1. Fibonacci

```
#include<iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;

int stepcount = 0;

void printFibonacciRecursive(int n)
{
    static int n1 = 0, n2 = 1, n3;

    if (n > 0)
    {
        stepcount++;
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
        cout << n3 << " ";
        printFibonacciRecursive(n - 1);
    }
}

void printFibonacciIterative(int number)
{
    int n1 = 0, n2 = 1, n3;
    cout << n1 << " " << n2 << " "; // Printing 0 and 1
    for(int i = 2; i < number; ++i) // Loop starts from 2 because 0 and 1
are already printed
    {
        n3 = n1 + n2;
        cout << n3 << " ";
        n1 = n2;
        n2 = n3;
    }
}

int main()
{
    int n;
    int choice;

    cout << "Enter the number of elements: ";
    cin >> n;

    cout << "Choose the method to generate Fibonacci series:\n";
    cout << "1. Recursive\n";
    cout << "2. Iterative\n";
    cout << "Enter choice (1 or 2): ";
    cin >> choice;

    auto start_time = high_resolution_clock::now();

    cout << "Fibonacci Series: ";
    if (choice == 1) {
        cout << "0 " << "1 "; // First two numbers
        printFibonacciRecursive(n - 2); // n-2 because 2 numbers are
already printed
```

```
    }
    else if (choice == 2) {
        printFibonacciIterative(n); // Iterative method
    }
    else {
        cout << "Invalid choice!";
        return 1;
    }

    auto end_time = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end_time - start_time);

    cout << "\nElapsed Time: " << duration.count() << " microseconds" <<
endl;

    // Additional space tracking
    cout << "Estimated Space Used: " << sizeof(int) * 3 * (n - 2) << "
bytes" << endl;

    return 0;
}

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------
```

2.HAUFFMAN

```
#include <iostream>
#include <vector>
#include <queue>
#include <chrono>
#include <map>
#include <cmath>

using namespace std;

class Node {
public:
    int freq;
    char symbol;
    Node* left;
    Node* right;
    Node(int freq, char symbol, Node* left = nullptr, Node* right =
nullptr)
        : freq(freq), symbol(symbol), left(left), right(right) {}

    bool operator<(const Node& other) const {
        return freq > other.freq;
    }
};

void calculateHuffmanCodes(const Node* node, const string& code,
map<char, string>& huffmanCodes) {
    if (node) {
        if (!node->left && !node->right) {
            huffmanCodes[node->symbol] = code;
        }
        calculateHuffmanCodes(node->left, code + "0", huffmanCodes);
        calculateHuffmanCodes(node->right, code + "1", huffmanCodes);
    }
}

int main() {
    int n;
    cout << "Enter the number of unique characters: ";
    cin >> n;

    vector<char> chars(n);
    vector<int> freqs(n);
    map<char, int> frequencyMap;

    cout << "Enter the characters and their frequencies:\n";
    for (int i = 0; i < n; ++i) {
        cout << "Character #" << (i + 1) << ": ";
        cin >> chars[i];
        cout << "Frequency of " << chars[i] << ": ";
        cin >> freqs[i];
        frequencyMap[chars[i]] = freqs[i];
    }

    // Create a priority queue for building the Huffman Tree
    priority_queue<Node> nodes;
    for (int i = 0; i < n; ++i) {
        nodes.push(Node(freqs[i], chars[i]));
    }

    auto start_time = chrono::high_resolution_clock::now();

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

        Node* newNode = new Node(left->freq + right->freq, '\0', left,
right);
        nodes.push(*newNode);
    }

    auto end_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end_time
- start_time);
    cout << "Huffman Tree Construction Elapsed Time: " <<
duration.count() << " microseconds" << endl;

    map<char, string> huffmanCodes;
    calculateHuffmanCodes(&nodes.top(), "", huffmanCodes);

    cout << "\nHuffman Codes:\n";
    for (const auto& kv : huffmanCodes) {
        cout << kv.first << " -> " << kv.second << endl;
    }

    double spaceUsed = 0;
    for (const auto& kv : huffmanCodes) {
        spaceUsed += frequencyMap[kv.first] * kv.second.length();
    }
    spaceUsed = ceil(spaceUsed / 8.0); // Convert bits to bytes
    cout << "\nEstimated Space Used for Huffman Codes: " << spaceUsed <<
" bytes" << endl;

    return 0;
}
```


----

3. fractional knapscak

```
#include <iostream>
#include <algorithm> // Include this for 'sort'
#include <chrono>
#include <iomanip>
using namespace std;

struct Item {
    int value;
    int weight;
};

class Solution {
public:
    static bool comp(Item a, Item b) {
        double r1 = static_cast<double>(a.value) /
static_cast<double>(a.weight);
        double r2 = static_cast<double>(b.value) /
static_cast<double>(b.weight);
        return r1 > r2;
    }

    double fractionalKnapsack(int W, Item arr[], int n) {
        sort(arr, arr + n, comp);
        int curWeight = 0;
        double finalValue = 0.0;

        for (int i = 0; i < n; i++) {
            if (curWeight + arr[i].weight <= W) {
                curWeight += arr[i].weight;
                finalValue += arr[i].value;
            } else {
                int remain = W - curWeight;
                finalValue += (arr[i].value /
static_cast<double>(arr[i].weight)) * static_cast<double>(remain);
                break;
            }
        }
        return finalValue;
    }
};

int main() {
    int n, weight;
    // User input for number of items and knapsack capacity
    cout << "Enter the number of items: ";
    cin >> n;
    cout << "Enter the maximum weight capacity of the knapsack: ";
    cin >> weight;

    // Dynamic array allocation based on user input
    Item* arr = new Item[n];

    // Taking values and weights for each item from the user
    for (int i = 0; i < n; i++) {
        cout << "Enter value and weight for item " << i + 1 << "
(separated by a space): ";
        cin >> arr[i].value >> arr[i].weight;
    }

    Solution obj;
    auto start_time = chrono::high_resolution_clock::now();
    double ans = obj.fractionalKnapsack(weight, arr, n);
    auto end_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end_time
- start_time);

    cout << "The maximum value is " << fixed << setprecision(2) << ans <<
endl;
    cout << "Elapsed Time: " << duration.count() << " microseconds" <<
endl;

    // Clean up dynamically allocated memory
    delete[] arr;
    return 0;
}
```

```
-------------------------------------------------------------------------
-------------------------------------------------------------------------
------------------------------------------------------------------
```

4. 0/1 knapsack

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

pair<int, vector<int>> knapsack_01(int n, vector<int>& values,
vector<int>& weights, int W) {
    // Create a 2D DP array to store the maximum value at each n, W
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    // Build the DP table
    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 (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i -
1]] + values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }

        // Print the DP table after processing each item
        cout << "DP table after considering item " << i << ":\n";
        for (int row = 0; row <= n; row++) {
            for (int col = 0; col <= W; col++) {
                cout << dp[row][col] << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    // Find out which items were included
    vector<int> selected_items;
    int i = n, w = W;
    while (i > 0 && w > 0) {
        if (dp[i][w] != dp[i - 1][w]) {
            selected_items.push_back(i - 1); // Item is included
            w -= weights[i - 1];
        }
        i--;
    }

    return {dp[n][W], selected_items}; // Return the maximum value and
selected items
}

int main() {
    int n, W;

    // Taking user input for number of items and knapsack capacity
    cout << "Enter the number of items: ";
    cin >> n;

    vector<int> values(n), weights(n);

    // Taking user input for values of the items
    cout << "Enter the values of the items: ";
    for (int i = 0; i < n; i++) {
        cin >> values[i];
    }

    // Taking user input for weights of the items
    cout << "Enter the weights of the items: ";
    for (int i = 0; i < n; i++) {
        cin >> weights[i];
    }

    // Taking user input for the knapsack capacity
    cout << "Enter the knapsack capacity: ";
    cin >> W;

    // Call knapsack function
    auto result = knapsack_01(n, values, weights, W);
    // Output the result
    cout << "Maximum value: " << result.first << endl;
    cout << "Selected items (0-indexed): ";
    for (int index : result.second) {
        cout << index << " "; // Output the indices of selected items
    }
    cout << endl;

    return 0;
}
```


----

5. NQUEENS

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

class NQueens {
public:
    vector<vector<string>> solveNQueens(int n, int first_queen_col) {
        // Initialize sets for columns and diagonals
        set<int> col, posDiag, negDiag;
        vector<vector<string>> res;
        vector<string> board(n, string(n, '.'));

        // Place the first queen
        col.insert(first_queen_col);
        posDiag.insert(0 + first_queen_col);
        negDiag.insert(0 - first_queen_col);
        board[0][first_queen_col] = 'Q';

        // Start backtracking from the second row
        backtrack(1, n, col, posDiag, negDiag, board, res);
        return res;
    }

private:
    void backtrack(int r, int n, set<int>& col, set<int>& posDiag,
set<int>& negDiag,
                   vector<string>& board, vector<vector<string>>& res) {
        if (r == n) {
            res.push_back(board);
            return;
        }

        for (int c = 0; c < n; c++) {
            if (col.count(c) || posDiag.count(r + c) || negDiag.count(r -
c)) {
                continue; // Skip if the column or diagonal is already
occupied
            }

            // Place queen
            col.insert(c);
            posDiag.insert(r + c);
            negDiag.insert(r - c);
            board[r][c] = 'Q';

            // Recur to place the next queen
            backtrack(r + 1, n, col, posDiag, negDiag, board, res);

            // Remove queen and backtrack
            col.erase(c);
            posDiag.erase(r + c);
            negDiag.erase(r - c);
            board[r][c] = '.';
        }
    }
};

int main() {
    int n, first_queen_col;
    cout << "Enter the size of the board (n): ";
    cin >> n;
    cout << "Enter the column index for the first queen (0 to " << n - 1
<< "): ";
    cin >> first_queen_col;

    if (first_queen_col < 0 || first_queen_col >= n) {
        cout << "Invalid column index!" << endl;
        return 1;
    }

    NQueens solver;
    auto solutions = solver.solveNQueens(n, first_queen_col);

    if (!solutions.empty()) {
        cout << "One of the solutions is:" << endl;
        for (const auto& row : solutions[0]) {
            cout << row << endl; // Output the first solution
        }
    } else {
        cout << "No solutions found." << endl;
    }

    return 0;
}
```

```
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------
```

BCT

3.Bank Account

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

contract BankAccount {
    address public owner;
    uint public balance;

    // Constructor marked as payable to allow receiving Ether on contract
creation
    constructor() payable {
        owner = msg.sender;
        balance = msg.value; // The contract will accept initial Ether
sent during deployment
    }

    // Deposit function to add funds to the account
    function deposit() public payable {
        balance += msg.value;
    }

    // Withdraw function to withdraw funds from the account
    function withdraw(uint amount) public {
        require(msg.sender == owner, "Only the owner can withdraw.");
        require(amount <= balance, "Insufficient balance.");
        balance -= amount;
        payable(msg.sender).transfer(amount);
    }

    // Show balance function
    function showBalance() public view returns (uint) {
        return balance;
    }
}

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-----------------------------------------------------------------
```

4.Product inventory

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

contract ProductInventory {
    address public owner;
    mapping(uint => uint) public productStock;

    // Constructor marked as payable to allow receiving Ether on contract
creation
    constructor() payable {
        owner = msg.sender;
    }

    // Modifier to restrict access to the owner only
    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can perform this
action.");
        _;
    }

    // Receive product to increase stock
    function receiveProduct(uint productId, uint quantity) public
onlyOwner {
        productStock[productId] += quantity;
    }

    // Sale product to decrease stock
    function sellProduct(uint productId, uint quantity) public onlyOwner
{
        require(productStock[productId] >= quantity, "Insufficient
stock.");
        productStock[productId] -= quantity;
    }

    // Show stock for a specific product
    function showStock(uint productId) public view returns (uint) {
        return productStock[productId];
    }
}
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-----------------------------------------------------------------
```

1.STUDENT

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

contract studentData{

    struct Student{
        uint id;
        string name;
        uint age;
        string course;
    }

    Student[] public students;
    mapping(uint=>bool) public studentExists;

    event studentAdded(uint id,string name,uint age,string course);
    event ReceivedEther(address indexed owner, uint value);

    fallback() external  payable {
        emit ReceivedEther(msg.sender, msg.value);
    }

    receive() external payable {
        emit ReceivedEther(msg.sender, msg.value);
    }

    function addStudent(uint id, string memory name, uint age,string
memory course) public {
        require(!studentExists[id],"Student exists already");
        students.push(Student(id,name,age,course));
        studentExists[id]=true;

        emit studentAdded(id, name, age, course);
    }
    function getStudent(uint id) public view returns(uint, string memory,
uint ,string memory){
        require(id<students.length,"Invalid index");
        return(students[id-1].id,students[id-1].name,students[id-
1].age,students[id-1].course);


    }

    function studentCount() public view returns(uint){
        return students.length;
    }






}
```


2.EMPLOYEE

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

contract EmployeeData {

    // Structure to hold employee data
    struct Employee {
        uint id;
        string name;
        uint age;
        string position;
        uint salary;
    }

    // Array to hold list of employees
    Employee[] public employees;
    uint public employeeCount = 0;

    // Function to add a new employee
    function addEmployee(string memory _name, uint _age, string memory
_position, uint _salary) public {
        employeeCount++;
        employees.push(Employee(employeeCount, _name, _age, _position,
_salary));
    }

    // Function to retrieve an employee by ID
    function getEmployee(uint _id) public view returns (uint, string
memory, uint, string memory, uint) {
        require(_id > 0 && _id <= employeeCount, "Invalid Employee ID");
        Employee memory emp = employees[_id - 1];  // Array is 0-indexed
        return (emp.id, emp.name, emp.age, emp.position, emp.salary);
    }

    // Fallback function
    fallback() external payable {
        revert("This contract does not accept direct payments");
    }

    // Receive function (optional, if you want to receive Ether)
    receive() external payable {
        // If you want to accept Ether, you can implement this
    }
}
```


5.CUSTOMER

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

contract CustomerData {

    // Structure to hold customer data
    struct Customer {
        uint id;
        string name;
        string email;
        uint balance;
    }

    // Array to store customers
    Customer[] public customers;
    uint public customerCount = 0;

    // Function to add a new customer
    function addCustomer(string memory _name, string memory _email, uint
_balance) public {
        customerCount++;
        customers.push(Customer(customerCount, _name, _email, _balance));
    }

    // Function to retrieve customer data by ID
    function getCustomer(uint _id) public view returns (uint, string
memory, string memory, uint) {
        require(_id > 0 && _id <= customerCount, "Customer ID is
invalid");
        Customer memory cust = customers[_id - 1]; // Array is 0-indexed
        return (cust.id, cust.name, cust.email, cust.balance);
    }

    // Fallback function - reverts any call with invalid function
    fallback() external payable {
        revert("This contract does not accept direct payments");
    }

    // Receive function (optional, if you want to handle Ether)
    receive() external payable {
        // Logic to handle Ether (if required)
    }
}
```