(ESTABLISHED UNDER KARNATAKA ACT NO. 16 OF 2013)
100 FEET RING ROAD, BENGALURU – 560 085, KARNATAKA, INDIA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SUBJECT: DESIGN AND ANALYSIS OF ALGORITHMS

CODE:UE17CS251

PROJECT REPORT

0N

"SUDOKU"

SUBMITTED BY

SUDHA NAIK	PES1201802416
PRIYANKA KUMARI	PES1201802433

SUDOKU

ABSTRACT

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column ,each row,and each of the 3x3 boxes(also called blocks or regions) contains the digit from 1 to 9,only one time each(that is, exclusively). The puzzle setter provides a partially completed grid.

INTRODUCTION

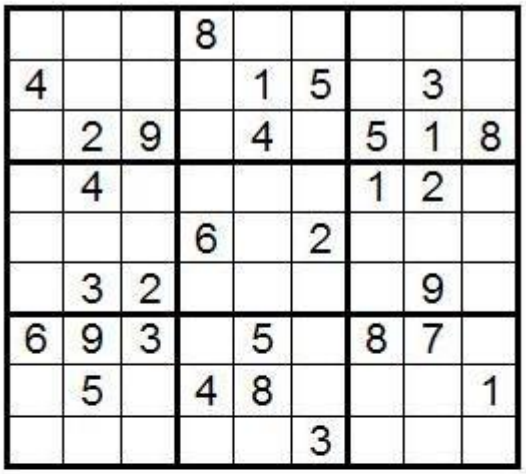
Sudoku is a puzzle that has enjoyed worldwide popularity since 2005. To solve a Sudoku puzzle, one needs to use a combination of logic and trial-and-error. More math is involved behind the scenes: combinatorics used in counting valid Sudoku grids, group theory used to describe ideas of when two grids are equivalent, and computational complexity with regards to solving Sudokus.

The game in its current form was invented by American Howard Garns in 1979 and published by Dell Magazines as "Numbers in Place." In 1984, Maki Kaji of Japan published it in the magazine of his puzzle company Nikoli. He gave the game its modern name of Sudoku, which means "Single Numbers." The puzzle became popular in Japan and was discovered there by New Zealander Wayne Gould, who then wrote a computer program that would generate Sudokus. He was able to get some puzzles printed in the London newspaper *The Times* beginning in 2004. Soon after, Sudoku-fever swept England. The puzzle finally became popular in the U.S. in 2005. It has become a regular feature in many newspapers and magazines and is enjoyed by people all over the globe.

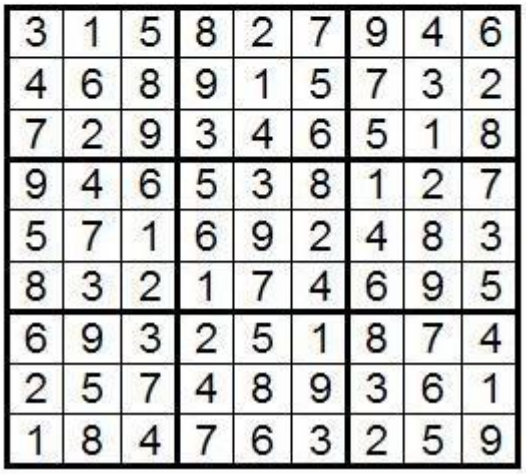
The standard version of Sudoku consists of a 9×9 square grid containing 81 **cells**. The grid is subdivided into nine 3×3 **blocks**. Some of the 81 cells are filled in with numbers from the set {1,2,3,4,5,6,7,8,9}. These filled-in cells are called **givens**. The goal is to fill in the whole grid using the nine digits so that each row, each column, and each block contains each number exactly once. We call this constraint on the rows, columns, and blocks the **One Rule**.

The above-described puzzle is called a Sudoku of rank 3. A Sudoku of **rank n** is an $n^2 \times n^2$ square grid, subdivided into n^2 blocks, each of size n×n. The numbers used to fill the grid in are 1, 2, 3, ..., n^2, and the One Rule still applies.

Here is an example of a Sudoku puzzle and its solution:



			8					
4				1	5		3	
	2	9		4		5	1	8
	4					1	2	
			6		2			
	3	2					9	
6	9	3		5		8	7	
	5		4	8				1
					3			



3	1	5	8	2	7	9	4	6
4	6	8	9	1	5	7	3	2
7	2	9	3	4	6	5	1	8
9	4	6	5	3	8	1	2	7
5	7	1	6	9	2	4	8	3
8	3	2	1	7	4	6	9	5
6	9	3	2	5	1	8	7	4
2	5	7	4	8	9	3	6	1
1	8	4	7	6	3	2	5	9

we can also conclude that a Sudoku is considered wrongly filled if it satisfies any of these criteria:

1. Any row contains the same number more than once.
2. Any column contains the same number more than once.
3. Any 3x3 sub-matrix has the same number more than once.

In backtracking, we first start with a sub-solution and if this sub-solution doesn't give us a correct final answer, then we just come back and change our sub-solution. We are going to solve our Sudoku in a similar way. The steps which we will follow are:

- If there are no unallocated cells, then the Sudoku is already solved. We will just return true.

- Or else, we will fill an unallocated cell with a digit between 1 to 9 so that there are no conflicts in any of the rows, columns, or the 3x3 sub-matrices.

- Now, we will try to fill the next unallocated cell and if this happens successfully, then we will return true.

- Else, we will come back and change the digit we used to fill the cell. If there is no digit which fulfills the need, then we will just return false as there is no solution of this Sudoku.

NAÏVE ALGORITHM

Generate all possible configurations of number from 1 to 9 fill the empty cells try the every configuration one by one until the correct configuration is found.

This is obviously a tedious process.To avoid this we use backtracking algorithm.

BACKTRACKING ALGORITHM

1. Assign number one by one to empty cells
2. Before assigning we check if it is safe or not.
3. After checking safety,we assign a number, and recursively check whether this assignment leads to a solution or not
    - Find the position i.e. row,column of an unassigned cell
    - If there is none
      Return true
    - For digits from 1-9
      If there is no conflict for digit at row,column then assign digit to row,column and recursively try to fill in the rest of the grid

      if recursion is successful
        return true
      else
        remove digit and try another

C code

```
#include <stdio.h>


#define SIZE 9


int matrix[9][9] = {
   {6,5,1,8,7,3,0,9,0},
   {0,0,0,2,5,0,0,0,8},
   {9,8,0,1,0,4,3,5,7},
   {1,0,5,0,0,0,0,0,0},
   {4,0,0,0,0,0,0,0,2},
   {0,0,0,0,0,0,5,0,3},
   {5,7,8,3,0,1,0,2,6},
   {2,0,0,0,4,8,9,0,0},
   {0,9,0,6,2,5,0,8,1}
   };
void print_sudoku()
{
   int i,j;
   for(i=0;i<SIZE;i++)
   {
      for(j=0;j<SIZE;j++)
```

```
        {
            printf("%d\t",matrix[i][j]);
        }
        printf("\n\n");
    }
}
int number_unassigned(int *row, int *col)
{
    int num_unassign = 0;
    int i,j;
    for(i=0;i<SIZE;i++)
    {
        for(j=0;j<SIZE;j++)
        {
            //cell is unassigned
            if(matrix[i][j] == 0)
            {
                *row = i;
                *col = j;
                num_unassign = 1;
                return num_unassign;
            }
```

```
        }
    }
     return num_unassign;
}

int is_safe(int n, int r, int c)
{
    int i,j;
    //checking in row
    for(i=0;i<SIZE;i++)
    {
        //there is a cell with same value
        if(matrix[r][i] == n)
            return 0;
    }
    //checking column
    for(i=0;i<SIZE;i++)
    {
          //there is a cell with the value equal to i
        if(matrix[i][c] == n)
            return 0;
    }
```

```
    int row_start = (r/3)*3;
    int col_start = (c/3)*3;
    for(i=row_start;i<row_start+3;i++)
    {
       for(j=col_start;j<col_start+3;j++)
       {
          if(matrix[i][j]==n)
          return 0;
       }
    }
    return 1;
}
int solve_sudoku()
{
    int row;
    int col;
    if(number_unassigned(&row, &col) == 0)
     return 1;
    int n,i;
    //number between 1 to 9
    for(i=1;i<=SIZE;i++)
    {
```

```
        if(is_safe(i, row, col))
        {
            matrix[row][col] = i;
            //backtracking
            if(solve_sudoku())
                return 1;
            matrix[row][col]=0;
        }
    }
    return 0;
}

int main()
{
    if (solve_sudoku())
        print_sudoku();
    else
        printf("No solution\n");
    return 0;
}
```

Explanation of the code

Initially, we are just making a matrix for Sudoku and filling its unallocated cells with 0. Thus, the matrix contains the Sudoku problem and the cells with value 0 are vacant cells.

print_sudoku() → This is just a function to print the matrix.

number_unassigned → This function finds a vacant cell and makes the variables 'row' and 'col' equal to indices of that cell. In C, we have used pointers to change the value of the variables (row, col) passed to this function (pass by reference). So, this function tells us if there is any unallocated cell or not. And if there is any unallocated cell then this function also tells us the indices of that cell.

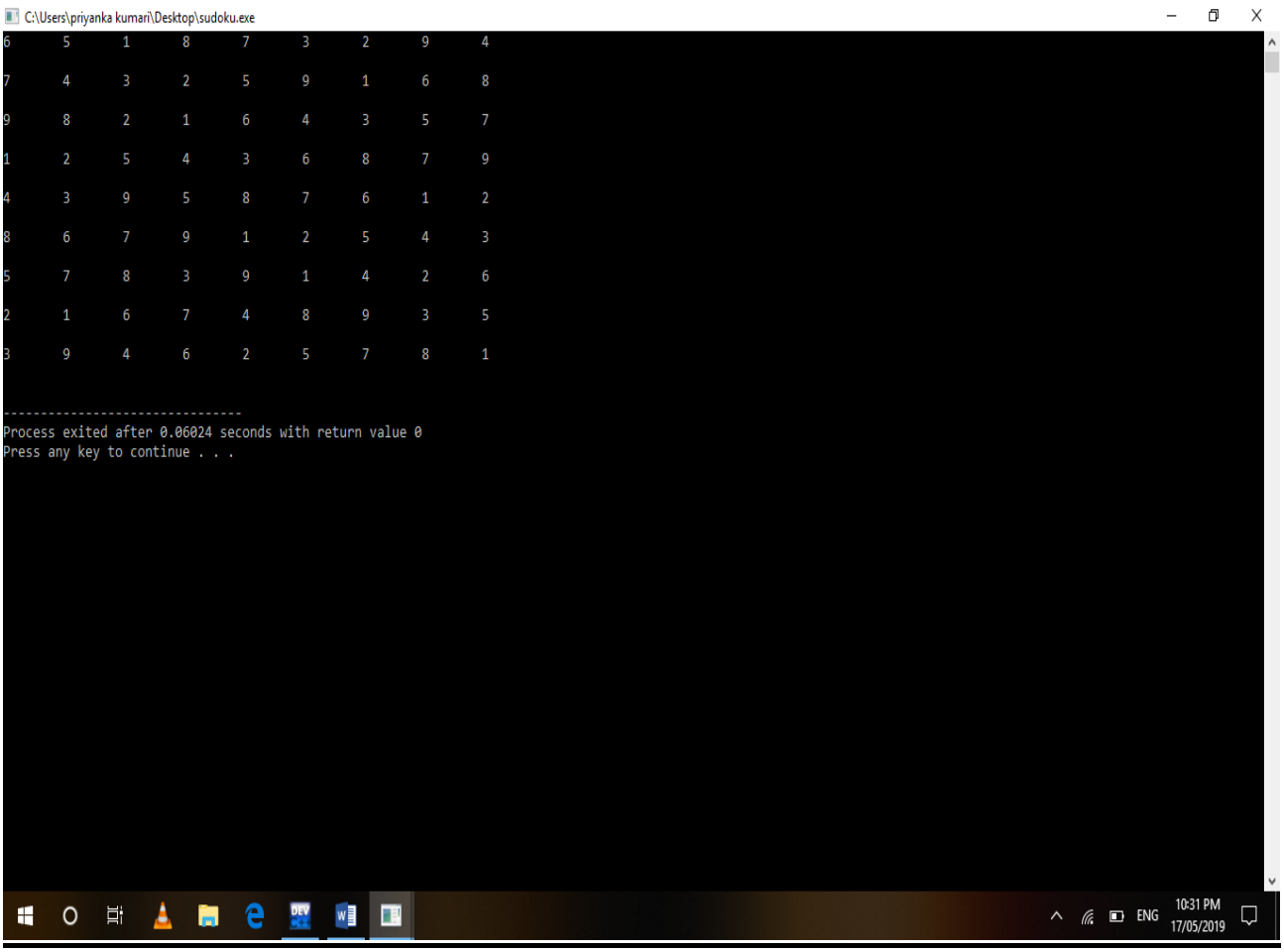
is_safe(int n, int r, int c) → This function checks if we can put the value 'n' in the cell (r, c) or not. We are doing so by first checking if there is any cell in the row 'r' with the value 'n' or not **– if(matrix[r][i] == n)**. Then we are checking if there is any cell with the value 'n' in the column 'c' or not **– if(matrix[i][c] == n).** And finally, we are checking for the sub-matrix**. (r/3)*3** gives us the starting index of the row r. For example, if the value of 'r' is 2 then it is in the sub-matrix which starts from (0, 0). Similarly, we are getting the value of starting column **by (c/3)*3**. Thus, if a cell is (2,2), then this cell will be in a sub-matrix which starts from (0,0) and we are getting this value by **(c/3)*3** and **(r/3)*3.** After getting the starting indices, we can easily iterate over the sub-matrix to check if the we can put the value 'n' in that sub-matrix or not.

solve_sudoku() → This is the actual function which solves the Sudoku and uses backtracking. We are first checking if there is any unassigned cell or not

by using the **number_unassigned** function and if there is no unassigned cell then the Sudoku is solved**. number_unassigned** function also gives us the indices of the vacant cell. Thus, if there is any vacant cell then we will try to fill this cell with a value between 1 to 9. And we will use the **is_safe** to check if we can fill a particular value in that cell or not. After finding a value, we will try to solve the rest of the Sudoku **solve_sudoku**. If this value fails to solve the rest, we will come back and try another value for this cell **matrix[row][col]=0;**. The loop will try other values in the cell.

Time complexicity of this algorithm is O(N^(N^2))

SNAPSHOTS


C:\Users\priyanka kumari\Desktop\sudoku.exe
6 5 1 8 7 3 2 9 4
7 4 3 2 5 9 1 6 8
9 8 2 1 6 4 3 5 7
1 2 5 4 3 6 8 7 9
4 3 9 5 8 7 6 1 2
8 6 7 9 1 2 5 4 3
5 7 8 3 9 1 4 2 6
2 1 6 7 4 8 9 3 5
3 9 4 6 2 5 7 8 1

Process exited after 0.06024 seconds with return value 0
Press any key to continue . . .
ENG
10:31 PM
17/05/2019