DAY 10

1.

```
# Function to print the board
def print_board(board):
    for row in board:
        print(" ".join(row))

# Backtracking function to solve the N-Queens problem
def solve_n_queens(N):
    board = [["." for _ in range(N)] for _ in range(N)]  # Create an empty board
    solutions = []
    solve(board, 0, N, solutions)
    return solutions

# Helper function for backtracking
def solve(board, row, N, solutions):
    if row == N:
        # If all queens are placed, add the solution
        solutions.append(["".join(row) for row in board])
        return

    for col in range(N):
        if is_safe(board, row, col, N):
            # Place the queen
            board[row][col] = "Q"
            solve(board, row + 1, N, solutions)  # Recur to place the next queen
            board[row][col] = "."  # Backtrack

# Function to check if it's safe to place a queen at board[row][col]
def is_safe(board, row, col, N):
    # Check the column
    for i in range(row):
        if board[i][col] == "Q":
            return False

    # Check upper left diagonal
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i][j] == "Q":
            return False
```

```
    # Check upper right diagonal
    for i, j in zip(range(row-1, -1, -1), range(col+1, N)):
        if board[i][j] == "Q":
            return False

    return True

# Visualizing the solutions for N = 4, N = 5, N = 8
def visualize_solutions(N):
    solutions = solve_n_queens(N)
    print(f"Visualizing solutions for N = {N}:\n")
    for idx, solution in enumerate(solutions, 1):
        print(f"Solution {idx}:")
        for row in solution:
            print(row)
        print()

# Test the function with N = 4, N = 5, N = 8
visualize_solutions(4)
visualize_solutions(5)
visualize_solutions(8)
```

2.

```
# Function to print the board
def print_board(board):
    for row in board:
        print(" ".join(row))

# Backtracking function to solve the N-Queens problem
def solve_n_queens(N, obstacles=None, restricted_columns=None):
    board = [["." for _ in range(N)] for _ in range(N)]  # Create an empty board
    solutions = []
    solve(board, 0, N, solutions, obstacles, restricted_columns)
    return solutions

# Helper function for backtracking
def solve(board, row, N, solutions, obstacles, restricted_columns):
    if row == N:
        # If all queens are placed, add the solution
```

```
        solutions.append([row.index("Q") + 1 for row in board])  # Store 1-based column index
        return

    for col in range(N):
        # Check for obstacles and restricted positions
        if obstacles and (row, col) in obstacles:
            continue
        if restricted_columns and row == 0 and col not in restricted_columns:
            continue
        if is_safe(board, row, col, N):
            # Place the queen
            board[row][col] = "Q"
            solve(board, row + 1, N, solutions, obstacles, restricted_columns)  # Recur to place the
next queen
            board[row][col] = "."  # Backtrack

# Function to check if it's safe to place a queen at board[row][col]
def is_safe(board, row, col, N):
    # Check the column
    for i in range(row):
        if board[i][col] == "Q":
            return False

    # Check upper left diagonal
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i][j] == "Q":
            return False

    # Check upper right diagonal
    for i, j in zip(range(row-1, -1, -1), range(col+1, N)):
        if board[i][j] == "Q":
            return False

    return True

# Visualizing the solutions for N = 5 with obstacles, and 6×6 with restricted positions
def visualize_solutions(N, obstacles=None, restricted_columns=None):
    solutions = solve_n_queens(N, obstacles, restricted_columns)
    print(f"Visualizing solutions for N = {N}:")
    for idx, solution in enumerate(solutions, 1):
        print(f"Solution {idx}: {solution}")
    print()

# Test the function with different cases
```

```
visualize_solutions(8, obstacles=None)  # Example for 8×10 (8 rows, 10 columns)
visualize_solutions(5, obstacles=[(2, 2), (4, 4)])  # Example for 5×5 with obstacles
visualize_solutions(6, restricted_columns=[2, 4])  # Example for 6×6 with restricted columns
```

3.

```
def solve_sudoku(board):
    def is_valid(board, row, col, num):
        # Check if num is in the current row
        for i in range(9):
            if board[row][i] == num:
                return False

        # Check if num is in the current column
        for i in range(9):
            if board[i][col] == num:
                return False

        # Check if num is in the 3x3 sub-box
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
                if board[start_row + i][start_col + j] == num:
                    return False

        return True

    def backtrack(board):
        # Try to find an empty space ('.')
        for row in range(9):
            for col in range(9):
                if board[row][col] == '.':
                    # Try placing each number from '1' to '9'
                    for num in '123456789':
                        if is_valid(board, row, col, num):
                            board[row][col] = num  # Place the number

                            if backtrack(board):  # Recur to place the next numbers
                                return True

                            board[row][col] = '.'  # Backtrack, reset the cell
```

```
            return False  # If no valid number is found, return False
        return True  # If the entire board is filled correctly, return True

    # Start the backtracking process
    backtrack(board)

# Test the function with the provided example
board = [
    ["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"]
]

solve_sudoku(board)

# Output the solved board
for row in board:
    print(row)
```

4.

```
def solve_sudoku(board):
    def is_valid(board, row, col, num):
        # Check if 'num' is in the current row
        for i in range(9):
            if board[row][i] == num:
                return False

        # Check if 'num' is in the current column
        for i in range(9):
            if board[i][col] == num:
                return False
```

```
        # Check if 'num' is in the 3x3 sub-box
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
                if board[start_row + i][start_col + j] == num:
                    return False

        return True

    def backtrack(board):
        # Try to find an empty space ('.')
        for row in range(9):
            for col in range(9):
                if board[row][col] == '.':
                    # Try placing each number from '1' to '9'
                    for num in '123456789':
                        if is_valid(board, row, col, num):
                            board[row][col] = num  # Place the number

                            if backtrack(board):  # Recur to place the next numbers
                                return True

                            board[row][col] = '.'  # Backtrack, reset the cell

                    return False  # If no valid number is found, return False
        return True  # If the entire board is filled correctly, return True

    # Start the backtracking process
    backtrack(board)

# Test the function with the provided example
board = [
    ["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"]
]

solve_sudoku(board)
```

```
# Output the solved board
for row in board:
    print(row)
```

5.

```
def find_ways_to_target(nums, target):
    total_sum = sum(nums)

    # If the total_sum and target have different parity or the target is larger than total_sum, return 0
    if (total_sum + target) % 2 != 0 or abs(target) > total_sum:
        return 0

    # Define the subset sum we're looking for
    sum_pos = (total_sum + target) // 2

    # Dynamic programming array to store the number of ways to reach each sum
    dp = [0] * (sum_pos + 1)
    dp[0] = 1  # There is one way to get a sum of 0, which is to select no elements

    # Iterate through each number in the input array
    for num in nums:
        # Traverse the dp array from the back to avoid overwriting the results of the current iteration
        for s in range(sum_pos, num - 1, -1):
            dp[s] += dp[s - num]

    return dp[sum_pos]

# Example 1:
nums1 = [1, 1, 1, 1, 1]
target1 = 3
print(find_ways_to_target(nums1, target1))  # Output: 5

# Example 2:
nums2 = [1]
target2 = 1
print(find_ways_to_target(nums2, target2))  # Output: 1
```

6.

```python
MOD = 10**9 + 7

def sumSubarrayMins(arr):
    n = len(arr)

    # Step 1: Initialize stacks for next smaller and previous smaller elements
    prev_smaller = [-1] * n
    next_smaller = [n] * n

    stack = []

    # Step 2: Calculate previous smaller for each element
    for i in range(n):
        while stack and arr[stack[-1]] >= arr[i]:
            stack.pop()
        if stack:
            prev_smaller[i] = stack[-1]
        stack.append(i)

    # Step 3: Calculate next smaller for each element
    stack.clear()
    for i in range(n - 1, -1, -1):
        while stack and arr[stack[-1]] > arr[i]:
            stack.pop()
        if stack:
            next_smaller[i] = stack[-1]
        stack.append(i)

    # Step 4: Calculate the sum of minimums of all subarrays
    total_sum = 0
    for i in range(n):
        left_count = i - prev_smaller[i]
        right_count = next_smaller[i] - i
        total_sum += arr[i] * left_count * right_count
        total_sum %= MOD

    return total_sum

# Example 1:
arr1 = [3, 1, 2, 4]
print(sumSubarrayMins(arr1))  # Output: 17
```

```python
# Example 2:
arr2 = [11, 81, 94, 43, 3]
print(sumSubarrayMins(arr2))  # Output: 444
```

7.

```python
def combinationSum(candidates, target):
    result = []

    def backtrack(start, target, current_combination):
        # If target is 0, we've found a valid combination
        if target == 0:
            result.append(list(current_combination))
            return

        # Try every candidate starting from 'start' index
        for i in range(start, len(candidates)):
            candidate = candidates[i]

            # If candidate exceeds the target, stop exploring further
            if candidate > target:
                continue

            # Add the current candidate to the combination and recurse
            current_combination.append(candidate)

            # Since we can reuse the same candidate, we pass the same index `i`
            backtrack(i, target - candidate, current_combination)

            # Backtrack, remove the last added element
            current_combination.pop()

    # Sort candidates to facilitate early termination in the backtracking process
    candidates.sort()

    # Start backtracking from index 0
    backtrack(0, target, [])

    return result
```

8.

```
def combinationSum2(candidates, target):
    result = []
    candidates.sort()  # Sort the candidates to handle duplicates more easily

    def backtrack(start, target, current_combination):
        # If the remaining target is 0, we've found a valid combination
        if target == 0:
            result.append(list(current_combination))
            return
        # Iterate through the candidates starting from 'start' index
        for i in range(start, len(candidates)):
            # Skip duplicates by checking if the current number is the same as the previous
            if i > start and candidates[i] == candidates[i - 1]:
                continue
            # If the current candidate exceeds the target, no point in exploring further
            if candidates[i] > target:
                break
            # Include the current candidate and recursively find combinations for the remaining
target
            current_combination.append(candidates[i])
            backtrack(i + 1, target - candidates[i], current_combination)  # Move forward (no repeat)
            # Backtrack by removing the last added candidate
            current_combination.pop()

    # Start the backtracking from index 0
    backtrack(0, target, [])
    return result
```

9.

```
def permute(nums):
    result = []

    def backtrack(path, remaining):
        # If the remaining list is empty, we have a complete permutation
        if not remaining:
            result.append(path)
            return
```

```
        for i in range(len(remaining)):
            # Explore the next number in the remaining list
            backtrack(path + [remaining[i]], remaining[:i] + remaining[i+1:])

    backtrack([], nums)
    return result
```

10.

```
def permuteUnique(nums):
    result = []

    # Sort the numbers to ensure duplicates are adjacent
    nums.sort()

    def backtrack(path, remaining):
        # If remaining is empty, we have a valid permutation
        if not remaining:
            result.append(path)
            return

        for i in range(len(remaining)):
            # Skip duplicates
            if i > 0 and remaining[i] == remaining[i - 1]:
                continue

            # Include the number at index i and recurse
            backtrack(path + [remaining[i]], remaining[:i] + remaining[i + 1:])

    backtrack([], nums)
    return result
```