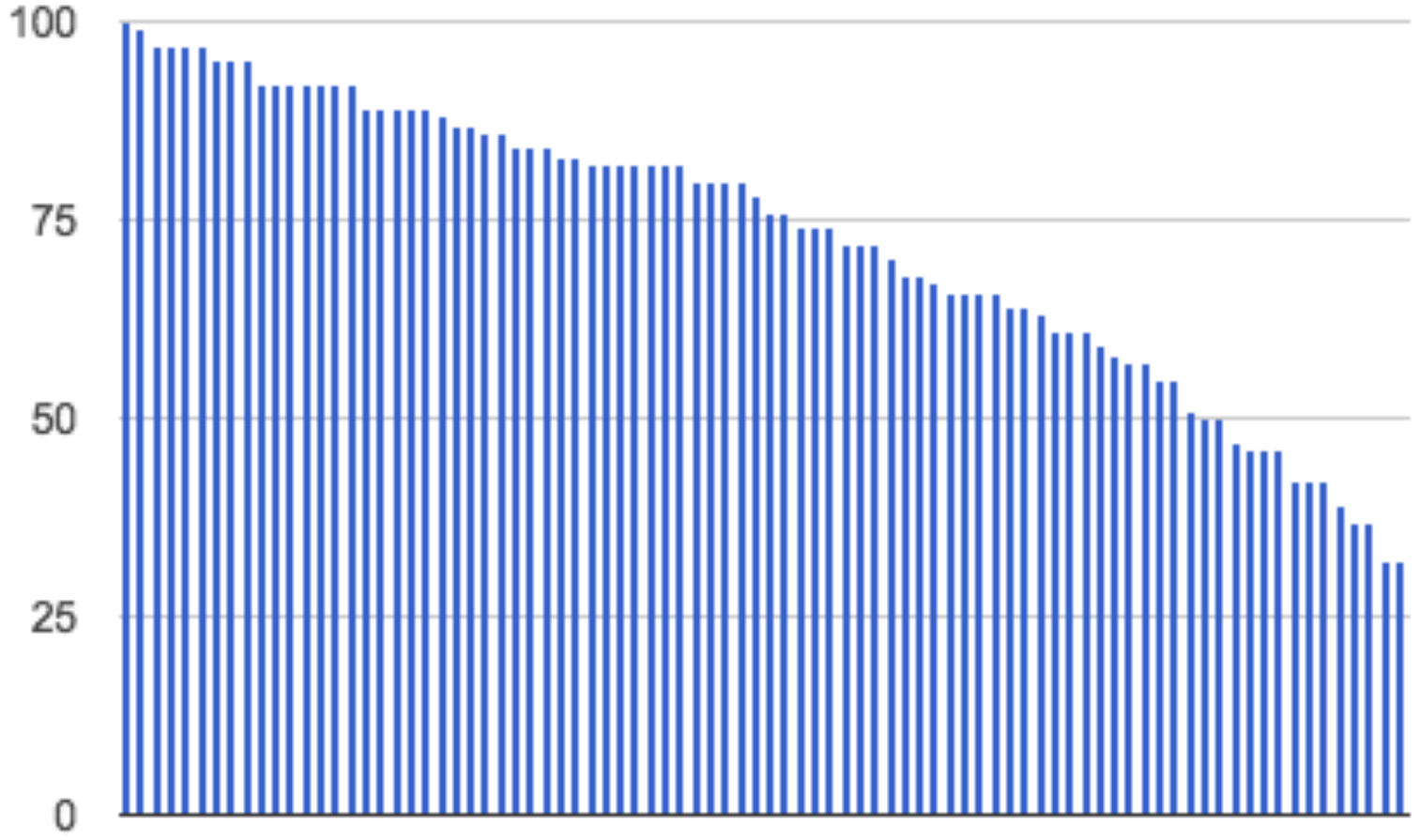
CPEN 422
Software Testing and Analysis

Midterm Review

	%
AVERAGE	73
MEDIAN	77
MAX	100
MIN	32



1.a. True/False

- (1 point) Software testing can be used to show the presence of bugs, but never to prove their absence.

"True. If a test case fails, it only shows that a bug exists in the system and is detected. If all test cases pass, we still don't have any proof that there are no bugs as software testing is **dynamic**, **partial**, **optimistic**, and thus **never complete**.

1.b. True/False

- (1 point) A 100% code coverage means that the test suite is effective in detecting bugs.

"False. Coverage does not show bug finding effectiveness of a test suite. For example you could have a test suite ***without any test assertions***. In this case, the test suite is not checking anything although the coverage might be 100%."

1.c. True/False

(1 point) Faults are observable manifestations of failures.

“False. Failures are manifestations of faults at runtime (not the other way around).”

1.d. True/False

(1 point) An **intra**-procedural control flow graph models the flow of program execution between functions.

“False. It models the flow of execution WITHIN functions not between (that would be **intra**-procedural or call graph)”

1.e. True/False

(1 point) Mutation testing is a test generation technique for assessing the quality of program code.

"False. There are two things wrong with this statement. First, mutation testing is **not a test generation technique**. Second, it is used for assessing the **quality of test cases** (not program code). Students should mention both to get the full mark."

1.f. True/False

(1 point) An equivalent mutant that remains alive after running a test suite T indicates that T is not effective and thus new test cases need to be written.

“False. Equivalent mutants are semantically the same as the original program and thus **cannot be killed** by writing new test cases. They do not indicate that the test suite is **ineffective**. ”

1.g. True/False

(1 point) Feedback-directed random testing uses inputs obtained from the execution of the same program to generate test cases more effectively.

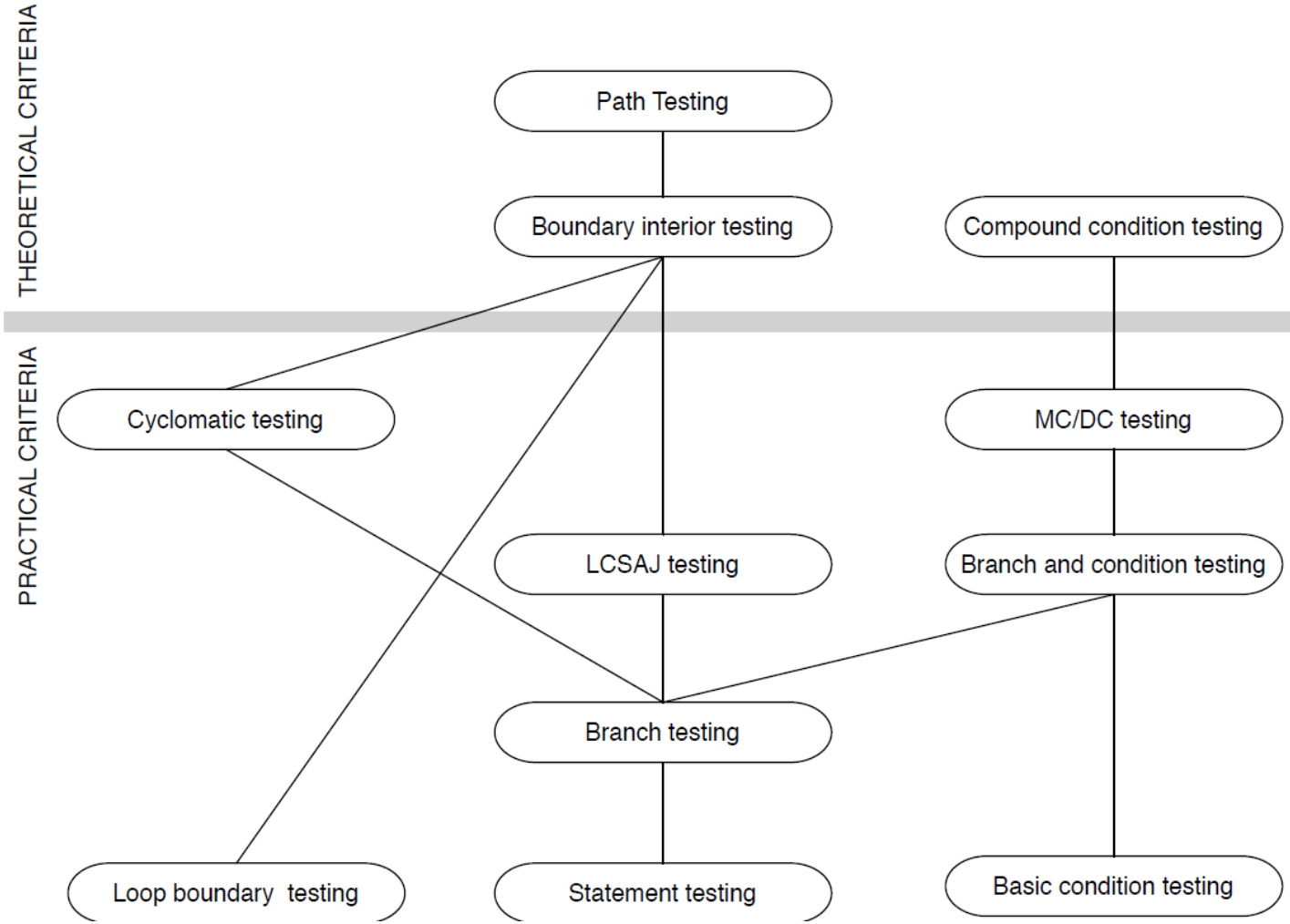
“True. It uses information from the execution of the same program to guide the random testing. ”

1.h. True/False

(1 point) Branch coverage subsumes basic condition.

“False. The two are incompatible (no relation)”

Subsumption relation



Wrong in the book on p.231!
See http://www.cs.uoregon.edu/~michal/book/errata.html

Open-ended Questions

Keep your answers short, neat, and to the point.

2.

Your colleague is developing a software component **C1**, which closely interacts with three other existing components **C2**, **C3**, and **C4**.

The components are independently released, and can be upgraded by end- users — a process that cannot be influenced. She wants to ensure correct functioning of the system for the available versions of each component. The versions are as follows:

- C1: 1, 2, 3
- C2: 1, 2
- C3: 1
- C4: 1, 2

2.a

(1 point) You advise her to take the risk of failing combinations into account. How many combinations would you test if the risk and cost of failure is **low**? And how many would you test if a failing combination could lead to **catastrophic consequences** (e.g., loss of life)?

Minimum: 3 test cases: once all at v1, once all but C3 at v2, and once C1 at 3.

Maximum: 12 test cases: 3*2*1*2 = 12 (all combinations)

2.b

(b) (1 point) As the risk is **moderate**, you propose the pairwise combination testing approach. Explain to her what pairwise combination testing is and how that can help her.

“Instead of testing all N-way combinations, just the 2-way (fixed pairs) combinations are tested. Saves time and resources while still error-prone combinations are tested.”

2.c

(3 points) Provide the **smallest** test suite making use of pairwise combination testing for the case at hand.

c1	c2	c3	c4
1	1	1	1
2	2	1	1
1	2	1	2
3	1	1	1
2	1	1	2
3	2	1	2

3 * 2 = 6 test cases max!

Some students gave more than 6 test cases, which can cover all pairs, but which uses more test cases than strictly necessary.

Q2.d

(d) (1 point) Under what circumstances do you consider pairwise combination testing suitable for the case at hand? Explain.

"Tradeoff between **cost of test execution** and **risks** involved in failure. If execution is cheap: just test all 16. If *expensive* and *independent*: just test two. Also: if reasonable to assume that only 2-way interactions occur, and not N-way, for N > 2. "

Q3

```
//Effects: If x==null throw NullPointerException
// else return the index of the last element in x that equals y.
// If no such element exists, return -1.

public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
          return i;
    }
  }
  return -1;
}
```

For the questions below when you are asked to provide test cases, write the test case in the form of: **Inputs, Expected output, and Actual output**

Q3.a

Identify the fault

```
public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
          return i;
    }
  }
  return -1;
}
```

Q3.a

Identify the fault

```
public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
          return i;
    }
  }
  return -1;
}
```

“The for-loop should include the 0 index:
for (int i=x.length-1; **i >= 0**; i–)”

Q3.b

Identify a test case that does not execute the fault.

```
public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
          return i;
    }
  }
  return -1;
}
```

Q3.b

Identify a test case that does not execute the fault.

```
public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
          return i;
    }
  }
  return -1;
}
```

A **null** value for x will result in a NullPointerException before the fault is evaluated. hence no execution of the fault.

Input: x=null; y=3

Expected Output: NullPointerException

Actual Output: NullPointerException

Q3.c

Identify a test case that executes the fault, but does not propagate it to a failure.

```
public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
          return i;
    }
  }
  return -1;
}
```

Q3.c

Identify a test case that executes the fault, but does not propagate it to a failure.

```
public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
         return i;
    }
  }
  return -1;
}
```

For any input where **y** appears in the second or later position, there is no error.
Also, if **x** is empty, there is no error.

Input: x=[2,3,5]; y=3;
Expected Output: 1
Actual Output: 1

OR input: x=[]; y=2
Expected Output: -1
Actual Output: -1

OR input: x=[2,4,7]; y=10
Expected Output: -1
Actual Output: -1

Q3.d

Identify a test case that can detect the failure.

```
public int findLast (int[] x, int y) {
  for (int i=x.length-1; i > 0; i--) {
    if (x[i] == y) {
         return i;
    }
  }
  return -1;
}
```

For any input where **y** appears in the first position,
input: x=[2,3,5]; y=2
Expected output: 0
Actual output: -1

Questions?