Student ID: **Student Name:**

Signature: **Course Number:**

Department of Electrical and Computer Engineering
SE465/ECE453/CS447/ECE653/CS647 Software Testing, Quality Assurance & Maintenance
Instructors: Lin Tan & Patrick Lam
Examination Date and Time: Tuesday April 21, 2015, 12:30 PM–3:00 PM
Rooms: PAC 4,5,6,7

Instructions:

- You have **2 hours and 30 minutes** to complete the exam.
- You can bring **printed or handwritten** material, e.g., books, slides, notes, etc.
- If you separate the pages, make sure your names and student IDs are on the top of every page.
- Unauthorized duplication or archival of these questions is not permitted. To be returned after the completion of the exam.
- If information appears to be missing from a question, make a reasonable assumption, state your assumption, and proceed. Do not simplify the question.
- Attempt to answer questions in the space provided. If necessary, you may use the back of another page. If you do this, please indicate it clearly.
- **Illegible answers receive NO points.**

Question	Mark	Points
Q1		**13**
Q2		**10**
Q3		**15**
Q4		**8**
Q5		**12**
Q6		**10**
Q7		**10**
Q8		**12**
Total:		**90**

1 Short Answers: Coverage Criteria, Tools (13 points)

Briefly answer the following questions. Illegible answers receive no points.

(a) Why aren't the coverage criteria T-Wise Coverage (TWC), Prime Path Coverage (PPC) and All Uses Coverage (AUC) more widely used in industry? List three reasons. Each reason should apply to at least one of the criteria above. [3 points]

(b) Give an example of a bug found more easily by a dynamic analysis tool than by a static analysis tool. Explain why. [2 points]

(c) Give an example of a bug found more easily by a static analysis tool than by a dynamic analysis tool. Explain why. [2 points]

(d) Give an example of a false positive returned by a static analysis tool. [2 points]

(e) List two fuzzing testing tools. [2 points]

(f) The following program contains a bug. Can Helgrind find this concurrency bug? Why or why not? [2 points]

```
1  // Expected behavior:
2  //  print the total number of threads.
3
4  #include <stdio.h>
5  #include <pthread.h>
6
7  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
8  int counter = 0; // shared variable
9
10 void * func() {
11         int tmp;
12
13         pthread_mutex_lock(&mutex1);
14         tmp = counter;
15         pthread_mutex_unlock(&mutex1);
16         tmp++;
17         pthread_mutex_lock(&mutex1);
18         counter = tmp;
19         pthread_mutex_unlock(&mutex1);
20 }
21
22 int main() {
23         pthread_t thread1, thread2;
24         pthread_create (&thread1, 0, func, 0);
25         pthread_create (&thread2, 0, func, 0);
26         pthread_join (thread1, 0);
27         pthread_join (thread2, 0);
28         printf("%d\n", counter);
29         return 0;
30 }
```

2 Coverage Criteria and Subsumption [10 points]

We define Simple Test Path Coverage (STPC) as follows. A simple test path is a path that is both a simple path and a test path. The set of test requirements for STPC contains all simple test paths.

(a) Does STPC subsume Node coverage (NC)? If yes, justify your answer. If no, give a counterexample. Simply saying yes or no receives no points. [5 points]

(b) Does Prime Path Coverage (PPC) subsume STPC? If yes, justify your answer. If no, give a counterexample. Simply saying yes or no receives no points. [5 points]

3 Graph Coverage [15 points]

The following functions, together, implement the **Pancake Sort** algorithm for an array of integers. The algorithm works on the principle that a spatula can be inserted at any point in a stack of pancakes, and then used to flip all pancakes above it. The algorithm considers each integer value to represent the diameter of a pancake in the stack, and the order of the integers denote the order of the pancakes in that stack. Index position 0 represents the top of the stack. The largest index position represents the bottom of the stack. A sort is a sequence of flips which arranges the pancakes from smallest to largest, with the largest pancakes at the bottom. The programmer invokes the `pancakeSort()` function on an array of integers to sort the integers.

```
1  public static void pancakeSort(int [] pancakes) {
2      if (pancakes.length <= 1)    // nothing to flip
3          return;
4      int i, the_spot;
5      // analyze the pancake stack from bottom to top
6      for (i = pancakes.length;
7           i > 1;
8           i--) {
9          // find the index of the largest pancake not yet sorted
10         the_spot = findFlipSpot(pancakes, i);
11         // if this pancake is already in place, continue
12         if(the_spot == i - 1)
13             continue;
14         // else, flip this pancake to index 0
15         if(the_spot > 0)
16             flip(pancakes, the_spot + 1);
17         // then flip this pancake to its place
18         flip(pancakes, i);
19     }
20     return;
21 }
22
23 // finds the largest pancake in the stack from index 0 to i
24 private static int findFlipSpot(int [] pancakes, int i) {
25     int the_spot = 0;
26     for(int eh = 0; eh < i; eh++) {
27         if(pancakes[eh] > pancakes[the_spot])
28             the_spot = eh;
29     }
```

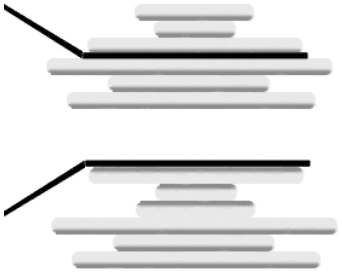

Figure 1: Pancake Flip

Method `findFlipSpot` is included for your information only. Do not include it in any graphs.

(a) Draw the minimal node (hint: 10) control flow graph (CFG) for `pancakeSort()`. Include line 1, which defines the `pancakes` variable. You must use only the line numbers provided above as the content of each node (e.g., 20–25, if lines 20–25 belong to a basic block), and label the nodes using only circled lowercase letters (ie nodes (a), (b), (c), (d), (e), ..., and (j)) sorted by first line number. Ensure the graph is clear and legible. Unclear graphs will not receive full points. [5 points]

(b) Draw the def-use CFG for variable `pancakes` of the function `pancakeSort()` with respect to your CFG in part (a) by making a copy of the diagram, then replacing line numbers with the def-sets and use-sets of the variable `pancakes`. Include the implicit definition at line 1 of the function. Note that `flip()` is a use and *then* a def of `pancakes` (the use is before the def), and `findFlipSpot()` is a use of `pancakes`. [4 points]

(c) Enumerate the du-paths and list the resulting test requirements for All-Uses Coverage (AUC) for variable `pancakes` for the function `pancakeSort()`. Justify your answers. [3 points]

(d) List a minimal test set that satisfies All-Uses Coverage (AUC) with respect to variable `pancakes`. Justify your answers. Just list test paths; don't specify the input and expected output. [3 points]

[Write your Q3 answer on this page.]

4 Applications of Statement Coverage [8 points]

Test case prioritization schedules test cases for regression testing in an order that attempts to maximize specific objectives, e.g., achieving coverage as fast as possible. *Total statement coverage* and *additional statement coverage* are two kinds of coverage-based test case prioritization strategies.

Total statement coverage: sorts test cases in descending order of total number of statements covered by each test case. If two test cases cover the same number of statements, randomly select one.

Additional statement coverage: selects, in turn, the next test case that covers the maximum number of statements not yet covered. If two test cases cover the same number of additional statements, randomly select one.

In this question, you will apply these two test case prioritization strategies on a sample program with 8 statements and 6 test cases. Each test case covers at least one statement. We've computed test case coverage for you in the following table. TC stands for "test case". An × means that a test case covers a statement, e.g, test case TC1 covers statements s1, s3, s5, and s6.

Statements	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
s1	×				×	×
s2		×			×	
s3	×	×			×	×
s4		×	×			
s5	×		×			×
s6	×					×
s7						×
s8				×		

(a) Write down prioritization orders with respect to *total statement coverage* and *additional statement coverage*. A prioritization order is a sequence of test cases. [6 points]

(b) For the sample program, which test case prioritization strategy do you think is better? Explain why. [2 points]

5 Applying Logic Coverage [12 points]

The following code calculates the final grade of some course (not this one) based on a student's term performance.

```
1  static double computeGrade (double project, double assignment, double midterm, double fin) {
2      double normal = 0.35 * project + 0.4 * assignment +
3          0.05 * midterm + 0.2 * fin;
4      double weightedMF = (0.15 * midterm + 0.5 * fin) / 0.65;
5
6      if (fin == 0 || midterm == 0)
7         return 0;
8
9      double finalGrade;
10     if (fin > 90) {
11        finalGrade = Math.max(normal, fin);
12     } else if (weightedMF < 50) {
13        finalGrade = Math.min(normal, weightedMF);
14     } else {
15        finalGrade = normal;
16     }
17
18     return finalGrade;
19 }
```

(a) Write down the predicates found at the following lines: [3 points]

Predicate #	Line Number	Predicate
1	6	
2	10	
3	12	

(b) List reachability conditions below. You need not simplify the predicates. You may define macros (e.g., `P1 := (final > 0)`) to shorten your answer. Include the reachability condition for the `else` as #4. [6 points]

Predicate #	Reachability Condition
1	
2	
3	
4	

(c) Fill in a set of concrete values which will satisfy predicate coverage (PC). [3 points]
(project = P; assignment = A; midterm = M; fin = F)

	True				False			
Predicate #	P	A	M	F	P	A	M	F
1								
2								
3								

6 Mutation Testing [10 points]

Consider the following original method:

```
1  static int findMax(int a[], int aLength) {
2      // if a is null, throw NullPointerException
3      // if aLength <= 0, undefined behaviour
4      int max = a[0];
5
6      for (int i=0; i < aLength; i++) {
7          if (a[i] > max) {
8              max = a[i];
9          }
10     }
11     return max;
12 }
```

Here is a mutant of the above method:

```
1  static int findMax(int a[], int aLength) {
2      // if a is null, throw NullPointerException
3      // if aLength <= 0, undefined behaviour
4      int max = 0; // <-- mutation
5
6      for (int i=0; i < aLength; i++) {
7        if (a[i] > max) {
8          max = a[i];
9        }
10     }
11     return max;
12 }
```

A test case includes the value of `a[]`, the return value, and the expected value. You may assume that `aLength` is the length of `a`.

(a) Write a test case that weakly kills (but does not strongly kill) the mutant. Justify your answer. [5 points]

(b) Write a test case that strongly kills the mutant. Justify your answer. [5 points]

7 Input Space Partitioning [10 points]

In their 2009 paper, Molnar et al.[1] use symbolic execution and a constraint solver to generate inputs that identify integer bugs, including bugs caused by integer overflows/underflows, width conversions, and signed/unsigned conversions. In this question, you will use input space partitioning to generate test cases to locate one such bug.

Consider the following C function, which copies a specified amount of memory.

```
1  int contrived_function(char * num_bytes_str, size_t buffer_size, char * p, char * q) {
2      if (num_bytes_str == NULL || *num_bytes_str == '\0')
3          return -1;
4
5      char * last_valid_char;
6      signed long num_bytes;
7
8      // Convert num_bytes_str to long
9      num_bytes = strtol(num_bytes_str, &last_valid_char, 10);
10
11     // Was the (whole) string a valid long?
12     if (errno == ERANGE || *last_valid_char != '\0')
13         return -1;
14
15     // Wouldn't want to copy more than the size of the buffer..
16     if (num_bytes > buffer_size)
17         return -1;
18
19     memcpy(p, q, num_bytes);
20     return 0;
21 }
```

Note 1. `strtol()` converts a string to a signed long integer corresponding to the string's value, scanned from the beginning to the first invalid character; if its input is out of range, `strtol()` sets `errno` to ERANGE.

Note 2. `size_t` is defined by the C standard to be unsigned; assume it is the same size as a `signed long`. Signed types can store numbers from $[-N, N-1]$ while signed types can store from $[0, 2N-1]$.

(a) Identify the signed/unsigned conversion bug in the code. [2 points]

(b) Write a complete, disjoint interface-based characteristic (i.e. set of blocks) for inputs `num_bytes_str` and `buffer_size`. The characteristic for `num_bytes_str` should have at least 5 blocks, while the characteristic for `buffer_size` should have at least 3 blocks. Briefly justify completeness and disjointness. Identify the combinations of blocks that trigger the bug. [4 points]

(c) Write a set of test requirements that satisfies Base Choice Coverage along with corresponding tests. Also identify which of your tests trigger the bug. [4 points]

[1] David Molnar, Xue Cong Li, and David Wagner. "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs." In: *USENIX Security Symposium*. 2009, pp. 67–82.

8A Test Design [12 points]

Choose one of either question 8A or 8B. If you answer both, we will mark only the first one.

As discussed in lecture in SE465-001, software engineering concepts such as design patterns also apply to test code. Test code can be well-designed or it can be poorly-designed. It is possible to improve test quality by refactoring tests. Consider the following JUnit test.

```
1    @Test
2    public void frob() {
3      Product product = createProduct();
4      Invoice invoice = createInvoice(product);
5
6      List<LineItem> lineItems = invoice.getLineItems();
7
8      if (lineItems.size() == 1) {
9          LineItem actual = lineItems.get(0);
10         assertEquals(actual.getInvoice(), invoice);
11         assertEquals(actual.getProduct(), product);
12         assertEquals(actual.getQuantity(), 5);
13         assertEquals(actual.getPrice(), new BigDecimal("69.96"));
14     } else {
15         fail("Invoice should have exactly one line item.");
16     }
17   }
```

Apply two techniques for improving tests to the `frob()` test (preferably those seen in lecture). That is, write down an improved version `frob()` which preserves its behaviour. Then, briefly describe your two design improvements and how they help.

8B Input Space Partitioning II [12 points]

Choose one of either question 8A or 8B. If you answer both, we will mark only the first one.

Consider the following C++ binary search implementation and the 3 partitions below:

```
1  int binary_search(vector<object>& vec, int start, int end, object& key)
2  {
3      if (start > end) { return -1; }  // Termination condition: start index greater than end index
4      int middle = start + ((end - start)/2);
5      if (vec[middle] == key) {
6          return middle; }
7      else if (vec[middle] > key) {
8          return binary_search(vec, start, middle - 1, key); }
9      return binary_search(vec, middle + 1, end, key);
10 }
```

Characteristic 1: vec contents.
- B1 vec is `null`.
- B2 vec is empty.
- B3 vec has at least one element.

Characteristic 2: relation between key and vec.
- B4 key in vec.
- B5 key not in vec.

Characteristic 3: relation between start and end.
- B6 start $>$ end.
- B7 start $=$ end.
- B8 start $\leq$ end.

(a) Does the partition "vec contents" satisfy the completeness and disjointness properties? If yes, justify your answer. If not, give a value of vec that does not fit any block. [4 points]

(b) Write test requirements for Pair-Wise Coverage (PWC) for the three partitions. You may write Bi–Bj to indicate blocks Bi through Bj, and we'll expand that in the obvious way. Indicate requirements that are not feasible. [8 points]