A-1

Fibonacci numbers

```
def fibonacci_iterative(n:int):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        dp = [0] * n
        dp[0] = 0
        dp[1] = 1
        for i in range(2,n):
            dp[i] = dp[i-1] + dp[i-2]
        return dp[n-1]

def fibonacci_recursive(n):
    cache = {
        1:0,
        2:1
    }
    return helper(n,cache)

def helper(n:int,cache):
    if n in cache:
        return cache[n]
    else:
        return helper(n-1,cache) + helper(n-2,cache)


n = int(input("Enter value of n(nth Fibonacci number): "))
print(f"Fibonacci Number(Iterative): {fibonacci_iterative(n)}")
print(f"Fibonacci Number(Recursive): {fibonacci_recursive(n)}")
```

```
Enter value of n(nth Fibonacci number):  10
```

```
Fibonacci Number(Iterative): 34
Fibonacci Number(Recursive): 34
```

A-2

Huffman encoding

```
class Node:
    def __init__(self,left=None,right=None,value=None,frequency=None):
        self.left = left
        self.right = right
        self.value = value
        self.frequency = frequency

    def children(self):
        return (self.left,self.right)



class Huffman_Encoding:
    def __init__(self,string):
        self.q = []
        self.string = string
        self.encoding = {}

    def char_frequency(self):
        count = {}
        for char in self.string:
            if char not in count:
                count[char] = 0
            count[char] += 1

        for char,value in count.items():
            node = Node(value=char,frequency=value)
            self.q.append(node)
        self.q.sort(key=lambda x: x.frequency)

    def build_tree(self):
        while len(self.q) > 1:
            n1 = self.q.pop(0)
            n2 = self.q.pop(0)
            node = Node(left=n1,right=n2,frequency=n1.frequency +
n2.frequency)
            self.q.append(node)
            self.q.sort(key = lambda x:x.frequency)


    def helper(self,node:Node,binary_str="",):
        if type(node.value) is str:
            self.encoding[node.value] = binary_str
            return
        l,r = node.children()
```

```python
        self.helper(node.left,binary_str + "0")
        self.helper(node.right,binary_str + "1")
        print(node.frequency)
        return


    def huffman_encoding(self):
        root = self.q[0]
        self.helper(root,"")


    def print_encoding(self):
        print(' Char | Huffman code ')
        for char,binary in self.encoding.items():
            print(" %-4r |%12s" % (char,binary))

    def encode(self):
        self.char_frequency()
        self.build_tree()
        self.huffman_encoding()
        self.print_encoding()

string = input("Enter string to be encoded: ")
# string = 'AAAAAAABBCCCCCCDDDEEEEEEEEE'
encode = Huffman_Encoding(string)
encode.encode()


# The time complexity for encoding each unique character based on its
frequency is O(nlog n).

# Extracting minimum frequency from the priority queue takes place
2*(n-1) times and its complexity is O(log n). Thus the overall
complexity is O(nlog n).
```

```
Enter string to be encoded:  my name is prajwal

2
4
2
2
4
8
2
2
4
6
10
18
 Char | Huffman code
```

```
 'm'  |        000
 'y'  |       0010
 'n'  |       0011
 'e'  |       0100
 'i'  |       0101
 's'  |       0110
 'p'  |       0111
 'r'  |       1000
 'j'  |       1001
 'w'  |       1010
 'l'  |       1011
 ' '  |        110
 'a'  |        111
```

A-3

Kanpsack

```
# Class to represent an item in the knapsack
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight

    # Function to calculate value-to-weight ratio
    def value_per_weight(self):
        return self.value / self.weight

# Function to solve the Fractional Knapsack Problem
def fractional_knapsack(items, capacity):
    # Sort items by value-to-weight ratio in descending order
    items.sort(key=lambda item: item.value_per_weight(), reverse=True)

    total_value = 0.0  # Total value accumulated in the knapsack
    total_weight = 0   # Total weight accumulated in the knapsack

    # Loop through the sorted items
    for item in items:
        if total_weight + item.weight <= capacity:
            # If adding the whole item doesn't exceed capacity, take
the whole item
            total_weight += item.weight
            total_value += item.value
        else:
            # Otherwise, take a fraction of the item to fill the
knapsack
            remaining_capacity = capacity - total_weight
```

```
            fraction = remaining_capacity / item.weight
            total_value += item.value * fraction
            total_weight += item.weight * fraction
            break  # Knapsack is now full

    return total_value

# Main function to test the Fractional Knapsack
if __name__ == "__main__":
    # Taking input from the user
    n = int(input("Enter the number of items: "))

    items = []
    for i in range(n):
        value = float(input(f"Enter value of item {i + 1}: "))
        weight = float(input(f"Enter weight of item {i + 1}: "))
        items.append(Item(value, weight))

    # Input: Capacity of the knapsack
    capacity = float(input("Enter the capacity of the knapsack: "))

    # Calculate the maximum value of the knapsack
    max_value = fractional_knapsack(items, capacity)

    print(f"\nMaximum value in Knapsack: {max_value:.2f}")
```

```
Enter the number of items:  5
Enter value of item 1:  20
Enter weight of item 1:  20
Enter value of item 2:  30
Enter weight of item 2:  30
Enter value of item 3:  40
Enter weight of item 3:  40
Enter value of item 4:  50
Enter weight of item 4:  50
Enter value of item 5:  60
Enter weight of item 5:  60
Enter the capacity of the knapsack:  50


Maximum value in Knapsack: 50.00
```

A-4

0-1 Knapsack

```
def knapsack(values,weights,capacity):
    dp = [[0 for i in range(capacity+1)] for j in range(len(values)
```

```
+1)]

    for item in range(1,len(values) + 1):
        for weight in range(1,capacity + 1):
            if weights[item - 1] <= weight:
                dp[item][weight] = max(dp[item-1][weight-weights[item-
1]]+values[item-1],dp[item-1][weight])
            else:
                dp[item][weight] = dp[item-1][weight]
    return dp[-1][-1]


while True:
    print("Press Ctrl+C to terminate...")
    n = int(input('Enter number of items: '))
    values = [int(i) for i in input("Enter values of items:").split("
")]
    weights = [int(i) for i in input("Enter weights of
items:").split(" ")]
    capacity = int(input("Enter maximum weight: "))
    maximum_value = knapsack(values,weights,capacity)
    print('The maximum value of items that can be carried:',
maximum_value)
```

```
Press Ctrl+C to terminate...

Enter number of items:  2
Enter values of items: 10
Enter weights of items: 30
Enter maximum weight:  40

The maximum value of items that can be carried: 10
Press Ctrl+C to terminate...
```

A-5

n-queens

```
class NQueens:
    def __init__(self) -> None:
        self.size = int(input("Enter size of chessboard: "))
        self.board = [[False]*self.size for _ in range(self.size)]
        self.count = 0
    def printBoard(self):
        for row in self.board:
            for ele in row:
                if ele == True:
                    print("Q",end=" ")
```

```python
                else:
                    print("X",end=" ")
            print()
        print()

    def isSafe(self,row:int,col:int) -> bool:

        # Check Column(above and below of the (row,col))
        for i in self.board:
            if i[col] == True:
                return False

        # Check backward slash(\) diagonal only in above direction
        i = row
        j = col
        while i >= 0 and j >= 0:
            if self.board[i][j] == True:
                return False
            i -= 1
            j -= 1
        # Check backward slash(\) diagonal only in below direction
        i = row
        j = col
        while i < self.size and j < self.size:
            if self.board[i][j] == True:
                return False
            i += 1
            j += 1

        # Check forward slash diagonal(/) only in above direction
        i = row
        j = col
        while i >= 0 and j < self.size:
            if self.board[i][j] == True:
                return False
            i -= 1
            j += 1

         # Check forward slash diagonal(/) only in below direction
        i = row
        j = col
        while i < self.size and j >= 0:
            if self.board[i][j] == True:
                return False
            i += 1
            j -= 1

        return True
```

```
    def set_position_first_queen(self):
        print("Enter coordinates of first queen: ")
        row = int(input(f"Enter row (1-{self.size}): "))
        col = int(input(f"Enter column (1-{self.size}): "))
        self.board[row-1][col-1] = True
        self.printBoard()

    def solve(self,row:int):
        if row == self.size:
            self.count += 1
            self.printBoard()
            return

        if any(self.board[row]) is True:
            self.solve(row+1)
            return

        for col in range(self.size):
            if self.isSafe(row,col) == True:
                self.board[row][col] = True
                self.solve(row+1)
                self.board[row][col] = False

    def displayMessage(self):
        if self.count > 0:
            print("Solution exists for the given position of the
queen.")
        else:
            print("Solution doesn't exist for the given position of
the queen.")

solver = NQueens()
solver.set_position_first_queen()
solver.solve(0)
solver.displayMessage()

Enter size of chessboard:  5

Enter coordinates of first queen:

Enter row (1-5):  4
Enter column (1-5):  3

X X X X X
X X X X X
X X X X X
X X Q X X
X X X X X

X Q X X X
```

```
X X X Q X
Q X X X X
X X Q X X
X X X X Q

X X X Q X
X Q X X X
X X X X Q
X X Q X X
Q X X X X

Solution exists for the given position of the queen.
```