Recursion

Divide and conquer. One of the uses that we make of recursion of the form called divide and conquer, which can be defined generally as the method of solving a problem by dividing it into two or more subproblems, each of which is similar to the original problem in nature, but smaller in size. Solutions to the subproblems are then obtained separately, and combined to produce the solution of the original problem. Hence, we can sort a list by dividing it into two sublists, sort them separately, and combine the results.

The Towers of Hanoi. In the nineteenth century a game called the Tower of Hanoi appeared in Europe, together with material explaining that the game represented a task underway in the Temple of Brahma. At the creation of the world, the priests were given a brass platform on which were 3 diamond needles. On the first needle were stacked 64 golden disks, each one slightly smaller than the one under it. The priests were assigned a task of moving all the golden disks from the first needle to the third, subject to the condition that only one disk can be moved at a time, and that no disk is ever allowed to be placed on top of a smaller disk. The priests were told that when they had finished moving the 64 disks, it would signify the end of the world.

Our task, of course, is to write a computer program that will type out a list of instructions for moving 64 disks from needle 1 to needle 3 using needle 2 for intermediate storage, which can be written as

move(64, 1, 2, 3);
The idea that gives a solution is to concentrate our attention not on the first step (which must be to move the top disk somewhere), but rather on the hardest step: moving the bottom disk. There is no way to reach the bottom disk until the top 63 disks have been moved, and, furthermore, they must all be on needle 2 so that we can move the bottom disk from needle 1 to needle 3. This is because only one disk can be moved at a time and the bottom (largest) one can never be on the top of any other, so that when we move the bottom one, there can be no other disks on needles 1 or 3. Thus, we can summarize the steps of our algorithm as

    move(63, 1, 2, 3);
    printf("move a disk from needle 1 to needle 3. 
");
    move(63, 2, 3, 1);

We now a small step toward the solution, only a very small one since we must still describe how to move the 63 disks two times, but a significant step nonetheless, since there is no reason why we cannot move the 63 remaining disks in the same way.

This is exactly the idea of recursion. We have described how to do the key step and asserted that the rest of the problem is done in essential the same way.

We can now write the complete program.

```c
#include <iostream>
#include <conio>
#include <stdio>
#include <dos>
#define DISKS 15

//Towers of Hanoi
void move(int, int, int, int);

void main (void)
{
    struct time t1, t2;
    gettimeofday(&t1);
    move(DISKS, 1, 3, 2);
    gettimeofday(&t2);
    printf("Process started at: %2d:%02d:%02d.%02d
",
        t1.ti_hour, t1.ti_min, t1.ti_sec, t1.ti_hund);
    printf("Process finished at: %2d:%02d:%02d.%02d
",
        t2.ti_hour, t2.ti_min, t2.ti_sec, t2.ti_hund);
    getch();
}

//Moves n disks from a to b using c for temporary storage

void move(int n, int a, int b, int c)
{
```
It can easily be seen that there are two recursive calls within `move(n, a, b, c)`. The way the calls work is illustrated in the following picture.
Generating permutations. Divide and conquer, by definition, involves two or more recursive calls within the algorithm being written. In this section we illustrate application of recursion using only one recursive call. In this application one case or one phase of the problem is solved without using recursion, and the work of remainder of the problem is postponed to the recursive call.

Consider a problem of generating the $n!$ permutations of $n$ objects as efficiently as possible. If we think of the number $n!$ as the product

$$n! = 1 \times 2 \times 3 \times \cdots \times n,$$

then the process of multiplication can be pictured as the tree. We can identify the permutations with the nodes of the tree. At the top is 1 by itself. We can obtain the two permutations of \{1,2\} by writing 2 first on the left, then on the right of 1. Similarly, the six permutations of \{1,2,3\} can be obtained by starting with one of the permutations (2,1) or (1,2) and inserting 3 into one of the three possible positions (left, center, or right). The task of generating permutations of \{1,2,\ldots,k\} can now be summarized as

*Take a given permutation of \{1,2,\ldots,k-1\} and regard it as an ordered list. Insert $k$, in turn, into each of the $k$ possible positions in this ordered list, thereby obtaining $k$ distinct permutations of \{1,2,\ldots,k\}.***

This algorithm illustrates the use of recursion to complete tasks that have been temporarily postponed. That is, we can write a function that will first insert 1 into an empty list, and then use a recursive call to insert the remaining numbers from 2 to $n$ into the list. The first recursive call will
insert 2 into the list containing only 1, and postpone further insertions to a recursive call. On the nth recursive call, finally, the integer n will be inserted. In this way, having begun with a tree structure as motivation, we have now developed an algorithm for which the given tree becomes the recursion tree.

**Nonattacking queens.** For another example of an algorithm where recursion allows the postponement of all but one case, let us consider the puzzle of how to place eight queens on a chessboard so that no queen can take another. Recall that a queen can take another piece that lies on the same row, the same column, or the same diagonal (either direction) as the queen. The chessboard has eight rows and columns.

It is by no means how to solve this puzzle, and its complete solution defied even the great C. F. Gauss, who attempted it in 1850. To convince you that solutions to this problem really do exist, two of them are shown in the figure.

![Chessboard with queens](image)

A person attempting to solve the Eight Queens problem will usually soon abandon attempts to find all (or even one) of the solutions by being clever and will start to put queens on the board, perhaps randomly or perhaps in some logical order, but always making sure that no queen placed can take another on the board. If the person is lucky enough to get eight queens on the board by proceeding in this way, then he has found a solution; if not, then one or more of the queens must be removed and placed elsewhere to continue the search for a solution.

Let us sketch this method in algorithmic form. We denote by n the number of queens on the board; initially n = 0. The key step is described as follows.

```c
void AddQueen(void)
```
for (every unguarded position p on the board){
    Place a queen in position p;
    n++;
    if (n == 8)
        Print the configuration;
    else
        AddQueen();
    Remove the queen from position p;
    n--;
}

This sketch illustrates the use of recursion to mean “Continue to the next stage and repeat the task.” Placing a queen in position p is only tentative; we leave it there only if we can continue adding queens until we have eight. Whether we reach eight or not, the function will return when it finds that it has finished or there are no further possibilities to investigate. After the inner call has returned, then it is time to remove the queen from position p, because all possibilities with it there have been investigated.

Program code of the algorithm is given below.

```c
#include <iostream>
#include <conio>
#include <stdio>

int col[8];             // Column with the queens
bool colfree[8];           // Is the column free?
bool upfree[15];            // Is the upward diagonal free?
bool downfree[15];         // Is the downward diagonal free?

int row = -1;              // Row whose queen is already placed
int sol = 0;               // Number of solutions found

void AddQueen(void);

//*********************Eight Queens Problem*************************

void main(void)
{
    int i;
    for (i = 0; i < 8; i++)
        colfree[i] = true;
    for (i = 0; i < 15; i++)
    {
        upfree[i] = true;
        downfree[i] = true;
    }
    AddQueen();
    getch();
}

//***AddQueens: attemps to place queens, backtracking when needed***

void AddQueen(void)
{
    int c;                    // Column being tried for the queen
row++;  
for (c = 0; c < 8; c++)
    if (colfree[c] && upfree[row+c] && downfree[row - c + 7]){
        col[row] = c;
        colfree[c] = false;
        upfree[row + c] = false;
        downfree[row - c + 7] = false;
        printf("\n");
        if (row == 7)       // Termination condition
            for (int i = 0; i < 8; i++)
                printf("col[%d] = %d
", i, col[i]);
        else
            AddQueen();       // Proceed recursively
                colfree[c] = true;    // Now backtrack by removing the queen
                upfree[row + c] = true;
                downfree[row - c + 7] = true;
    }
    row--;
}

******************************************************************  

Sample run:

col[0] = 6  
col[1] = 1  
col[2] = 5  
col[3] = 2  
col[4] = 0  
col[5] = 3  
col[6] = 7  
col[7] = 4  

col[0] = 7  
col[1] = 1  
col[2] = 4  
col[3] = 2  
col[4] = 0  
col[5] = 6  
col[6] = 3  
col[7] = 5  

col[0] = 7  
col[1] = 3  
col[2] = 0  
col[3] = 2  
col[4] = 5  
col[5] = 1  
col[6] = 6  
col[7] = 4  

The algorithm can be traced using the following tree. One can easily see the use of backtracking by looking at the tree.