SOLID:

Theory:

SOLID is an acronym for five design principles intended to help software developers design better, more maintainable, and more flexible software. Let's break down each of the five SOLID principles with C++ code examples:

1. Single Responsibility Principle (SRP)

A class should have only one reason to change, i.e., it should only have one job or responsibility.

Example:

```
#include <iostream>
#include <string>

// Class responsible for handling student grades
class Grade {
public:
    std::string grade;
    Grade(std::string g) : grade(g) {}
};

// Class responsible for displaying information (separate responsibility)
class GradeDisplay {
public:
    void displayGrade(const Grade& g) {
        std::cout << "Student grade: " << g.grade << std::endl;
    }
};

int main() {
    Grade studentGrade("AA");
    GradeDisplay display;
    display.displayGrade(studentGrade);
    return 0;
}
```

Explanation: Grade is responsible for the grade data, and GradeDisplay is responsible for displaying the grade. This adheres to SRP, ensuring each class has one reason to change.

2. Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Example:

```
#include <iostream>

// Base class with a virtual method
class Shape {
public:
    virtual double area() const = 0; // pure virtual function
    virtual ~Shape() = default;
};

// Circle class extending Shape
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return 3.14 * radius * radius;
    }
};

// Rectangle class extending Shape
class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() const override {
        return width * height;
    }
};

int main() {
    Circle c(5);
    Rectangle r(3, 4);

    std::cout << "Area of Circle: " << c.area() << std::endl;
    std::cout << "Area of Rectangle: " << r.area() << std::endl;

    return 0;
}
```

Explanation: We can add new shapes (such as Circle, Rectangle, etc.) without modifying the existing Shape class. This adheres to the Open/Closed Principle.

3. Liskov Substitution Principle (LSP)

Objects of a subclass should be able to replace objects of the parent class without altering the correctness of the program.

Example:

```cpp
#include <iostream>

// Base class
class Bird {
public:
    virtual void fly() {
        std::cout << "Flying" << std::endl;
    }
};

// Derived class
class Sparrow : public Bird {
public:
    void fly() override {
        std::cout << "Sparrow flying" << std::endl;
    }
};

// Derived class
class Penguin : public Bird {
public:
    void fly() override {
        // Penguins can't fly, so we might break LSP
        std::cout << "Penguin can't fly" << std::endl;
    }
};

int main() {
    Bird* b1 = new Sparrow();
    b1->fly();  // Sparrow flying

    Bird* b2 = new Penguin();
    b2->fly();  // Penguin can't fly

    delete b1;
    delete b2;

    return 0;
}
```

Explanation: The Penguin class breaks the Liskov Substitution Principle because it doesn't adhere to the behavior expected of Bird. To fix this, we can redesign the system by introducing an interface or abstract class to represent the flying behavior.

4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. It encourages the creation of small, focused interfaces.

Example:

```
#include <iostream>

// Interface for flying animals
class IFlyable {
public:
    virtual void fly() = 0;
};

// Interface for swimming animals
class ISwimmable {
public:
    virtual void swim() = 0;
};

// Bird class implementing IFlyable
class Bird : public IFlyable {
public:
    void fly() override {
        std::cout << "Bird is flying" << std::endl;
    }
};

// Fish class implementing ISwimmable
class Fish : public ISwimmable {
public:
    void swim() override {
        std::cout << "Fish is swimming" << std::endl;
    }
};

int main() {
    Bird bird;
    Fish fish;
```

```
    bird.fly();  // Bird is flying
    fish.swim(); // Fish is swimming

    return 0;
}
```

Explanation: Bird only implements IFlyable and Fish only implements ISwimmable, which means they are only dependent on the interfaces they actually need. This adheres to the Interface Segregation Principle.

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

High-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

Example:

```
#include <iostream>

// Abstraction (interface)
class IPrinter {
public:
    virtual void print() = 0;
};

// Low-level module
class LaserPrinter : public IPrinter {
public:
    void print() override {
        std::cout << "Printing with Laser Printer" << std::endl;
    }
};

// High-level module
class Document {
private:
    IPrinter* printer;
public:
    Document(IPrinter* p) : printer(p) {}
    void print() {
        printer->print();
    }
};

int main() {
    LaserPrinter laserPrinter;
    Document doc(&laserPrinter);
```

```
    doc.print();  // Printing with Laser Printer

    return 0;
}
```

Explanation: The Document class depends on the IPrinter abstraction, not on a specific printer implementation. The LaserPrinter is a low-level module that implements the IPrinter interface. This adheres to the Dependency Inversion Principle.

Code :

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

string LCS(string X, string Y) {
    int m = X.size();
    int n = Y.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    X = '0' + X;
    Y = '0' + Y;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i] == Y[j]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    string ans = "";
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (X[i] == Y[j]) {
            ans = string(1, X[i]) + ans;
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    return ans;
}

int main() {
    vector<string> v;

    for (int i = 0; i < 20; i++) {
        string s;
        cin >> s;
        v.push_back(s);
    }
```

```
    string ans = v[0];
    for (int i = 1; i < 20; i++) {
        ans = LCS(ans, v[i]);
    }

    cout << ans << endl;

    return 0;
}
```

Input:

```
aabbccddff
ababccddff
bbaacccddf
abccddffbb
abbcdcdff
ffbbccddab
abffccddbb
ccabbbddff
bbffcdabcc
aaccbbffdd
cdabbccffb
bcabffddcc
ffabccddbb
aabccddffb
bbcdffccaa
abccffbbdd
abbcffddcc
ccabffbbdd
bbaaccddff
abbbccddff
```

Output:

```
cd
```

Longest Common Subsequence

Algorithm:

```
LCS (x, y)
m ← length (X)
n ← length (Y)

// Create a 2D array dp of size (m+1) x (n+1) initialised
// to 0
   dp ← array of size (m+1) x (n+1) filled with 0

// fill the dp tables
  for i from 1 to m:
    for j from 1 to n:
      if X[i-1] == Y[j-1]   // Characters match
          dp[i][j] ← dp[i-1][j-1] + 1
    else:                  // Characters do not match
      dp[i][j] ← max(dp[i-1][j], dp[i][j-1]

 // Backtrack to find the LCS String
   lcs ← empty string
   i ← m, j ← n
   while i > 0 and j > 0:
     if X[i-1] == Y[j-1]:          // Characters match.
       lcs ← X[i-1] + lcs          // Add to result
        i ← i-1
        j ← j-1
```

else if dp[i-1][j] >dp [i] [j-1] : // move up
i <- j-1
else : // move left
j <- j-1

return lcs

int main ():

// input Vector of String for 20 Students

Vector < String > V (20)

String ans = " ";
ans = Lcs (V[0] , V[1])

for (i <- 2 to i< 20 , i++):
ans = ~~Lcs~~ lcs (V[i], ans);

Cout << ans ;

return 0

Time Complexity :
function Lcs :

$O(n^2)$ for dp table formation
$O(n^2)$ for back tracking answer string

Total : $2 \times O(n^2)$. $\leftarrow$ for one String.

lcs for 20 strings = 20 x function lcs

$= 20 \times 2 \times O(n^2)$
$= 40\,O(n^2) \approx O(n^2)$

DAA Lab 6

Code for Matrix Chain Multiplication :

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

int matrixMultiplication(vector<int> &arr)
{

    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int len = 2; len < n; len++)
    {
        for (int i = 0; i < n - len; i++)
        {
            int j = i + len;
            dp[i][j] = INT_MAX;

            for (int k = i + 1; k < j; k++)
            {
                int cost = dp[i][k] + dp[k][j] + arr[i] * arr[k] * arr[j];
                dp[i][j] = min(dp[i][j], cost);
            }
        }
    }

    return dp[0][n - 1];
}

int main()
{

    vector<int> arr = {2,3,4,5};
    cout<<"The Minimum Cost for the multiplication of these matrix are : ";
    cout << matrixMultiplication(arr);
    return 0;
}
```

OutPut:

```
The Minimum Cost for the multiplication of the matrix {2,3,4,5} is  : 64
ashutosh kumar@Ashutosh-PC MINGW64 /e/programs/code forces/output
$
```

Conclusion of Matrix Chain Multiplication Experiment:

The Matrix Chain Multiplication experiment highlights the efficiency of dynamic programming in determining the optimal order of matrix multiplication, significantly reducing scalar multiplications compared to naive recursive methods. With a time complexity of O(n^3) and space complexity of O(n^2), it demonstrates scalability for moderate-sized problems while avoiding redundant computations through memoization. The study emphasizes the importance of optimal substructure and overlapping subproblems, with applications in areas like computer graphics and scientific computing, though handling extremely large chains may require further optimizations.

Matrix Chain Multiplication :

Algorithm :

\# input: Dimension of array
\# output: Minimum Cost for matrix multiplication

```
MatrixMultiplication (Vector <int> arr) {
  n ← arr.size() ;
  dp [n×n] = {0}     // initialize DP

  for (int i = 2 → i=n-1) {
     for (int j = 0 → j = n-i-1) {
        int k = i+j ;
        dp [j][k] = INT_Max ;
        for (int x = j+1 → x = k-1) {
           Cost = dp[j][x] + dp[x][k]
                + arr [j] * arr [x] * arr [k]
        dp [j][k] = Min( dp[j][k] , Cost );
     }
  }
}
 return dp [0][n-1];
```

Time Complexity

1) Outer loop : runs from 2 to n-1 - it has run for n-2 time.

2) the Center or the Second loop has run from also aproximatly n times.

3) the innermost loop has run also for n times

So the time complexity is $O((n-2) n \times n)$

$$(n-2)n^2 = n^3 - 2n^2$$

$$O(n^3 - 2n^2) \simeq O(n^3).$$

Space Complexity

1) DP table:
It requires $O(n^2)$ Space.

2) Other Variables
Variables like Cost, i, j, k, etc. takes $O(1)$ Space

Thus the Space Complexity for the algorith is

$O(n^2)$