

# BACKTRACKING

## EXPERIMENT-1

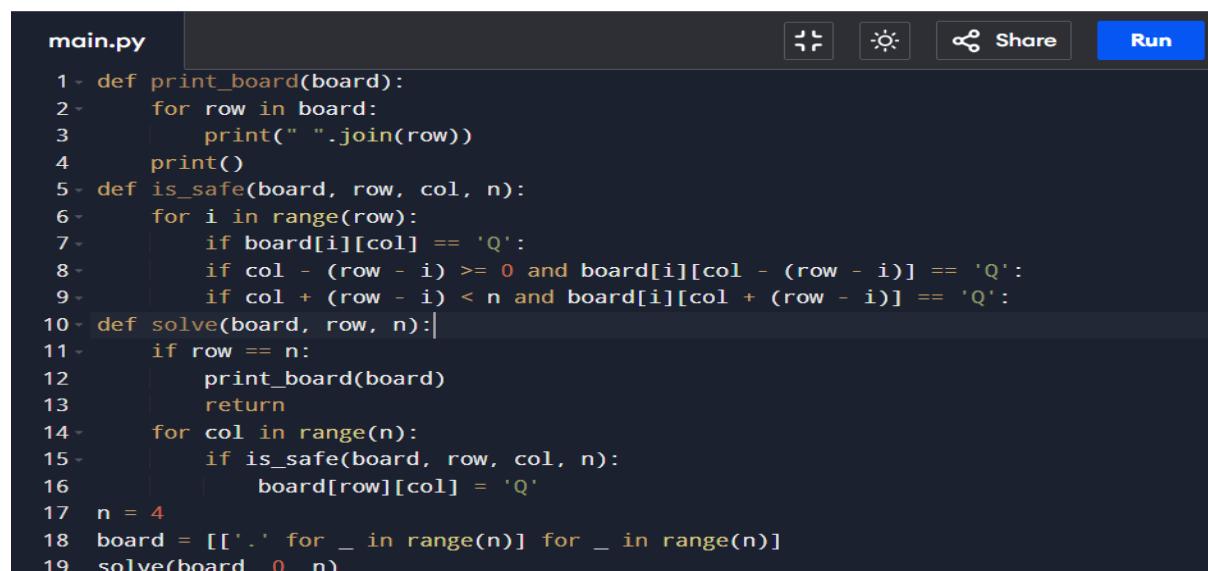
### AIM:

To place N queens on a chessboard so that no two queens attack each other using backtracking.

### PROCEDURE:

- Place queens row by row.
- Check if the position is safe (no other queen in same column or diagonal).
- If safe, place the queen and move to next row.
- If not safe, backtrack and try another column.

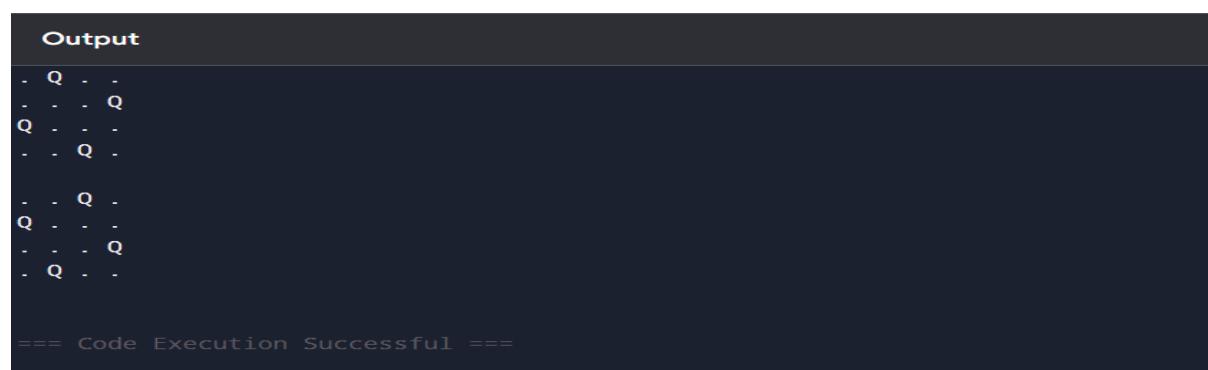
### PROGRAM:



```
main.py
```

```
1 def print_board(board):
2     for row in board:
3         print(" ".join(row))
4     print()
5 def is_safe(board, row, col, n):
6     for i in range(row):
7         if board[i][col] == 'Q':
8             if col - (row - i) >= 0 and board[i][col - (row - i)] == 'Q':
9                 if col + (row - i) < n and board[i][col + (row - i)] == 'Q':
10                def solve(board, row, n):
11                    if row == n:
12                        print_board(board)
13                        return
14                    for col in range(n):
15                        if is_safe(board, row, col, n):
16                            board[row][col] = 'Q'
17    n = 4
18    board = [['.' for _ in range(n)] for _ in range(n)]
19    solve(board, 0, n)
```

### OUTPUT:



```
Output
```

```
- Q . .
- . . Q
Q . . .
- . Q .

. . Q .
Q . . .
- . . Q
- Q . .

==== Code Execution Successful ====
```

## RESULT:

Thus the program implemented successfully.

## EXPERIMENT-2

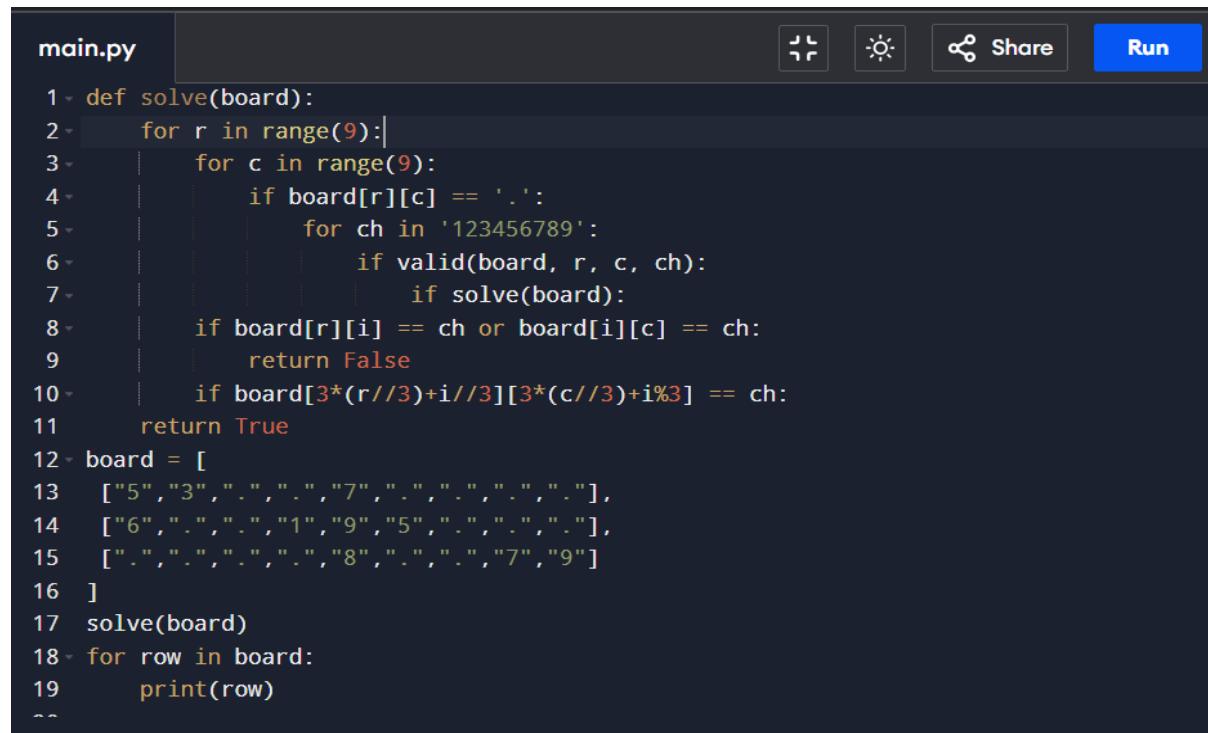
### AIM:

To solve a  $9 \times 9$  Sudoku puzzle using backtracking.

### PROCEDURE:

- Find an empty cell.
- Try filling it with numbers 1–9.
- Check if the number is valid (row, column, and box).
- If valid, continue; if not, backtrack.

### PROGRAM:



The screenshot shows a code editor window with a dark theme. The file is named "main.py". The code implements a backtracking algorithm to solve a 9x9 Sudoku board. The board is represented as a 2D list of strings. The solve function iterates through each cell, trying to place a digit from 1 to 9. It checks for validity by looking at the row, column, and 3x3 box. If a digit is valid, it places it and recursively calls itself for the next cell. If it reaches a point where no digit is valid, it backtracks by returning False. If all cells are filled correctly, it returns True. Finally, the solve function is called with an initial board configuration, and the resulting solved board is printed.

```
main.py
1 def solve(board):
2     for r in range(9):
3         for c in range(9):
4             if board[r][c] == '.':
5                 for ch in '123456789':
6                     if valid(board, r, c, ch):
7                         if solve(board):
8                             if board[r][c] == ch or board[c][r] == ch:
9                                 return False
10                        if board[3*(r//3)+i//3][3*(c//3)+i%3] == ch:
11                            return True
12    return False
13
14 board = [
15     ["5","3",".",".","7",".",".",".","."],
16     ["6",".",".","1","9","5",".",".","."],
17     [".",".",".","8",".",".","7","9"]
18 ]
19 solve(board)
20 for row in board:
21     print(row)
```

## OUTPUT:

```
Output
['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']

==== Code Execution Successful ====
```

## RESULT:

Thus the program implemented successfully.

## EXPERIMENT-3

### AIM:

To find how many ways '+' or '-' can be added to numbers so the total equals a target.

### PROCEDURE:

- For each number, choose either '+' or '-'.
- Calculate total recursively.
- If the total equals the target, count it.

### PROGRAM:

```
main.py
1 def findWays(nums, target):
2     count = 0
3     def backtrack(i, total):
4         nonlocal count
5         if i == len(nums):
6             if total == target:
7                 count += 1
8             return
9         backtrack(i + 1, total + nums[i])
10        backtrack(i + 1, total - nums[i])
11    backtrack(0, 0)
12    return count
13
14 nums = [1, 1, 1, 1, 1]
15 target = 3
16 print("Number of ways:", findWays(nums, target))
17
18
```

## OUTPUT:

```
Output
Number of ways: 5
==== Code Execution Successful ====
```

## RESULT:

Thus the program implemented successfully.

## EXPERIMENT-4

### AIM:

To find the sum of the minimum value of all possible contiguous subarrays of a given array using a monotonic stack for efficiency.

### PROCEDURE:

- For each element arr[i], find:
- left[i]: number of subarrays ending at i where arr[i] is the minimum.
- right[i]: number of subarrays starting at i where arr[i] is the minimum.
- Multiply arr[i] \* left[i] \* right[i] to find its total contribution.
- Sum all contributions and take modulo ( $10^9 + 7$ ).

### PROGRAM:

```
main.py
1 - def sumSubarrayMins(arr):
2     MOD = 10**9 + 7
3     n = len(arr)
4     stack = []
5     left = [0] * n
6     right = [0] * n
7     for i in range(n):
8         count = 1
9         while stack and stack[-1][0] > arr[i]:
10            count += stack.pop()[1]
11            left[i] = count
12         while stack and stack[-1][0] >= arr[i]:
13            count += stack.pop()[1]
14            right[i] = count
15         stack.append((arr[i], count))
16     for i in range(n):
17         ans = (ans + arr[i] * left[i] * right[i]) % MOD
18 arr = [3, 1, 2, 4]
19 print("Sum of Subarray Minimums:", sumSubarrayMins(arr))
```

## OUTPUT:

```
Output
Sum of Subarray Minimums: 17
==== Code Execution Successful ====
```

## RESULT:

Thus the program implemented successfully.

## EXPERIMENT-5

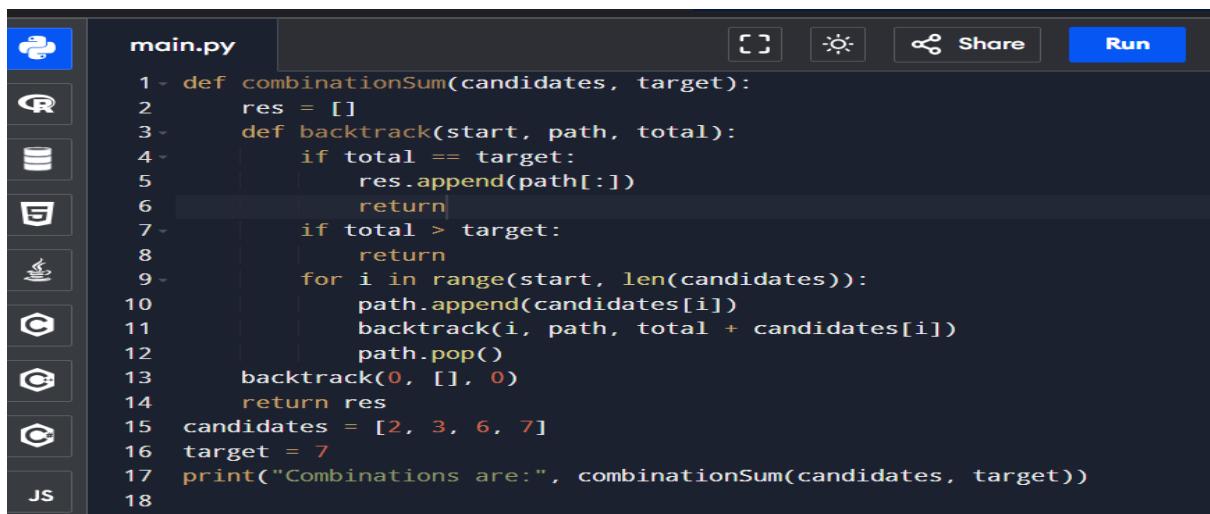
### AIM:

To find all unique combinations of numbers from a list that sum to a given target. Each number may be used multiple times.

### PROCEDURE:

- Sort the candidate list (optional).
- Use a recursive function backtrack(start, path, total):
- If total == target → store the path.
- If total > target → stop exploring.
- Return all valid combinations.

### PROGRAM:



The screenshot shows a code editor interface with a dark theme. On the left is a sidebar with icons for file operations like Open, Save, and Run. The main area has a tab labeled "main.py". The code is as follows:

```
main.py
1 - def combinationSum(candidates, target):
2 -     res = []
3 -     def backtrack(start, path, total):
4 -         if total == target:
5 -             res.append(path[:])
6 -             return
7 -         if total > target:
8 -             return
9 -         for i in range(start, len(candidates)):
10 -             path.append(candidates[i])
11 -             backtrack(i, path, total + candidates[i])
12 -             path.pop()
13 -     backtrack(0, [], 0)
14 -     return res
15 - candidates = [2, 3, 6, 7]
16 - target = 7
17 - print("Combinations are:", combinationSum(candidates, target))
18 -
```

## **OUTPUT:**

```
Output Clear
Combinations are: [[2, 2, 3], [7]]
==== Code Execution Successful ====

```

## **RESULT:**

Thus the program implemented successfully.