Software Testing II

Assessment 2

Lab: Queue

MSc Projects

Common Errors

- Can be from a particular programming community.
- Well-instrumented organisations monitor and summarise error occurrences.
- Professional good practice should make you sensitive to the errors you make personally.
- The following are the "top three" from David Reilly's top ten Java programming errors
  - Concurrent access to shared variables by threads (3)
  - Capitalization errors (2)
  - Null pointers (1)

Concurrent access to shared variables by threads

```
public class MyCounter {
  private int count = 0; // count starts at zero

  public void incCount(int amount) {
    count = count + amount;
  }

  public int getCount() {
    return count;
  }
}
...
                MyCounter c;
// Thread 1                     // Thread 2
c.incCount(1);                  c.incCount(1);
                // join
                c.getCount() == ?
```

Concurrent access to shared variables by threads


```
public class MyCounter {
  private int count = 0; // count starts at zero
```



```
  public synchronized void incCount(int amount) {
    count = count + amount;
  }
```



```
  public int getCount() {
    return count;
  }
}
```

Capitalization Errors

- All methods and member variables in the Java API begin with lowercase letters.

- All methods and member variables use capitalization where a new word begins – e.g. getDoubleValue().

Null pointer

```
public static void main(String args[]) {
  String[] list = new String[3]; // Accept up to 3 parameters
  int index = 0;

  while( (index < args.length) && (index < 3) ) {
    list[index] = args[index];
    index++;
  }

  // Check all the parameters
  for(int i = 0; i < list.length; i++) {
    if(list[i].equals("-help")) {
      // .........
    } else if(list[i].equals("-cp")) {
      // .........
    }
    // [else .....]
  }
}
```

Test Coverage

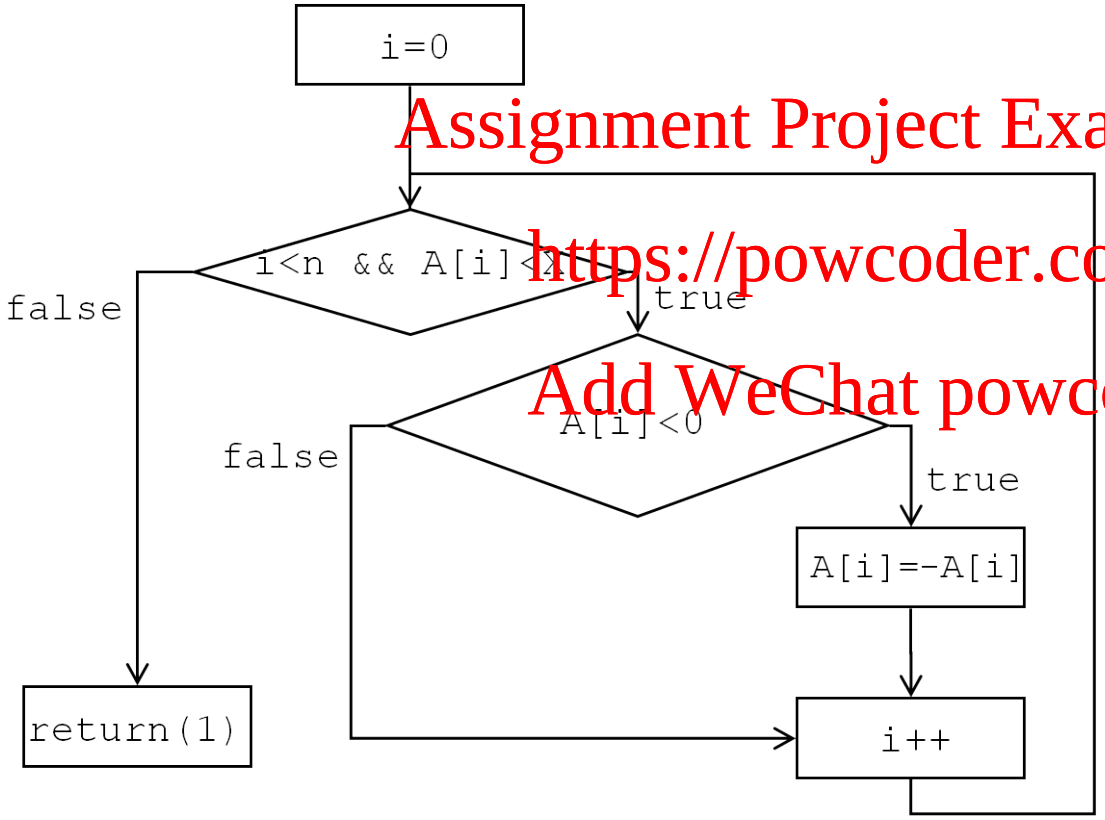
- Statement Coverage
- Branch Coverage
- Condition Coverage

Statement Testing

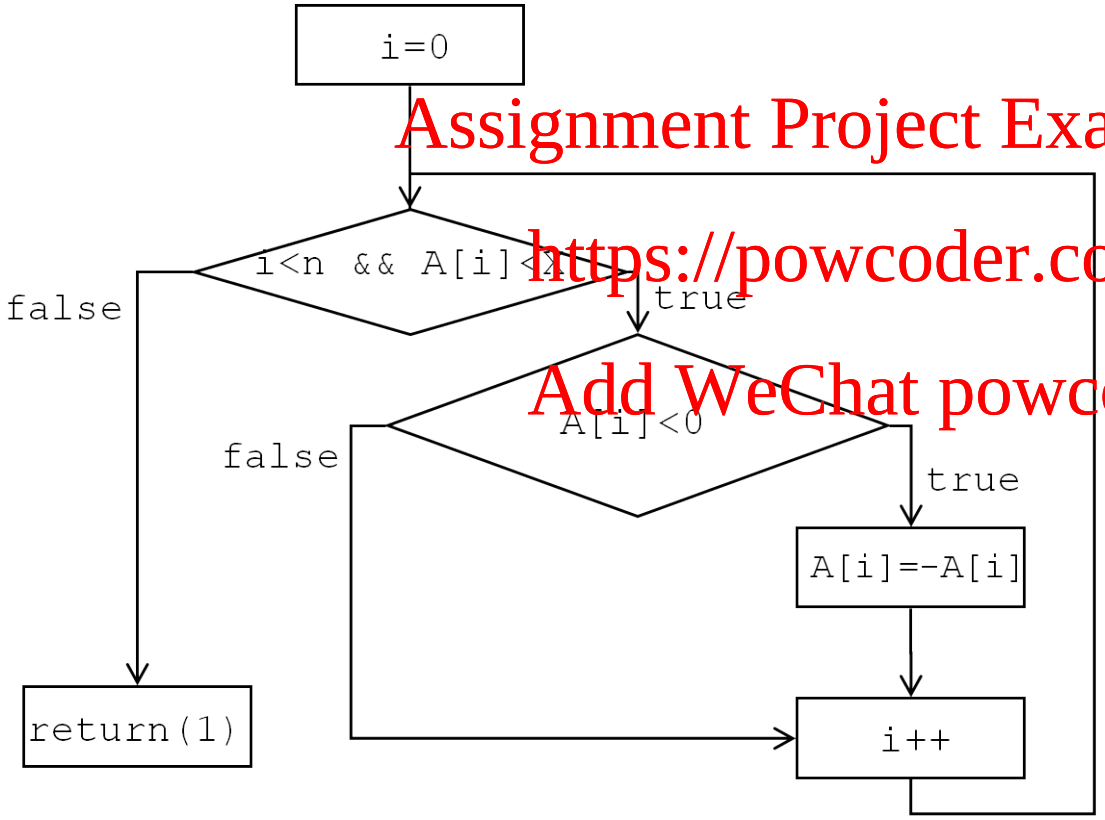
- **Statement Adequacy**: all statements have been executed by at least one test.

- **Statement Coverage:** for a particular test T, this is the quotient of the number of statements executed during a run of T (not counting repeats) and the number of statements in the program.

Running example



Running example



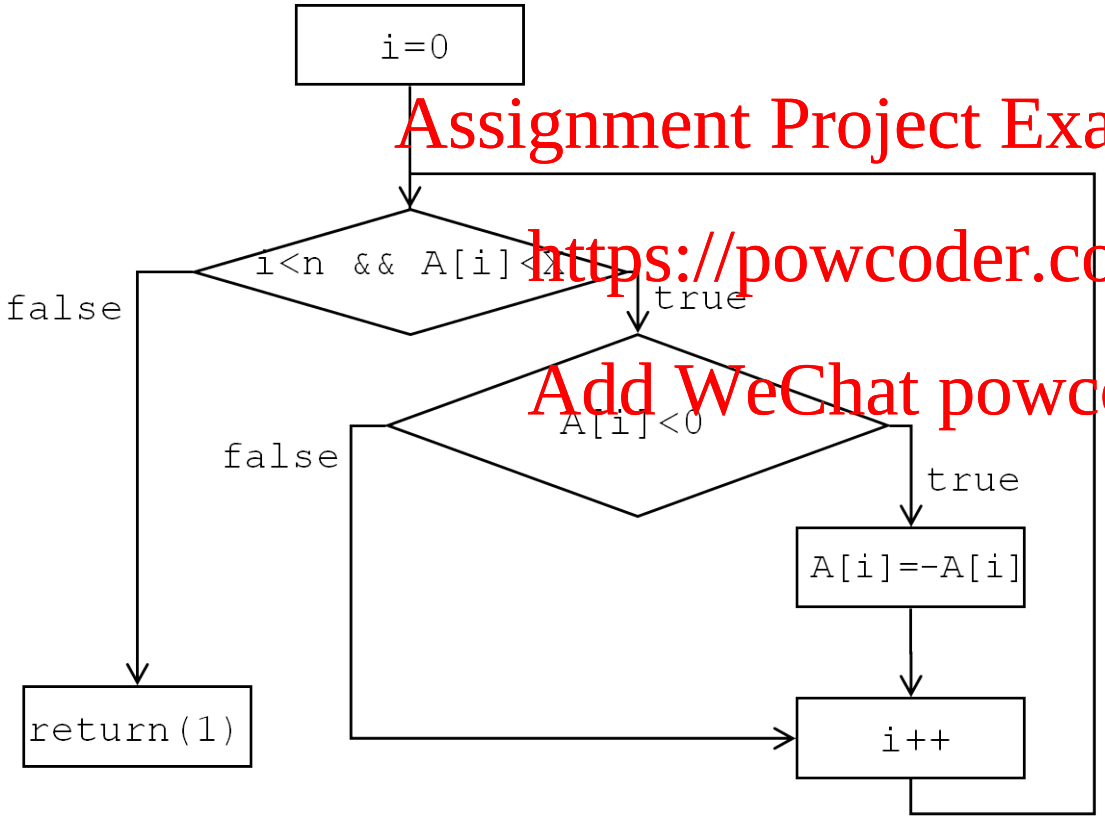
- A = [-1], X=2

Branch Coverage

- **Branch adequacy**:
  Let T be a test suite for a program P. T satisfies the branch adequacy criterion if for each branch B of P there exists at least one test case that exercises B

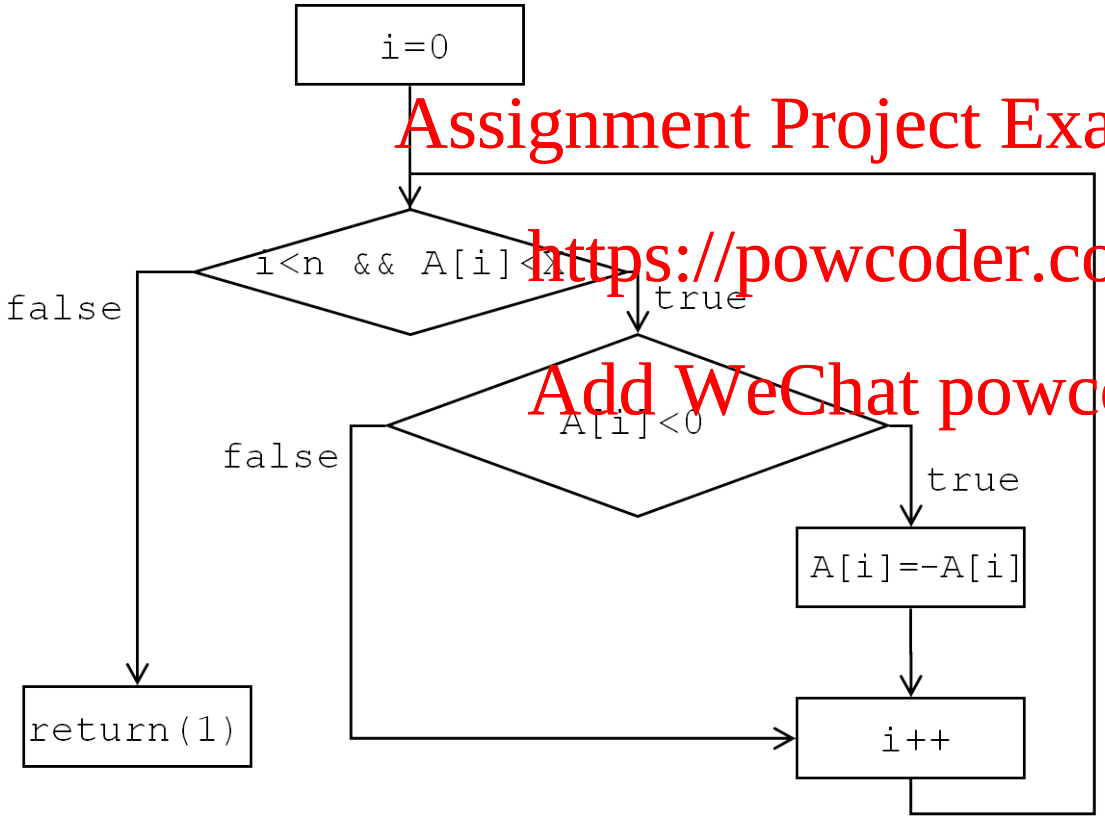
- **The branch coverage** for a test suite is the ratio of branches tested by the suite and the number of branches in the program under test.

Running example



- A = [-1], X=2

Running example

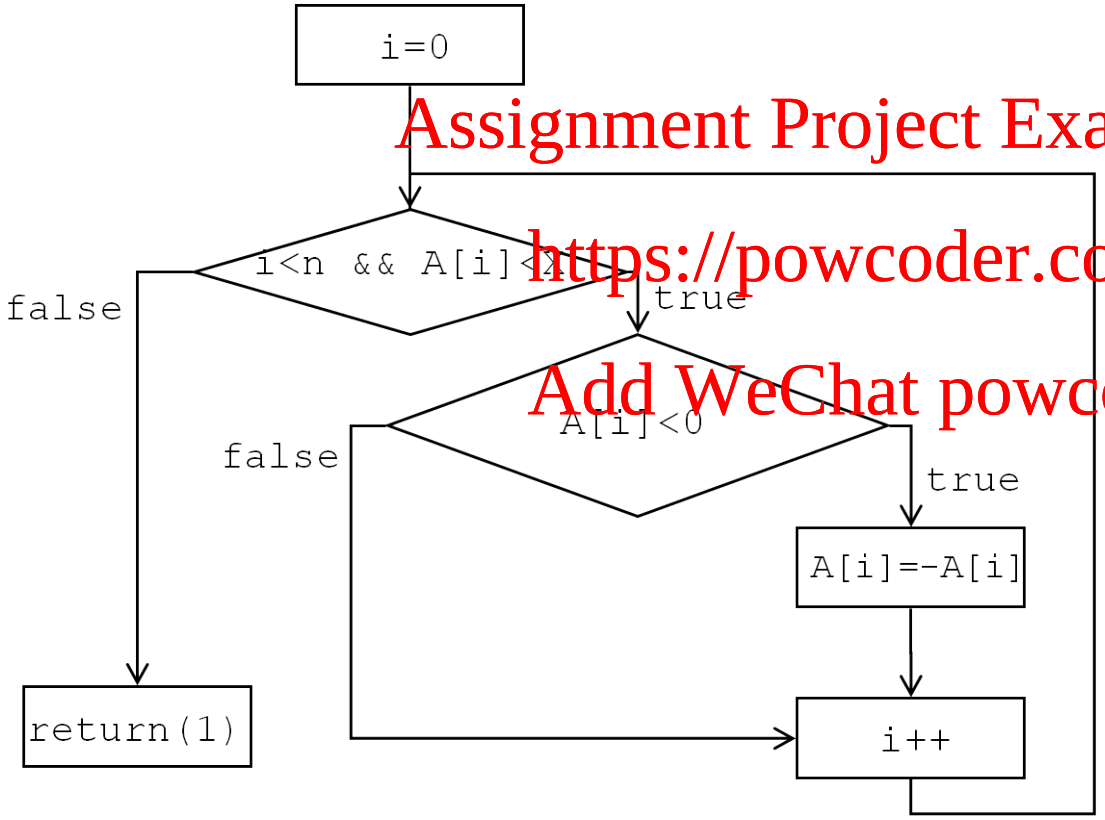


- A = [-1], X=2
- A = [-1,1], X=2

Condition coverage

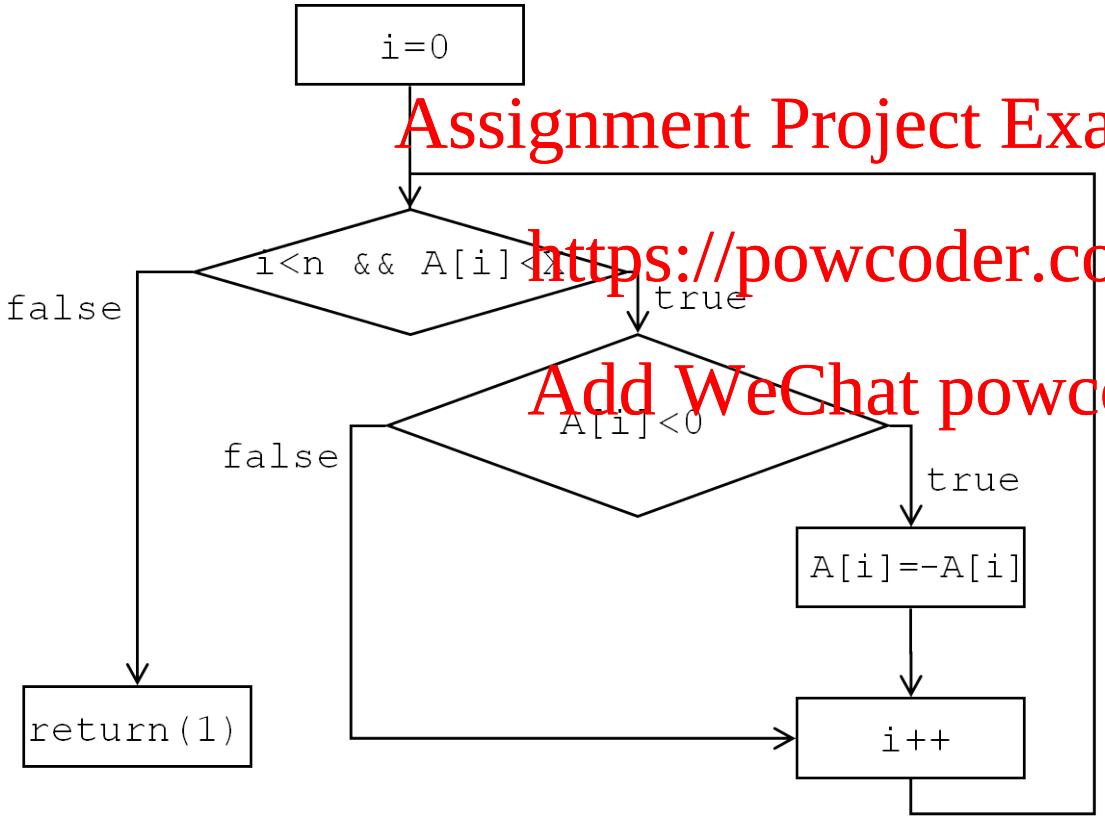
- **Condition adequacy** criterion is:
  Let T be a test suite for program P. T covers all the basic conditions of P if in each basic condition of P evaluates to true under some test in T and evaluates to false under some test in T.

Running example



- A = [-1], X=2
- A = [-1,1], X=2

Running example



- A = [-1], X=2
- A = [-1,1], X=2
- A = [-1,1,2], X = 2

Mutation Testing

- What is a mutation?
- What is mutation testing?
- When should we use mutation testing?
- Examples

What is a mutation?

- A mutation is a small change in a program.
- Such small changes are intended to model low level defects that arise in the process of coding systems
- Ideally mutations should model low-level defect creation.

What is Mutation Testing?

- Mutation testing is a structural testing method aimed at assessing/improving the **adequacy** of test suites, and estimating the number of faults present in systems under test.
- The process, given program P and test suite T, is as follows:

  - We systematically apply mutations to the program P to obtain a sequence P1, P2,... Pn of mutants of P. Each mutant is derived by applying a single mutation operation to P.
  - We run the test suite T on each of the mutants, T is said to kill mutant Pj if it detects an error.
  - If we kill k out of n mutants the adequacy of T is measured by the quotient
  - k/n. T is mutation adequate if k = n.

When should we use mutation testing?

- Structural test suites are directed at identifying defects in the code. One goal of mutation testing is to assess or improve the efficacy of test suites in discovering defects.
- When we are carrying out structural testing we are worried about defects remaining in the code. Often we are keen to measure the **Residual Defect Density (RDD)** in the program P under test.
- The Residual Defect Density is usually measured in defects per thousand lines of code.

Using Mutation Testing to Estimate the RDD

- Suppose we have an estimate r of the RDD of programs produced by our development process before the one subject to test (this could be gathered using production data and field experience, or it could be based on the number of faults our tests have already detected).
- Generate n mutants of the program P.
- Test each mutant with the test suite T .
- Find the number, k, of mutants that are killed by T. To yield a non-zero RDD. we need to test enough mutants to ensure that $0 < k < n$.
- Use r. $(n - k)/k$ as the estimate for the RDD of the tested program.
- k/n is a measure of the adequacy of T in finding defects in P

Kinds of Mutation

- **Value Mutations**: these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds – being one out on the start or finish is a very common error.
- **Decision Mutations**: this involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs, e.g. a typical mutation might be replacing a > by a < in a comparison
- **Statement Mutations**: these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

Offutt's Mutations for Inter-Class Testing

Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	super keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Value Mutation


```
public int segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;
      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have t length 1 and have l < t[0] < u, then the program would fail to count t[0] and return 0 rather than 1 as a result

Decision Mutation

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to t[i]>u will cause miscounting

We can model “one-off” errors in the loop bound by changing this condition to i<=t.length - provided array bounds are checked exactly this will provoke an error on every execution.

Statement Mutation

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k)
}
```


Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.

Observations

- Mutations model low level errors in the mechanical production process. Modelling design errors is much harder because they involve large changes coordinated throughout the program.
- Ensuring test sets satisfy coverage criteria are often enough to ensure they kill mutants (because mutants often do not "make sense" and so provoke a failure if they are ever executed).
- Black-box test sets are poorer at killing mutants – we'd expect this because black-box tests are driven more by the operational profile than by the need to cover statements.
- We could see mutation testing as a way of forcing more diversity on the development of test sets if we use a black-box approach as our primary test development approach.

Regression Testing

- Regression testing is applied to code immediately after changes a
- The goal is to assure that the changes have not had unintended consequences on the behaviour of the test object re made.
- We can apply regression testing during development and in the field after the system has been upgraded or maintained in some other way.
- Give us confidence that we can change the object of test while maintaining its intended behaviour.
- • Regression testing is an important way of monitoring the effects of change.

More system testing

- Capacity Testing
- Stress Testing
- Usability Testing
- Security Testing
- Performance Testing
- Reliability Testing
- Compliance Testing
- Recoverability Testing
- Documentation Testing
- Configuration Testing

PI 8.1

```
if(a || b)) {
    test1 = true;
} else {
    if(c) {
            test2 = true
        }
}
```

How many of the following statements is correct?

For full statement coverage, we need a=true b=false; a=false b=false c=true

For full branch coverage, we need a=true b=false; a=false b=false c=true; a=false b=false c=false

For full condition coverage, we need a=true b=false; a=false b=false c=true; a=false b=false c=false; a=false, b=true

A: 0
B: 1
C: 2
D: 3

PI 8.1

```
if(a || b)) {
    test1 = true;
} else {
    if(c) {
            test2 = true
        }
}
```


How many of the following statements is correct?

For full statement coverage, we need a=true b=false; a=false b=false c=true

For full branch coverage, we need a=true b=false; a=false b=false c=true; a=false b=false c=false

For full condition coverage, we need a=true b=false; a=false b=false c=true; a=false b=false c=false; a=false, b=true

A: 0
B: 1
C: 2
D: 3