**Rajan Gautam**

**19BCP101**

**Div. II, CE 19**

**SOT, PDEU**

**Pandit Deendayal Energy University**

**School of Technology**

**Design & Analysis of Algorithm (20CP209P)**

**B. Tech - Computer Science & Engineering (Sem-IV)**

**Table of Contents**

[**Lab 6 Assignment: Solving Optimization Problems through Backtracking technique and draw state-space diagram. 2**](#_Toc70271634)

[**AIM: To Solve Optimization Problems through Backtracking technique and draw state-space diagram. 2**](#_Toc70271635)

[**Problem 1: 2**](#_Toc70271636)

[**CODE: 2**](#_Toc70271637)

[**OUTPUT: 5**](#_Toc70271638)

[**TIME COMPLEXITY: 5**](#_Toc70271639)

[**Problem-2: 6**](#_Toc70271640)

[**CODE: 6**](#_Toc70271641)

[**OUTPUT: 8**](#_Toc70271642)

[**TIME COMPLEXITY: 8**](#_Toc70271643)

[**Problem 3: 9**](#_Toc70271644)

[**CODE: 9**](#_Toc70271645)

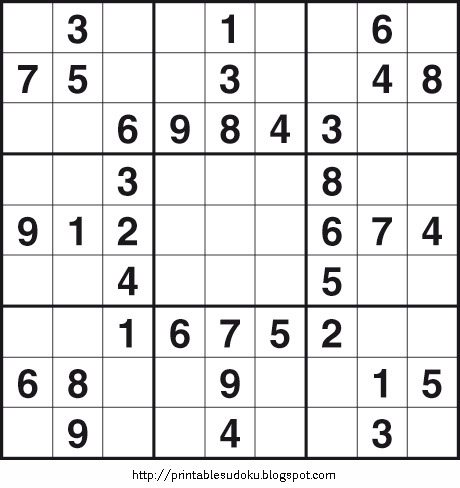
[**OUTPUT: 11**](#_Toc70271646)

[**TIME COMPLEXITY: 11**](#_Toc70271647)

# Lab 6 Assignment: Solving Optimization Problems through Backtracking technique and draw state-space diagram.

# AIM: To Solve Optimization Problems through Backtracking technique and draw state-space diagram.

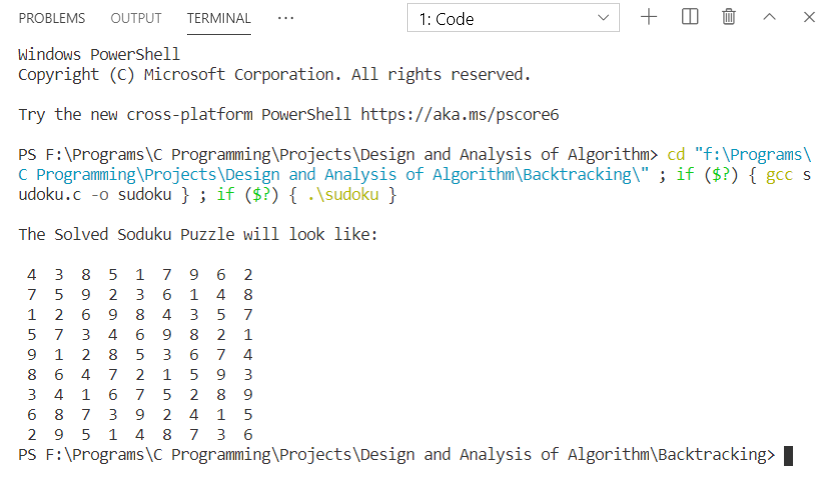
Problem 1: **Write a C/C++ program to solve given sudoku puzzle using Backtracking Approach. Analyze its time complexity.**



### CODE:

1. **#include <stdio.h>**
2. **#include <stdbool.h>**
3. **#define N 9**
5. **// Function to print result**
6. **void PrintSolution(int sudokuBoard[N][N])**
7. **{**
8. **for (int i = 0; i < N; i++)**
9. **{**
10. **for (int j = 0; j < N; j++)**
11. **{**
12. **printf(" %d ", sudokuBoard[i][j]);**
13. **}**
14. **printf("\n");**
15. **}**
16. **}**
18. **// Function to check whether it's legal or not**
19. **bool SudokuCheck(int sudokuBoard[N][N], int row, int col, int num)**
20. **{**
21. **// If we find the same number in similar row, then we will return 0**
22. **for (int i = 0; i < N; i++)**
23. **{**
24. **if (sudokuBoard[row][i] == num)**
25. **{**
26. **return false;**
27. **}**
28. **}**
29. **// If we find the same number in similar column, then we will return 0**
30. **for (int i = 0; i < N; i++)**
31. **{**
32. **if (sudokuBoard[i][col] == num)**
33. **{**
34. **return false;**
35. **}**
36. **}**
37. **// If we find the same number in particular 3x3 matrix, then we will return 0**
38. **int startRow = row - row % 3;**
39. **int startCol = col - col % 3;**
40. **for (int i = 0; i < 3; i++)**
41. **{**
42. **for (int j = 0; j < 3; j++)**
43. **{**
44. **if (sudokuBoard[i + startRow][j + startCol] == num)**
45. **{**
46. **return 0;**
47. **}**
48. **}**
49. **}**
50. **return true;**
51. **}**
53. **// Funtion to solve Sudoku puzzle**
54. **bool SudokuSolver(int sudokuBoard[N][N], int row, int col)**
55. **{**
56. **// If we have reached the 8th row and 8th column, we are returing true to avoid backtracking**
57. **if (row == N - 1 && col == N)**
58. **{**
59. **return true;**
60. **}**
61. **// If column becomes 9th, we set it to 0 and move to next row**
62. **if (col == N)**
63. **{**
64. **row++;**
65. **col = 0;**
66. **}**
67. **// If current position value > 0, we iterate for next column**
68. **if (sudokuBoard[row][col] > 0)**
69. **{**
70. **return SudokuSolver(sudokuBoard, row, col + 1);**
71. **}**
73. **for (int num = 1; num <= N; num++)**
74. **{**
75. **if (SudokuCheck(sudokuBoard, row, col, num))**
76. **{**
77. **// Assigning the number**
78. **sudokuBoard[row][col] = num;**
80. **//Checking for possibility with next column**
81. **if (SudokuSolver(sudokuBoard, row, col + 1))**
82. **{**
83. **return true;**
84. **}**
85. **}**
86. **// Removing assigned number since our assumption was wrong. We will go for next assumption with different value**
87. **sudokuBoard[row][col] = 0;**
88. **}**
89. **return false;**
90. **}**
92. **int main()**
93. **{**
94. **int unsolvedSoduku[N][N] = {{0, 3, 0, 0, 1, 0, 0, 6, 0},**
95. **{7, 5, 0, 0, 3, 0, 0, 4, 8},**
96. **{0, 0, 6, 9, 8, 4, 3, 0, 0},**
97. **{0, 0, 3, 0, 0, 0, 8, 0, 0},**
98. **{9, 1, 2, 0, 0, 0, 6, 7, 4},**
99. **{0, 0, 4, 0, 0, 0, 5, 0, 0},**
100. **{0, 0, 1, 6, 7, 5, 2, 0, 0},**
101. **{6, 8, 0, 0, 9, 0, 0, 1, 5},**
102. **{0, 9, 0, 0, 4, 0, 0, 3, 0}};**
104. **if (SudokuSolver(unsolvedSoduku, 0, 0) == true)**
105. **{**
106. **printf("\nThe Solved Soduku Puzzle will look like: \n\n");**
107. **PrintSolution(unsolvedSoduku);**
108. **}**
109. **else**
110. **{**
111. **printf("No Solution Exists");**
112. **}**
114. **return 0;**
115. **}**

### OUTPUT:



### TIME COMPLEXITY:

Time Complexity of Sudoku Puzzle using Backtracking 🡪 O(N^M)

where N = Size of Matrix (Generally 9 in classic Sudoku)

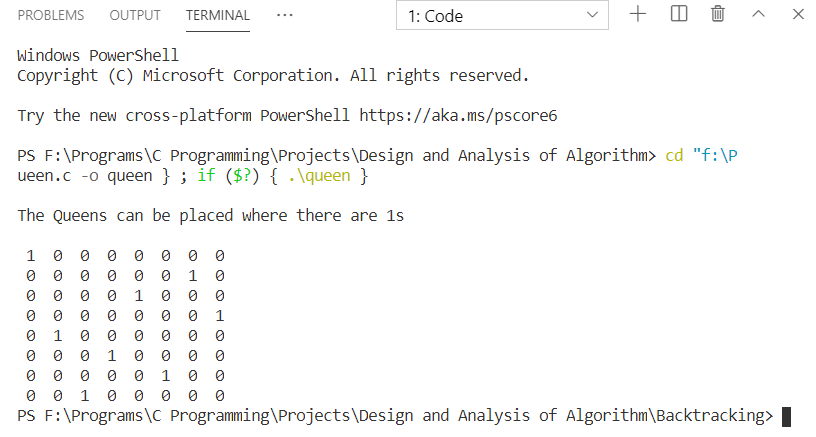
M = Number of blank spaces

Problem-2: **Write a C/C++ program to solve 8-Queen's problem. Analyze the time required by algorithm using Backtracking to solve the problem.**

### CODE:

1. **#include <stdio.h>**
2. **#include <stdbool.h>**
3. **#define N 8**
5. **// Function to print result**
6. **void PrintSolution(int chessBoard[N][N])**
7. **{**
8. **for (int i = 0; i < N; i++)**
9. **{**
10. **for (int j = 0; j < N; j++)**
11. **{**
12. **printf(" %d ", chessBoard[i][j]);**
13. **}**
14. **printf("\n");**
15. **}**
16. **}**
18. **// Function to check whether it's a safe move or not**
19. **bool SafeMove(int chessBoard[N][N], int row, int col)**
20. **{**
21. **int i, j;**
22. **// Checking for the left column**
23. **for (i = 0; i < col; i++)**
24. **{**
25. **if (chessBoard[row][i])**
26. **{**
27. **return false;**
28. **}**
29. **}**
30. **// Checking for the left top diagonal**
31. **for (i = row, j = col; i >= 0 && j >= 0; i--, j--)**
32. **{**
33. **if (chessBoard[i][j])**
34. **{**
35. **return false;**
36. **}**
37. **}**
38. **// Checking for the left bottom diagonal**
39. **for (i = row, j = col; j >= 0 && i < N; i++, j--)**
40. **{**
41. **if (chessBoard[i][j])**
42. **{**
43. **return false;**
44. **}**
45. **}**
46. **return true;**
47. **}**
49. **// Function to solve N Queen problem**
50. **bool QueenSolver(int chessBoard[N][N], int col)**
51. **{**
52. **// If all Queens are already placed at correct position, then simply return true**
53. **if (col >= N)**
54. **{**
55. **return true;**
56. **}**
58. **// Consider the column at 0th position and try placing the Queen in all rows one by one**
59. **for (int i = 0; i < N; i++)**
60. **{**
61. **// If safe, place the Queen**
62. **if (SafeMove(chessBoard, i, col))**
63. **{**
64. **chessBoard[i][col] = 1;**
66. **// Recursive call to other Queens**
67. **if (QueenSolver(chessBoard, col + 1))**
68. **{**
69. **return true;**
70. **}**
71. **chessBoard[i][col] = 0;**
72. **}**
73. **}**
74. **return false;**
75. **}**
77. **int main()**
78. **{**
79. **int board[N][N] = {{0, 0, 0, 0, 0, 0, 0, 0},**
80. **{0, 0, 0, 0, 0, 0, 0, 0},**
81. **{0, 0, 0, 0, 0, 0, 0, 0},**
82. **{0, 0, 0, 0, 0, 0, 0, 0},**
83. **{0, 0, 0, 0, 0, 0, 0, 0},**
84. **{0, 0, 0, 0, 0, 0, 0, 0},**
85. **{0, 0, 0, 0, 0, 0, 0, 0},**
86. **{0, 0, 0, 0, 0, 0, 0, 0}};**
88. **if (QueenSolver(board, 0) == true)**
89. **{**
90. **printf("\nThe Queens can be placed where there are 1s \n\n");**
91. **PrintSolution(board);**
92. **}**
93. **else**
94. **{**
95. **printf("Solution does not exist");**
96. **return false;**
97. **}**
98. **return 0;**
99. **}**

### OUTPUT:



### TIME COMPLEXITY:

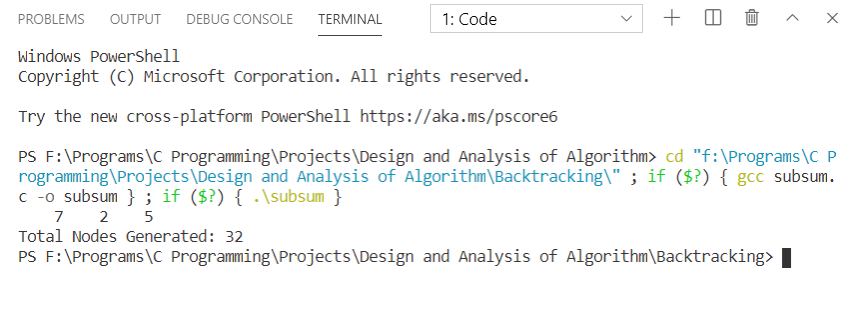
Time Complexity of N Queen's Problem ==> O(N!)

Problem 3: **Write a C/C++ program to solve Sum of subset problem for given array A= {7,3,2,5,8}, Sum=14. Print all the subsets for the given sum.**

### CODE:

1. **#include <stdio.h>**
2. **#include <stdlib.h>**
3. **#define N 5**
5. **int totalNodes;**
7. **void PrintSolution(int A[], int size)**
8. **{**
9. **for (int i = 0; i < size; i++)**
10. **{**
11. **printf("%\*d", N, A[i]);**
12. **}**
13. **printf("\n");**
14. **}**
16. **/\***
17. **A -> Set Vector**
18. **V -> Tuplet Vector**
19. **ASize -> Set Size**
20. **VSize -> Tuplet Size so Far**
21. **Sum -> Sum so far**
22. **Nodes -> Nodes Count**
23. **m -> Sum to be found**
24. **\*/**
26. **void subsetSum(int A[], int V[], int ASize, int VSize, int sum, int Nodes, int m)**
27. **{**
28. **totalNodes++;**
29. **if (m == sum)**
30. **{**
31. **// We found subset**
32. **PrintSolution(V, VSize);**
34. **// Consider next item to find another combination**
35. **subsetSum(A, V, ASize, VSize - 1, sum - A[Nodes], Nodes + 1, m);**
36. **return;**
37. **}**
38. **else**
39. **{**
40. **// Generate Nodes along the breadth**
41. **for (int i = Nodes; i < ASize; i++)**
42. **{**
43. **V[VSize] = A[i];**
45. **// Consider next level node along depth**
46. **subsetSum(A, V, ASize, VSize + 1, sum + A[i], i + 1, m);**
47. **}**
48. **}**
49. **}**
51. **// A Function to generate subset**
52. **void SubsetGenerator(int A[], int size, int m)**
53. **{**
54. **int \*TupletVector = (int \*)malloc(size \* sizeof(int));**
56. **subsetSum(A, TupletVector, size, 0, 0, 0, m);**
58. **free(TupletVector);**
59. **}**
61. **int main()**
62. **{**
63. **// Getting the input weights**
64. **int Weight[] = {7, 3, 2, 5, 8};**
66. **SubsetGenerator(Weight, N, 14);**
68. **printf("Total Nodes Generated: %d", totalNodes);**
70. **return 0;**
71. **}**

### OUTPUT:



### TIME COMPLEXITY:

Time Complexity of N Queen's Problem ==> O(2N)

**Link:** <https://github.com/rgautam320/Design-and-Analysis-of-Algorithm-Lab/tree/master/Lab_6_Backtracking>