Experiment 1: Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyse their time and space complexity.

With ite

```
def fibonacci_iterative(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b

# Example usage:
n = 10  # Change n to the desired Fibonacci number
result = fibonacci_iterative(n)
print(f"The {n}-th Fibonacci number is {result}")
```

Experiment 1

with recursive

```
def fibonacci_recursive(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

# Example usage:
n = 10  # Change n to the desired Fibonacci number
result = fibonacci_recursive(n)
print(f"The {n}-th Fibonacci number is {result}")
```

experiment 2

Write a program to solve a fractional Knapsack problem using a greedy method

```
def fractional_knapsack(items, capacity):
    items.sort(key=lambda x: x[1] / x[0], reverse=True)
    total_value = 0.0
    remaining_capacity = capacity

    for item in items:
        weight, value = item
        if weight <= remaining_capacity:
            total_value += value
            remaining_capacity -= weight
        else:
            fraction = remaining_capacity / weight
            total_value += fraction * value
            break

    return total_value

# Example usage:
items = [(10, 60), (20, 100), (30, 120)]  # Each item is represented as (weight, value)
capacity = 50
max_value = fractional_knapsack(items, capacity)
```

```
print("Maximum value that can be obtained:", max_value)
```

experiment 3

Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

```
def knapsack_0_1(values, weights, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]


    for i in range(n + 1):
        for w in range(capacity + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i - 1] <= w:
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i
- 1][w])
            else:
                dp[i][w] = dp[i - 1][w]


    return dp[n][capacity]


# Example usage:
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
max_value = knapsack_0_1(values, weights, capacity)
print("Maximum value that can be obtained:", max_value)
```

Experiment 4

Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.

```
def is_safe(board, row, col, n):
    # Check the left side of this row
    for i in range(col):
        if board[row][i] == 1:
            return False

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

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

    return True

def solve_nqueens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
```

```
    if solve_nqueens_util(board, 0, n) is False:
        print("No solution exists")
        return False

    print_solution(board, n)
    return True

def solve_nqueens_util(board, col, n):
    if col == n:
        return True

    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
            if solve_nqueens_util(board, col + 1, n):
                return True
            board[i][col] = 0

    return False

def print_solution(board, n):
    for i in range(n):
        for j in range(n):
```

```
            print(board[i][j], end=" ")
        print()

# Example usage:
n = 8  # Change the value of n to the desired board size
solve_nqueens(n)
```

Experiment 5

Write a program for analysis of quick sort by using deterministic and randomized variant

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

# Example usage:
arr = [5, 3, 1, 9, 8, 2, 4, 7]
sorted_arr = quick_sort(arr)
print("Sorted array:", sorted_arr)
```