SUDOKU:

Sudoku is a number-placement puzzle where the objective is to fill a square grid of size ‘n’ with numbers between 1 to ‘n’. The numbers must be placed so that each column, each row, and each of the sub-grids (if any) contains all of the numbers from 1 to ‘n’.

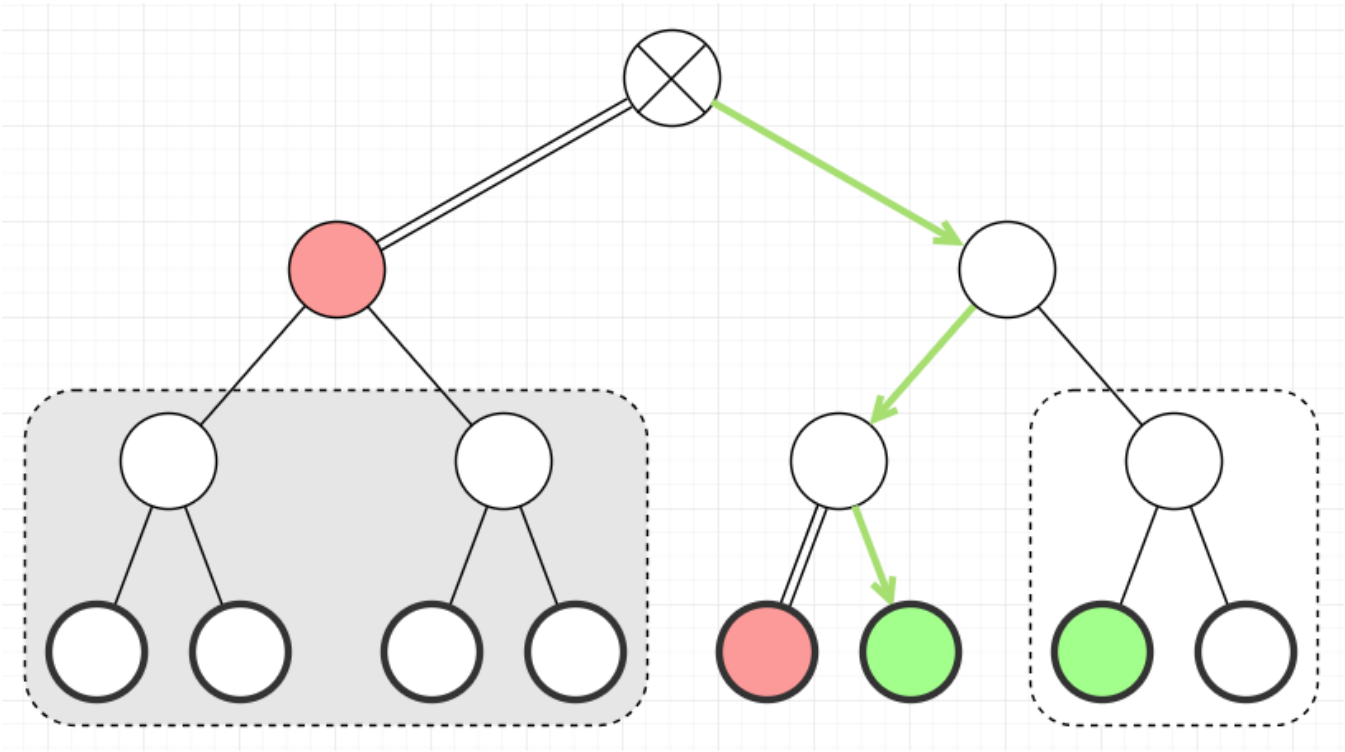
The most common Sudoku puzzles use a 9x9 grid. The grids are partially filled (with hints) to ensure a solution can be reached. Sudoku is a logic-based puzzle. Solving one requires a series of logical moves and might require a bit of guesswork.

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

ABSTRACT:

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

A backtracking algorithm can be thought of as a tree of possibilities. In this tree, the root node is the original problem, each node is a candidate and the leaf nodes are possible solution candidates. We traverse the tree depth-first from root to a leaf node to solve the problem.



The tree diagram shows 2 groups — Unexplored Possible Candidates and Impossible Candidates.

Unexplored Possible Candidates marks nodes that were never explored. Some of them could have been viable candidates, leading to another solution. Since we never explored them, we can never know. Problems where multiple solutions are acceptable won't have this group.

Impossible Candidates groups are the obvious ones. It contains nodes which have a failed candidate node as one of their ancestor nodes. None of the nodes in this group are candidate nodes and none of the leaf nodes are solution nodes.

PROBLEM DEFINITION:

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

APPROACH FOR SOLVING SUDOKU USING RECURSIVE BACKTRACKING ALGORTIHM:

1. We can solve Sudoku by one by one assigning numbers to empty cell.
2. Before assigning a number, we need to confirm that the same number is not present in current row, current column and current 3x3 grid.
3. If the number is not present in respective row, column or subgrid, we can assign the number, and recursively check if this assignment lead to a solution or not.If the assignment doesn't lead to a solution, then we try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, we return false.

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

ALGORITHM:

1. Make a list of all the empty spots.

2. Select a spot and place a number, between 1 and ‘n’, in it and validate the candidate grid.

3. If any of the constraints fails, abandon candidate and repeat step 2 with the next number. Otherwise, check if the goal is reached.

4. If a solution is found, stop searching. Otherwise, repeat steps 2 to 4.

IMPLEMENTATION:

All the cells of completely solved sudoku array must have assigned valid values. Sudoku have unassigned values i.e. 0 in its cells, which is considered to be incomplete or wrong.

```
int main() {

	int grid[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
		{5, 2, 0, 0, 0, 0, 0, 0, 0},
		{0, 8, 7, 0, 0, 0, 0, 3, 1},
		{0, 0, 3, 0, 1, 0, 0, 8, 0},
		{9, 0, 0, 8, 6, 3, 0, 0, 5},
		{0, 5, 0, 0, 9, 0, 6, 0, 0},
		{1, 3, 0, 0, 0, 0, 2, 5, 0},
		{0, 0, 0, 0, 0, 0, 0, 7, 4},
		{0, 0, 5, 2, 0, 6, 3, 0, 0}};

	if (solve(grid)) {
		print_grid(grid);
	} else {
		printf("no solution");
	}

	return 0;
}
```

Below three methods are used top check, if number is present in current row, current column and current 3x3 subgrid or not.

```
int is_exist_row(int grid[N][N], int row, int num){
	for (int col = 0; col < 9; col++) {
		if (grid[row][col] == num) {
			return 1;
		}
	}
	return 0;
}
```

```
int is_exist_col(int grid[N][N], int col, int num) {
	for (int row = 0; row < 9; row++) {
		if (grid[row][col] == num) {
			return 1;
		}
	}
	return 0;
}
```

```
int is_exist_box(int grid[N][N], int startRow, int startCol, int num) {
	for (int row = 0; row < 3; row++) {
		for (int col = 0; col < 3; col++) {
			if (grid[row + startRow][col + startCol] == num) {
				return 1;
			}
		}
	}
	return 0;
}
```

is_safe_num() method uses above three methods to check if it is safe to assign number in the cell. If any of above three method returns true, it means particular number is not allowed in the cell.

```
int is_safe_num(int grid[N][N], int row, int col, int num) {
	return !is_exist_row(grid, row, num)
			&& !is_exist_col(grid, col, num)
			&& !is_exist_box(grid, row - (row % 3), col - (col %3), num);
}
```

find_unassigned() method checks cells in each row one by one and picks up first cell with unassigned value.

```
int find_unassigned(int grid[N][N], int *row, int *col) {
	for (*row = 0; *row < N; (*row)++) {
		for (*col = 0; *col < N; (*col)++) {
			if (grid[*row][*col] == 0) {
				return 1;
			}
		}
	}
	return 0;
}
```

solve() method starts traversing from top left cell to the right side. It checks cells in each row and column and picks up the first unassigned cell by calling find_unassigned() function. It checks for allowed numbers from 1 to 9, assign the first possible option to cell and again call solve() method. It follows the same process recursively. If is_safe_num() function returns false, control will backtrack to previously assigned cell and try for another number there. If it finds any allowed number, then it assigns this new number to the cell and process continues again otherwise it will backtrack to the previously assigned cell and the process continues until the solution is found.

```
int solve(int grid[N][N]) {

	int row = 0;
	int col = 0;

	if (!find_unassigned(grid, &row, &col)){
		return 1;
	}

	for (int num = 1; num <= N; num++ ) {

		if (is_safe_num(grid, row, col, num)) {
			grid[row][col] = num;

			if (solve(grid)) {
				return 1;
			}

			grid[row][col] =0;
		}
	}

	return 0;
}
```

print_grid() method is used to display 9x9 array.

```
void print_grid(int grid[N][N]) {
	for (int row = 0; row < N; row++) {
		for (int col = 0; col < N; col++) {
			printf("%2d", grid[row][col]);
		}
		printf("\n");
	}
}
```

The time complexity for the generation of this sudoku is O(n^m) where n is the range of the numbers and m is the number of unassigned values in the start.

CONCLUSION:

In conclusion, we say that backtracking algorithm is one of the most popular algorithms for Sudoku solving. This algorithm ensures that a solution is guaranteed (as long as the puzzle is valid) and solving time is mostly unrelated to degree of difficulty. The algorithm (and therefore the program code) is simpler than other algorithms, especially compared to strong algorithms that ensure a solution to the most difficult puzzles.The disadvantage of this method is that the solving time may be slow compared to algorithms modeled after deductive methods.

Backtracking can be used to solve puzzles or problems include: Puzzles such as eight queens puzzle, crosswords, verbal arithmetic and Peg Solitaire. Combinatorial optimization problems such as parsing and the knapsack problem.