Ayushi Kalantri

Chapter 6

Recursion

6. 1 Introduction

Recursion is a fundamental concept in mathematics and in computer science. Functions in mathematics and subprograms in a program are often defined recursively. In many programming languages (including "C") subprograms are called functions. From the perspective of a programming language, a function is called recursive if the function calls itself. This feature of calling the same function from itself was not supported in older programming languages like Fortran-77, or GW-Basic. But almost all modern programming languages do support recursion. The reason why recursion has become more popular among computer scientists is that it is easier to define algorithms recursively. Such definitions are sometimes elegant and are more amenable to mathematical manipulations for the purpose of verification and analysis. However, it must be borne in mind that recursive definition of an algorithm does not always provide the most efficient solution. In fact, it may be noted that all recursive algorithms can be converted to an equivalent non-recursive version and often the non-recursive versions are preferred because of reasons related to performance.

Understandably, if a function calls itself unconditionally, then no function after being called would ever be able to return and an infinite sequence of calls to the same function would be produced. Such functions will not terminate. Hence recursive functions usually calls itself on a condition. On some other conditions (known as terminating conditions) the recursive functions terminate enabling its caller to proceed. A useful class of algorithms known as divide and conquer algorithms has been evolved which exploits the concept of recursion.

Recursive algorithms have already been developed in previous chapters of this book. This chapter introduces concepts and tools for designing and analyzing the

basic structures of recursive algorithms. In doing so, several illustrative problems and their solutions have been explained in detail. This chapter also deals with techniques to remove recursion and ways to analyze recursive algorithms.

6.2 Simple Recursion and Recursion Tree

Most of the readers are familiar with recursive functions for problems like computing the factorial of an integer or finding the n-th number of a Fibonacci series. Consider the definition of the factorial of a positive integer "n" as given in the following.

$$\text{Factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * \text{Factorial}(n-1), & \text{if } n > 0 \end{cases}$$

Note that the function "Factorial" is defined in terms of itself for n > 0. This is why the function "Factorial" is said to be defined recursively. Converting recursive definition of a function to an algorithm is often straightforward. The function "Factorial" can easily be converted to the following algorithm.

```
int factorial (int n)
{
   if (n == 0)
         return 1;
   return (n*factorial (n – 1));
}
```

It is easy to verify that the number of multiplications required by the function "factorial" for an integer "n" is "n". Also note that a simple iterative (non-recursive) function can be written for computing the factorial which is presented in the following.

```
int iterativeFactorial (int n)
{
   int i, fac = 1;
   for (i = 0; i<n; i++)
         fac *= (i+1);
   return fac;
}
```

If the function "factorial" is invoked with an actual parameter 5 then, as a result, factorial (4) is invoked. Once "factorial(4)" is called, "factorial(3)" is called and so on. This process continues until "factorial(0)" is called which returns immediately

without calling any other function. The sequence of calls can be depicted pictorially as shown in Figure 6.1. Each node in the diagram represents a call to factorial and the label of the node denotes the value of the actual parameter.

An edge from a node "x" to another node "y" in Figure 6.1 signifies the fact that the call to "factorial" represented by node "x" necessitates another call to "factorial" represented by node "y". Once the call to "factorial(0)" is over, control goes to the caller function which is the suspended function "factorial(1)" in this case. The curved arrows in Figure 6.1 denote the return of a call to the "factorial" function. The number labeling a curved arrow denotes the value returned by the called function. The necessary tasks (a multiplication in this case) are performed and the function "factorial(1)" returns the value "1" to its caller, "factorial(2)". In "factorial(2)", "2" is multiplied with the value returned by "factorial(1)" (i.e., 1) and the result "2" is returned to its caller "factorial(3)". This process continues until control returns to "factorial(5)" where 5 is multiplied with the value returned by "factorial(4)" and the result is returned to the caller.

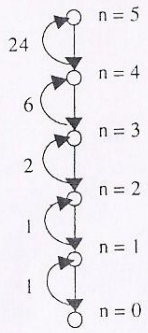

Figure 6.1 Sequence of recursive calls generated when "factorial (5)" is called. The circles denote processing of one particular call to "factorial". The value of the parameter "n" as passed to this call is labelled against each circle. The arrows denote a call to "factorial" from within the function "factorial" itself. Such a diagram is often referred to as a "recursion tree".

Note that time complexities of both the iterative and recursive algorithms for computing the factorial of an integer are the same. At a first glance, it may appear that the function "iterativeFactorial" requires more space for execution than the function "factorial" because the iterative function requires space to store values of an integer "i". However, this observation is not true. Recall that a call to "factorial(5)" gave rise to five successive calls to the same function "factorial". Whenever a subroutine is called, the context of the caller at the point of the call must be remembered so that

when control comes back to the caller, the caller can smoothly resume its operations. Moreover, in each call to a function, space for all local variables (including the formal parameters) has to be reserved. Therefore, to execute a call to "factorial(5)", space is to be reserved for "n" for all functions which are called. Consequently, it is not true that the function "factorial" is not using any space other than the space required to store the value of "n". In fact, a call to "factorial(5)" requires space for storing five integers, each one being used for storing the actual parameter needed to recursively call the function "factorial". More surprisingly, the actual time taken by the recursive algorithm is considerably greater than that of the iterative algorithm although the time complexities of both these algorithms are the same. Thus, it may not be easy to say that a recursive algorithm always performs better. This is further elaborated in the following example.

The notion of Fibonacci Sequence of integers has already been discussed in Chapter 1. A recursive algorithm for finding the n-th integer in the Fibonacci Sequence has also been described. Recall the algorithm in the present context.

```
int Fibo (int n)
  {
        if (n ==0)
              return 0;
        if (n ==1)
              return 1;
        return (Fibo (n-1) + Fibo (n-2));
  }
```

Assume that "Fibo(5)" is invoked. To evaluate "Fibo(5)", "Fibo(4)" and "Fibo(3)" are to be invoked. The sequence of calls to the function "Fibo" may be depicted in the same way as the sequence of calls to the function "factorial" was shown in Figure 6.1.

In fact, it is possible to draw a diagram such as Figure 6.2 for depicting sequence of recursive calls for any recursive function. Such pictorial representation of calls to recursive functions is called a "recursion tree", which is very useful in analyzing recursive algorithms.

As it was the case in Figure 6.1, each node in the Figure 6.2 represents a call to the function "Fibo". Each node is labelled with the value of "n", which is the actual parameter for this call. As before, an edge from a node "x" to nodes "y_1", "y_2" signifies that a call to "Fibo" corresponding to node "x" recursively calls "Fibo" corresponding to "y_1" and "y_2". Moreover, the edges are ordered from left to right so that a call represented by the left most edge is made first and then the call represented by right edge is made. For example, to complete a call to "Fibo(5)", a call to Fibo(4) is made first and then a call to Fibo(3) is made.

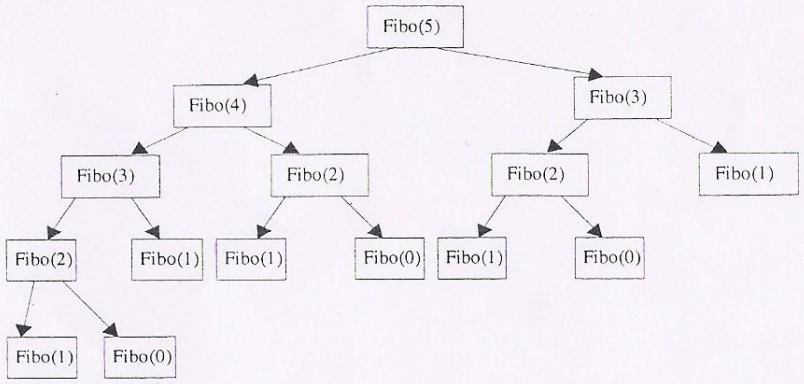

Figure 6.2 Recursion tree for the function "fibo" when called with an actual parameter "5".

The structure of the recursion tree highlights important characteristics of the underlying recursion. For example, it is evident from the figure that the function "Fibo" is called several times with the same actual parameter. Note that, "Fibo(3)" is called twice and "Fibo(2)" is called three times. It would be probably more efficient if the values of "Fibo(n)" are remembered and reused without reevaluating it for a number of times. This observation actually points out why the recursive algorithm is far less efficient than an iterative solution to the same problem. Recall that the time complexity of the algorithm "Fibo" is $O(\phi^n)$ for a constant ϕ while the time complexity of a simple iterative solution (as given in Chapter 1) is just O(n).

6.3 Divide and Conquer

Divide and conquer class of algorithms have already been discussed in Chapter 3 in the context of polynomial multiplication. The basic idea behind this class of problems may be summarized in the following.

DivideAndConquer (problem, "p", of size "n")

{

if the size "n" is small enough

Solve the problem and return.

Split the problem to sub-problems p_0, p_1, ..., p_{m-1} so that size of each p_i is less than "n".

For each i (from "0" to "m-1")

Call DivideAndConquer(p_i) to obtain a solution s_i.

Extend solutions s_0, s_1, ..., s_{m-1} to obtain the overall solution.

}

The words printed in *italics* in the above description represent non-trivial operations. In fact, designing a divide and conquer algorithm often boils down to defining these actions. Many algorithms arising in the domain of sorting and searching belong to this class. They are described in Chapter 9. Various tree and graph algorithms also belong to this class. They are described in Chapter 7 and Chapter 10 respectively.

6.4 Tower of Hanoi Problem

A French mathematician Edouard Lucas invented a cute puzzle called the "Tower of Hanoi" in 1883. The problem is as follows. A tower of eight disks is initially stacked in decreasing size on one of three available pegs as shown in the Figure 6.3. The goal is to transfer the entire tower of disks from one peg (say, "A") to another peg (say, "B") using the third available peg (say, "C") so that only one disk can be transferred at one time and a smaller disk can never be placed on a larger disk during this process of transfer. In fact Lucas attached a legend with his puzzle by associating it with the famous temple of Brahma. The tower of Brahma contains 64 golden disks placed on one of three diamond needles. Lucas said that at the time of creation of the universe God placed the disks on the first needle and subsequently asked some priests to move the disks from the first to the third needle using the second needle subject to the two conditions already mentioned. It was believed that the day priests would finish the task, the world would come to an end.

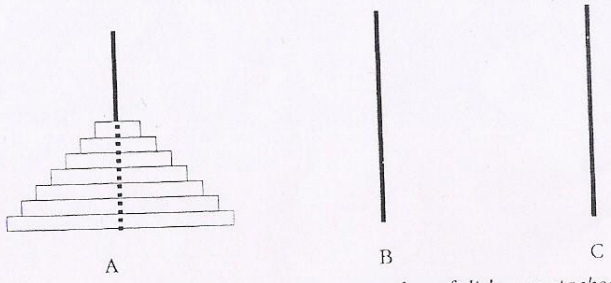

Figure 6.3 The Tower Of Hanoi problem. A number of disks are stacked on Peg "A" in decreasing order of size. They are to be transferred to Peg "C" using Peg "B" satisfying two conditions mentioned in the text.

Consider that the problem is to write a program to find out the moves required to transfer "n" disks from peg "1" to peg "3" using peg "2". Note that the problem is trivial if "n" is "1". Then, the disk is moved directly from peg "1" to peg "3". Assume

that a solution to the problem of transferring (n-1) disks from one peg to another using a third auxiliary peg is known to us. In other words, the solution is assumed for a smaller value of n. Then, the upper (n-1) disks may be assumed to be moved from peg "1" to peg "2" using peg "3". The largest disk which is remaining on peg "1" can be directly moved to peg "3". Once this is done, the (n-1) disks can be moved from peg "2" to peg "3" using peg "1". Note that the above description extends a solution for transferring (n-1) disks to a solution for transferring "n" disks. Thus, a recursive definition of the problem is obtained. Let move (n, s, d, u) denote moving "n" disks from peg numbered "s" to peg "d" using peg "u". Then "move" function can be recursively defined as follows.

$$\text{move}(n, s, d, u) = \begin{cases} \text{move}(n-1, s, u, d) \text{ and} \\ \text{move the n-th disk from peg "s" to peg "d" and} \\ \text{move}(n-1, u, d, s) & \text{,if } n>1 \\ \text{move the disk from peg "s" to peg "d"} & \text{,if } n=1 \end{cases}$$

The "C" function for this problem is described next.

```
void move (int n, int s, int d, int u)
{
    if (n == 1)
    {
        printf ("move the disk from %d to %d\n", s, d);
        return;
    }
    move (n-1, s, u, d);
    printf ("move the disk from %d to %d\n", s, d);

    move (n-1, u, d, s);
}
```

Note that a non-recursive formulation of the above problem is not at all easy. Recursion is particularly suitable for a problem like this. To analyze the time complexity of function "move", note that a call to "move" results in two calls to the same function and some fixed task (like printing a line). If T(n) denotes the time required by "move" if it is called with an actual parameter "n" then T(n) can be expressed recursively as follows.

$$T(n) = \begin{cases} T(n-1) + C + T(n-1) & \text{,if } n > 1 \\ C & \text{,if } n = 1 \end{cases}$$

where C is a constant.

Thus, $T(n) = 2T(n-1) + C$ for $n > 1$

$= 2(2T(n-2) + C) + C$

$= 2^2T(n-2) + C(1 + 2)$

$= 2^3T(n-3) + C(1 + 2 + 2^2)$

.

.

.

$= 2^{n-1}T(n - n + 1) + C(1 + 2 + 2^2 + \ldots + 2^{n-2})$

$= 2^{n-1}C + C(1 + 2 + \ldots + 2^{n-2})$

$= C.2^n$

$= O(2^n)$

Consider the trace of the algorithm when three disks are initially placed on peg "1" and these disks are to be moved to peg "3" using peg "2". That is, "move(3, 1, 3, 2)" is called. See the recursion tree as shown in Figure 6.4(a). In turn, a call is made to "move(2, 1, 2, 3)". Subsequently, a call is made to "move (1, 1, 3, 2)". Consequently, the first line of the output of Figure 6.4(b) is generated and the function call returns to the caller (i.e. in the function call "move(2, 1, 2, 3)"). The second line of output is produced and the call "move(1, 3, 2, 1)" is made. This call returns to its caller (which is the suspended function "move (2, 1, 2, 3)") after printing the third line of the output as shown in Figure 6.4(b). The function "move(2, 1, 2, 3)" is now complete and it returns to its caller which is the function "move(3, 1, 3, 2)". This process continues until the function "move(3, 1, 3, 2)" returns. Readers are encouraged to trace the remaining flow of control through the recursion tree and to verify that the output generated by a call to move(3, 1, 3, 2) is in deed as given in Figure 6.4(b).

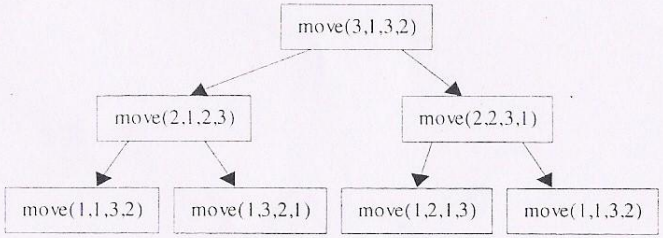

Figure 6.4 (a) Recursion tree for the function "move".

move the disk from 1 to 3
,, ,, ,, ,, 1 to 2
,, ,, ,, ,, 3 to 2
,, ,, ,, ,, 1 to 3
,, ,, ,, ,, 2 to 1
,, ,, ,, ,, 2 to 3
,, ,, ,, ,, 1 to 3

Figure 6.4(b) The lines of output generated by a call to "move(3, 1, 3, 2)".

6.5 The Permutation generation

Consider another interesting problem of printing all possible permutations of integers 1, 2,..., n. Clearly, there are n! such permutations. To illustrate the idea of recursively generating permutations, carefully study the diagram in Figure 6.5. This diagram is a tree where the top node represents the permutations of a set of integers containing only one element, "1". Each node in the tree represents a permutation. The tree may be horizontally divided into four levels as shown in the figure. The nodes at the i-th level represent permutations of integers 1, 2,..., i. Assume that the generation of permutations start from the level 1 of the tree. There is only one node at level "1". This node represents the only permutation possible with one integer "1". That permutation is denoted by (1). To obtain the nodes in the next level, just insert "2" to all possible positions of the permutation represented by its parent node. Each permutation obtained by such insertion is represented by nodes in the 2nd level. For example "2" can be placed in two positions in the list (1) producing the lists (1, 2) and (2, 1). These two permutations are the two nodes in level 2 and they are the possible permutations of integers 1 and 2. Take all the nodes from the second level one by one. Put "3" in all possible positions of the permutation represented by the node currently taken. The resultant permutations

are the ones achievable from the set of integers 1, 2, 3. Thus, the tree provides a systematic way of generating permutations which can be formally summarized as follows. Consider a function "Permutation(k)" which generates the list of all permutations of the set of integers {1, 2, 3,..., k}.

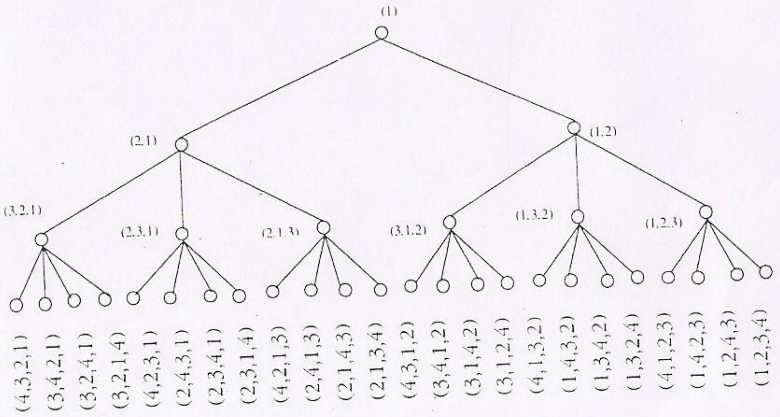

Figure 6.5 Tree for generating permutations

Permutation(1) is obvious and it returns the list (1). Permutation(k) can be defined in terms of Permutation(k-1) in the following way. Let "L" be the list generated by Permutation(k-1). Each element of "L" is a permutation. So, an element in "L" is once again a list of all integers from 1 to k-1. Insert "k" at all possible positions to each element in "L". Note that the elements in "L" contain all integers from 1 to k-1 and hence insertion of k increases the size of these elements. "k" can be inserted to such a list in "k" different positions. So, from each element of "L", a new set of "k" elements is generated. Moreover, as the elements in "L" are distinct, the resultant elements after insertion of "k" also remain distinct. To illustrate these ideas, assume that Permutation(2) is given and Permutation(3) is to be generated. That is, k = 3, L = ((1, 2), (2, 1)). So, "L" contains two elements (1, 2) and (2, 1) each of which is a list. Take the element (1, 2). Insert "3" in all possible positions of (1, 2). There are "3" positions and 3 new elements are generated from (1, 2) which are (3, 1, 2), (1, 3, 2) and (1, 2, 3). Take the other element of "L" which is (2, 1). "3" may be inserted in three different positions of (2, 1) yielding (3, 2, 1), (2, 3, 1) and (2, 1, 3).

These ideas can be easily translated to the following recursive algorithm. However, there is a problem of finding an appropriate return data type for the function. In the definition of "Permutation", it is assumed that the result of the function is a list of permutations, i.e. a list of list. Moreover, the definition freely expands the permuta-

tions which are lists themselves. To eliminate the burden of passing a list of list to each function call, a global list is maintained. As the permutation generation process starts with "1" and continues indefinitely, a termination condition is required. A global integer "n" is kept for this purpose. If "k" is equal to "n" then recursion is terminated. The algorithm is presented next.

```
Algorithm PrintPermutations(int k)
{
  /* Let L denote one permutation of the set of integers 1, 2,..., k-1. */
        for each possible position in "L"
        {
              insert "k" to "L"
              /* Naturally, size of L grows */
              if(k == n)
                    print "L";                    /* reursion terminates */
              else
                    PrintPermutations(k+1);       /* recursive call */
              remove "k" from "L"
        }
}
```

If all permutations of integers 1, 2,..., 10 is required to be printed then "PrintPermutations(1)" is to be called after assigning 10 to "n" and "L" is initialized to NULL. Trace the algorithm when it is called with k = 1 and the value of the global integer "n" is 4. As, "L" is empty, there is only one place to insert. So, "L" becomes (1) after inserting "1" to an empty list. As k ≠ n, the same function is called with k = 2.

When "PrintPermutations(2)" is called with L = (1), there are two positions in "L" where "k", i.e., 2, can be inserted. For the first position, when "2" is inserted to "L", "L" becomes(2, 1). Once again, as "k"(i.e., 2) is not equal to "n"(i.e., 4), "PrintPermutations (3)" is called. Only after this call returns, this iteration of the for loop will be finished and the next iteration will start. The sequence of calls till now is shown in Figure 6.6(a).

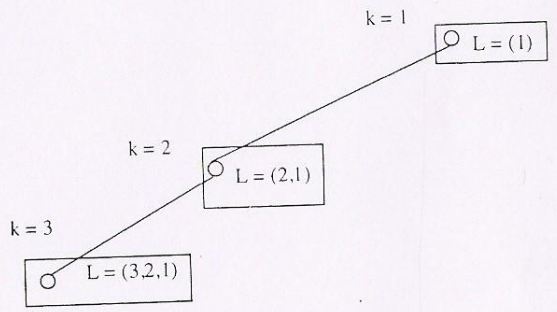

Figure 6.6(a). In this recursion tree, each box represents one instance of a call to printPermutations. The value of "k" is written against each box. Inside a call (i.e., a box), each iteration is shown as a circle. The value of "L" at the time of entering into the "for" loop is shown against each circle.

Now, "PrintPermutations(3)" is called with L = (2, 1) There are three possible positions to insert "3" into "L". The first possibility is to add "3" to the beginning of "L". In the first iteration of the "for" loop, "L" becomes (3, 2, 1). As k ≠ n, the same function is once again called with k = 4 suspending the remaining task in this "for" loop.

When "PrintPermutations(4)" is called with L = (3, 2, 1), there are four possible positions to insert "4". In the for loop, insertion of "4" to "L" is done. As the condition (k == n) becomes true now, no more recursive call results. Instead, "L" is being printed in each iteration of the for loop. Consequently, the following four lines would be printed as a result of this call to "PrintPermutations(4)".

4, 3, 2, 1
3, 4, 2, 1
3, 2, 4, 1
3, 2, 1, 4

At the end of each iteration, "4" is removed from the expanded "L" and so, this call returns to its caller with "L" remaining as (3, 2, 1). Once the control comes back to the previous instance of the call "PrintPermutation(3)" the suspended iteration of the for loop of this function becomes complete after removing "3" from "L". In the next iteration, "3" is inserted in the next position so that "L" becomes (2, 3, 1). The sequence of calls at this stage is shown in Figure 6.6(b). Once again "PrintPermutation(4)" is called with this "L". As k == n in the call "PrintPermutation(4)",

the following outputs will be generated.

4, 2, 3, 1
2, 4, 3, 1
2, 3, 4, 1
2, 3, 1, 4

Once "PrintPermutations(4)" has been called from all three iterations of the for loop of the current instance of the call "PrintPermutations(3)" with L = (2, 1) control goes back to the call of instance of "PrintPermutations(2)" with L = (1).

Once control comes back to the call "PrintPermutations(2)" witl L = (1), the first iteration of the for loop can be completed and the next iteration begins. In the next iteration "L" becomes (1,2) after inserting "2" to "L". The recursion tree upto this point has been drawn in Figure 6.6(c). Once again, "PrintPermutation(3)" is called with this "L". And the process continues. Readers are encouraged to manually complete this process of constructing the recursion tree.

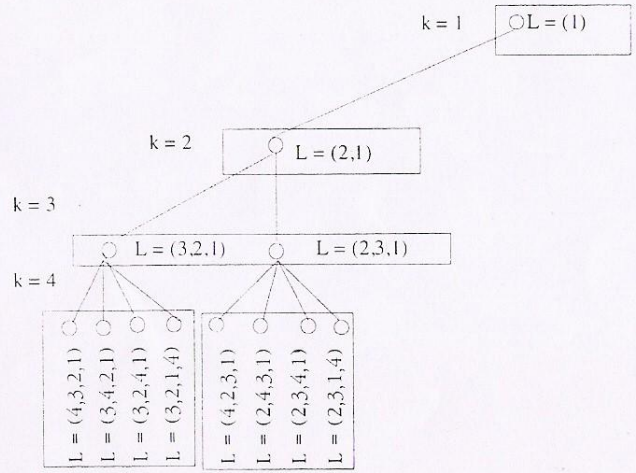

Figure 6.6(b) Partial Recursion tree where each box represents a call to the function "printPermutataions" and each circle inside a box represents one iteration inside a call as explained in Figure 6.6 (a). The second iteration in the call "PrintPermutations(2) is complete at this stage.

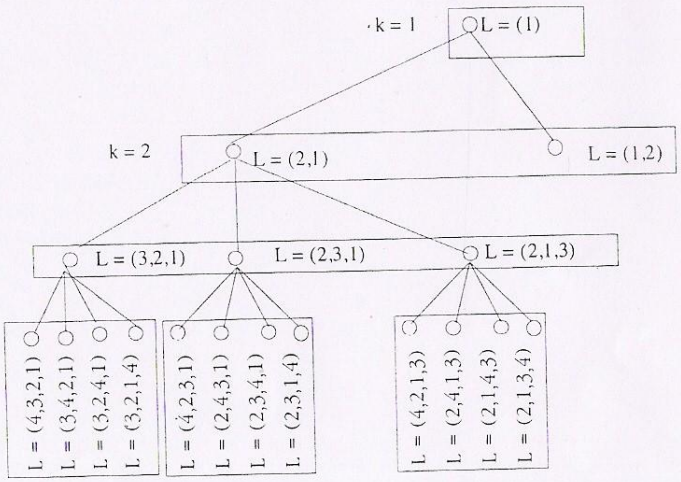

Figure 6.6(c) Partial Recursion tree where each box represents a call to the function "PrintPermutataions" and each circle inside a box represents one iteration inside a call as explained in Figure 6.6(a). The first iteration in the call PrintPermuutations(2) is complete and the second iteration has just begun at this stage.

"C" Function for Permutation

The only non-trivial part of converting the algorithm "PrintPermutations" to a "C" function is deciding upon a data structure of the global list "L" and writing "insert" and "delete" routines for that list. If the list is implemented using an array then the insertions and deletions become inefficient. So, a linked representation of "L" becomes an automatic choice. However, an additional advantage in this case is that the maximum length of the list is known, which is "n". Since this length of the list "L" is known a priori, the linked list representation may be conveniently implemented using arrays. Recall that one step of the algorithm involves insertion of "k" to "L" in all possible positions. This implies that "k" is to be inserted at the beginning (i.e. before the first node) of the list and as well as "k" has to be inserted after every node of "L". To make things uniform it is better to have a dummy node in "L" so that "k" is always inserted after some node.

Array implementation of linked lists has been discussed in Chapter 3. Such an implementation typically requires two arrays: one for storing the data and the other to maintain the links. In this case, however, two arrays are not required. Interestingly, both the data and the links can be managed in one array "Link". The trick is

to use the indices of the array as data elements and the contents of the array as links. Thus, the list (0, 3, 2, 1) can be represented as shown.

3	0	1	2							

By default, the first index of the array is "0" which is its first data element. This is correct because "0" acts as the first dummy element of every list. Link to the next element is found in Link[0] which is "3". Since the value of the index is the value of the data, the data of the second element is "3". Link to the next element is to be found in Link[3] which is "2". Once again, the third element in the list is "2". This process continues until Link[k] becomes "0" for some "k".

In such a representation, adding "k" after the node "p" can be done by setting Link[k] to Link[p] and then setting Link[p] to "k". Deletion of "k" which occurs after "p" is also straightforward. Just set Link[p] to Link[k] and the original link from "p" to "k" vanishes. With this background, the "C" function of the algorithm "PrintPermutations" is easy to obtain.

```
void PrintPermutations(int k)
{
   int p = 0;
   do
   {
         Link[k] = Link[p];                    /* inserting k */
         Link[p] = k;
         if (k == n)
               print();
         else
               PrintPermutations(k+1);
         Link[p] = Link[k];                    /* deleting k */
         p = Link[p];
   }     while(p != 0);          /* the loop to insert k in al positions of L */
}
```

Before this function is called, the array "Link" has to be defined globally as

```
int Link [20];
```

and "Link" must be initialized to contain just the dummy element "0" by the following assignment.


```
Link [0] = 0;
```

"Print" is a simple function to traverse the list and print its elements as given in the following:

```
void print( )
{
      int k = 0;
      while(Link[k] != 0)
      {
            printf("%d", Link[k]);
            k = Link[k];
      }
      printf("\n");
}
```

6.6 Backtracking: Eight Queens problem

This is a very famous puzzle in which eight queens are to be placed on a 8X8 chessboard so that no two queens may attack each other. Note that a queen can attack another piece if they lie on the same horizontal or vertical line or on the same diagonal in either direction. This is not a very easy problem to solve because there are so many ways to place the queens that an exhaustive computation is not humanly possible. The diagram in Figure 6.7 illustrates one configuration of the chess board where eight queens are placed in non-attacking positions. However, it must be stated that this is not the only solution configuration for eight queens" problem.

Analyze the problem a little closely. A chess board may be represented by a matrix (i.e., a two dimensional array) "chess" so that each cell in the chess board may be represented by an element of the matrix. All elements of the array may be initialized to zero. An element of the array could be set to "1" whenever a queen is placed on the corresponding cell.

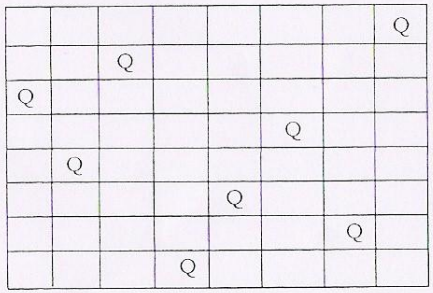

Figure 6.7. A solution configuration where eight queens are placed on a chessboard so that they can not attack each other.

That is, if chess[i][j] = 1 then, it is understood that a queen is placed on the j-th column of the i-th row of the chess board. It can be easily seen that two queens can not be placed on the same row in a solution configuration because, in that case, they would be in an attacking position with respect to each other. Thus, it is known that each row in the array contains exactly one queen in the solution configuration. However, the column number where the queen for the i-th row is going to be placed is not known. Thus a one-dimensional array "column" is sufficient as a suitable data structure. If column[i] = "j" then it is understood that the queen of the i-th row is placed on the j-th column. With this information, try to think of the algorithm. First, put the queen at the first row. That is, set column[1] to "1". Now, the queen at the second row can not be placed on the first or second column. So, set column[2] to "3". Next, column[3] can not be set to 1, 2, 3, 4. It can only be set to "5". column[4] may be set to "2". column[5] may be set to "4". Now, it is impossible to place the queen in the 6-th row in a non-attacking cell. This situation is shown in Figure 6.8.

So, it is required to revisit the placement of queen in the 5-th row and put it to a new column. This means that a back tracking is needed to solve this puzzle. So, a first description of the algorithm may be described as follows.

```
NonattackingQueens( )
{
    int columnNo;
    row++;
    for(columnNo = 1; columnNo <= 8; columnNo++)
    {
    if no queen placed in rows from 1 to (row-1) is attacking the position(row,
    columnNo)
```



```
    {
        column[row] = columnNo; /* place the queen on this row. */
        if(row == 8)      /* All queens have been successfully placed. */
        {
            print the board configuration;
            return;
        }
        NonattackingQueens( ); /*place another queen in */
                               /* the next row. */
    }
    }
    row—;
}
```

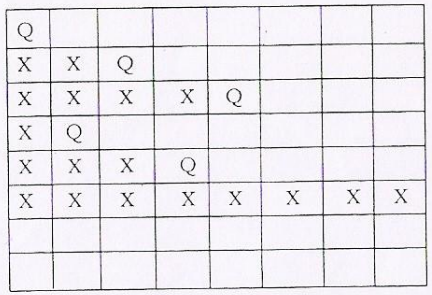

Figure 6.8 The configuration according to the previous description. "Q" is written to indicate that a queen in placed in that cell. "X" is marked to indicate that no queen can be placed in that cell as some other queen in previous rows would have attacked a piece in this cell. Note that a queen could not be placed on the 6th row so that the queens already placed in the first five rows do not attack it.

The function is called by setting the global variable "row" to "0". The function finds out a column where the queen in the current row may be placed. Such a column must be between 1 and 8. If such a column is found, the next queen is placed using the same function. If such a column is not found then the queen in the previous row must be placed on a different column. This is done by returning the current call to its caller after decrementing "row". It is not difficult to check, whether a given position(r, c) is non-attacking for queens in rows from 1 to row-1. The positions of

those queens can be retrieved by examining the array "column" for indices 1 to row – 1. Note that a queen in the i-th. row can attack another queen placed at cell (i, j) of the board if any of the two following conditions is true.

(i) The queens are placed on the same column. That is, column[i] is equal to "c".

(ii) The queens are placed on the same diagonal. That is, the magnitude of (r-i) is the same as the magnitude of (c – column[i]).

Writing a "C" function for eight queens problem is left as an exercise. The algorithm "NonAttackingQueens" can easily be generalized to n-queens problem where "n" queens are to be placed in a nXn chessboard so that no two queens attack each other. It is also instructive to draw the recursion tree of this algorithm for n = 4. At each node of the recursion tree the board configuration should be drawn to understand the algorithm better. This problem serves as an example of a class of problems which are solved by the method of trial and error. Whenever a dead end is reached, the algorithm backtracks by undoing the last step and tries to make a new attempt. As it was the case with other problems, backtracking can also be solved in a non-recursive manner.

6.7 Removal of Recursion

It has already been argued that some problems can be easily solved in a recursive manner and it is easier to prove the correctness of recursive algorithms. But it is also said that recursion involves space and time due to function calls. Moreover, some programming languages do not support recursion. Even in such circumstances the advantages of using recursion may be obtained. One can write the algorithm recursively and prove its correctness. Subsequently, the recursive algorithm may be translated into a non-recursive version by applying certain rules mechanically. Fortunately, any recursive procedure can be converted to a non-recursive procedure. Any call to a subprogram (recursive or non-recursive) requires that the subprogram has a storage area of its own where it can store its local variables and its actual parameters. Moreover, return address to the calling procedure must also be preserved before the call is made. These tasks are performed by the compiler and thus, they are usually transparent to the programmer. If a recursive call is to be replaced by a jump to the beginning of the function then appropriate care must be taken to preserve local variables and to remember the return address. Otherwise, each call will use the same area for local variables and destroy the values in the storage space for the outer call and the outer call cannot be completed successfully. Thus, any translation must clearly understand the beginning of a recursive call and the end of that call. When a recursive call begins, the values of local variables, parameters and the return address at the time of the call must be remembered. The end of a recursive call is indicated by a "return" statement. At that point of time the previous outer call must be recalled with resetting of the local variables and actions must resume from where it was suspended.

As the execution of subroutine calls work in a Last In First Out (LIFO) manner, local variables, calling parameters and the return address for each call should be stored in a stack. These items, viz. the local variables, actual parameters and the return address actually define the activation record for an activation of a recursive procedure. After the innermost call is finished via one "return" statement, the stack will be popped to recover the return address and all the local variables for the immediate outer call. When the outermost call completes its work, the stack is exhausted.

The above ideas are used for converting a recursive algorithm to a non-recursive one. The rules are given in the following.

Initialization

1. Declare a stack that will hold the activation records consisting of all local variables, parameters called by value and labels to specify where the function is recursively called(if it calls itself from several places).

   The non-recursive function for a recursive function starts with an initialization block which initializes the stack to be empty.

   The stack and the stack top pointer are defined as global variables.

2. To enable each recursive call to start at the beginning of the original function, its first executable statement after the stack initialization block is associated with a label.

The following steps should be done at each place inside the function where it is recursively called.

3. Push all local variables and parameters called by value into the stack.
4. Push an integer "i" into the stack if this is the i-th place from where the function is called recursively (On return, this integer "i" will indicate the part of the code from where to resume. See rule 7.)
5. Set the formal parameters called by value to the values given in the new call to the recursive function.
6. Replace the call to the recursive function with a "goto" to the statement which is the first statement after the stack-initialization block. A label is associated to this statement in rule 2.
7. Make a new statement label Li (if this is the i-th place where the function is called recursively) and attach the label to the first statement after the call to the same function(so that a return can be made to this label).

At the end of the recursive function or whenever the function returns to the calling program, the following steps should be performed.

8. If the stack is empty, then the recursion has finished; make a normal return.
9. Otherwise, pop the stack to restore the values of all local variables and parameters called by value.

10. Pop the integer "i" from the stack and use this to go to the statement labelled by Li.

Try the above rules to convert the function "move" which solves the "Tower of Hanoi" problem to an equivalent non-recursive version. The corresponding non-recursive version is presented in the following. The rules applied are commented appropriately. Observe that a cute and simple recursive algorithm turns into an ugly, complicated non-recursive version. This is partly because the rules were applied mechanically. The non-recursive version can be made more attractive by applying more thought and intelligence as will be described later.

```
void moveNonRecursive(int n, int s, int d, int u)
{
   int t,i;
   stackOfIntegers s1;                    /* rule 1 */
   initialize(&s1);                       /* rule 1 */
   label0:                                /* rule 2 */
   if(n == 1)
   {
        printf("Move the disk from %d to %d \n", s, d);
        if (isempty(s1))                  /* rule 8 */
              return;
        i = pop(&s1);                     /* rule 9 */
        t = pop(&s1);
        u = pop(&s1);
        d = pop(&s1);
        s = pop(&s1);
        n = pop(&s1);
        if(i == 1)                        /* rule 10 */
              goto label1;
        else
              goto label2;
   }
   push(&s1, n);                          /* rule 3 */
   push(&s1, s);
   push(&s1, d);
   push(&s1, u);
```



```
   push(&s1, t);
   push(&s1, 1);                          /* rule 4 */
   n = n-1;                               /* rule 5 */
   t = d;
   d = u;
   u = t;
   goto label0;                           /* rule 6 */
   label1:                                /* rule 7 */
   printf("Move the disk from %d to %d \n", s, d);
   push(&s1, n);                          /* rule 3 */
   push(&s1, s);
   push(&s1, d);
   push(&s1, u);
   push(&s1, t);
   push(&s1, 2);                          /* rule 4 */
   n = n-1;                               /* rule 5 */
   t = s;
   s = u;
   u= t;
   goto label0;                           /* rule 6 */
   label2:                                /* rule 7 */
   if (isempty(s1))                       /* rule 8 */
   return;
   i = pop(&s1);                          /* rule 9 */
   t = pop(&s1);
   u = pop(&s1);
   d = pop(&s1);
   s = pop(&s1);
   n = pop(&s1);
   if(i == 1)                             /* rule 10 */
         goto label1;
   else
         goto label2;
```

```
}
```

The above function uses a user defined type "stackOfIntegers" and four auxiliary functions "initialize", "push", "pop" and "isEmpty". These are defined in the following.

```
typedef struct stackOfIntegers
{
        int top;
        int Elements[1000];
}stackOfIntegers;
void push(stackOfIntegers* s, int x)
{
   s->top++
   s->elements[s->top] = x;
}
int pop(stackOfIntegers* s)
{
   int x;
   x = s->elements[s->top];
   s->top —;
   return x;
}
int isEmpty(stackOfIntegers s)
{
   return((s.top == -1)? 1 : 0);
}
void initialize(stackOfIntegers* s)
{
   s->top = -1;
}
```

Readers must have noticed that once recursion is removed by applying the described set of rules, the algorithm becomes a mess of goto's. Further simplification of the procedure to remove recursion is possible if the recursive call occurs once inside a function. Note that the recursion tree in such a case becomes a chain as shown in Figure 6.1. Consider the following algorithm, which is a general description of a recursive function that includes only one call to itself.


```
void f (parameters)
{
   while (not termination condition of recursion)
   {
        A;
        f (actual parameters);
        B;
   }
   C;
}
```

In the above algorithm, blocks "A", "B", "C" stand for any block of statements. Before applying the rules to remove recursion, note that the rule 10 is useless in this context, as there is no ambiguity for the return address. The simplest non-recursive version is presented next.

```
void fNR (parameters)
{
   initialize stack s;
   label0:
   while (not termination condition of recursion)
   {
        A;
        push parameters into stack s;
        goto label0;
   label1:
   B;
   }
   C;
   if(! isEmpty(s))
   {
        pop data from s;
        goto label1;
   }
}
```

Note that the statement "goto label0" is unnecessary if the body of the while loop

does not include the block "B". But then, once block "B" is done, control must go to the beginning of the "while" loop. This can be accomplished by enclosing the while loop inside a do-while loop. Moreover, "goto label1" may be avoided by directly executing Block "B" at that place. As a final point, note that the stack will be empty in the beginning. The next version utilizes these ideas to get rid of "goto" statements and labels.

```
void fNR1(parameters)
{
    intilize stack s;
    do
    {
        if(! isEmpty(s))
        {
            pop data from s;
            B;
        }
        while(not termination condition of recursion)
        {
            A;
            push data to s;
            change parameters;
        }
        C;
    } while(! isEmpty(s));
}
```

Tail Recursion

The general recursion removal technique described earlier, pushes actual parameters, local variables and the return address to the stack before initiating a recursive call. In the return statement (i.e. when the call terminates) values of all the local variables are popped from the stack so that the local variables are reset with these values. Suppose that, as a special case, the last executed statement of a function is a recursive call to itself. This special case is called tail recursion. According to rules 3, 4, 5, 6, 7, the call is replaced by a number of push operations. Since this is also the last statement of the function, the immediate "return" is replaced by a number of "pop" operations according to rules 8, 9 and 10. In such circumstances it is pointless to do the push and pop operations mindlessly on the stack. In fact, there

is not even any need to use the stack. Instead, the actual parameters used to call the function recursively may be assigned to their new values and then a branch to the beginning of the function is all that is needed to eliminate recursion. The process of transformation can be viewed pictorially in Figure 6.9.

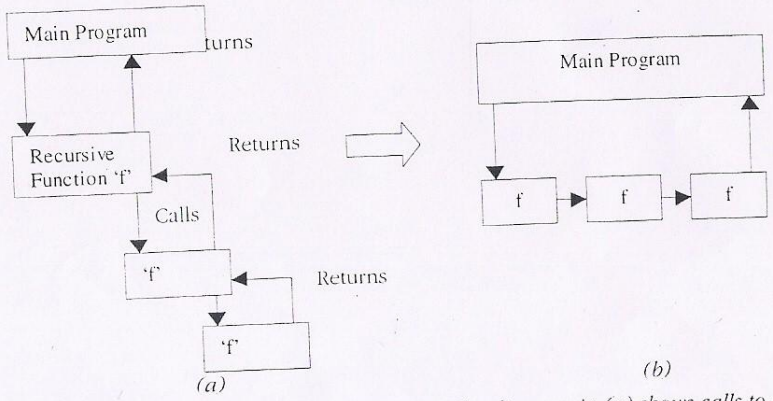

Figure 6.9 Illustration of removing tail recursion. The diagram in (a) shows calls to a tail recursive program. The same sequence of calls can be redrawn as a series of calls as shown in the diagram (b).

Observe that the function "move" to solve the "Tower Of Hanoi" problem is an example of tail recursive function because the last action in "move" is a call to itself.

To remove tail recursion, note that the values of actual parameters "u" and "s" in the last recursive call to "move" are the values of "s" and "u" in the previous call. Thus these values are to be swapped before branching to the beginning of the function after eliminating recursion.

```
void move(int n, int s, int d, int u)
{
    int t;
    l:
    if(n == 1)
    {
        printf("move the disk from %d to %d\n", s, d);
    }
    else
```

```
        move(n-1, s, u, d);
        printf("move the disk from %d to %d\n", s, d);
        n = n-1;
        t = u;
        u = s;
        s = t;
        goto l;
    }
}
```

Note that once tail recursion is removed, the recursion tree of the function "move" becomes a chain. More about recursion removal will be discussed in the context of tree traversal algorithms in Chapter 7.

6.8 Analysis of recursive algorithms

In this chapter and in the previous chapters, a number of recursive algorithms have been developed. It has already been mentioned that the basic principle in these algorithms is to decompose a problem into smaller sub-problems, to solve the sub-problems and finally to combine the solutions of the sub-problems into a solution of the original problem. In this section, some basic results on analyzing such algorithms are looked into. Understanding the mathematical properties of the formulas in this section may provide insight into the performance properties of algorithms discussed in the book.

If a recursive function looks through the input of size "n" to eliminate one of them then its time complexity, T(n), follows the following recurrence relation.

$T(n) = T(n-1)+k.n$ where k is a constant.

In this case $T(n) = O(n^2)$ if $T(1) = k'$, a constant

$$T(n) = T(n-1) + k.n$$
$$= T(n-2) + k(n-1) + k.n$$
.
.
.
$$= T(n - n + 1) + k(n + (n-1) + \ldots + 1)$$
$$= T(1) + k(1 + 2 + 3 + \ldots + n - 1 + n)$$
$$= k' + k(n(n+1)/2)$$

If a recursive function halves the input in one step and works only in one half after

that step then its time complesity satisfies the following recurrence relation.

$$T(n) = T(n/2) + k$$

If $n = 2^m$ for some "m" then $T(n) = O(\log_2 n)$ provided $T(1) = k'$

$$T(n) = T(n/2) + k$$
$$= T(n/4) + k + k$$
.
.
.
$$= T(n/2^m) + k.m$$
$$= k' + k.\log_2 n$$

If a recursive function halves the input into two and has to make a linear pass through the input before, during or after the input is split into two halves then the time complexity satisfies the following recurrence relation.

$T(n) = 2T(n/2) + k.n$ for a constant "k".

If $n = 2^m$ then $T(n) = O(n \log n)$

$$T(2) = 2T(2^{m-1}) + k2^m$$
$$= 2(2T(2^{m-2}) + k2^{m-1}) + k.2^m$$
$$= 2^2T(2^{m-2}) + k.2^m(1 + 1)$$
.
.
.
$$= 2^{m-1}T(2^{m-m+1}) + kn(1 + 1 + 1 + \ldots + 1)$$
$$= (n/2)k' - k.n + k.n.m$$
$$= O(n \log n)$$

A general recurrence relation can be written as

$$T(n) = aT(n/b) + f(n)$$

where a ≥ 1 and b > 1 are constants and f(n) is a positive function. This relation describes a recursive algorithm which decomposes a problem of size "n" to "a" number of subproblems of size n/b. The total cost of decomposition and the cost of combinations of the results of the subproblems is given by f(n). In this case, T(n) may be expressed as

$$T(n) = \begin{cases} k & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

where "i" is a positive integer. Then,

$$T(n) = kn^{\log_b a} + \Sigma_{(i=0,\ \log_b n-1)}(a^i f(n/b^i))$$

This result can be easily seen by unfolding the recurrence.

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + a^2 T(n/b^2) \\ &= f(n) + af(n/b) + a^2 T(n/b^2) + \ldots \\ &\quad a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \end{aligned}$$

Since, $a^{\log_b n} = n^{\log_a b}$, the result follows.

Let $g(n) = \Sigma_{(i=0,\ \log_b n-1)}(a^i f(n/b^i))$.

Depending on f(n), g(n) can be bounded asymptotically for exact powers of "b" as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ then $g(n) = O(n^{\log_b a})$.
2. If $f(n) = \theta(n^{\log_b a})$ then $g(n) = \theta(n^{\log_b a} \log n)$
3. If $af(n/b) \leq cf(n)$ for some constant $c < 1$ and $n \geq b$ then $g(n) = \theta(f(n))$.

It can also be shown that T(n) = g(n) for all threes cases mentioned here. The proof for these results are left as an exercise.

6.9 Recursion versus iteration

It is already discussed that recursion implicitly uses a stack in which activation records of recursive calls are pushed. These activation records are popped when a call to a function returns to its caller. Thus, any call to a sub-routine is a time consuming operation during execution of a program and therefore, recursive functions perform better only after eliminating recursion, using rules that were outlined in a previous section. In addition to this, some more points have also been observed. It was argued that recursion should not be used for the computation of the factorial of an integer or computation of the n-th Fibonacci number. Time complexity of a recursive algorithm to compute the n-th Fibonacci number was seen to be too large as compared to an iterative solution. On the other hand, recursion was seen to be the natural choice for solving the "Tower of Hanoi" problem and for generating permutations. Thus, it is important to find out the fundamental characteristics of a problem to decide whether a recursive or a non-recursive solution would be better. Construction of a recursion tree helps in making such decisions as it provides much useful information about the nature of the underlying recursion.

If the recursion tree is a chain then each call to the function results in another call to the same function. For example, the recursion tree for the function "factorial" is a chain. In such a situation, if the recursion tree is generated in a bottom up fashion then the need for recursion is no longer present. It is interesting to note that recursion is inherently a top-down approach in solving a problem. When a recursion

tree reduces to a chain then converting a recursive function to an iterative function is often easy and will result in a saving of both space and time.

Now, consider the recursion tree for the solution to find out the n-th Fibnacci number. There are many nodes in the tree representing same tasks done repeatedly. Since recursion implicitly uses a stack data structure, results once obtained are popped and not remembered thereafter. This is why "Fibo(2)" is called so many times. If some data structure other than the stack could be used then results once obtained might be preserved for future references. An automatic choice is an ordered list(represented by an array) to store the intermediate results. The iterative version of the solution for computing the n-th. Fibonacci number is a little tricky because it does not involve an array. But a straight forward bottom-up solution to this problem is found at once if an array is used to store the Fibonacci numbers computed upto any given point as shown below.

```
int Fibo(int n)
{
    int i;
    int Fibonacci[1000];
    Fibonacci[0] = 0
    Fibonacci[1] = 1;
    for(i = 2; i <= n; i++)
        Fibonacci[i] = Fibonacci[i-1] + Fibonacci[i-2];
    return(Fibonacci[n]);
}
```

The above function can not be used to compute Fibo(n) for n≥1000. This function is also a little general in the sense that the same technique can be applied to compute a series of positive integers as defined in the following

$$G(0) = 0;\ G(1) = 1;\ G(n) = G(n-1) + G(n/2);$$

Note that the trick used in Chapter 1 for the solution of finding the n-th Fibonacci number can not be used for computing the series G as described now.

Sometimes, the symmetry of a recursion tree can also be exploited to avoid recursion. This trick is used to remove recursion from an important sorting algorithm called "merge sort" and is discussed in Chapter 9. As a final comment, it must be mentioned that while any recursion can be replaced by a stack and iteration, any iteration with a stack can also be replaced by recursion. The programmer should first find out which approach leads to a more natural and efficient solution before attempting to improve it.

6.10 Points to be remembered

- Recursion is a powerful mathematical tool to express ideas and functions. Most modern programming languages support recursive function calls. Recursion often offers elegant solutions to difficult problems.
- Recursion tree illustrates the sequence in which calls are made to the same function. Recursion tree is particularly helpful in finding out peculiarities of the underlying recursion.
- A significant class of problems which are solved by recursion is classified as "divide and conquer" problem. Such problems are first decomposed into smaller sub-problems. Recursion is used to solve the smaller problems. Then, these solutions are combined to obtain the overall solution. Several sorting algorithms(e.g. merge sort, quick sort) and binary search algorithm to be discussed in chapter 9 are classical examples of this class of problems.
- Two illustrative problems, namely the "Tower of Hanoi" problem and the problem of generation of permutations are studied in detail to bring out the feature of recursive programming.
- Another class of problems where recursion is useful involves backtracking. These are problems which are solved by trial and error. One illustrative problem of this class is known as eight queens problem which has been studied to show the use of recursion in solving backtracking problems.
- Recursive programs are usually more time consuming because of the overhead of calling functions repeatedly. Thus, it is instructive to convert a recursive algorithm to non-recursive one. There are a set of rules which may mechanically applied for the conversion. The conversion essentially utilizes a stack for this purpose.
- If the recursion tree is just like a chain or the recursion is tail recursion then the set of rules to convert recursive algorithms to non-recursive can be further simplified.
- Analysis of recursive algorithms boils down to solving recurrence relations.

Exercises

1. The binominal co-efficient C is defined as follows.

$$^{n}C_{m} = \frac{n!}{m!(n-m)!}$$

The same can be recursively defined as given below.

$$^{n}C_{m} = {}^{n-1}C_{m} + {}^{n-1}C_{m-1}$$

and $^{n}C_{0} = {}^{n}C_{n} = 1$.

Write a recursive function to compute $^{n}C_{m}$. Analyze the time complexity of your function.

2. Ackermann's function A(m, n) is defined as follows.

A(m, n) = n+1, if m = 0

= A(m-1, 1) , if n = 0

= A(m-1, A(m, n-1)) , otherwise.

Write a recursive function to compute A(m, n).

3. Let "S" be a set of n elements such that S ={1, 2, 3,...., n}. A power set of "S" is the set of all possible sub-sets of "S". For example, if n = 3, S = {1, 2, 3}. Power set of S = {{ }, {1}, {2}, {3}, {1, 2}, {1,3}, {2, 3}, {1, 2, 3}. Write a recursive function to compute the power set of "S".
4. Write a recursive function to solve the "Josephus" problem as discussed in Chapter 3.
5. Write a recursive function to compute the greatest common divisor of two positive numbers using Euclid's Algorithm. See Chapter 1 for a discussion on the Euclid's algorithm.
6. Write a complete "C" program to solve the eight queen's problem as discussed in this chapter.
7. Let T(n) be defined as follows.

$$T(n) = \begin{cases} k & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^{i} \end{cases}$$

Prove that

(i) If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$ then $T(n) = O(n^{\log_b a})$.

(ii) If $f(n) = \theta(n^{\log_b a})$ then $T(n) = \theta(n^{\log_b a}.\log n)$

(iii) If $af(n/b) \leq cf(n)$ for some constant $c < 1$ and $n \geq b$ then $T(n) = \theta(f(n))$.

8. Write a recursive function to compute the largest integer less than $\log_2 n$.
9. Remove recursion from the algorithms developed for Exercises 1, 2, 3, 4 and 5 and then optimize the non-recursive versions.
10. Draw a recursion tree for the algorithm for the n-queens problem where n = 4.
11. Write a non-recursive "C" function for solving the n-queens problem.
12. Remove recursion from the "C" function that prints permutation of "n" integers.
13. Remove recursion from the algorithm "move" without tail recursion so that the non-recursive version does not have a "goto" statement.

is traversed. Quite understandably, the left (as well as the right) sub-tree of "T" could be another binary tree and therefore, traversing the left (or right) sub-tree of "T" is to be carried out in the same fashion as traversing is to be done on "T". In case "T" is empty, then traversal of "T" involves doing nothing. The "C" function for inorder traversal of a tree is as follows. It is assumed that the type of each node in the tree is "bNode" as described earlier . Each node contains a character as its information and it is also assumed that visiting each node involves printing the information associated with that node.

```
void Inorder (bNode *T)
{
        if (T != NULL)
        {
                Inorder (T->left);
                printf ("%c ", T->data);
                Inorder (T->right);
        }
        return;
}
```

If this function is executed on the tree of Figure 7.8(a) then the following output would be produced.

D G B H E A F I C

7.6.2 Preorder Traversal

If a tree, "T", is traversed in "preorder" fashion then the root node of "T" is visited first and then the left sub-tree of "T" is traversed and finally the right sub-tree of "T" is traversed. As before, if T is NULL then traversal of "T" involves doing nothing. Translating these ideas, the following "C" function for Preorder traversal is obtained.

```
void Preorder (bNode *T)
{
        if (T != NULL)
        {
                printf ("%c ", T->data);
                Preorder (T->left);
                Preorder (T->right);
        }
        return;
}
```


If the binary tree of Figure 7.8(a) is traversed in Preorder manner, then the following output would be generated.

A B D G E H C F I

7.6.3 Postorder Traversal

If a tree, "T", is traversed in "postorder" manner, then the left sub-tree of "T" is traversed first, then the right sub-tree of "T" is traversed and finally the root node of "T" is visited. For an empty tree, the traversal implies doing nothing. The "C" function for Postorder traversal of the tree, T, is presented below.

```
void Postorder (bNode *T)
{
        if (T != NULL)
        {
                Postorder (T->left);
                Postorder (T->right);
                printf ("%c ", T->data);
        }
        return;
}
```

The Postorder traversal of the binary tree of Figure 7.8(a) produces the following output.

G D H E B I F C A

7.6.4 Non-recursive Traversal of a binary tree

In the previous chapter, techniques to remove recursion using a stack have been discussed. In this section, this technique will be applied to the recursive traversal algorithms to convert them to equivalent non-recursive ones. Improvements on these non-recursive algorithms will also be attempted and discussed. First, consider transformation of the recursive function "inorder" which is used to traverse a binary tree in an "inorder" manner. The rules discussed in Chapter 6 are applied to the function "Inorder" to obtain the following non-recursive version.

```
void nonRecursiveInorder(bNode * T)
{
   int i;
   bNodeStack s;
   initialize(&s);
   L1: if(T != NULL)
```

```
{
        pushTree(&s, T);
        pushAddress(&s, 1);
        T = T->left;
        goto L1;
L2:     printf("%c ", T->data);
        pushTree(&s, T);
        pushAddress(&s, 2);
        T = T->right;
        goto L1;
}
L3: if( !isEmpty(s))
{
        i = popAddress(&s);
        T = popTree(&s);
        if(i ==1)
                goto L2;
        else
                goto L3;
}
}
```

The function "nonRecursiveInorder" assumes definitions of some user-defined types and a few functions implementing a stack. The stack contains either a pointer to a tree or an integer (representing a return address). Further, it is assumed that overflow does not occur and hence error checking due to overflow has not been incorporated.

The type declaration for bNodeStack is as follows.

```
typedef struct bNodeStack
{
   int top;
   treeOrAddress Elements[100];
}bNodeStack;
```

The type of the contents of the stack is either a pointer to a tree or a label as described in the following.


```
typedef union treeOrAddress
{
   bNode * t;
   int label;
}treeOrAddress;
```

The operations to be done on such a stack include initialization, push and pop. These operations are described next.

```
initialize(bNodeStack *s)
{
   s->top = -1;
}

void pushTree(bNodeStack *s, bNode * tree)
{
   s->top += 1;
   (s->Elements[s->top]).t = tree;
}

bNode * popTree(bNodeStack *s)
{
   bNode *x;
   x = (s->Elements[s->top]).t;
   s->top -= 1;
   return x;
}

pushAddress(bNodeStack *s, int x)
{
   s->top += 1;
   (s->Elements[s-> top]).label = x;
}
int popAddress(bNodeStack *s)
{
   int x;
```

```
    x = (s->Elements[s->top]).label;
    s->top -= 1;
    return x;
}
int isEmpty(bNodeStack s)
{
return((s.top == -1)? 1: 0);
}
```

A closer inspection of the recursive function "Inorder" reveals that it is tail recursive. Hence, its tail recursion can be immediately removed and the resultant algorithm is described next.

```
void Inorder1(bNode * T)
{
L1:if (T != NULL)
   {
          Inorder1(T->left);
          printf("%c", T->data);
          T = T->right;
          goto L1;
   }
   return;
}
```

The function "Inorder1" is still recursive. A non-recursive version of the function "Inorder1" is presented next.

```
void nonRecursiveInorder1(bNode * T)
{
   int i;
   bNodeStack s;
   initialize(&s);
   L1:    if (T != NULL)
   {
          pushTree(&s, T);
          pushAddress(&s, 1);
          T = T->left;
          goto L1;
```



```
L2:       printf("%c ", T->data);
          T = T->right;
          goto L1;
   }
   if(!isEmptyStack(s))
   {
          i = popAddress(&s);
          T=popTree(&s);
          if(i == 1)
                goto L2;
   }
}
```

It is interesting to note that only one label (i.e., '1') is always pushed as an address in the previous function. Therefore, the value of the address being popped out is always "1". Thus, there is no need to push the address into the stack or pop it out. The mesh of goto's can also be easily removed by having two loops one nested in another. Observe that the first part of the algorithm (from the label "L1" upto the start of "L2") can be enclosed in a while-loop. The statement "goto L2" may be avoided by replacing it with the statements starting from the label "L2" and yielding control to the beginning of the while loop just described. The function to traverse a tree in an inorder manner without "goto" statements is described in the following.

```
void nonRecursiveInorder2(bNode * T)
{
   bNodeStack s;
   initialize(&s);
   while(1)
   {
          while(T != NULL)
          {
                pushTree(&s, T),
                T = T->left;
          }
          if (isEmpty(s))
                return;
          T = popTree(&s);
```