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

```python
def print_board(board):
    for row in board:
        print(" ".join(row))
    print()
def is_safe(board, row, col, n):
    for i in range(row):
        if board[i][col] == 'Q':
        if col - (row - i) >= 0 and board[i][col - (row - i)] == 'Q':
        if col + (row - i) < n and board[i][col + (row - i)] == 'Q':
def solve(board, row, n):
    if row == n:
        print_board(board)
        return
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 'Q'
n = 4
board = [['.' for _ in range(n)] for _ in range(n)]
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 9×9 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:

main.py

Share Run

```
def solve(board):
    for r in range(9):
        for c in range(9):
            if board[r][c] == '.':
                for ch in '123456789':
                    if valid(board, r, c, ch):
                        if solve(board):
        if board[r][i] == ch or board[i][c] == ch:
            return False
        if board[3*(r//3)+i//3][3*(c//3)+i%3] == ch:
    return True
board = [
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".",".",".",".","8",".",".","7","9"]
]
solve(board)
for row in board:
    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.

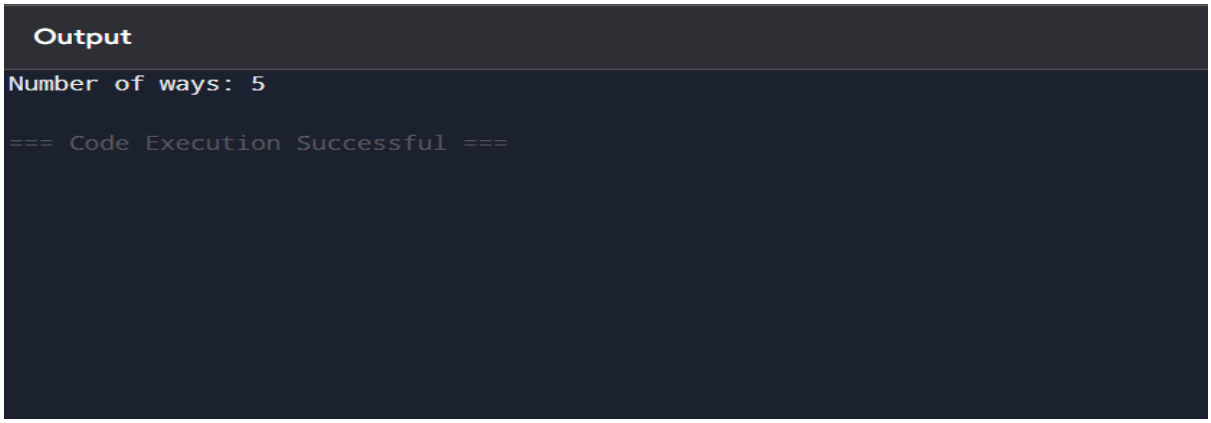
PROCEDURE:

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

PROGRAM:

```
main.py                                                    Share   Run
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:



RESULT:

Thus the program implemented successfully.

EXPERIMENT-4

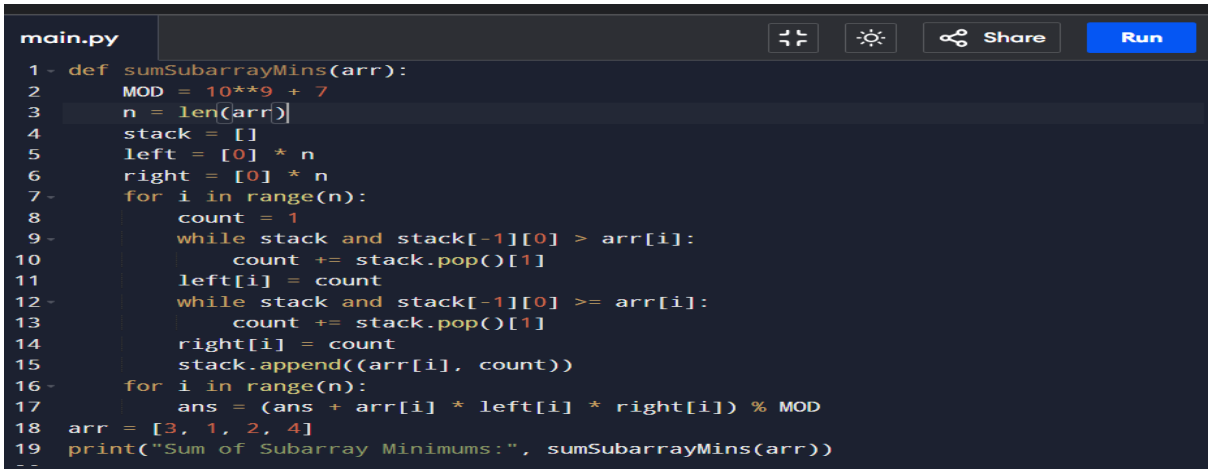
AIM:

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

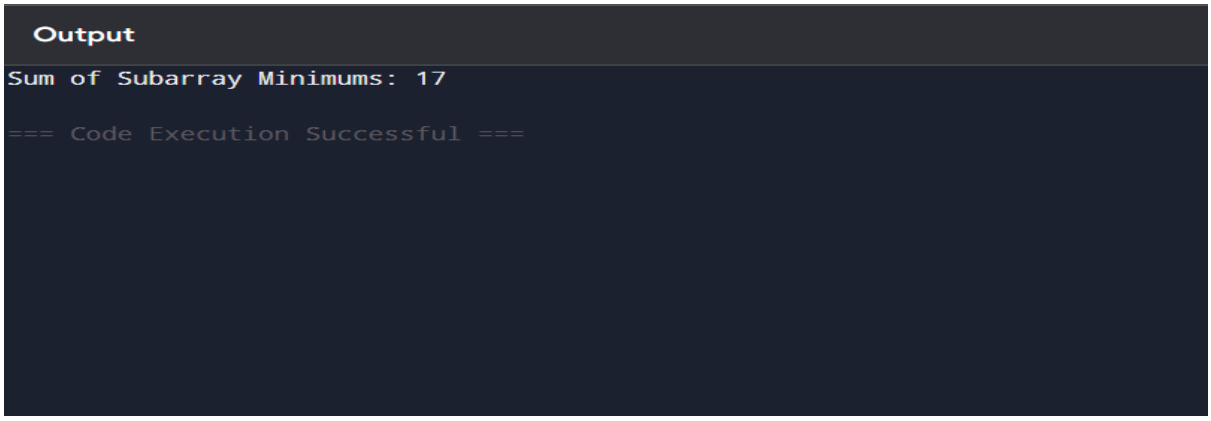
PROCEDURE:

- 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:



OUTPUT:



RESULT:

Thus the program implemented successfully.

EXPERIMENT-5

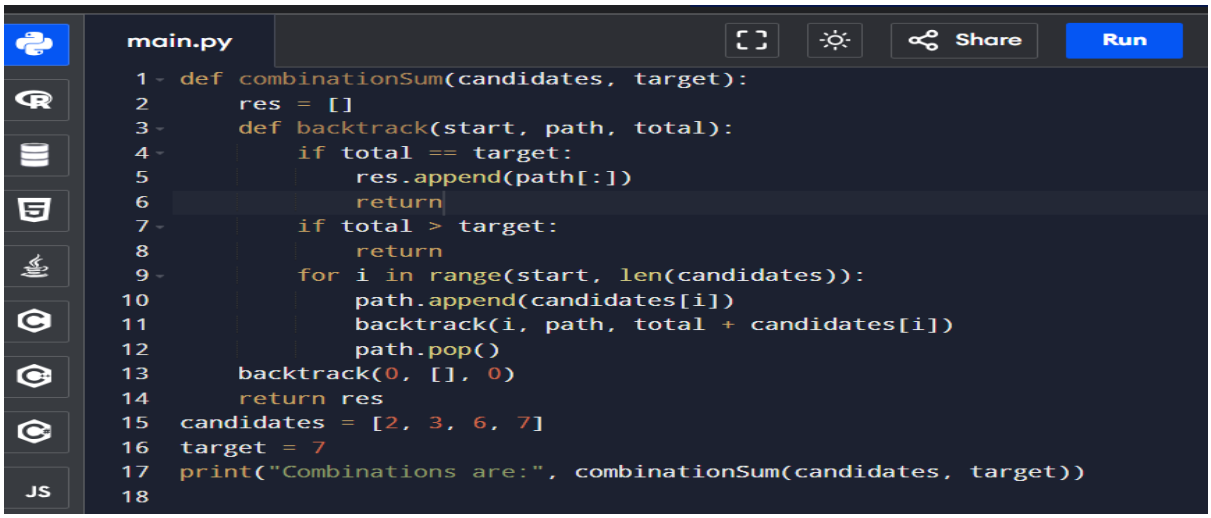
AIM:

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.

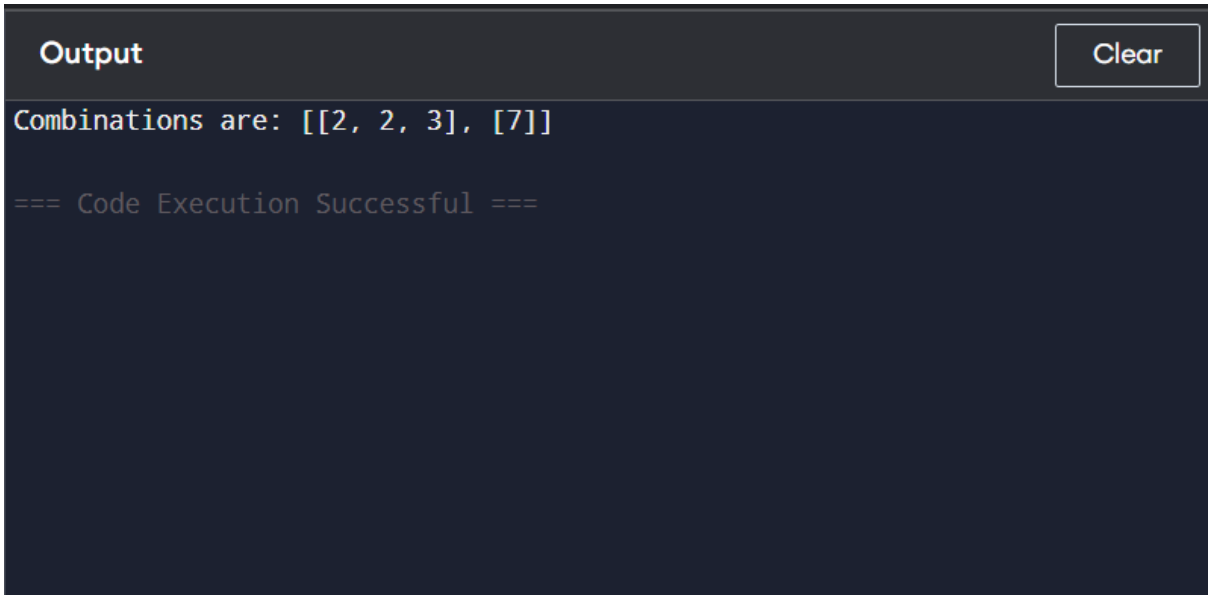
PROGRAM:




```
def combinationSum(candidates, target):
    res = []
    def backtrack(start, path, total):
        if total == target:
            res.append(path[:])
            return
        if total > target:
            return
        for i in range(start, len(candidates)):
            path.append(candidates[i])
            backtrack(i, path, total + candidates[i])
            path.pop()
    backtrack(0, [], 0)
    return res
candidates = [2, 3, 6, 7]
target = 7
print("Combinations are:", combinationSum(candidates, target))

```

OUTPUT:



RESULT:

Thus the program implemented successfully.