Backtracking - The Famous N Queen Problem

Problem Statement

The N-Queen problem states that given an NxN Chessboard we have to find a configuration of placing N queens such that each row and each column contains a queen and none of the queens attack each other.

Backtracking Technique.

The N-Queen problem is a classic example of the backtracking technique. The backtracking technique builds incremental candidates to the solutions, abandoning a candidate as soon as it positively determines that the current candidate cannot be a solution. Typical backtracking solutions proceed recursively following a Depth-first approach. At Each point it checks if the current candidate can be completed to a solution. If it does, the recursion stops there and returns the candidate. Otherwise it proceeds to the next candidate.

Naive Method

The naive approach is to try out all different board configurations such that there is one queen in every row and every column. Then for each configuration we check if our condition of no queen attacking each other is satisfied.

Disadvantages of the naive approach.

To generate each different possible combinations there would be ${}^{(n\hat{}2)}\mathbf{C}_{\mathbf{n}}$ different configurations. Applying the constrain that each row can contain only one queen we can bring down the number of configurations to $\mathbf{n^n}$ which is a huge number. It is computationally very slow even in modern bleeding edge computers to calculate such configurations. We apply the technique of backtracking to reduce the time complexity.

Backtracking Method

We apply the backtracking method to this problem to smartly solve this and reduce the computational complexity. The idea is to place the queen column by column, starting from the leftmost column. Before placing the queen in a new row, we check for clashes with queens already placed. If it is safe we place the queen and check if the current candidate leads to a solution. If it doesn't we remove the queen and move forward to the next row.

Algorithm

1) Start in the leftmost Column.
2) If queens are placed in every column
   return true
3) Try all rows in current column.
   a) In each row, check if current cell is attacked by a previously placed queen and if it isn't, mark the current position and recursively check if placing the queen here leads to a solution
   b) If placing a queen leads to a solution
      return true
   c) If it doesn't lead to a solution, then unmark the current place and move to the next row
4) If all rows have been exhausted return false to trigger backtracking and to check for another candidate.

Code:

```
#include<iostream>
#include<string.h> //For the memset function
using namespace std;

#define MAX_N 1010
int board[MAX_N][MAX_N];
void print(int n)
{

    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            if(board[i][j])
            {
                cout<<"Q ";
            }
            else
            {
                cout<<"X ";
            }
        }
        cout<<"\n";
    }
}
```

```
bool rowisclear(int n,int x,int y)
{
    for(int i=0;i<n;i++)
        if(board[x][i])
            return false;
    return true;
}

bool UpperDiagIsClear(int n,int x,int y)
{
    //We start at (x,y) and move up the
    //diagonal to check if there's a queen
    for(int i=x,j=y;i>=0 and j>=0;i--,j--)
        if(board[i][j])
            return false;
    return true;
}

bool LowerDiagIsClear(int n,int x,int y)
{
    //Again We start at (x,y) and move down this time
    for(int i=x,j=y;i<n and j>=0;i++,j--)
        if(board[i][j])
            return false;
    return true;
}
bool issafe(int n,int x,int y)
{
    //issafe function assumes that the queens have been placed
    //only from column 0 to y-1

    //We check if a queen is already placed in the current row.
    if(!rowisclear(n,x,y))
        return false;

    //Since queens are placed on the left side we check if the upper
    //diagonal to the left has a queen
    if(!UpperDiagIsClear(n,x,y))
        return false;

    //Also we need to check the lower diagonal in the left if it
    //has a queen
    if(!LowerDiagIsClear(n,x,y))
        return false;
    return true;
}
```

```
//Solve moves column by column trying to place a queen in
//every row until it leads to a valid solution.
bool solve(int n,int pos)
{
    if(pos>=n)
        return true;
    for(int i=0;i<n;i++)
    {
        if(issafe(n,i,pos))
        {
            board[i][pos]=1;
            if(solve(n,pos+1))
                return true;
            board[i][pos]=0;
            // If code reaches this point then the
            // current configuration failed and we need to place
            // the queen in a different row in the current column.
        }
    }
    return false;
}
int main()
{
    int n;
    cin>>n; // Take Input for getting the board size.
    int board[n][n];
    memset(board,0,sizeof board); //Sets the whole array as 0
    //0 - Signifies an empty position in the board
    //1 - Signifies a queen in that position.
    if(solve(n,0) == false)
    {
        cout<<"No Solution exists\n";
    }
    else
    {
        print(n);
    }
    return 0;
}
```

Time Analysis

Let **T(n)** be the time this recursive function takes to solve the given problem.
$T(n) = n*T(n-1) + O(n^2)$
T(n-1) is the time it takes for solve to complete for size **(n-1)**. This is called n times.
Within solve we try placing the queen at every position trying **n** possibilties and at each position we check if a previous queen is attacking it. Thus running an **$O(n^2)$** within the recursive function.
The solution to the above recursion is **O(n!)**.
Thus the worst case running time of our algorithm is **O(n!)**. This is only achieved if no solution exists and the solve function verifies every possible configuration. On an average since the function terminates upon receiving the first correct configuration, the code must have a much better running time.

O(n!) is definetly better than **$O(n^n)$**. To put things in perspective here are the values of **n!** and **n^n** for the first 15 natural numbers.

n	**n^n**	**n!**
1	1	1
2	4	2
3	27	6
4	256	24
5	3125	120
6	46656	720
7	823543	5040
8	16777216	40320
9	387420489	362880
10	10000000000	3628800
11	285311670611	39916800
12	8916100448256	479001600
13	302875106592253	6227020800
14	11112006825558016	87178291200

15	437893890380859375	1307674368000

Conclusion

The backtracking is a smart technique to reduce complex problems by selectively pruning candidates of answers. Although it is a non-polynomial technique it is used widely where other normal brute force fails.

Output