LATE

minutes remaining

Hide Time

Manual Save

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit
- Pledge & Submit

Question 1. Word Bank Matching (1 point each, 14 points total)

For each statement below, input the letter of the term that is *best* described. Note that you can click each word (cell) to mark it off. Each word is used at most once.

A. — A/B Testing	B. — Agile Development	C. — Alpha Testing	D. — Beta Testing
E. — Competent Programmer Hypothesis	F. — Confounding Variable	G. — Dynamic Analysis	H. — Formal Code Inspection
I. — Fuzz Testing	J. — Integration Testing	K. — Milestone	L. — Mocking
M. — Oracle	N. — Pair Programming	O. — Passaround Code Review	P. — Perverse Incentives
Q. — Race Condition	R. — Regression Testing	S. — Risk	T. — Sampling Bias
U. — Software Metric	V. — Static Analysis	W. — Streetlight Effect	X. — Triage
Y. — Unit Testing	Z. — Waterfall Model		

Q1.1: The company GlazeBook wants to reward programmers for their bakery social media website with a pay raise for finding more bugs than other programmers. This leads to programmers intentionally creating and reporting new bugs, rather than finding and resolving existing ones.

Q1.2: When writing any sort of innovative software, developers often use several processes to account for this important source of uncertainty. Once considered, it is typically mitigated or reduced.

Q1.3: Startup winter social media company Sleddit just released their website and received over 1000 bug reports in the span of a week. They only have 10 programmers so they have to prioritize some bugs over others and decide which to address first.

Q1.4: Developers want to check how fast their program runs under a variety of conditions. They use execution time profiling to run their instrumented code on a variety of test inputs.

Q1.5: Train company StationWide wants to be reactive to changing requirements as they create software that shows users the trains they can ride from one place to another. They created a very early prototype of the software during a two-week sprint to get feedback and fix problems the original code had during the next sprint. Daily meetings are used to keep everyone on the same page.

Q1.6: For each class Aidan writes, they include local test cases to ensure that class is working as intended.

Q1.7: Video conferencing company Nyoom wants to test their latest chat feature to include custom emojis. To do so, they randomly generate 1000 valid and invalid emojis and use them in the chat feature to see whether the program responds as it should.

Q1.8: To check how well-received their new yellow colored buttons are, video-sharing company BlueTube sends out a survey to a group of people that only like the color green. Based on the survey results, BlueTube mistakenly concludes that the new yellow colored buttons would be disliked by all users.

Q1.9: After implementing a more advanced internet structure to their old fighting game, the company Twister sends a version of the game to a small number of end users to make sure it works before the feature releases.

Q1.10: BlipPng is a program that generates special pngs that automatically delete themselves after a period of time. They want to add a notification feature to tell users that the generated png has deleted itself. They release this feature to half of users but not to the other half of users. They include a survey to see how satisfied users are with the new feature.

Q1.11: After Mollie fixed a small mistake where they had a less-than (<) sign instead of a less-than-or-equal-to (<=) sign, Mollie's grade went from 5% to 100% on the Autograder.

LATE

minutes remaining

Hide Time

Manual Save

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit
- Pledge & Submit

Q1.12: Bird adoption company Flapple uses inexpensive functions with pre-determined outputs while initially testing their code base.

Q1.13: Daniel finds a bug in their code and fixes it. To prevent this same bug from recurring later, Daniel writes a test case to detect the presence of that bug.

Q1.14: Conner is trying to find which methods take longer to run. A first analysis finds that methods with more lines of code often have longer running times. However, this analysis does not account for the algorithmic complexity (e.g., Big-Oh) of the code. Ignoring that aspect means the analysis is misleading: some methods with fewer lines of code may still take a long time to run because they contain complex algorithms.

Question 2. Code Coverage (20 points)

You are given the following `C` functions. Assume that statement coverage applies only to statements marked **STMT_#**. In this question, we consider the **entire** program. That is, even if program execution starts from one particular method, we consider coverage with respect to the contents of all methods shown.

```
void Euphoria(str a, str b, int c, int d) {
    STMT_1;
    if (c < d) {
        medicine(b, a);
    }
    STMT_2;
    apple_juice(d, c);
}

void medicine(str a, str b) {
    STMT_3;
    if (a == 'rue') {
        STMT_4;
    }
    if (b == 'jules') {
        STMT_5;
    }
}

void apple_juice(int c, int d) {
    if (c == d) {
        STMT_6;
        return;
    }
    STMT_7;
    apple_juice(c, c);
    STMT_8;
}

```

(a) (6 points) Provide **1** input (i.e., all four arguments) to `Euphoria(str a, str b, int c, int d)` such that the *statement* coverage will be **50%**. (We only consider statements marked **STMT_#**.) Use a format such as `("hello", "goodbye", 123, 456)` if possible.

Your answer here.

(b) (2 points) **True / False**: there exists a test suite of size > 0 such that the test suite obtains **100%** *statement* coverage. (We only consider statements marked **STMT_#**.)

- ○ True
- ○ False

(c) (2 points) **True / False**: your answer from **Q2a** provides the lowest possible *path* coverage for the given code snippet.

- ○ True
- ○ False

(d) (5 points) Give a *minimum* test suite to reach **100%** *branch* coverage. Provide the test cases with their input in the form

`Euphoria(str a, str b, int c, int d)`. For `a` and `b`, choose from only the values `{'rue', 'jules'}`. For `c` and `d`, choose from only the values `{1, 2}`. Write each test input on a separate line, using a format such as `("hello", "goodbye", 123, 456)` for each input if possible.

LATE

minutes remaining

Hide Time

Manual Save

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit
- Pledge & Submit

Your answer here.

(e) (5 points) Describe a scenario in which a test suite that achieves 100% *statement* coverage might miss a bug in a program. Then describe what other approach (testing, coverage, analysis, etc.) could find that bug. Use 4 sentences or fewer.

Your answer here.

Question 3. Short Answer (5 points each, 25 points)

(a) (5 points) Consider the following two pairs of tools, techniques, or processes. For each pair, give a class of defects or a situation for which the *first* element performs better than the second (i.e., is more likely to succeed and reduce software engineering effort and/or improve software engineering outcomes) and explain why.

a. integration testing better than maximizing branch coverage
b. spiral development better than waterfall model

Your answer here.

(b) (5 points) Identify two risks associated with Netflix's adaptation and usage of the *Chaos Monkey* dynamic analysis. Identify a measurement that could be used to guide decisions to reduce each risk.

Your answer here.

(c) (5 points) Here are two examples of bugs that need to be triaged:

- A conversion error causes integers to occasionally flip signs (e.g., 4 becomes -4 and -4 becomes 4).
- A graphical error causes images to display 1.5x as large as expected, resulting in cropping.

For each bug, give an example of a situation where it would have high severity and a situation where the bug would have low severity and explain why.

Your answer here.

(d) (5 points) Give an example of a software situation where fuzzing would be a better testing method than unit testing in terms of finding many bugs. Then give a situation where unit testing would be a better testing method than fuzzing in terms of the time or cost required. What kinds of bugs are likely to be revealed by fuzzing?

Your answer here.

(e) (5 points) You are a new team lead at *Mozzarella* and are in charge of leading a group of several developers. Your manager asks you to begin collecting the following developer efficacy data:

LATE

minutes remaining

Hide Time

Manual Save

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit
- Pledge & Submit

- Lines of code written per day
- Pull requests accepted into the master branch per month
- Peer ratings from an annual survey completed by coworkers

For each measurement, describe why it might not accurately represent a worker's efficacy and explain one way a malicious worker might exploit it.

Your answer here.

Question 4. Mutation Testing & Invariants (15 points)

Consider the code snippet below defining a function `modest_liskov`.

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

(a) (5 points) A *postcondition* is similar to an *invariant*, but is always true just as or just after a function returns. (Informally, you can think of it as an assertion right at the end of the function.)

Consider the *postcondition*: `baz >= 7`.

The postcondition, `baz >= 7`, may be falsified by a first-order mutant of the original `modest_liskov` function. Create that mutant by making at most one edit to the below definition of `modest_liskov`. (To phrase this another way, you should make a single change to the program so that on some input it does not satisfy the postcondition.) Create the mutant by clicking inside the code window below and directly changing the initial code.

Mutant 1 (click inside to edit this directly):

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

(b) (10 points) Create two *additional* first-order mutants of `modest_liskov` by making *exactly* one edit to each of the following definitions of `modest_liskov`. These two should target the same postcondition as your first mutant. (There is only one postcondition: consider the same one each time.) Note that the mutants you create must be different from the original `modest_liskov`, the first mutant above, and from each other.

Below, you will then be asked to provide a single test input to `modest_liskov` such that the mutation adequacy score of your suite of three mutants, when each is given that single input, is exactly 1/3. We consider a mutant that fails to satisfy a postcondition as failing that test (i.e., such a mutant is killed). You may use this requirement to guide how you create the mutants.

Mutant 2:

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

Mutant 3:

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

What is a single test input to `modest_liskov` such that the mutation adequacy score for the three mutants is 1/3? All test inputs must be integers. Express your answer as a list in the form `[x, y, z]`. For example, if your inputs are `x = 3, y = 4, z = 5`, then you would write `[3, 4, 5]`.

Your answer here.

Question 5: Dataflow Analysis (11 points total)

Consider a *live variable dataflow analysis* for three variables, `a`, `x`, and `q` used in the control-flow graph below. We associate with each variable a separate analysis fact: either the variable is possibly read on a later path before it is overwritten (live) or it is not (dead). We track the set of live variables at each point: for example, if `a` and `x` are alive but `q` is not, we write `{a, x}`. The

LATE

minutes remaining

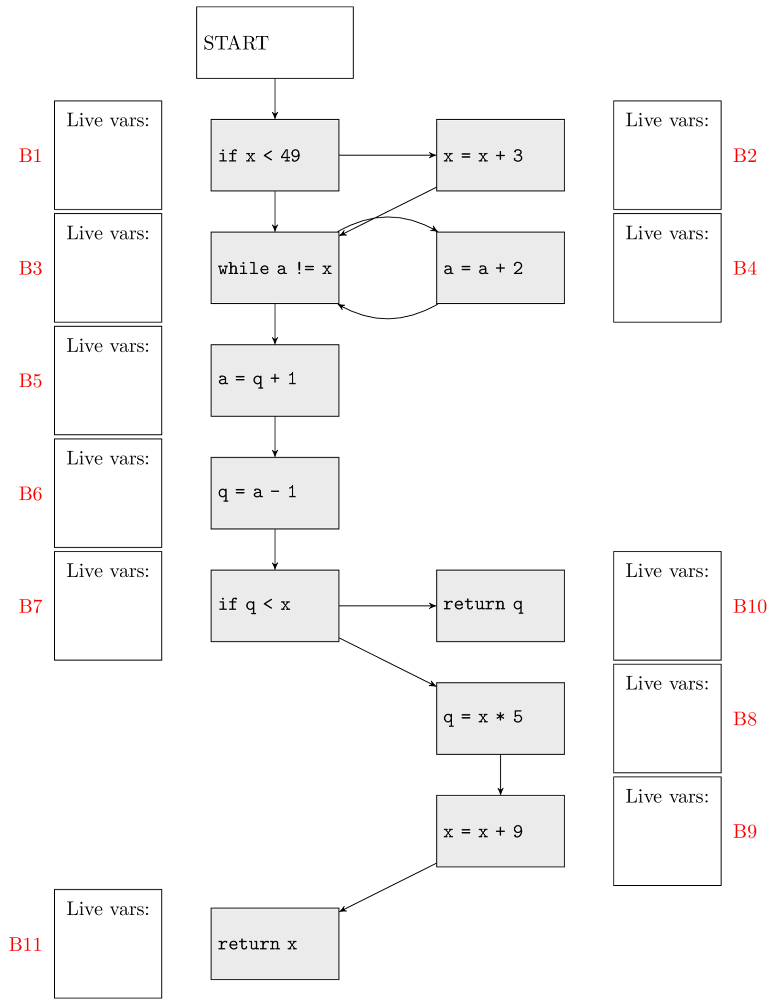
Hide Time

Manual Save

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit
- Pledge & Submit

special statement `return` reads, but does not write, its argument. In addition, `if` and `while` read, but do not write, all of the variables in their predicates. (You must determine if this is a forward or backward analysis.)



(1 point each) For each basic block `B1` through `B11`, write down the list of variables that are live *right before* the start of the corresponding block in the control flow graph above. Please list only the variable names in lowercase without commas or other spacing (e.g., use either `ab` or `ba` to indicate that `a` and `b` are alive before that block).

B1	B2	B3	B4
B5	B6	B7	B8
B9	B10	B11	

Question 6. Dynamic Analysis (15 points)

We decide to write our own dynamic analysis tool, Checkers, to help us deal with race conditions. Checkers works by following a standard *lockset* algorithm. For each shared variable, Checkers maintains a candidate set of locks that might protect that variable. The first time a shared variable is accessed by a thread, Checkers notes the set locks that thread currently holds. Every subsequent time that shared variable is accessed by a thread, the candidate set of locks guarding that variable is intersected with the currently-held locks of that thread. At the end, if a shared variable is not protected by any locks, a race condition is reported.

As part of its operation, Checkers instruments the program to log variable reads, variable writes, lock acquisition, and lock release. All such operations are instrumented to write the name and arguments of the operation, as well as a thread ID, to a log file.

(Note: This lockset algorithm works just like the one discussed in class. There are no hidden tricks or mistakes or changes in the description above, it is simply a summary for your convenience.)

(a) (2 points each, 4 points) We run Checkers on a series of programs and examine its output. For each of the programs below, consider if Checkers would report a race condition (i.e., if the computed lockset for a shared variable is empty) by examining the contents of the Checkers log file.

Variables with names that include `local` are thread-local variables that are not relevant for race conditions. Variables with names that include `shared` are shared variables that *can* be involved in race conditions. Variables with names that include `mu` are locks (short for *mutex* or *mutual exclusion*).

(ai) (2 points) Program:

```
int sharedA = 0;
mutex muA;
int sharedB = 0;
mutex muB;

void thread1() {
    muA.lock();
    muB.lock();
```

```
    sharedA = 10;
    sharedB = 20;
    muB.unlock();
    muA.unlock();
}

void thread2() {
    muB.lock();
    sharedB = 20;
    muA.lock();
    sharedA = 10;
    muA.unlock();
    muB.unlock();
}
```

Checkers log file:

```
thread 1: lock muA
thread 2: lock muB
thread 2: write sharedB
thread 1: lock muB
thread 2: lock muA
```

True/False: a race condition can be detected from the log file.

- ◯ True
- ◯ False

(aii) (2 points) Program:

```
int shared = 0;
mutex mu;

void thread1() {
    mu.lock();
    shared += shared;
    mu.unlock();
}

void thread2() {
    int local;
    local = 12;
    mu.lock();
    shared -= 2;
    mu.unlock();
}
```

Checkers log file:

```
thread 2: write local
thread 1: lock mu
thread 1: read shared
thread 1: read shared
thread 1: write shared
thread 1: unlock mu
thread 2: lock mu
thread 2: read shared
thread 2: write shared
thread 2: unlock mu
```

True/False: a race condition can be detected from the log file.

- ◯ True
- ◯ False

(b) (4 points) We view Checkers as an analysis for helping us to conclude that a program has no race conditions. In this view, Checkers is sound if and only if it reports all such defects (i.e., has no false negatives). Is Checkers a sound analysis? Is it complete? Explain your reasoning in at most four sentences.

Your answer here.

(c) (4 points) Support or refute the following statement: "A dynamic lockset algorithm such as Checkers is better suited than a static analysis tool would be for race condition detection."

Your answer here.

(d) (3 points) Suppose we want to test our dynamic analysis — that is, we want to gain confidence that it correctly reports a race condition if and only if the subject program has a race condition. To do so, we need a suite of subject programs for which we know whether each subject program has a race condition or not. Creating such a suite would be expensive. We decide to use just one part of mutation from mutation analysis: start with a single known-good program and randomly delete a call to lock or unlock to produce a new subject program that should now have a race condition. Support or refute the claim that using this simple part of mutation would be a good way to produce a test suite for Checkers. (Note that in this question a test input to the Checkers analysis is, itself, another program, which also has its own input. Note also that this question is about using a mutation operator, but is not about standard mutation analysis.)

Your answer here.

LATE

minutes remaining

Hide Time

Manual Save

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit
- Pledge & Submit

Extra Credit

Each question below is for 1 point of extra credit unless noted otherwise. We are strict about giving points for these answers. No partial credit.

(1) What is your favorite part of the class so far?

Your answer here.

(2) What is your least favorite part of the class so far?

Your answer here.

(3) If you read any optional reading, identify it and demonstrate to us that you have read it critically. (2 points)

Your answer here.

(4) If you read any *other* optional reading, identify it and demonstrate to us that you have read it critically. (2 points)

Your answer here.

(5) In your own words, identify and explain any of the bonus psychology effects or ethical considerations presented in class on the colored bordered slides or in a "long instructor post" on Piazza. (2 points)

Your answer here.

Honor Pledge and Exam Submission

You must check the boxes below before you can submit your exam.

- [ ] I have neither given nor received unauthorized aid on this exam.
- [ ] *I am ready to submit my exam.*

LATE

minutes remaining

Hide Time

Manual Save

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit
- Pledge & Submit

Note that your submission will be marked as late. You can still submit, and we will retain all submissions you make, but unless you have a documented extenuating circumstance, we will not consider this submission.

Submit My Exam

Once you submit, you will be able to leave the page without issue. Please don't try to mash the button.

The exam is graded out of 100 points.