#fibonacci#

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

// Recursive function to calculate Fibonacci numbers
int fibonacciRecursive(int n) {
    if (n <= 1) return n;
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}

// Iterative function to calculate Fibonacci numbers
int fibonacciIterative(int n) {
    if (n <= 1) return n;

    int prev1 = 0, prev2 = 1, result;
    for (int i = 2; i <= n; i++) {
        result = prev1 + prev2;
        prev1 = prev2;
        prev2 = result;
    }
    return result;
}

int main() {
    int n;
```

```
    cout << "Enter the position of the Fibonacci number: ";
    cin >> n;

    // Timing the recursive approach in nanoseconds
    auto start_recursive = high_resolution_clock::now();
    int result_recursive = fibonacciRecursive(n);
    auto end_recursive = high_resolution_clock::now();
    auto duration_recursive = duration_cast<nanoseconds>(end_recursive - start_recursive);

    cout << "Recursive Fibonacci of " << n << " is " << result_recursive << endl;
    cout << "Time taken by recursive approach: " << duration_recursive.count() << "
nanoseconds" << endl;
    cout << "Space complexity of recursive approach: O(n) (due to recursion stack)" << endl;

    // Timing the iterative approach in nanoseconds
    auto start_iterative = high_resolution_clock::now();
    int result_iterative = fibonacciIterative(n);
    auto end_iterative = high_resolution_clock::now();
    auto duration_iterative = duration_cast<nanoseconds>(end_iterative - start_iterative);

    cout << "Iterative Fibonacci of " << n << " is " << result_iterative << endl;
    cout << "Time taken by iterative approach: " << duration_iterative.count() << " nanoseconds" << endl;
    cout << "Space complexity of iterative approach: O(1) (constant space)" << endl;

    return 0;
}
```

#Huffman#

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

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

    Node(char character, int frequency) {
        ch = character;
        freq = frequency;
        left = right = nullptr;
    }
};

// Comparator to order nodes in the priority queue based on frequency
struct Compare {
    bool operator()(Node* left, Node* right) {
        return left->freq > right->freq;
    }
};
```

```
// Function to print the Huffman codes from the root of Huffman Tree
void printCodes(Node* root, string str) {
    if (!root)
        return;

    // If this is a leaf node, print the character and its code
    if (!root->left && !root->right) {
        cout << root->ch << ": " << str << endl;
    }

    // Recursively print codes for left and right children
    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

// Function to build Huffman Tree and generate codes
void buildHuffmanTree(unordered_map<char, int> &freqMap) {
    priority_queue<Node*, vector<Node*>, Compare> minHeap;

    // Create a leaf node for each character and add it to the priority queue
    for (auto pair : freqMap) {
        minHeap.push(new Node(pair.first, pair.second));
    }

    // Iterate while size of heap doesn't become 1
    while (minHeap.size() != 1) {
        // Extract the two nodes with the lowest frequency
        Node *left = minHeap.top();
        minHeap.pop();
```

```
        Node *right = minHeap.top();
        minHeap.pop();

        // Create a new internal node with frequency equal to the sum of the two nodes
        int sum = left->freq + right->freq;
        Node *newNode = new Node('\0', sum);
        newNode->left = left;
        newNode->right = right;

        // Add this node to the priority queue
        minHeap.push(newNode);
    }

    // Root of Huffman Tree
    Node* root = minHeap.top();

    // Print Huffman codes using the Huffman Tree built
    printCodes(root, "");
}

int main() {
    int n;
    unordered_map<char, int> freqMap;

    // Take input from user
    cout << "Enter the number of characters: ";
    cin >> n;
```

```
    for (int i = 0; i < n; i++) {
        char ch;
        int freq;
        cout << "Enter character:" << endl;
        cin >> ch;
        cout<<"enter the frequency:"<<endl;
        cin>>freq;
        freqMap[ch] = freq;
    }

    // Build Huffman Tree and print codes
    buildHuffmanTree(freqMap);

    return 0;
}
```

#fractional Knap#

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

struct Item {
    int weight;
    int value;
    float ratio;
};
```

```
bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}

float fractionalKnapsack(int W, vector<Item>& items) {
    sort(items.begin(), items.end(), compare);

    int currWeight = 0;
    float totalValue = 0.0;



    cout << "Item   Weight   Value   Ratio     Taken\n";
    cout << "----------------------------------------\n";

    for (int i = 0; i < items.size(); i++) {
        if (currWeight + items[i].weight <= W) {

            currWeight += items[i].weight;
            totalValue += items[i].value;



            cout << i + 1 << "      " << items[i].weight << "        "
                << items[i].value << "       "
                << items[i].ratio << "      Fully\n";

        } else {
```

```
            int remainingWeight = W - currWeight;
            float fraction = (float)remainingWeight / items[i].weight;
            totalValue += items[i].value * fraction;



            cout << i + 1 << "      " << items[i].weight << "        "
                << items[i].value << "       "
                << items[i].ratio << "      Partially (" << (fraction * 100) << "%)\n";
            break;
        }
    }

    return totalValue;
}

int main() {
    int n, W;



    cout << "Enter the number of items: ";
    cin >> n;
    cout << "Enter the capacity of the knapsack: ";
    cin >> W;

    vector<Item> items(n);



    for (int i = 0; i < n; i++) {
        cout << "Enter weight and value for item " << i + 1 << ": ";
```

```
        cin >> items[i].weight >> items[i].value;

        items[i].ratio = (float)items[i].value / items[i].weight;
    }

    cout << "\nTable of Items (sorted by value-to-weight ratio):\n";
    float maxValue = fractionalKnapsack(W, items);
    cout << "\nMaximum value we can obtain = " << maxValue << endl;

    return 0;
}
```

#0/1 knapsack#

```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono; // For timing

void knapSack(int W, int wt[], int val[], int n) {
    int dp[n + 1][W + 1];

    // Start timing for DP table filling (O(n * W))
    auto start = high_resolution_clock::now();

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= W; j++) {
            if (i == 0 || j == 0)
```

```
            dp[i][j] = 0;
        else if (wt[i - 1] <= j)
            dp[i][j] = max(val[i - 1] + dp[i - 1][j - wt[i - 1]], dp[i - 1][j]);
        else
            dp[i][j] = dp[i - 1][j];
    }
}

auto end = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(end - start);

// Print DP table (optional for debugging)
cout << "\nDP Table:\n";
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= W; j++) {
        cout << dp[i][j] << "\t";
    }
    cout << endl;
}

cout << "\nTime taken to fill DP table: " << duration.count() << " microseconds\n";

// Backtracking to find the items included in the optimal solution (O(n))
vector<int> taken(n, 0);
int j = W;
for (int i = n; i > 0 && j > 0; i--) {
    if (dp[i][j] != dp[i - 1][j]) {
        taken[i - 1] = 1;
        j -= wt[i - 1];
```

```
        }
    }

    // Output taken items and maximum profit
    cout << "Objects taken (1 = taken, 0 = not taken):\n";
    cout << "{ ";
    for (int i = 0; i < n; i++) {
        cout << taken[i] << " ";
    }
    cout << "}\n";

    cout << "Maximum profit: " << dp[n][W] << endl;
}

int main() {
    int n, W;

    // Input for number of items
    cout << "Enter the number of items: ";
    cin >> n;

    int profit[n], weight[n];

    // Input for profits
    cout << "Enter the profits of the items: ";
    for (int i = 0; i < n; i++) {
        cin >> profit[i];
    }
```

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

    // Input for knapsack capacity
    cout << "Enter the knapsack capacity: ";
    cin >> W;

    // Calculate knapsack maximum profit and items taken
    knapSack(W, weight, profit, n);

    return 0;
}
```

#nqueen#

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

bool isSafe(int row, int col, int board[], int n) {
    // Check for queens in the same column or diagonals
    for (int i = 0; i < row; ++i) {
        if (board[i] == col || abs(i - row) == abs(board[i] - col)) {
            return false;
```

```
        }
    }
    return true;
}

void printBoard(int board[], int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (board[i] == j) {
                cout << "Q ";
            } else {
                cout << "_ ";
            }
        }
        cout << endl;
    }
    cout << endl;
}

void solveNQueensBacktracking(int row, int board[], int n, int& count) {
    if (row == n) {
        count++;
        cout << "Solution " << count << ":\n";
        printBoard(board, n);
        return;
    }

    for (int col = 0; col < n; ++col) {
        if (isSafe(row, col, board, n)) {
```

```
            board[row] = col;
            solveNQueensBacktracking(row + 1, board, n, count);
            board[row] = -1; // Backtrack
        }
    }
}

int main() {
    int n;
    cout << "Enter the size of the board (n): ";
    cin >> n;

    int board[n]; // Array to store the column position of queens in each row
    int count = 0;

    // Initialize board with -1 (no queen placed in any row)
    for (int i = 0; i < n; ++i) {
        board[i] = -1;
    }

    // Take the first queen's position from the user
    int initialRow, initialCol;
    cout << "Enter the row (0 to " << n-1 << ") for the first queen: ";
    cin >> initialRow;
    cout << "Enter the column (0 to " << n-1 << ") for the first queen: ";
    cin >> initialCol;

    // Place the first queen at the user-specified position
    if (initialRow < 0 || initialRow >= n || initialCol < 0 || initialCol >= n) {
```

```
        cout << "Invalid position for the first queen.\n";
        return 1;
    }

    board[initialRow] = initialCol;

    cout << "\nBacktracking Solutions:\n";
    solveNQueensBacktracking(initialRow + 1, board, n, count); // Start from the next row

    if (count == 0) {
        cout << "No solutions found with the first queen placed at (" << initialRow << ",
" << initialCol << ").\n";
    }
    return 0;
}
```

python

```
def print_sol(board):
    for row in board:
        print(' '.join(map(str, row)))

def is_safe(row, col, rows, left_diagonals, right_diagonals):
    if rows[row] or left_diagonals[row + col] or right_diagonals[col - row + N - 1]:
        return False
    return True
```

```
def solve(board, col, rows, left_diagonals, right_diagonals):
    if col >= N:
        return True
    for i in range(N):
        if is_safe(i, col, rows, left_diagonals, right_diagonals):
            rows[i] = True
            left_diagonals[i + col] = True
            right_diagonals[col - i + N - 1] = True
            board[i][col] = 1

            if solve(board, col + 1, rows, left_diagonals, right_diagonals):
                return True

            rows[i] = False
            left_diagonals[i + col] = False
            right_diagonals[col - i + N - 1] = False
            board[i][col] = 0

    return False

if _name_ == "_main_":
    N = int(input("Enter the number of rows for the square board: "))

    # Take the position for the first queen as input
    first_row = int(input("Enter the row position of the first queen (0-based index): "))
    first_col = int(input("Enter the column position of the first queen (0-based index): "))

    board = [[0] * N for _ in range(N)]
    rows = [False] * N
```

```
    left_diagonals = [False] * (2 * N - 1)
    right_diagonals = [False] * (2 * N - 1)

    # Place the first queen and update the tracking arrays
    board[first_row][first_col] = 1
    rows[first_row] = True
    left_diagonals[first_row + first_col] = True
    right_diagonals[first_col - first_row + N - 1] = True

    # Start solving from the next column after the initial position of the first queen
    ans = solve(board, first_col + 1, rows, left_diagonals, right_diagonals)

    if ans:
        print_sol(board)
    else:
        print("Solution does not exist")
```

##Blockchain Technology

*Code-3(Bank Sol/remix ide)

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

contract Bank {
    address public accHolder;
```

```
uint256 private balance = 0; // Make balance private for better encapsulation

constructor() {
    accHolder = msg.sender;
}

function withdraw(uint256 amount) public payable {
    require(msg.sender == accHolder, "You are not the account owner.");
    require(amount > 0, "Withdraw amount must be greater than 0.");
    require(amount <= balance, "You don't have enough balance.");

    // Transfer the specified amount to the account holder
    payable(msg.sender).transfer(amount);

    // Deduct the withdrawn amount from the balance
    balance -= amount;
}

function deposit() public payable {
    require(msg.sender == accHolder, "You are not the account owner.");
    require(msg.value > 0, "Deposit amount should be greater than 0.");

    // Increase the balance by the deposited amount
    balance += msg.value;
}

function showBalance() public view returns (uint256) {
    require(msg.sender == accHolder, "You are not the account owner.");
    return balance;
```

```
    }
}
```

*Code-4(Student Sol/remix ide)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract StudentData {
    // Owner of the contract
    address public owner;
    // Structure to store student details
    struct Student {
        string name;
        uint age;
        string course;
        uint marks;
        bool isExist;
    }
    // Dynamic array of students
    Student[] public students;
    // Event to log the addition of a new student
    event StudentAdded(string name, uint age, string course, uint marks);
    // Modifier to allow only the contract owner to add students
    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can perform this action.");
        _;
    }
    // Constructor to initialize the owner of the contract
    constructor() {
```

```
        owner = msg.sender;
    }
    function addStudent(string memory _name, uint _age, string memory _course, uint _marks) public
onlyOwner {
        // Add the new student to the array
        students.push(Student(_name, _age, _course, _marks, true));
        emit StudentAdded(_name, _age, _course, _marks);
    }
    function getStudent(uint index) public view returns (string memory name, uint age, string memory
course, uint marks) {
        require(index < students.length, "Student does not exist.");
        Student storage student = students[index];
        return (student.name, student.age, student.course, student.marks);
    }
    receive() external payable {
        // Logic for receiving Ether can be added if needed
    }
    fallback() external payable {
        // Fallback logic can be added if needed
    }
    function getContractBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```