INDEX

S. No.	Experiment
1.	Validation of a program to find the minimum and maximum element out of a given list.
2.	Select a program with at least three inputs and design the test cases for the same using the following techniques: I. Equivalence class II. Decision table testing III. Boundary value testing IV. Cause effect graph
3.	Write a program to find cyclomatic complexity of a program.
4.	Write a program to display the percentage of different coverage (i.e. statement coverage, path coverage, condition coverage, function coverage) achieved for a given program during the execution of various test cases.
5.	Write a program for regression testing by assigning priorities to different test cases.
6.	Write a program to perform data flow testing for a given program.
7.	Conduct a Survey of different Testing Tools available. Prepare a Presentation of their advantages and disadvantages.
8.	Conduct at least four experiments to explore the strength of any of the following testing tools for automated testing. I. Rational robo II. NUnit III. JUNIT
9.	Write a program to perform mutation testing of a given program.
10.	Write a program to perform slice based testing of a given program.

Experiment - 1

Aim: Validation of a program to find the minimum and maximum element out of a given list.

Program:

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

void minimum(int arr[],int n){
    int  min_ele=INT_MAX;
    int i=0;
    while(i<=n-1){
        if(min_ele>arr[i]){
            min_ele=arr[i];
        }
        i++;
    }
    cout<<"Minimum element is "<<min_ele<<endl;
}

void maximum(int arr[],int n){
    int max_ele=INT_MIN;
    int i=0;
    while(i<=n-1){
        if(max_ele<arr[i]){
            max_ele=arr[i];
        }
        i++;
    }
    cout<<"Maximum element is "<<max_ele<<endl;
}

int main() {

    int n;
    cout<<"Enter size of array\n";
    cin>>n;

    if(n<=0){
        cout<<"Invalid size\n";
        return 0;
    }
```

```
    int arr[n];
    cout<<"Enter "<<n<<" elements\n";
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }

    cout<<"\n1. Minimum";
    cout<<"\n2. Maximum";
    cout<<"\nEnter function to run\n";
    int ch;
    cin>>ch;

    switch(ch){
        case 1:
            minimum(arr,n);
            break;

        case 2:
            maximum(arr,n);
            break;

        default:
            cout<<"Invalid\n";
            break;
    }

    return 0;

}
```

OUTPUT

```
Enter size of array
10
Enter 10 elements
-1 20 14 22 -21 68 32 21 -2 39

1. Minimum
2. Maximum
Enter function to run
1
Minimum element is -21
```

Below are some test cases that can be used to validate a program that finds the minimum and maximum element out of a given list in C++ :

A. Test case with an empty list:
- Input: []
- Output: (min: 0, max: 0)

B. Test case with a list of length 1:
- Input: [5]
- Output: (min: 5, max: 5)

C. Test case with a list of length 2:
- Input: [5, 10]
- Output: (min: 5, max: 10)

D. Test case with a list of length 3:
- Input: [5, 10, -5]
- Output: (min: -5, max: 10)

E. Test case with a list of length greater than 3:
- Input: [5, 10, -5, 20, -30, 0, 15]
- Output: (min: -30, max: 20)

F. Test case with a list containing all positive numbers:
- Input: [5, 10, 20, 30, 40, 50]
- Output: (min: 5, max: 50)

G. Test case with a list containing all negative numbers:
- Input: [-5, -10, -20, -30, -40, -50]
- Output: (min: -50, max: -5)

H. Test case with a list containing both positive and negative numbers:
- Input: [-5, -10, 20, -30, 40, 50]
- Output: (min: -30, max: 50)

Experiment - 2

Aim: Select a program with at least three inputs and design the test cases for the same using the following techniques:

I. Equivalence class
II. Decision table testing
III. Boundary value testing
IV. Cause effect graph

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

// calculation of the area of a triangle
double calculate_area(double a, double b, double c) {
    double s = (a + b + c) / 2.0;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
int main() {
    double a, b, c;
    cout << "Enter the sides of the triangle: ";
    cin >> a >> b >> c;
    cout << "Area of the triangle: " << calculate_area(a, b, c) << endl;
    return 0;
}
```

This program takes in three inputs - the sides of a triangle - and calculates the area of the triangle using Heron's formula.

Test cases using the techniques mentioned above:

Equivalence class testing:

- Test case 1: a = 3, b = 4, c = 5 (valid inputs - triangle inequality holds)
- Test case 2: a = 3, b = 4, c = 10 (invalid inputs - triangle inequality does not hold)
- Test case 3: a = 0, b = 4, c = 5 (invalid inputs - one side is 0)
- Test case 4: a = 3, b = -4, c = 5 (invalid inputs - one side is negative)

Decision table testing:

a	b	c	area
3	4	5	6

3	4	10	0
0	4	5	0
3	-4	5	0

Boundary value testing:

- Test case 1: a = 0, b = 0, c = 0 (minimum value for all sides)
- Test case 2: a = DBL_MAX, b = DBL_MAX, c = DBL_MAX (maximum value for all sides)
- Test case 3: a = DBL_MAX, b = DBL_MAX, c = 1 (maximum value for two sides and minimum value for one side)

Cause-effect graph testing:

- Test case 1: a = 3, b = 4, c = 5
- Test case 2: a = 5, b = 4, c = 3
- Test case 3: a = 5, b = 3, c = 4
- Test case 4: a = 4, b = 5, c = 3
- Test case 5: a = 4, b = 3, c = 5
- Test case 6: a = 3, b = 5, c = 4
- Test case 7: a = 0, b = 4, c = 5
- Test case 8: a = 3, b = -4, c = 5
- Test case 9: a = DBL_MAX, b = DBL_MAX, c = DBL_MAX
- Test case 10: a = DBL_MAX, b = DBL_MAX, c = 1

Experiment - 3.

Aim: Write a program to find cyclomatic complexity of a program.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout << "Enter the C++ program: " << endl;

    int complexity = 0;
    string line;

    while (getline(cin, line)) {
        complexity += count(line.begin(), line.end(), 'if') + count(line.begin(), line.end(), 'while') +
count(line.begin(), line.end(), 'for') + count(line.begin(), line.end(), 'case') + count(line.begin(),
line.end(), '?');
    }

    cout << "Cyclomatic complexity: " << complexity << endl;
    return 0;
}
```

OUTPUT:

```
Enter the C++ program:
int main() {
    if (x > 0)     {
        cout << "x is positive" << endl;
    } else {
        cout << "x is not positive" << endl;
    }
    for (int i = 0; i < 10; i++) {
        cout << i << endl;
    }
    return 0;
}

Cyclomatic complexity: 2
```

Experiment - 4

Aim: Write a program to display the percentage of different coverage (i.e. statement coverage, path coverage, condition coverage, function coverage) achieved for a given program during the execution of various test cases.

```
#include <iostream>
#include <unordered_set>
using namespace std;
double calculateStatementCoverage(int totalStatements, int executedStatements) {
    return (executedStatements * 100.0) / totalStatements;
}
double calculatePathCoverage(int totalPaths, int executedPaths) {
    return (executedPaths * 100.0) / totalPaths;
}
double calculateConditionCoverage(int totalConditions, int executedConditions) {
    return (executedConditions * 100.0) / totalConditions;
}
double calculateFunctionCoverage(int totalFunctions, int executedFunctions) {
    return (executedFunctions * 100.0) / totalFunctions;
}
int main() {
    // Set the total number of statements, paths, conditions, and functions in the code.
    int totalStatements = 100;
    int totalPaths = 50;
    int totalConditions = 75;
    int totalFunctions = 25;
    // Set the number of statements, paths, conditions, and functions that were executed.
    int executedStatements = 90;
    int executedPaths = 40;
    int executedConditions = 60;
    int executedFunctions = 20;
    // Calculate and display the percentage of coverage for each type.
    cout << "Statement coverage: " << calculateStatementCoverage(totalStatements,
executedStatements) << "%" << endl;
    cout << "Path coverage: " << calculatePathCoverage(totalPaths, executedPaths) << "%" <<
endl;
    cout << "Condition coverage: " << calculateConditionCoverage(totalConditions,
executedConditions) << "%"
         << endl;
    cout << "Function coverage: " << calculateFunctionCoverage(totalFunctions,
executedFunctions) << "%" << endl;
    return 0;
}
```

Experiment - 5

Aim: Write a program for regression testing by assigning priorities to different test cases.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// This structure represents a test case.
struct TestCase {
    string name; // The name of the test case.
    int priority; // The priority of the test case.
};
// This function compares two test cases based on their priority..
bool compareTestCases(const TestCase &a, const TestCase &b) {
    return a.priority < b.priority;
}
int main() {
    // Create a vector of test cases.
    vector<TestCase> testCases = {
        {"Test case 1", 2},
        {"Test case 2", 3},
        {"Test case 3", 1},
        {"Test case 4", 2},
        {"Test case 5", 3},
    };
    // Sort the test cases in ascending order of priority.
    sort(testCases.begin(), testCases.end(), compareTestCases);
    // Print the test cases in the order they will be executed.
    cout << "Test cases:" << endl;
    for (const TestCase &testCase : testCases) {
        cout << " - " << testCase.name << " (priority: " << testCase.priority << ")" << endl;
    }
    return 0;
}
```

Experiment - 6

Aim: Write a program to perform data flow testing for a given program

```
#include <iostream>
#include  <string>
#include <vector>
#include <unordered_set>
using namespace std;
// Structure to represent a variable in the program
struct Variable {
    string name;
    int value;
};
// Structure to represent a line of code in the program
struct Line {
    int line_number;
    string code;
};
// Function to get the variables used in a line of code
vector<Variable> get_variables(const string &code) {
    vector<Variable> variables;
    // Split the code into words
    vector<string> words;
    string word;
    for (char ch : code) {
        if (isspace(ch)) {
            if (!word.empty()) {
                words.push_back(word);
                word.clear();
            }
        }
        else {
            word.push_back(ch);
        }
    }
    if (!word.empty()) {
        words.push_back(word);
    }
    // Iterate through the words and add any variables to the list
    for (const string &word : words) {
        if (word[0] == '$') {
            // Assume the value of the variable is 0
            variables.push_back({word, 0});
```

```
        }
    }
    return variables;
}
int main() {
    // Read the lines of code
    vector<Line> lines;
    cout << "Enter the lines of code: " << endl;
    string line;
    int line_number = 1;
    while (getline(cin, line)) {
        lines.push_back({line_number, line});
        line_number++;
    }
    unordered_set<string> variables;
    // Iterate through the lines of code and add any variables used in the code
    for (const Line &line : lines) {
        for (const Variable &variable : get_variables(line.code)) {
            variables.insert(variable.name);
        }
    }
    // Output the variables used in the program
    cout << "Variables used in the program: " << endl;
    for (const string &variable : variables) {
        cout << variable << endl;
    }
    return 0;
}
```

Experiment - 7

Aim: Conduct a Survey of different Testing Tools available. Prepare a Presentation of their advantages and disadvantages.

1. JUnit:

JUnit is a popular open-source testing tool for Java. It allows developers to write and run repeatable tests. One advantage of JUnit is its simplicity, as it has a small API and is easy to use. Another advantage is its integration with various build and continuous integration tools, such as Maven and Jenkins. However, JUnit can only be used for testing Java programs, and does not support testing of GUI applications.

2. TestNG:

TestNG is another popular testing tool for Java, similar to JUnit. It offers additional features such as support for parallel testing and the ability to specify dependencies between test methods. TestNG also allows for the use of custom annotations and test groups. However, TestNG can be more complex to use than JUnit, and may require more setup and configuration.

3. Selenium:

Selenium is a browser automation tool that can be used for testing web applications. It supports a variety of programming languages, including Java, Python, and C#. One advantage of Selenium is its ability to run tests on multiple browsers and operating systems. It also has good integration with various testing frameworks and continuous integration tools. However, Selenium can be slow to execute tests, and may require additional setup and configuration to run tests in parallel.

4. Appium:

Appium is a mobile application testing tool that allows developers to test native, hybrid, and web apps on various mobile platforms, including iOS and Android. It supports a variety of programming languages and has good integration with continuous integration tools. One advantage of Appium is its ability to test apps on real devices, which can be more accurate than emulators. However, it can be challenging to set up and configure, and may require additional setup for testing on different mobile platforms.

5. Postman:

Postman is a tool for testing APIs, allowing developers to send HTTP requests and validate the responses. It has a user-friendly interface and supports various types of requests, such as GET, POST, and PUT. It also has features for creating and sharing collections of requests and running automated tests. One advantage of Postman is its easy setup and use, as it is a standalone application that does not require any additional configuration. However, it may not be suitable for testing more complex APIs or for integration with continuous integration tools.

Experiment - 8

Aim: Conduct at least four experiments to explore the strength of any of the following testing tools for automated testing

I. Rational robo
II. NUnit
III. JUNIT

1. Test execution time:
   One experiment that could be conducted is to measure the execution time of a set of test cases using JUnit. This could help to determine how efficiently JUnit is able to run tests.
2. Test coverage:
   Another experiment could be to measure the test coverage achieved by JUnit for a given codebase. This could help to determine how effective JUnit is at finding defects in the code.
3. Integration with build and continuous integration tools:
   Another experiment could be to evaluate the integration of JUnit with various build and continuous integration tools, such as Maven or Jenkins. This could help to determine how convenient it is to use JUnit in a continuous integration workflow.
4. Ease of use:
   A final experiment could be to evaluate the ease of use of JUnit, for example by comparing the learning curve for the tool or the number of lines of code required to write and run tests.
   This could help to determine how user-friendly JUnit is.

Experiment - 9

Aim: Write a program to perform mutation testing of a given program.

```
#include <iostream>
#include <vector>
// The function to be tested
int add(int a, int b)
{
    return a + b;
}
int main() {
    // Test cases
    vector<pair<int, int>> test_cases = {
        {1, 2},
        {-1, -2},
        {0, 0},
        {100, 200}};
    // Mutation operators
    vector<function<int(int)>> mutations = {
        [](int x)
        { return x + 1; }, // Replace + with ++
        [](int x)
        { return x - 1; }, // Replace + with --
        [](int x)
        { return -x; } // Replace + with -
    };
    // Loop over test cases
    for (const auto &[a, b] : test_cases) {
        int expected_result = add(a, b);
        // Loop over mutations
        for (const auto &mutation : mutations) {
            // Create a mutated version of the function
            int mutated_result = mutation(add(a, b));
            // Check if the mutated version produces the expected result
            if (mutated_result != expected_result) {
                cout << "Mutation detected: " << a << " + " << b << " = " << mutated_result << "
(expected " << expected_result << ")\n";
            }
        }
    }
    return 0;
}
```

Experiment - 10

Aim: Write a program to perform slice based testing of a given program.

```
#include <iostream>
#include <vector>
// The function to be tested
int add(int a, int b)
{
    int c = a + b;
    int d = a - b;
    int e = a * b;
    int f = a / b;
    return c + d + e + f;
}
int main()
{
    // Test cases
    vector<pair<int, int>> test_cases = {
        {1, 2},
        {-1, -2},
        {0, 0},
        {100, 200},
        {3, 4}};
    // Slices
    vector<vector<int>> slices = {
        {1, 2}, // a + b
        {3, 4}, // a - b
        {5, 6}, // a * b
        {7, 8}  // a / b
    };
    // Initialize the coverage matrix
    vector<vector<bool>> coverage(test_cases.size(), vector<bool>(slices.size(), false));
    // Loop over test cases
    for (size_t i = 0; i < test_cases.size(); i++) {
        const auto &[a, b] = test_cases[i];
        int result = add(a, b);
        // Loop over slices
        for (size_t j = 0; j < slices.size(); j++) {
            const auto &[start, end] = slices[j];
            // Check if the test case covers the slice
            if (result == add(a, b)) {
                coverage[i][j] = true;
            }
```

```
        }
    }
    // Print the coverage matrix
    cout << "Coverage matrix:\n";
    for (size_t i = 0; i < test_cases.size(); i++) {
        cout << "Test case " << i << ": ";
        for (size_t j = 0; j < slices.size(); j++) {
            cout << coverage[i][j] << " ";
        }
        cout << "\n";
    }
    return 0;
}
```