1. (1pt) **Name:** _____

INSTRUCTIONS: Carefully read each question, and write the answer in the space provided. If answers to free response questions are written obscurely, zero credit will be awarded. The correct answer to a free response question with a short answer (i.e., one word or phrase) will never contain any significant words used in the question itself (i.e., "crossword rules"). You are permitted to use one 8.5x11 inch sheet of paper (double-sided) containing **hand-written** notes; all other aids (other than your brain) are forbidden. Questions may be brought to the instructor.

You have **90 minutes** to complete the exam.

For **TRUE** or **FALSE** and multiple choice questions, circle your answer.

On free response questions only, you will receive **20%** credit for any question which you leave blank (i.e., do not attempt to answer). Do not waste your time or mine by making up an answer if you do not know. (Note though that most questions offer partial credit, so if you know part of the answer, it is almost always better to write something rather than nothing.)

To get credit for this question, you must:

- Print your name (e.g., "Martin Kellogg") in the space provided on this page.
- Print your UCID (e.g., "mjk76") in the space at the top of **each** page of the exam.

Contents (blanks for graders only):

Writing your name and UCID:	1 / 1
I. Very Short Answer:	17 / 17
II. Matching:	20 / 20
III. Short Answer:	24 / 24
IV. DBQs:	38 / 38
V. Extra Credit:	1 / 0
Total:	10x / 100

I. Multiple Choice and Very Short Answer (17pts). In the following section, either circle your answer (possible answers appear in **bold**) or write a very short (one word or one phrase) answer in the space provided. No partial credit is possible in this section.

2. For each of the following pairs of types of coverage, circle the one that subsumes the other. If neither subsumes the other, circle neither. If they subsume each other (that is, if they are equivalent), circle both.

   (a) (1pt) **branch coverage** / **statement coverage**

   (b) (1pt) **branch coverage** / **path coverage**

   (c) (1pt) **decision coverage** / **statement coverage**

   (d) (1pt) **condition coverage** / **decision coverage**

3. (2pt) For which of the following kinds of programs would a grammar-based fuzzer be well-suited, compared to a mutational fuzzer? (Circle all answers that apply.)

   A a PDF viewer

   B a Mario-style platformer game

   C a search engine

   D a calculator

4. (2pt) When running a service like a website, you can do **A/B testing** to quantify the impact of small changes to your service on user behavior.

5. (1pt) **TRUE** or **FALSE**: you can directly compare the mutation scores of two test suites for the same program: higher scores are better.

6. (2pt) In practice, fuzzing often finds security bugs, since **buffer overflows** are the largest cause of such bugs; a fuzzer can detect them with a bit of extra instrumentation.

7. (2pt) The typical way to show that a problem is **undecidable** is to reduce it to the halting problem.

8. (2pt) Test-driven development requires an engineer to actually **run** the new test to show that it fails before beginning to fix a bug.

9. (2pt) The **trivial compiler equivalence** technique for detecting equivalent mutants takes advantage of common semantics-preserving program analyses.

A. path predicate	**B.** coupling hypothesis	**C.** concolic testing	**D.** edge case
E. unit testing	**F.** race condition	**G.** fitness function	**H.** genetic algorithm
I. regression testing	**J.** satisfying assignment	**K.** partition testing	**L.** competent programmer hypothesis
M. flaky test	**N.** halting problem	**O.** implicit oracle	**P.** property-based testing
Q. mutation score	**R.** branch coverage	**S.** arid statement	**T.** pretty printing

II. Matching (20pts). This section contains a collection of terms discussed in class in an "Answer Bank" (choices **A.** through **T.**). Each question in this section describes a situation associated with an answer in the Answer Bank. Write the letter of the term in the Answer Bank that best describes each situation. Each answer in the Answer Bank will be used at most once.

10. (2pt) **S. arid statement** Baruch's mutation testing tool avoids mutating certain code patterns, such as code that writes to logs.

11. (2pt) **F: race condition** Cong is debugging a problem that seems to only occur when threads are scheduled in a particular way.

12. (2pt) **P: property-based testing** Instead of an oracle that specifies a concrete output value, Iulian writes a logical formula over program variables that should be true on any execution.

13. (2pt) **C: concolic testing** Yiannis is using symbolic execution to generate tests, but his solver keeps timing out on a particular formula. He decides to run the program with a particular input to see what the actual value of that formula is at run time.

14. (2pt) **Q: mutation score** Kasthuri's manager is asking her to achieve 100% of this, but she argues that she cannot, because program equivalence is undecidable in general.

15. (2pt) **G: fitness function** Shantanu is reading the automated program repair literature, and he notices a common issue with this: it is often defined by the number of test cases that pass, which is not continuous.

16. (2pt) **A: path predicate or J: satisfying assignment** Zephyr wants to know the input values that will make the program reach a specific program point.

17. (2pt) **K: partition testing** Jing splits the input space for her program in half along three important axes, and then writes just one test for each of the eight combinations.

18. (2pt) **L: competent programmer hypothesis** Przem believes that most programs are nearly correct, and that most bugs are small and can be corrected with a few keystrokes.

19. (2pt) **I: regression testing** Xiaoning believes in monotonically-increasing code quality, and this belief informs his testing strategy.

III. Short answer (24pts). Answer the questions in this section in at most three sentences.

20. (4pt) Support or refute the following claim: when performing refactoring, differential testing is both *applicable* (i.e., refactoring is a situation where it makes sense to use differential testing) and *effective* (i.e., differential testing is likely to produce good outcomes when applied in a refactoring context). **Likely support. When refactoring, you're usually trying to improve a non-functional property of a software system. You can differentially test the system before the refactor with the system after the refactor (on any random inputs!) to check that you did not unintentionally introduce semantic changes..**

21. (4pt) Suppose that you are a software engineer at Goldman's Socks, a bank named after footwear. You are implementing a new financial product for consumers that has an "account balance" that is measured in US dollars—that is, in real money. When implementing the `AddMoney` and `RemoveMoney` features of this new financial product, your manager directs you to be "extra careful" to get it right. You recall from CS 684 that *metamorphic testing* is one of the most effective testing techniques, when it is applicable: it has great value in terms of increasing your confidence in a system's correctness vs effort you need to put in! Design a metamorphic relation for each of the `AddMoney` and `RemoveMoney` features that you could use to increase your confidence that your implementations are correct. **The best and simplest answer is that `AddMoney` should be monotonically increasing, and `RemoveMoney` should be monotonically decreasing. That is, the account balance should be bigger after a call to `AddMoney`, and smaller after a call to `RemoveMoney`. Another full credit answer is that `AddMoney` and `RemoveMoney` are inverses, so for all X, the account balance is the same after calling `AddMoney(X); RemoveMoney(X)`.**

22. (6pt) Provide a minmal MC/DC-adequate test suite for the following expression: `(a && (b || !c))`
You may assume that `a`, `b`, and `c` are booleans. Write only the test cases in the space provided here; if you need space to work, there are blank sheets at the end of the exam. **The following four test cases (numbered 1-4, with the "->" notation to show the overall result) suffice: 1. a=0, b=0, c=0 -> 0**
2. a=1, b=0, c=1 -> 0
3. a=1, b=0, c=0 -> 1
4. a=1, b=1, c=1 -> 1

Independence is shown for a by 1 vs 3, for b by 2 vs 4, and for c by 2 vs 3. You need at least one TC with b=0, c=1 for decision coverage.

23. (4pt) Suppose that you are a site reliability engineer at NetFlocks, a company that provides an immersive virtual bird-watching experience for bird-watching enthusiasts. You have been tasked with reducing the frequency and duration of *cascading outages*: that is, outages in microservices that are caused by bugs or degraded performance in one of NetFlocks' other microservices (which the first microservice depends on). Describe an automated or mostly-automated testing strategy that you could implement to detect situations where such cascading outages might occur. **The easiest answer here is to import NetFlix's Chaos Monkey approach wholesale; suggesting that gets full credit. Other answers are possible, but without a strong justification answers like "fuzzing" get only 1 or 2 points, depending on argument strength.**

24. When using symbolic execution to generate test cases, recall that it is generally not possible to enumerate all of the paths in a particular method.

    (a) (2pt) Why not? Justify your answer. **There could be infinitely many paths, because of loops. 1 point for infinitely many, another for loops.**

    (b) (2pt) In one sentence each, describe two specific techniques that a symbolic execution tool could use to approximate the paths in a method instead. **Any two of the following from the slides are good answers:**

    - **Consider only acyclic paths (corresponds to taking each loop zero times or one time)**
    - **Consider only taking each loop at most k times**
    - **Enumerate breadth first or depth first, and stop after k paths have been enumerated**
    - **Concretely execute the program and see what it does (i.e., concolic testing)**

    (c) (2pt) Compare and contrast the two specific techniques in your answer to question 24b by briefly describing two situations: one where the first strategy is better in some way, and another where the second strategy is better. **Answers vary, but generally anything reasonable here gets full credit as long as it is well-explained. "CTE" (carried-through-error) scoring is used, so even if you got part b) wrong, I evaluated your answer; some people got full credit here despite totally nonsensical answers to part b.**

IV. Document-based Questions (38pts). All questions in this section refer to a documents **A-C**. These documents appear at the end of the exam (I recommend that you tear them out and refer to them as you answer the questions).

25. For each of the following kinds of coverage, rank the tests suites in **Document B** (for the `foo` method in **Document A**) from lowest coverage to highest coverage. Write your answer in the form "X < Y = Z", using "<" to show test suites that differ on the given metric and "=" to show test suites that are the same. (No partial credit on subquestions.)

    (a) (2pt) statement coverage: **Z < X < Y**

    (b) (2pt) branch coverage: **Z < X < Y**

    (c) (2pt) path coverage: **X = Y = Z**

    (d) (2pt) condition coverage: **Z < X < Y**

26. (4pt) Support or refute the following claim: on *this particular program*, two of these coverage metrics are always equal, for any test case. **Must support: branch coverage and condition coverage are the same on this program, because each branch is controlled by a single atomic condition.**

27. (8pt) Propose a pair of realistic mutants that could be generated by an automatic mutation testing tool like the one you wrote for HW6 that differentiate test suites X, Y, and Z by mutation score. Write your mutants in the form "L: S -> T", where "L" is the line number you are mutating, "S" is the statement that you are mutating, and "T" is the mutated statement. Answers with more than two mutations or that fail to differentiate two of the test suites can receive partial (but not full) credit. For this question, you may assume that "doesn't terminate in < 30 seconds" also kills a mutant. **Z cannot detect changes to line 10, so the easiest way to differentiate it from X and Y is to change the constant returned there. The easiest way to differentiate X and Y is to mutate line 5, which only X and Y execute, in such a way that one of them gets a different answer than the other. In particular, the difference between their third test case is useful here. Another option exists because X cannot detect changes to line 7, so one could differentiate X and Y with a mutation on line 7 that Z cannot detect (but that Y can). This is difficult, though, because every Z test case executes line 7. Regardless, the answer should result in mutation scores of 0 for Z, 50% for Y/X, and 100% for X/Y. An example solution is "10: -1 -> -2", "5: e - 1 -> 0". Half-credit for failing to differentiate X and Y.**

28. Consider the set of potential invariants in **Document C**, for the program in **Document A**.

(a) (10pt) A Daikon-like dynamic invariant detection algorithm could **NOT falsify** which of these invariants, if given test suite X (from **Document B**) as input? (Just write the Roman numerals of the unfalsified invariants.) **II, IV, VI, XII, XIV, XVI.**
Negative scoring: -1 for each wrong or missing answer.

(b) (5pt) Which invariants in **Document C** are actually true of the program? (Hint: there is at least one difference between the answer to this question and question 28a.) **VI, XIV, and XVI are real invariants. II and IV are false because there is nothing preventing the user from passing b and c in such a way that at the start of the program, they are false (however, the program will crash in that case!).**

(c) (3pt) Write a new test case that, if added to test suite X, would cause a Daikon-like algorithm to get the right answer (aka, match your answer to question 28b). **Any test where b is larger than c gets full credit, regardless of the oracle.**

V. Extra Credit. Questions in this section do not count towards the denominator of the exam score.

29. (1pt) In section III (Matching), there is a theme to the names used in the situation descriptions. What is the theme? **First names of tenure-track NJIT CS professors.**

30. (1pt) Method `foo` in Document A implements a classic algorithm. Which one? **binary search**

31. (2pt) Which of the proposed invariants in Document C are *loop invariants* for the loop in the program in Document A? Would any of these loop invariants be useful for proving that this program is correct? **XIV and XVI are loop invariants, but not particularly useful ones (any interesting proof about binary search will require invariants about the target (d) and the array, too, but those aren't in the set in Document C).**

32. (1pt) Explain something interesting that you learned in this class, but that this exam didn't test. **Any reasonable answer gets the point.**

33. (1pt) How could we apply one of the testing techniques we've discussed in this course to testing the correctness of the output of a chatbot backed by a large language model, like ChatGPT? **Any reasonable answer gets the point.**

Document A:

```
0:  # code is Python 2.7, but that shouldn't matter for this question
1:  def foo(a, b, c, d):
2:    while (b <= c):
3:      e = b + (c - b) / 2
4:      if (d < a[e]):
5:        c = e - 1
6:      elif (d > a[e]):
7:        b = e + 1
8:      else:
9:        return e
10:   return -1
```

Document B:

```
# Setup
a123 = [1, 2, 3]
a12345 = [1, 2, 3, 4, 5]

# Test suite X:
assert(foo(a=a123, b=0, c=2, d=2) == 1)
assert(foo(a=a123, b=2, c=2, d=2) == -1)
assert(foo(a=a12345, b=0, c=4, d=0) == -1)

# Test suite Y:
assert(foo(a=a123, b=0, c=2, d=1) == 0)
assert(foo(a=a123, b=0, c=2, d=3) == 2)
assert(foo(a=a123, b=0, c=2, d=0) == -1)

# Test suite Z:
assert(foo(a=[-1,0,1], b=1, c=2, d=1) == 2)
assert(foo(a=a12345, b=0, c=4, d=5) == 4)
assert(foo(a=[55,55,57,58,59,61,63], b=2, c=5, d=61) == 5)
```

Document C:

These invariants are in the form "L: I", where L is a line number and I is the invariant. The program location referenced by the invariant is always the location immediately BEFORE the given line. Each invariant is numbered with a Roman numeral; use these Roman numerals when answering questions about the invariants. "ret" denotes foo's return value. Location "P" refers to foo's preconditions; "Q" to its postconditions.

This page has two copies of the set of invariants, for your convenience. More copies are available from the proctors.

I.	P: b < c	XII.	4: e >= 0
II.	P: b <= c	XIII.	4: e > b
III.	2: b < c	XIV.	4. e >= b
IV.	2: b <= c	XV.	4: e < c
V.	3: b < c	XVI.	4: e <= c
VI.	3: b <= c	XVII.	5: e > c
VII.	3: b > 0	XVIII.	5: e >= c
VIII.	3: b >= 0	XIX.	7: e < b
IX.	3: c > 0	XX.	7: e <= b
X.	3: c >= 0	XXI.	Q: ret == d
XI.	4: e > 0	XXII.	Q: ret == -1

I.	P: b < c	XII.	4: e >= 0
II.	P: b <= c	XIII.	4: e > b
III.	2: b < c	XIV.	4. e >= b
IV.	2: b <= c	XV.	4: e < c
V.	3: b < c	XVI.	4: e <= c
VI.	3: b <= c	XVII.	5: e > c
VII.	3: b > 0	XVIII.	5: e >= c
VIII.	3: b >= 0	XIX.	7: e < b
IX.	3: c > 0	XX.	7: e <= b
X.	3: c >= 0	XXI.	Q: ret == d
XI.	4: e > 0	XXII.	Q: ret == -1

This page intentionally left blank (you may use it as scratch paper, and the proctor will have more scratch paper at the front if you need more).