DAA practical no 1:

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

In [1]:

```
def fibonacci_non_recursive(n):
    fib = [0, 1]
    while len(fib) <= n:
        fib.append(fib[-1] + fib[-2])
    return fib

def fibonacci_recursive(n):
    return [0] if n == 0 else [0, 1] if n == 1 else fibonacci_recursive(n - 1)

n = 6

print("Fibonacci sequence up to Fibonacci({}) using non-recursive method:".forr
print(fibonacci_non_recursive(n))

print("Fibonacci sequence up to Fibonacci({}) using recursive method:".format(r
print(fibonacci_recursive(n))
```

```
Fibonacci sequence up to Fibonacci(6) using non-recursive method:
[0, 1, 1, 2, 3, 5, 8]
Fibonacci sequence up to Fibonacci(6) using recursive method:
[0, 1, 1, 2, 3, 5, 8]
```

Let's analyze the time and space complexity of the provided programs for calculating Fibonacci numbers:

Non-Recursive Program: Time Complexity: O(n) Space Complexity: O(n) In the non-recursive program, the time complexity is O(n) because it iterates through the Fibonacci sequence once, filling in a list from 2 to n. The space complexity is also O(n) because it uses an additional list of size n+1 to store the Fibonacci numbers.

Recursive Program: Time Complexity: O(2^n) Space Complexity: O(n) (due to the function call stack) The recursive program has exponential time complexity, O(2^n), because it recalculates Fibonacci numbers for the same values multiple times, resulting in a significant amount of redundant computation. The space complexity is O(n) because of the function call stack. In the worst case, the stack can have at most n function calls.

So, in summary:

Non-Recursive Program: Time O(n), Space O(n) Recursive Program: Time O(2^n), Space O(n)
The non-recursive program is significantly more efficient, especially for large values of 'n', as it

In [ ]:

PRACTICAL NO 2

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

In [2]:

```
def fractional_knapsack(items, capacity):
    items.sort(key=lambda x: -x[1] / x[0])
    V, K = 0, []
    for w, v in items:
        f = min(1, capacity / w)
        V, K, capacity = V + f * v, K + [(w * f, v * f)], capacity - w * f
    return V, K

# Example usage:
items = [(2, 60), (3, 50), (5, 70), (7, 30)]
W = 5
max_value, selected_items = fractional_knapsack(items, W)
print("Maximum value:", max_value)
print("Selected items:", selected_items)
```

```
Maximum value: 110.0
Selected items: [(2, 60), (3, 50), (0.0, 0.0), (0.0, 0.0)]
```

In [ ]:

PRATICAL NO 3

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

In [3]:
```
def knapsack_01(v, w, c):
    n, dp, s = len(v), [0] * (c + 1), []

    for i in range(n):
        for j in range(c, 0, -1):
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]) if w[i] <= j else dp[j]

    i, j = n - 1, c

    while i >= 0 and j > 0:
        if w[i] <= j and dp[j] != dp[j - w[i]] + v[i]:
            s.append(i)
            j -= w[i]
        i -= 1

    return dp[c], s

# Example usage:
v, w, c = [60, 100, 120], [10, 20, 30], 50
m, s = knapsack_01(v, w, c)
print("Maximum value:", m)
print("Selected items:", s)
```

```
Maximum value: 220
Selected items: [1]
```

PRACTICAL NO 4

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

In [29]:
```
def is_safe(board, row, col):
    return all(
        board[i] != col and abs(board[i] - col) != row - i
        for i in range(row)
    )

def solve_n_queens(board, row, n):
    if row == n:
        return True

    for col in range(n):
        if is_safe(board, row, col):
            board[row] = col
            if solve_n_queens(board, row + 1, n):
                return True

    return False

n = 4  # Change this to the desired board size
board = [-1] * n

if solve_n_queens(board, 0, n):
    for row in board:
        print(" ".join("Q" if i == row else "." for i in range(n)))
else:
    print("No solution found.")
```

```
. Q . .
. . . Q
Q . . .
. . Q .
```

PRACTICAL NO 5

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

In [31]:
```
# import random, time

def partition(arr, low, high, randomized=False):
    if randomized: pivot_idx = random.randint(low, high); arr[pivot_idx], arr[]
    pivot, left, right = arr[low], low + 1, high
    while 1:
        while left <= right and arr[left] <= pivot: left += 1
        while arr[right] >= pivot and right >= left: right -= 1
        if right < left: break
        arr[left], arr[right] = arr[right], arr[left]
    arr[low], arr[right] = arr[right], arr[low]
    return right

def quick_sort(arr, low, high, randomized=False):
    if low < high:
        pivot_idx = partition(arr, low, high, randomized)
        quick_sort(arr, low, pivot_idx - 1, randomized)
        quick_sort(arr, pivot_idx + 1, high, randomized)

def analyze_quick_sort_performance(sizes):
    print("Input Size\tDeterministic Time\tRandomized Time")
    for size in sizes:
        arr = [random.randint(1, 10000) for _ in range(size)]
        arr_copy = arr.copy()
        start_time = time.time()
        quick_sort(arr, 0, size - 1)
        end_time = time.time()
        det_time = end_time - start_time
        start_time = time.time()
        quick_sort(arr_copy, 0, size - 1, randomized=True)
        end_time = time.time()
        rnd_time = end_time - start_time
        print(f"{size}\t\t{det_time:.6f}\t\t{rnd_time:.6f}")

analyze_quick_sort_performance([100, 1000, 10000])
```

```
Input Size      Deterministic Time      Randomized Time
100             0.000000                0.000998
1000            0.001786                0.008901
10000           0.029345                0.036304
```

deterministic variant.

In [33]:

```
import random, time

def partition(arr, low, high):
    pivot = arr[low]
    left, right = low + 1, high
    while left <= right:
        while left <= right and arr[left] <= pivot: left += 1
        while arr[right] >= pivot and right >= left: right -= 1
        if right < left:
            break
        arr[left], arr[right] = arr[right], arr[left]
    arr[low], arr[right] = arr[right], arr[low]
    return right

def quick_sort(arr, low, high):
    if low < high:
        pivot_idx = partition(arr, low, high)
        quick_sort(arr, low, pivot_idx - 1)
        quick_sort(arr, pivot_idx + 1, high)

def analyze_quick_sort_performance(sizes):
    print("Input Size\tDeterministic Time")
    for size in sizes:
        arr = [random.randint(1, 10000) for _ in range(size)]
        start_time = time.time()
        quick_sort(arr, 0, size - 1)
        end_time = time.time()
        print(f"{size}\t\t{end_time - start_time:.6f}")

analyze_quick_sort_performance([100, 1000, 10000])
```

```
Input Size      Deterministic Time
100             0.000954
1000            0.003004
10000           0.042854
```

In [ ]:

In [ ]: