LP3 - DESIGN AND ANALYSIS ALGORITHM

Samruddhi Bagul BE-1 405A028

```
# Practical -1 : Write a program to calculate Fibonacci numbers and find its step count

# Both recursive and non-recursive

def recursive_fibonacci(n): #defining a recursive function
  if n <= 1:
    return n
  else:
    return recursive_fibonacci(n-1) + recursive_fibonacci(n-2) #will return sum of two terms

def non_recursive_fibonacci(n):
  first = 0
  second = 1
  step_count = 2
  print(first)
  print(second)
  while step_count < n:
    third = first + second
    first = second
    second = third
    print(third)
    step_count += 1
  return step_count

n = int(input("Enter the nth term: "))
print("The fibonacci sequence using recursive function:")
for i in range(n):
  print(recursive_fibonacci(i))

print("The fibonacci sequence using non-recursive function")
steps = non_recursive_fibonacci(n)
print(f"Steps: {steps}")
```

Enter the nth term: 6
The fibonacci sequence using recursive function:
0
1
1
2
3
5
The fibonacci sequence using non-recursive function
0
1
1
2
3
5
Steps: 6

```python
# Practical-2 : Huffman encoding using greedy strategy

import heapq

def calculate_frequency(s):
    frequency = {}
    for char in s:
        if char not in frequency:
            frequency[char] = 0
        frequency[char] += 1
    return frequency

def huffman_encode(frequency):
    heap = [[weight, [char, ""]] for char, weight in frequency.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

s =input("Enter the string or words to generate their huffman encoding:")
frequency = calculate_frequency(s)
huff = huffman_encode(frequency)
print(f"frequency of the chars of given string is: {frequency}")
print("Char | Huffman code ")
print("----------------------")
for char, huffman_code in huff:
    print(f"  {char} | {huffman_code}")
```

```
Enter the string or words to generate their huffman encoding:hello i am threpsham
frequency of the chars of given string is: {'h': 3, 'e': 2, 'l': 2, 'o': 1, ' ': 3, 'i': 1, 'a': 3, 'm': 2, 'p': 1, 'r': 1, 't': 1, 's': 1}
Char | Huffman code
----------------------
   | 100
 a | 101
 h | 110
 l | 000
 m | 001
 e | 1110
 p | 0100
 r | 0101
 s | 0110
 t | 0111
 i | 11110
 o | 11111
```

```
# Practical-3 : Solve a fractional Knapsack problem using a greedy method

def fractional_knapsack():
    weights=[10,20,30]
    values=[60,100,120]
    capacity=50
    res=0
    # Pair : [Weight,value]
    for pair in sorted(zip(weights,values), key= lambda x: x[1]/x[0], reverse=True):
        if capacity<=0: # Capacity completed - Bag fully filled
            break
        if pair[0]>capacity: # Current's weight with highest value/weight ratio Available Capacity
            res+=int(capacity * (pair[1]/pair[0]))  # Completely fill the bag
            capacity=0
        elif pair[0]<=capacity: # Take the whole object
            res+=pair[1]
            capacity-=pair[0]
    print(res)

if __name__=="__main__":
    fractional_knapsack()
```

240

```python
# Practical 4 : Write a program to solve a 0-1 Knapsack problem using dynamic programming or
# branch and bound strategy

# DPP

def knapSack_DP(W, wt, val, n):
   K = [[0 for w in range(W + 1)] for i in range(n + 1)]

   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(knapSack_DP(W, wt, val, n))
```

220

#BB

```
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 knapSack_BB(wt, val, capacity):
    iVal = []
    for i in range(len(wt)):
        iVal.append(ItemValue(wt[i], val[i], i))

    iVal.sort(reverse=True)

    totalValue = 0
    for i in iVal:
        curWt = int(i.wt)
        curVal = int(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

val = [60, 100, 120]
wt = [10, 20, 30]
capacity = 50
print(knapSack_BB(wt, val, capacity))
```

240.0

Practical 5 : Design 8-Queens matrix having first Queen placed. Use backtracking to place
remaining Queens to generate the final 8-queen's matrix.

```
def print_board(board):
   n = len(board)
   for i in range(n):
      for j in range(n):
         print(board[i][j], end=" ")
      print()

def is_safe(board, row, col):
   n = len(board)

   # Check left side of the current row
   for i in range(col):
      if board[row][i] == 1:
         return False

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

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

   return True

def solve_n_queens(board, col):
   n = len(board)

   if col >= n:
      # All queens are placed, return True
      return True

   if col == first_queen_col:
      # First queen is placed, move to next column
      return solve_n_queens(board, col + 1)

   # Try placing the queen in each row of the current column
   for i in range(n):
      if i != first_queen_row and is_safe(board, i, col):
         # Place the queen
         board[i][col] = 1

         # Recur to place the rest of the queens
```

```
        if solve_n_queens(board, col + 1):
            return True

        # If placing the queen in board[i][col] doesn't lead to a solution, backtrack
        board[i][col] = 0

    return False

def main():
    n = 8  # Change 'n' to the desired board size
    board = [[0 for _ in range(n)] for _ in range(n)]

    # Place the first queen at
    board[first_queen_row][first_queen_col] = 1

    # Call the backtracking function to solve the rest of the board
    if solve_n_queens(board, 0):
        print("Solution exists:")
        print_board(board)
    else:
        print("No solution exists.")

if __name__ == "__main__":
    main()
```

```
Solution exists:
0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
```