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

```
import timeit

def fibonacci(n):
    """Non recursive fibonacci function"""
    for i in range(2, n + 1):
        fib_list[i] = fib_list[i - 1] + fib_list[i - 2]
    return fib_list[n]


def fibonacci_recursive(n):
    """Recursive fibonacci function"""
    if n == 0:
        return 0
    if n == 1:
        return 1
    fib_recur_list[n] = fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
    return fib_recur_list[n]


N = 20
RUNS = 1000
print(f"Given N = {N}\n{RUNS} runs")

fib_list = [0] * (N + 1)
fib_list[0] = 0
fib_list[1] = 1
print(
    "Fibonacci non-recursive:",
    fibonacci(N),
    "\tTime:",
    f'{timeit.timeit("fibonacci(N)", setup=f"from __main__ import fibonacci;N={N}",
number=RUNS):5f}',
    "O(n)\tSpace: O(1)",
)

fib_recur_list = [0] * (N + 1)
fib_recur_list[0] = 0
fib_recur_list[1] = 1
print(
    "Fibonacci recursive:\t",
    fibonacci_recursive(N),
    "\tTime:",
    f'{timeit.timeit("fibonacci_recursive(N)", setup=f"from __main__ import
fibonacci_recursive;N={N}", number=RUNS,):5f}',
    "O(2^n)\tSpace: O(n)",
)
```

This code defines two functions for calculating the Fibonacci sequence, one using a non-recursive approach and the other using a recursive approach. It also measures the execution time of these functions using the `timeit` module and prints the results.

Here's a step-by-step explanation of the code:

1. Import the `timeit` module, which is used for measuring the execution time of code.

2. Define a function called `fibonacci(n)` for calculating the nth Fibonacci number using a non-recursive approach. This function initializes a list `fib_list` to store Fibonacci numbers and then iteratively computes Fibonacci numbers up to the nth element. It uses a for loop to fill the `fib_list` from index 2 to `n` with Fibonacci numbers.

3. Define a function called `fibonacci_recursive(n)` for calculating the nth Fibonacci number using a recursive approach. This function uses recursion to calculate Fibonacci numbers. It has base cases for `n` equal to 0 and 1, and for other values of `n`, it recursively calls itself to compute Fibonacci numbers.

4. Set the value of `N` to 20, which determines the Fibonacci number to be calculated.

5. Set the value of `RUNS` to 1000, which determines the number of times the functions will be executed for timing measurements.

6. Initialize the `fib_list` and `fib_recur_list` lists with zeros. These lists will be used to store the Fibonacci numbers calculated by the non-recursive and recursive functions, respectively. The initial values for `fib_list` and `fib_recur_list` for the first two Fibonacci numbers (0 and 1).

7. Calculate and print the Fibonacci number for `N` using the non-recursive function `fibonacci(N)`. It also measures the execution time of the `fibonacci` function using `timeit.timeit` and prints the result. The time complexity for this non-recursive approach is O(n), and the space complexity is O(1).

8. Calculate and print the Fibonacci number for `N` using the recursive function `fibonacci_recursive(N)`. It also measures the execution time of the `fibonacci_recursive` function using `timeit.timeit` and prints the result. The time complexity for this recursive approach is O(2^n), and the space complexity is O(n).

In summary, this code defines two functions for calculating Fibonacci numbers, one using a non-recursive approach and the other using a recursive approach. It then compares the performance of these functions by measuring their execution times and prints the results, including their time and space complexities. The non-recursive approach is significantly more efficient for larger values of `N` due to its lower time complexity.

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

```
class ItemValue:

    def __init__(self, wt_, val_, ind_):
        self.wt = wt_
        self.val = val_
        self.ind = ind_
        self.cost = val_ // wt_

    def __lt__(self, other):
        return self.cost < other.cost


def fractionalKnapSack(wt, val, capacity):
    iVal = [ItemValue(wt[i], val[i], i) for i in range(len(wt))]

    # sorting items by cost
    iVal.sort(key=lambda x: x.cost, reverse=True)

    totalValue = 0
    for i in iVal:
        curWt = i.wt
        curVal = i.val
        if capacity - curWt >= 0:
            capacity -= curWt
            totalValue += curVal
        else:
            fraction = capacity / curWt
            totalValue += curVal * fraction
            capacity = int(capacity - (curWt * fraction))
            break
    return totalValue


if __name__ == "__main__":
    wt = [10, 40, 20, 30]
    val = [60, 40, 100, 120]
    capacity = 50

    # Function call
    maxValue = fractionalKnapSack(wt, val, capacity)
    print("Maximum value in Knapsack =", maxValue)
```

This code implements the Fractional Knapsack Problem using a Python class and a function. The Fractional Knapsack Problem is a classic optimization problem where you are given a set of items, each with a weight (`wt`) and a value (`val`), and a knapsack with a maximum weight capacity (`capacity`). The goal is to determine the maximum total value of items that can be placed in the knapsack without exceeding its weight capacity. You are allowed to take fractional parts of items, i.e., you can take a portion of an item if needed.

Here's a step-by-step explanation of the code:

1. `ItemValue` class: This class represents an item with its weight, value, and an index for reference. It also calculates the cost of an item, which is the value-to-weight ratio (`val` / `wt`) and is used for sorting the items.

- `_init_` method: This constructor initializes the item's weight (`wt`), value (`val`), index (`ind`), and cost (`cost`).

- `_lt_` method: This method is used for comparison when sorting items. It compares items based on their cost, allowing them to be sorted in descending order.

2. `fractionalKnapSack` function: This is the main function that solves the Fractional Knapsack Problem.

- It takes three parameters: `wt` (a list of item weights), `val` (a list of item values), and `capacity` (the maximum weight capacity of the knapsack).

- It creates a list of `ItemValue` objects (`iVal`) based on the weights, values, and indices of the items. Each object contains information about a single item.

- It sorts the list of items (`iVal`) based on their costs in descending order. This sorting ensures that items with the highest value-to-weight ratio come first.

- It initializes `totalValue` to 0, which will keep track of the maximum value obtained from the knapsack.

- It iterates through the sorted items and adds them to the knapsack as long as there is room (`capacity`) in the knapsack.

- If the current item's weight can fit entirely into the knapsack, it adds the full item's value and deducts the item's weight from the available capacity.

- If the current item's weight cannot fit entirely, it calculates the fraction that can fit, adds the corresponding fraction of the item's value to `totalValue`, and updates the available capacity. The loop then breaks because the knapsack is full.

- Finally, it returns the `totalValue`, which represents the maximum value that can be obtained without exceeding the knapsack's capacity.

3. In the `if __name___== "_main_":` block, the code defines the weights, values, and capacity for a specific instance of the knapsack problem

4. It calls the `fractionalKnapSack` function with the given inputs and prints the result, which is the maximum value that can be obtained in the knapsack.

This code demonstrates a solution to the Fractional Knapsack Problem by sorting items based on their cost and greedily selecting items until the knapsack is full, ensuring that the most valuable items are chosen first.

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

```
def knapsack_dp(W, wt, val, n):
    """A Dynamic Programming based solution for 0-1 Knapsack problem
    Returns the maximum value that can"""
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
    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]



val = [60, 100, 120]

wt = [10, 20, 30]
W = 50
n = len(val)
print("Maximum possible profit =", knapsack_dp(W, wt, val, n))
```

This code implements a solution for the 0-1 Knapsack problem using dynamic programming. The 0-1 Knapsack problem is a classic optimization problem where you are given a set of items, each with a weight (`wt`) and a value (`val`), and a knapsack with a maximum weight capacity (`W`). The goal is to determine the maximum total value of items that can be placed in the knapsack without exceeding its weight capacity. In this variant, you can either take an item (0) or leave it (1), hence the "0-1" Knapsack problem.

Here's a step-by-step explanation of the code:

1. `knapsack_dp` function: This is the main function that solves the 0-1 Knapsack problem using dynamic programming.

   - It takes four parameters: `W` (maximum weight capacity of the knapsack), `wt` (a list of item weights), `val` (a list of item values), and `n` (the number of items available).

   - It initializes a 2D table `K` of dimensions `(n+1) x (W+1)` with all elements initially set to 0. `K[i][w]` will represent the maximum value that can be obtained using the first `i` items while having a weight capacity of `w`.

- It uses a nested loop to iterate over all possible item indices and knapsack weight capacities. The outer loop iterates over the item indices from 0 to `n`, and the inner loop iterates over knapsack capacities from 0 to `W`.

- Within the inner loop, the code considers three cases:
  1. If there are no items (`i == 0`) or the knapsack capacity is 0 (`w == 0`), the maximum value is 0.
  2. If the weight of the current item (`wt[i - 1]`) is less than or equal to the current knapsack capacity (`w`), the code calculates the maximum value either by including the current item (`val[i - 1]`) and the best value obtained for the remaining capacity (`K[i - 1][w - wt[i - 1]]`) or by excluding the current item (`K[i - 1][w]`). It selects the maximum of these two values.
  3. If the weight of the current item is greater than the current knapsack capacity, the code simply copies the value obtained from the previous row (`K[i - 1][w]`).

- After completing the loop, the maximum value that can be obtained by selecting items from the first `n` items and having a knapsack capacity of `W` is stored in `K[n][W]`, and it returns this value.

2. In the main part of the code:

- It defines the values `val`, weights `wt`, the maximum knapsack capacity `W`, and the number of items `n`.

- It calls the `knapsack_dp` function with these inputs and prints the result, which is the maximum possible profit that can be obtained while satisfying the knapsack capacity constraint.

This code demonstrates a dynamic programming solution to the 0-1 Knapsack problem, which efficiently computes the maximum value that can be obtained while considering the weight and value constraints

Job

```
def printJobScheduling(arr, t):
    # length of array
    n = len(arr)

    # Sort all jobs according to
    # decreasing order of profit
    for i in range(n):
        for j in range(n - 1 - i):
            if arr[j][2] < arr[j + 1][2]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    # To keep track of free time slots
    result = [False] * t

    # To store result (Sequence of jobs)
    job = ['-1'] * t

    # Iterate through all given jobs
    for i in range(len(arr)):

        # Find a free slot for this job
        # (Note that we start from the
        # last possible slot)
        for j in range(min(t - 1, arr[i][1] - 1), -1, -1):

            # Free slot found
            if result[j] is False:
                result[j] = True
                job[j] = arr[i][0]
                break

    # print the sequence
    print(job)


# Driver's Code
if __name__ == '__main__':
    arr = [['a', 2, 100],  # Job Array
           ['b', 1, 19],
           ['c', 2, 27],
           ['d', 1, 25],
           ['e', 3, 15]]

    print("Following is maximum profit sequence of jobs")

    # Function Call
    printJobScheduling(arr, 3)
```

This code is for solving the Job Scheduling problem with the goal of maximizing the total profit. In this problem, you are given a set of jobs, each with a start time, finish time, and profit. The objective is to find a sequence of non-overlapping jobs that maximizes the total profit.

Here's a step-by-step explanation of the code:

1. `printJobScheduling` function: This function is used to find and print the sequence of jobs that maximizes the total profit. It takes two parameters: `arr`, which is a list of jobs with their respective start times, finish times, and profits, and `t`, which is the maximum possible time slot.

2. The length of the `arr` list is stored in the variable `n`.

3. The code sorts the list of jobs in descending order of their profits using a simple bubble sort algorithm. The inner loop compares the profits of adjacent jobs, and if the profit of the current job is greater than the next job, the two jobs are swapped. This sorting ensures that jobs with higher profits come first.

4. Two lists, `result` and `job`, are created. The `result` list is used to keep track of free time slots, and the `job` list is used to store the final sequence of jobs. Both lists are initialized accordingly.

5. The code iterates through all the given jobs using a for loop.

6. For each job, it searches for a free slot where the job can be scheduled. It starts searching from the last possible slot (the time slot closest to the finish time of the job).

7. If a free slot is found (indicated by `result[j]` being `False`), it sets `result[j]` to `True` to mark the slot as occupied, and it adds the job's name (in this case, 'a', 'b', 'c', etc.) to the `job` list for that time slot. The loop then breaks.

8. After processing all the jobs, the code prints the sequence of jobs stored in the `job` list, which represents the maximum profit sequence.

9. In the main part of the code:

   - It defines the list `arr`, where each element is a list representing a job with a name, start time, finish time, and profit.

   - It prints a message indicating that it's going to find the maximum profit sequence of jobs.

   - It calls the `printJobScheduling` function with the provided list of jobs (`arr`) and the maximum time slot `3`.

   - The result is printed, which will be the sequence of jobs that maximizes the total profit.

This code efficiently solves the Job Scheduling problem by sorting jobs based on their profit and greedily selecting non-overlapping jobs. The result is a sequence of jobs that maximizes the total profit while adhering to the time slot constraint.

Write a program to implement Huffman Encoding using a greedy strategy.

```
class Node:
    """A Huffman Tree Node"""

    def __init__(self, freq_, symbol_, left_=None, right_=None):
        # frequency of symbol
        self.freq = freq_

        # symbol name (character)
        self.symbol = symbol_

        # node left of current node
        self.left = left_

        # node right of current node
        self.right = right_

        # tree direction (0/1)
        self.huff = ""


def print_nodes(node, val=""):
    """Utility function to print huffman codes for all symbols in the newly created Huffman tree"""
    # huffman code for current node
    new_val = val + str(node.huff)

    # if node is not an edge node then traverse inside it
    if node.left:
        print_nodes(node.left, new_val)
    if node.right:
        print_nodes(node.right, new_val)

    # if node is edge node then display its huffman code
    if not node.left and not node.right:
        print(f"{node.symbol} -> {new_val}")


# characters for huffman tree
chars = ["a", "b", "c", "d", "e", "f"]

# frequency of characters
freq = [5, 9, 12, 13, 16, 45]

# list containing huffman tree nodes of characters and frequencies
nodes = [Node(freq[x], chars[x]) for x in range(len(chars))]

while len(nodes) > 1:
    # sort all the nodes in ascending order based on their frequency
    nodes = sorted(nodes, key=lambda x: x.freq)

    # pick 2 smallest nodes
    left = nodes[0]
    right = nodes[1]

    # assign directional value to these nodes
    left.huff = 0
    right.huff = 1
```

```
    # combine the 2 smallest nodes to create new node as their parent
    newNode = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)

    # remove the 2 nodes and add their parent as new node among others
    nodes.remove(left)
    nodes.remove(right)
    nodes.append(newNode)


print("Characters :", f'[{", ".join(chars)}]')
print("Frequency  :", freq, "\n\nHuffman Encoding:")
print_nodes(nodes[0])
```

This code demonstrates the creation of a Huffman Tree and the generation of Huffman codes for a given set of characters based on their frequencies. The Huffman coding algorithm is used to generate variable-length codes, with shorter codes assigned to more frequent characters.

Here's a step-by-step explanation of the code:

1. `Node` class: This class defines the structure of a node in the Huffman Tree.

   - The constructor `__init__` takes four parameters:
     - `freq_`: The frequency of the symbol (character).
     - `symbol_`: The symbol name (character).
     - `left_`: The left child node (default is `None`).
     - `right_`: The right child node (default is `None`).

   - Each node has attributes to store the frequency (`freq`), symbol name (`symbol`), left child node (`left`), right child node (`right`), and a `huff` attribute to represent the tree direction (0 or 1).

2. `print_nodes` function: This is a utility function used to print the Huffman codes for all symbols in the newly created Huffman tree. It performs a depth-first traversal of the tree, accumulating the Huffman code along the way.

   - The function takes two parameters: `node`, which is the current node, and `val`, which is the accumulated Huffman code up to the current node.

   - It recursively traverses the Huffman tree and builds the Huffman code (`new_val`) for each symbol.

   - When it reaches an edge node (a leaf node with no children), it prints the symbol and its corresponding Huffman code.

3. The code defines a list of characters (`chars`) and their corresponding frequencies (`freq`). These represent the characters that we want to generate Huffman codes for.

4. It creates a list of `Node` objects (`nodes`) where each node represents a character and its frequency. Each node is initialized with a frequency and a symbol (character).

5. The main part of the code contains the Huffman Tree construction logic:

   - It enters a loop that continues as long as there are more than one node in the `nodes` list.

- Inside the loop, it sorts the nodes in ascending order based on their frequencies using the `sorted` function with a custom sorting key.

- It selects the two smallest nodes from the sorted list (`left` and `right`).

- It assigns the `huff` attribute to these nodes, with `left.huff` set to 0 and `right.huff` set to 1.

- It creates a new parent node (`newNode`) by combining the two smallest nodes. The frequency of the new node is the sum of the frequencies of its children, and its symbol is a concatenation of the symbols of its children.

- It removes the two smallest nodes from the `nodes` list and adds the new parent node (`newNode`) to the list.

- The loop continues until only one node remains in the `nodes` list, which represents the root of the Huffman Tree.

6. After constructing the Huffman Tree, it prints the characters and their corresponding frequencies, and then it prints the Huffman encoding for each character using the `print_nodes` function.

The code effectively constructs a Huffman Tree and generates Huffman codes for a set of characters based on their frequencies, enabling efficient data compression and decompression.

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

```
class NQBacktracking:
    def __init__(self, x_, y_):
        """self.ld is an array where its indices indicate row-col+N-1
        (N-1) is for shifting the difference to store negative indices"""
        self.ld = [0] * 30

        """ self.rd is an array where its indices indicate row+col and used
        to check whether a queen can be placed on right diagonal or not"""
        self.rd = [0] * 30

        """column array where its indices indicates column and
        used to check whether a queen can be placed in that row or not"""
        self.cl = [0] * 30

        """Initial position of 1st queen"""
        self.x = x_
        self.y = y_

    def printSolution(self, board):
        """A utility function to print solution"""
        print(
            "N Queen Backtracking Solution:\nGiven initial position of 1st queen at row:",
            self.x,
            "column:",
            self.y,
            "\n",
        )
        for line in board:
            print(" ".join(map(str, line)))
```

```
def solveNQUtil(self, board, col):
    """A recursive utility function to solve N
    Queen problem"""

    # base case: If all queens are placed then return True
    if col >= N:
        return True

    # Overlook the column where 1st queen is placed
    if col == self.y:
        return self.solveNQUtil(board, col + 1)

    for i in range(N):
        # Overlook the row where 1st queen is placed
        if i == self.x:
            continue
        # Consider this column and try placing
        # this queen in all rows one by one

        # Check if the queen can be placed on board[i][col]
        # A check if a queen can be placed on board[row][col].
        # We just need to check self.ld[row-col+n-1] and self.rd[row+coln]
        # where self.ld and self.rd are for left and right diagonal respectively
        if (self.ld[i - col + N - 1] != 1 and self.rd[i + col] != 1) and self.cl[
            i
        ] != 1:

            # lace this queen in board[i][col]
            board[i][col] = 1
            self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 1

            # recur to place rest of the queens
            if self.solveNQUtil(board, col + 1):
                return True

            # If placing queen in board[i][col]
            # doesn't lead to a solution,
            # then remove queen from board[i][col]
            board[i][col] = 0  # BACKTRACK
            self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 0

            # If the queen cannot be placed in
            # any row in this column col then return False
            # print("col:", col, "i:", i, board)
    return False

def solveNQ(self):
    """This function solves the N Queen problem using
    Backtracking. It mainly uses solveNQUtil() to
    solve the problem. It returns False if queens
    cannot be placed, otherwise, return True and
    prints placement of queens in the form of 1s.
    Please note that there may be more than one
    solutions, this function prints one of the
    feasible solutions."""
    board = [[0 for _ in range(N)] for _ in range(N)]
    board[self.x][self.y] = 1
```

```
        self.ld[self.x - self.y + N - 1] = self.rd[self.x + self.y] = self.cl[
            self.x
        ] = 1

        if not self.solveNQUtil(board, 0):
            print("Solution does not exist")
            return False
        self.printSolution(board)
        return True


if __name__ == "__main__":
    N = 8
    x, y = 3, 2

    NQBt = NQBacktracking(x, y)
    NQBt.solveNQ()
```

This code implements a solution to the N-Queens problem using backtracking. The N-Queens problem is a classic combinatorial problem where you have to place N queens on an N×N chessboard in such a way that no two queens threaten each other. That is, no two queens can be in the same row, column, or diagonal.

Here's a step-by-step explanation of the code:

1. `NQBacktracking` class: This class is used to represent and solve the N-Queens problem with backtracking.

   - The constructor `_init_` takes two parameters: `x_` and `y_`, which represent the initial position of the 1st queen. It also initializes three lists (`ld`, `rd`, and `cl`) to keep track of conflicts in the left diagonal, right diagonal, and columns.

   - `ld` (left diagonal) is an array where its indices indicate row - col + N - 1 (N - 1 is used to shift the difference to store negative indices).
   - `rd` (right diagonal) is an array where its indices indicate row + col.
   - `cl` (column) is an array where its indices indicate columns. It is used to check whether a queen can be placed in that row or not.

   - The initial position of the 1st queen is set as `self.x` and `self.y`.

2. `printSolution` method: This method is a utility function used to print the solution (placement of queens on the chessboard).

3. `solveNQUtil` method: This is a recursive utility function used to solve the N-Queens problem.

- The base case is when all queens are placed (`col` >= `N`), in which case it returns `True`.

- It overlooks the column where the 1st queen is placed (`col == self.y`) and calls itself recursively for the next column.

- It iterates through each row (`i`) in the current column (`col`) and checks if a queen can be placed in that cell without any conflicts in the left diagonal (`ld`), right diagonal (`rd`), and column (`cl`).

- If a queen can be placed, it sets the cell to 1 and updates the `ld`, `rd`, and `cl` arrays to mark conflicts. Then, it calls itself recursively for the next column. If the recursion succeeds (`True` is returned), it means a solution has been found, and it returns `True`. Otherwise, it backtracks by setting the cell to 0 and resetting the arrays, indicating that the current placement didn't lead to a solution.

- If no solution is found for the current column, it returns `False`.

4. `solveNQ` method: This method solves the N-Queens problem using backtracking. It initializes the chessboard with the initial position of the 1st queen and marks the corresponding `ld`, `rd`, and `cl` arrays. It then calls `solveNQUtil` to find the solution. If a solution is found, it prints the placement of queens. If no solution exists, it prints "Solution does not exist."

5. In the main part of the code:

- It sets the value of `N` to 8 (representing an 8x8 chessboard) and the initial position of the 1st queen to row 3 and column 2.

- It creates an instance of the `NQBacktracking` class with these parameters and then calls the `solveNQ` method to find and print the solution.

The code efficiently solves the N-Queens problem by utilizing backtracking to explore possible placements of queens on the chessboard while ensuring that no two queens threaten each other.