Lecture 12

Sum of 2 squares

- Suppose you have a positive integer N, we would like to know how many pairs of positive integers $0 \leq a \leq b$ there are where $a*a + b*b = N$. Write a program to find out.

Sum of 2 squares

We will use 2 int variables a and b.
We want to check to see how many pairs of values of a and b give a*a + b*b = N.

Suppose we started with a == 0 and b == N+1
With these values clearly a*a + b*b > N

There is no point in picking a higher value for b and we can not pick a lower value for a, so the values for a and b that we need to check are $0 \leq a \leq b \leq N+1$

Sum of 2 squares

Now, let us consider the expression a*a + b*b

There are 3 possibilities

a*a + b*b < N

a*a + b*b == N

a*a + b*b > N

What should we do with each of these results?

Sum of 2 squares

a*a + b*b < N

We need to make the expression larger, so increase a by 1

a*a + b*b == N

We have a match so add 1 to our result and either increase a by 1 or decrease b by 1.

a*a + b*b > N

We need to make the expression smaller, so decrease b by 1

Sum of 2 squares

```
int main()
{
      int N ;
      int a ;
      int b ;
      int r ;

// assume we have read a value into N, or defined it to be some value

// main program

}
```

Sum of 2 squares

```
a = 0 ;
b = N+1 ;
r = 0 ;

while (a != b)
{
   if (a*a + b*b < N)
      {
        a = a+1 ;
      }
   else if (a*a + b*b == N)
      {
        r = r+1 ;
        a = a+1 ;
      }
   else if (a*a + b*b > N)
      {
        b = b-1 :
      }
}
// a == b. Need final check to see if these values satisfy equation
```

Sum of 2 squares

```
// a == b. Need final check to see if these values satisfy equation

if (a*a + b*b == N)
   {
     r = r+1 ;
   }

printf(“the number of pairs which satisfy the equation = %d: “, r);
```

Sum of 2 squares

In this problem we started at the lower and upper bounds of the values we might want to consider and we gradually moved “inwards” until we had reduced the range of values to a single value.

We have already seen another example like this when we looked at the Binary Chop.

You might like to compare these two programs and see what parts are similar and what parts are different.

Maximum location

We have an array int f[100]

We want to know which location in f contains the maximum value in f.

We could solve this in 2 steps

do a **Reduction** to determine the largest value and then
do a **Linear Search** to find the location which contains that value.

Maximum location

Let us start by looking at two elements f[a] and f[b].

Suppose f[a] <= f[b]

Do we need to consider location a? Could it hold the maximum value?

Suppose f[b] <= f[a]

Do we need to consider location b? Could it hold the maximum value?

Maximum location

So if f[a] <= f[b] we can “forget” location a

and if f[b] <= f[a] we can “forget” location b

We can now use these facts to develop a program

Maximum location

```
int f[100] ;
int a ;
int b ;
// assume f already has values

a = 0 ;
b = 99 ;
while (a != b)
{
   if (f[a] <= f[b])
   {
      a = a+1 ;
   }
   else if (f[b] < f[a]])
   {
      b = b-1 ;
   }
}
// the value in f[a] is the largest in f, so a is the location of the largest value.
```

Maximum location

Once again we have a program which solves the problem
by starting at both ends of the array and moving inwards until
we find the solution.

This way to solve problems is quite common, so it is a nice technique
to learn.

Recursion

- A lot of programming problems can be solved using recursive functions.
- These are functions which can call themselves.
- However, sometimes we might not want to use recursion. So in these cases we need to know what to do.
- First, let us look at some recursive functions

Recursion

count_digit (n) = 1 <= n<10
count_digit (n) = 1 + count_digit (n div 10) <= n>=10

This can be written in C as

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

Recursion

sum_digit (n) = n <= n<10
sum_digit (n) = n mod 10 + sum_digit (n div 10) <= n>=10

This can be written in C as

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

Recursion

product_digit (n) = n <= n<10
product_digit (n) = n mod 10 * product_digit (n div 10) <= n>=10

This can be written in C as

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

Recursion

The general shape of functions like this is

```
f(x)  =  c(x)                 <=  not(b(x))
f(x)  =  h(x) @ f(g(x))  <=  b(x)
```

We would write this is C as follows

```
int f (int x)
{
   if ( !b(x) )
      { return c(x) ; }
   else if ( b(x) )
      { return h(x) @ f( g(x) ) ; }
}
```

Recursion

If we have such a recursive definition
with a shape like these then there is a
standard way to transform it into a loop program

Recursion

Functions with a shape like this

f(x) = c(x) <= not(b(x))
f(x) = h(x) @ f(g(x)) <= b(x)

We can write this as a loop in C as follows

```
r = Id@ ;
while ( b(x) )
{
   r = r @ h(x) ;
   x = g(x) ;
}
r = r @ c(x)

// r = f(x)
```

Recursion

count_digit (n) = 1 <= n<10
count_digit (n) = 1 + count_digit (n div 10) <= n>=10

This can be written in C as

```
r = 0 ;
while ( n>=10 )
{
   r = r + 1 ;
   n = n/10 ;
}
r = r + 1;
```

Recursion

sum_digit (n) = n <= n<10
sum_digit (n) = n mod 10 + sum_digit (n div 10) <= n>=10

This can be written in C as

```
r = 0 ;
while ( n>=10 )
{
   r = r + (n%10) ;
   n = n/10 ;
}
r = r + n ;
```

Recursion

product_digit (n) = n <= n<10
product_digit (n) = n mod 10 * product_digit (n div 10) <= n>=10

This can be written in C as

```
r = 1 ;
while ( n>=10 )
{
   r = r * (n%10) ;
   n = n/10 ;
}
r = r * n ;
```

Recursion

This is one of a set of techniques for transforming recursive functions into loop programs.

Knowing techniques such as this is one of the skills that good Computing Scientists know.

Menu driven systems

- On many current systems you click on the toolbar on the top of the window and a list of different functions is presented.
- You click on the one you want to choose it.
- On earlier systems a simple menu which normally had the names of the functions and a number associated with each one was presented to the user.
- They then typed the number to get the system to perform the function.

My System

1 Create new account

2 Edit existing account

3 Delete existing account

4 Print all account details

5 Quit system

Please enter choice (1..5) :

The algorithm behind this menu system would have the following structure.

```
“Display menu”
“Get user’s choice”
While  ( choice != 5 )
{
        if ( choice == 1 )
            { create_account () ; }
        else if ( choice == 2 )
            { edit_account () ; }
        else if ( choice == 3 )
            { delete_account () ; }
        else if ( choice == 4 )
            { print_account_details () ; }
        “Display menu”
        “Get user’s  choice”
}
```