DAA 1

```
#include <iostream>

using namespace std;

// Recursive function to calculate Fibonacci number at position n

int fibonacci(int n) {

   if (n <= 1)

      return n;

   return fibonacci(n - 1) + fibonacci(n - 2);

}

int main() {

   int n;

   // --- Iterative Method ---

   cout << "Fibonacci Series in C++ Without Using Recursion (Iterative):\n";

   cout << "Enter the number of terms: ";

   cin >> n;

   int t1 = 0, t2 = 1, nT;

   cout << "Fibonacci Series: ";

   for (int i = 1; i <= n; ++i) {

      cout << t1 << " ";

      nT = t1 + t2;

      t1 = t2;

      t2 = nT;

   }

   // --- Recursive Method ---

   int p;

   cout << "\n\nFibonacci in C++ Using Recursion:\n";

   cout << "Enter the position (n): ";

   cin >> p;

   cout << "Fibonacci number at position " << p << " is: " << fibonacci(p) << endl;

   return 0;

}
```

DAA 2

```
#include <iostream>
#include <queue>
#include <vector>
#include <unordered_map>
using namespace std;
struct Node {
char ch;
int freq;
Node *left, *right;

Node(char c, int f) {
ch = c;
freq = f;
left = right = nullptr;
}
};

struct compare {
bool operator()(Node* l, Node* r) {
return l->freq > r->freq;
}
};

void printCodes(Node* root, string str) {
if (!root) return;

if (!root->left && !root->right) {
cout << root->ch << ": " << str << "\n";
return;
}

printCodes(root->left, str + "0");
printCodes(root->right, str + "1");
}
void HuffmanCode(char data[], int freq[], int size) {
priority_queue<Node*, vector<Node*>, compare> minHeap;

for (int i = 0; i < size; i++) {
minHeap.push(new Node(data[i], freq[i]));
}

while (minHeap.size() > 1) {
Node *left = minHeap.top(); minHeap.pop();
Node *right = minHeap.top(); minHeap.pop();

Node *top = new Node('$', left->freq + right->freq);
top->left = left;
top->right = right;

minHeap.push(top);
}
printCodes(minHeap.top(), "");
}

int main() {
char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
int freq[] = {5, 9, 12, 13, 16, 45};
int size = sizeof(arr) / sizeof(arr[0]);
cout << "Character With their Huffman Codes:\n";
HuffmanCode(arr, freq, size);

return 0;
}
```

DAA3

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Structure to represent an item
struct Item {
    double value;
    double weight;
};

// Comparator function to sort items by value/weight ratio
bool cmp(Item a, Item b) {
    double r1 = a.value / a.weight;
    double r2 = b.value / b.weight;
    return r1 > r2; // Descending order
}

// Function to solve fractional knapsack
double fractionalKnapsack(int n, double W, vector<Item> &items) {
    // Sort items by value/weight ratio
    sort(items.begin(), items.end(), cmp);

    double totalValue = 0.0;

    for (int i = 0; i < n; i++) {
        if (items[i].weight <= W) {
            // Take whole item
            W -= items[i].weight;
            totalValue += items[i].value;
        } else {
            // Take fractional part
            totalValue += items[i].value * (W / items[i].weight);
            break; // Knapsack is full
        }
    }

    return totalValue;
}

int main() {
    int n;
    double W;
    cout << "Enter number of items: ";
    cin >> n;
    vector<Item> items(n);

    cout << "Enter value and weight of each item:\n";
    for (int i = 0; i < n; i++) {
        cin >> items[i].value >> items[i].weight;
    }

    cout << "Enter capacity of knapsack: ";
    cin >> W;

    double maxValue = fractionalKnapsack(n, W, items);
    cout << "Maximum value in the knapsack = " << maxValue << endl;

    return 0;
}

// Input

// Number of items: 3
// Values & weights:
// 60 10
// 100 20
// 120 30
// Knapsack capacity: 50
```

DAA4

```
#include <iostream>
#include <vector>
using namespace std;

int knapsack(int W, vector<int> &wt, vector<int> &val, int n) {
    // Create DP table
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    // Build table dp[][] in bottom-up manner
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;  // Base case
            else if (wt[i - 1] <= w)
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    // dp[n][W] contains the maximum value
    return dp[n][W];
}

int main() {
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;

    vector<int> val(n), wt(n);
    cout << "Enter value and weight of each item:\n";
    for (int i = 0; i < n; i++)
        cin >> val[i] >> wt[i];

    cout << "Enter capacity of knapsack: ";
    cin >> W;

    int maxValue = knapsack(W, wt, val, n);
    cout << "Maximum value in 0-1 Knapsack = " << maxValue << endl;

    return 0;
}

// Input

// Number of items: 3
// Values & weights:
// 60 10
// 100 20
// 120 30
// Knapsack capacity: 50
```

DAA5

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;


// Function to print the board

void printBoard(const vector<vector<int>>& board, int N) {

   for (int i = 0; i < N; i++) {

      for (int j = 0; j < N; j++)

         cout << board[i][j] << " ";

      cout << endl;

   }

}


// Function to check if placing a queen at (row, col) is safe

bool isSafe(const vector<vector<int>>& board, int row, int col, int N) {

   // Check same column

   for (int i = 0; i < row; i++)

      if (board[i][col] == 1) return false;


   // Check upper-left diagonal

   for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)

      if (board[i][j] == 1) return false;


   // Check upper-right diagonal

   for (int i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++)

      if (board[i][j] == 1) return false;


   return true;

}


// Backtracking function to place queens

bool solveNQueens(vector<vector<int>>& board, int row, int N) {

   if (row >= N) return true; // All queens placed


   // Skip row if queen already placed

   if (find(board[row].begin(), board[row].end(), 1) != board[row].end())

      return solveNQueens(board, row + 1, N);


   for (int col = 0; col < N; col++) {

      if (board[row][col] == 0 && isSafe(board, row, col, N)) {

         board[row][col] = 1; // Place queen

         if (solveNQueens(board, row + 1, N))

            return true;

         board[row][col] = 0; // Backtrack

      }

   }


   return false; // No valid position

}


int main() {

   int N;

   cout << "Enter size of board (N): ";

   cin >> N;


   vector<vector<int>> board(N, vector<int>(N, 0));


   int firstRow, firstCol;

   cout << "Enter position of first queen (row and column, 0-based index): ";

   cin >> firstRow >> firstCol;


   if (firstRow >= N || firstCol >= N || firstRow < 0 || firstCol < 0) {

      cout << "Invalid position!" << endl;
```

```
        return 0;
    }

    board[firstRow][firstCol] =
1; // Place first queen

    // Solve remaining
queens starting from row 0
    if (solveNQueens(board,
0, N)) {
        cout << "\nN-Queens
solution:\n";
        printBoard(board, N);
    } else {
        cout << "No solution
exists with the first queen at
the given position." << endl;
    }

    return 0;
}

// Input :
// Size of board: 4
// Position of first queen: 0
1

//size of board: 8
//position of first queen: 0 0
```