact.
- Some programming languages don’t have statements to do loops, instead everything is done by recursion.

Sum the digits

- Often it is easy to modify a recursive solution to a problem to get a solution to a different problem.
- The solutions often have the same “shape”
- Suppose we wanted to add the digits in an integer...

Sum the digits

```
int digit_sum (int n)
        {
if (n < 10)
        {return n;}
else
        {return ((n % 10) + digit_sum ( n / 10 ));}
}
```

Sum the digits; Loop

We can also write this as a loop.

```
int digit_sum_loop (int n)
    {
        int result = 0 ;
        while (!(n < 10))
        {
            result = result + n%10 ;
            n = n/10 ;
        }

        result = result + n ;
        return result ;
}
```

Multiply the digits

- Now suppose we wanted to multiply the digits in an integer.

- The solution will have a similar shape.

Multiply the digits

```
int digit_product (int n)
{
if (n < 10)
        {return n;}
else
        {return ((n % 10) * digit_product ( n / 10 ));}
}
```

Multiply the digits; Loop

We can also write this as a loop.

```
int digit_multiply_loop (int n)
    {
        int result = 1 ;

        while (!(n < 10))
        {
            result = result * n%10 ;
            n = n/10 ;
        }

        result = result * n ;
        return result ;
}
```

A challenge

- Write a program which reads in a date since 1st January 1800 in the form d, m, y (day, month, year) and print out the date 1 day later.

- E.g. 1 3 1903 -> 2 3 1903

- E.g. 30 11 2004 -> 1 12 2004

A challenge

- What might be useful functions to have in solving this?
- How would you make the solution readable for other people?
- Are there any functions you have already written that might be useful to use again/