RECURSION – II

In this section we shall study some more issues related to recursion.

We shall study
the development of algorithm of Towers of Hanoi problem,

trace a recursive function

and finally try to improve upon the time complexity of the Fibonacci sequence generation.

TOWERS OF HANOI

We have seen that the Tower of Honoi problem for 3 disks can be worked as follows:

move tower of size 2 from **start** to **temp**
move a single disk from **start** to **finish**
move tower of size 2 from **temp** to **finish**

If there were 2 disks, then the movement sequence would be as follows:

move tower of size 1 from **start** to **temp**
move a single disk from **start** to **finish**
move tower of size 1 from **temp** to **finish**

If there were 4 disks, then the movement sequence would be as follows;

move tower of size 3 from **start** to **temp**
move a single disk from **start** to **finish**
move tower of size 3 from **temp** to **finish**

In every case, we initially simplify the problem to move tower of size one less than the given size to the temporary needle.

Then the last disk is moved to the finish needle.

Finally, we attempt to solve the problem of moving the tower on temporary needle (of size one less than given size) to the finish needle.

Thus the general case for n disks can be written down as ;

move tower of size n-1 from **start** to **temp**
move a single disk from **start** to **finish**
move tower of size n-1 from **temp** to **finish**

TRACING A RECURSIVE FUNCTION

We have traced earlier the factorial recursive function for n=4.
Here we study another recursive function, and trace it when it is called as f(1,3):

```
int  f (int x, int y)
{
if (x == 0  && y >= 0)
        return y + 1;
else if (x > 0 && y == 0)
        return (f(x-1,1));
else if (x > 0 && y > 0)
        return (f(x-1, f(x, y-1)));
}
```

Trace for call f(1,3)

x

= f(1, 3)

= f(0, f(1, 2))

= f(0, f(0, f(1, 1)))

= f(0, f(0, f(0, f(1,0))))

= f(0, f(0, f(0, f(0,1))))

= f(0, f(0, f(0, 2)))

= f(0, f(0, 3))

= f(0,4)

= 5

A BETTER ALGORITHM FOR GENERATION OF Fibonacci Sequence:

We have seen that the algorithm proposed earlier had an exponential time complexity. However, if we exploit the structure of the problem we can considerably reduce the time , while the solution still remains recursive.

In fact a large number of sequences can be generated based on choice of the first two numbers.

•If you were to select the first two terms as 2 and 4 the sequence you would generate would be:

2, 4, 6, 10, 16, 26, 42, 68, 110, 178, 288, 466,…

•The general class of sequences which follow this pattern are called additive sequences.

•Using the concept of an additive sequence it is possible
to convert the problem of finding the n^{th} term in the Fibonacci sequence
into the more general problem of

finding the n^{th} term in an additive sequence whose first two terms are t_0 and t_1 .

•An additive sequence function requires three arguments:
the number of terms of interest in the series, and the first two terms in the series.

•The C prototype for this function :

```
int AdditiveSequence(int n, int t0, int t1);
```

•Given such a function, the Fibonacci series can be generated as shown below:

```
int fib(int n)
{
    return (AdditiveSeq(n, 0, 1));
}
```

•The body consists of a single line of code
that does nothing but call another function,
passing along the additional arguments.

•Functions of this sort, are called wrapper functions. Wrapper functions are very common in recursive programming

•A implementation in C of the AdditiveSequence is given below:

```
int  AdditiveSeq(intn,  int  t0,int
t1))
{
    if (n == 0)   return (t0);
    if (n == 1)   return (t1);
  return AdditiveSeq(n-1,1,t0+t1));
}
•
```

•Using this AdditiveSeq function, let's determine the value of Fibonacci(6).

fib(6)
= AdditiveSeq (6, 0, 1)
= AdditiveSeq (5, 1, 1)
= AdditiveSeq (4, 1, 2)
= AdditiveSeq (3, 2, 3)
= AdditiveSeq (2, 3, 5)
= AdditiveSeq (1, 5, 8)
= 8

•Notice how much more efficiently the recursion occurs using the AdditiveSeq function.

The table on the next page illustrates this further.

Series Size	Total Number of Calls	
	Original Function	New Function
1	1	2
2	3	3
3	5	4
4	9	5
5	15	6
10	177	11
15	1,973	16
20	21,891	21
25	242,785	26
30	2,692,573	31
35	29,860,703	36
40	331,160,281	41