ASSIGNMENT NO. 2

TITLE: White Box Testing – Control Flow and Data Flow Testing Techniques

AIM: To design and develop test cases using White Box Testing techniques for the given C program and to perform:

1. Calculation of Cyclomatic Complexity
2. Control Flow Testing
3. Data Flow Testing

OBJECTIVES:

1. To understand the concept of White Box Testing.
2. To construct a Control Flow Graph (CFG).
3. To calculate Cyclomatic Complexity.
4. To identify independent paths.
5. To perform Data Flow Testing using def-use pairs.
6. To design adequate test cases based on structural coverage.

PREREQUISITES:

- Knowledge of C programming
  Understanding of decision statements and loops
- Basic understanding of software testing concepts
- Knowledge of control flow and data flow

THEORY (Detailed – Student Understanding)

1. Introduction to White Box Testing

White Box Testing is a structural testing technique in which I test the internal logic and structure of the program. Unlike Black Box Testing, where only inputs and outputs are tested, White Box Testing involves examining:

- Program structure
- Control flow
- Data flow
- Loops and conditions
- Logical branches

It is also called:

- Structural Testing
- Glass Box Testing
- Clear Box Testing

The primary objective of white box testing is to ensure that:

- All statements are executed at least once
- All branches are tested
- All independent paths are exercised
- Internal logic errors are detected

White box testing requires knowledge of the source code.

2. Goals of White Box Testing

The main goals are:

- To verify internal program correctness
- To ensure all logical paths are tested
- To detect logical errors in conditions and loops
- To measure structural coverage

It focuses on structural elements such as:

- Statements
- Branches
- Loops
- Paths
- Variables

3. Test Adequacy Criteria

Test adequacy criteria help determine whether sufficient testing has been performed.
They help in:

1. Selecting program properties to test
2. Selecting appropriate test data
3. Measuring the degree of coverage
4. Determining when testing can stop

A test set is considered:

- Statement Adequate → if all statements execute
- Branch Adequate → if all branches execute
- Path Adequate → if all independent paths execute

4. Test Coverage Criteria
Test coverage defines the percentage of code covered during testing.
Coverage is usually measured using tools.
Common Coverage Types:

1. Statement Coverage
2. Branch Coverage
3. Path Coverage
4. Condition Coverage
5. Decision Coverage

Achieving 100% coverage may not always be possible due to:
1. Nature of the unit:

- Some branches may not be reachable
- Some code may be defensive or safety-related

2. Resource constraints:

- Time limitations
- Lack of testers
- Tool limitations
- Project deadlines

WHITE BOX TESTING TECHNIQUES

5. Control Flow Testing
Control Flow Testing focuses on:

- The flow of execution in a program
- Testing all possible execution paths

It uses a graphical representation called a **Control Flow Graph (CFG).**

5.1 Control Flow Graph (CFG)
A Control Flow Graph is a graphical representation of program structure.
In CFG:

- Nodes represent statements or blocks
- Edges represent control flow
- Decision nodes represent conditions
- Loops create cycles

The CFG helps:

- Evaluate testability
- Identify independent paths
- Calculate cyclomatic complexity

6. Cyclomatic Complexity
Cyclomatic Complexity is a quantitative measure of program complexity.
It is calculated using:
V(G)=E−N+2V(G) = E - N + 2V(G)=E−N+2
Where:

- E = Number of edges
- N = Number of nodes

It can also be calculated as:
V(G)=Number of Decision Nodes+1V(G) = Number\ of\ Decision\ Nodes + 1V(G)=Number of Decision Nodes+1
Cyclomatic complexity represents:

- Number of independent paths
- Minimum number of test cases required

Higher complexity indicates:

- More risk
- More testing effort required

7. Independent Paths
An independent path is:
A path that introduces at least one new edge not previously included in other paths.
The set of independent paths is called a **Basis Set**.
Testing all independent paths ensures:

- Complete branch coverage
- Strong structural validation

8. Data Flow Testing
Data Flow Testing focuses on:

- Definition of variables
- Use of variables
- Def-use relationships

8.1 Definition (Def)
A variable is defined when its value is assigned or modified.
Example:
Y = 26 * X;
Y is defined.

8.2 Use (Use)
A variable is used when its value is utilized.
Two types:
Predicate Use (p-use)
Used in decision conditions.
Example:
```
if (X > 0)
```
Computational Use (c-use)
Used in calculations.
Example:
```
sum = sum + X;
```

8.3 Def-Use Path
A path from a variable definition to its use without redefinition is called a Def-Use Path.

8.4 Data Flow Coverage Criteria
1. All-Def Coverage
2. All-Uses Coverage
3. All p-uses
4. All c-uses
5. All Def-Use Paths (Strongest Criterion)

All Def-Use Paths ensures:
- Every definition reaches every possible use

9. Loop Testing
Loop testing is a white box technique that tests loop structures.
Testing includes:
1. Zero iterations
2. One iteration
3. Two iterations
4. Typical number of iterations
5. Maximum iterations
6. Nested loops

Loop testing ensures boundary errors are detected.

10. Advantages of White Box Testing

- Ensures structural coverage
- Detects logical errors
- Helps measure complexity
- Improves code quality
- Identifies unreachable code

11. Limitations of White Box Testing

- Requires programming knowledge
- Time-consuming
- Cannot detect missing functionality
  May not detect user-level errors

RESULT:

Control Flow Graph was constructed, Cyclomatic Complexity was calculated, independent paths were identified, and test cases were developed using Control Flow and Data Flow testing techniques.

CONCLUSION:

Through this assignment, I gained practical understanding of White Box Testing techniques. I learned how to construct a Control Flow Graph, calculate cyclomatic complexity, and identify independent paths for structural coverage. I also applied Data Flow Testing by identifying variable definitions and uses and generating test cases based on def-use pairs. These techniques help improve internal code reliability and ensure adequate structural testing of software programs.

C Program for Fractional Knapsack

```
#include <stdio.h>

// Structure for items
struct Item {
    int profit;
    int weight;
    float ratio;
};

// Function to swap items
void swap(struct Item *a, struct Item *b) {
    struct Item temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int n, i, j;
    float capacity, totalProfit = 0.0;

    printf("Enter number of items: ");
    scanf("%d", &n);

    struct Item items[n];

    // Input profit and weight
    for(i = 0; i < n; i++) {
        printf("Enter profit and weight for item %d: ", i+1);
        scanf("%d %d", &items[i].profit, &items[i].weight);
        items[i].ratio = (float)items[i].profit / items[i].weight;
    }

    printf("Enter knapsack capacity: ");
    scanf("%f", &capacity);

    // Sort items based on ratio (descending order)
    for(i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
```

```
            if(items[j].ratio < items[j+1].ratio) {
                swap(&items[j], &items[j+1]);
            }
        }
    }

    // Apply Greedy Method
    for(i = 0; i < n; i++) {
        if(capacity >= items[i].weight) {
            capacity -= items[i].weight;
            totalProfit += items[i].profit;
        } else {
            totalProfit += items[i].ratio * capacity;
            break;
        }
    }

    printf("Maximum Profit = %.2f\n", totalProfit);

    return 0;
}
```