White-Box Testing

Techniques of White-Box Testing

(1) **Statement Coverage**: Every line of code should run atleast once.

(2) **Branch Coverage**: Each if and else branch must be tested.

(3) **Path Coverage**: Every possible execution path is tested.

(4) **Condition Coverage**: Each condition in a decision must be tested for both true and false.

eg.

```
int findmax (int a, int b) {
    if (a>b)
        return a
    else
        return b
}
```

Test case 1: If a > b

Test case 2: If a <= b

True branch: a > b

False branch: a <= b

Path 1: If (a > b) return a

Path 2: else return b

Testing Logic and Functions

Testing Logic : It checks the logical correctness of algorithm

Testing Functionality : Each functions or methods perform its intended operations.

Advantages of white - box testing

(a) helps in understanding internal implementation

(b) programs follow correct path or not is checked

(c) enhances performance of a product

(d) easier to find hidden defects

(e) easy to detect security checks

Challenges

(a) implementation is complex

(b) time consuming

(c) expensive

(d) detailed knowledge of the code

(e) high dependencies

(9) Write a program to find the largest of three nos.

code

```
def largestNum (a, b, c):
    if a > b and a > c:
        return a
    elif b > a and b > c:
        return b
    elif c > a and c > b:
        return c
    else:
        print ("All the nos. are equal")
        return a
```

(d) condition coverage

a>b, a>c, b>a, b>c, c>a, c>b

(a) statement coverage: To cover all statements, we need test cases that execute each branch. eg. (5, 3, 2) → executes return a

(b) Branch coverage: (1) a > b and a > c → True; (2) a > b and a > c → False but b > a and b > c → True; (3) first two false but c > a and c > b → True; (4) All are false (equal nos.)

(c) Path coverage: (1) if (a largest); (2) elif (b largest); (3) elif (c largest); (4) else (all equal)

SOFTWARE TESTING

Visualize Control Flow

Control Flow Graph (CFG)

Start

↓

Decision (condition A)

↙ ↘

Yes | No

↓ ↓

Block A | Block B

↘ ↙

Exit

Tools for code coverage

(a) JaCoCo

(b) Cobertura

(c) gcov

Techniques for testing code logic

(a) Control flow testing

(b) Data flow testing

(c) Decision Table testing

(d) Loop testing

```
void function (num){

    if (num>0)

        cout << "Positive"

    else if (num <0)

        cout << "Negative"

    else

        cout << "zero"

}
```

num >0 → print positive

num <0 → print negative

neither → print zero

Test Case

Test I/P	Expected O/P	Path Count
5	Positive	Path 1
-3	Negative	Path 2
0	zero	Path 3

Data Flow Diagram

```
void function (){

    int total,

    int a =20;

    int b =10;

    total =a+b;

    cout << total>>

}
```

Here,

a and b is declared.

a and b is assigned

total is defined and assigned

Testing techniques for code logic

(a) Control Flow Testing

It checks how the program flows from one statement to another. It uses CFG to visualize all possible execution cycle.

(b) Data Flow Testing

It focuses on how variables are used throughout the program like where they are declared or assigned or where they are used.

(c) Decision Table Testing

It is useful when there are multiple conditions that control the output (especially in business logic).

It lists all possible combinations of input and their expected output in table form.

(d) Loop Testing

It ensures that loops in the program works correctly under all conditions.

code complexity

Cyclomatic Complexity - a metrics that measures no. of independent paths in a program.

$$CC = E - N + 2P$$

E = no. of edges

N = no. of nodes

P = no. of connected components

↳ for single program $P = 1$.

→ 1 + {No. of decision points}

```
int main (){

    int n;

    cin >> n;

    if (n > 0){

        cout << "positive" >>;

    }else (n < 0) {

        cout << "negative">>;

    }else{

        cout << "zero" >>;

    }

    return 0;

}
```

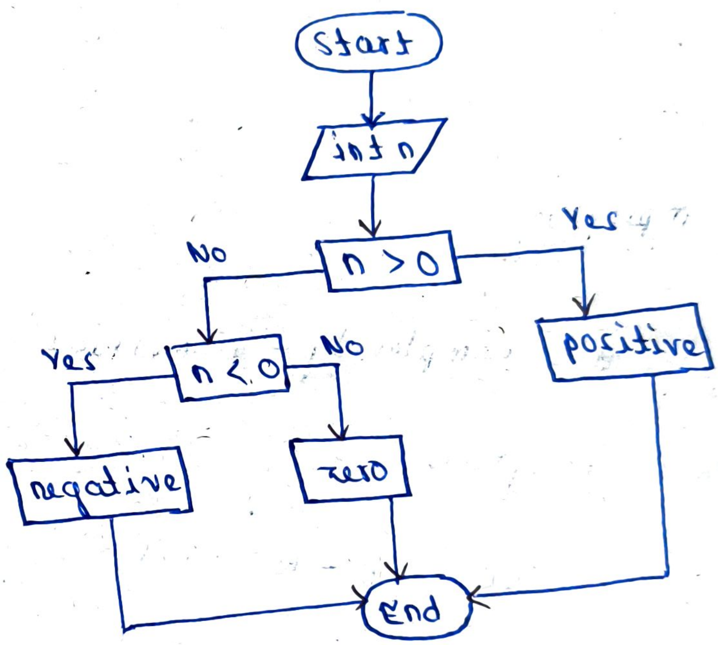
Cyclomatic complexity

1 + {Decision Paths}

= 1 + 2

= 3

control flow diagram



edge = 9

node = 8

Interpretation

CC ≤ 10 → simple and maintainable code

10 ≤ CC ≤ 20 → Moderately complex code

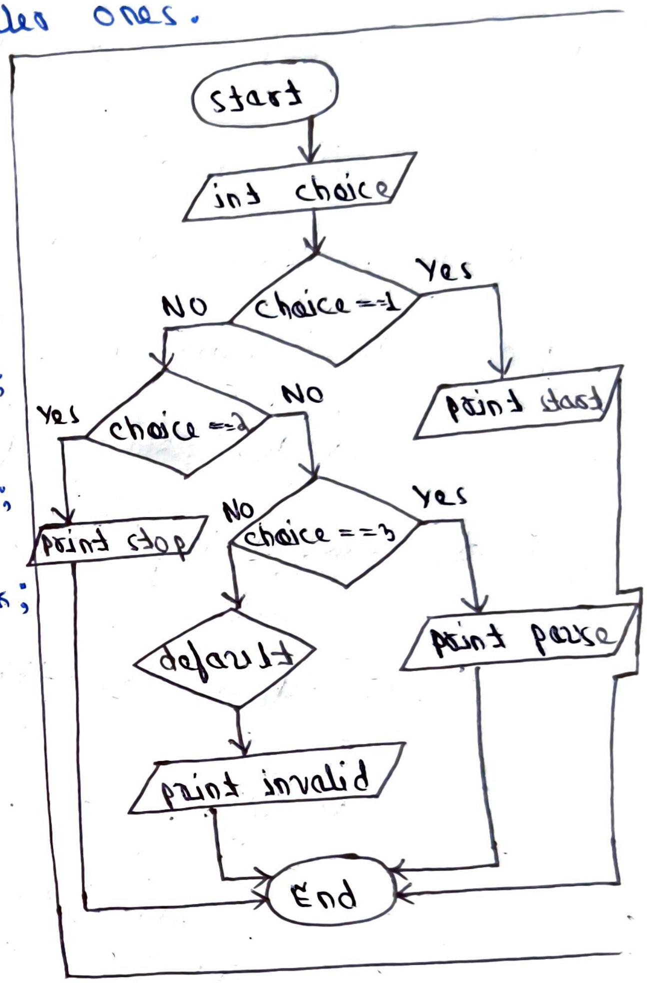
CC > 20 → High complexity code

Reducing Cyclomatic Complexity

(a) reduce nested cond-n's or loops

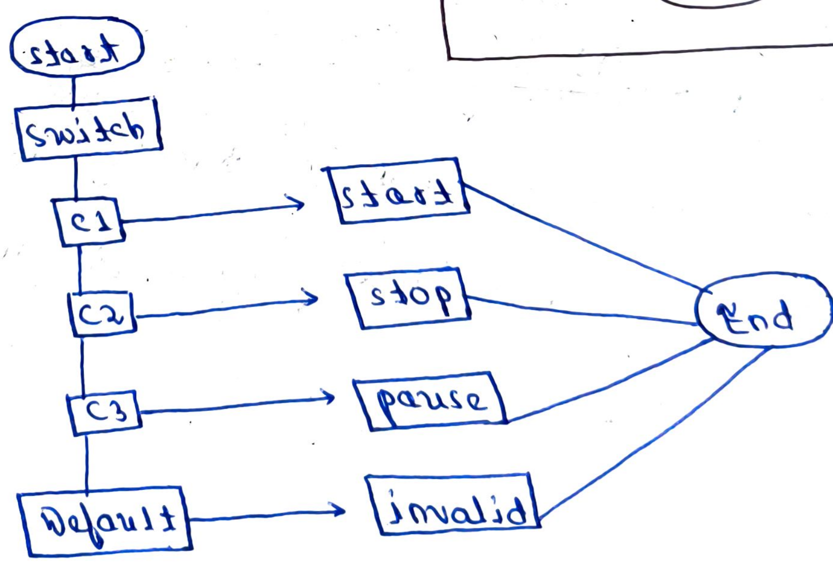
(b) use polymorphism

(c) break large function into smaller ones.




```
void menu (int choice)
    switch (choice) {
        case 1: cout << "start", break;
        case 2: cout << "stop", break;
        case 3: cout << "Pause", break;
        default: cout << "Invalid"
    }
```

Control Flow Graph (CFG)



Halstead Metrics

n_1: no. of distinct operators

n_2: no. of distinct operands

N_1: total occurances of operators

N_2: total occurances of operands

Key Metrics

(a) Program vocabulary – $n = n_1 + n_2$

(b) Program length – $N = N_1 + N_2$

(c) Difficult (D) – $D = (n_1/2) \times (N_2/n_2)$

(d) volume – $N \times \log_2(n)$

(e) effort (E) = $D \times V$

Eg.

```
int max (int a, int b) {
    int m;
    if (a>b)
        m = a;
    else
        m = b;
    return m;
}
```

Operators	Count
,	1
;	4
>	1
=	2
( )	2
{ }	1

$\therefore n_1 = 6$

$N_1 = 1+4+1+2+2+1$

$= 11$

and,

Operands	count
max	1
m	4
a	3
b	3

$\therefore n_2 = 4$

$N_2 = 1 + 4 + 3 + 3$

$= 11$

(a) Program vocabulary

$n = n_1 + n_2$

$\Rightarrow n = 6 + 4$

$= 10$

(b) Program length

$N = N_1 + N_2$

$\Rightarrow N = 11 + 11$

$= 22$

(c) Difficulty (D)

$D = (n_1/2) \times (N_2/n_2)$

$= (6/2) \times (11/4)$

$= 3 \times \frac{11}{4}$

$= 8.25$

(d) Volume

$V = N \times \log_2(n)$

$= N \times \log_2(10)$

$= 22 \times \log_2(10)$

$= 22 \times 3.322$

$= 73.04$

(e) Effort

$E = D \times V$

$= 8.25 \times 73.04$

$= 602.58$

Maintainability Index (MI)

$$MI = 171 - 5.2 \ln(\text{Halstead Volume}) - 0.23 \times (\text{cyclomatic complexity}) - 16.2 \ln(LOC)$$

$$= 171 - 5.2 \ln(73.04) - 0.23 \times 2 - 16.2 \ln(7)$$

Range of MI

(a) $MI > 85$ → Highly Maintainable Code

(b) $65 \le MI \le 85$ → Moderately Maintainable Code

(c) $MI < 65$ → Difficult to Maintain

Assessing Effectiveness of Test Adequacy Metrics

(a) Code Coverage Metrics

(1) statement

(2) branch

(3) path

(4) condition

(b) Defect Detection Efficiency (DDE)

$$DDE = \frac{\text{Total Defects Detected by Testing}}{\text{Total Defects}} \times 100$$

Total Defects → Detected + Undetected defects eg. Detected by testers

Defects = 40

Defects = 10

found after release

} Total Defect = 50

$$\therefore DDE = \frac{40}{\cancel{50}} \times \overset{2}{\cancel{100}}$$

$$= 80\%$$

(c) Fault Injection - intentionally adding errors to check how software behaves.

compile-time → developer modifies source code to introduce faults

Run-time → errors injected during program execution