UNIT-4

Selection, Minimization and Prioritization of Test Cases for Regression Testing

WHAT IS REGRESSION TESTING?

When we develop software, we use development testing to obtain confidence in the correctness of the software. Development testing involves constructing a test plan that describes how we should test the software and then, designing and running a suite of test cases that satisfy the requirements of the test plan. When we modify software, we typically re-test it. This process of re-testing is called regression testing.

Hence, regression testing is the process of re-testing the modified parts of the software and ensuring that no new errors have been introduced into previously tested source code due to these modifications. Therefore, regression testing tests both the modified source code and other parts of the source code that may be affected by the change. It serves several purposes like:

- Increases confidence in the correctness of the modified program.
- Locates errors in the modified program.
- Preserves the quality and reliability of the software.
- Ensures the software's continued operation.

We typically think of regression testing as a software maintenance activity; however, we also perform regression testing during the latter stage of software development. This latter stage starts after we have developed test plans and test suites and used them initially to test the software. During this stage of development, we fine-tune the source code and correct errors in it, hence these activities resemble maintenance activities. The comparison of development testing and regression testing is given in Table 7.1.

Table 7.1. Comparison of regression and development testing

S.No.	Development Testing	Regression Testing
1.	We write test cases.	We may use already available test cases.
2.	We want to test all portions of the source code.	We want to test only modified portion of the source code and the portion affected by the modifications.
3.	We do development testing just once in the lifetime of the software.	We may have to do regression testing many times in the lifetime of the software.
4.	We do development testing to obtain confidence about the correctness of the software.	We do regression testing to obtain confidence about the correctness of the modified portion of the software.
5.	Performed under the pressure of release date.	Performed in crisis situations, under greater time constraints.
6.	Separate allocation of budget and time.	Practically no time and generally no separate budget allocation.
7.	Focus is on the whole software with the objective of finding faults.	Focus is only on the modified portion and other affected portions with the objective of ensuring the correctness of the modifications.
8.	Time and effort consuming activity (40% to 70%).	Not much time and effort is consumed as compared to development testing.

• Regression Testing Process

Regression testing is a very costly process and consumes a significant amount of resources. The question is "how to reduce this cost?" Whenever a failure is experienced, it is reported to the software team. The team may like to debug the source code to know the reason(s) for this

failure. After identification of the reason(s), the source code is modified and we generally do not expect the same failure again. In order to ensure this correctness, we re-test the source code with a focus on modified portion(s) of the source code and also on affected portion(s) of the source code due to modifications. We need test cases that target the modified and affected portions of the source code. We may write new test cases, which may be a 'time and effort consuming' activity. We neither have enough time nor reasonable resources to write new test cases for every failure. Another option is to use the existing test cases which were designed for development testing and some of them might have been used during development testing. The existing test suite may be useful and may reduce the cost of regression testing. As we all know, the size of the existing test suite may be very large and it may not be possible to execute all tests. The greatest challenge is to reduce the size of the existing test suite for a particular failure. The various steps are shown in Figure 7.1. Hence, test case selection for a failure is the main key for regression testing.

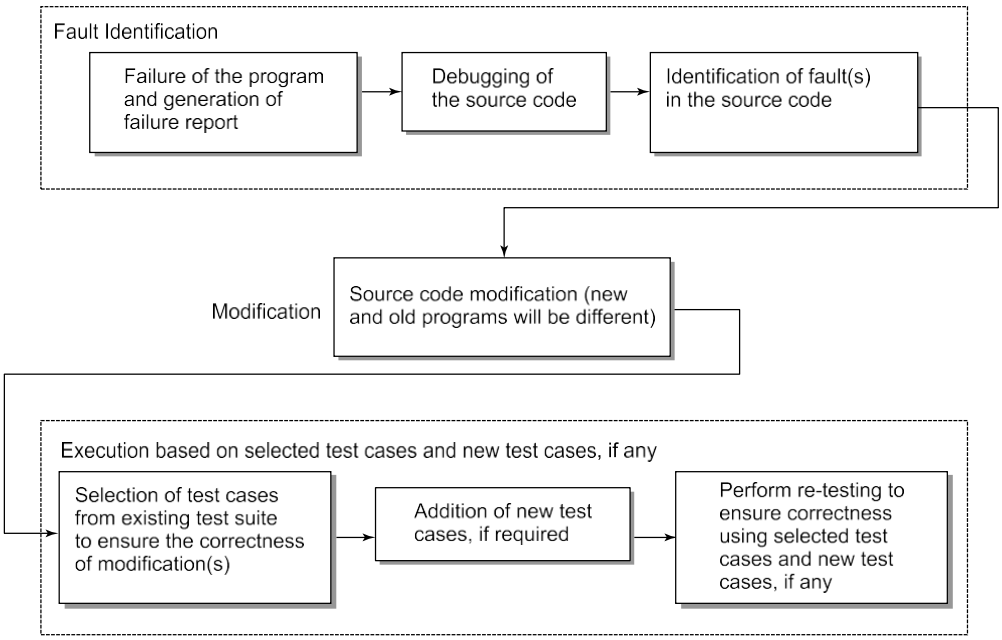

Figure 7.1. Steps of regression testing process

- ## Selection of Test Cases

We want to use the existing test suite for regression testing. How should we select an appropriate number of test cases for a failure? The range is from "one test case" to "all test cases". A 'regression test cases' selection technique may help us to do this selection process. The effectiveness of the selection technique may decide the selection of the most appropriate test cases from the test suite. Many techniques have been developed for procedural and object oriented programming languages. Testing professionals are, however, reluctant to omit any test case from a test suite that might expose

a fault in the modified program. We consider a program given in Figure 7.2 along with its modified version where the modification is in line 6 (replacing operator '*' by '-'). A test suite is also given in Table 7.2.

```
1.  main( )                          1.  main ( )
2.  {                                2.  {
3.  int a, b, x, y, z;               3.  int a, b, x, y, z;
4.  scanf ("%d, %d", &a, &b);        4.  scanf ("%d, %d", &a, &b);
5.  x = a + b ;                      5.  x = a + b;
6.  y = a* b;                        6.  y = a – b;
7.  if (x ≥ y) {                     7.  if (x ≥ y) {
8.  z = x / y ;                      8.  z = x / y ;
9.  }                                9.  }
10. else {                           10. else {
11. z = x * y ;                      11. z = x * y ;
12. }                                12. }
13. printf ("z = %d \ n", z );       13. printf ("z = %d \ n", z);
14. }                                14. }
```

(a) Original program with fault in line 6. (b) Modified program with modification in line 6.

Figure 7.2. Program for printing value of *z*

Table 7.2. Test suite for program given in Figure 7.2

Set of Test Cases S. No.	Inputs		Execution History
	a	b	
1	2	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
2	1	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
3	3	2	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14
4	3	3	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14

In this case, the modified line is line number 6 where 'a*b' is replaced by 'a-b'. All four test cases of the test suite execute this modified line 6. We may decide to execute all four tests for the modified program. If we do so, test case 2 with inputs a = 1 and b = 1 will experience a 'divide by zero' problem, whereas others will not. However, we may like to reduce the number of test cases for the modified program. We may select all test cases which are executing the modified line. Here, line number 6 is modified. All four test cases are executing the modified line (line number 6) and hence are selected. There is no reduction in terms of the number of test cases. If we see the execution history, we find that test case 1 and test case 2 have the same execution history. Similarly, test case 3 and test case 4 have the same execution history. We choose any one test case of the same execution history to avoid repetition. For execution history 1 (i.e. 1, 2, 3, 4, 5, 6, 7, 8, 10, 11), if we select test case 1, the program will execute well, but if we select test case 2, the program will experience 'divide by zero' problem. If several test cases execute a particular modified line, and all of these test cases reach a particular

affected source code segment, minimization methods require selection of only one such test case, unless they select the others for coverage elsewhere. Therefore, either test case 1 or test case 2 may have to be selected. If we select test case 1, we miss the opportunity to detect the fault that test case 2 detects. Minimization techniques may omit some test cases that might expose fault(s) in the modified program. Hence, we should be very careful in the process of minimization of test cases and always try to use safe regression test selection technique (if at all it is possible). A safe regression test selection technique should select all test cases that can expose faults in the modified program.

REGRESSION TEST CASES SELECTION

Test suite design is an expensive process and its size can grow quite large. Most of the times, running an entire test suite is not possible as it requires a significant amount of time to run all test cases. Many techniques are available for the selection of test cases for the purpose of regression testing.

- **Select All Test Cases**

This is the simplest technique where we do not want to take any risk. We want to run all test cases for any change in the program. This is the safest technique, without any risk. A program may fail many times and every time we will execute the entire test suite. This technique is practical only when the size of the test suite is small. For any reasonable or large sized test suite, it becomes impractical to execute all test cases.

- **Select Test Cases Randomly**

We may select test cases randomly to reduce the size of the test suite. We decide how many test cases are required to be selected depending upon time and available resources. When we decide the number, the same number of test cases is selected randomly. If the number is large, we may get a good number of test cases for execution and testing may be of some use. But, if the number is small, testing may not be useful at all. In this technique, our assumption is that all test cases are equally good in their fault detection ability. However, in most of the situations, this assumption may not be true. We want to re-test the source code for the purpose of checking the correctness of the modified portion of the program. Many randomly selected test cases may not have any relationship with the modified portion of the program. However, random selection may be better than no regression testing at all.

- **Select Modification Traversing Test Cases**

We select only those test cases that execute the modified portion of the program and the portion which is affected by the modification(s). Other test cases of the test suite are discarded.

Actually, we want to select all those test cases that reveal faults in the modified program. These test cases are known as fault revealing test cases. There is no effective technique by which we can find fault revealing test cases for the modified program. This is the best selection approach, which we want, but we do not have techniques for the same. Another lower objective

may be to select those test cases that reveal the difference in the output of the original program and the modified program. These test cases are known as modification revealing test cases. These test cases target that portion of the source code which makes the output of the original program and the modified program differ. Unfortunately, we do not have any effective technique to do this. Therefore, it is difficult to find fault revealing test cases and modification revealing test cases.

The reasonable objective is to select all those test cases that traverse the modified source code and the source code affected by modification(s). These test cases are known as modification traversing test cases. It is easy to develop techniques for modification traversing test cases and some are available too. Out of all modification traversing test cases, some may be modification revealing test cases and out of some modification revealing test cases, some may be fault revealing test cases. Many modification traversing techniques are available but their applications are limited due to the varied nature of software projects. Aditya Mathur has rightly mentioned that [MATH08]:

> *"The sophistication of techniques to select modification traversing tests requires automation. It is impractical to apply these techniques to large commercial systems unless a tool is available that incorporates at least one safe test minimization technique. Further, while test selection appears attractive from the test effort point of view, it might not be a practical technique when tests are dependent on each other in complex ways and that this dependency cannot be incorporated in the test selection tool".*

We may effectively implement any test case selection technique with the help of a testing tool. The modified source code and source code affected by modification(s) may have to be identified systematically and this selected area of the source code becomes the concern of test case selection. As the size of the source code increases, the complexity also increases, and need for an efficient technique also increases accordingly.

REDUCING THE NUMBER OF TEST CASES

Test case reduction is an essential activity and we may select those test cases that execute the modification(s) and the portion of the program that is affected by the modification(s). We may minimize the test suite or prioritize the test suite in order to execute the selected number of test cases.

- ### Minimization of Test Cases

We select all those test cases that traverse the modified portion of the program and the portion that is affected by the modification(s). If we find the selected number very large, we may still reduce this using any test case minimization technique. These test case minimization techniques attempt to find redundant test cases. A redundant test case is one which achieves an objective which has already been achieved by another test case. The objective may be source code coverage, requirement coverage, variables coverage, branch coverage, specific lines of source

code coverage, etc. A minimization technique may further reduce the size of the selected test cases based on some criteria. We should always remember that any type of minimization is risky and may omit some fault revealing test cases.

- **Prioritization of Test Cases**

We may indicate the order with which a test case may be addressed. This process is known as prioritization of test cases. A test case with the highest rank has the highest priority and the test case with the second highest rank has the second highest priority and as so on. Prioritization does not discard any test case. The efficiency of the regression testing is dependent upon the criteria of prioritization. There are two varieties of test case prioritization i.e. general test case prioritization and version specific test case prioritization. In general test case prioritization, for a given program with its test suite, we prioritize the test cases that will be useful over a succession of subsequent modified versions of the original program without any knowledge of modification(s). In the version specific test case prioritization, we prioritize the test cases, when the original program is changed to the modified program, with the knowledge of the changes that have been made in the original program.
Prioritization guidelines should address two fundamental issues like:

(i) What functions of the software must be tested?
(ii) What are the consequences if some functions are not tested?

Every reduction activity has an associated risk. All prioritization guidelines should be designed on the basis of risk analysis. All risky functions should be tested on higher priority. The risk analysis may be based on complexity, criticality, impact of failure, etc. The most important is the 'impact of failure' which may range from 'no impact' to 'loss of human life' and must be studied very carefully.

The simplest priority category scheme is to assign a priority code to every test case. The priority code may be based on the assumption that "test case of priority code 1 is more important than test case of priority code 2." We may have priority codes as follows:

Priority code 1	:	Essential test case
Priority code 2	:	Important test case
Priority code 3	:	Execute, if time permits
Priority code 4	:	Not important test case
Priority code 5	:	Redundant test case

There may be other ways for assigning priorities based on customer requirements or market conditions like:

Priority code 1	:	Important for the customer
Priority code 2	:	Required to increase customer satisfaction
Priority code 3	:	Help to increase market share of the product

We may design any priority category scheme, but a scheme based on technical considerations always improves the quality of the product and should always be encouraged.

RISK ANALYSIS

Unexpected behaviours of a software programme always carry huge information and most of the time they disturb every associate person. No one likes such unexpected behaviour and everyone prays that they never face these situations in their professional career. In practice, the situation is entirely different and developers do face such unexpected situations frequently and, moreover, work hard to find the solutions of the problems highlighted by these unexpected behaviours.

We may be able to minimize these situations, if we are able to minimize the risky areas of the software. Hence, risk analysis has become an important area and in most of the projects we are doing it to minimize the risk.

• What is Risk?

Tomorrow's problems are today's risks. Therefore, a simple definition of risk is a problem that may cause some loss or threaten the success of the project, but, which has not happened yet. Risk is defined as the "probability of occurrence of an undesirable event and the impact of occurrence of that event." To understand whether an event is really risky needs an understanding of the potential consequences of the occurrences / non-occurrences of that event. Risks may delay and over-budget a project. Risky projects may also not meet specified quality levels. Hence, there are two things associated with risk as given below:

(i) Probability of occurrence of a problem (i.e. an event)
(ii) Impact of that problem

Risk analysis is a process of identifying the potential problems and then assigning a 'probability of occurrence of the problem' value and 'impact of that problem' value for each identified problem. Both of these values are assigned on a scale of 1 (low) to 10 (high). A factor 'risk exposure' is calculated for every problem which is the product of 'probability of occurrence of the problem' value and 'impact of that problem' value. The risks may be ranked on the basis of its risk exposure. A risk analysis table may be prepared as given in Table 7.3. These values may be calculated on the basis of historical data, past experience, intuition and criticality of the problem. We should not confuse with the mathematical scale of probability values which is from 0 to 1. Here, the scale of 1 to 10 is used for assigning values to both the components of the risk exposure.

Table 7.3. Risk analysis table

S. No.	Potential Problem	Probability of occurrence of problem	Impact of that Problem	Risk Exposure
1.				
2.				
3.				
4.				

The case study of 'University Registration System' given in chapter 5 is considered and its potential problems are identified. Risk exposure factor for every problem is calculated on the

basis of 'probability of occurrence of the problem' and 'impact of that problem'. The risk analysis is given in Table 7.4.

Table 7.4. Risk analysis table of 'University Registration System'

S. No.	Potential Problems	Probability of occurrence of problem	Impact of that Problem	Risk Exposure
1.	Issued password not available	2	3	6
2.	Wrong entry in students detail form	6	2	12
3.	Wrong entry in scheme detail form	3	3	9
4.	Printing mistake in registration card	2	2	4
5.	Unauthorised access	1	10	10
6.	Database corrupted	2	9	18
7.	Ambiguous documentation	8	1	8
8.	Lists not in proper format	3	2	6
9.	Issued login-id is not in specified format	2	1	2
10.	School not available in the database	2	4	8

The potential problems ranked by risk exposure are 6, 2, 5, 3, 7, 10, 1, 8, 4 and 9.

- **Risk Matrix**

Risk matrix is used to capture identified problems, estimate their probability of occurrence with impact and rank the risks based on this information. We may use the risk matrix to assign thresholds that group the potential problems into priority categories. The risk matrix is shown in Figure 7.3 with four quadrants. Each quadrant represents a priority category.

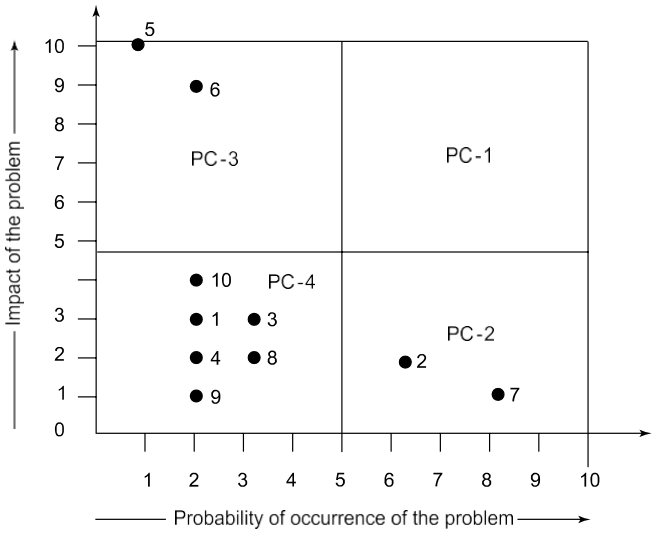

Figure 7.3. Threshold by quadrant

The priority category in defined as:

Priority category 1 (PC-1)	=	High probability value and high impact value
Priority category 2 (PC-2)	=	High probability value and low impact value
Priority category 3 (PC-3)	=	Low probability value and high impact value
Priority category 4 (PC-4)	=	Low probability value and low impact value

In this case, a risk with high probability value is given more importance than a problem with high impact value. We may change this and may decide to give more importance to high impact value over the high probability value and is shown in Figure 7.4. Hence, PC-2 and PC-3 will swap, but PC-1 and PC-4 will remain the same.

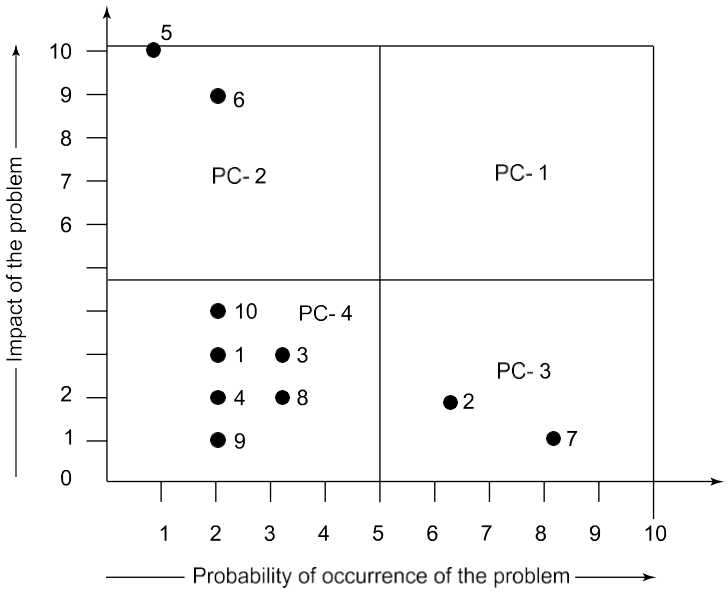

Figure 7.4. Alternative threshold by quadrant

There may be situations where we do not want to give importance to any value and assign equal importance. In this case, the diagonal band prioritization scheme may be more suitable as shown in Figure 7.5. This scheme is more appropriate in situations where we have difficulty in assigning importance to either 'probability of occurrence of the problem' value or 'impact of that problem' value.

We may also feel that high impact value must be given highest priority irrespective of the 'probability of occurrence' value. A high impact problem should be addressed first, irrespective of its probability of occurrence value. This prioritization scheme is given in Figure 7.6. Here, the highest priority (PC-1) is assigned to high impact value and for the other four quadrants; any prioritization scheme may be selected. We may also assign high priority to high 'probability of occurrence' values irrespective of the impact value as shown in Figure 7.7. This scheme may not be popular in practice. Generally, we are afraid of the impact of the problem. If the impact value is low, we are not much concerned. In the risk analysis table (see Table 7.4), ambiguous documentations (S. No. 7) have high 'probability of occurrence of problem' value (8), but

impact value is very low (1). Hence, these faults are not considered risky faults as compared to 'unauthorized access' (S. No. 5) where 'probability of occurrence' value is very low (1) and impact value is very high (10).

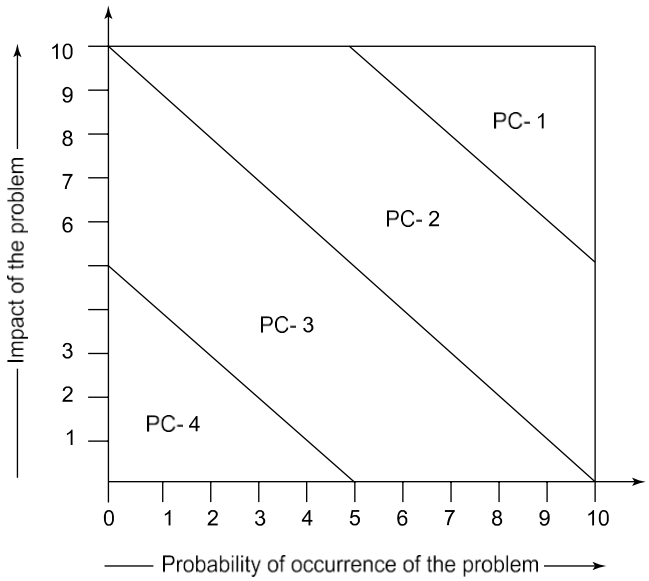

Figure 7.5. Threshold by diagonal quadrant

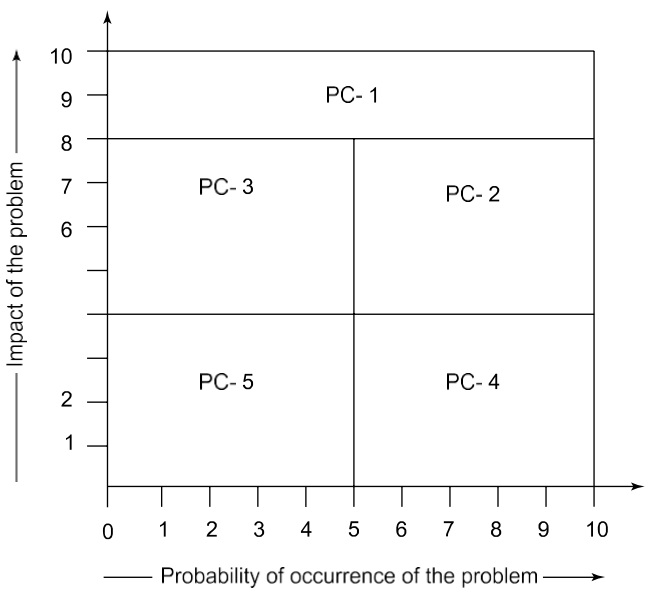

Figure 7.6. Threshold based on high 'Impact of Problem' value

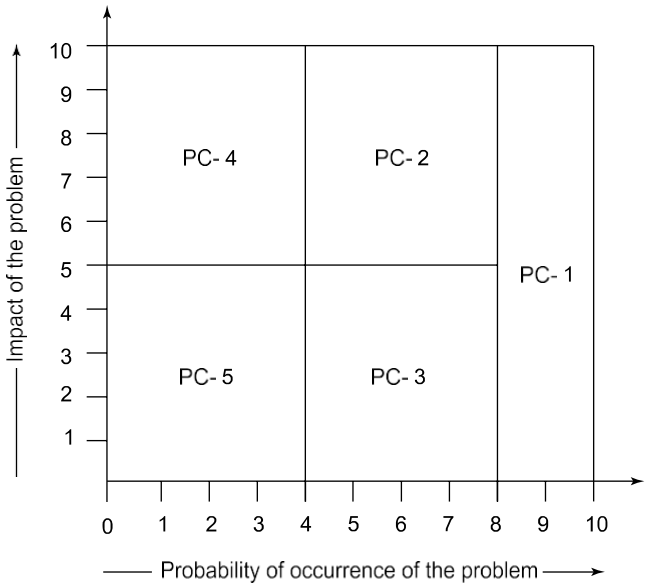

Figure 7.7. Threshold based on high 'probability of occurrence of problem' value

After the risks are ranked, the high priority risks are identified. These risks are required to be managed first and then other priority risks in descending order. These risks should be discussed in a team and proper action should be recommended to manage these risks. A risk matrix has become a powerful tool for designing prioritization schemes. Estimating the probability of occurrence of a problem may be difficult in practice. Fortunately, all that matters when using a risk matrix is the relative order of the probability estimates (which risks are more likely to occur) on the scale of 1 to 10. The impact of the problem may be critical, serious, moderate, minor or negligible. These two values are essential for risk exposure which is used to prioritize the risks.

CODE COVERAGE PRIORITIZATION TECHNIQUE

We consider a program P with its modified program P′ and its test suite T created to test P. When we modify P to P′, we would like to execute modified portion(s) of the source code and the portion(s) affected by the modification(s) to see the correctness of modification(s). We neither have time nor resources to execute all test cases of T. Our objective is to reduce the size of T to T′ using some selection criteria, which may help us to execute the modified portion of the source code and the portion(s) affected by modification(s).

A code coverage based technique [KAUR06, AGGA04] has been developed which is based on version specific test case prioritization and selects T′ from T which is a subset of T. The technique also prioritizes test cases of T′ and recommends use of high priority test cases first and then low priority test cases in descending order till time and resources are available or a reasonable level of confidence is achieved.

- **Test Cases Selection Criteria**

The technique is based on version specific test case prioritization where information about changes in the program is known. Hence, prioritization is focused around the changes in the modified program. We may like to execute all modified lines of source code with a minimum number of selected test cases. This technique identifies those test cases that:

(i) Execute the modified lines of source code at least once
(ii) Execute the lines of source code after deletion of deleted lines from the execution history of the test case and are not redundant.

The technique uses two algorithms – one for 'modification' and the other for 'deletion'. The following information is available with us and has been used to design the technique:

(i) Program P with its modified program P′.
(ii) Test suite T with test cases t1, t2, t3,…..tn.
(iii) Execution history (number of lines of source code covered by a test case) of each test case of test suite T.
(iv) Line numbers of lines of source code covered by each test case are stored in a two dimensional array (t_{11}, t_{12}, t_{13},……t_{ij}).

- **Modification Algorithm**

The 'modification' portion of the technique is used to minimize and prioritize test cases based on the modified lines of source code. The 'modification' algorithm uses the following information given in Table 7.5.

Table 7.5. Variables used by 'modification' algorithm

S. No.	Variable name	Description
1.	T1	It is a two dimensional array and is used to store line numbers of lines of source code covered by each test case.
2.	modloc	It is used to store the total number of modified lines of source code.
3.	mod_locode	It is a one-dimensional array and is used to store line numbers of modified lines of source code.
4.	nfound	It is a one-dimensional array and is used to store the number of lines of source code matched with modified lines of each test case.
5.	pos	It is a one-dimensional array and is used to set the position of each test case when nfound is sorted.
6.	candidate	It is a one-dimensional array. It sets the bit to 1 corresponding to the position of the test case to be removed.
7.	priority	It is a one-dimensional array and is used to set the priority of the selected test case.

The following steps have been followed in order to select and prioritize test cases from test suite T based on the modification in the program P.

Step I: Initialization of variables

Consider a hypothetical program of 60 lines of code with a test suite of 10 test cases. The execution history is given in Table 7.6. We assume that lines 1, 2, 5, 15, 35, 45, 55 are modified.

Table 7.6. Test cases with execution history

Test case Id	Execution history
T1	1, 2, 20, 30, 40, 50
T2	1, 3, 4, 21, 31, 41, 51
T3	5, 6, 7, 8, 22, 32, 42, 52
T4	6, 9, 10, 23, 24, 33, 43, 54
T5	5, 9, 11, 12, 13, 14, 15, 20, 29, 37, 38, 39
T6	15, 16, 17, 18, 19, 23, 24, 25, 34, 35, 36
T7	26, 27, 28, 40, 41, 44, 45, 46
T8	46, 47, 48, 49, 50, 53, 55
T9	55, 56, 57, 58, 59
T10	3, 4, 60

The first portion of the 'modification' algorithm is used to initialize and read values of variables T1, modloc and mod_locode.

First portion of the 'modification' algorithm

1. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) Initialize array T1[i][j] to zero
2. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) Store line numbers of line of source code covered by each test case.
3. Repeat for i=1 to number of modified lines of source code
   (a) Store line numbers of modified lines of source code in array mod_locode.

Step II: Selection and prioritization of test cases

The second portion of the algorithm counts the number of modified lines of source code covered by each test case (nfound).

Second portion of the 'modification' algorithm

2. Repeat for all true cases
   (a) Repeat for i=1 to number of test cases
      (i) Initialize array nfound[i] to zeroes
      (ii) Set pos[i] =i
   (b) Repeat for i=1 to number of test cases
      (i) Initialize l to zero

```
(ii) Repeat for j=1 to length of the test case
        If candidate[i] ≠ 1 then
        Repeat for k=1 to modified lines of source code
            If t1[i][j]=mod_locode[k] then
                Increment nfound[i] by one
                Increment l by one
```

The status of test cases covering modified lines of source code is given in Table 7.7.

Table 7.7. Test cases with number of matches found

Test Cases	Numbers of lines matched	Number of Matches (nfound)
T1	1, 2	2
T2	1	1
T3	5	1
T4	-	0
T5	5, 15	2
T6	15, 35	2
T7	45	1
T8	55	1
T9	55	1
T10	-	0

Consider the third portion of 'modification' algorithm. In this portion, we sort the nfound array and select the test case with the highest value of nfound as a candidate for selection. The test cases are arranged in increasing order of priority.

Third portion of the '"modification' algorithm

```
(c) Initialize l to zero
(d) Repeat for i=0 to number of test cases
    (i) Repeat for j=1 to number of test cases
            If nfound[i]>0 then
                t=nfound[i]
                nfound[i]=nfound[j]
                nfound[j]=t
                t=pos[i]
                pos[i]=pos[j]
                pos[j]=t
(e) Repeat for i=1 to number of test cases
    (i) If nfound[i]=1 then
            Increment count
(f) If count = 0 then
    (i) Goto end of the algorithm
(g) Initialize candidate[pos[0]] = 1
(h) Initialize priority[pos[0]]= m+1
```

The test cases with less value have higher priority than the test cases with higher value. Hence, the test cases are sorted on the basis of number of modified lines covered as shown in Table 7.8.

Table 7.8. Test cases in decreasing order of number of modified lines covered

Test Cases	Numbers of lines matched	Number of Matches (nfound)	Candidate	Priority
T1	1, 2	2	1	1
T5	5, 15	2	0	0
T6	15, 35	2	0	0
T2	1	1	0	0
T3	5	1	0	0
T7	45	1	0	0
T8	55	1	0	0
T9	55	1	0	0
T4	-	0	0	0
T10	-	0	0	0

The test case with candidate=1 is selected in each iteration. In the fourth portion of the algorithm, the modified lines of source code included in the selected test case are removed from the mod_locode array. This process continues until there are no remaining modified lines of source code covered by any test case.

Fourth portion of the 'modification' algorithm

```
(a) Repeat for i=1 to length of selected test cases
    (i) Repeat for j=1 to modified lines of source code
            If t1[pos[0]][i] = mod[j] then
                mod[j] = 0
```

Since test case T1 is selected and it covers 1 and 2 lines of source code, these lines will be removed from the mod_locode array.

mod_locode = [1, 2, 5, 15, 35, 45, 55] - [1, 2] = [5, 15, 35, 45, 55]

The remaining iterations of the 'modification' algorithm are shown in tables 7.9-7.12.

Table 7.9. Test cases in descending order of number of matches found (iteration 2)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T5	2	5, 15	1	2
T6	2	15, 35	0	0
T3	1	5	0	0
T7	1	45	0	0
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [5, 15, 35, 45, 55] – [5, 15] = [35, 45, 55]

Table 7.10. Test cases in descending order of number of matches found (iteration 3)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T6	1	35	1	3
T7	1	45	0	0
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [35, 45, 55] – [35] = [45, 55]

Table 7.11. Test cases in descending order of number of matches found (iteration 4)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T7	1	45	1	4
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [45, 55] – [45] = [55]

Table 7.12. Test cases in descending order of number of matches found (iteration 5)

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T8	1	55	1	5
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [55] – [55] = [Nil]

Hence test cases T1, T5, T6, T7 and T8 need to be executed on the basis of their corresponding priority. Out of ten test cases, we need to run only 5 test cases for 100% code coverage of modified lines of source code. This is 50% reduction of test cases.

- **Deletion Algorithm**

The 'deletion' portion of the technique is used to (i) update the execution history of test cases by removing the deleted lines of source code (ii) identify and remove those test cases that cover only those lines which are covered by other test cases of the program. The information used in the algorithm is given in Table 7.13.

Table 7.13. Variables used by 'deletion' algorithm

S. No.	Variable	Description
1.	T1	It is a two-dimensional array. It keeps the number of lines of source code covered by each test case i.
2.	deloc	It is used to store the total number of lines of source code deleted.
3.	del_locode	It is a one-dimensional array and is used to store line numbers of deleted lines of source code.
4.	count	It is a two-dimensional array. It sets the position corresponding to every matched line of source code of each test case to 1.
5.	match	It is a one-dimensional array. It stores the total count of the number of 1's in the count array for each test case.
6.	deleted	It is a one-dimensional array. It keeps the record of redundant test cases. If the value corresponding to test case i is 1 in deleted array, then that test case is redundant and should be removed.

Step I: Initialization of variables

We consider a hypothetical program of 20 lines with a test suite of 5 test cases. The execution history is given in Table 7.14.

Table 7.14. Test cases with execution history

Test case Id	Execution history
T1	1, 5, 7, 15, 20
T2	2, 3, 4, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19
T5	1, 2, 6, 8, 9, 13

We assume that line numbers 6, 13, 17 and 20 are modified, and line numbers 4, 7 and 15 are deleted from the source code. The information is stored as:

delloc = 3
del_locode = [4, 7, 15]
modloc = 4
mod_locode = [6, 13, 17, 20]

First portion of the "deletion" algorithm

1. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to length of test case i
      (i) Repeat for l to number of deleted lines of source code
         If T1[i][j]=del_locode then
            Repeat for k=j to length of test case i
              T1[i][k]=T1[i][k+1]
            Initialize T1[i][k] to zero
            Decrement c[i] by one

After deleting line numbers 4, 7, and 15, the modified execution history is given in Table 7.15.

Table 7.15. Modified execution history after deleting line numbers 4, 7 and 15

Test case Id	Execution history
T1	1, 5, 20
T2	2, 3, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19
T5	1, 2, 6, 8, 9, 13

Step II: Identification of redundant test cases

We want to find redundant test cases. A test case is a redundant test case, if it covers only those lines which are covered by other test cases of the program. This situation may arise due to deletion of a few lines of the program.

Consider the second portion of the 'deletion' algorithm. In this portion, the test case array is initialized with line numbers of lines of source code covered by each test case.

Second portion of the 'deletion' algorithm

2. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) Initialize array t1[i][j] to zero
      (ii) Initialize array count[i][j] to zero
3. Repeat for i=1 to number of test cases
   (a) Initialize deleted[i] and match [i] to zero
4. Repeat for i=1 to number of test cases
   (a) Initialize c[i] to number of line numbers in each test case i
   (b) Repeat for j=1 to c[i]
   (c) Initialize t1[i][j] to line numbers of line of source code covered by each test case

The third portion of the algorithm compares lines covered by each test case with lines covered by other test cases. A two-dimensional array count is used to keep the record of line number matched in each test case. If all the lines covered by a test case are being covered by some other test case, then that test case is redundant and should not be selected for execution.

Third portion of the 'deletion' algorithm

5. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) If i≠j and deleted[j] ≠1 then
            Repeat for k=1 to until t1[i][k]≠0
              Repeat for l=1 until t1[j][l]≠0
            If t1[i][k]=t1[j][l] then
                Initialize count [i][k]=1
   (b) Repeat for m=1 to c[i]
      (i) If count[i][m]=1 then
            Increment match[i] with 1
   (c) If match[i]=c[i] then
      (i) Initialize deleted[i] to 1
6. Repeat for i=1 to number of test cases
   (a) If deleted[i] =1 then
      Remove test case i (as it is a redundant test case)

On comparing all values in each test case with all values of other test cases, we found that test case 1 and test case 5 are redundant test cases. These two test cases do not cover any line which is not covered by other test cases as shown in Table 7.16.

Table 7.16.

Test Case	Line Number of LOC	Found In Test Case	Redundant Y/N
T1	1	T4	Y
	5	T2	Y
	20	T2	Y
T5	6	T3	Y
	8	T3	Y
	9	T3	Y
	1	T4	Y
	2	T2	Y
	13	T3	Y

The remaining test cases are = [T2, T3, T4] and are given in Table 7.17.

Table 7.17. Modified table after removing T1 and T5

Test case Id	Execution history
T2	2, 3, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19

Now we will minimize and prioritize test cases using 'modification' algorithm given in section 7.5.2. The status of test cases covering the modified lines is given in Table 7.18.

Table 7.18. Test cases with modified lines

Test Cases	Number of lines matched (found)	Number of matches (nfound)
T2	20	1
T3	6, 13, 17	3
T4	17	1

Test cases are sorted on the basis of number of modified lines covered as shown in tables 7.19-7.20.

Table 7.19. Test cases in descending order of number of modified lines covered

Test Cases	Number of matches (nfound)	Numbers of lines matched	Candidate	Priority
T3	3	6, 13, 17	1	1
T2	1	20	0	0
T4	1	17	0	0

mod_locode = [6, 13, 17, 20] – [6, 13, 17] = [20]

Table 7.20. Test cases in descending order of number of modified lines covered (iteration 2)

Test Cases	Number of matches (nfound)	Numbers of lines matched	Candidate	Priority
T2	1	20	1	2
T4	0	-	0	0

Hence, test cases T2 and T3 are needed to be executed and redundant test cases are T1 and T5.

Out of the five test cases, we need to run only 2 test cases for 100% code coverage of modified code coverage. This is a 60% reduction. If we run only those test cases that cover any modified lines, then T2, T3 and T4 are selected. This technique not only selects test cases, but also prioritizes test cases.