Program 1:

```
import matplotlib.pyplot as plt
import time
import random


class TVChannel:
    def __init__(self, name, viewers, viewing_time, rating):
        self.name = name
        self.viewers = viewers
        self.viewing_time = viewing_time
        self.rating = rating


def create_channel(i):
    return TVChannel(f"Channel {i}", random.randint(500, 2000000),
random.randint(30, 180), rate_channel())


def rate_channel():
    viewers = random.randint(500, 2000000)
    if viewers >= 1500000:
        return 1
    elif viewers >= 1000000:
        return 2
    elif viewers >= 500000:
        return 3
    elif viewers >= 300000:
        return 4
    elif viewers >= 100000:
        return 5
    else:
        return 6


def quick_sort(arr):
    if len(arr) <= 1: return arr
    pivot = arr[0]
    smaller = [x for x in arr[1:] if x <= pivot]
    greater = [x for x in arr[1:] if x > pivot]
    return quick_sort(smaller) + [pivot] + quick_sort(greater)
```

```python
def plot_sorting_algorithms():
    sorting_algorithms = ["Quick Sort", "Merge Sort"]
    running_times = []
    data = [random.randint(1, 200000) for _ in range(1, 10000)]

    for algorithm in sorting_algorithms:
        start_time = time.time()
        if algorithm == "Quick Sort":
            quick_sort(data)
        elif algorithm == "Merge Sort":
            merge_sort(data)
        end_time = time.time()
        running_times.append(end_time - start_time)

    plt.bar(sorting_algorithms, running_times)
    plt.xlabel("Sorting Algorithms")
    plt.ylabel("Running Time (seconds)")
    plt.title("Running Time of Sorting Algorithms")
    plt.show()


def merge_sort(nlist):
    if len(nlist) <= 1: return nlist
    mid = len(nlist) // 2
    lefthalf = merge_sort(nlist[:mid])
    righthalf = merge_sort(nlist[mid:])

    i = j = k = 0
    while i < len(lefthalf) and j < len(righthalf):
        if lefthalf[i] < righthalf[j]:
            nlist[k] = lefthalf[i]
            i += 1
        else:
            nlist[k] = righthalf[j]
            j += 1
        k += 1

    nlist[k:] = lefthalf[i:] + righthalf[j:]
    return nlist


# Main function
```

```
channels = [create_channel(i + 1) for i in range(10)]
print("Channels created successfully!")

for channel in channels:
    print(f"{channel.name}:\n {channel.viewers} viewers,
{channel.viewing_time} hours, Rank: {channel.rating}")

plot_sorting_algorithms()
```

Program 2:

```
def find_optimal_schedule(jobs):
    total_time = total_profit = 0
    schedule = []
    for duration, profit in sorted(jobs, key=lambda x: x[1] / x[0],
reverse=True):
        if total_time + duration <= profit:
            schedule.append((duration, profit))
            total_time += duration
            total_profit += profit
    return schedule

def display_schedule(schedule):
    total_time = sum(duration for duration, _ in schedule)
    total_profit = sum(profit for _, profit in schedule)
    print("Optimal Schedule:")
    for i, (duration, profit) in enumerate(schedule):
        print(f"Job {i + 1}: Duration={duration}, Profit={profit}")
    print(f"\nTotal Time: {total_time}\nTotal Profit:
{total_profit}")

# Example usage
jobs = [(3, 5), (4, 6), (2, 2), (1, 8), (5, 10)]  # Format:
(duration, profit)
display_schedule(find_optimal_schedule(jobs))
```

Program 3:

```
def find_route(streets, money):
    n = len(streets)
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        if streets[i - 1] in money:
            dp[i] = max(dp[i], dp[i - 1] + money[streets[i - 1]])
            dp[i] = max(dp[i], dp[i - 1])
        route = []
    i = n
    while i > 0:
        if dp[i] != dp[i - 1]:
            route.append(streets[i - 1])
            i -= 1
            route.reverse()
    return route

streets = ['a', 'b', 'c', 'd', 'e']
money = {'a': 1, 'b': 2, 'c': 3, 'd': 5, 'e': 4}
perfect_route = find_route(streets, money)
print("The perfect path is: ", perfect_route)
```

Program 4:

```
import sys

visited = [0 for i in range(0, 10)]
n = int(input("Enter the number of nodes:"))
print("Enter the adjacency matrix:\n")
cost = []
for i in range(0, n):
    item = []
    for j in range(0, n):
        value = int(input())
        item.append(value)
    cost.append(item)

for item in cost:
    for index, value in enumerate(item):
        if value == 0:
            item[index] = sys.maxsize

visited[0] = 1
ne = 1
a = b = u = v = 0
mincost = 0

while ne < n:
    minimum = sys.maxsize
    for i in range(0, n):
        if visited[i] != 0:
            for j in range(0, n):
                if visited[j] == 0 and cost[i][j] < minimum:
                    minimum = cost[i][j]
                    a = u = i
                    b = v = j
    if visited[u] == 0 or visited[v] == 0:
        print(f"\nEdge {ne}:({a} {b}) cost: {minimum}")
        ne += 1
        mincost += minimum
        visited[b] = 1
    cost[a][b] = cost[b][a] = sys.maxsize

print(f"\nMinimum cost = {mincost}")
```

Program 5:

```
import heapq
from collections import Counter

class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right
        self.huff = ''

    def __lt__(self, other):
        return self.freq < other.freq

def printNodes(node, val=''):
    if not node:
        return
    newVal = val + str(node.huff)
    if not node.left and not node.right:
        print(f"{node.symbol} -> {newVal}")
    printNodes(node.left, newVal)
    printNodes(node.right, newVal)

string = input("Enter the string: ").lower()
res = Counter(string)
nodes = [Node(freq, char) for char, freq in res.items()]
heapq.heapify(nodes)

while len(nodes) > 1:
    left = heapq.heappop(nodes)
    right = heapq.heappop(nodes)
    left.huff = '0'
    right.huff = '1'
    new_node = Node(left.freq + right.freq, left.symbol +
right.symbol, left, right)
    heapq.heappush(nodes, new_node)

printNodes(nodes[0])
```

Program 6:

```
def knapSack(W, wt, val, n):
    if n == 0 or W == 0:
        return 0
    if wt[n - 1] > W:
        return knapSack(W, wt, val, n - 1)
    return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),
knapSack(W, wt, val, n - 1))

profit = [60, 100, 120]
weight = [10, 20, 30]
n = len(profit)
W = 10
print(knapSack(W, weight, profit, n))
```

Program 7:

```
import itertools


def calculate_distance(x1, y1, x2, y2):
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5


def calculate_total_distance(points_order, distances):
    total_distance = sum(distances[point1][point2] for point1,
point2 in zip(points_order, points_order[1:]))
    return total_distance


def find_optimal_drilling_time(points, toolbox_time):
    num_points = len(points)
    distances = [[calculate_distance(x1, y1, x2, y2) for x2, y2, _
in points] for x1, y1, _ in points]

    optimal_drilling_time = float('inf')

    for permutation in itertools.permutations(range(num_points)):
        total_distance = calculate_total_distance(permutation,
distances)
        drilling_time = total_distance + (num_points - 1) *
toolbox_time
        optimal_drilling_time = min(optimal_drilling_time,
drilling_time)

    return optimal_drilling_time


def main():
    points = [(0, 0, 1), (3, 0, 2), (0, 4, 1), (3, 4, 2)]
    toolbox_time = 5
    optimal_time = find_optimal_drilling_time(points, toolbox_time)
    print("Optimal Drilling Time:", optimal_time)


if __name__ == "__main__":
    main()
```

Program 8:

```
from itertools import combinations


def calculate_min_time_slots(subjects):
    students = set(student for subject in subjects for student in
subject)
    graph = {student: set() for student in students}

    for subjects_list in subjects:
        for student1, student2 in combinations(subjects_list, 2):
            graph[student1].add(student2)
            graph[student2].add(student1)

    def graph_coloring():
        color = {}
        for student in graph:
            neighbors_color = set(color.get(neighbor, None) for
neighbor in graph[student])
            for c in range(1, len(graph) + 1):
                if c not in neighbors_color:
                    color[student] = c
                    break
        return max(color.values())

    min_time_slots = graph_coloring()
    return min_time_slots


def main():
    subjects = [
        ["Alice", "Bob", "Charlie"],
        ["Bob", "David"],
        ["Charlie", "Eve", "Frank"],
        ["David", "Frank"]
    ]
    min_time_slots = calculate_min_time_slots(subjects)
    print("Minimum Time Slots:", min_time_slots)


if __name__ == "__main__":
    main()
```

Program 9:

```
def isSafe(mat, r, c):
    for i in range(r):
        if mat[i][c] == 'Q':
            return False
        if c - r + i >= 0 and mat[i][c - r + i] == 'Q':
            return False
        if c + r - i < len(mat) and mat[i][c + r - i] == 'Q':
            return False
    return True


def nQueen(mat, r):
    if r == len(mat):
        printSolution(mat)
        return

    for i in range(len(mat)):
        if isSafe(mat, r, i):
            mat[r][i] = 'Q'
            nQueen(mat, r + 1)
            mat[r][i] = '-'


def printSolution(mat):
    for r in mat:
        print(''.join(r))
    print()


if __name__ == '__main__':
    print("Enter the number of queens")
    N = int(input())
    mat = [['-' for _ in range(N)] for _ in range(N)]
    nQueen(mat, 0)
```