Task (2)

Description: Restricted Tower of Hanoi Consider the version of the Tower of Hanoi puzzle in which n disks have to be moved from peg A to peg C using peg B so that any move should either place a disk on peg B or move a disk from that peg.

Tower of Hanoi

```
#include <iostream>

#include <cmath>

using namespace std;

void Restricted_Hanoi(int n, char A, char B, char C) {

    if (n == 0) return;

    Restricted_Hanoi(n - 1, A, B, C);

    cout << "Move disk " << n << " from " << A << " to " << B << endl;

    Restricted_Hanoi(n - 1, C, B, A);

    cout << "Move disk " << n << " from " << B << " to " << C << endl;

    Restricted_Hanoi(n - 1, A, B, C);

}

int main () {

    int n;

   cout << "Enter number of disks: ";

    cin >> n;

    cout << "\nSequence of moves:\n";

    Restricted_Hanoi(n, 'A', 'B', 'C');

    int total_moves = 3 * pow(2, n - 1) - 2;

    cout << "\nTotal moves (Restricted): " << total_moves << endl;

    return 0;

}
```

Complexity Analysis:

The Restricted Tower of Hanoi is a special version of the classic Tower of Hanoi puzzle where every move must pass through the middle peg (peg B). This means disks can only be moved between A ↔ B or B ↔ C, and never directly from A ↔ C.

The algorithm follows a divide-and-conquer principle:

- Move n−1 disks from A to C (using B as intermediate).
- Move the largest disk from A → B.
- Move n−1 disks from C → A (using B as intermediate).
- Move the largest disk from B → C.
- Move n−1 disks again from A → C (using B as intermediate).

Why recursion:

Recursion simplifies the process by reducing the problem into smaller subproblems of the same type until the base case n = 0 is reached. Each recursive step mirrors a smaller restricted puzzle.

How codes work:

- The Restricted_Hanoi() function performs recursive calls.
- Each cout statement displays a single disk move that follows the restricted rule.
- Main () reads n, calls the function, and prints the total number of moves calculated by 3×2^(n−1) − 2.

Algorithm Analysis:

Time and Space Complexity (for the Restricted Tower of Hanoi):

Type	Meaning	Explanation
Time Complexity	$O(2^n)$	The total number of moves grows exponentially with the number of disks
Space Complexity	O(n)	Each recursive call must wait for the next one to finish, creating a call stack of depth n. So, the amount of memory used increases linearly with the number of disks.

Comparison with Standard Tower of Hanoi:

Version	**Rule**	**Formula for Moves**	**Example (n = 3)**	**Time Complexity**
Standard	Move between any pegs	$2^n - 1$	7 moves	$O(2^n)$
Restricted	Must go through peg B	$3 \times 2^{n-1} - 2$	10moves	$O(2^n)$

Iterative vs Recursive Approach

- **Recursive**:
  - Uses natural function calls.
  - Easier to understand and implement.
  - But uses more memory
- **Iterative**
  - Uses loops or a manual stack to simulate recursion.
  - Avoids recursion overhead and can be faster in practice.
  - However, it's more complex to write and less intuitive.

Number of Moves:

The minimum number of moves in the restricted Tower of Hanoi is given by:

$$M(n)=3\times 2^{(2-1)}-2$$

n	**Moves**
1	1
2	4
3	10
4	22

Bonus: Tower of Hanoi Visualization (SFML)
Overview: The following program visually demonstrates Tower of Hanoi using the SFML/Graphics library. Each disk movement is animated to help visualize recursion.

Explanation:

The SFML visualization graphically represents the recursive steps of the Tower of Hanoi. Each disk movement is drawn and animated to help visualize how recursion solves the problem step by step.

Tower_of_Hanoi_GUI.cpp

```
#include <SFML/Graphics.hpp>
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>
#include <chrono>
using namespace std;
using namespace sf;

const int MOVE_DELAY = 500;
RenderWindow window(VideoMode(900, 600), "Tower of Hanoi -
Visualization");

struct Disk { RectangleShape shape; int peg; };

void drawScene(vector<vector<Disk>>& pegs) {
    window.clear(Color(30, 30, 30));
    for (int i = 0; i < 3; i++) {
        RectangleShape peg(Vector2f(10, 300));
        peg.setFillColor(Color(200, 200, 200));
        peg.setPosition(200 + i * 250, 250);
        window.draw(peg);
    }
    for (int i = 0; i < 3; i++) {
        int height = 0;
        for (auto& disk : pegs[i]) {
            disk.shape.setPosition(150 + i * 250 +
                (50 - disk.shape.getSize().x / 2), 530 - height * 25);
            window.draw(disk.shape);
            height++;
        }
    }
    window.display();
    this_thread::sleep_for(chrono::milliseconds(MOVE_DELAY));
}

void Tower_of_Hanoi(int n, int src, int to, int aux,
vector<vector<Disk>>& pegs) {
    if (n == 1) {
        Disk disk = pegs[src].back();
        pegs[src].pop_back();
        pegs[to].push_back(disk);
        drawScene(pegs);
        return;
    }
    Tower_of_Hanoi(n - 1, src, aux, to, pegs);
    Disk disk = pegs[src].back();
    pegs[src].pop_back();
    pegs[to].push_back(disk);
    drawScene(pegs);
    Tower_of_Hanoi(n - 1, aux, to, src, pegs);
}

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

    vector<vector<Disk>> pegs(3);
    for (int i = n; i >= 1; i--) {
        Disk d;
        d.shape.setSize(Vector2f(30 + i * 20, 20));
        d.shape.setFillColor(Color(100 + i * 20, 50 + i * 30, 200 - i *
15));
        d.peg = 0;
        pegs[0].push_back(d);
    }

    drawScene(pegs);
    this_thread::sleep_for(chrono::seconds(1));
    Tower_of_Hanoi(n, 0, 2, 1, pegs);

    while (window.isOpen()) {
        Event event;
        while (window.pollEvent(event)) {
            if (event.type == Event::Closed)
                window.close();
        }
    }

}
```