A roadmap

design → implementation → testing → debugging → integration

we are here.
Testing

○ The process of ensuring that a product works as planned (verification).
○ Testing is described via a test plan.
○ Quality of testing is subjective: whether something is sufficiently tested is a matter of comfort with risks.
Kinds of testing

Function: an individual function.
Module: a cohesive set of functions.
Integration: to put modules together
   Bottom-up
   Top-down

Today: function testing.
Testing quandaries

When can something be considered to be "sufficiently tested?"
How does one decide upon the tests to perform?
When does one test?
Prior observations about testing
- Complete testing is **intractable** (not undecidable).
- Testing is a matter of **verifying postconditions for specific sets of preconditions**.
- A test consists of a set of preconditions $P$ and a set of expected postconditions $Q$.
- E.g., a test is a pair $(P,Q)$.
- A **test suite** is a set of tests.
- Question is not whether a test suite is **complete** (a meaningless concept), but whether it is **sufficient**.
A basic principle of testing:
   Testing is **sufficient** when the **cost** of finding further deficiencies or defects is exceeds the **value** of finding them.

Paraphrased from a basic principle of security:
   A solution is **secure** if the **cost** of circumventing its protections exceeds the **value** of circumventing them.

**This is not in your book!** This is an **evolving principle**.
At this point, security testing is evolving into a **dominant** testing activity!
Vocabulary of testing
○ **alpha-testing**: done by developer based upon test plans.
○ A product is "in alpha" if the developer hasn't signed off on it.
○ **beta-testing**: done by independent small population of potential customers.
○ A product is "in beta" if the developer has released it to be used by a small **well-known group of customers**.
○ The next state after "alpha" and "beta" is "**full release**", i.e., made available to the **general public**.
Sufficient testing is a matter of risk

A test suite is **sufficient** if risks are **sufficiently small**. A test suite is insufficient if unacceptable risks remain after all tests pass.

Example: open-source software
- Risk: clients won't download and use it.
- Profit potential: low.
- Sufficient testing: release to public in **alpha state!**

Example: purchased software
- Risk: clients won't buy it.
- Profit potential: high.
- Sufficient testing: wait until **full release** to sell to public.
What is "Sufficient"?
○ "Sufficient" has a **different meaning** than in Math.
  ○ In Math, a **sufficient condition implies a conclusion**.
○ Here, a **sufficient condition** makes a conclusion **plausible**. E.g, "more likely than not!"
  ○ Sufficiency involves **insufficient evidence to refute the null hypothesis** (that the software works)!
  ○ Testing concerns **statistical reasoning**, and not logic!
Kinds of testing

- **Unit testing** checks function of components of a solution.
- **Integration testing** checks how components fit together.
- **White-box testing** uses knowledge of the source code for the tested component.
- **Black-box testing** ignores source code and utilizes requirements and documentation.
Our first concern: unit testing

- Situation: a new component has been written.
- Task: verify function of the component **before** integrating it with others.
- This is **unit testing**.

Steps in unit testing

- **Design a test suite** of tests sufficient to verify function.
- Create a **test scaffolding** (alternative language: "**truss the component**") so that one can run the tests.
- Run all tests within the scaffolding
- Report all failures to the developers.
Several concepts of sufficiency

Boundary conditions (at the edges of the preconditions) all work properly. (black-box)

Several flavors of white box testing:

Every line of code has been executed.
Every branch statement has been taken in both directions.

• A set of representative paths through the code have been tested.
Designing a test suite (for a unit)
  Key idea: what is sufficient?

Several ideas of sufficiency.
  Classify inputs; test every "kind" of input.
    "Black-box" testing via "equivalence partitioning"
  Construct tests that execute every line of code at least once.
    "white-box testing"
  Execute a representative set of paths through code.
    "cyclomatic basis testing"
Equivalence partitioning

Input: the preconditions and postconditions for the unit.

Procedure:
Classify kinds of inputs based upon acceptable preconditions.
Test boundary and non-boundary representatives of each class.

Rationale:
Most errors are made with boundary conditions. E.g., assuming arrays start at 1 rather than 0. Thus, one can find most errors by looking at some non-boundary input and all boundary inputs.
A **partition** of a set is a set of mutually exclusive subsets. Example: one partition of \{1,2,3,4\} is \{\{1\},\{2,4\},\{3\}\}  

Idea of equivalence partitioning:
- partition the inputs to a program,
- try one case from each partition.
What makes equivalence partitioning practical: can partition in several dimensions

E.g., a partition of one input can be combined with a partition of another to create a partition for both.

Example:

\[ x < 0, \ 0 \leq x \leq 9, \ x > 9 \] partitions values for \( x \)

\[ y < 1, \ 1 \leq y \leq 4, \ y > 4 \] partitions values for \( y \)

Thus a partitioning for \((x,y)\) is

\[ \{x < 0, \ 0 \leq x \leq 9, \ x > 9\} \times \{y < 1, \ 1 \leq y \leq 4, \ y > 4\} \]

\[ = \{(x < 0, y < 1), \ (x < 0, 1 \leq y \leq 4), \ (x < 0, y > 4), \]

\[ (0 \leq x \leq 9, y < 1), \ (0 \leq x \leq 9, 1 \leq y \leq 4), \ (0 \leq x \leq 9, y > 4), \]

\[ (x > 9, y < 1), \ (x > 9, 1 \leq y \leq 4), \ (x > 9, y > 4)\} \]

Where we notate \( \{y = 1, y = 2, y = 3, y = 4\} \) as "1\( \leq y \leq 4\)"
Cartesian product of partitions of two parameters is a partition of the ordered pairs:
Example of equivalence partitioning:
void sort(int a[], int size);

Preconditions:
   a is an array of between 0 and 100 numbers.
   size is the number of elements contained in a.

Postconditions:
   a contains the same numbers, but sorted into increasing order.
The equivalence classes of inputs:

First attribute: size of array
   Middle condition: size=20
   Boundary conditions: size=0, 1, 99, 100
   Partition \{s=0, s=1, 2<=s<=98, s=99, s=100\}

Second attribute: whether items are same or different.
   Middle condition: 1/2 of the items are the same as others.
   Boundary conditions: all the same, all different.
   Partition=\{all the same, all different, some same\}

Third attribute: pre-existing ordering
   Middle condition: random order.
   Boundary conditions: already sorted, already sorted in reverse order.
   Partition=\{sorted, reverse, random\}
Using Cartesian products

- Often the easiest way to express equivalence classes is as a Cartesian product of sets of conditions:
- Let $X$ be the Cartesian product of two sets
  (e.g., $\{a,b\} \times \{1,2\} = \{(a,1),(a,2),(b,1),(b,2)\}$
- The equivalence classes for sort can be expressed as a product space:
  $\{\text{size}=0,1,20,99,100\} \times \{\text{all same, all different, 1/2 same}\} \times \{\text{sorted, sorted reverse, random order}\}$
  Where some of these classes have no members.

E.g., it is not possible to construct a reasonable instance of
(size=1, all the same, random order)!

The instances for a triple are the intersection of the
instances for each element, which can be empty!
Categories for tests to be done involve all three attributes
(but some combinations are nonsense)

size=0, all the same, already sorted
size=1, all the same, already sorted
size=20, all the same, already sorted
size=20, 1/2 same, already sorted
size=20, 1/2 same, already sorted in reverse
size=20, 1/2 same, random order
size=20, all different, already sorted
size=20, all different, already sorted in reverse
size=20, all different, random order
size=99, all the same, already sorted
size=99, 1/2 same, already sorted
size=99, 1/2 same, already sorted in reverse
size=99, 1/2 same, random order
size=99, all different, already sorted
size=99, all different, already sorted in reverse
size=99, all different, random order
size=100, all same, already sorted
size=100, 1/2 same, already sorted
size=100, 1/2 same, already sorted in reverse
size=100, 1/2 same, random order
size=100, all different, already sorted
size=100, all different, already sorted in reverse
size=100, all different, random order
Actual tests are representatives of categories
size=0, all the same, already sorted
   ([], [])
size=1, all the same, already sorted
   ([5],[5])
size=20, all the same, already sorted
   ([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
   [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
size=20, 1/2 same, already sorted
   ([1,2,3,4,5,5,5,6,6,7,7,7,7,8,8,9,9,9,9,10],
   [1,2,3,4,5,5,5,6,6,7,7,7,7,8,8,9,9,9,9,10])
size=20, 1/2 same, already sorted in reverse
   ([10,9,9,9,9,8,8,7,7,7,6,6,5,5,5,4,3,2,1],
   [1,2,3,4,5,5,5,6,6,7,7,7,7,8,8,9,9,9,9,10])
etc.
Is this sufficient?

For every test plan, there is a version of the function that passes it and then fails on other tests.
No, it isn't sufficient; for every plan, there is a counter-example.