Practical 1

```
def iterative_sum(n):
    sum = 0
    for i in range(1, n + 1):
        sum += i
    return sum
def recursive_sum(n):
    if n == 1:
        return 1
    return n + recursive_sum(n - 1)
def main():
    n = 10  # Sum of first 10 numbers = 55
    iterative_result = iterative_sum(n)
    recursive_result = recursive_sum(n)
    print("Iterative approach result:", iterative_result)
    print("Recursive approach result:", recursive_result)
if __name__ == "__main__":
    main()
```

practical 2

```
import heapq
class HuffmanNode:
    def __init__(self, freq, ch):
        self.freq = freq
        self.ch = ch
        self.left = None
        self.right = None
    # for priority queue comparison
    def __lt__(self, other):
        return self.freq < other.freq
def print_code(root, code):
    if root is None:
        return
    if root.left is None and root.right is None and root.ch.isalpha():
        print(f"{root.ch} -> {code}")
        return
    print_code(root.left, code + "0")
    print_code(root.right, code + "1")
def main():
    chars = input("Enter characters (without spaces): ")
    freq_input = input("Enter corresponding frequencies (separated by spaces): ")
    freq = list(map(int, freq_input.split()))
    q = []
    for i in range(len(chars)):
        node = HuffmanNode(freq[i], chars[i])
        heapq.heappush(q, node)
    while len(q) > 1:
        x = heapq.heappop(q)
        y = heapq.heappop(q)
        f = HuffmanNode(x.freq + y.freq, '-')
        f.left = x
        f.right = y
        heapq.heappush(q, f)


    root = q[0]
    print("Huffman Codes:")
    print_code(root, "")
if __name__ == "__main__":
    main()
```

Practical 3

```
def knapSack(W, wt, val, n):
    K = [[0 for _ in range(W + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i - 1] <= w:
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w])
            else:
                K[i][w] = K[i - 1][w]
    return K[n][W]  # Maximum profit
def main():
    val = list(map(int, input("Enter values of items (space-separated): ").split()))
    wt = list(map(int, input("Enter weights of items (space-separated): ").split()))
    W = int(input("Enter Knapsack Capacity: "))
    n = len(val)
    max_profit = knapSack(W, wt, val, n)

    print("Maximum profit that can be achieved:", max_profit)

if __name__ == "__main__":

    main()
```

Practical 4

```
def print_solution(board, N):

    for i in range(N):

        print("[", end="")

        for j in range(N):

            if board[i][j] == 1:

                print(" Q ", end="")

            else:

                print(" - ", end="")

        print("]")

    print()

def is_safe(board, row, col, N):

    # Check left side of the current row

    for i in range(col):

        if board[row][i] == 1:

            return False

    # Check upper-left diagonal

    i, j = row, col

    while i >= 0 and j >= 0:

        if board[i][j] == 1:

            return False

        i -= 1
        j -= 1

    # Check lower-left diagonal

    i, j = row, col

    while i < N and j >= 0:

        if board[i][j] == 1:

            return False

        i += 1

        j -= 1

    return True

def solve_nq_util(board, col, N):

    if col >= N:

        print_solution(board, N)

        return True

    res = False

    for i in range(N):

        if is_safe(board, i, col, N):

            board[i][col] = 1

            res = solve_nq_util(board, col + 1, N) or res

            board[i][col] = 0  # backtrack

    return res

def solve_nq(N):

    board = [[0 for _ in range(N)] for _ in range(N)]

    if not solve_nq_util(board, 0, N):

        print("Solution does not exist")

def main():

    N = int(input("Enter value of N (number of queens): "))
    solve_nq(N)
if __name__ == "__main__":
    main()
```

Practical 5

```
import random
import time
def deterministic_partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
def deterministic_quick_sort(arr, low, high):
    if low < high:
        pi = deterministic_partition(arr, low, high)
        deterministic_quick_sort(arr, low, pi - 1)
        deterministic_quick_sort(arr, pi + 1, high)



def randomized_partition(arr, low, high):
    random_index = random.randint(low, high)
    arr[random_index], arr[high] = arr[high], arr[random_index]
    return deterministic_partition(arr, low, high)
```

```
def randomized_quick_sort(arr, low, high):
    if low < high:
        pi = randomized_partition(arr, low, high)
        randomized_quick_sort(arr, low, pi - 1)
        randomized_quick_sort(arr, pi + 1, high)
def generate_random_array(size):
    return [random.randint(-10**9, 10**9) for _ in range(size)]
def main():
    sizes = [100, 1000, 10000, 100000]
    for size in sizes:
        arr = generate_random_array(size)
        arr_copy = arr[:]
        start_time = time.time()
        deterministic_quick_sort(arr, 0, len(arr) - 1)
        deterministic_time = time.time() - start_time
        start_time = time.time()
        randomized_quick_sort(arr_copy, 0, len(arr_copy) - 1)
        randomized_time = time.time() - start_time
        print(f"Array size: {size}")
        print(f"Deterministic Quick Sort time: {deterministic_time:.6f} seconds")
        print(f"Randomized Quick Sort time: {randomized_time:.6f} seconds\n")
if __name__ == "__main__":
    main()
```