Summary of Test Case Design Techniques

Brian Nielsen,

Arne Skou

{bnielsen | ask}@cs.auc.dk

BRICS
Basic Research
in Computer Science

Development of Test Cases

Complete testing is impossible

$\Downarrow$

Testing cannot guarantee the absence of faults

$\Downarrow$

How to select subset of test cases from all possible test cases with a high chance of detecting most faults ?

$\Downarrow$

Test Case Design Strategies

Whitebox Testing Overview

- White box testing
  - Flowgraphs
  - Test criteria/coverage
    - Statement / branch / decision / condition / path coverage
    - MC/DC coverage
  - Looptesting
  - Def-use pairs
  - Efficiency of different criteria

BEWARE: Expected outcome cannot be determined from code!

White-Box : Statement Testing

- Execute every statement of a program
- Relatively weak criterion
- Weakest white-box criterion

Example : Statement Testing

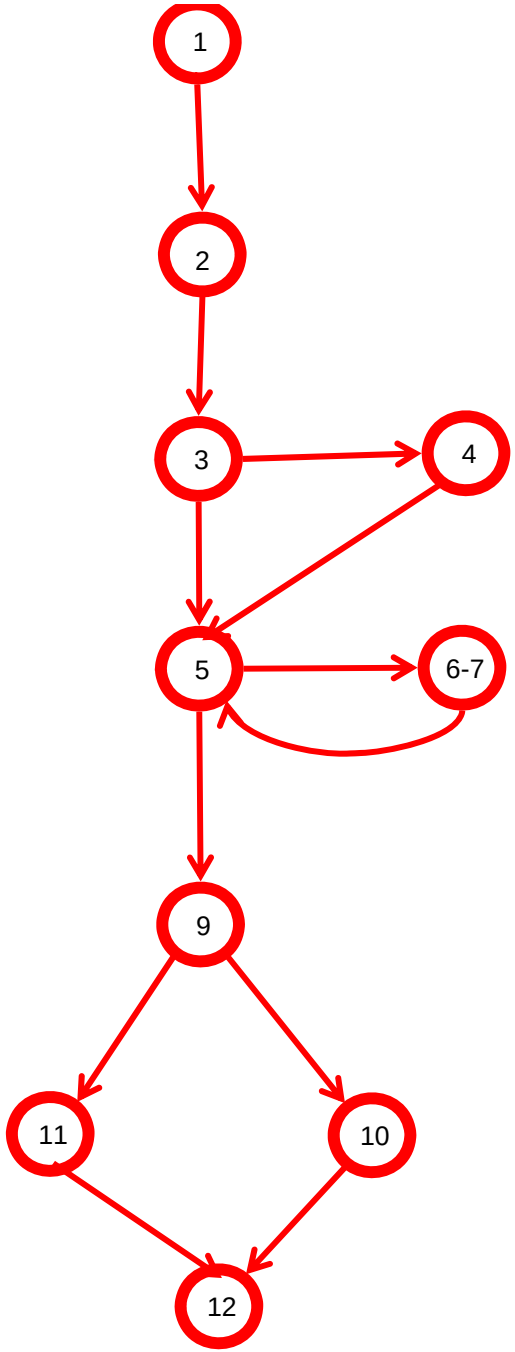
(*result* = 0+1+…+|value|, if this <= *maxint*, error otherwise)

```
PROGRAM  maxsum ( maxint, value : INT )
        INT    result := 0 ;   i := 0 ;
        IF   value < 0
        THEN   value  :=  - value ;
        WHILE   ( i < value )  AND  ( result <= maxint )
        DO       i  :=  i + 1 ;
                 result  :=  result + i ;
        OD;
        IF   result <= maxint
        THEN   OUTPUT ( result )
        ELSE   OUTPUT ( “too large” )
END.
```

Flowgrah

Node = statement (blocks)
Edges = possible successor (control flow)

```
1   PROGRAM  maxsum ( maxint, value : INT )
2                    INT    result := 0 ;   i := 0 ;
3                    IF   value < 0
4                    THEN   value  :=  - value ;
5                    WHILE   ( i < value )  AND  ( result <= maxint )
6                    DO          i  :=  i + 1 ;
7                                result  :=  result + i ;
8                    OD;
9                    IF   result <= maxint
10                   THEN   OUTPUT ( result )
11                   ELSE   OUTPUT ( “too large” )
12          END.
```



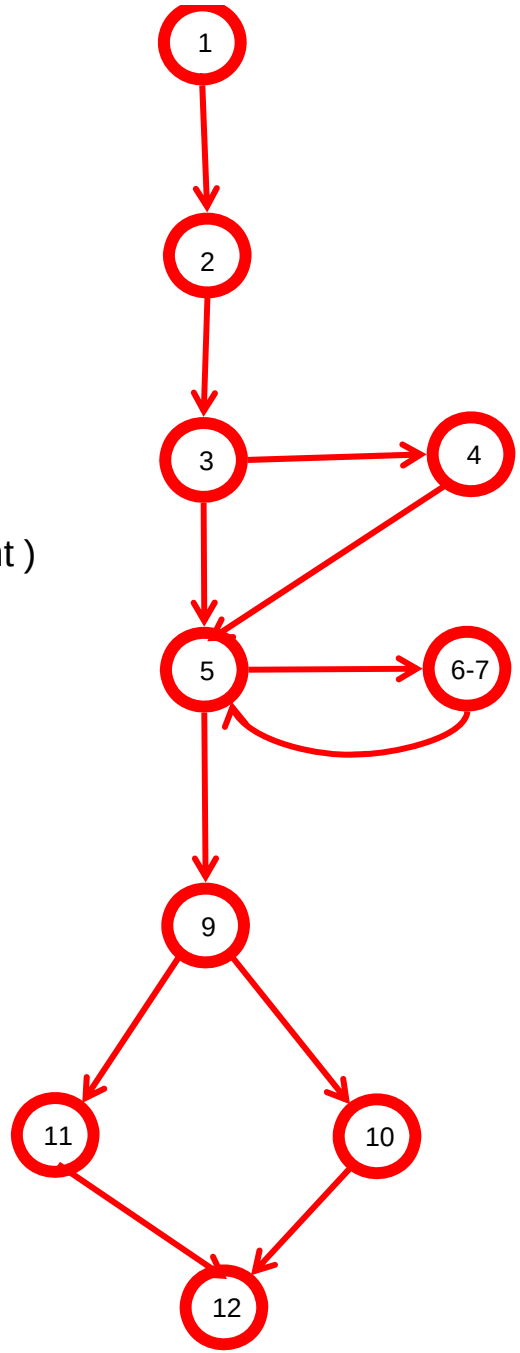
CISS

Flow graph: Cyclomatic complexity

- #edges - #nodes + 2
- Defines the maximal number of test cases needed to provide statement coverage
- Mostly applicable for Unit testing
- Strategy for statement coverage:
  1. Derive flow graph
  2. Find cyclomatic complexity #c
  3. Determine at most #c independent paths through the program (add one new edge for each test case)
  4. Prepare test cases covering the edges for each path (possibly fewer than #c cases)

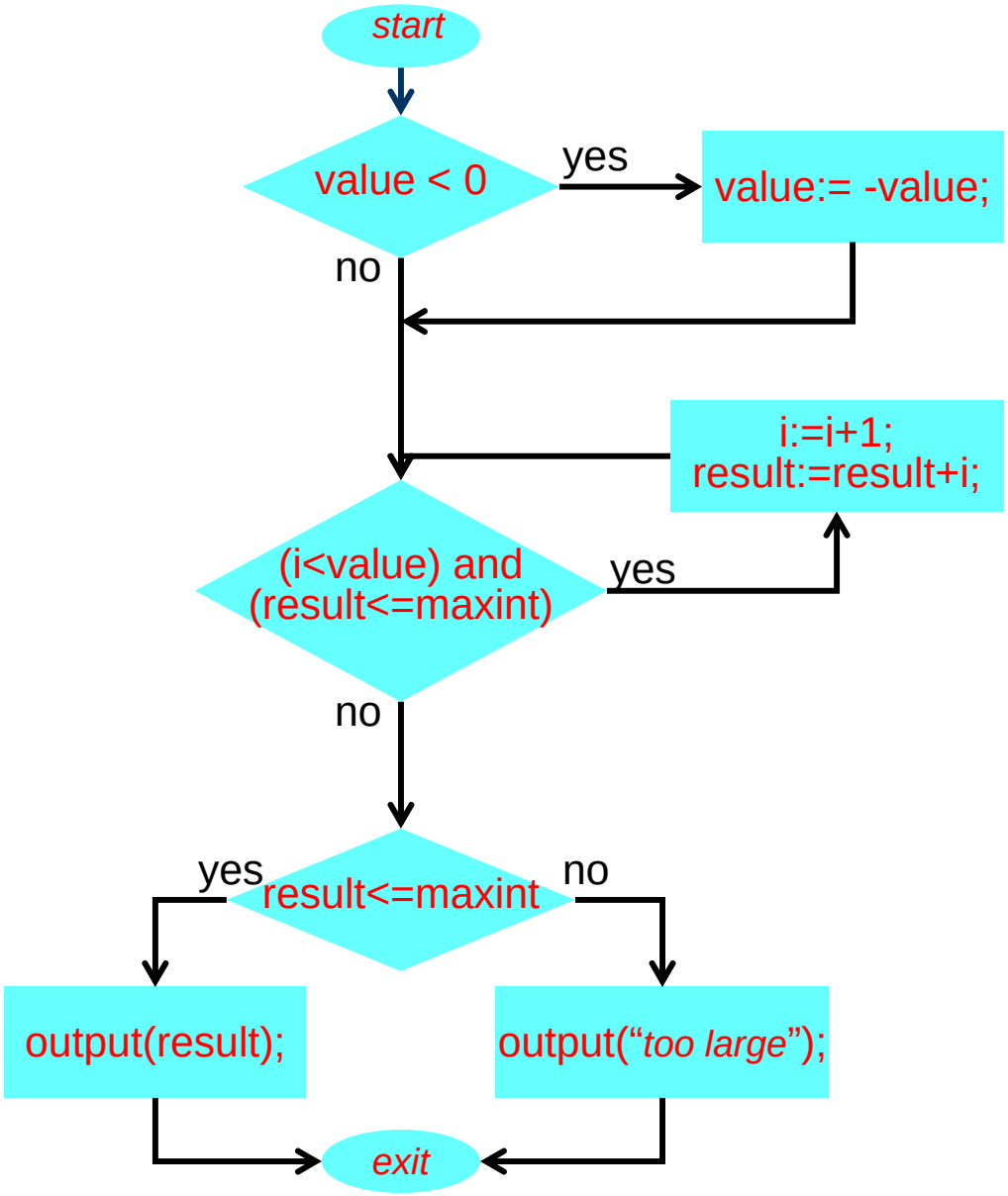
Cyclomatic complexity?

```
1   PROGRAM  maxsum ( maxint, value : INT )
2                    INT    result := 0 ;   i := 0 ;
3                    IF   value < 0
4                    THEN   value  :=  - value ;
5                    WHILE   ( i < value )  AND  ( result <= maxint )
6                    DO          i  :=  i + 1 ;
7                                result  :=  result + i ;
8                    OD;
9                    IF   result <= maxint
10                   THEN   OUTPUT ( result )
11                   ELSE   OUTPUT ( “too large” )
12          END.
```



CISS

Example : Statement Testing



Tests for complete statement coverage:

maxint	*value*
10	-1
0	-1

CISS

White-Box : Path Testing

- Execute every possible *path* of a program,

  i.e., every possible sequence of statements
- Strongest white-box criterion
- Usually impossible: infinitely many paths (in case of loops)
- So: not a realistic option
- But note : enormous reduction w.r.t. all possible test cases

  (each sequence of statements executed for only one value)

White-Box : Branch Testing

- Branch testing == decision testing
- Execute every branch of a program :

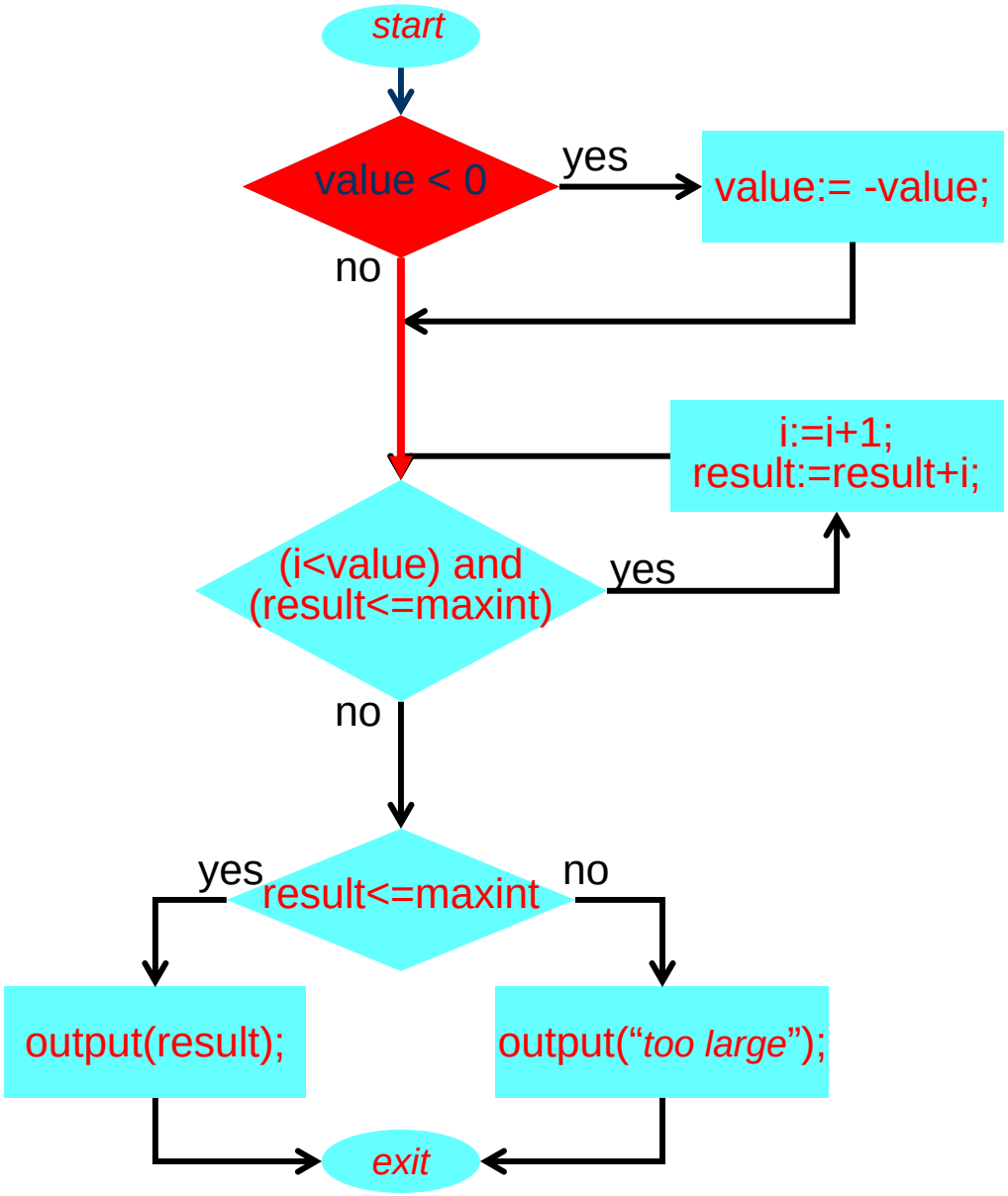
  each possible outcome of each decision occurs at least once
- Example:
  - IF b THEN s1 ELSE s2
  - IF b THEN s1; s2
  - CASE x OF

    1 : ....

    2 : ....

    3 : ....

Example : Branch Testing



Tests for complete statement coverage:

maxint	*value*
10	-1
0	-1

is not sufficient for branch coverage;

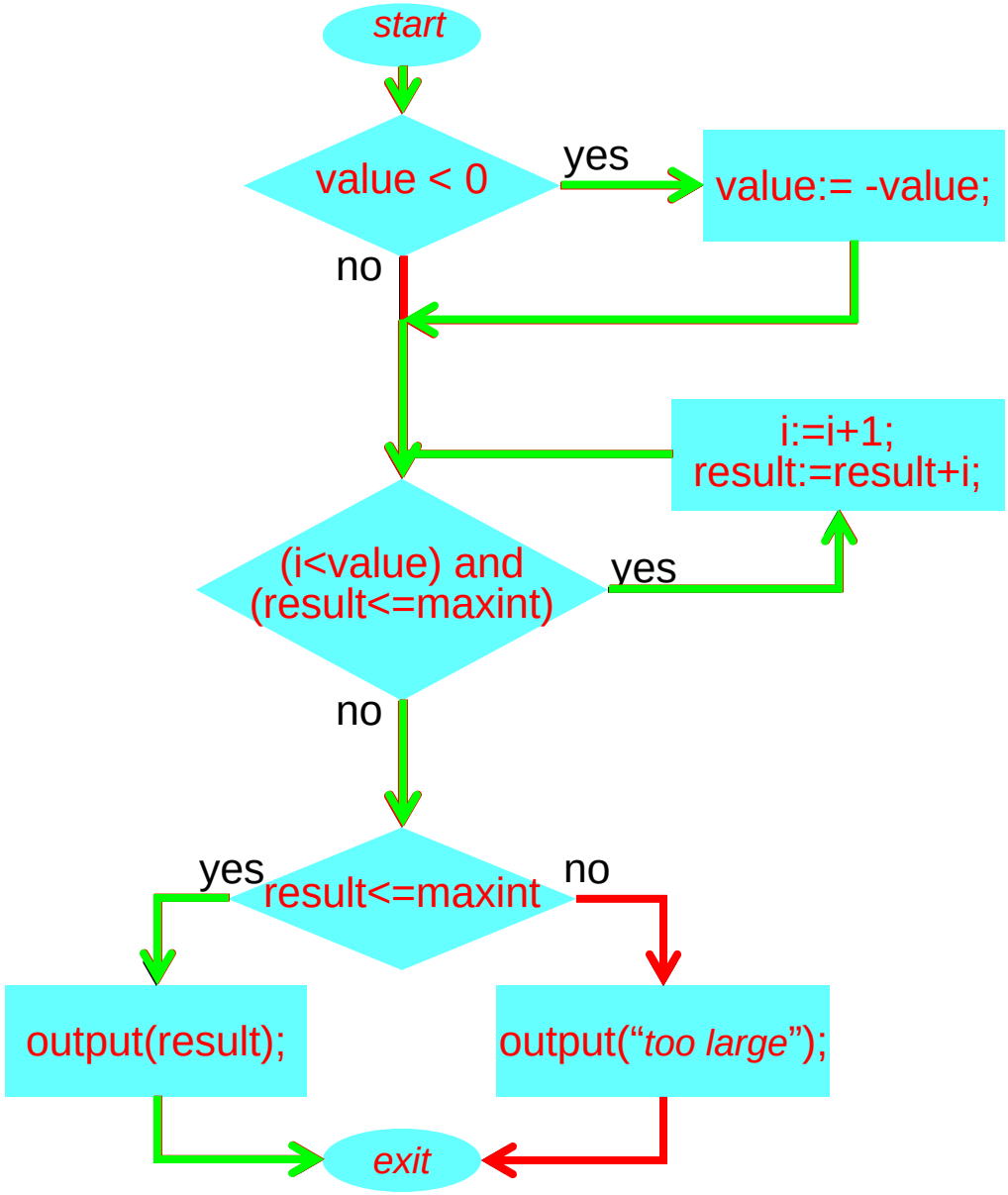
Take:

maxint	*value*
10	3
0	-1

for complete branch coverage

CISS

Example : Branch Testing



maxint	*value*
-1	-1
-1	-1
10	3

But:

No green path !

Needed :

Combination of decisions

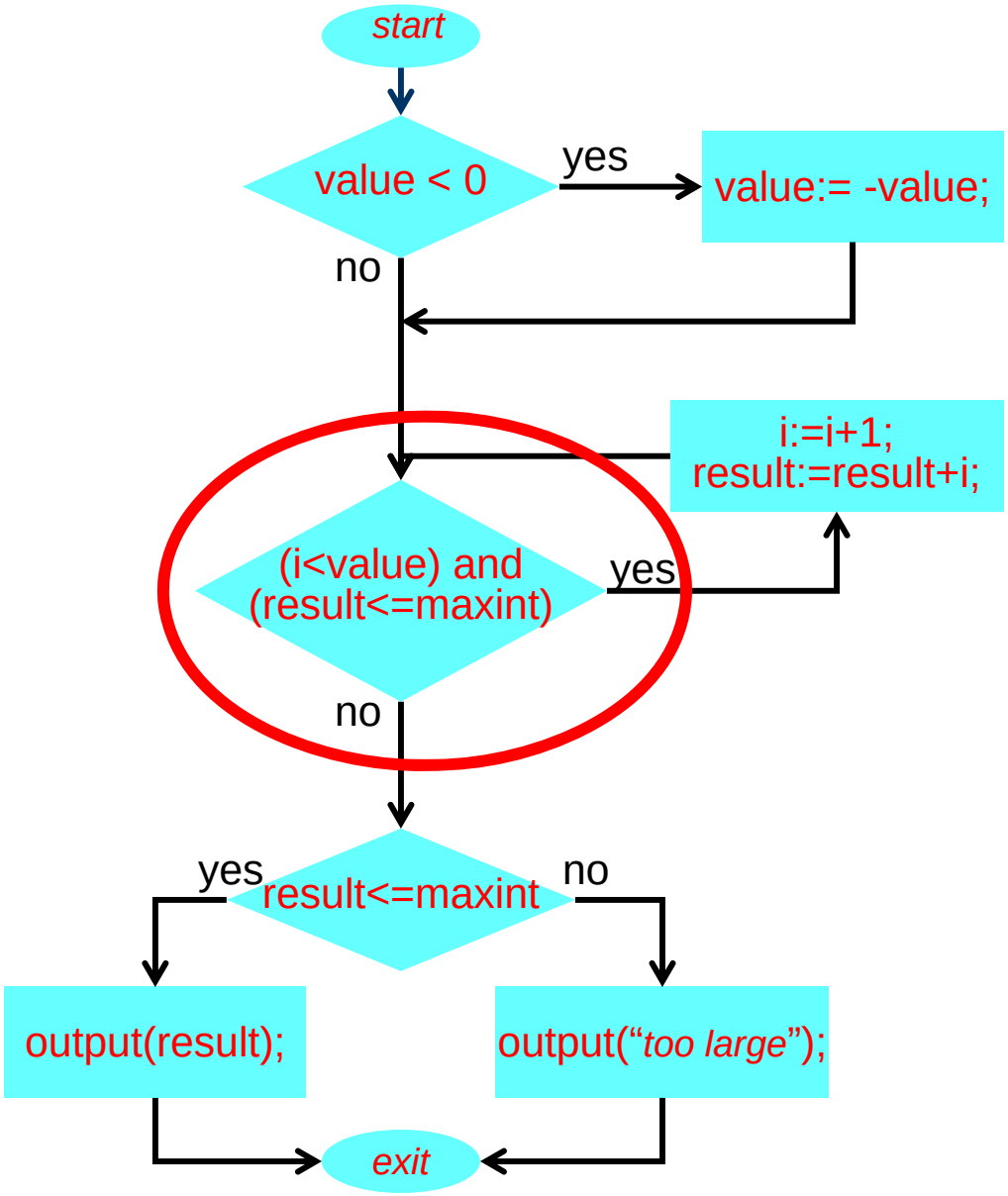
10 -3

CISS

White-Box : Condition Testing

- Design test cases such that each possible outcome of each condition in each decision occurs at least once
- Example:
  - decision (i < value) AND (result <= maxint)
    consists of two conditions : (i < value) AND (result <= maxint)
    test cases should be designed such that each gets value true and false at least once

Example : Condition Testing



(i = result = 0) :

maxint value	i<value	result<=maxint
-1	1	true	false
1	0	false	true

gives condition coverage
for all conditions

But it does not preserve
decision coverage

⇓

always take care that
condition coverage
preserves decision coverage :
decision / condition coverage

CISS

White-Box : Multiple Condition Testing

- Design test cases for each combination of conditions
- Example:
  - (i < value) (result <= maxint)

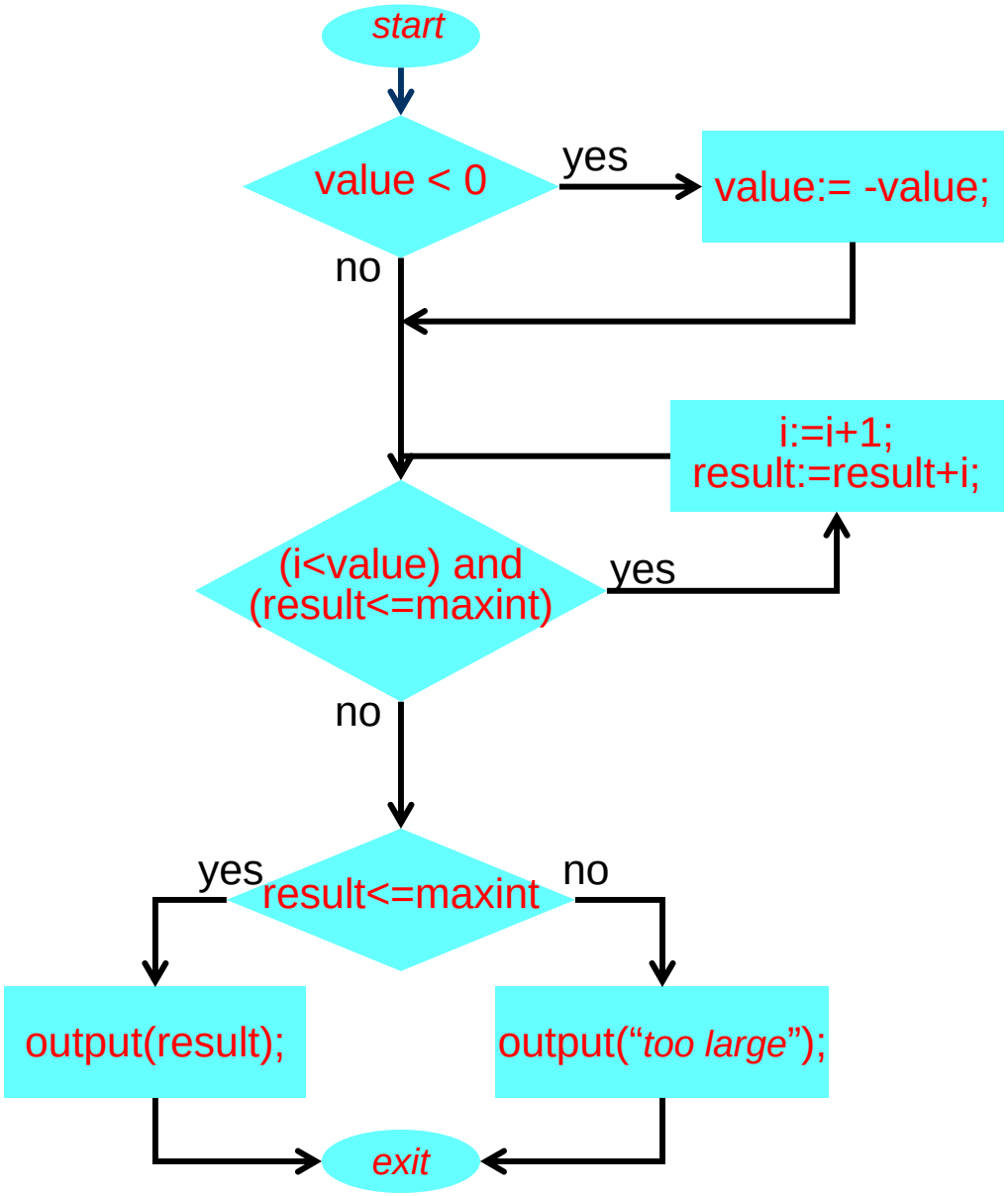
(i < value)	(result <= maxint)
false	false
false	true
true	false
true	true

- Implies decision-, condition-, decision/condition coverage
- But : exponential blow-up
- Again : some combinations may be infeasible

White-box: loop testing

- Statement and branch coverage are not sufficient
- Single loop strategy:
  - Zero iterations
  - One iteration
  - Two iterations
  - Typical number of iterations
  - n-1, n, and n+1 iterations (n maximum number of allowable iterations)
- Nested loop strategy:
  - Single loop strategy often intractable
  - Select minimum values for outer loop(s)
  - Treat inner loop as a single loop
  - Work ‘outwards’ and choose typical values for inner loops
- Concatenated loops:
  - Treat as single, if independent
  - Treat as nested, if dependent

Example : Loop testing



Tests for complete loop coverage:

maxint	*value*
15	0
15	1
15	2
15	3
6	4
15	5

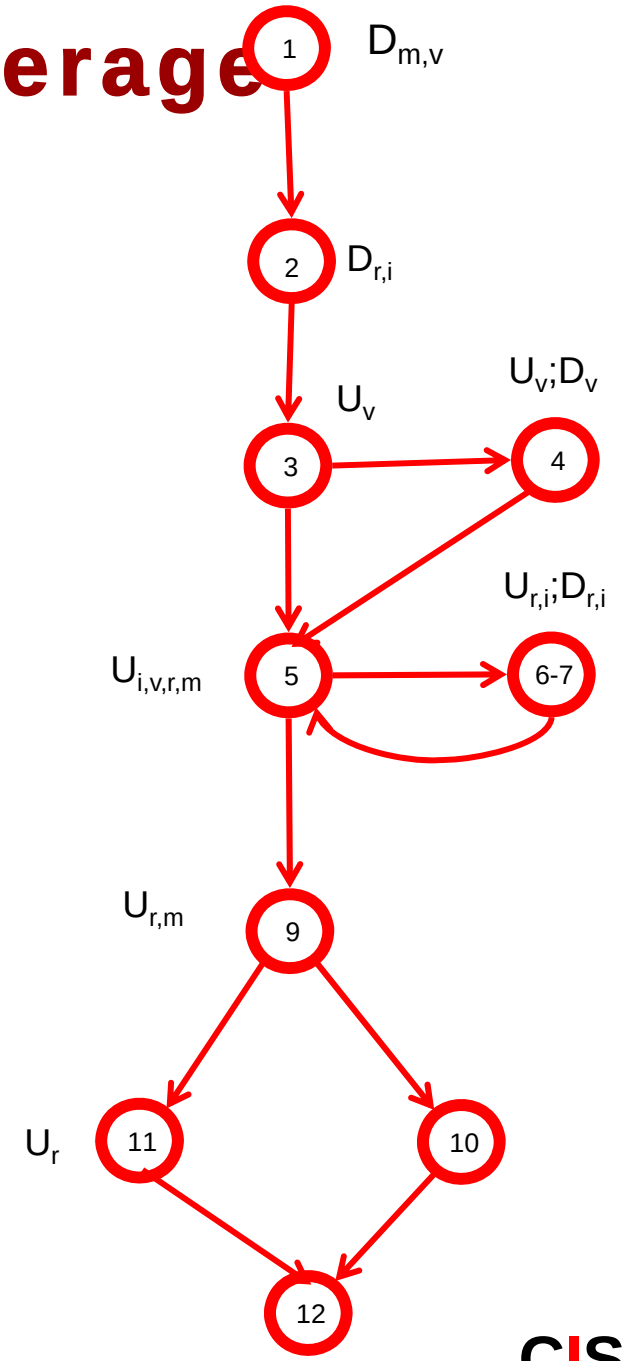
CISS

White-box testing: Data Flow Criteria

- Basic idea: For each variable definition (assignment), find a path (and a corresponding test case), to its use(s). A pair (definition,use) is often called a DU pair.
- Three dominant strategies:
  - All-defs (AD) strategy: follow at least one path from each definition to some use of it
  - All-uses (AU) strategy: follow at least one path for each DU pair
  - All-du-uses strategy (ADUP): follow all paths between a DU pair
- Complements the testing power of decision coverage

Example: All-uses coverage

```
1   PROGRAM  maxsum ( maxint, value : INT )
2                    INT    result := 0 ;   i := 0 ;
3                    IF   value < 0
4                    THEN   value  :=  - value ;
5                    WHILE   ( i < value )  AND  ( result <= maxint )
6                    DO          i  :=  i + 1 ;
7                                result  :=  result + i ;
8                    OD;
9                    IF   result <= maxint
10                   THEN   OUTPUT ( result )
11                   ELSE   OUTPUT ( “too large” )
12          END.
```



Def-use pairs:

1-3,1-5,1-9,1-4

2-5,2-9,2-6

4-5

6-5,6-9,6-11

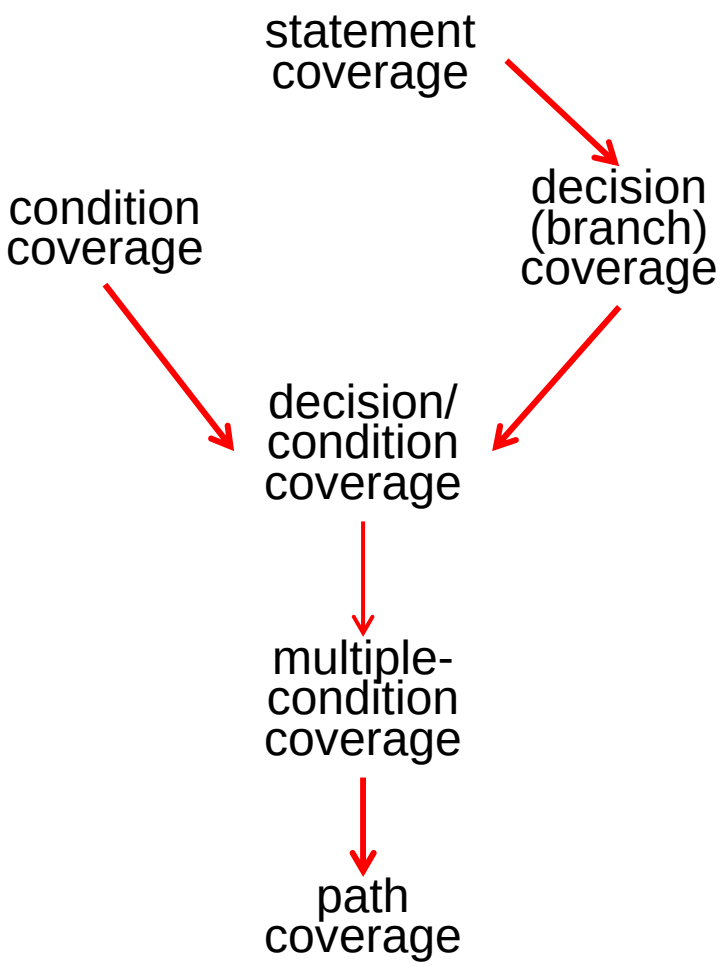
6-5-6

Tests for complete all-uses coverage:

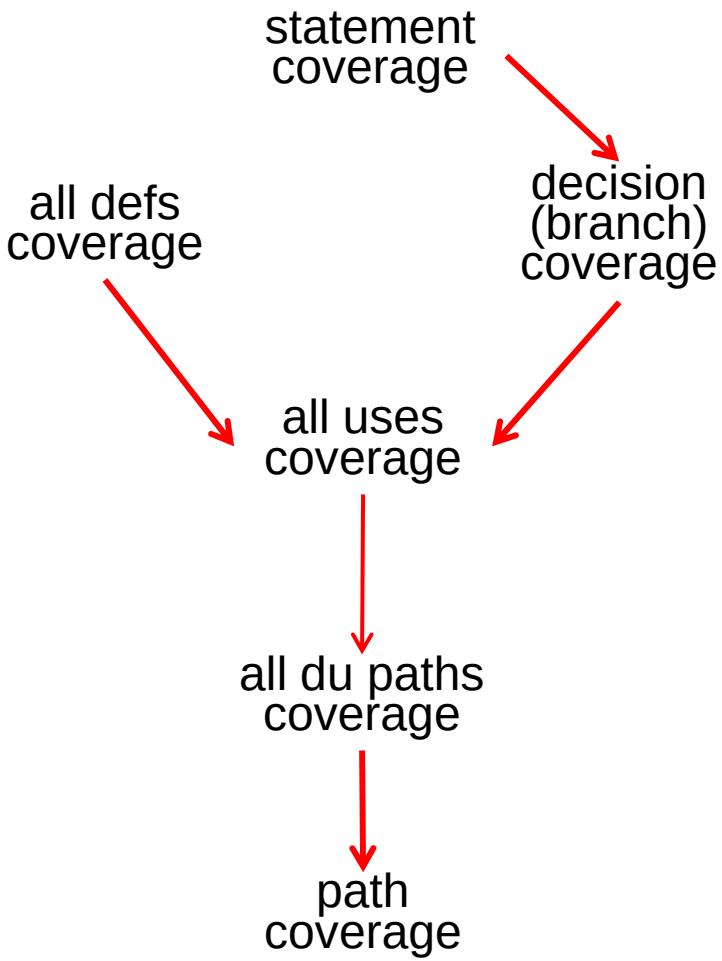
maxint	*value*
0	0
0	-1
10	1
10	2

CISS

White-Box : Overview



White-Box : Overview



Additional techniques: mutation and random testing

- Mutation testing:
  - Intended for evaluating the test cases
  - Create at set of slightly modified mutants of the original program containing errors
  - Run the test cases against the mutants
  - Criteria
    - All mutants must fail (strong)
    - All mutants will eventually fail (weak)
- Random testing:
  - Basic idea: run the program with arbitrary inputs
  - Inherent problems: How to define the oracle for arbitrary inputs and how to decide to stop?
  - Advantage: The program structure can be ignored

Efficiency of white-box techniques: two studies

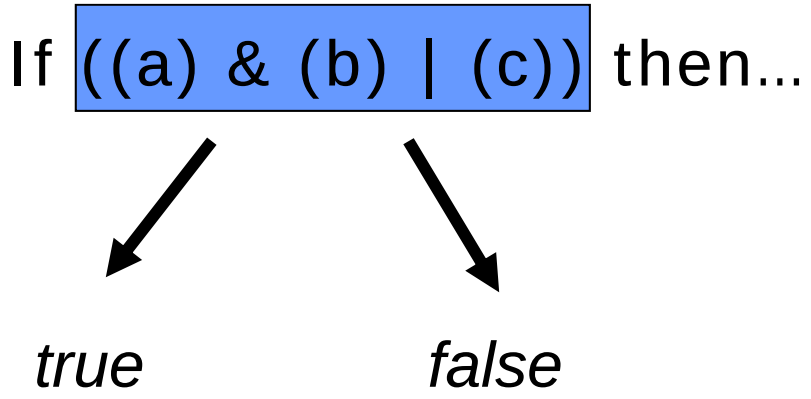
Strategy	#test cases	%bugs found
Random	35	93.7
Branch	3.8	91.6
All-uses	11.3	96.3
Random	100	79.5
Branch	34	85.5
All-uses	84	90.0

What is MC/DC?

- MC/DC stands for **M**odified **C**ondition / **D**ecision **C**overage
- Main idea: Each condition must be shown to independently affect the outcome of a decision, i.e. the outcome of a decision changes as a result of changing a single condition.

1. MC/DC Requirement

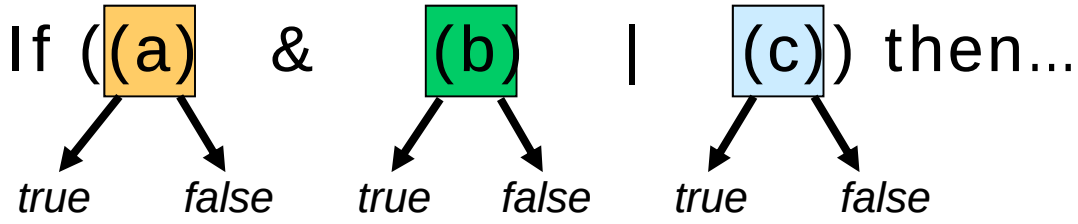
- The decision has taken all possible outcomes at least once.



- We could also say we cover both the true and the false branch (like we do in Branch Testing).

2. MC/DC Requirement

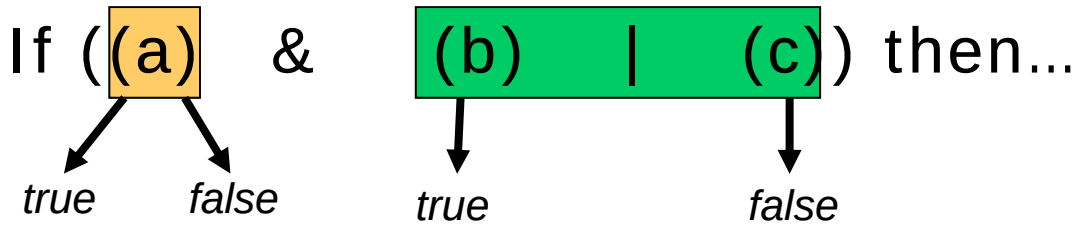
- Every condition in the decision has taken all possible outcomes at least once.



- Aims to cover compound conditions (like Condition/Decision Coverage)

3. MC/DC Requirement

- Every condition in the decision independently affects the decision's outcome.



Change the value of each condition individually while keeping all other conditions constant.

Creating MC/DC test cases

`If (A and B) then…`

- (1) create truth table for conditions.
- (2) Extend truth table so that it indicated which test cases can be used to show the independence of each condition.

A B	result
T T	T
T F	F
F T	F
F F	F

number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

Creating MC/DC test cases cont'd

number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

- Show independence of **A**:
  - Take 1 + 3
- Show independence of **B**:
  - Take 1 + 2
- Resulting test cases are
  - 1 + 2 + 3
  - (T , T) + (T , F) + (F , T)

More advanced example

If (A and (B or C)) then…

number	ABC	result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

Note: We want to determine the MINIMAL set of test cases

Here:

{2,3,4,6}

{2,3,4,7}

Non-minimal set is:

{1,2,3,4,5}

CISS

Argument in the software industry

- Federal Aviation Administration's requirement that test suites be MC/DC adequate (DO-178B).

- In the avionics domain, complex Boolean expressions are much more common.

N	1	2	3	4	5	6–10	11–15	> 15
Software Tools	446	72	9	0	0	0	0	0
Booch Components	9048	402	52	0	0	0	0	0
EFIS avionics display	1343	182	38	16	18	11	3	0

Fig. 4 Complexity of expressions in representative domains

Argument in the software industry

- Federal Aviation Administration's requirement that test suites be MC/DC adequate.
- Argument – Too expensive
  - "For example, one of our industrial partners reports that for one of its products of about 20,000 lines of code, the MC/DC-adequate test suite requires seven weeks to run."
- Counter Argument – Significantly more effective
  - A recent empirical study performed during the real testing of the attitude-control software for the HETE-2 (High Energy Transient Explorer) found that the test cases generated to satisfy the MC/DC coverage requirement detected important errors not detectable by functional testing.

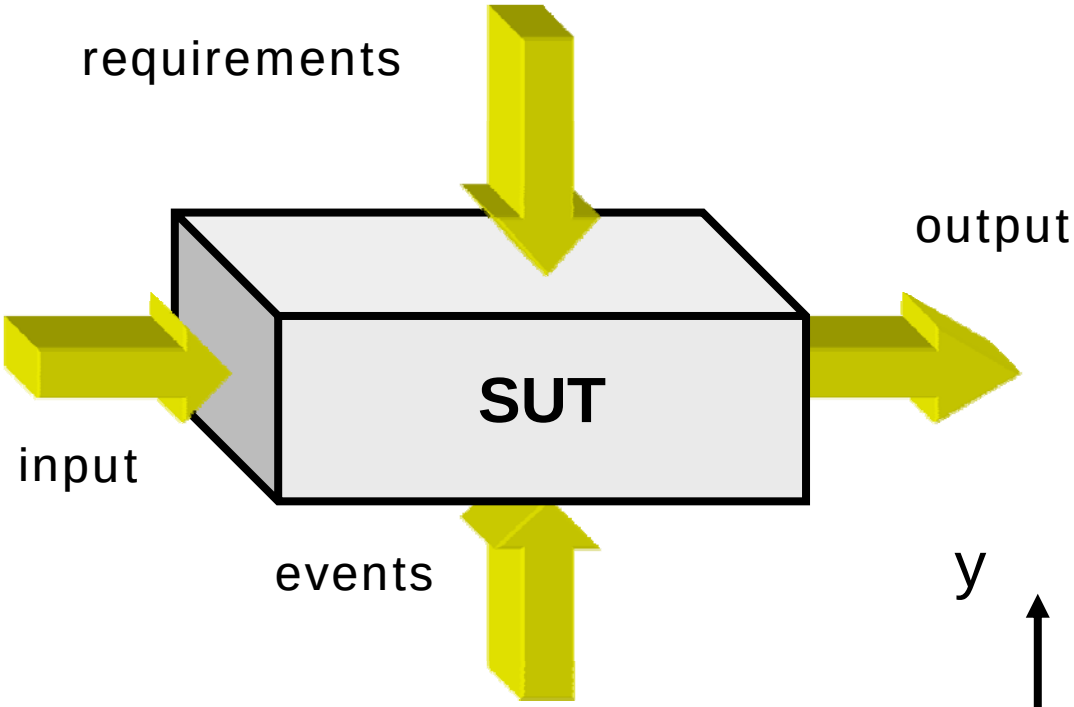
Blackbox Testing Overview

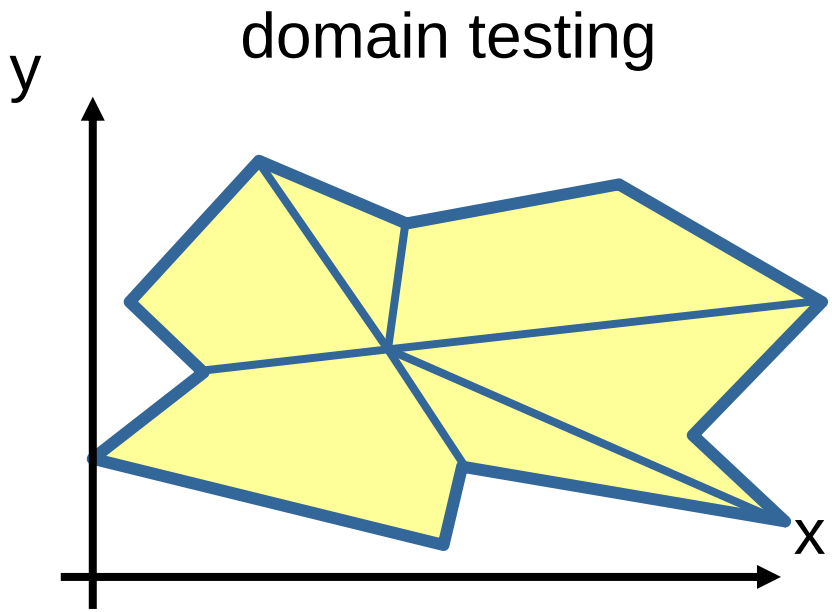
- Black-box testing (or functional testing):
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing

  } Domain analysis

  - Behavioural testing / state transition testing
  - Random testing, Error guessing etc…
  - Others: Classification Tree Method, Pairwise testing, Use Case Testing, Syntax testing, ..
- Basics : heuristics and experience
- How to use black-box and white-box testing in combination

Black box testing





CISS

Black-box: Equivalence Partitioning

Strategy :

- Identify input equivalence classes
  - Based on conditions on inputs / outputs in specification / description
  - Both *valid* and *invalid* input equivalence classes
  - Based on heuristics and experience
    - "input x in [1..10]" $\rightarrow$ classes : $x < 1$, $1 \leq x \leq 10$, $x > 10$
    - "enumeration A, B, C $\rightarrow$ classes : A, B, C, not{A,B,C,}
    - ……..
- Define one / couple of test cases for each class
  - Test cases that cover valid eq. classes
  - Test cases that cover at most one invalid eq. class

Example: Equivalence Partitioning

- Test a function for calculation of absolute value of an integer
- Equivalence classes :

Condition	*Valid eq. classes*	*Invalid eq.* Classes
nr of inputs	1	0, > 1
Input type	integer	non-integer
particular *abs*	< 0, >= 0	

- Test inputs :
  - *x* = -10, *x* = 100
  - *x* = “XYZ”, x = - *x* = 10 20

A Self-Assessment Test [Myers]

- “A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene (uligesidet), isosceles (ligebenet) , or equilateral (ligesidet).”

Write a set of test cases to test this program.

A Self-Assessment Test [Myers]

Test cases for:

1. valid scalene triangle ?
2. valid equilateral triangle ?
3. valid isosceles triangle ?
4. 3 permutations of previous ?
5. side = 0 ?
6. negative side ?
7. one side is sum of others ?
8. 3 permutations of previous ?
9. one side larger than sum of others ?
10. 3 permutations of previous ?
11. all sides = 0 ?
12. non-integer input ?
13. wrong number of values ?
14. for each test case: is expected output specified ?
15. check behaviour after output was produced ?

Example: Equivalence Partitioning

- Test a program that computes the sum of the first *value* integers as long as this sum is less than *maxint.* Otherwise an error should be reported. If *value* is negative, then it takes the absolute value
- Formally:

  Given integer inputs *maxint* and *value* compute *result* :

$$result = \sum_{K=0}^{|value|} k \quad \text{if this} <= maxint, \quad \text{error otherwise}$$

Example: Equivalence Partitioning

- Equivalence classes :

Condition	*Valid eq. classes*	*Invalid eq. classes*
Nr of inputs	2	< 2, > 2
Type of input	int int	int no-int, no-int int
Abs(*value*)	$value < 0$, $value \geq 0$	
maxint	$\sum k \leq maxint$, $\sum k > maxint$	

- Test Cases :

	maxint	*value*	*result*
Valid	100	10	55
	100	-10	55
	10	10	error
Invalid	10	-	error
	10 20	30	error
	"XYZ"	10	error
	100	9.1E4	error

Black-box: Boundary Value Analysis

- Based on experience / heuristics :
  - Testing *boundary conditions* of eq. classes is more effective
    i.e. values directly on, above, and beneath edges of eq. classes
  - Choose input boundary values as tests in input eq. classes
    instead of, or additional to arbitrary values
  - Choose also inputs that invoke *output boundary values*
    (values on the boundary of output classes)
  - Example strategy as extension of equivalence partitioning:
    - choose one (*n*) arbitrary value in each eq. class
    - choose values exactly on lower and upper boundaries of eq. class
    - choose values immediately below and above each boundary
      (if applicable)

Example: Boundary Value Analysis

- Test a function for calculation of absolute value of an integer
- Valid equivalence classes :

Condition	*Valid eq. classes*	*Invalid eq.* Classes
particular *abs*	< 0, >= 0	

- Test cases :

class x < 0, arbitrary value:	*x* = -10
class x >= 0, arbitrary value	x = 100
classes x < 0, x >= 0, on boundary :	x = 0
classes x < 0, x >= 0, below and above:	x = -1, x = 1

Example: Boundary Value Analysis

- Given integer inputs *maxint* and *value* compute *result* :

$$result = \sum_{K=0}^{|value|} k \quad \text{if this} <= maxint, \quad \text{error otherwise}$$

- Valid equivalence classes :

Condition	*Valid eq. Classes*
Abs(*value*)	$value < 0, \quad value \geq 0$
maxint	$\sum k \leq maxint, \quad \sum k > maxint$

- Should we also distinguish between *maxint* < 0 and *maxint* >= 0 ?

maxint	$maxint < 0, \quad 0 \leq maxint < \sum k, \quad maxint \geq \sum k$

Example: Boundary Value Analysis

- Valid equivalence classes :

Abs(*value*)	$value < 0, \quad value \geq 0$
maxint	$maxint < 0, \quad 0 \leq maxint < \sum k, \quad maxint \geq \sum k$

- Test Cases :

maxint	*value*	*result*	*maxint*	*value*	*result*
55	10	55	100	0	0
54	10	error	100	-1	1
56	10	55	100	1	1
0	0	0	....	....	...
-10	1	error			

- How to combine the boundary conditions of different inputs ?
  Take all possible boundary combinations ? This may blow-up.

Black-box: Cause Effect Graphing

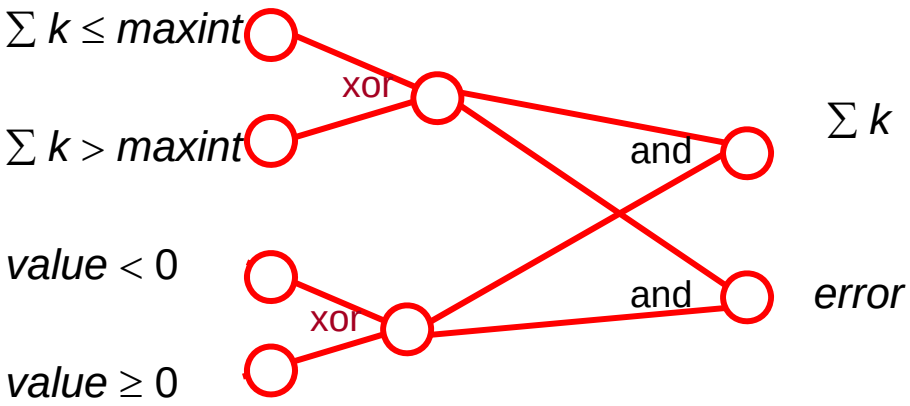
- Black-box testing technique to analyse combinations of input conditions
- Identify *causes* and *effects* in specification

causes	effects
↓	↓
inputs	outputs
current state	new state

- Make Boolean Graph linking causes and effects
- Annotate impossible combinations of causes and effects
- Develop decision table from graph with in each column a particular combination of inputs and outputs
- Transform each column into test case

CISS

Black-Box: Cause Effect Graphing



Causes	$\sum k \leq maxint$	1	1	0	0
inputs	$\sum k > maxint$	0	0	1	1
	$value < 0$	1	0	1	0
	$value \geq 0$	0	1	0	1
Effects	$\sum k$	1	1	0	0
outputs	$error$	0	0	1	1

Testcase

CISS

Black-box: Cause Effect Graphing

- Systematic method for generating test cases representing combinations of conditions
- Combinatorial explosion of number of possible combinations
- Some heuristics to reduce this combinatorial explosion
- Starting point is effects (outputs) then working ‘backwards’
- ‘light-weight’ formal methods:
  transformation into semi-formal Boolean graph
- *A* technique : to be combined with others

Black-box: Error Guessing

- Just 'guess' where the errors are ……
- Intuition and experience of tester
- Ad hoc, not really a technique
- Strategy:
  - Make a list of possible errors or error-prone situations (often related to boundary conditions)
  - Write test cases based on this list

Black-box : Error Guessing

- More sophisticated ‘error guessing’ : *Risk Analysis*
- Try to identify critical parts of program (high risk code sections):
  - parts with unclear specifications
  - developed by junior programmer while his wife was pregnant ……
  - complex code :

    measure code complexity - tools available (McGabe, Logiscope,…)
- High-risk code will be more thoroughly tested

  (or be rewritten immediately ….)

Black-Box Testing: Which One ?

- Black-box testing techniques :
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing
  - Error guessing
  - ……..
- Which one to use ?
  - None is complete
  - All are based on some kind of heuristics
  - They are complementary

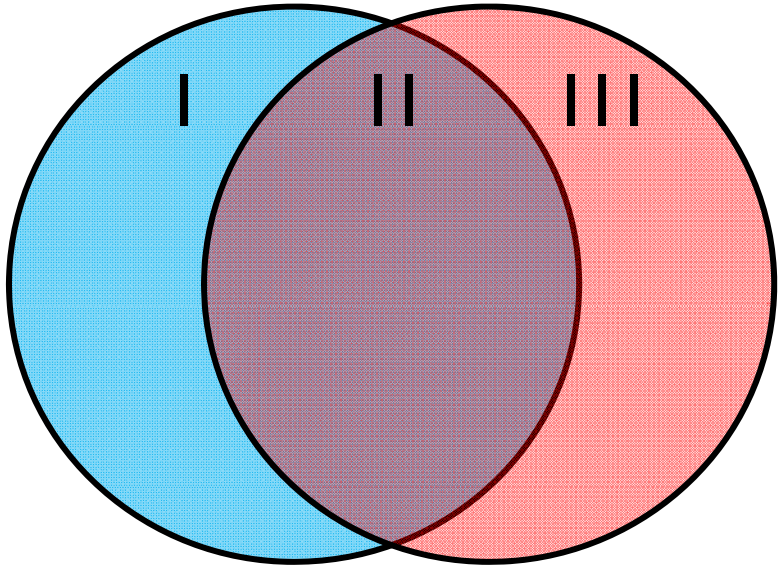
Black-Box Testing: Which One ?

- Always use a combination of techniques
  - When a formal specification is available try to use it
  - Identify valid and invalid input equivalence classes
  - Identify output equivalence classes
  - Apply boundary value analysis on valid equivalence classes
  - Guess about possible errors
  - Cause-effect graphing for linking inputs and outputs

White-Box testing : How to Apply ?

- Don’t start with designing white-box test cases !
- Start with black-box test cases (because requirements must be tested anyway)
  - equivalence partitioning, boundary value analysis,
    cause effect graphing, test derivation with formal methods, …..
- Check white-box coverage
  (statement-, branch-, condition-, ….. coverage)
- Use a *coverage tool* – maybe combined with a Unit framework
- Design additional white-box test cases for not covered code

Blackbox vs. Whitebox



Specified Behavior
(Blackbox)

Implemented Behavior
(Whitebox)

I: Specified, not implemented
II: Implemented and specified
III:Implemented, but not specified

A Coverage Tool : gcov

- Standard Gnu tool gcov
- Only *statement coverage*, but may be applied manually for multiple condition coverage
- Compile your program under test with a special option
- Run a number of test cases
- A listing indicates how often each statement was executed and percentage of statements executed

A Coverage Tool : BullseyeCoverage

- Supports *decision/condition*
- Compile your program under test with a special option
- Run a number of test cases
- A listing indicates the coverage

END

BRICS
Basic Research
in Computer Science