ASSIGNMENT 1

#include <bits/stdc++.h>

using namespace std;

class Fibonacci

{

public:

int n;

Fibonacci(int n)

{

this->n = n;

}

int recursive(int i)

{

if (i <= 1)

{

return i;

}

return recursive(i - 1) + recursive(i - 2);

}

void iterative()

{

int t1 = 0;

int t2 = 1;

int nextTerm = 0;

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

{

if (i == 1)

{

cout << t1 << " ";

continue;

}

if (i == 2)

{

cout << t2 << " ";

continue;

}

nextTerm = t1 + t2;

t1 = t2;

t2 = nextTerm;

cout << nextTerm << " ";

}

cout << endl;

}

};

int main()

{

int n;

cout << "Enter N:" << endl;

cin >> n;

Fibonacci fb(n);

cout << "Using Iteration fibonacci series: " << endl;

fb.iterative();

cout << "Using Recursion fibonacci series: " << endl;

int i = 0;

while (i < n)

{

cout << fb.recursive(i) << " ";

i++;

}

cout << endl;

return 0;

}

ASSIGNMENT 2

#include <bits/stdc++.h>

**using** **namespace** std;

// A Huffman tree node

**struct** MinHeapNode {

    // One of the input characters

**char** data;

    // Frequency of the character

    unsigned freq;

    // Left and right child

    MinHeapNode \*left, \*right;

    MinHeapNode(**char** data, unsigned freq)

    {

        left = right = NULL;

**this**->data = data;

**this**->freq = freq;

    }

};

// For comparison of

// two heap nodes (needed in min heap)

**struct** compare {

**bool** operator()(MinHeapNode\* l, MinHeapNode\* r)

    {

**return** (l->freq > r->freq);

    }

};

// Prints huffman codes from

// the root of Huffman Tree.

**void** printCodes(**struct** MinHeapNode\* root, string str)

{

**if** (!root)

**return**;

**if** (root->data != '$')

        cout << root->data << ": " << str << "\n";

    printCodes(root->left, str + "0");

    printCodes(root->right, str + "1");

}

// The main function that builds a Huffman Tree and

// print codes by traversing the built Huffman Tree

**void** HuffmanCodes(**char** data[], **int** freq[], **int** size)

{

**struct** MinHeapNode \*left, \*right, \*top;

    // Create a min heap & inserts all characters of data[]

    priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>,

                   compare>

        minHeap;

**for** (**int** i = 0; i < size; ++i)

        minHeap.push(**new** MinHeapNode(data[i], freq[i]));

    // Iterate while size of heap doesn't become 1

**while** (minHeap.size() != 1) {

        // Extract the two minimum

        // freq items from min heap

        left = minHeap.top();

        minHeap.pop();

        right = minHeap.top();

        minHeap.pop();

        // Create a new internal node with

        // frequency equal to the sum of the

        // two nodes frequencies. Make the

        // two extracted node as left and right children

        // of this new node. Add this node

        // to the min heap '$' is a special value

        // for internal nodes, not used

        top = **new** MinHeapNode('$',

                              left->freq + right->freq);

        top->left = left;

        top->right = right;

        minHeap.push(top);

    }

    // Print Huffman codes using

    // the Huffman tree built above

    printCodes(minHeap.top(), "");

}

// Driver Code

**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]);

    HuffmanCodes(arr, freq, size);

**return** 0;

}

ASSIGNMENT 3

def solve\_knapsack():

# Taking input for value array

val = list(map(int, input("Enter values separated by space:").split()))

# Taking input for weight array

wt = list(map(int, input("Enter weights separated by space: ").split()))

# Taking input for W (total knapsack capacity)

W = int(input("Enter total knapsack capacity: "))

n = len(val)

def knapsack(W, wt, val, n):

# base case

if n == 0 or W == 0:

return 0

# Higher weight than available

if wt[n-1] > W:

return knapsack(W,wt,val, n - 1)

else:

return max(val[n-1] + knapsack(W - wt[n-1],wt,val, n - 1), knapsack(W,wt,val, n - 1))

# max(including , not including)

print(knapsack(W, wt, val, n))

solve\_knapsack()

ASSIGNMENT 4

#include <bits/stdc++.h>

using namespace std;

#define N 5

int ld[30] = { 0 };

int rd[30] = { 0 };

int cl[30] = { 0 };

void printSolution(int board[N][N])

{

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

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

cout << " " << (board[i][j]==1?"Q":".") << " ";

cout << endl;

}

}

bool solveNQUtil(int board[N][N], int col)

{

if (col >= N)

return true;

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

if ((ld[i - col + N - 1] != 1 && rd[i + col] != 1)

&& cl[i] != 1) {

board[i][col] = 1;

ld[i - col + N - 1] = rd[i + col] = cl[i] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0; // BACKTRACK

ld[i - col + N - 1] = rd[i + col] = cl[i] = 0;

}

}

return false;

}

bool solveNQ()

{

int board[N][N] = { 0, 0 };

if (solveNQUtil(board, 0) == false) {

cout << "Solution does not exist";

return false;

}

printSolution(board);

return true;

}

int main()

{

solveNQ();

return 0;

}

ASSIGNMENT 5

def partition(array, low, high):

pivot = array[high]

i = low - 1

for j in range(low, high):

if array[j] <= pivot:

i = i + 1

array[i], array[j] = array[j], array[i]

array[i + 1], array[high] = array[high], array[i + 1]

return i + 1

def quicksort(array, low, high):

if low < high:

pi = partition(array, low, high)

quicksort(array, low, pi - 1)

quicksort(array, pi + 1, high)

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

# Taking user input for the array

array = list(map(int, input("Enter space-separated elements of the array: ").split()))

N = len(array)

# Function call

quicksort(array, 0, N - 1)

print('Sorted array:')

for x in array:

print(x, end=" ")

BT ASSN 1

// SPDX-License-Identifier: UNLICENSED

pragma solidity > 0.7.0;

contract Bank{

mapping (address => uint) public user\_account;

mapping (address => bool) public user\_exist;

function create\_account() public payable returns(string memory)

{

require(user\_exist[msg.sender] == false,"Account Already Exist");

user\_account[msg.sender] = msg.value;

user\_exist[msg.sender] = true;

return "Account Created Successfully";

}

function deposit(uint amount) public payable returns(string memory)

{

require(user\_exist[msg.sender]==true,"Account should be created first");

require(amount >0 ,"Amount should be greater than zero");

user\_account[msg.sender] += amount;

return "Money Deposited Successfully";

}

function withdraw(uint amount) public payable returns(string memory)

{

require(user\_exist[msg.sender] == true,"Account should be created first");

require(amount > 0,"Amount should be greater than zero");

require(user\_account[msg.sender] >= amount,"Ammount entered is greater than amount");

user\_account[msg.sender] -= amount;

return "Money withdrawal done successfully";

}

function account\_balance() public view returns( uint){

return user\_account[msg.sender];

}

function account\_exist() public view returns(bool)

{

return user\_exist[msg.sender];

}

}

BT ASSN 2

// SPDX-License-Identifier: UNLICENSED

pragma solidity >= 0.7.0;

contract Student\_management{

struct Student{

int stud\_id;

string Name;

string Department;

}

Student[] Students;

function add\_stud(int stud\_id, string memory Name, string memory Department) public{

Student memory stud = Student(stud\_id, Name, Department);

Students.push(stud);

}

function getStudent(int stud\_id) public view returns(string memory, string memory){

for(uint i = 0; i < Students.length; i++){

Student memory stud = Students[i];

if(stud.stud\_id == stud\_id){

return(stud.Name, stud.Department);

}

}

return("Name Not Found", "Department Not Found");

}

fallback() external payable {

// Add a default student when Ether is sent to the contract without a function call

Students.push(Student(999, "Default Student", "Default Department"));

emit EtherReceived(msg.sender, msg.value);

}

receive() external payable {

emit EtherReceived(msg.sender, msg.value);

}

// Event to log Ether received

event EtherReceived(address indexed sender, uint256 amount);

}