Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

Group-A: Assignment No.: 1

Title: Recursive and Non-recursive function

Problem statement: To calculate Fibonacci numbers using recursive and non-recursive functions and analyze their time and space complexity.

Objective: Analyzing time and space complexity

Theory:

- **Fibonacci series**

The Fibonacci series is a series of natural numbers where the next number is equivalent to the sum of the previous two numbers.

So, you wrote a recursive algorithm, for example, a recursive function example for up to 5

fibonacci(5) = fibonacci(4) + fibonacci(3)

fibonacci(3) = fibonacci(2) + fibonacci(1)

fibonacci(4) = fibonacci(3) + fibonacci(2)

fibonacci(2) = fibonacci(1) + fibonacci(0)

The first two numbers in the Fibonacci sequence are either 1 and 1, or 0 and 1, and each subsequent number is the sum of the previous two numbers.

- **Fibonacci Series Logic:**

Initializing number: the nth term is the sum of (n-1)th term and (n-2)th term Call Same Function Until Origin Condition

- **Algorithm: Iterative Algorithm**

```
1   Algorithm Fibonacci(n)
2   // Compute the nth Fibonacci number.
3   {
4       if (n ≤ 1) then
5           write (n);
6       else
7       {
8           fnm2 := 0; fnm1 := 1;
9           for i := 2 to n do
10          {
11              fn := fnm1 + fnm2;
12              fnm2 := fnm1; fnm1 := fn;
13          }
14          write (fn);
15      }
16  }
```

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

Analysis:

Two cases (1) n = 0 or 1 and (2) n > 1.

1) When n = 0 or 1, lines 4 and 5 get executed once each.
2) When n > 1, lines 4, 8, and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed n-1 times each.

The total steps for the case n > 1 is therefore 4n + 1.

Iterative Program:

 # Program to display the Fibonacci sequence up to the n-th term

```
nterms = int(input("Enter number of terms "))

# first two terms
n1, n2 = 0, 1
count = 0

# check if the number of terms is valid
if nterms <= 0:
print("Please enter a positive integer")

# If there is only one term, return n1
elif nterms == 1:
print("Fibonacci sequence upto", nterms,":")
print(n1)

#generate Fibonacci sequence else:
print("Fibonacci sequence:")
while count nterms:print(n1)
nth = n1 + n2
n2 = nth
n1=n2
count+=1
```

Output: Enter a number of terms 4 Fibonacci sequence:

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

0
1
1
2

Analysis:

$T(n) = T(n-1) + T(n-2) + c$

$= 2T(n-1) + c$ //from the approximation $T(n-1) \sim T(n-2)$

$= 2*(2T(n-2) + c) + c$

$= 4T(n-2) + 3c$

$= 8T(n-3) + 7c$

$= 2^k * T(n - k) + (2^k - 1)*c$

Let's find the value of k for which: $n - k$

$= 0$ $k = n$

$T(n) = 2^n * T(0) + (2^n - 1)*c$

<u>Conclusion:</u>

Hence, we have studied recursive and non-recursive functions to calculate Fibonacci numbers.

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

Group-A: Assignment No.: 2
Title: Implement Huffman Encoding Using a Greedy Strategy
Problem Statement:
To implement Huffman Encoding for data compression using a greedy Algorithm
Objective:

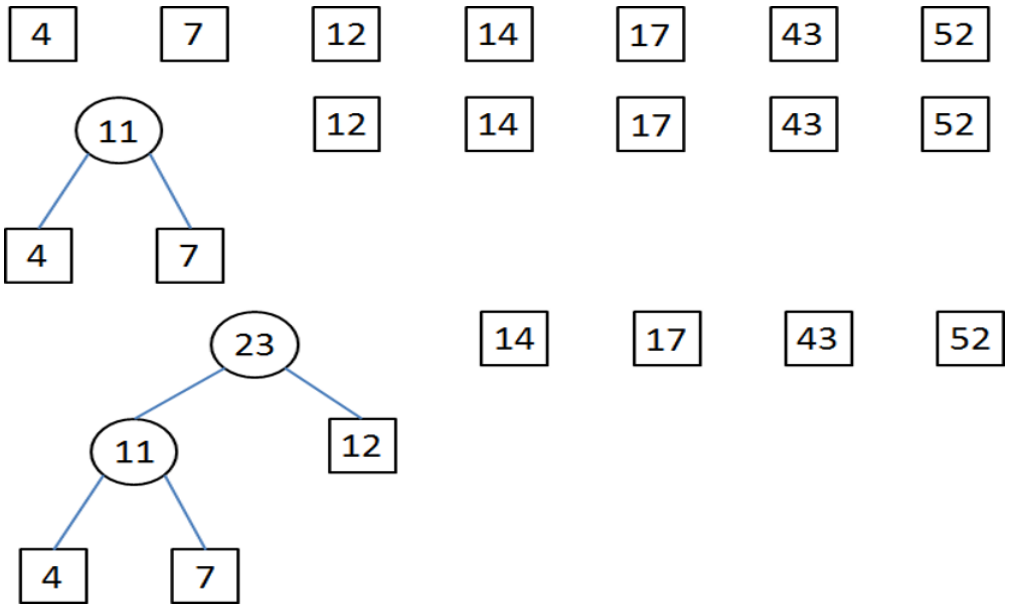
1. To understand and implement Huffman Encoding using a Greedy search Algorithm

Theory:
Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent fractional knapsack problem using the greedy algorithm gets the smallest code and the least frequent character gets the largest code.
There are mainly two major parts in Huffman Coding:

1. **Build a Huffman Tree from input characters.**
2. **Traverse the Huffman Tree and assign codes to characters.**

1. **Steps to build Huffman Tree**
Step 1: Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.
Step 2: Create a leaf node for each unique character and build a minimum heap of all leaf nodes
Step 3: The value of the frequency field is used to compare two nodes in the min heap.
Step 4: Extract two nodes with the minimum frequency from the min heap.
Step 5: Create a new internal node with a frequency equal to the sum of the two nodes' frequencies.
Step 6: Repeat steps#2 and #3 until the heap contains only one node.
Let us understand the algorithm with an example:

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)



Algorithm:

```
def printNodes(node, val=''):
newVal = val
str(node.huff)
if(node.left):
        print nodes(node.left, newVal)
        if(node.right):
        print nodes(node.right, newVal)
        if(not node.left and not node.right):
                print(f"{node. symbol} -> {newVal}")

 # characters for Huffman tree
 chars = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

 # frequency of characters
 freq = [ 4, 7, 12, 14, 17, 43, 54]

 # list containing unused nodes
 nodes = []

# converting characters and frequencies into Huffman tree nodes
for x in range(len(chars)):
```

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

```
nodes.append(node(freq[x], chars[x]))
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 a 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 a new node among other
nodes.remove(left)
nodes.remove(right) nodes.append(newNode)

# Huffman Tree is ready!
print nodes(nodes[0])
```

Output

```
a -> 0000 b -> 0001 c -> 001
d -> 010
e -> 011
f -> 10
g -> 11
```

Conclusion:

Hence, we have successfully implemented Huffman encoding using a greedy strategy.

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

Group-A: Assignment No.: 3
Title: Fractional Knapsack Problem Using Greedy Algorithm
Problem Statement: To solve a fractional Knapsack problem using a greedy method.
Objective:
Interpret Fractional Knapsack Problem
THEORY:

- **What is the Knapsack Problem?**

A thief went to a store to steal some items. There are multiple items available of different weights & profits. Let's suppose there are 'n' No. of items & weight of these are W1, W2,……Wn respectively, and the profit of these items are P1, P2, and Pn respectively.
Knapsack Problem may be of 2 types:

1. 0/1 Knapsack problem
2. Fractional Knapsack problem

1. <u>0/1 Knapsack problem</u>

- In this problem, either a whole item is selected (1) or the whole item is not to be selected (0).
- Here, the thief can't carry a fraction of the item.
- In the LPP (Linear programming problem) form, it can be described as:

```
1   Algorithm GreedyKnapsack(m, n)
2   // p[1 : n] and w[1 : n] contain the profits and weights respectively
3   // of the n objects ordered such that p[i]/w[i] ≥ p[i + 1]/w[i + 1].
4   // m is the knapsack size and x[1 : n] is the solution vector.
5   {
6       for i := 1 to n do x[i] := 0.0; // Initialize x.
7       U := m;
8       for i := 1 to n do
9       {
10          if (w[i] > U) then break;
11          x[i] := 1.0; U := U − w[i];
12      }
13      if (i ≤ n) then x[i] := U/w[i];
14  }
```

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

- **ALGORITHM**

```
def fractional_knapsack(value, weight, capacity):
ratio = [v/w for v, w in zip(value, weight)]
index.sort(key=lambda i: ratio[i], reverse=True)
max_value = 0
fractions = [0]*len(value)
for i in index:
if weight[i] <= capacity:
fractions[i] = 1
max_value += value[i]
capacity -= weight[i]
else:
fractions[i] = capacity/weight[i]
max_value += value[i]*capacity/weight[i]
break
return max_value, fractions
n = int(input('Enter number of items: '))
value = input('Enter the values of the {} item(s) in order: '.format(n)).split()
value = [int(v) for v in value]
weight = input('Enter the positive weights of the {} item(s) in order: format(n)).split()
weight = [int(w) for w in weight]
capacity = int(input('Enter maximum weight: '))
```

Program Output: Enter the number of items: 3
Enter the values of the 3 item(s) in order: 24 15 25
Enter the positive weights of the 3 item(s) in order: 15 10 18
Enter maximum weight: 20
The maximum value of items that can be carried: 31.5
The fractions in which the items should be taken: [1, 0.5, 0]

CONCLUSION:

Hence, We have successfully studied the fractional knapsack problem using a greedy algorithm.

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

Group-A: Assignment No.: 4

Title: Implementation of 0/1 knapsack using Dynamic Programming or Branch and Bound Strategy

Problem Statement: To solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

THEORY:

Branch and bound is an algorithm design paradigm that is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in the worst case. Branch and Bound solve these problems relatively quickly.

Let us consider below 0/1 Knapsack problem below to understand Branch and Bound.

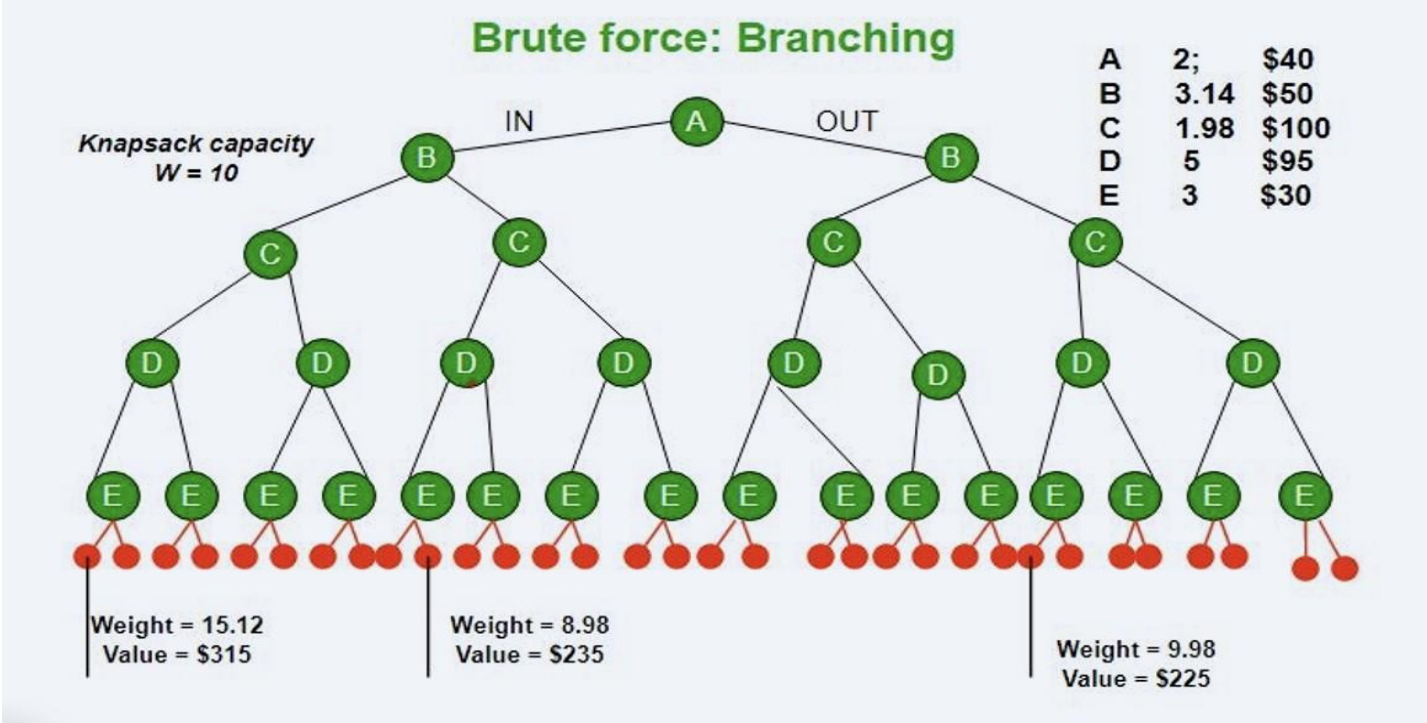
Given two integer arrays **val[0..n-1]** and **wt[0..n-1]** that represent values and weights associated with n items respectively.

Find out the maximum value subset of val[] such that the sum of the weights of this subset is smaller than or equal to Knapsack capacity W. Let us explore all approaches for this problem.

1. A Greedy approach is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problems and may not produce the correct result for 0/1 knapsack.

2. We can use Dynamic Programming (DP) for the 0/1 Knapsack problem. In DP, we use a 2D table of size n x W.

3. Since the DP solution doesn't always work, a solution is to use Brute Force. With n items, there are 2^n solutions to be generated, check each to see if they satisfy the constraint, and save the maximum solution that satisfies the constraint. This solution can be expressed as a tree**.**



4. We can use **Backtracking** to optimize the Brute Force solution. In the tree representation, we can do DFS of the tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring.

CONCLUSION:

Hence, we have successfully implemented the 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

Group-A: Assignment No.: 5

Title: Implementation of Queens Matrix using Backtracking

Problem statement:

To implement the Queen's matrix having first Queen placed. Also, use backtracking to place the remaining Queens to generate the final 8-queen matrix.

Objective:

Implement 8-Queens problem using Backtracking

THEORY

- **Backtracking Algorithm**

A backtracking algorithm is a problem-solving algorithm that uses a brute-force approach to find the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions. The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions.

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

Algorithm:

Step 1: Start in the leftmost column

Step 2: If all queens are placed return true

Step 3: Try all rows in the current column. Do the following for every tried row.

a. If the queen can be placed safely in this row, then mark this as part of the solution and recursively check if placing the queen here leads to a solution.
b. If placing the queen in leads to a solution then return true.
c. If placing queen doesn't lead to a solution then unmark this and go to step (a) to try other rows.
d. If all rows have been tried and nothing worked, return false to trigger backtracking.

- **C++ program to solve N Queen Problem using backtracking**

```
#include
```

Laboratory Practice-3: [DAA + ML + BCT]
Group-A (DAA)

```
<bits/stdc++.h>
using namespace;
bool isSafe(int board[N][N], int row, int col){int i, j;
  if (board[row][i])
  return false;
for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
if (board[i][j])
return false;
for (i = row, j = col; j >= 0 && i < N; i++, j--)
if (board[i][j])
return false;
return true;
```

- **A recursive utility function to solve the N Queen problem**

```
bool solveNQUtil(int board[N][N], int col)
if (col >= N)
return true;
for (int i = 0; i < N; i++) {
if (isSafe(board, i, col)) {
if (solveNQUtil(board, col + 1))
return true;

// BACKTRACK
```

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, returns true and prints the placement of queens in the form of 1s. Please note that there may be more than one solution, this function prints one of the feasible solutions.

```
bool solveNQ(){
int board[N][N] = { { 0, 0, 0, 0 },
                    { 0, 0, 0, 0 },
```

```
                    { 0, 0, 0, 0 },
                    { 0, 0, 0, 0 } };
If (solveNQUtil (board, 0) ==   false)
cout<<"Solution does not exist"; return false;
print solution(board); return true;}
```

Output:

0010
1000
0001
0100

CONCLUSION:

Hence, we have successfully implemented N-Queen's problem using the backtracking algorithm.