Assignment 1:

```
def karatsuba(x, y):
    if x < 10 or y < 10:
        return x * y

    # size of no
    n = max(len(str(x)), len(str(y)))
    half = n // 2

    high_x = x // 10**half
    low_x = x % 10**half
    high_y = y // 10**half
    low_y = y % 10**half

    z0 = karatsuba(low_x, low_y)
    z1 = karatsuba((low_x + high_x), (low_y + high_y))
    z2 = karatsuba(high_x, high_y)

    # Combine the results
    return (z2 * 10**(2 * half)) + ((z1 - z2 - z0) * 10**half) + z0

def square_large_number(n):
    return karatsuba(n, n)
#defining the number
large_number = int(input("Enter a 20 digit number to square: "))
result = square_large_number(large_number)
print(f"Square of {large_number} is: {result}")
```

Assignment 2:

```
def job_scheduling(tasks, n): #task list of n elemnts. ie task[deadline, profit]

    tasks.sort(key=lambda x: x[1], reverse=True)  # Sorting by profit i.e second tuple of list


    max_deadline = max(task[0] for task in tasks) #iterating first tuple for finding max deadline


    result = [-1] * max_deadline
    total_profit = 0

    for task in tasks:
        deadline = task[0]
        profit = task[1]

        # Find a free slot for this job (starting from the latest possible slot)
        for slot in range(min(deadline - 1, max_deadline - 1), -1, -1):
            if result[slot] == -1:
                result[slot] = task
                total_profit += profit
                break

    return result, total_profit
n = int(input("Enter the number of tasks: "))
tasks = []

for i in range(n):
    deadline = int(input(f"Enter deadline for task {i+1}: "))
    profit = int(input(f"Enter profit for task {i+1}: "))
    tasks.append((deadline, profit))

# Schedule the tasks and calculate the total profit
schedule, profit = job_scheduling(tasks, n)

# Display the results
print("Scheduled tasks:", schedule)
print("Total profit:", profit)
```

Assignment 3:

```
INF = float(&#39;inf&#39;)
def floyd_warshall(dist, n):
    #  Floyd-Warshall algorithm
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    # Printing the result matrix
    print(&quot;\nThe following matrix shows the minimum costs between every pair of
cities:&quot;)
    for i in range(n):
        for j in range(n):
            if dist[i][j] == INF:
                print(&quot;INF&quot;, end=&quot;\t&quot;)
            else:
                print(dist[i][j], end=&quot;\t&quot;)
        print()

def main():
    n = int(input(&quot;Enter the number of cities: &quot;))
    dist = []
    print(&quot;\nEnter the cost matrix (use &#39;INF&#39; for no direct connection):&quot;)
    print(&quot;\nEnter in the A0 matrix form&quot;)
    for i in range(n):
        row = input().split()

        row = [INF if x == &#39;INF&#39; else int(x) for x in row]
        dist.append(row)
    floyd_warshall(dist, n)
if __name__ == &quot;__main__&quot;:
main()
```

Assignment 4:

```python
import heapq
from collections import defaultdict

def networkDelayTime(times, N, K):
    graph = defaultdict(list) #adjecncy list for graph
    for u, v, w in times:
        graph[u].append((v, w))

    min_heap = [(0, K)]

    shortest_time = {}

    # Dijkstra's algorithm
    while min_heap:
        time, node = heapq.heappop(min_heap)

        # If node is already visited, skip
        if node in shortest_time:
            continue

        shortest_time[node] = time

        # Traverse neighbors
        for neighbor, t in graph[node]:
            if neighbor not in shortest_time:
                heapq.heappush(min_heap, (time + t, neighbor))

    # Check if all nodes are reached
    if len(shortest_time) == N:
        return max(shortest_time.values())
    else:
        return -1

# user input
N = int(input("Enter the number of nodes (N): "))
M = int(input("Enter the number of edges (M): "))

times = []
print("Enter the edges in the format: 'u v w' where u -> v with weight w")
for i in range(M):
    u, v, w = map(int, input(f"Edge {i+1}: ").split())
    times.append([u, v, w])
```

```
K = int(input("Enter the starting node (K): "))

result = networkDelayTime(times, N, K)
print(f"The time it takes for the signal to reach all nodes is: {result}")
```

Assignment 5:

```
def is_valid(x, y, board, N):
    return 0 <= x < N and 0 <= y < N and board[x][y] == -1

def print_solution(board, N):
    for row in board:
        for cell in row:
            print(f"{cell:2}", end=" ")
        print()

def solve_knights_tour(N, start_x, start_y):
    board = [[-1 for _ in range(N)] for _ in range(N)]
    move_x = [2, 1, -1, -2, -2, -1, 1, 2]
    move_y = [1, 2, 2, 1, -1, -2, -2, -1]

    board[start_x][start_y] = 0

    if not solve_knights_tour_util(start_x, start_y, 1, board, move_x,
move_y, N):
        print("Solution does not exist!")
    else:
        print_solution(board, N)


def solve_knights_tour_util(x, y, move_i, board, move_x, move_y, N):

    if move_i == N * N:
        return True

    for k in range(8):
        next_x = x + move_x[k]
        next_y = y + move_y[k]
```

```
        if is_valid(next_x, next_y, board, N):
            board[next_x][next_y] = move_i


            if solve_knights_tour_util(next_x, next_y, move_i + 1, board,
move_x, move_y, N):
                return True

            board[next_x][next_y] = -1

    return False

if __name__ == "__main__":

    N = int(input("Enter the size of the chessboard (N x N): "))

    start_x = int(input(f"Enter the starting x-coordinate (0 to {N-1}):
"))
    start_y = int(input(f"Enter the starting y-coordinate (0 to {N-1}):
"))

    if 0 <= start_x < N and 0 <= start_y < N:
        solve_knights_tour(N, start_x, start_y)
    else:
        print("Invalid starting position!")
```

Assignment 6:

```
def get_cost_matrix():
    N = int(input("Enter the number of students/clubs (N): "))
    print("Enter the cost matrix (NxN) row by row, with each value
separated by a space:")
    cost_matrix = []
    for i in range(N):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        cost_matrix.append(row)
    return cost_matrix

# Function to check if assigning the current student to the club is valid
def is_valid_assignment(student, club, current_assignment):

    return club not in current_assignment

def assign_clubs(student, current_cost, current_assignment, cost_matrix,
N):
    global min_cost, optimal_assignment

    if student == N:
        if current_cost < min_cost:
            min_cost = current_cost
            optimal_assignment = current_assignment.copy()
        return


    for club in range(N):

        if is_valid_assignment(student, club, current_assignment):
            new_cost = current_cost + cost_matrix[student][club]


            if new_cost < min_cost:

                current_assignment[student] = club

                assign_clubs(student + 1, new_cost, current_assignment,
cost_matrix, N)

                current_assignment[student] = -1
```

```
# Main function to start the club assignment
def solve_club_assignment():
    global min_cost, optimal_assignment
    cost_matrix = get_cost_matrix()
    N = len(cost_matrix)

    min_cost = float('inf')

    optimal_assignment = [-1] * N

    current_assignment = [-1] * N

    assign_clubs(0, 0, current_assignment, cost_matrix, N)

    # Output the result
    print("Minimum Assignment Cost:", min_cost)
    print("Optimal Assignment (Student to Club):", [x + 1 for x in
optimal_assignment])

solve_club_assignment()
```