```
void TowersOfHanoi(int n, char frompeg, char topeg, char auxpeg) {
    /* If only 1 disk, make the move and return */
    if(n==1) {
        printf("Move disk 1 from peg %c to peg %c",frompeg, topeg);
        return;
    }
    /* Move top n-1 disks from A to B, using C as auxiliary */
    TowersOfHanoi(n-1, frompeg, auxpeg, topeg);

    /* Move remaining disks from A to C */
    printf("\nMove disk %d from peg %c to peg %c", n, frompeg, topeg);

    /* Move n-1 disks from B to C using A as auxiliary */
    TowersOfHanoi(n-1, auxpeg, topeg, frompeg);
}
```

Problem-2 Given an array, check whether the array is in sorted order with recursion.

Solution:

```
int isArrayInSortedOrder(int A[],int n){
    if(n == 1)
      return 1;
    return (A[n-1] < A[n-2])?0:isArrayInSortedOrder(A,n-1);
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*) for recursive stack space.

2.10 What is Backtracking?

Backtracking is an improvement of the brute force approach. It systematically searches for a solution to a problem among all available options. In backtracking, we start with one possible option out of many available options and try to solve the problem if we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other option and try to solve it. If none if the options work out we will claim that there is no solution for the problem.

Backtracking is a form of recursion. The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of

options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't.

Backtracking can be thought of as a selective tree/graph traversal method. The tree is a way of representing some initial starting position (the root node) and a final goal state (one of the leaves). Backtracking allows us to deal with situations in which a raw brute-force approach would explode into an impossible number of options to consider. Backtracking is a sort of refined brute force. At each node, we eliminate choices that are obviously not possible and proceed to recursively check only those that have potential.

What's interesting about backtracking is that we back up only as far as needed to reach a previous decision point with an as-yet-unexplored alternative. In general, that will be at the most recent decision point. Eventually, more and more of these decision points will have been fully explored, and we will have to backtrack further and further. If we backtrack all the way to our initial state and have explored all alternatives from there, we can conclude the particular problem is unsolvable. In such a case, we will have done all the work of the exhaustive recursion and known that there is no viable solution possible.

- Sometimes the best algorithm for a problem is to try all possibilities.
- This is always slow, but there are standard tools that can be used to help.
- Tools: algorithms for generating basic objects, such as binary strings [2^n possibilities for n-bit string], permutations [$n!$], combinations [$n!/r!(n - r)!$], general strings [k –ary strings of length n has k^n possibilities], etc...
- Backtracking speeds the exhaustive search by pruning.

2.11 Example Algorithms of Backtracking

- Binary Strings: generating all binary strings
- Generating k – ary Strings
- N-Queens Problem
- The Knapsack Problem
- Generalized Strings
- Hamiltonian Cycles [refer to *Graphs* chapter]
- Graph Coloring Problem

2.12 Backtracking: Problems & Solutions

Problem-3 Generate all the strings of n bits. Assume $A[0..n - 1]$ is an array of size n.

Solution:

```
void Binary(int n) {
    if(n < 1)
        printf("%s", A);              //Assume array A is a global variable
    else {
        A[n-1] = 0;
        Binary(n - 1);
        A[n-1] = 1;
        Binary(n - 1);
    }
}
```

Let $T(n)$ be the running time *of binary*(n). Assume function *printf* takes time O(1).

$$T(n) = \begin{cases} c, & \text{if } n < 0 \\ 2T(n-1) + d, & \text{otherwise} \end{cases}$$

Using Subtraction and Conquer Master theorem we get: $T(n) = O(2^n)$. This means the algorithm for generating bit-strings is optimal.

Problem-4 Generate all the strings of length n drawn from 0... $k - 1$.

Solution: Let us assume we keep current k-ary string in an array $A[0..\ n - 1]$. Call function *k-string*(n, k):

```
void k-string(int n, int k) {
    //process all k-ary strings of length m
    if(n < 1)
        printf("%s",A);               //Assume array A is a global variable
    else {
        for (int j = 0 ; j < k ; j++) {
            A[n-1] = j;
            k-string(n- 1, k);
        }
    }
}
```

Let $T(n)$ be the running time of $k - string(n)$. Then,

$$T(n) = \begin{cases} c, & if\ n < 0 \\ kT(n-1) + d, & otherwise \end{cases}$$

Using Subtraction and Conquer Master theorem we get: $T(n) = O(k^n)$.

Note: For more problems, refer to *String Algorithms* chapter.

Problem-5 Finding the length of connected cells of 1s (regions) in an matrix of Os and 1s: Given a matrix, each of which may be 1 or 0. The filled cells that are connected form a region. Two cells are said to be connected if they are adjacent to each other horizontally, vertically or diagonally. There may be several regions in the matrix. How do you find the largest region (in terms of number of cells) in the matrix?

Sample Input: 11000 Sample Output: 5
01100
00101
10001
01011

Solution: The simplest idea is: for each location traverse in all 8 directions and in each of those directions keep track of maximum region found.

```
int getval(int (*A)[5],int i,int j,int L, int H){
    if (i< 0 || i >= L || j< 0 || j >= H)
        return 0;
    else
        return A[i][j];
}
void findMaxBlock(int (*A)[5], int r, int c,int L,int H,int size, bool **cntarr,int &maxsize){
    if ( r >= L || c >= H)
        return;
    cntarr[r][c]=true;
    size++;
    if  (size > maxsize)
        maxsize = size;
    //search in eight directions
    int direction[][2]={{-1,0},{-1,-1},{0,-1},{1,-1},{1,0},{1,1},{0,1},{-1,1}};
    for(int i=0; i<8; i++)        {
        int newi =r+direction[i][0];
        int newj=c+direction[i][1];
        int val=getval (A,newi,newj,L,H);
        if (val>0  && (cntarr[newi][newj]==false)){
            findMaxBlock(A,newi,newj,L,H,size,cntarr,maxsize);
        }
    }
    cntarr[r][c]=false;
}
int getMaxOnes(int (*A)[5], int rmax, int colmax){
    int maxsize=0;
    int size=0;
    bool **cntarr=create2darr(rmax,colmax);
    for(int i=0; i< rmax; i++){
        for(int j=0; j< colmax; j++){
            if (A[i][j] == 1){
                findMaxBlock(A,i,j,rmax,colmax, 0,cntarr,maxsize);
            }
        }
    }
    return maxsize;
}
```

Sample Call:

```
int zarr[][5]={{1,1,0,0,0},{0,1,1,0,1},{0,0,0,1,1},{1,0,0,1,1},{0,1,0,1,1}};
cout << "Number of maximum 1s are " << getMaxOnes(zarr,5,5) << endl;
```

Problem-6 Solve the recurrence $T(n) = 2T(n - 1) + 2^n$.

Solution: At each level of the recurrence tree, the number of problems is double from the previous level, while the amount of work being done in each problem is half from the previous level. Formally, the i^{th} level has 2^i problems, each requiring 2^{n-i} work. Thus the i^{th} level requires exactly 2^n work. The depth of this tree is n, because at the i^{th} level, the originating call will be $T(n - i)$. Thus the total complexity for $T(n)$ is $T(n2^n)$.