VU Programm- und Systemverifikation
Assignment 1: Assertions, Testing, and Coverage

Name: _____ Matr. number: _____

Due: April 30, 1pm

1 Coverage Metrics

Consider the following program fragment and test suite:

```
unsigned gcd (unsigned m, unsigned n) {
  unsigned i;
  if (m > n) {
    i = n;
  } else {
    i = m;
  }
  bool done = false;
  while ((i > 0) && !done) {
    if ((m % i == 0) && (n % i == 0) {
        done = true;
    } else {
        i = i - 1;
    }
  }
  return i;
}
```

Inputs		Outputs
m	n	return value
0	1	0
1	0	0
1	1	1
2	3	1

1.1 Control-Flow-Based Coverage Criteria (3 points)

Indicate (✓) which of the following coverage criteria are satisfied by the test-suite above.

	satisfied	
Criterion	yes	no
statement coverage	✓	
decision coverage	✓	
condition coverage		✓
modified condition/decision coverage		✓

For each coverage criterion that is *not* satisfied, explain why this is the case:

In decision `((m % i == 0) && (n % i == 0)`, condition `(m % i == 0)` is never false (test case (1,1), iteration 1: (true, true); test case (2,3), iteration 1: (true, false) , iteration 2: (true, true)). Therefore, condition coverage is not achieved; it follows that MC/DC is also not achieved.

1.2 Data-Flow-Based Coverage Criteria (4 points)

Indicate (✓) which of the following coverage criteria are satisfied by the test-suite above (here, the parameters of the function do not constitute definitions, the `return` statement is a c-use):

	satisfied	
Criterion	yes	no
all-defs	✓	
all-c-uses		✓
all-p-uses		✓
all-c-uses/some-p-uses		✓
all-p-uses/some-c-uses		✓

For each coverage criterion that is not satisfied, explain why this is the case:

all-c-uses: There are 3 reasons why `all-c-uses` is not achieved:

1. `i = n` is not executed in any test case that enters the loop, hence the definition is never used in `i = i - 1`.
2. Moreover, no test case visits `i = i - 1` in two subsequent iterations, therefore there is no definition-use chain with `i = i - 1` as first and last element, and the definition `i = i - 1` is never used in the computation `i = i - 1`.
3. Finally, since the return statement is a computational use, there must be a def-clear du-chain starting with the definition `i = i - 1` and ending with `return i`; however, all du-chains starting with `i = i - 1` visit the loop head twice and are therefore not def-clear.[1]

all-p-uses: `i = n` is not executed in any test case that enters the loop, hence the definition is never used in `((m % i == 0) && (n % i == 0)`.

all-c-uses/some-p-uses: because all-c-uses has not been achieved.

all-p-uses/some-c-uses: because all-p-uses has not been achieved.

[1]Note that in the given example, there *can't* be a def-clear du-chain from `i = i - 1` to `return i` that doesn't visit the loop head twice, since this would require `i` to be 0 after executing `i = i - 1`. However, that's not possible, since the branch `i = i - 1` is never entered when `i` is 1. Therefore, `all-c-uses` *cannot* be achieved in this example.

1.3 Achieving Full Coverage (1 point)

Consider the two coverage criteria below.

- If the test-suite from above does not satisfy the coverage criterion, augment it with the *minimal* number of test-cases such that this criterion is satisfied. If full coverage cannot be achieved, explain why.
- If the coverage criterion is already achieved, explain why.

MC/DC

Inputs		Outputs
`m`	`n`	result
3	2	1

all-p-uses

Inputs		Outputs
`m`	`n`	result
3	2	1

1.4 Modified Condition/Decision Coverage (1 point)

Consider the expression $((a \vee b) \wedge c)$, where a, b, and c are Boolean variables. Provide a *minimal* number of test cases such that modified condition/decision coverage is achieved for the expression. Clarify for each test case *which* condition(s)/value(s) independently affect(s) the outcome.

MC/DC

Inputs			Outcome	Covers
`a`	`b`	`c`	`(a || b) && c`	
0	0	1	0	a = 0, b = 0
1	0	1	1	a = 1, c = 1
0	1	1	1	b = 1, c = 1
1	1	0	0	c = 0

2 Equivalence Partitioning and Boundary Testing

The resources for performing RT-PCR tests to determine whether a patient has contracted the COVID-19 virus are extremely limited; consequently, critical patients will have to be prioritized. The function

```
priority triage (enum countries travel,
                 enum symptoms sympt,
                 int age);
```

is used to determine the priority with which a person should be tested or not. It uses the following data-types:

- `priority` is an enum type defined as `enum priority {high, medium, low};`
- `countries` is an enum type listing 196 countries used to represent the country the patient most recently traveled to (if any). It is defined as follows:

  ```
  enum countries { None = 0, China = 1, Iran = 2, Italy = 3, ...};
  ```

  The first entry (0) indicates that the patient has not traveled outside Austria recently; the following k entries are countries that are classified as critical, and the remaining $196 - k$ entries are countries that are (still) considered safe.

- `symptoms` is an enum type listing 100 symptoms defined as follows:

  ```
  enum symptoms {
    None = 0, Tiredness, Aches, Cough, Fever, ..., };
  ```

  The first entry (0) indicates that the patient has no symptoms, the following m symptoms are common symptoms of COVID-19, and the remaining symptoms ($m + 1$ to 100) are not known to be related to the new virus.

- The parameter `age` represents the age of the patient.

The function `triage` is supposed to implement the following rules:

- Patients who have no recent travel history to critical countries or show none of the typical symptoms are considered low priority.
- Patients who have traveled to a country classified as critical and report a relevant symptom are medium priority if they are younger than 65, and high priority if they are 65 and above.

2.1 Equivalence Partitioning (3.5 points)

From the specification above, derive equivalence classes for the function `triage`. Use the table below to partition them into *valid equivalence classes* (valid inputs) and *invalid equivalence classes* (invalid inputs). Label each of the equivalence classes clearly with a number (in the according column). For each correct *equivalence class* you can score $\frac{1}{2}$ a point (up to 3.5 points).
(Do not provide test-cases here – that's task 2.2)

2.1.1 Valid Equivalence Classes

Condition	**ID**
`travel` $= 0$	1
$1 \leq$ `travel` $\leq k$	2
$k <$ `travel` ≤ 196	3
`sympt` $= 0$	4
$0 <$ `sympt` $\leq m$	5
$m <$ `sympt` ≤ 100	6
$0 \leq$ `age` < 65	7
$65 \leq$ `age`$(< max)$	8

2.1.2 Invalid Equivalence Classes

Condition	**ID**
`travel` > 196	9
`sympt` > 100	10
`age` < 0	11
`age` $\geq max$	12 (optional)

2.2 Boundary Value Testing (3.5 points)

Use *Boundary Value Testing* to derive a test-suite for the function `triage`. Specify the inputs points for `triage`. Indicate clearly which equivalence classes each test-case covers by referring to the numbers from task (a). You can receive up to 3.5 points ($\frac{1}{2}$ a point per test-case), where redundant test-cases and test-cases that do not represent boundary values do not count.

Note:

- We should test all possible outputs `priority` $\in \{$`high`, `medium`, `low`$\}$.
- Equivalence classes can be *combined* in different ways (e.g., we can find test cases that cover $(1, 4, 6)$, $(2, 4, 6)$, $(3, 4, 6)$, $(1, 5, 6)$, $(2, 5, 6)$, $(3, 5, 6)$, ..., leading to a large number of test cases.[2] Ideally, we should test all of them, but which combinations do we pick if our resources are limited?
  - If there is a *dependency* between input variables (in our example, all of them are interdependent), then we should consider which combinations are relevant: for instance, the phrase "if patients have traveled to a critical country *and* report a relevant symptom" clearly indicates a dependency between `travel` and `sympt`; so we should pick test cases that cover the classes $(2, 4)$ and a test case outside $(2, 4)$. (Moreover, we should combine $(2, 4)$ with classes 6 and 7, since this influences the outcome of `triage`.) On the other hand, all patients who have no alarming travel history are considered low priority; therefore, including several combinations that include class 1 seems wasteful (think of the MC/DC criterion).
  - If there's no dependency between two variables (or equivalence classes, respectively), then testing their combinations will not result in any new behavior of the program.
- The set of test cases below is not "complete" (in the sense of the dependencies above); also the (combinatiosn of) boundary values are not exhaustively enumerated.

Input	**Output**	**Classes Covered**
`travel` $= 0$, `sympt` $= 1$, `age` $= 0$	low	1,5,7
`travel` $= 1$, `sympt` $= 0$, `age` $= 65$	low	2,4,8
`travel` $= k + 1$, `sympt` $= m$, `age` $= 64$	low	3,5,7
`travel` $= 196$, `sympt` $= 100$, `age` $= 64$	low	3,6,7
`travel` $= 1$, `sympt` $= 1$, `age` $= 64$	medium	2,5,7
`travel` $= k$, `sympt` $= m$, `age` $= 64$	medium	2,5,7
`travel` $= k$, `sympt` $= 1$, `age` $= 65$	high	2,5,8
...	...	...
Test invalid equivalence classes individually:		
`travel` $= 197$, `sympt` $= 1$, `age` $= 65$	invalid	9,5,8
`travel` $= 1$, `sympt` $= 101$, `age` $= 65$	invalid	2,10,8
`travel` $= 196$, `sympt` $= 100$, `age` $= -1$	invalid	3,6,11
...	...	...

[2] In fact, we could already construct our equivalence classes in a way such that these combinations are already reflected, e.g., (`countries` $= 0) \wedge (0 <$ `symptoms` $\leq m) \wedge (0 \leq$ `age` $< 65)$, ...

3 Invariants (4 points)

Consider the following program, where `x` and `y` are non-negative natural numbers (possibly 0):

```
x = y + 1;
while (x != y) {
  x = x + (y % 2);
  y = y + (x % 2);
}
```

Consider the formulas below; tick the correct box (☑) to indicate whether they are loop invariants for the program above.

- If the formula is an inductive invariant for the loop, provide an informal argument that the invariant is inductive.
- If the formula P is an invariant that is *not* inductive, give values of `x` and `y` before and after the loop body such that $P \wedge B$ (where B is `(x != y)`) holds before the execution of

  $$\texttt{x} = \texttt{x} + (\texttt{y}\%2);\ \texttt{y} = \texttt{y} + (\texttt{x}\%2);$$

  and P does not hold anymore afterwards.
- Otherwise, provide values of `x` and `y` that correspond to a reachable state showing that the formula is *not* an invariant.

$(x - y) \leq 1$ ✓ Inductive Invariant ☐ Non-inductive Inv. ☐ Neither

Justification:

Base case: x is 1 larger than y, therefore $x - y$ is 1.

Induction step: We have $x \neq y$, otherwise the loop is never entered and the invariant is trivially unchanged. Moreover, since $(x - y) \leq 1$, x can be at most 1 larger than y. If y is more than 1 larger than x, $(x - y) \leq 1$ still holds after the loop body, since no variable can be increased by more than 1. Therefore, we only need to consider the cases where $x - y = 1$:

1. Assume y is odd, then x must be even. Then the new value of x is $x + 1$ (and therefore odd) and the new value of y is $y + 1$; therefore still $(x - y) \leq 1$.
2. Assume y is even, then x must be odd. Thus x remains unmodified, and the new value of y is $y + 1$, therefore $x = y$

$(x - y) \leq 2$ ☐ Inductive Invariant ✓ Non-inductive Inv. ☐ Neither

Justification:

- Counterexample to induction: $x = 3$, $y = 1$, and after the loop $x = 4$, $y = 1$.
- $(x - y) \leq 2$ is implied by invariant $(x - y) \leq 1$, and hence an invariant.

$(x - y)\%2 = 1$ ☐ Inductive Invariant ☐ Non-inductive Inv. ✓ Neither

Justification:

Violated when $x = y$.

$(x \geq y)$ ✓ Inductive Invariant ☐ Non-inductive Inv. ☐ Neither

Justification:

Base case: x is 1 larger than y, therefore $x \geq y$.

Induction step: We have $x \neq y$ (otherwise the loop is never entered and the invariant is trivially unchanged); together with $x \geq y$ this guarantees that $x > y$ upon loop entry. Also, y is increased by at most 1 in the loop body, hence it cannot be larger than x after the loop body has been executed.