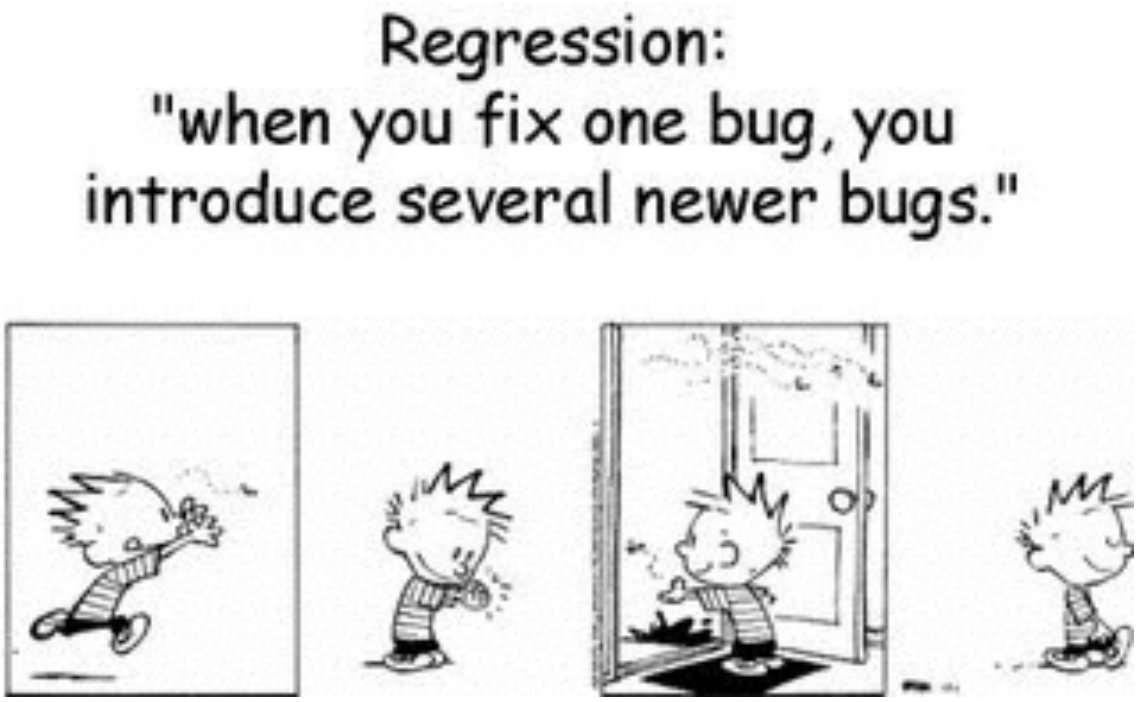
EECE 416
Software Testing and Analysis

Lecture 18
Regression Testing

REGRESSION TESTING

REGRESSION TESTING



REGRESSION TESTING

More formally:

Regression testing is the execution of a *set of test cases* on a program in order to ensure that its revision does not inject faults, does not "regress" - that is, become less effective than it has been in the past.

- Test maintenance (testing new features, repairing, regression testing)
- Regression testing
  - Test Selection
  - Test Minimization
  - Test Prioritization
  - Test Augmentation
- Test Repair and Evolution

Learning Objectives

- **Importance** and **challenges** of regression testing
- Test suite **maintenance**
- Regression testing techniques
  - Test Selection
  - Test Prioritization
  - Test Augmentation
  - Test Minimization

Software Evolution

Corrective Maintenance: changes made to debug a system after a failure is observe

Adaptive Maintenance: changes made to achieve continuing compatibility with the target environment

Perfective Maintenance: changes designed/made to improve or add capabilities

Preventive Maintenance: changes made to improve quality of software

Which ones require regression testing?

Corrective Maintenance: changes made to debug a system after a failure is observe

Adaptive Maintenance: changes made to achieve continuing compatibility with the target environment

Perfective Maintenance: changes designed/made to improve or add capabilities

Preventive Maintenance: changes made to improve quality of software

Which ones require regression testing?

RT is required for each of these: to reveal *side effects* and *bad fixes (2-20%)*

Corrective Maintenance: changes made to debug a system after a failure is observe

Adaptive Maintenance: changes made to achieve continuing compatibility with the target environment

Perfective Maintenance: changes designed/made to improve or add capabilities

Preventive Maintenance: changes made to improve quality of software

Why does a test fail after a change?

Why does a test fail after a change?

- Specs and code have changed
  - Test suite does not reflect these **valid** changes (false alarms)
- There is a new regression ***bug***

Which one is more important?

- bug in new features
- regression bugs in existing features

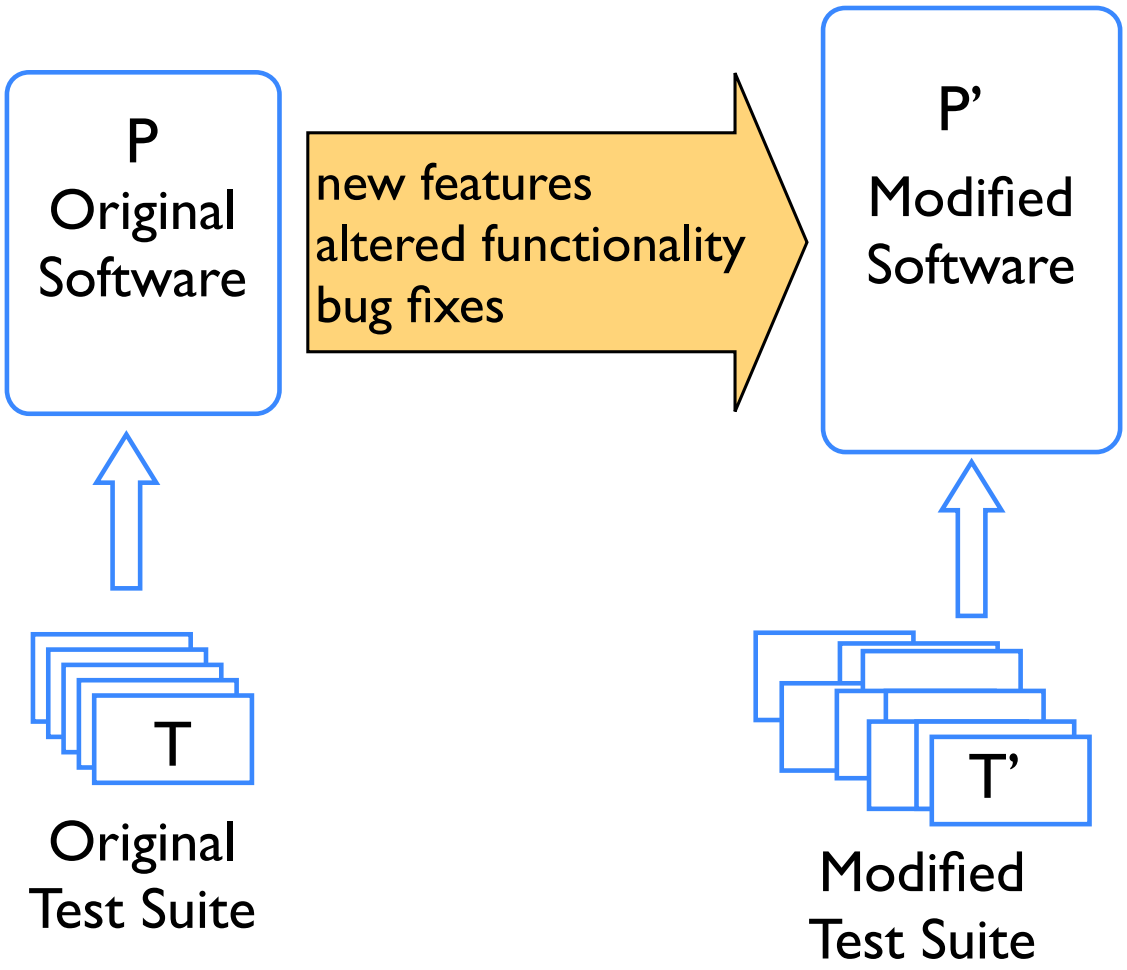
Regression bugs are bad, very bad

- For a new release, customer might be OK if the newly added functionality has bugs, since the common understanding is new code is buggy. and not all users use newly added functions right away
- But customer is not OK if what worked previously is broken in the new release!

Don’t piss off the customer...

When confidence building failed dramatically

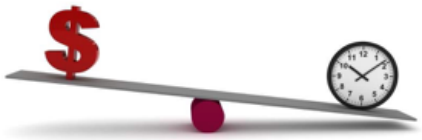
- Regression testing ensures:
  - code carried over unchanged, continues to behave correctly
  - newly added or modified code behaves correctly


P
Original
Software
new features
altered functionality
bug fixes
P'
Modified
Software
T
Original
Test Suite
T'
Modified
Test Suite

Challenges of Regression Testing

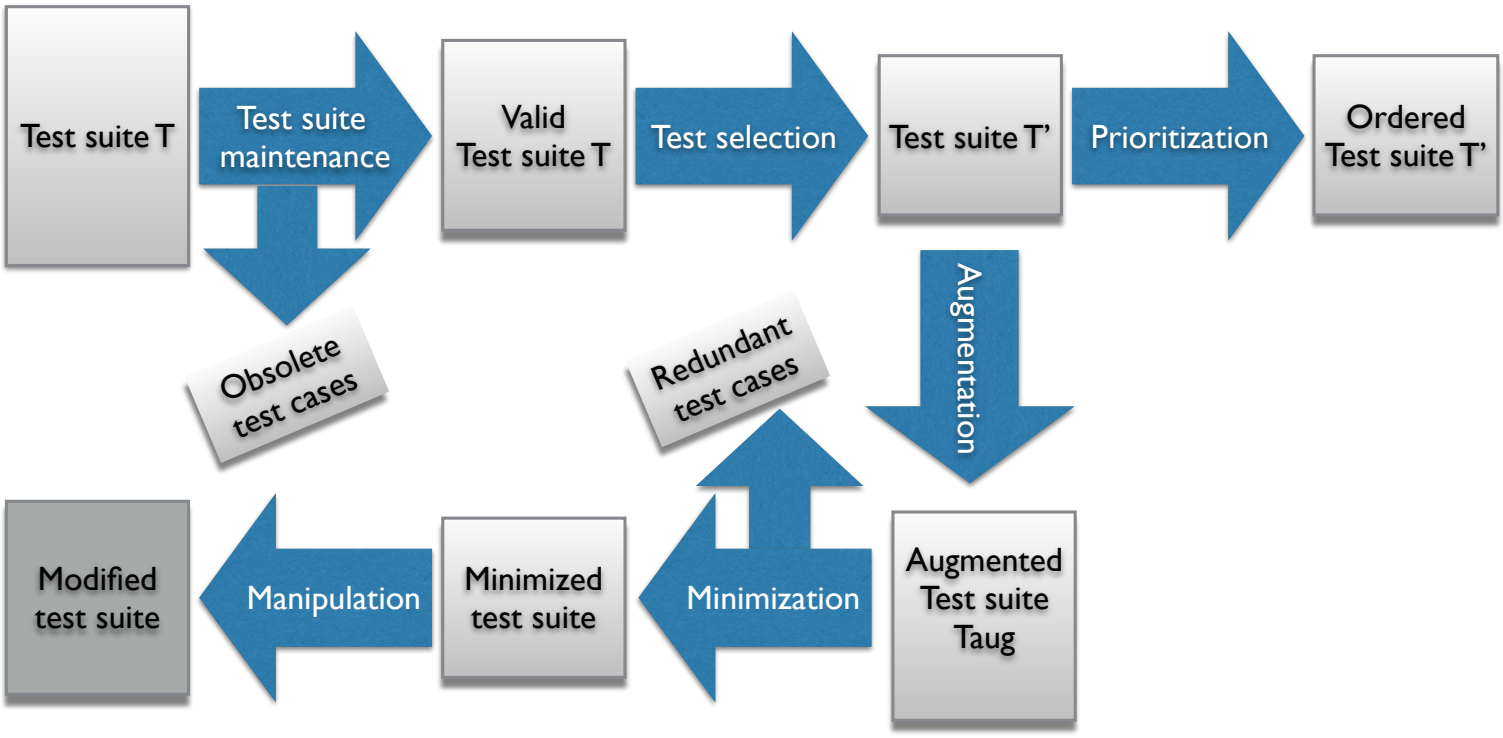
Challenges of Regression Testing

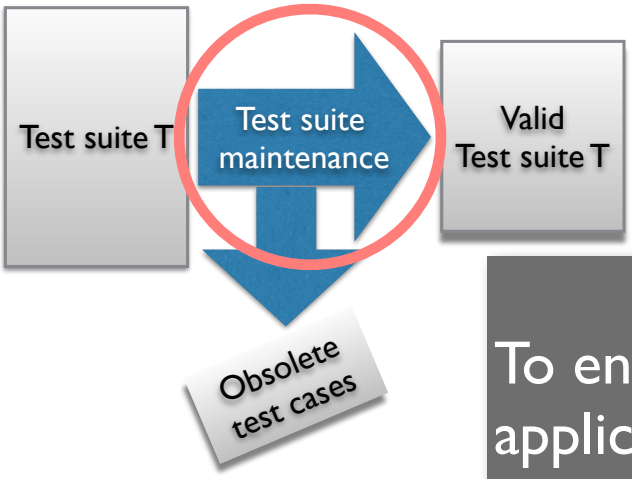
- Cost of maintaining the test suite
  - If I change feature X, how many test cases must be revised because they use feature X?
  - Which test cases should be removed or replaced? Which test cases should be added?
- Cost of re-testing



  - Execution time, CPU usage
  - Proportional to product size, not change size
  - Demotivating developers if exec time is larger than Y (10?) minutes

Solutions?


Test suite T
Test suite maintenance
Valid Test suite T
Test selection
Test suite T'
Prioritization
Ordered Test suite T'
Obsolete test cases
Redundant test cases
Augmentation
Modified test suite
Manipulation
Minimized test suite
Minimization
Augmented Test suite Taug



To ensure that only valid tests that are applicable to P' are used during regression testing.

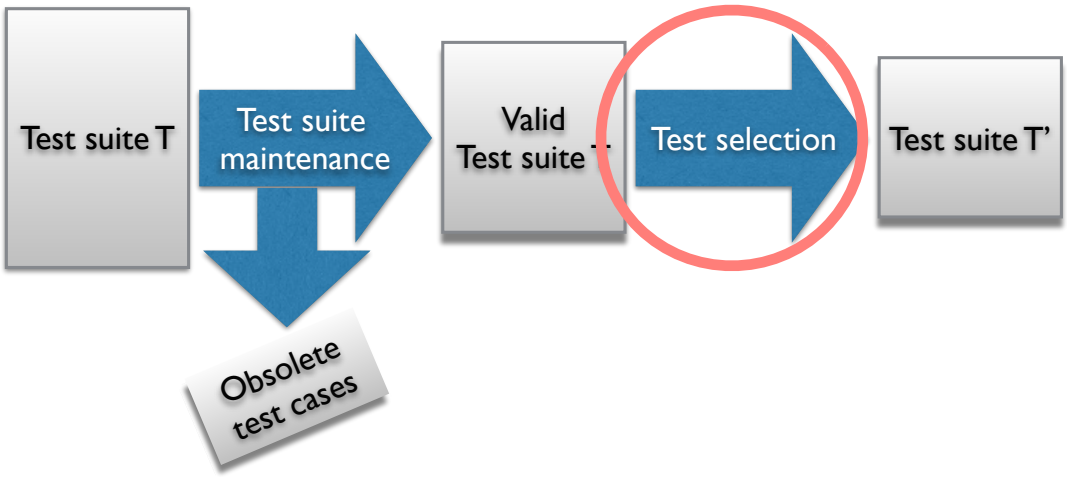
Remove:

1. Broken tests: fail to run
2. Obsolete tests: no longer match specs

Obsolete Tests

- A test case that is no longer valid
  - Tests features that have been modified, substituted, or removed
  - Should be removed from the test suite

For example, $t \in T$ used to exercise a boundary, which is no longer there in the new version. $t \in T$ is obsolete as it no longer tests a construct (boundary) of interest.



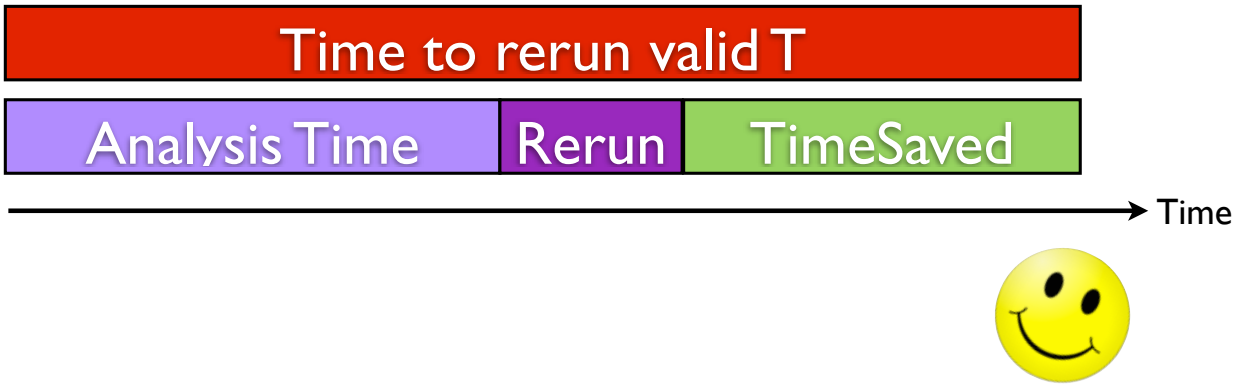
Should we run

Or t

Test Selection Problem

Given: The program, P, the modified version P', and a test suite, T.

Problem: Find a subset of T, T', which adequately tests P'.



Regression Test Selection Techniques

Test All

Often not a practical solution as T is often **too large**, running all tests takes **too long**

Random Selection

– Select randomly tests from T

– The tester decides how many tests to select

– May turn out to be better than no regression testing at all

– But may **miss** tests that exercise modified parts of the code!

Modification Traversing Tests

– Selecting a subset of T such that only tests that guarantee the **execution of modified code** and **code that might be impacted** by the modified code in P' are selected.

– A technique that does not discard any tests traversing a modified or impacted statement is known as a **"safe"** regression test selection technique.

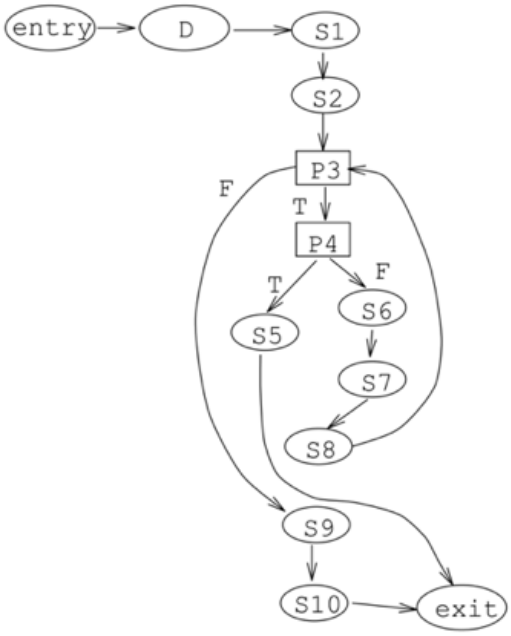
Test Selection Using Execution Traces

1. Run test suite T against P and P'
   - use execution traces to build CFGs
   - test(n) = set of tests that hit node **n** at least once.
2. Identify nodes/edges in P and P' that are equivalent
3. Selection

   For each node $n \in G$ that does not have an equivalent node in G' (changed code):

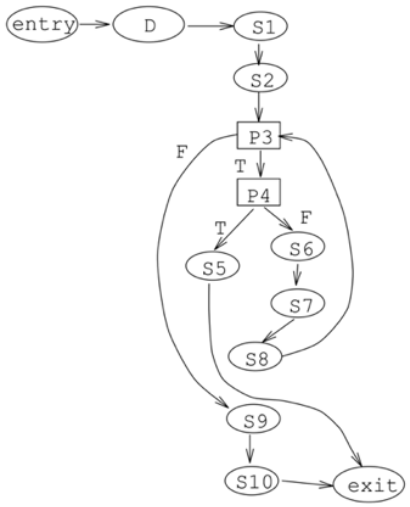
   $T_{RT} = T_{RT} \cup test(n)$

Example



Procedure avg

```
S1.  count = 0
S2.  fread(fileptr,n)
P3.  while (not EOF) do
P4.     if (n<0)
S5.        return(error)
        else
S6.        numarray[count] = n
S7.        count++
        endif
S8.     fread(fileptr,n)
     endwhile
S9.  avg = calcavg(numarray,count)
S10. return(avg)
```



Procedure avg

```
S1.  count = 0
S2.  fread(fileptr,n)
P3.  while (not EOF) do
P4.     if (n<0)
S5.        return(error)
        else
S6.        numarray[count] = n
S7.        count++
        endif
S8.     fread(fileptr,n)
     endwhile
S9.  avg = calcavg(numarray,count)
S10. return(avg)
```

test	input	output	edges traversed
t1	empty file	0	(entry,D),(D,S1), (S1,S2)(S2,P3),(P3,S9), (S9,S10),(S10,exit)
t2	-1	error	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S5),(S5,exit)
t3	1 2 3	2	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S6),(S6,S7),(S7,S8) (S8,P3),(P3,S9),(S9,S10), (S10,exit)

test(n)

1	t1,t2,t3
2	t1,t2,t3
3	t1,t2,t3
4	t2,t3
5	t2
6	t2,t3
7	t3
8	t3
9	t1,t3
10	t1,t3

test	input	output	edges traversed
t1	empty file	0	(entry,D),(D,S1), (S1,S2)(S2,P3),(P3,S9), (S9,S10),(S10,exit)
t2	-1	error	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S5),(S5,exit)
t3	1 2 3	2	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S6),(S6,S7),(S7,S8) (S8,P3),(P3,S9),(S9,S10) (S10,exit)

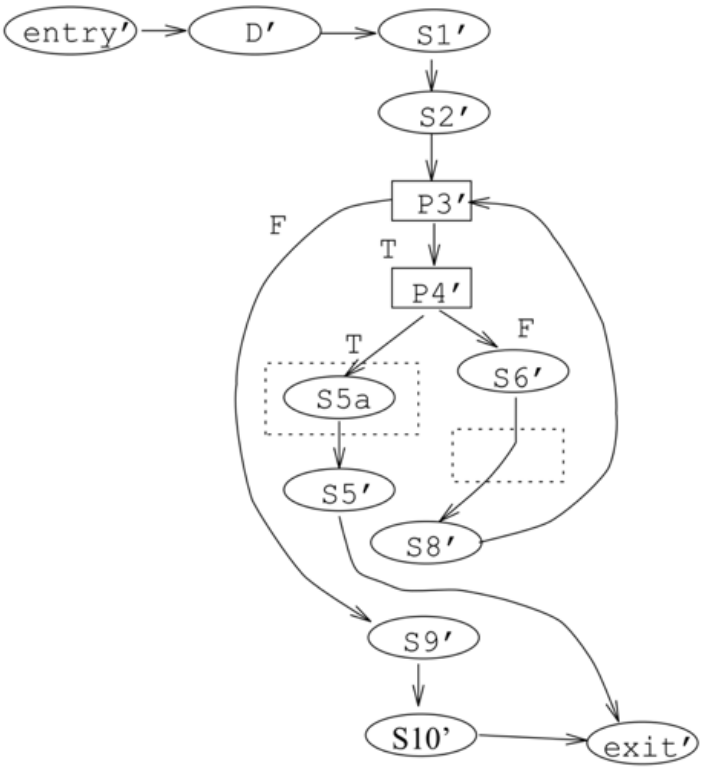
Procedure avg2

```
S1'. count = 0
S2'. fread(fileptr,n)
P3'. while (not EOF) do
P4'.    if (n<0)
S5a.       print("bad input")
S5'.       return(error)
        else
S6'.       numarray[count] = n
        endif
S8'.    fread(fileptr,n)
     endwhile
S9'. avg = calcavg(numarray,count)
S10'.return(avg)
```

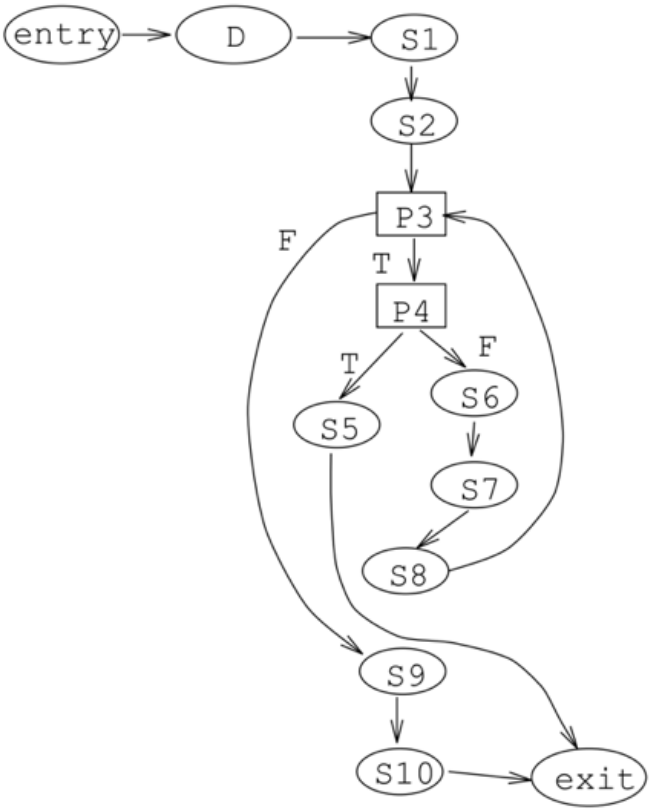
Procedure avg

```
S1.  count = 0
S2.  fread(fileptr,n)
P3.  while (not EOF) do
P4.     if (n<0)
S5.        return(error)
        else
S6.        numarray[count] = n
S7.        count++
        endif
S8.     fread(fileptr,n)
     endwhile
S9.  avg = calcavg(numarray,count)
S10. return(avg)
```

P’


entry’
D’
S1’
S2’
P3’
F
T
P4’
T
F
S5a
S6’
S5’
S8’
S9’
S10’
exit’

P


entry
D
S1
S2
P3
F
T
P4
T
F
S5
S6
S7
S8
S9
S10
exit

Test Cases

test	input	output	edges traversed
t1	empty file	0	(entry,D),(D,S1), (S1,S2)(S2,P3),(P3,S9), (S9,S10),(S10,exit)
t2	-1	error	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S5),(S5,exit)
t3	1 2 3	2	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S6),(S6,S7),(S7,S8) (S8,P3),(P3,S9),(S9,S10), (S10,exit)

Tests selected for rerun: t2 and t3

test	input	output	edges traversed
t1	empty file	0	(entry,D),(D,S1), (S1,S2)(S2,P3),(P3,S9), (S9,S10),(S10,exit)
t2	-1	error	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S5),(S5,exit)
t3	1 2 3	2	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S6),(S6,S7),(S7,S8) (S8,P3),(P3,S9),(S9,S10), (S10,exit)

Exercise: test selection

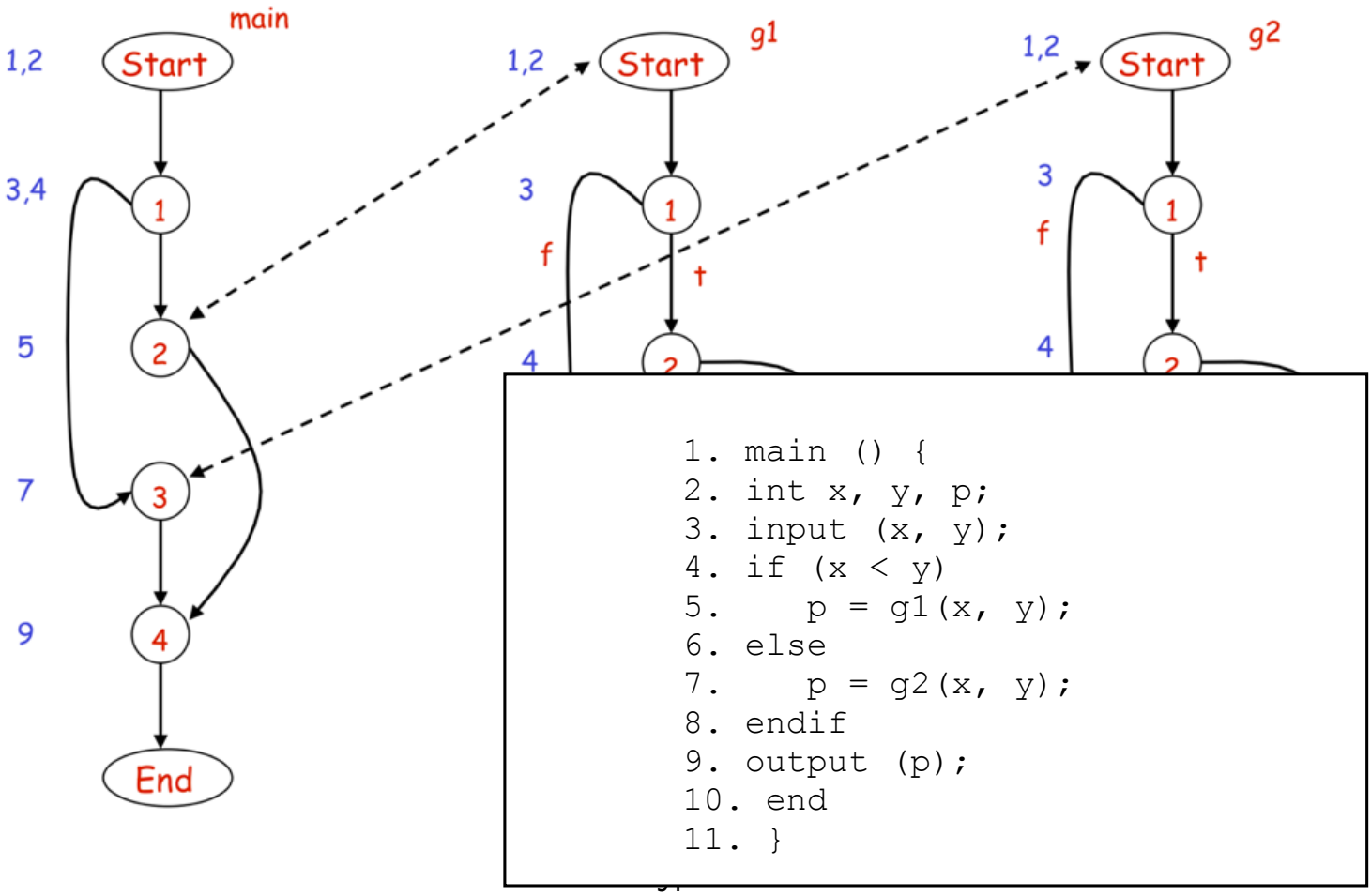
t1 : <x=1, y=3>
t2 : <x=2, y=1>
t3 : <x=1, y=2>

```
1. main () {
2. int x, y, p;
3. input (x, y);
4. if (x < y)
5.    p = g1(x, y);
6. else
7.    p = g2(x, y);
8. endif
9. output (p);
10. end
11. }
```

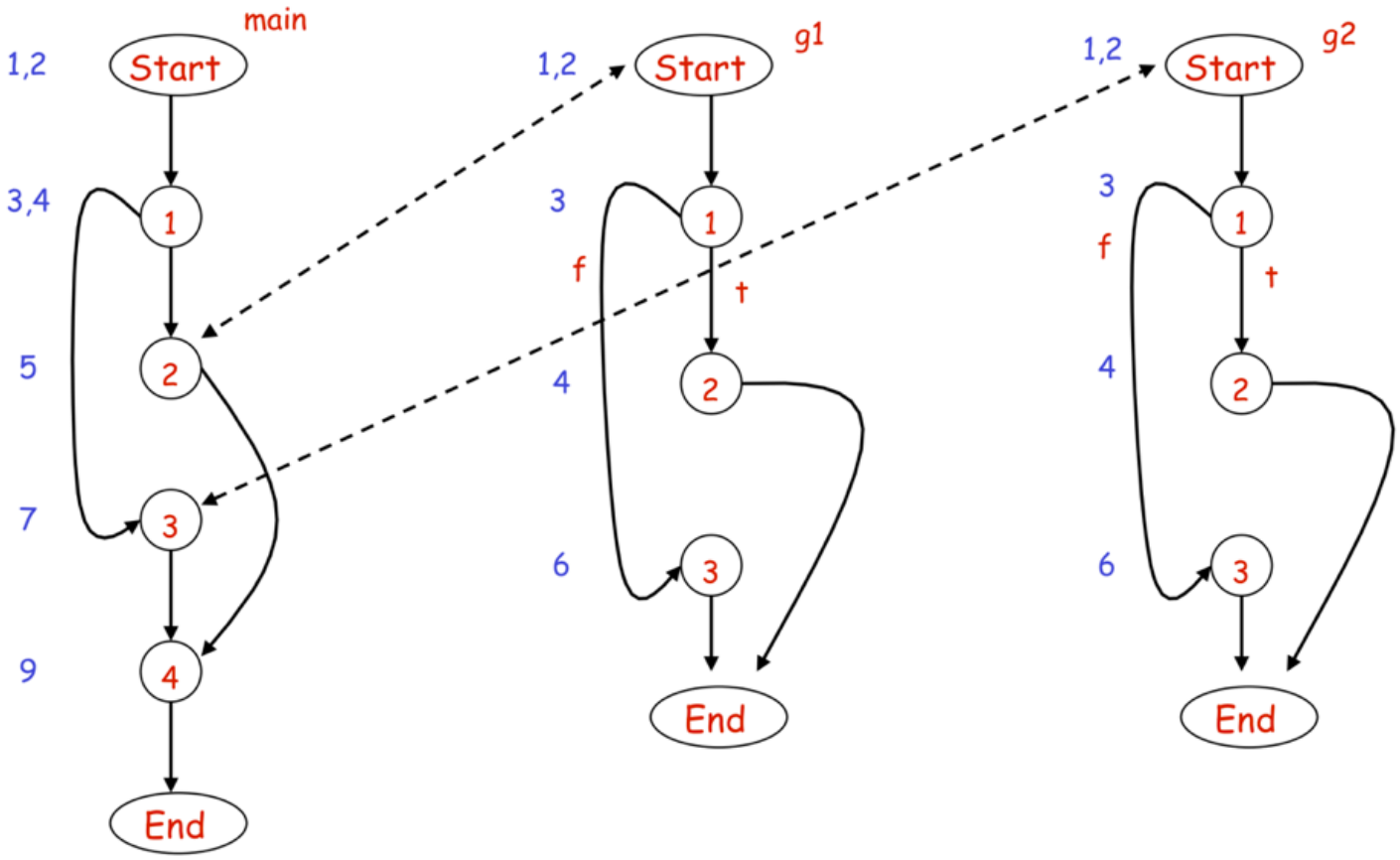
```
1. int g1 (int a, b) {
2. int a, b;
3. if (a + 1 == b)//modified: a - 1 == b
4.    return (a*a);
5. else
6.    return (b*b);
}
```

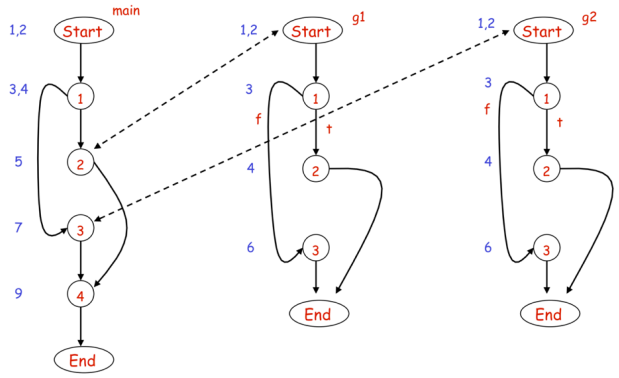
```
1. int g2 (int a, b) {
2. int a, b;
3. if (a == (b + 1))
4.    return (b*b);
5. else
6.    return (a*a);
}
```

Control Flow Graph



Control Flow Graph





Test(t)	Execution Trace (trace(t))
t1	main.Start, main.1, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End
t2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2,End, main.3, main.4, main.End
t3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End

t1 : <x=1, y=3>

t2 : <x=2, y=1>

t3 : <x=1, y=2>

Test vector

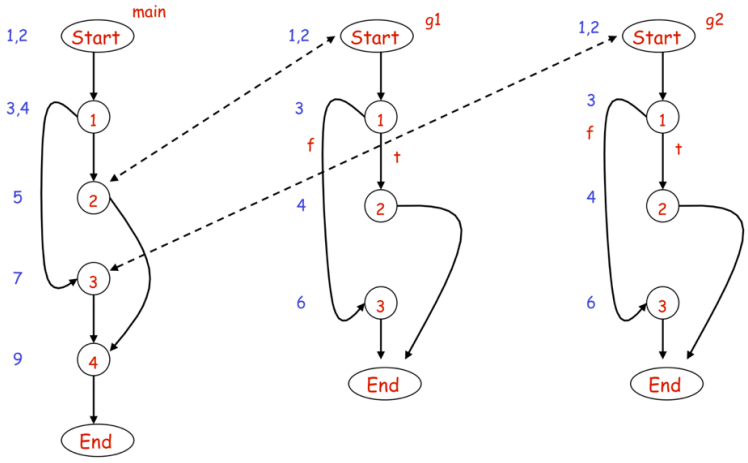
A test vector for node n, denoted by test(n), is the set of tests that traverse node n in the CFG. For program P we obtain the following test vectors:

	n1	n2	n3	n4
main	t1,t2,t3	t1,t3	t2	t1,t2,t3
g1	t1,t3	t3	t1	-
g2	t2	t2	None	-

Suppose that function g1 in P is modified as follows.

```
1. int g1 (int a, b) {
2. int a, b;
3. if (a - 1 == b) <- modified
4.     return (a*a);
5. else
6.     return (b*b);
```

Focus: node g1.1



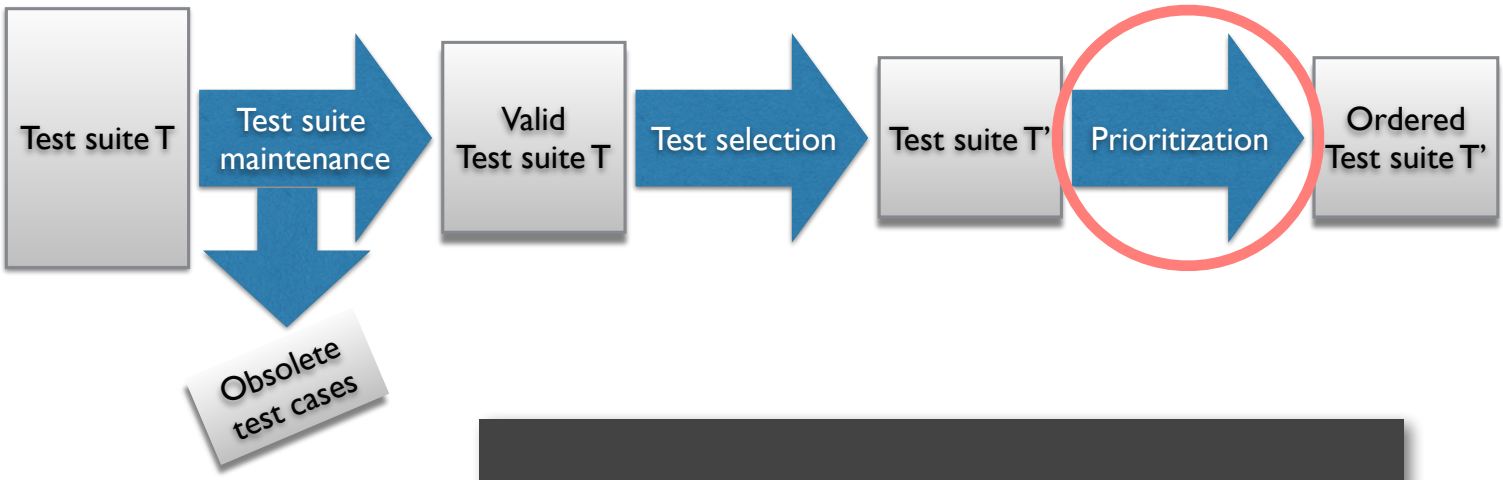
t1 : <x=1, y=3>
t2 : <x=2, y=1>
t3 : <x=1, y=2>

Test(t)	Execution Trace (trace(t))
t1	main.Start, main.1, g1.Start, **g1.1**, g1.3, g1.End, main.2, main.4, main.End
t2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2,End, main.3, main.4, main.End
t3	main.Start, main.1, main.2, g1.Start, **g1.1**, g1.2, g1.End, main.2, main.4, main.End

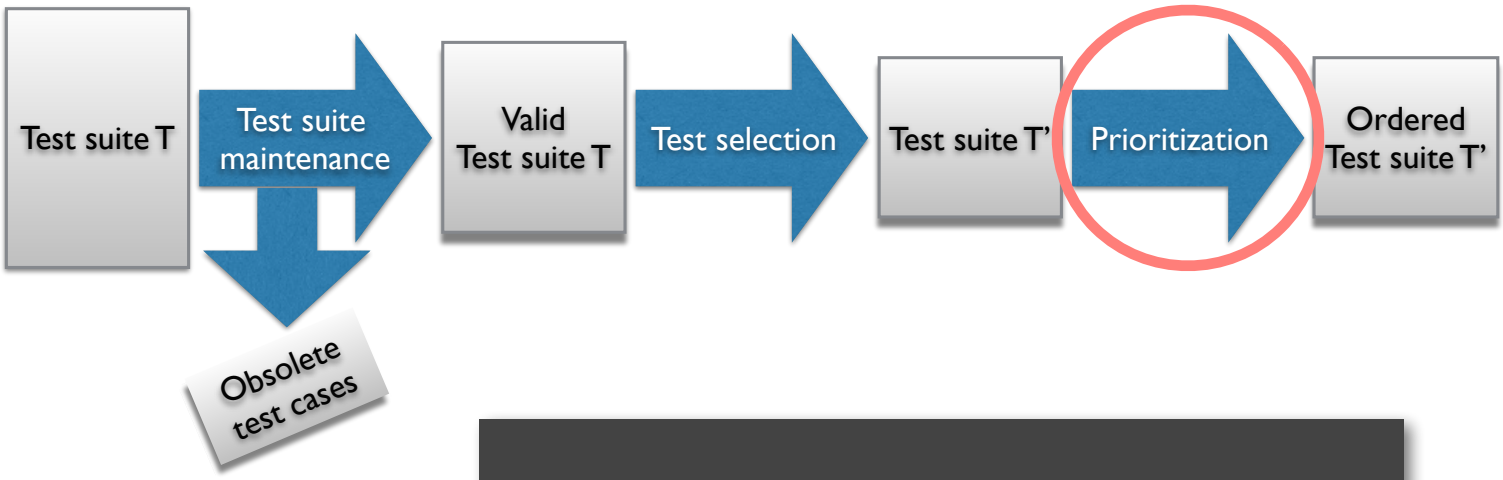
func	1	2	3	4
main	t1,t2,t3	t1,t3	t2	t1,t2,t3
g1	**t1,t3**	t3	t1	-
g2	t2	t2	None	-

Selected tests:

T_{RT}={t1, t3}



Prioritizing tests based on some criteria – Why



Prioritizing tests based on some criteria

– Why

1. After maintenance and selection, there might still be too many test cases left to execute (cannot afford to execute them all).
2. If time is limited, make sure the important tests are executed first.

Prioritization

- In practice, sufficient resources may not be available to execute all selected tests.
- Solution: rank tests and execute high-priority tests first

Ranking Criteria

– **cost** (e.g., execution time): Tests with lower costs are ranked first while test with higher costs are ranked last
– **risk** (expected risk of not executing a test): Tests with higher risks are ranked first while test with lower risks are ranked last

Residual coverage

- refers to the number of elements that remain to be covered w.r.t. a given coverage criterion.
- One way to prioritize tests is to give **higher priority** to tests that lead to a **smaller residual coverage.**

Consider a program P consisting of four classes C1, C2, C3, and C4. Each of these classes has one or more methods as follows:
C1 = {m1, m12, m16}
C2 = {m2, m3, m4}
C3 = {m5, m6, m10, m11}
C4 = {m7, m8, m9, m13, m14, m15}.

Test(t)	Methods covered (cov(t))	|cov(t)|
t1	1,2,3,4,5,10,11,12,13,14,16	11
t2	1,2,4,5,12,13,15,16	8
t3	1,2,3,4,5,12,13,14,16	9
t4	1,2,4,5,12,13,14,16	8
t5	1,2,4,5,6,7,8,10,11,12,13,15,16	13

Test(t)	Methods covered (cov(t))	|cov(t)|
t1	1,2,3,4,5,10,11,12,13,14,16	11
t2	1,2,4,5,12,13,15,16	8
t3	1,2,3,4,5,12,13,14,16	9
t4	1,2,4,5,12,13,14,16	8
t5	1,2,4,5,6,7,8,10,11,12,13,15,16	13

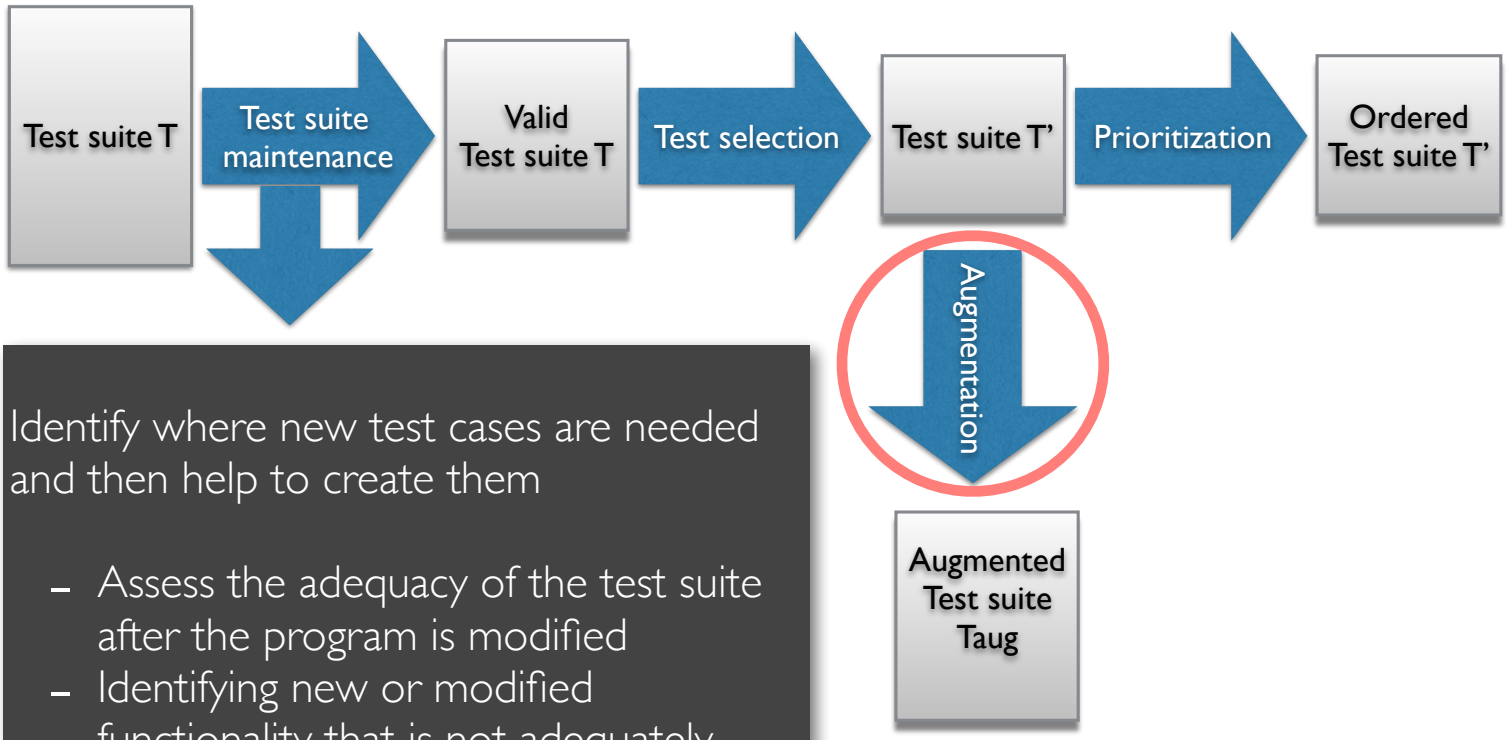
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	t	R
x	x	x	x	x	?	?	?	?	x	x	x	x	x	?	x	1	
x	x	?	x	x	?	?	?	?	?	?	x	x	?	x	x	2	
x	x	x	x	x	?	?	?	?	?	?	x	x	x	?	x	3	
x	x	?	x	x	?	?	?	?	?	?	x	x	x	?	x	4	
x	x	?	x	x	x	x	x	?	x	x	x	x	?	x	x	5	

Test(t)	Methods covered (cov(t))	|cov(t)|
t1	1,2,3,4,5,10,11,12,13,14,16	11
t2	1,2,4,5,12,13,15,16	8
t3	1,2,3,4,5,12,13,14,16	9
t4	1,2,4,5,12,13,14,16	8
t5	1,2,4,5,6,7,8,10,11,12,13,15,16	13

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	t	R
x	**x**	x	**x**	**x**	**?**	**?**	**?**	?	**x**	**x**	**x**	**x**	x	**?**	**x**	1	
x	**x**	?	**x**	**x**	**?**	**?**	**?**	?	**?**	**?**	**x**	**x**	?	**x**	**x**	2	
x	**x**	x	**x**	**x**	**?**	**?**	**?**	?	**?**	**?**	**x**	**x**	x	**?**	**x**	3	
x	**x**	?	**x**	**x**	**?**	**?**	**?**	?	**?**	**?**	**x**	**x**	x	**?**	**x**	4	
x	**x**	?	**x**	**x**	**x**	**x**	**x**	?	**x**	**x**	**x**	**x**	?	**x**	**x**	5	1

Test(t)	Methods covered (cov(t))	|cov(t)|
t1	1,2,3,4,5,10,11,12,13,14,16	11
t2	1,2,4,5,12,13,15,16	8
t3	1,2,3,4,5,12,13,14,16	9
t4	1,2,4,5,12,13,14,16	8
t5	1,2,4,5,6,7,8,10,11,12,13,15,16	13

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	t	R
x	x	x	x	x	?	?	?	?	x	x	x	x	x	?	x	1	2
x	x	?	x	x	?	?	?	?	?	?	x	x	?	x	x	2	
x	x	x	x	x	?	?	?	?	?	?	x	x	x	?	x	3	3
x	x	?	x	x	?	?	?	?	?	?	x	x	x	?	x	4	
x	x	?	x	x	x	x	x	?	x	x	x	x	?	x	x	5	1

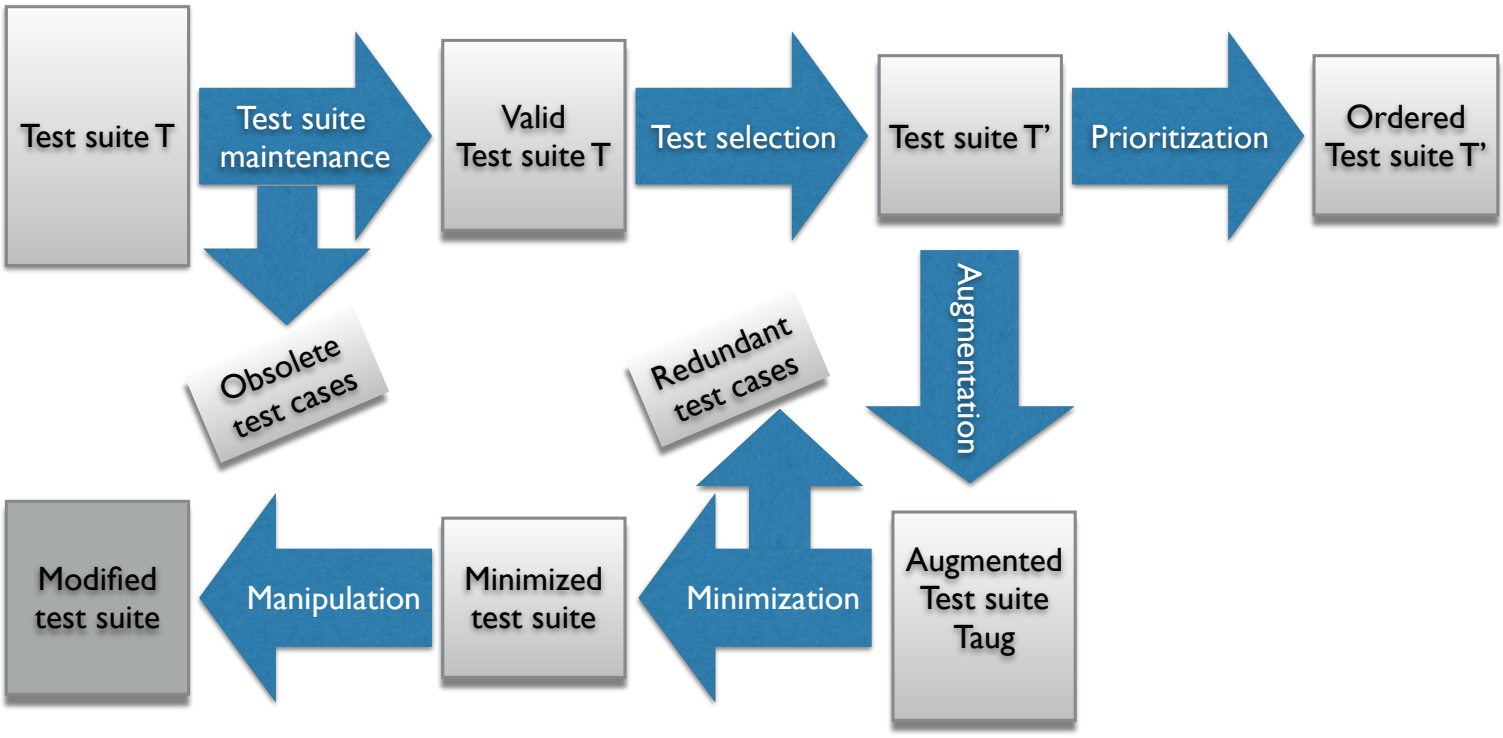


Identify where new test cases are needed and then help to create them

- Assess the adequacy of the test suite after the program is modified
- Identifying new or modified functionality that is not adequately exercised by the existing test suite
- Create

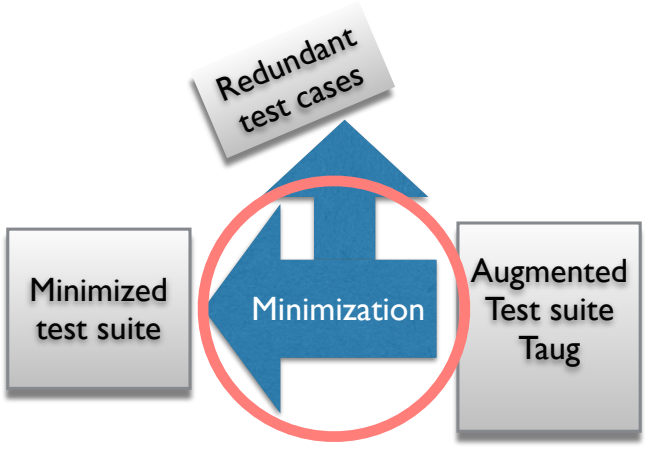
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	t	R
x	x	x	x	x	?	?	?	?	x	x	x	x	x	?	x	1	
x	x	?	x	x	?	?	?	?	?	?	x	x	?	x	x	2	
x	x	x	x	x	?	?	?	?	?	?	x	x	x	?	x	3	2
x	x	?	x	x	?	?	?	?	?	?	x	x	x	?	x	4	
x	x	?	x	x	x	x	x	?	x	x	x	x	?	x	x	5	1

Method 9 is not tested at all. Augment it!


Test suite T
Test suite maintenance
Valid Test suite T
Test selection
Test suite T'
Prioritization
Ordered Test suite T'
Obsolete test cases
Redundant test cases
Augmentation
Modified test suite
Manipulation
Minimized test suite
Minimization
Augmented Test suite Taug

Reduce the size of a test suite by eliminating
from the test suite.

Is it possible to reduce T to T' such that T' still covers all the testable entities that are covered by T?



Redundant Test

- A test case that does not differ significantly from other tests (based on some test adequacy criterion)
  - Unlikely to find a fault missed by similar test cases
  - Has extra cost in re-execution
  - Has extra cost to maintain
  - May or may not be removed, depending on costs!

Greedy Algorithm: maximum coverage

1. Find a test t in T that covers the maximum number of requirements R.
2. Add t to the return set, and remove it from T and the requirements it covers from R
3. Repeat the same procedure until all requirements in R have been covered.

	R1	R2	R3	R4	R5	R6	R7
t1			x	x		x	
t2	x					x	
t3					x		x
t4					x		
t5			x				
t6		x		x			
t7		x	x		x		x

T'= ?

		R1	R2	R3	R4	R5	R6	R7
3	t1			x	x		x	
2	t2	x					x	
2	t3					x		x
1	t4					x		
1	t5			x				
2	t6		x		x			
4	t7		x	x		x		x

T'= ?

		R1	R2	R3	R4	R5	R6	R7
2	t1			x	x		x	
2	t2	x					x	
	t3					x		x
	t4					x		
	t5			x				
1	t6		x		x			
	t7		x	x		x		x

T'= {t7,?}

	R1	R2	R3	R4	R5	R6	R7
t1			x	x		x	
t2	x					x	
t3					x		x
t4					x		
t5			x				
t6		x		x			
t7		x	x		x		x

T'= {t7,t1,t2}

GRE Heuristic: essential tests

1. Concept: selects all essential test cases. A test case is essential when **only this test case** covers one specific requirement.
2. Remove what is not essential

Find essential tests

	R1	R2	R3	R4	R5	R6
t1	x	x	x			
t2	x			x		
t3		x			x	
t4			x			x
t5					x	

GRE heuristic

	R1	R2	R3	R4	R5	R6
t1	x	x	x			
t2	x			x		
t3		x			x	
t4			x			x
t5					x	

GRE heuristic

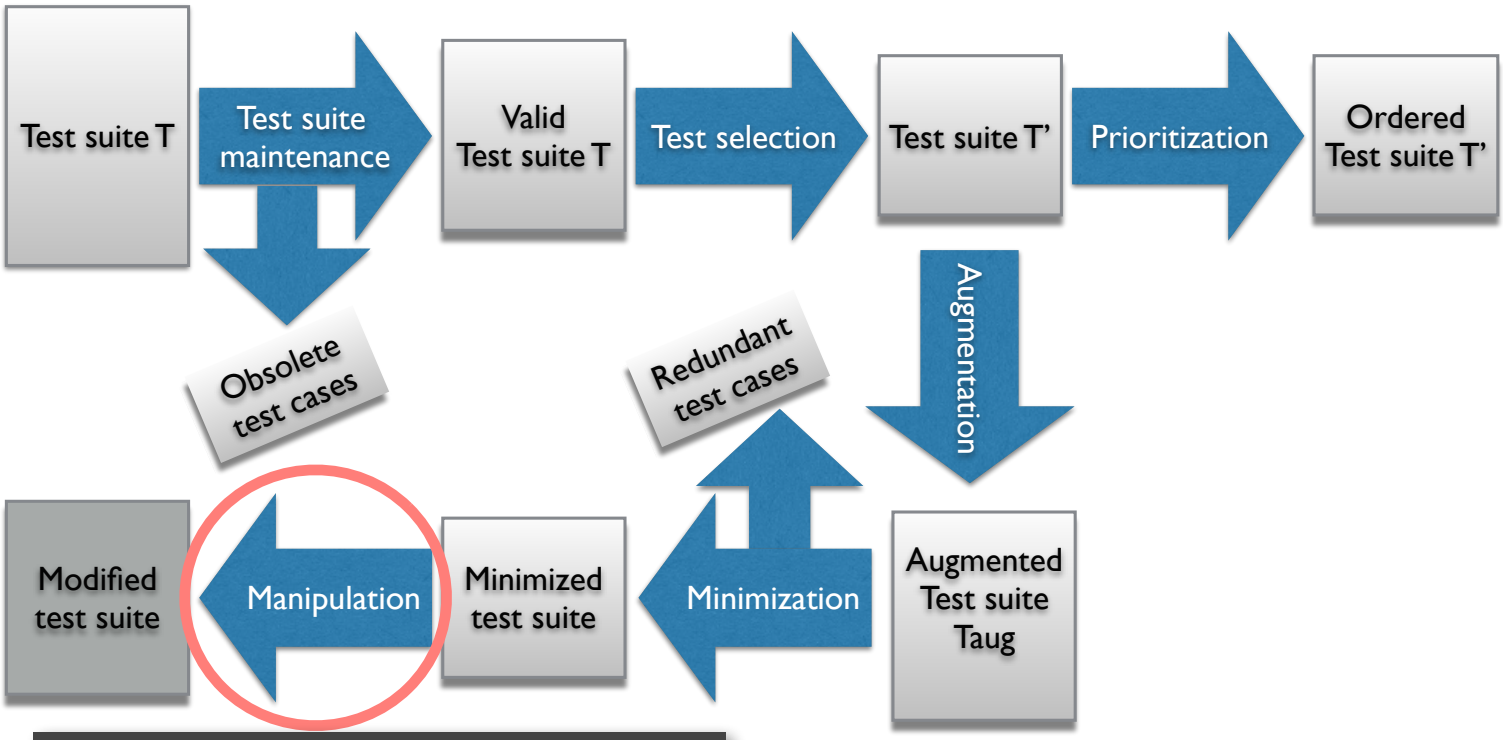
	R1	R2	R3	R4	R5	R6
t1	x	x	x			
t2	x			x		
t3		x			x	
t4			x			x
t5					x	

GRE heuristic

	R1	R2	R3	R4	R5	R6
t1	x	x	x			
t2	x			x		
t3		x			x	
t4			x			x
t5					x	

GRE heuristic

	R1	R2	R3	R4	R5	R6
t1	x	x	x			
t2	x			x		
t3		x			x	
t4			x			x
t5					x	



Add new tests for new functionality.

Bugs in tests

- Just as prevalent as bugs in production code
- False Alarms: test fails, while production code is correct
- Silent Horrors: test passes, while production code is incorrect

Open research challenges

- Automated test case repair
- Detecting silent horror test bugs

Summary

- Regression testing is crucial for software evolution
- Different techniques to avoid running all test cases on each new (small) revision
- Test selection, prioritization, augmentation, and minimization
- Maintain your test code just like your production code