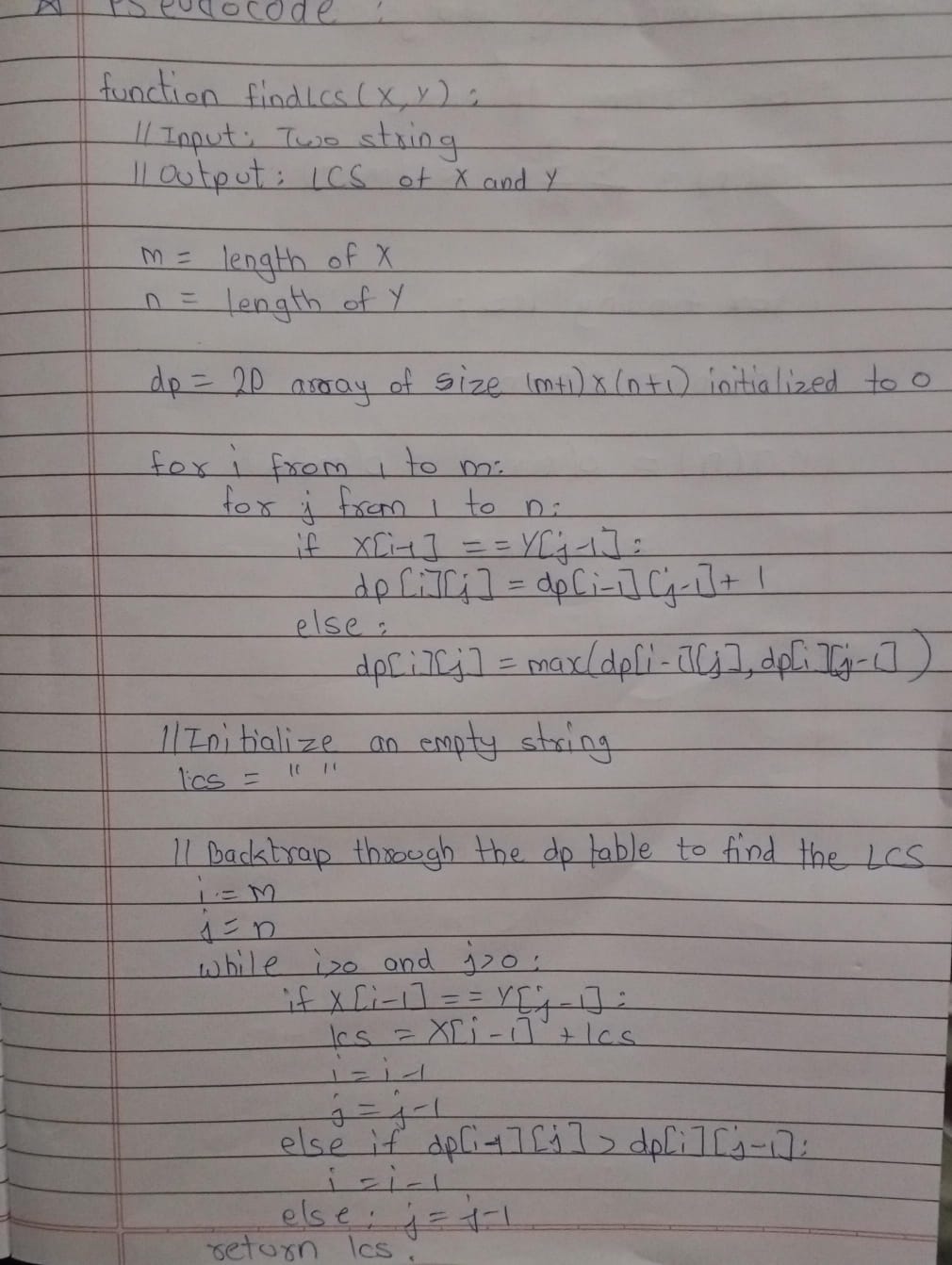
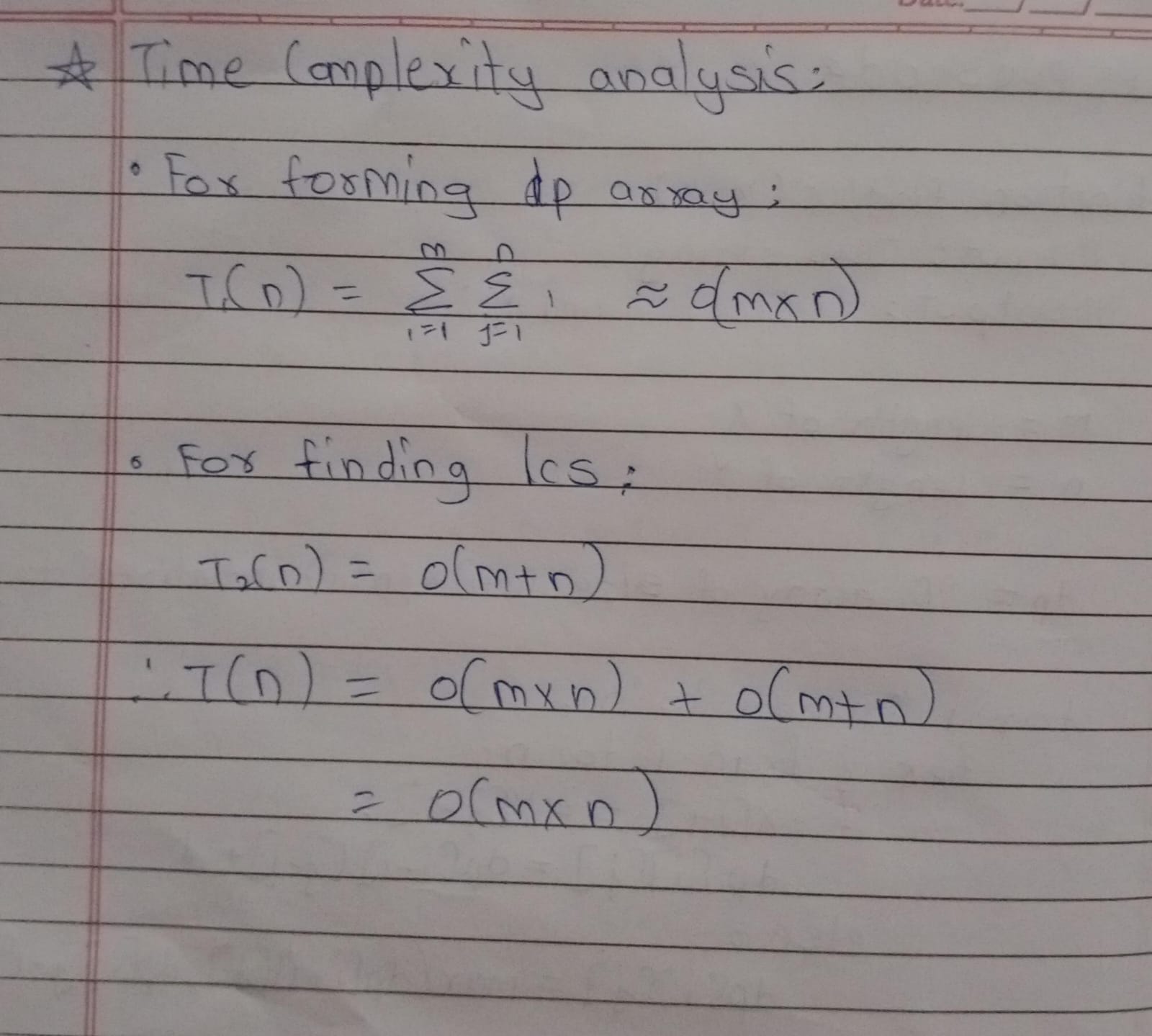
DAA LAB -06

**Task 1:**

**AIM:** Consider grades received by 20 students, like AA, AB, BB, ..., FF of each student. Computer the Longest common sequence of grades among students.

**THEORY:**

****

****

**CODE:**

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

string findLCS(const string &X, const string &Y) {

int m = X.size(), n = Y.size();

vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (X[i - 1] == Y[j - 1])

dp[i][j] = dp[i - 1][j - 1] + 1;

else

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

}

}

string lcs = "";

int i = m, j = n;

while (i > 0 && j > 0) {

if (X[i - 1] == Y[j - 1]) {

lcs = X[i - 1] + lcs;

i--;

j--;

} else if (dp[i - 1][j] > dp[i][j - 1]) {

i--;

} else {

j--;

}

}

return lcs;

}

string longestCommonSubsequence(const vector<string> &grades)

{

if (grades.empty())

return "";

string lcs = grades[0];

for (int i = 1; i < grades.size(); i++)

{

lcs = findLCS(lcs, grades[i]);

}

return lcs;

}

int main()

{

vector<string> grades; // will contain array of strings of grade

string result = longestCommonSubsequence(grades);

cout << "Longest Common Subsequence among students' grades: " << result << endl;

return 0;

}

**OUTPUT:**

****

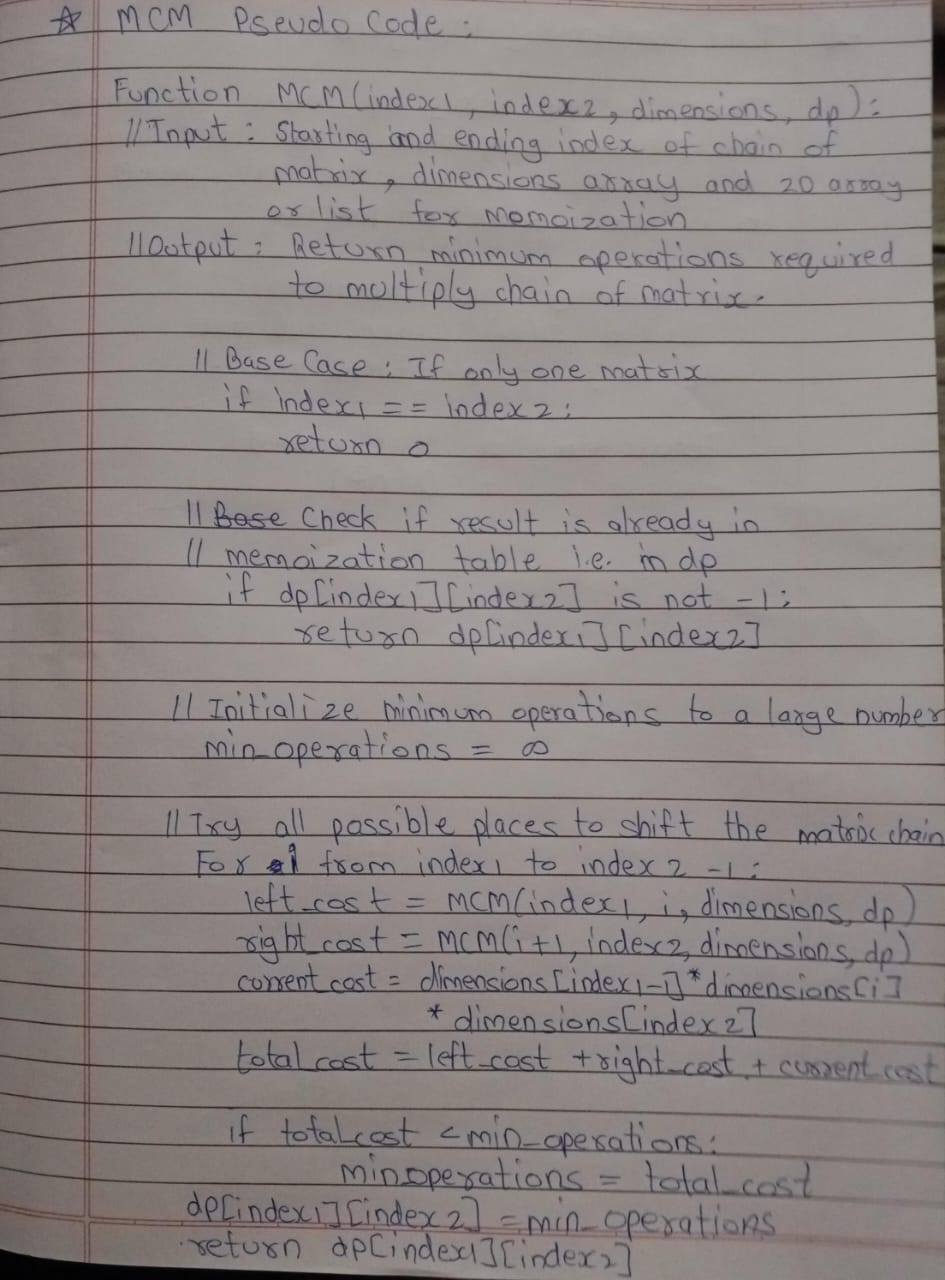
**Conclusion:**

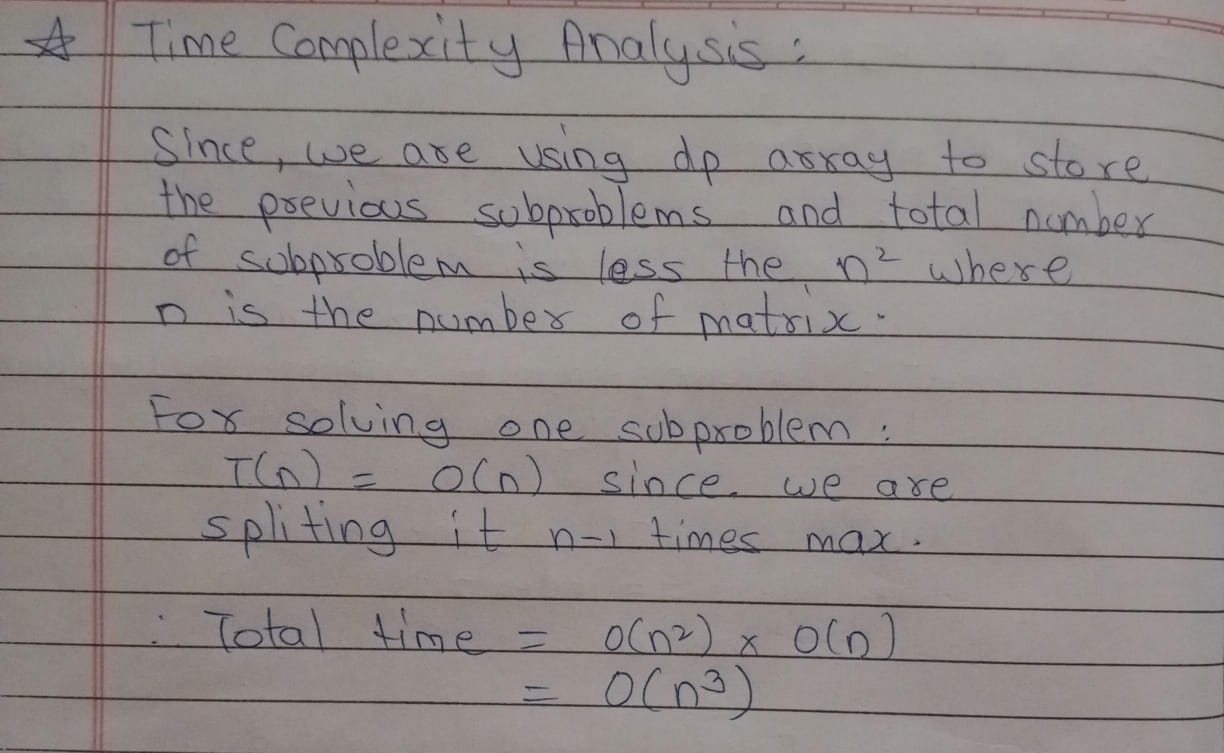
The findLCS function efficiently computes the longest common subsequence (LCS) of two strings using dynamic programming, with a time complexity of O(m×n) where m and n are the lengths of the input strings. The algorithm ensures optimal results by breaking the problem into smaller subproblems and storing intermediate solutions in a 2D table. Overall, this approach is both time-efficient and accurate for finding the LCS.

**Task 2**

**AIM:** Consider meteorological data like temperature, dew point, wind direction, wind speed, cloud cover, cloud layer(s) for each city. This data is available in a two dimensional array for a week. Assuming all tables are compatible for multiplication. You have to implement the matrix chain multiplication algorithm to find the fastest way to complete the matrix multiplication to achieve timely prediction.

**THEORY:**

****

****

**CODE:**

**#include <iostream>**

**#include <vector>**

**#include <climits>**

**using namespace std;**

**int MCM(int index1, int index2, vector<int> &dim, vector<vector<int>> &dp)**

**{**

**if (index1 == index2)**

**return 0;**

**if (dp[index1][index2] != -1)**

**return dp[index1][index2];**

**int min\_oper = INT\_MAX;**

**for (int i = index1; i < index2; i++)**

**{**

**int left\_oper = MCM(index1, i, dim, dp);**

**int right\_oper = MCM(i + 1, index2, dim, dp);**

**int curr\_oper = dim[index1 - 1] \* dim[i] \* dim[index2];**

**int total\_oper = left\_oper + right\_oper + curr\_oper;**

**min\_oper = min(min\_oper, total\_oper);**

**}**

**return dp[index1][index2] = min\_oper;**

**}**

**int main()**

**{**

**int n = 8;**

**vector<int> mat\_dim(n);**

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

**{**

**cin >> mat\_dim[i];**

**}**

**vector<vector<int>> dp(n, vector<int>(n, -1));**

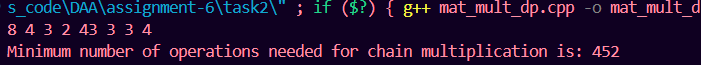
**cout << "Minimum number of operations needed for chain multiplication is: " << MCM(1, n-1, mat\_dim, dp);**

**return 0;**

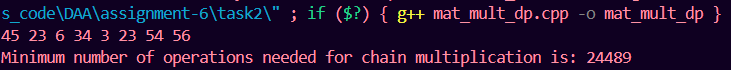
**}**

**OUTPUT:**

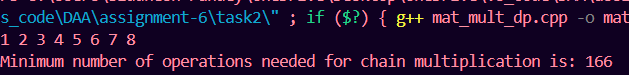
Test Case 1 :

****

Test Case 2 :

****

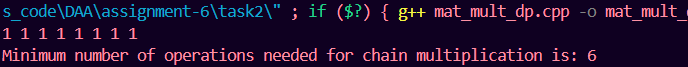
Test Case 3 :

****

Test Case 4:

****

Test Case 5 :

****

**CONCLUSION:**

The conclusion is that using matrix chain multiplication optimizes the computation time for processing meteorological data, making weather predictions faster and more efficient. This method helps handle large data sets quickly, which is crucial for timely and accurate forecasts.

**Task -1 : Read and understand SOLID principles of software development.**

**Task -1 :** Read and understand SOLID principles of software development. Write a sample code for each principle.

**SOLID PRINCIPLES:**

**1. Single Responsibility Principle (SRP):**

A class should have only one responsibility.

Before SRP:

class Employee {

public:

void calculateTax() { /\* calculate tax \*/ }

void saveEmployeeData() { /\* save data \*/ }

};

After SRP:

class Employee {

public:

void save() { /\* save employee data \*/ }

};

class TaxCalculator {

public:

void calculate(double salary) { /\* calculate tax \*/ }

};

Employee handles both data and tax calculation. After applying SRP, we separate concerns into different classes.

**2. Open/Closed Principle (OCP):**

Classes should be open for extension, but closed for modification.

Before OCP:

class Rectangle {

public:

double width, height;

double area() { return width \* height; }

};

class AreaCalculator {

public:

double calculate(Rectangle& r) { return r.area(); }

};

After OCP:

class Shape {

public:

virtual double area() = 0;

};

class Rectangle : public Shape {

public:

double width, height;

double area() override { return width \* height; }

};

class Circle : public Shape {

public:

double radius;

double area() override { return 3.14 \* radius \* radius; }

};

By using polymorphism, new shapes can be added without modifying existing code.

**3. Liskov Substitution Principle (LSP):**

Subtypes must be substitutable for their base types without altering the correctness of the program.

Before LSP:

class Bird {

public:

virtual void fly() { /\* fly \*/ }

};

class Ostrich : public Bird {

public:

void fly() override { throw "Cannot fly"; }

};

After LSP:

class Bird {

public:

virtual void move() = 0;

};

class Sparrow : public Bird {

public:

void move() override { /\* fly \*/ }

};

class Ostrich : public Bird {

public:

void move() override { /\* run \*/ }

};

Subclasses now behave as expected without breaking the base class's behavior.

**4. Interface Segregation Principle (ISP):**

Clients should not be forced to depend on interfaces they do not use.

Before ISP:

class Worker {

public:

virtual void work() = 0;

virtual void eat() = 0;

};

After ISP:

class Workable {

public:

virtual void work() = 0;

};

class Eatable {

public:

virtual void eat() = 0;

};

class Worker : public Workable, public Eatable {

public:

void work() override { /\* work \*/ }

void eat() override { /\* eat \*/ }

};

We separate the work and eat functionalities into different interfaces.

**5. Dependency Inversion Principle (DIP):**

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Before DIP:

class LightBulb {

public:

void turnOn() { /\* turn on \*/ }

};

class Switch {

private:

LightBulb bulb;

public:

void operate() { bulb.turnOn(); }

};

After DIP:

class Switchable {

public:

virtual void turnOn() = 0;

};

class LightBulb : public Switchable {

public:

void turnOn() override { /\* turn on \*/ }

};

class Switch {

private:

Switchable\* device;

public:

Switch(Switchable\* dev) : device(dev) {}

void operate() { device->turnOn(); }

};

The high-level Switch class depends on an abstraction (Switchable), not the concrete LightBulb.