RECURSION – IV

Finding square root of a number
Generating Permutations
Additive sequence

Finding Square Root of a number:

We can use the Newton's method, which updates a given guess. Let us take an example. Let the given number be 36. Let us guess that its root is 9. Now since 36/9 is 4, it is certain that the new value of the root has to lie between 9 and 4, as their product equals the given number. So the best way to make the next guess would be to choose a number lying somewhere in the middle of the range (4, 9). This would be (4 + 9)/2, i.e. 6.5 which is more closer to the actual root 6 than the first guess 9 was. We can proceed further by computing the next factor 36/6.5 and then taking the mid value of this factor and 6.5. We see that we very rapidly approach 6, the true value of the root. The steps can be repeated till we find that the current step did not produce appreciable change in the value of the root.

Thus given a number x, the first guess could be r = x/2.
We keep on generating and testing successive approximations of r until we find the value that is close enough to stop. The updating process may be terminated when
| r * r – x | < epsilon.
A recursive solution to this problem would need the routine to be called repeatedly with the new value of the root. The stopping condition for the recursive function is already indicated above.

```
#include <stdio.h>
double sqroot(double num, double root);

int main ( )
{
 double x,r;
printf("\n enter x=");
scanf("%lf", &x);
r = x/2;
r = sqroot(x,r);
 printf("\n root =  %10.2f",r);
printf("\n");
}
double sqroot(double num, double root)
{
  double newroot,error;
  error = root*root - num;

  if ( abs(  error )<= 0.00001)
        return root;
  else {
        newroot= 0.5*(root + num/root);
        return sqroot(num,newroot);
          }
}
```

Generating Permutations:

Permutations are often needed in games like Scrabble, where we want to consider all possible rearrangements with a given set of letters.

Different permutations of a given string can be generated by keeping the first letter fixed and recursively finding permutations of remaining letters. Then the first letter can be exchanged with one of the other letters and the process repeated.

Consider the string CD.
It has only two permutations obtained by exchanging the characters :
CD
DC

Any string of 2 characters would have 2 possible permutations.

Next, consider the string BCD.
Keep B fixed and permute CD to yield
BCD
BDC

Get back to original sequence BCD. Then exchange B with C to yield CBD.
Now keep C fixed and permute BD to yield
CBD
CDB

Get back to original sequence BCD. Finally exchange B with letter D to yield DCB.
Now keep D fixed and permute CB to yield
DCB
DBC

Thus a string of 3 characters has 6 possible permutations. A string of 4 characters has 12 possible permutations. For a string of length n , the number of permutations are n!

To generate all possible permutations for a string of n characters the following recursive algorithm can be used:
Keep the first k letters fixed and permute the remaining. Start with k = 0 and permute the remaining. For each permutation, increase k and continue to permute using a *for* loop. Print the characters when k equals n.

•The C prototype for this function :

```
void ListPermutations( char str[] )
{
       RecursivePermute( str, 0 );
}



void RecursivePermute( char str[],   int k  )
{
     int j;
     if ( k ==  strlen(str))

          printf("%s\n", str);
      else {
            for(j= k;   j< strlen(str);  j++)
             {
                 ExchangeCharacters(str, k,j);
                 RecursivePermute( str, k+1 );
                 ExchangeCharacters(str,  j,  k);
             }
            }
}
```

Here **ExchangeCharacters(str, k,j) is a routine which swaps the character at str[k] with that located at str[j].**

An Efficient Algorithm for Fibonacci Sequence:

We have seen that the algorithm proposed earlier had an exponential time complexity. It is because each call generates two recursive calls to the same function. However, if we exploit the structure of the problem we can considerably reduce the time, while still using a recursive solution.

$S_n = S_{n-1} + S_{n-2}$
For Fibonacci sequence, the first two numbers are $S_0 = 0$ and $S_1 = 1$.

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 following sequence would be generated:

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.

•a 'C' prototype for this function :

```
int AdditiveSeq(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 ‘C’ implementation of the AdditiveSequence function is given below:

```
int AdditiveSeq(int n, int t0,int t1))
{
    if (n == 0)   return (t0);
    if (n == 1)   return (t1);
    return(AdditiveSeq(n-1, t1,  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 in the AdditiveSeq function.

The time to find nth Fibonacci number reduces to that of finding the (n-1)th number, with extra time needed for one addition operation and one subtraction operation:

$T(n) = T(n – 1) + 2$

What is the time complexity of this recurrence relation?

The following table illustrates this point further, where the number of function calls for the original Fibonacci recursive function are compared with the number of function calls made by the additive sequence.

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