N-Queens -

```
/* class Solution {
public:
    void solve(int col, vector<vector<string>> &ans, vector<string>& board, int n,
            vector<int>& upperDiagonal, vector<int>& lowerDiagonal, vector<int>& leftRow) {
            if(col == n) {
                ans.push_back(board);
                return ;
            }

            for(int row=0 ; row<n ; row++) {
                if(upperDiagonal[row+col] == 0 && lowerDiagonal[(n-1)+col-row] == 0
                    && leftRow[row] == 0) {
                        board[row][col] = 'Q';
                        upperDiagonal[row+col] = 1;
                        lowerDiagonal[(n-1)+col-row] = 1;
                        leftRow[row] = 1;
                        solve(col+1, ans, board, n, upperDiagonal, lowerDiagonal, leftRow);
                        board[row][col] = '.';
                        upperDiagonal[row+col] = 0;
                        lowerDiagonal[(n-1)+col-row] = 0;
                        leftRow[row] = 0;
                    }
            }
        }

    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> ans;
        vector<string> board(n);
        string s(n, '.');
        for(int i=0 ; i<n ; i++) {
            board[i] = s;
        }

        vector<int> upperDiagonal(2*n-1, 0), lowerDiagonal(2*n-1, 0), leftRow(n, 0);
        solve(0, ans, board, n, upperDiagonal, lowerDiagonal, leftRow);

        return ans;
    }
}; */

/* class Solution {
public:
    bool isSafe(int row, int col, vector<string>& board, int n) {
        int x = row;
        int y = col;

        //checking for same row
        while(y >= 0){
            if(board[x][y] == 'Q') {
                return false;
            }
            y--;
        }

        x = row;
        y = col;
        // checking upper diagonal
        while(y>=0 && x>=0) {
            if(board[x][y] == 'Q') {
                return false;
            }
            x--;
            y--;
        }

        x = row;
        y = col;
        while(x < n && y >= 0) {
            if(board[x][y] == 'Q') {
                return false;
            }
            x++;
            y--;
        }

        return true;
    }

    void solve(int col, vector<vector<string>>& ans, vector<string>& board, int n) {
        if(col == n) {
            ans.push_back(board);
            return ;
        }

        for(int row = 0; row<n ; row++) {
            if(isSafe(row, col, board, n)) {
                board[row][col] = 'Q';
                solve(col+1, ans, board, n);
                board[row][col] = '.';
            }
        }
    }

    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> ans;
        vector<string> board(n);
        string s(n, '.');
        for(int i=0 ; i<n ; i++) {
            board[i] = s;
        }

        solve(0, ans, board, n);

        return ans;
    }
}; */
```

Huffman Encoding Greedy and Recursive Approaches –

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

// Node structure for Huffman Tree
struct Node {
    char data;
    int freq;
    Node *left, *right;

    Node(char data, int freq) {
        this->data = data;
        this->freq = freq;
        left = right = nullptr;
    }
};

// Comparison class for priority queue
struct Compare {
    bool operator()(Node* l, Node* r) {
        return l->freq > r->freq;
    }
};

// Recursive Approach
class HuffmanRecursive {
private:
    void encode(Node* root, string str, unordered_map<char, string>& huffmanCode) {
```

```
    if (root == nullptr) return;

    // Found a leaf node
    if (!root->left && !root->right) {
      huffmanCode[root->data] = str;
    }

    encode(root->left, str + "0", huffmanCode);
    encode(root->right, str + "1", huffmanCode);
  }

public:
  unordered_map<char, string> buildHuffmanTree(string text) {
    // Count frequency of characters
    unordered_map<char, int> freq;
    for (char ch : text) {
      freq[ch]++;
    }

    // Create priority queue
    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Create leaf nodes and add to priority queue
    for (auto pair : freq) {
      pq.push(new Node(pair.first, pair.second));
    }

    // Build Huffman Tree
    while (pq.size() > 1) {
      Node* left = pq.top(); pq.pop();
      Node* right = pq.top(); pq.pop();
```

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

            pq.push(parent);
        }

        // Generate Huffman codes
        unordered_map<char, string> huffmanCode;
        encode(pq.top(), "", huffmanCode);

        return huffmanCode;
    }
};

// Greedy Approach
class HuffmanGreedy {
public:
    unordered_map<char, string> buildHuffmanTree(string text) {
        // Count frequency of characters
        unordered_map<char, int> freq;
        for (char ch : text) {
            freq[ch]++;
        }

        // Create min heap using priority queue
        priority_queue<pair<int, char>, vector<pair<int, char>>, greater<pair<int, char>>> minHeap;

        // Add all characters to min heap
        for (auto pair : freq) {
```

```
    minHeap.push({pair.second, pair.first});
}

unordered_map<char, string> huffmanCode;

// Build codes greedily
while (minHeap.size() > 1) {
    auto first = minHeap.top(); minHeap.pop();
    auto second = minHeap.top(); minHeap.pop();

    // Add '0' to all codes of first
    for (auto& code : huffmanCode) {
        if (code.first == first.second) {
            code.second = "0" + code.second;
        }
    }
    if (huffmanCode.find(first.second) == huffmanCode.end()) {
        huffmanCode[first.second] = "0";
    }

    // Add '1' to all codes of second
    for (auto& code : huffmanCode) {
        if (code.first == second.second) {
            code.second = "1" + code.second;
        }
    }
    if (huffmanCode.find(second.second) == huffmanCode.end()) {
        huffmanCode[second.second] = "1";
    }

    // Add combined frequency back to heap
```

```
            minHeap.push({first.first + second.first, min(first.second, second.second)});
        }

        return huffmanCode;
    }
};

// Example usage
int main() {
    string text = "hello world";

    // Recursive approach
    HuffmanRecursive huffmanRecursive;
    auto recursiveCode = huffmanRecursive.buildHuffmanTree(text);

    cout << "Recursive Approach Huffman Codes:\n";
    for (auto pair : recursiveCode) {
        cout << pair.first << ": " << pair.second << endl;
    }

    // Greedy approach
    HuffmanGreedy huffmanGreedy;
    auto greedyCode = huffmanGreedy.buildHuffmanTree(text);

    cout << "\nGreedy Approach Huffman Codes:\n";
    for (auto pair : greedyCode) {
        cout << pair.first << ": " << pair.second << endl;
    }

    return 0;
}
```

```
/*
Recursive Approach Huffman Codes:
 : 1111
w: 1110
l: 10
o: 110
d: 000
e: 001
r: 010
h: 011

Greedy Approach Huffman Codes:
l: 0
w: 1
r: 10
 : 1100
o: 00
d: 1
e: 10
h: 1
*/
```

Fractional Knapsack –

```
#include<bits/stdc++.h>
using namespace std;

double fractionalKnapsack(vector<int>& values, vector<int>& weights, int w, int n) {
    vector<pair<int, int>> arr(n);
    for(int i=0 ; i<n ; i++){
        arr[i] = {values[i], weights[i]};
    }

    sort(arr.begin(), arr.end(), [](pair<int, int>&a, pair<int, int>& b) {
        // return a.second < b.second;
        double r1 = (double)a.first / (double)a.second;
        double r2 = (double)b.first / (double)b.second;

        return r1 > r2;
    });

    for(auto it : arr) {
        cout << it.first << " " << it.second << endl;
    }

    double ans = 0;
    for(int i=0 ; i<arr.size() ; i++) {
        int value = arr[i].first;
        int weight = arr[i].second;

        if(weight <= w) {
            ans += value;
            w -= weight;
        } else {
            ans += ((double)value / (double)weight) * w;
            break;
        }
    }

    return ans;
}
```

```
int main(){
   int n;
   cout << "Enter the number of elements - ";
   cin >> n;

   vector<int> values(n, 0), weights(n, 0);
   cout << "Enter the values - " << endl;
   for(int i=0 ; i<n ; i++) {
      cin >> values[i];
   }

   cout << "Enter the weights - " << endl;
   for(int i=0 ; i<n ; i++) {
      cin >> weights[i];
   }

   int w;
   cout << "Enter the weight of Knapsack - ";
   cin >> w;

   cout << "\nValues - ";
   for(int i=0 ; i<n ; i++) {
      cout << values[i] << " ";
   }

   cout << "\nWeights - ";
   for(int i=0 ; i<n ; i++) {
      cout << weights[i] << " ";
   }

   cout << "\nWeight of your Knapsack is - " << w << endl;
```

```
    double ans = fractionalKnapsack(values, weights, w, n);
    cout << "\nMaximum value that can be obtained is - " << ans << endl;

    return 0;
}

/*
Enter the number of elements - 10
Enter the values -
8 2 10 1 9 7 2 6 4 9
Enter the weights -
10 1 7 7 5 1 8 6 8 7
Enter the weight of Knapsack - 21

Values - 8 2 10 1 9 7 2 6 4 9
Weights - 10 1 7 7 5 1 8 6 8 7
Weight of your Knapsack is - 21

7 1
2 1
9 5
10 7
9 7
6 6
8 10
4 8
2 8
1 7

Maximum value that can be obtained is - 37
```

```
*/
```

0/1 Knapsack Problem –

```
#include<bits/stdc++.h>
using namespace std;

int greedy(vector<int>& values, vector<int>& weights, int w, int n) {
   vector<pair<int, int>> arr(n);
   for(int i = 0; i < n; i++) {
       arr[i] = {values[i], weights[i]};
   }

   sort(arr.begin(), arr.end(), [](pair<int, int>& a, pair<int, int>& b) {
       double r1 = (double)a.first / (double)a.second;
       double r2 = (double)b.first / (double)b.second;

       return r1 > r2;
       // return a.first > b.first;
   });

   int ans = 0;
   for(int i = 0; i < n; i++) {
       int value = arr[i].first;
       int weight = arr[i].second;

       if(weight <= w) {
           w -= weight;
           ans += value;
       } else {
           break;
       }
   }

   return ans;
}

int recursive(vector<int>& values, vector<int>& weights, int w, int ind, int n) {
   if(ind >= n || w <= 0) {
       if(weights[ind] <= w) {
           return values[ind];
       }
       return 0;
   }

   int take = INT_MIN;
   if(weights[ind] <= w) {
       take = values[ind] + recursive(values, weights, w-weights[ind], ind+1, n);
   }

   int notTake = 0 + recursive(values, weights, w, ind+1, n);

   return max(take, notTake);
}

int memoization(vector<int>& values, vector<int>& weights, int w, int ind, int n,
vector<vector<int>>& dp) {
   if(ind >= n) {
       if(weights[ind] <= w) {
           return values[ind];
       }
       return 0;
   }

   if(dp[ind][w] != -1) {
       return dp[ind][w];
   }

   int take = INT_MIN;
   if(weights[ind] <= w) {
       take = values[ind] + memoization(values, weights, w-weights[ind], ind+1, n, dp);
   }
```

```
   int notTake = 0 + memoization(values, weights, w, ind+1, n, dp);

   return dp[ind][w] = max(take, notTake);
}
int main() {
   int n;
   cout << "Enter the number of elements - ";
   cin >> n;

   vector<int> values(n), weights(n);
   cout << "Enter the values - ";
   for (int i = 0; i < n; i++) {
      cin >> values[i];
   }

   cout << "Enter the weights - ";
   for (int i = 0; i < n; i++) {
      cin >> weights[i];
   }

   int w;
   cout << "Enter the weight of Knapsack - ";
   cin >> w;

   cout << "\nValues - ";
   for(int i=0 ; i<n ; i++) {
      cout << values[i] << " ";
   }

   cout << "\nWeights - ";
```

```
    for(int i=0 ; i<n ; i++) {
        cout << weights[i] << " ";
    }

    cout << "\nWeight of your Knapsack is - " << w << endl;

    // int ans = greedy(values, weights, w, n);
    // int ans = recursive(values, weights, w, 0, n);

    vector<vector<int>> dp(n, vector<int> (w+1, -1));
    int ans = memoization(values, weights, w, 0, n, dp);
    cout << "\nMaximum profit that can be obtained is - " << ans << endl;

    return 0;
}

/*
Enter the number of elements - 5
Enter the values - 12 35 41 25 32
Enter the weights - 20 24 36 40 42
Enter the weight of Knapsack - 100

Values - 12 35 41 25 32
Weights - 20 24 36 40 42
Weight of your Knapsack is - 100

Maximum profit that can be obtained is - 101

*/
```

Write a smart contract on a test network, for Bank account of a customer for following operations: • Deposit money • Withdraw Money • Show balance

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Bank {
   // Mapping to store each customer's balance
   mapping(address => uint) private balances;

   // Event to log deposits
   event Deposit(address indexed customer, uint amount);

   // Event to log withdrawals
   event Withdrawal(address indexed customer, uint amount);

   // Function to deposit money into the account
   function deposit() external payable {
      require(msg.value > 0, "Deposit amount must be greater than zero.");
      balances[msg.sender] += msg.value;
      emit Deposit(msg.sender, msg.value);
   }

   // Function to withdraw money from the account
   function withdraw(uint amount) external {
      require(amount > 0, "Withdrawal amount must be greater than zero.");
      require(balances[msg.sender] >= amount, "Insufficient balance.");

      balances[msg.sender] -= amount;
      payable(msg.sender).transfer(amount);
      emit Withdrawal(msg.sender, amount);
```

```
    }

    // Function to check the balance of the account
    function getBalance() external view returns (uint) {
        return balances[msg.sender];
    }
}
```

Write a program in solidity to create Student data. Use the following constructs: • Structures • Arrays • Fallback Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract StudentData {
    // Define a structure to hold student information
    struct Student {
        uint id;
        string name;
        uint age;
    }

    // Array to store the list of students
    Student[] public students;

    // Mapping to track if a student ID already exists
    mapping(uint => bool) private studentExists;

    // Event to emit when a student is added
```

```
event StudentAdded(uint id, string name, uint age);

// Function to add a new student
function addStudent(uint _id, string memory _name, uint _age) public {
   // Check if the student ID already exists
   require(!studentExists[_id], "Student with this ID already exists.");

   // Create a new student and add it to the array
   students.push(Student(_id, _name, _age));
   studentExists[_id] = true;

   // Emit an event when a new student is added
   emit StudentAdded(_id, _name, _age);
}

// Function to retrieve a student by index
function getStudent(uint index) public view returns (uint, string memory, uint) {
   require(index < students.length, "Invalid index.");
   Student memory student = students[index];
   return (student.id, student.name, student.age);
}

// Fallback function to handle unknown function calls or direct transfers
fallback() external payable {
   revert("Invalid function call. Please use a valid function.");
}

// Receive function to accept Ether directly to the contract
receive() external payable {}

// Function to check the contract balance (if any Ether is sent)
```

```
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```