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

```
def countFibs_non_recursive(low, high):
    f1 = 0
    f2 = 1
    f3 = 1

    result = 0
    fib_numbers = []
    while f1 <= high:
        if f1 >= low:
            result += 1
            fib_numbers.append(f1)
        f1, f2, f3 = f2, f3, f1 + f2
    return result, fib_numbers


def countFibs_recursive(n, low, high, memo={}):
    if n <= 1:
        return n, [n] if n >= low and n <= high else []

    if n not in memo:
        fib_n_minus_1, fib_numbers_n_minus_1 = countFibs_recursive(n - 1, low, high, memo)
        fib_n_minus_2, fib_numbers_n_minus_2 = countFibs_recursive(n - 2, low, high, memo)

        memo[n] = fib_n_minus_1 + fib_n_minus_2, fib_numbers_n_minus_1 + fib_numbers_n_minus_2

    total_fib_n, fib_numbers_n = memo[n]

    if total_fib_n >= low and total_fib_n <= high:
        fib_numbers_n.append(total_fib_n)

    return total_fib_n, fib_numbers_n

def countFibs_recursive_wrapper(low, high):
```

```
    result = 0
    n = 0
    fib_value, fib_numbers = countFibs_recursive(n, low, high)
    while fib_value <= high:
        if fib_value >= low:
            result += 1
        n += 1
        fib_value, fib_numbers = countFibs_recursive(n, low, high)
    return result, fib_numbers

# Test the non-recursive approach
low = 10
high = 100
count, fib_numbers = countFibs_non_recursive(low, high)
print("Count of Fibonacci Numbers (non-recursive) is", count)
print("Fibonacci Numbers (non-recursive) are", fib_numbers)

# Test the recursive approach
count, fib_numbers = countFibs_recursive_wrapper(low, high)
print("Count of Fibonacci Numbers (recursive) is", count)
print("Fibonacci Numbers (recursive) are", fib_numbers)
```

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

```
class Item:
    def __init__(self, profit, weight):
        self.profit = profit
        self.weight = weight

def cmp(a, b):
    ratio_a = a.profit / a.weight
    ratio_b = b.profit / b.weight
    return ratio_a > ratio_b

def fractionalKnapsack(W, arr):
    # Sorting items based on the profit/weight ratio
    arr.sort(key=lambda x: x.profit / x.weight, reverse=True)
    final_value = 0.0

    for i in range(len(arr)):
        if arr[i].weight <= W:
            final_value += arr[i].profit
            W -= arr[i].weight
        else:
            final_value += (arr[i].profit / arr[i].weight) * W
            break

    return final_value

# Driver code
W = 50
arr = [Item(60, 10), Item(100, 20), Item(120, 30)]
N = len(arr)

print("Maximum value that can be obtained:", fractionalKnapsack(W, arr))
```

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

```
def knapsack_greedy(max_weight, values, weights):
    n = len(values)
    items = list(zip(values, weights, range(n)))
    items.sort(key=lambda x: x[0] / x[1], reverse=True)  # Sort by value-to-weight ratio

    total_value = 0
    knapsack = [0] * n

    for value, weight, index in items:
        if max_weight >= weight:
            knapsack[index] = 1
            total_value += value
            max_weight -= weight

    return total_value, knapsack

# Driver code
max_weight = 50
values = [60, 100, 120]
weights = [10, 20, 30]

best_value, selected_items = knapsack_greedy(max_weight, values, weights)
print("Best value:", best_value)
print("Selected items:", selected_items)
```

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

```
def solve_n_queens(n):
    def is_safe(board, row, col):
        for i in range(col):
            if board[row][i] == 1:
                return False
            if row - i >= 0 and board[row - i][col - i] == 1:
                return False
            if row + i < n and board[row + i][col - i] == 1:
                return False
        return True

    def place_queen(board, col):
        if col >= n:
            solutions.append(["".join("Q" if cell == 1 else "." for cell in row) for row in board])
            return
        for i in range(n):
            if is_safe(board, i, col):
                board[i][col] = 1
                place_queen(board, col + 1)
                board[i][col] = 0

    solutions = []
    board = [[0] * n for _ in range(n)]
    place_queen(board, 0)
    return solutions

# Example usage:
n = 4  # Change to the desired board size
```

```
solutions = solve_n_queens(n)
for i, solution in enumerate(solutions):
    print(f"Solution {i + 1}:")
    for row in solution:
        print(row)
    print()
```

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

```
import random

def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[-1]  # Deterministic pivot (choose any element)
    left, right = [], []

    for element in arr[:-1]:
        if element < pivot:
            left.append(element)
        else:
            right.append(element)

    return quick_sort(left) + [pivot] + quick_sort(right)

def randomized_quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = random.choice(arr)  # Randomized pivot
    left, right = [], []

    for element in arr:
        if element < pivot:
```

```
            left.append(element)
        else:
            right.append(element)

    return randomized_quick_sort(left) + [pivot] + randomized_quick_sort(right)

# Example usage:
arr = [10, 7, 8, 9, 1, 5]
sorted_arr = quick_sort(arr)
print("Deterministic QuickSort:", sorted_arr)

sorted_arr_random = randomized_quick_sort(arr)
print("Randomized QuickSort:", sorted_arr_random)
```