Experiment No. : 7

Title: 8 Queen Problem using Backtracking

Batch:A2 **Roll No.:16010421073** **Experiment No.: 7**

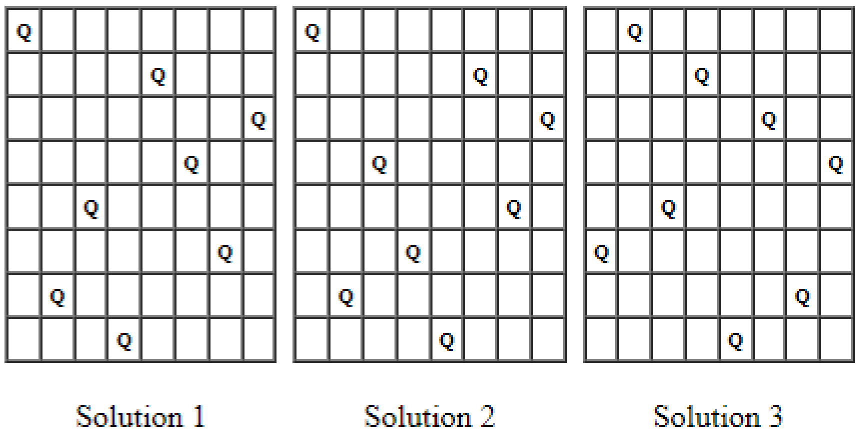
Aim: To Implement 8 Queen problem using Backtracking.

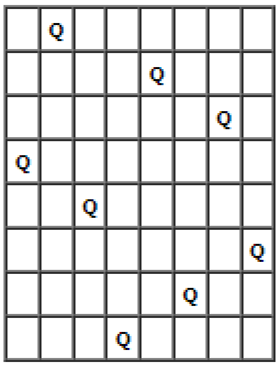
Algorithm of 8 Queen problem:
Let's go through the steps below to understand how this algorithm of solving the 8 queens problem using backtracking works:
START

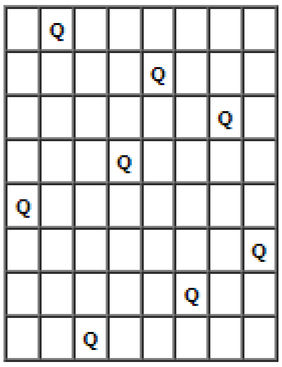
Step 1: Traverse all the rows in one column at a time and try to place the queen in that position.
Step 2: After coming to a new square in the left column, traverse to its left horizontal direction to see if any queen is already placed in that row or not. If a queen is found, then move to other rows to search for a possible position for the queen.
Step 3: Like step 2, check the upper and lower left diagonals. We do not check the right side because it's impossible to find a queen on that side of the board yet.
Step 4: If the process succeeds, i.e. a queen is not found, mark the position as '1' and move ahead.
Step 5: Recursively use the above-listed steps to reach the last column. Print the solution matrix if a queen is successfully placed in the last column.
Step 6: Backtrack to find other solutions after printing one possible solution.

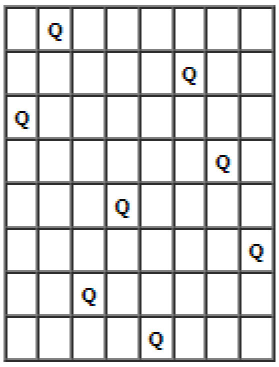
END
Output :

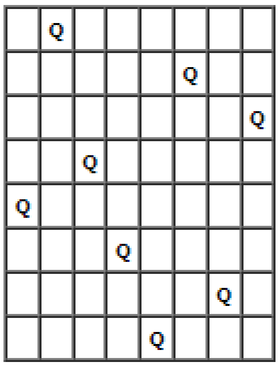
1. It asks in how many ways eight queens can be placed on a chess board so that no two attack each other.
2. Each of the twelve solutions shown on this page represents an equivalence class of solutions resulting from each other by rotating the chessboard and/or flipping it along one of its axes of symmetry.
3. Each of the equivalence classes represented by Solutions 1,2,...,11 consists of eight solutions. Since Solution 12 is invariant under rotating the chessboard by 180 degrees, its equivalence class consists of only four solutions.
4. Altogether, this page represents 92 solutions to the problem of eight queens; brute force shows that no other solutions exist.

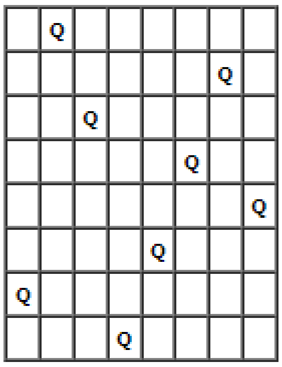


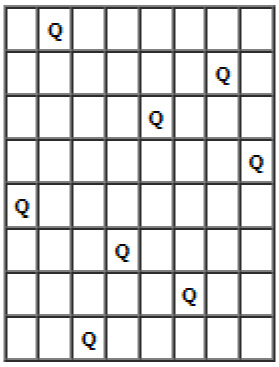

Solution 4

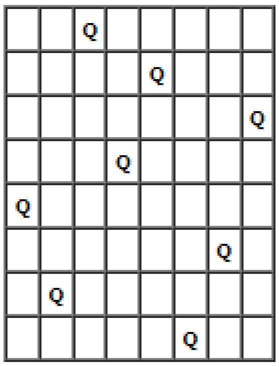

Solution 5

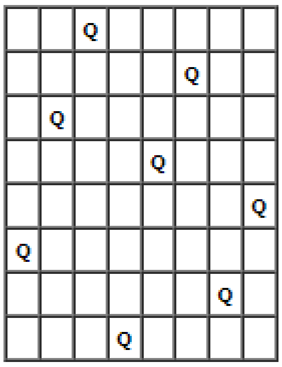

Solution 6

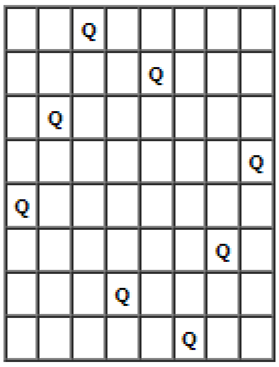

Solution 7


Solution 8


Solution 9


Solution 10


Solution 11


Solution 12

Working of 8 Queens Problem:

Problem Statement
Given an 8x8 chess board, you must place 8 queens on the board so that no two queens attack each other. Print all possible matrices satisfying the conditions with positions with queens marked with '1' and empty spaces with '0'. You must solve the 8 queens problem using backtracking.
Note 1: A queen can move vertically, horizontally and diagonally in any number of steps.
Note 2: You can also go through the N-Queen Problem for the general approach to solving this problem.

Solution

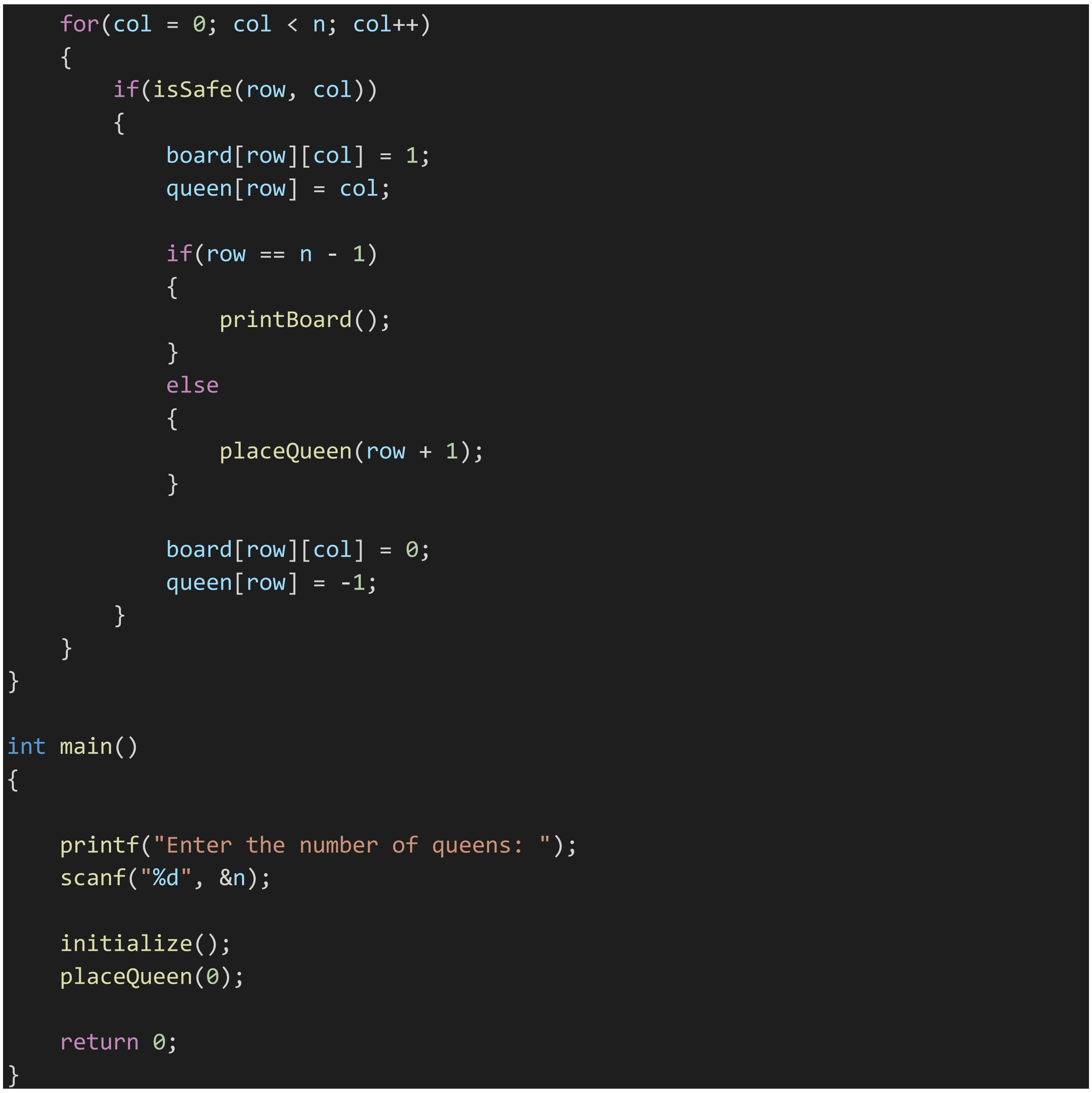
Derivation of 8 Queen problem:

Time complexity Analysis:
For the first column we will have N choices, then for the next column, we will have N-1 choices, and so on. Therefore the total time taken will be N*(N-1)*(N-2)...., which makes the time complexity to be O(N!).
the isPossible method takes O(n) time
for each invocation of loop in nQueenHelper, it runs for O(n) time
the isPossible condition is present in the loop and also calls nQueenHelper which is recursive
adding this up, the recurrence relation is:
T(n) = O(n^2) + n * T(n-1)
solving the above recurrence by iteration or recursion tree,
the time complexity of the nQueen problem is = O(N!)

Program(s) of 8 Queen problem:

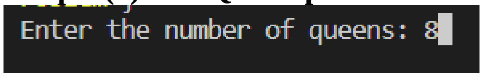
```
#include <stdio.h>
#include <stdlib.h>

#define N 20

int board[N][N];
int queen[N];
int n;

void initialize()
{
    int i, j;
    for(i = 0; i < n; i++)
    {
        queen[i] = -1;
        for(j = 0; j < n; j++)
            board[i][j] = 0;
    }
}
```

```c
int isSafe(int row, int col)
{
    int i, j;

    for(i = 0; i < row; i++)
    {
        if(board[i][col])
            return 0;
    }

    for(i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if(board[i][j])
            return 0;
    }

    for(i = row, j = col; i >= 0 && j < n; i--, j++)
    {
        if(board[i][j])
            return 0;
    }

    return 1;
}

void printBoard() {
    int i, j;

    printf("\nSolution: \n\n");
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
        {
            if(board[i][j] == 1)
                printf("Q ");
            else
                printf("_ ");
        }
        printf("\n");
    }
    printf("\n");
}

void placeQueen(int row)
{
    int col;
```

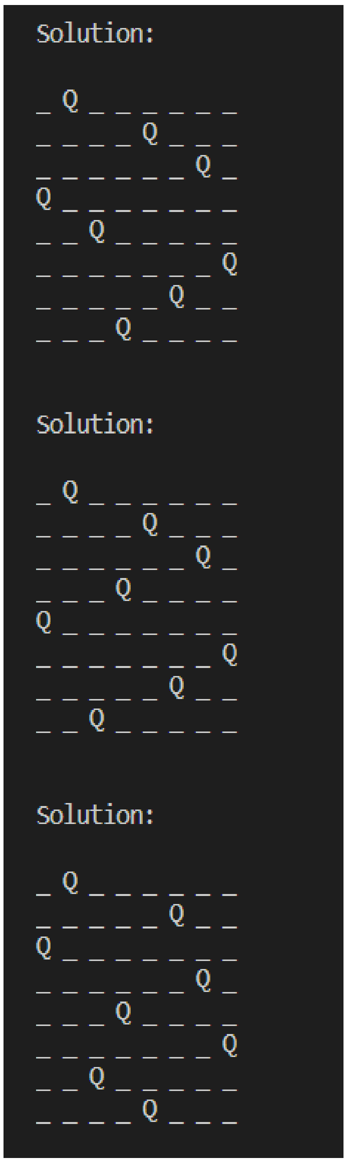


```
    for(col = 0; col < n; col++)
    {
        if(isSafe(row, col))
        {
            board[row][col] = 1;
            queen[row] = col;

            if(row == n - 1)
            {
                printBoard();
            }
            else
            {
                placeQueen(row + 1);
            }

            board[row][col] = 0;
            queen[row] = -1;
        }
    }
}

int main()
{

    printf("Enter the number of queens: ");
    scanf("%d", &n);

    initialize();
    placeQueen(0);

    return 0;
}
```

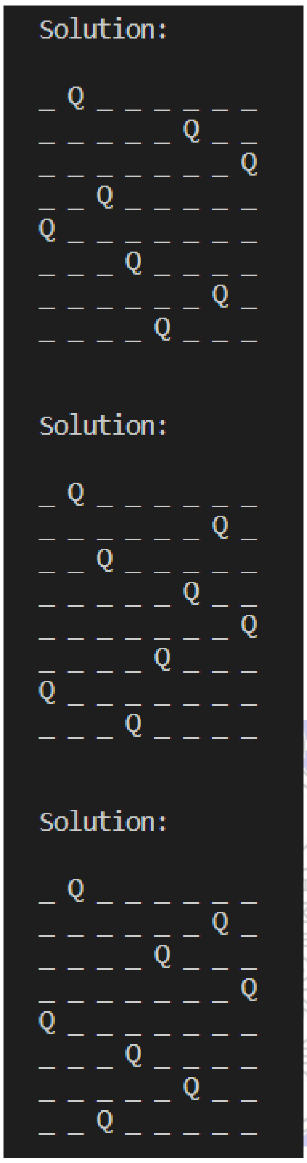
Output(o) of 8 Queen problem:



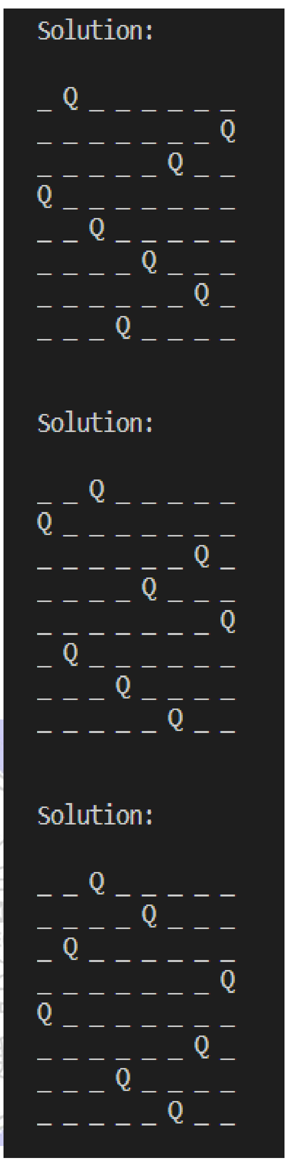
```
Enter the number of queens: 8
```



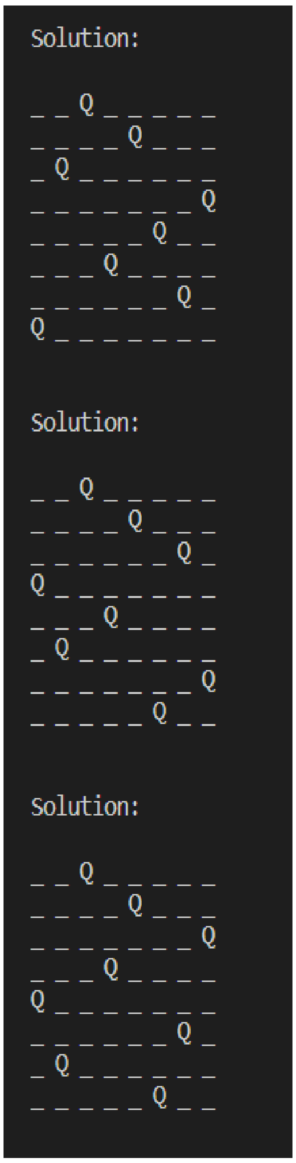
```
Solution:

_ Q _ _ _ _ _ _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ Q _
Q _ _ _ _ _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _

Solution:

_ Q _ _ _ _ _ _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ Q _
_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ Q _ _ _ _ _

Solution:

_ Q _ _ _ _ _ _
_ _ _ _ _ Q _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ Q _ _ _ _
_ _ _ _ _ _ _ Q
_ _ Q _ _ _ _ _
_ _ _ _ Q _ _ _
```



```
Solution:

_ Q _ _ _ _ _ _
_ _ _ _ _ Q _ _
_ _ _ _ _ _ _ Q
_ _ Q _ _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _

Solution:

_ Q _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ Q _ _ _ _ _
_ _ _ _ _ Q _ _
_ _ _ _ _ _ _ Q
_ _ _ _ Q _ _ _
Q _ _ _ _ _ _ _
_ _ _ Q _ _ _ _

Solution:

_ Q _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ _ Q
Q _ _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _
_ _ Q _ _ _ _ _
```



```
Solution:

_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
Q _ _ _ _ _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ Q _
_ _ _ Q _ _ _ _

Solution:

_ _ Q _ _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ _ Q
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _

Solution:

_ _ Q _ _ _ _ _
_ _ _ _ Q _ _ _
_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _
```



```
Solution:

_ _ Q _ _ _ _ _
_ _ _ _ Q _ _ _
_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ _ Q _
Q _ _ _ _ _ _ _

Solution:

_ _ Q _ _ _ _ _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ Q _
Q _ _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _

Solution:

_ _ Q _ _ _ _ _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ _ Q
_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ Q _ _ _ _ _ _
_ _ _ _ _ Q _ _
```

Post Lab Questions:- Describe the process of backtracking ? State the advantages of backtracking as against brute force algorithms?

Ans: Backtracking is a problem-solving algorithmic technique that involves searching through all possible solutions to find the one that satisfies a given set of constraints. The general process of backtracking can be described in the following steps:

1. **Define the problem:** Clearly define the problem to be solved and the constraints that must be satisfied.
2. **Generate possible solutions:** Generate all possible solutions to the problem, starting with an initial solution or state.
3. **Test the solution:** Test the current solution to see if it satisfies the constraints of the problem.

4. **Update the solution:** If the current solution does not satisfy the constraints, backtrack to the previous solution and modify it.
5. **Repeat steps 3 and 4:** Continue testing and modifying solutions until a satisfactory solution is found or all possible solutions have been exhausted.
6. **Output the solution:** Once a satisfactory solution has been found, output it as the solution to the problem.

Backtracking algorithms have several advantages over brute force algorithms, including:

- **Faster runtime:** Backtracking algorithms are generally faster than brute force algorithms because they intelligently narrow down the search space and avoid exploring solutions that are guaranteed to be invalid.
- **Less memory usage:** Backtracking algorithms often use less memory than brute force algorithms because they only store a single solution at a time, rather than generating and storing all possible solutions.
- **Ability to handle complex problems**: Backtracking algorithms are particularly well-suited to handling complex problems that have many constraints or that involve searching through a large solution space.

Conclusion: (Based on the observations):
Thus we successfully implemented 8 queen problem using backtracking technique with user input for queens.

Outcome:
CO3 : Implement Backtracking and Branch-and-bound algorithms.

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.