1 Black Box Test Data Generation Techniques

1.1 Equivalence Partitioning

Introduction

Equivalence partitioning is based on the premise that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component. Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition.

Example

Consider a component, *generate_grading,* with the following specification:

> *The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:*
>
> *greater than or equal to 70 - 'A'*
> *greater than or equal to 50, but less than 70 - 'B'*
> *greater than or equal to 30, but less than 50 - 'C'*
> *less than 30 - 'D'*
>
> *Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.*

Initially the equivalence partitions are identified and then test cases derived to exercise the partitions. Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

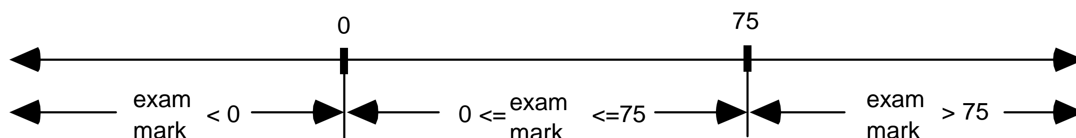
The partitions for the two inputs are initially identified. The *valid* partitions can be described by:

$0 \leq$ exam mark ≤ 75
$0 \leq$ coursework mark ≤ 25

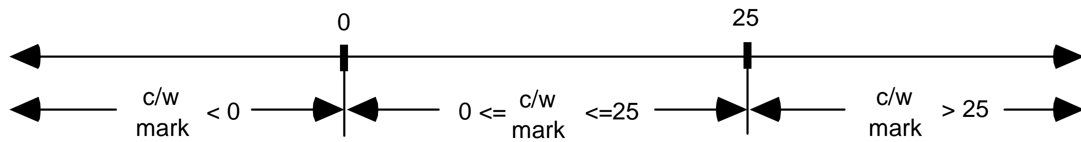
The most obvious *invalid* partitions based on the inputs can be described by:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0

Partitioned ranges of values can be represented pictorially, therefore, for the input, exam mark, we get:



And for the input, coursework mark, we get:



Less obvious invalid input equivalence partitions would include any other inputs that can occur not so far included in a partition, for instance, non-integer inputs or perhaps non-numeric inputs. So, we could generate the following invalid input equivalence partitions:

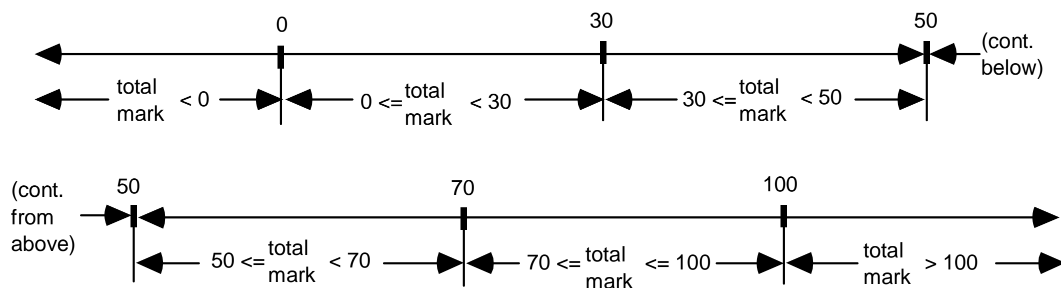
exam mark = real number (a number with a fractional part)
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic

Next, the partitions for the outputs are identified. The *valid* partitions are produced by considering each of the valid outputs for the component:

'A'	is induced by	70 ≤ total mark ≤ 100
'B'	is induced by	50 ≤ total mark < 70
'C'	is induced by	30 ≤ total mark < 50
'D'	is induced by	0 ≤ total mark < 30
'Fault Message'	is induced by	total mark > 100
'Fault Message'	is induced by	total mark < 0

where total mark = exam mark + coursework mark. Note that 'Fault Message' is considered as a valid output as it is a *specified* output.

The equivalence partitions and boundaries for total mark are shown pictorially below:



An invalid output would be any output from the component other than one of the five specified. It is difficult to identify unspecified outputs, but obviously they must be considered as if we can cause one then we have identified a flaw with either the component, its specification, or both. For this example three unspecified outputs were identified and are shown below. This aspect of equivalence partitioning is very subjective and different testers will inevitably identify different partitions which *they* feel could possibly occur.

output = 'E'
output = 'A+'
output = 'null'

Thus the following nineteen equivalence partitions have been identified for the component (remembering that for some of these partitions a certain degree of subjective choice was required, and so a different tester would not necessarily duplicate this list exactly):

$0 \leq$ exam mark ≤ 75
exam mark > 75
exam mark < 0
$0 \leq$ coursework mark ≤ 25
coursework mark > 25
coursework mark < 0
exam mark = real number
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic
$70 \leq$ total mark ≤ 100
$50 \leq$ total mark < 70
$30 \leq$ total mark < 50
$0 \leq$ total mark < 30
total mark > 100
total mark < 0
output = 'E'
output = 'A+'
output = 'null'

Having identified all the partitions then test cases are derived that 'hit' each of them. Two distinct approaches can be taken when generating the test cases. In the first a test case is generated for each identified partition on a one-to-one basis, while in the second a minimal set of test cases is generated that cover all the identified partitions.

The one-to-one approach will be demonstrated first as it can make it easier to see the link between partitions and test cases. For each of these test cases only the single partition being targetted is stated explicitly. Nineteen partitions were identified leading to nineteen test cases.

The test cases corresponding to partitions derived from the input exam mark are:

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	$0 \leq e \leq 75$	$e < 0$	$e > 75$
Exp. Output	'B'	'FM'	'FM'

Note that the input coursework (c/w) mark has been set to an arbitrary valid value of 15.

The test cases corresponding to partitions derived from the input coursework mark are:

Test Case	4	5	6
Input (exam mark)	40	40	40
Input (c/w mark)	8	-15	47
total mark (as calculated)	48	25	87
Partition tested (of c/w mark)	$0 \leq c \leq 25$	$c < 0$	$c > 25$
Exp. Output	'C'	'FM'	'FM'

Note that the input exam mark has been set to an arbitrary valid value of 40.

The test cases corresponding to partitions derived from possible invalid inputs are:

Test Case	7	8	9	10
Input (exam mark)	48.7	q	40	40
Input (c/w mark)	15	15	12.76	g
total mark (as calculated)	63.7	not applicable	52.76	not applicable
Partition tested	exam mark = real number	exam mark = alphabetic	c/w mark = real number	c/w mark = alphabetic
Exp. Output	'FM'	'FM'	'FM'	'FM'

The test cases corresponding to partitions derived from the valid outputs are:

Test Case	11	12	13
Input (exam mark)	-10	12	32
Input (c/w mark)	-10	5	13
total mark (as calculated)	-20	17	45
Partition tested (of total mark)	$t < 0$	$0 \le t < 30$	$30 \le t < 50$
Exp. Output	'FM'	'D'	'C'

Test Case	14	15	16
Input (exam mark)	44	60	80
Input (c/w mark)	22	20	30
total mark (as calculated)	66	80	110
Partition tested (of total mark)	$50 \le t < 70$	$70 \le t \le 100$	$t > 100$
Exp. Output	'B'	'A'	'FM'

The input values of exam mark and coursework mark have been derived from the total mark, which is their sum.

The test cases corresponding to partitions derived from the invalid outputs are:

Test Case	17	18	19
Input (exam mark)	-10	100	null
Input (c/w mark)	0	10	null
total mark (as calculated)	-10	110	null+null
Partition tested (output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'

It should be noted that where invalid input values are used (as above, in test cases 2, 3, 5-11, and 16-19) it may, depending on the implementation, be impossible to actually execute the test case. For instance, in Ada, if the input variable is declared as a positive integer then it will not be possible to assign a negative value to it. Despite this, it is still worthwhile *considering* all the test cases for completeness.

It can be seen above that several of the test cases are similar, such as test cases 1 and 14, where the main difference between them is the partition targetted. As the component has two inputs and one output, each test case actually 'hits' three partitions; two input partitions and one output partition.. Thus it is possible to generate a smaller 'minimal' test set that still 'hits' all the identified partitions by deriving test cases that are designed to exercise more than one partition. The following test case suite

of eleven test cases corresponds to the minimised test case suite approach where each test case is designed to hit as many new partitions as possible rather than just one. Note that here all three partitions are explicitly identified for each test case.

Test Case	1	2	3	4
Input (exam mark)	60	40	25	15
Input (c/w mark)	20	15	10	8
total mark (as calculated)	80	55	35	23
Partition (of exam mark)	$0 \le e \le 75$	$0 \le e \le 75$	$0 \le e \le 75$	$0 \le e \le 75$
Partition (of c/w mark)	$0 \le c \le 25$	$0 \le c \le 25$	$0 \le c \le 25$	$0 \le c \le 25$
Partition (of total mark)	$70 \le t \le 100$	$50 \le t < 70$	$30 \le t < 50$	$0 \le t < 30$
Exp. Output	'A'	'B'	'C'	'D'

Test Case	5	6	7	8
Input (exam mark)	-10	93	60.5	q
Input (c/w mark)	-15	35	20.23	g
total mark (as calculated)	-25	128	80.73	-
Partition (of exam mark)	$e < 0$	$e > 75$	e = real number	e = alphabetic
Partition (of c/w mark)	$c < 0$	$c > 25$	c = real number	c = alphabetic
Partition (of total mark)	$t < 0$	$t > 100$	$70 \le t \le 100$	-
Exp. Output	'FM'	'FM'	'FM'	'FM'

Test Case	9	10	11
Input (exam mark)	-10	100	'null'
Input (c/w mark)	0	10	'null'
total mark (as calculated)	-10	110	null+null
Partition (of exam mark)	$e < 0$	$e > 75$	-
Partition (of c/w mark)	$0 \le c \le 25$	$0 \le c \le 25$	-
Partition (of total mark)	$t < 0$	$t > 100$	-
Partition (of output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'

The one-to-one and minimised approaches represent the two approaches to equivalence partitioning. The disadvantage of the one-to-one approach is that it requires more test cases and if this causes problems a more minimalist approach can be used. Normally, however, the identification of partitions is far more time consuming than the generation and execution of test cases themselves and so any savings made by reducing the size of the test case suite are relatively small compared with the overall cost of applying the technique. The disadvantage of the minimalist approach is that in the event of a test failure it can be difficult to identify the cause due to several new partitions being exercised at once. This is a debugging problem rather than a testing problem, but there is no reason to make debugging more difficult than it is already.

Some testers would say that a variable's input domain equivalence partitions must be combined with every other variables' input domains equivalence partitions. This is an extreme view as it leads to an explosion of number of tests. It hard however to argue against it on the ground of completeness.

Too many tests are a problem since deriving the expected output can be time consuming and error prone (on the other hand actual test execution and checking against the expected results can be done using tools and is therefore not a problem in itself).

1.2 Boundary Value Analysis

Introduction

Boundary Value Analysis is based on the following premise. Firstly, that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component (in the same way as in equivalence partitioning) and, secondly, that developers are prone to making errors in their treatment of the boundaries of these classes. Thus test cases are generated to exercise these boundaries.

Example

Consider a component, *generate_grading,* with the following specification:

> *The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:*
>
> *greater than or equal to 70 - 'A'*
> *greater than or equal to 50, but less than 70 - 'B'*
> *greater than or equal to 30, but less than 50 - 'C'*
> *less than 30 - 'D'*
>
> *Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.*

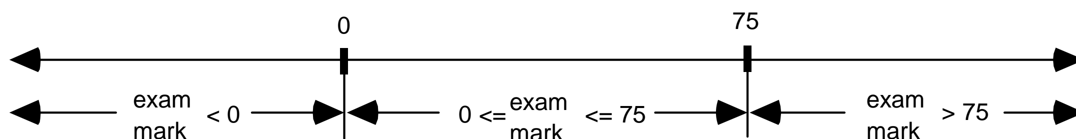
Initially the equivalence partitions are identified, then the boundaries of these partitions are identified, and then test cases are derived to exercise the boundaries. Equivalence partitions are identified from both the inputs and outputs of the component and both valid and invalid inputs and outputs are considered.

$0 \leq$ exam mark ≤ 75
$0 \leq$ coursework mark ≤ 25

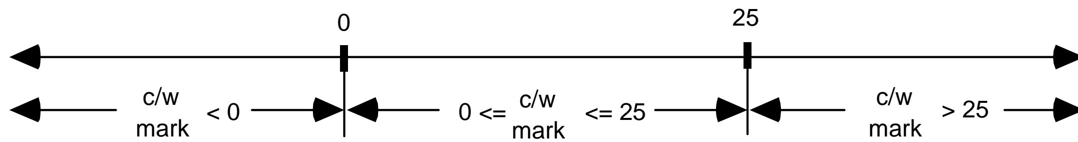
The most obvious *invalid* partitions can be described by:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0

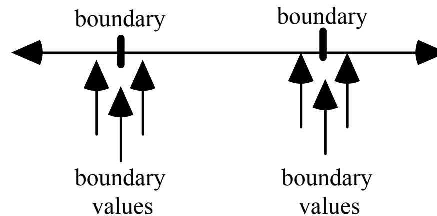
Partitioned ranges of values can be represented pictorially, therefore, for the input, exam mark, the same notation leads to:



And for the input, coursework mark, we get:



For each boundary three values are used, one on the boundary itself and one either side of it, the smallest significant distance away, as shown below:



Thus the six test cases derived from the input exam mark are:

Test Case	1	2	3	4	5	6
Input (exam mark)	-1	0	1	74	75	76
Input (c/w mark)	15	15	15	15	15	15
Boundary tested (exam mark)		0			75	
Exp. Output	'FM'	'D'	'D'	'A'	'A'	'FM'

Note that the input coursework (c/w) mark has been set to an arbitrary valid value of 15.

The test cases derived from the input coursework mark are thus:

Test Case	7	8	9	10	11	12
Input (exam mark)	40	40	40	40	40	40
Input (c/w mark)	-1	0	1	24	25	26
Boundary tested (c/w mark)		0			25	
Exp. Output	'FM'	'C'	'C'	'B'	'B'	'FM'

Note that the input exam mark has been set to an arbitrary valid value of 40.

Less obvious invalid input equivalence partitions would include any other inputs that can occur not so far included in a partition, for instance, non-integer inputs or perhaps non-numeric inputs. In order to be considered an equivalence partition those values within it must be expected, from the specification, to be treated in an equivalent manner by the component. Thus we could generate the following invalid input equivalence partitions:

exam mark = real number
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic
etc.

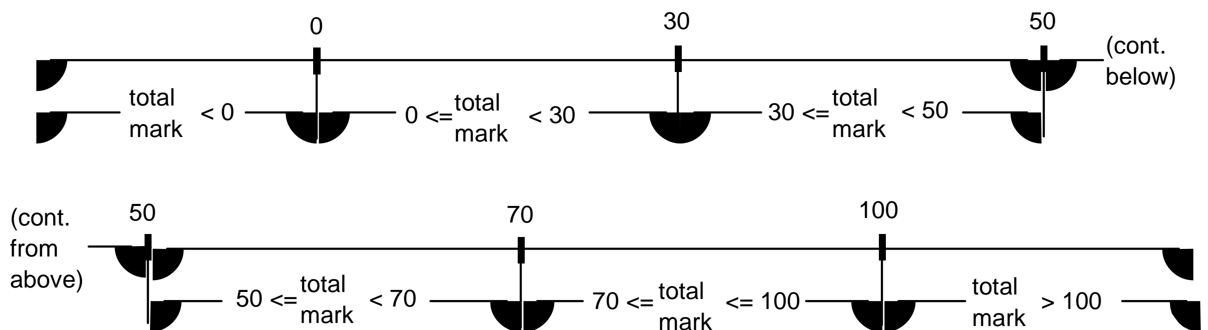
Although these are valid equivalence partitions they have no identifiable boundaries and so no test cases are derived.

Next, the partitions and boundaries for the outputs are identified. The *valid* partitions are produced by considering each of the valid outputs for the component thus:

'A'	is induced by	70 ≤ total mark ≤ 100
'B'	is induced by	50 ≤ total mark < 70
'C'	is induced by	30 ≤ total mark < 50
'D'	is induced by	0 ≤ total mark < 30
'Fault Message'	is induced by	total mark > 100
'Fault Message'	is induced by	total mark < 0

where total mark = exam mark + coursework mark.

'Fault Message' is considered here as it is a specified output. The equivalence partitions and boundaries for total mark are shown below:



Thus the test cases derived from the valid outputs are:

Test Case	13	14	15	16	17	18	19	20	21
Input (exam mark)	-1	0	0	29	15	6	24	50	26
Input (c/w mark)	0	0	1	0	15	25	25	0	25
total mark (as calculated)	-1	0	1	29	30	31	49	50	51
Boundary tested (total mark)		0			30			50	
Exp. Output	'FM'	'D'	'D'	'D'	'C'	'C'	'C'	'B'	'B'

Test Case	22	23	24	25	26	27
Input (exam mark)	49	45	71	74	75	75
Input (c/w mark)	20	25	0	25	25	26
total mark (as calculated)	69	70	71	99	100	101
Boundary tested (total mark)		70			100	
Exp. Output	'B'	'A'	'A'	'A'	'A'	'FM'

The input values of exam mark and coursework mark have been derived from the total mark, which is their sum.

An invalid output would be any output from the component other than one of the five specified. It is difficult to identify unspecified outputs, but obviously they must be considered as if we can cause one then we have identified a flaw with either the component, its specification, or both. For this example three unspecified outputs were identified ('E', 'A+', and 'null'), but it is not possible to group these

possible outputs into ordered partitions from which boundaries can be identified and so no test cases are derived.

So far several partitions have been identified that appear to be bounded on one side only. These are:

exam mark > 75
exam mark < 0
coursework mark > 25
coursework mark < 0
total mark > 100
total mark < 0

In fact these partitions are bounded on their other side by implementation-dependent maximum and minimum values. For integers held in sixteen bits these would be 32767 and -32768 respectively.

Thus, the above partitions can be more fully described by:

75 < exam mark ≤ 32767
-32768 ≤ exam mark < 0
25 < coursework mark ≤ 32767
-32768 ≤ coursework mark < 0
100 < total mark ≤ 32767
-32768 ≤ total mark < 0

It can be seen that by bounding these partitions on both sides a number of additional boundaries are identified, which must be tested. This leads to the following additional test cases:

Test Case	28	29	30	31	32	33
Input (exam mark)	32766	32767	32768	-32769	-32768	-32767
Input (c/w mark)	15	15	15	15	15	15
Boundary tested (exam mark)	32767			-32768		
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Test Case	34	35	36	37	38	39
Input (exam mark)	40	40	40	40	40	40
Input (c/w mark)	32766	32767	32768	-32769	-32768	-32767
Boundary tested (c/w mark)	32767			-32768		
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

Test Case	40	41	42	43	44	45
Input (exam mark)	16383	32767	1	0	-16384	-32768
Input (c/w mark)	16383	0	32767	-32767	-16384	-1
total mark (as calculated)	32766	32767	32768	-32767	-32768	-32769
Boundary tested (total mark)	32767			-32768		
Exp. Output	'FM'	'FM'	'FM'	'FM'	'FM'	'FM'

It should be noted that where invalid input values are used (as above, in test cases 1, 6, 7, 12, 13, and 27-45) it may, depending on the implementation, be impossible to actually execute the test case. For instance, in Ada, if the input variable is declared as a positive integer then it will not be possible to

assign a negative value to it. Despite this, it is still worthwhile *considering* all the test cases for completeness.

2 White Box Test Data Generation Techniques

2.1 Statement Testing and Coverage

Introduction

This structural test technique is based upon the decomposition of the component into constituent statements.

Example

The two principal questions to answer are:

- what is a statement?
- which statements are executable?

In general a statement should be an atomic action, that is a statement should be executed completely or not at all. For instance:

```
IF a THEN b ENDIF
```

is considered as more than one statement because `b` may or may not be executed depending upon the condition `a`. The definition of *statement* used for statement testing need not be the one used in the language definition.

We would expect statements which are associated with machine code to be regarded as executable. For instance, we would expect all of the following to be regarded as executable:

- assignments;
- loops and selections;
- procedure and function calls;
- variable declarations with explicit initialisations;
- dynamic allocation of variable storage on a heap.

However, most other variable declarations can be regarded as non executable.

Consider, the following C code:

```
a;
if (b) {
      c;
      }
d;
```

Any test case with b TRUE will achieve full statement coverage. Note that full statement coverage can be achieved without exercising with b FALSE.

2.2 Branch (AKA Decision) Testing and Coverage

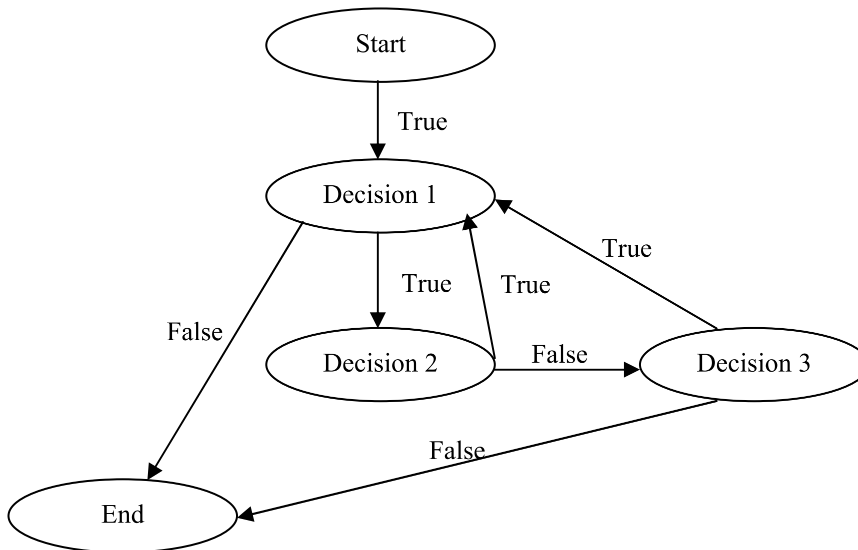
Consider the example:

> *The component shall determine the position of a word in a table of words ordered alphabetically. Apart from the word and table, the component shall also be passed the number of words in the table to be searched. The component shall return the position of the word in the table (starting at zero) if it is found, otherwise it shall return "-1".*

The corresponding code is drawn from [K&R]. The three decisions have been highlighted.

```
int binsearch (char *word, struct key tab[], int n) {
  int cond;
  int low, high, mid;
  low = 0;
  high = n - 1;
  while (low <= high) {                //Decision 1
    mid = (low+high) / 2;
    if ((cond = strcmp(word, tab[mid].word)) < 0) //Decision 2
      high = mid - 1;
    else if (cond > 0)                 //Decision 3
      low = mid + 1;
    else
      return mid;
  }
  return -1;
}
```

Branch coverage may be demonstrated through coverage of the control flow graph of the program. The easiest way is to have a node in the graph per decision (and two extra node nodes End and Start):



Apart from the Start branch which is always coverd by any test there are 6 Decisions to cover. For example whenever `n == 0`, only the Decision 1/False is covered leading to $1/6^{th}$ or 17% decision coverage. Another test could be following the path: **[Start/True, Decision 1/True, Decision 2/True, Decision 1/True, Decision 2/False, Decision 3/True, Decision 1/True, Decision 2/False, Decision 3/False]** . This path arises when the search first observes that the entry is in the first half of the table, then the second half of that (i.e., 2nd quarter) and then finds the entry.

These test cases are shown below:

test case	inputs			expected outcome
	word	tab	n	
1	chas	'empty table'	0	-1
2	chas	alf bert chas dick eddy fred geoff	7	2

Decision coverage are both normally measured using a software tool, however: **there exists no tool that can automatically generate tests to achieve any coverage measures**; hence test data generation is still a long and slow manual process.

2.3 Condition Testing and Coverage

Introduction

Branch Condition Testing, Branch Condition Combination Testing, and Modified Condition Decision Testing are closely related, as are the associated coverage measures. For convenience, these test case design and test measurement techniques are collectively referred to as condition testing.

Condition testing is based upon an analysis of the conditional control flow within the component and is therefore a form of structural testing.

Example

Consider the following fragment of code:

```
if A or (B and C) then
      do_something;
else
      do_something_else;
end if;
```

The Boolean operands within the decision condition are A, B and C. These may themselves be comprised of complex expressions involving relational operators. For example, the Boolean operand A could be an expression such as X>=Y. However, for the sake of clarity, the following examples regard A, B and C as simple Boolean operands.

2.3.1 Branch Condition Testing and Coverage

Branch Condition Coverage would require Boolean operand A to be evaluated both TRUE and FALSE, Boolean operand B to be evaluated both TRUE and FALSE, and Boolean operand C to be evaluated both TRUE and FALSE.

Branch Condition Coverage may therefore be achieved with the following set of test inputs (note that there are alternative sets of test inputs which will also achieve Branch Condition Coverage):

Case	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	TRUE	TRUE

Branch Condition Coverage can often be achieved with just two test cases, irrespective of the number of actual Boolean operands comprising the condition.

A further weakness of Branch Condition Coverage is that it can often be achieved without testing both the TRUE and FALSE branches of the decision. For example, the following alternative set of test inputs achieve Branch Condition Coverage, but only test the TRUE outcome of the overall Boolean condition. Thus Branch Condition Coverage will not necessarily subsume Branch Coverage.

Case	A	B	C
1	TRUE	FALSE	FALSE
2	FALSE	TRUE	TRUE

2.3.2 Branch Condition Combination Testing and Coverage

Branch Condition Combination Coverage would require all combinations of Boolean operands A, B and C to be evaluated. For the example condition, Branch Condition Combination Coverage can only be achieved with the following set of test inputs:

Case	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	FALSE	FALSE
3	FALSE	TRUE	FALSE
4	FALSE	FALSE	TRUE
5	TRUE	TRUE	FALSE
6	FALSE	TRUE	TRUE
7	TRUE	FALSE	TRUE
8	TRUE	TRUE	TRUE

Branch Condition Combination Coverage is very thorough, requiring 2^n test cases to achieve 100% coverage of a condition containing n Boolean operands. This rapidly becomes unachievable for more complex conditions.

2.3.3 Modified Condition Decision Testing and Coverage

Modified Condition Decision Coverage (MCDC) is a pragmatic compromise which requires fewer test cases than Branch Condition Combination Coverage. It is widely used in the development of avionics software, as required by RTCA/DO-178B.

Modified Condition Decision Coverage requires test cases to show that each Boolean operand (A, B and C) can independently affect the outcome of the decision. This is less than all the combinations (as required by Branch Condition Combination Coverage).

For the example decision condition [`A` or (`B` and `C`)], we first require a pair of test cases where changing the state of A will change the outcome, but B and C remain constant, i.e. that A can independently affect the outcome of the condition:

Case	A	B	C	Outcome
A1	FALSE	FALSE	TRUE	FALSE
A2	TRUE	FALSE	TRUE	TRUE

Similarly for B, we require a pair of test cases which show that B can independently affect the outcome, with A and C remaining constant:

Case	A	B	C	Outcome
B1	FALSE	FALSE	TRUE	FALSE
B2	FALSE	TRUE	TRUE	TRUE

Finally for C we require a pair of test cases which show that C can independently affect the outcome, with A and B remaining constant:

Case	A	B	C	Outcome
C1	FALSE	TRUE	TRUE	TRUE
C2	FALSE	TRUE	FALSE	FALSE

Having created these pairs of test cases for each operand separately, it can be seen that test cases A1 and B1 are the same, and that test cases B2 and C1 are the same. The overall set of test cases to provide 100% MCDC of the example expression is consequently:

Case	A	B	C	Outcome
1 (A1 and B1)	FALSE	FALSE	TRUE	FALSE
2 (A2)	TRUE	FALSE	TRUE	TRUE
3 (B2 and C1)	FALSE	TRUE	TRUE	TRUE
4 (C2)	FALSE	TRUE	FALSE	FALSE

In summary:

A is shown to independently affect the outcome of the decision condition by test cases 1 and 2;
B is shown to independently affect the outcome of the decision condition by test cases 1 and 3;
C is shown to independently affect the outcome of the decision condition by test cases 3 and 4.

Note that there may be alternative solutions to achieving MCDC. For example, A could have been shown to independently affect the outcome of the condition by the following pair of test cases:

Case	A	B	C	Outcome
A3	FALSE	TRUE	FALSE	FALSE
A4	TRUE	TRUE	FALSE	TRUE

Test case A3 is the same as test case C2 (or 4) above, but test case A4 is one which has not been previously used. However, as MCDC has already been achieved, test case A4 is not required for coverage purposes.

To achieve 100% Modified Condition Decision Coverage requires a minimum of n+1 test cases, and a maximum of 2n test cases, where n is the number of Boolean operands within the decision condition. In contrast, Branch Condition Combination Coverage requires n^2 test cases. MCDC is therefore a practical compromise with Branch Condition Combination Coverage where condition expressions involve more than just a few Boolean operands.

2.3.4 Final Remarks on Condition Coverage Testing

One weakness of these condition testing and test measurement techniques is that they are vulnerable to the placement of Boolean expressions which control decisions being placed outside of the actual decision condition. For example:

```
FLAG := A or (B and C);
if FLAG then
      do_something;
else
      do_something_else;
end if;
```

To combat this vulnerability, a practical variation of these condition testing and condition coverage techniques is to design tests for all Boolean expressions, not just those used directly in control flow decisions.

Some programming languages and compilers short circuit the evaluation of Boolean operators.

For example, the C and C++ languages always short circuit the Boolean "and" (&&) and "or" (||) operators, and the Ada language provides special short circuit operators **and then** and **or else**. With these examples, when the outcome of a Boolean operator can be determined from the first operand, then the second operand will not be evaluated.

The consequence is that it will be infeasible to show coverage of one value of the second operand. For a short circuited "and" operator, the feasible combinations are True:True, True:False and False:X, where X is unknown. For a short circuited "or" operator, the feasible combinations are False:False, False:True and True:X.

Other languages and compilers may short circuit the evaluation of Boolean operators in any order. In this case, the feasible combinations are not known. The degree of short circuit optimisation of Boolean operators may depend upon compiler switches or may be outside the user's control.

Short circuited control forms present no obstacle to Branch Condition Coverage or Modified Condition Decision Coverage, but they do obstruct the measurement of Branch Condition Combination Coverage. There are situations where it is possible to design test cases which should achieve 100% coverage (from a theoretical point of view), but where it is not possible to actually measure that 100% coverage has been achieved.