Software Testing

Test the following code

- Given three integers representing the sides of a triangle, the problem should return "*scalene*", "*isosceles*", or "*equilateral*"
  - Scalene: No two sides equal
  - Isosceles: Only two sides equal
  - Equilateral: All three sides equal
- What are the test cases?

Checklist

- Did you:
  - Have a test case for each possible correct input?
    - For isosceles, three permutations? (2, 2, 3), (2, 3, 2), (3, 2, 2)
  - Test for negative inputs? (-2, 4, 4)
  - Test for non-integer values? (3.5, 3.5, 4)
  - Test if one or more sides is zero? (0, 0, 0)
  - Test for three inputs that don't satisfy the triangle inequality? (1, 2, 3)
  - Test for non-integer inputs?
  - Test for wrong number of inputs? (2, 3)
  - Test for no input?
  - Specify the correct output for each case?

Testing mindset

- What is software testing?
- **Testing is the process of quality assurance through error finding**
  - It usually involves executing the program
- Testing should be seen as *constructive*
- The programmer should not test their own code

Software errors

- Incorrect output
- Incorrect error handling
- Memory leak, resource hogging
- Crash, locking
- Security errors: buffer overflow, use-after-free, parsing, etc.
- …

Causes of software errors

- Typos
- Control flow error
- Missed cases
- Misunderstood requirements
- Incorrect assumptions
- API usage
- Code changing
- Memory referencing errors
- …

Testing techniques

- Human testing techniques (code review)
  - Code inspection
  - Walkthrough
- Software testing techniques
  - Test case design
    - Black-box, white-box
  - Unit testing
  - Integration testing
  - Usability testing

Code Inspection

- Manually inspecting code as a team
- Process is slow: usually no more than 200 statements per hour
- Team members:
  - Original programmer: explains the code
  - Moderator: senior coder that leads and organizes the inspection
  - Tester: specialist that is familiar with testing code
  - Possibly other programmers
- Use a checklist

Code Inspection checklist (example)

- **Data reference (e.g. arrays)**: Are all referenced variables set? Are any references out of bounds? User controlled references? Off-by-one errors?
- **Initialization**: Are variables declared? If not, are the defaults correct? Are variable declarations consistent with variable type?
- **Comparison**: Any confusion between greater/greater or equal to? Are Boolean expressions used correctly?
- **Control flows**: Do loops terminate? Are looping conditions changed during loop?

Walkthrough

- First, a tester prepares a list of test cases
- Team examines the code by going through these test cases manually, discussing whether or not the code performs well in these test cases
- Compared to automated testing:
  - Humans can give qualitative answers instead of quantitative ones
  - Reviews can discuss efficiency and improve readability
  - Can address edge cases and discuss correct response to unexpected inputs

White-box testing

- Also known as structural testing
- Derive test cases from examining the code and the requirements
- Advantage: Using knowledge of the implementation, we can derive thorough test suites that <u>cover</u> all cases
- Disadvantage: Since tests are based on specific implementation, test quality drops if implementation changes
- For now we focus on unit testing

White-box testing

- When can we say we have tested a piece of code **completely?**
- Statement coverage: Every line of code is run at least once

```
boolean is_prime(int input) {
    if (input == 1) {return false;}
    if (input >= 2) {return true;}
}
```

- Two test cases: 1 and 2 will cover all statements
- Clearly, statement coverage is not sufficient

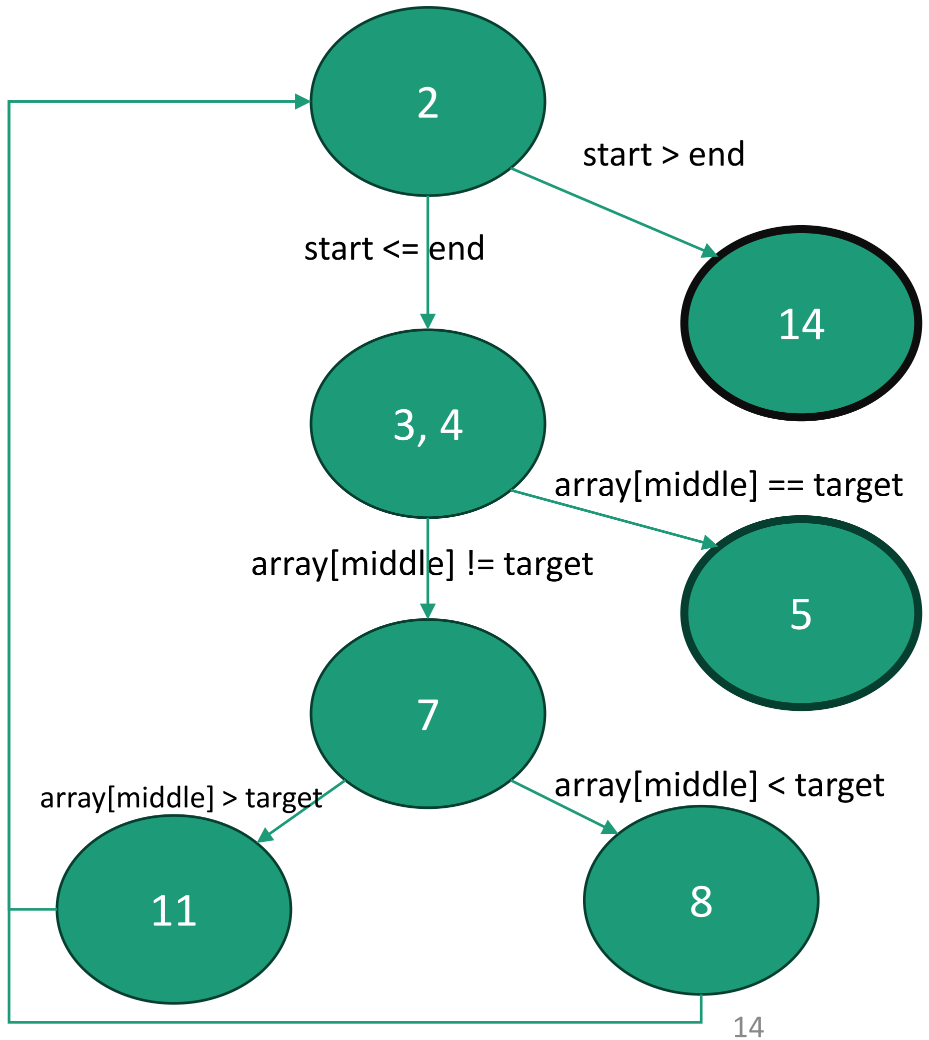
Control Flow Graph

- Control-flow graph: shows the program logic around control flow statements (while, for, if…)
- Draw a control-flow graph of the following function:

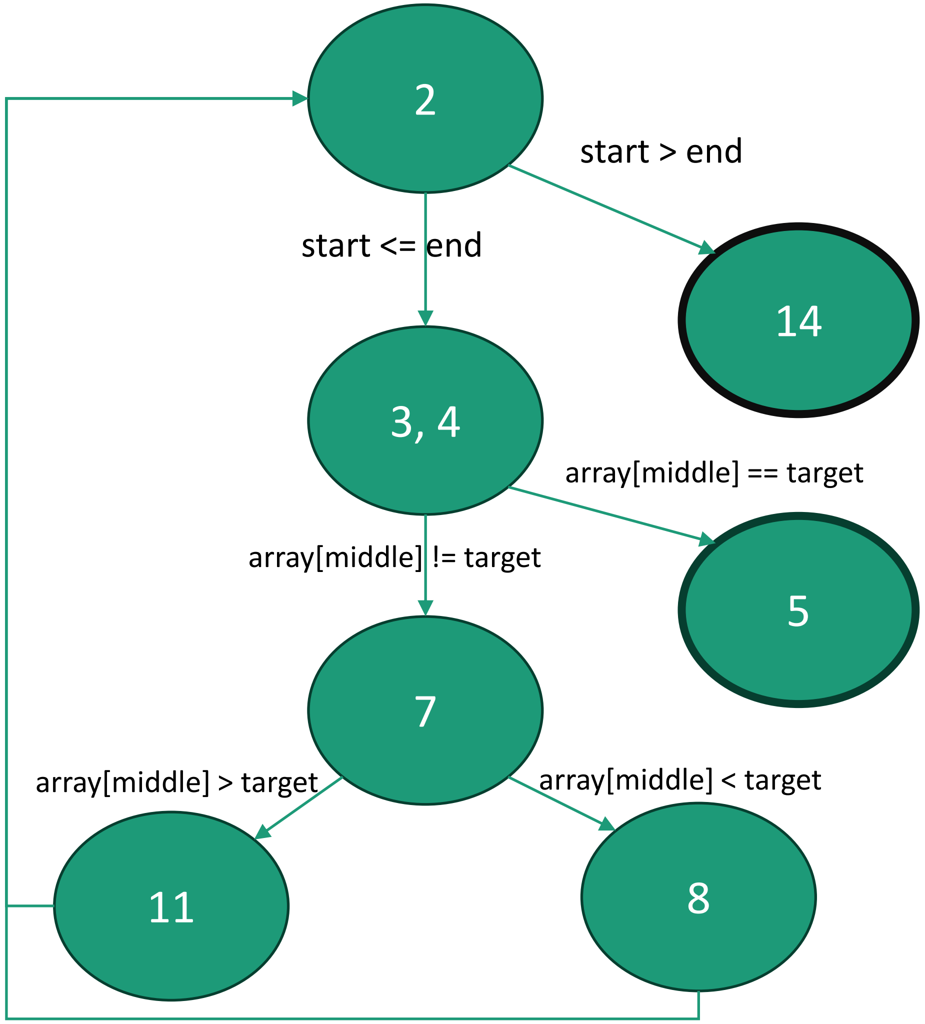
```
int binary_search(int array[], int target, int start, int end) {
    while (start <= end) {
        int middle = (start + end) / 2;
        if (array[middle] == target) {
            return middle;
        }
        else if (array[middle] < target) {
            start = middle + 1;
        }
        else {
            end = middle - 1;
        }
    }
    return -1;
}
```

Control Flow Graph

```
int binary_search(int array[], int target, int start, int end) {
    while (start <= end) {
        int middle = (start + end) / 2;
        if (array[middle] == target) {
            return middle;
        }
        else if (array[middle] < target) {
            start = middle + 1;
        }
        else {
            end = middle - 1;
        }
    }
    return -1;
}
```

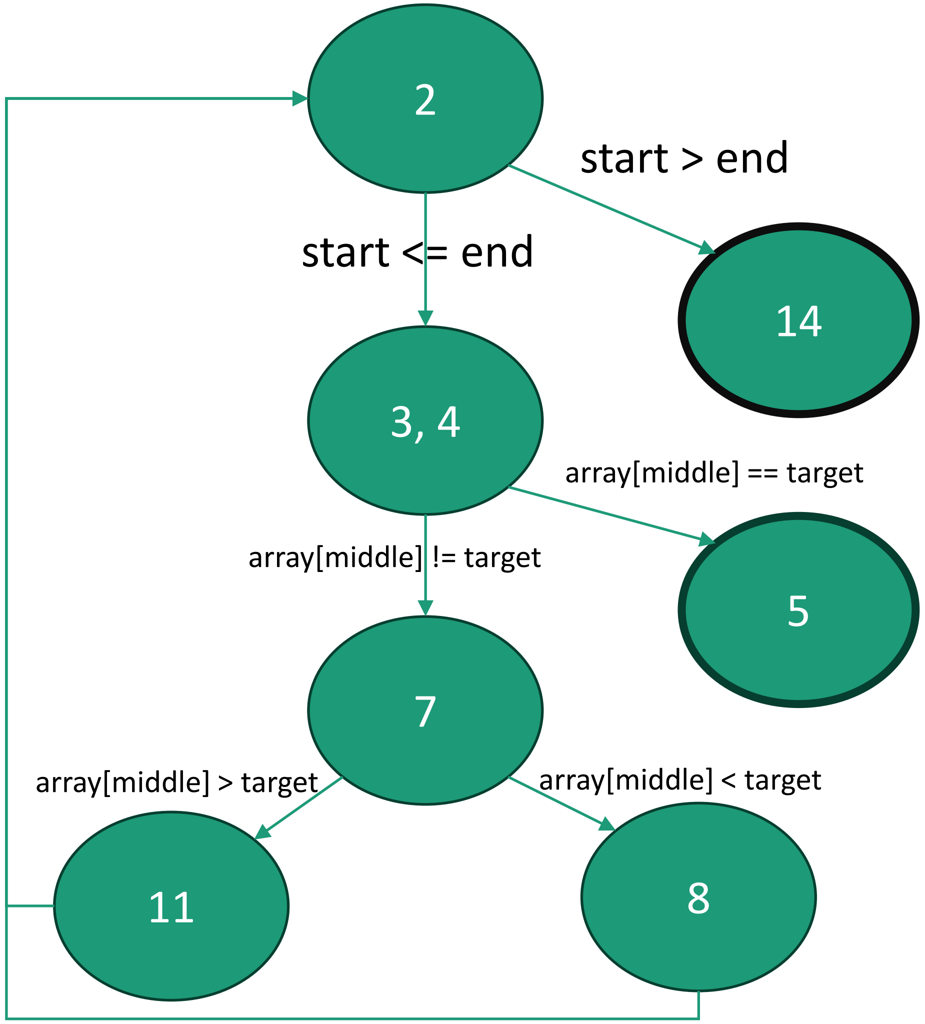


Branch coverage



- Node coverage: All nodes are executed at least once
- Branch coverage: All branches are traversed at least once
- Find 2 test cases that will cover all branches and nodes
  - [1, 3, 4, 5, 6], find 2
  - [1, 3, 4, 5, 6], find 5

Path coverage



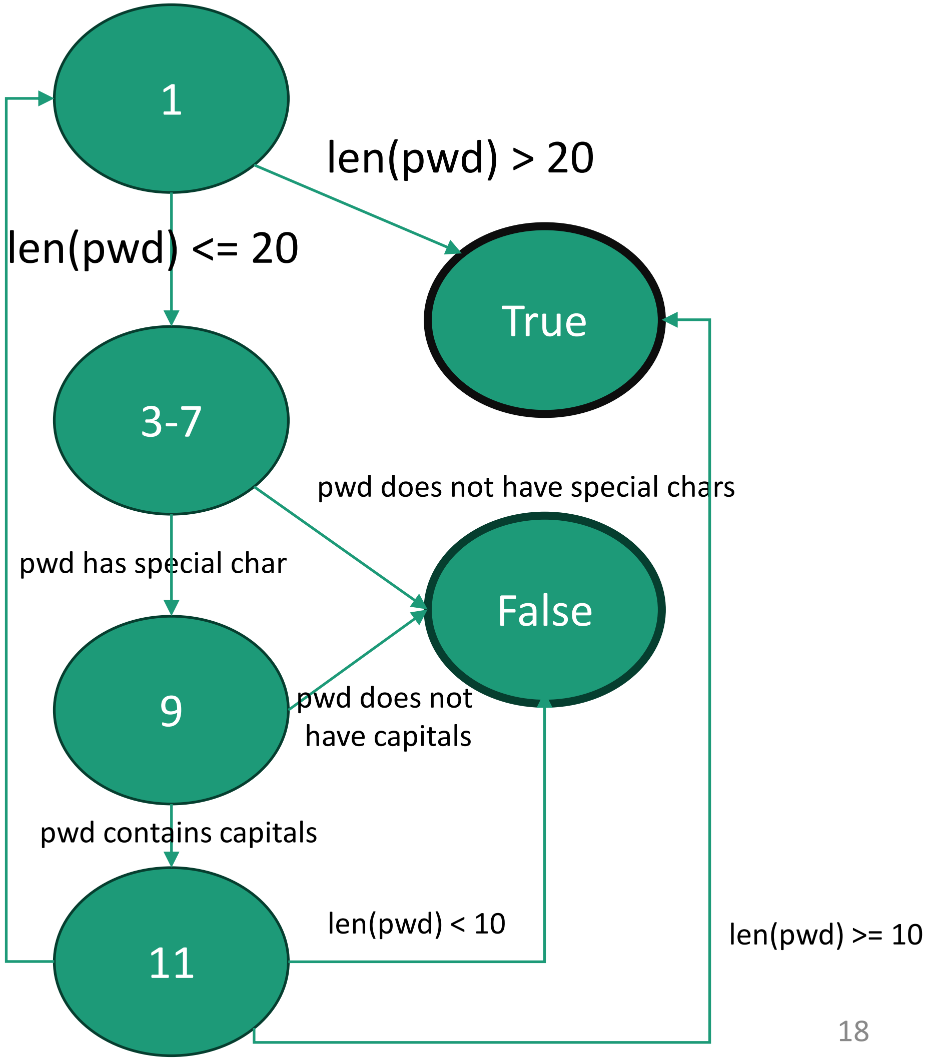
- A path is a list of nodes traversed by a test case
- [1, 3, 4, 5, 6], find 5:
  - (2), (3, 4), (7), (8), (2), (3, 4), (5)
- Path coverage can help determine test set quality
  - Attempt to cover all paths (within a limit)
  - Find and remove repetitive test cases

Path coverage

- A *simple path* is a path with no node repetitions, except the start and end can be the same
- A *prime path* is a simple path that cannot be lengthened any further
  - This implies no prime path is a substring of another prime path
- Prime path coverage: What percentage of prime paths have been tested?

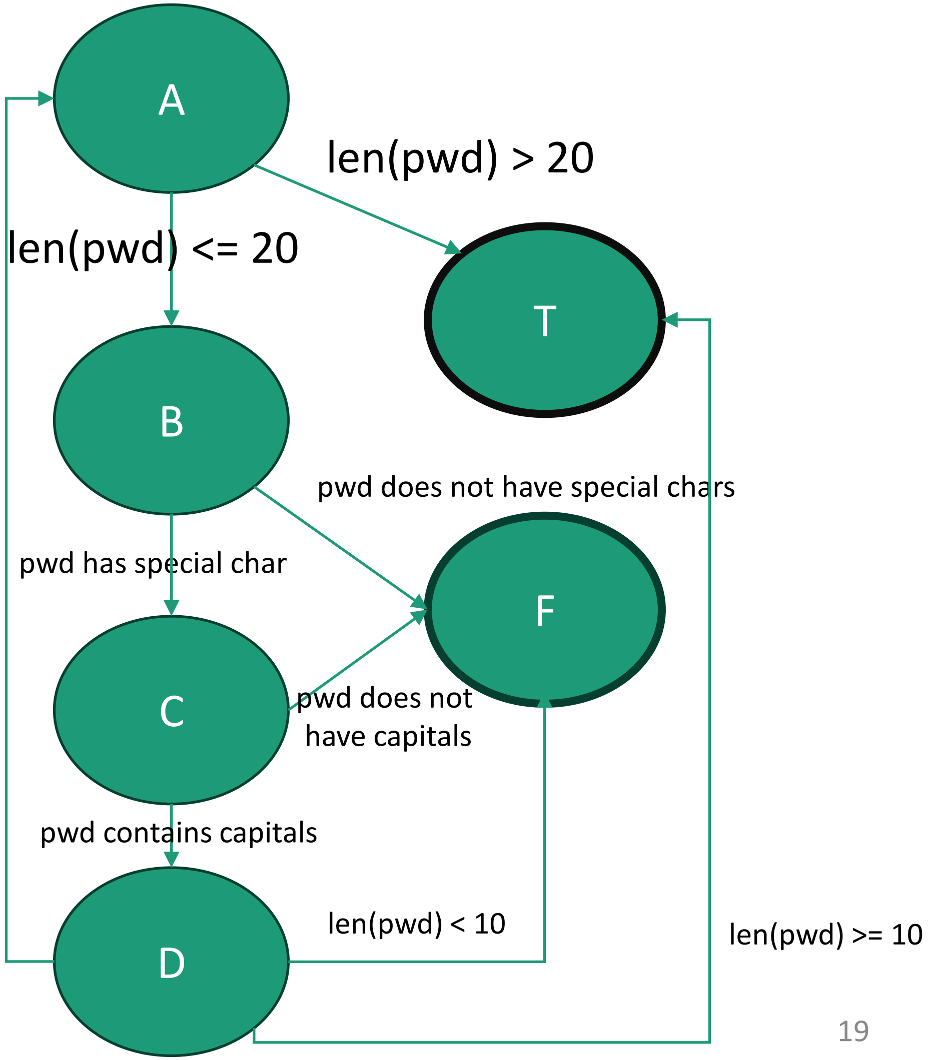
Path coverage

```
if (len(password) > 20):
    return True
has_special_chars = False
for (x in password):
    if (x in special_chars):
        has_special_chars = True
if (!has_special_chars):
    return False
if (password.lower() == password):
    return False
if (len(password) < 10):
    return False
return True
```



Path coverage

- Prime paths are:
  - A, T
  - A, B, F
  - A, B, C, F
  - A, B, C, D, F
  - A, B, C, D, T
- Test cases should cover these five paths



Logic coverage: MC/DC

- *Modified condition/decision coverage*
  - Used by e.g. NASA for critical software
- Decision coverage: Final decision needs to be T/F at least once
- Condition coverage: Each condition in a decision needs to take on all possible values at least once

```
if (total < 50 or final < 50) {
    return False;
}
return True;
```

- Decision coverage:
  (total = 40, final = 60), (total = 60, final = 60)
- Condition coverage:
  (total = 60, final = 40), (total = 40, final = 60)

MC/DC

- MC/DC requires both decision and condition coverage, and:

Every condition in a decision has been shown to independently affect that decision's outcome.

- For example, if the relevant conditions are A, B, and C, then:
  1. For A:
     - There needs to be two cases, A is True and A is False, where the outcome is different
     - The values of B and C for those two cases needs to be the same
  2. Repeat (1) and find two cases for B and C as well

Black-box testing

- Test cases are built only on specifications
- Without knowledge of program logic, it is harder to build complete test cases
- Test cases are more likely to be useful if code changes
- Special case: Pentesting

Equivalence partitioning

- Derive “invalid” and “valid” ranges for each input value
  - e.g. Age 18-65: Equivalences classes are <18, [18, 65], >65
  - e.g. Score 50+: Equivalence classes are <50, >= 50
  - e.g. Triangle testing code, three inputs: Equivalence classes are “two or fewer inputs”, “three inputs”, “more than three inputs”
- If program handles possible values of inputs differently, treat them as different equivalence classes
  - e.g. Grade displaying software: User is “student”, “teacher”, “admin” – three equivalence classes
- Finally: there should be one test case for each equivalence class
  - A test case can cover multiple **valid** equivalence classes, but only one **invalid** equivalence class

Equivalence partitioning

- Example: Password code equivalence classes
  - Length: < 10 is invalid, 10-20 depends, >20 is valid
  - Special chars: 0 is invalid, 1+ is valid
  - Capital letters: 0 is invalid, 1+ is valid
- Valid test cases:
  - Length > 20
  - Length 10-20, special char, capital letter
- Invalid test cases:
  - Length < 10, has special char, capital
  - Length 10-20, no special char, has capital
  - Length 10-20, has special char, no capital

```
if (len(password) > 20):
    return True
has_special_chars = False
for (x in password):
    if (x in special_chars):
        has_special_chars = True
if (!has_special_chars):
    return False
if (password.lower() == password):
    return False
if (len(password) < 10):
    return False
return True
```

Boundary-value testing

- Experience tells us that values on the boundary are more likely to be wrong
- Derive boundary values from equivalence classes
- Example: Code that performs safe addition of integers
  - If a+b > INT_MAX or a+b < INT_MIN, we have a buffer overflow
  - Equivalence classes: a+b < INT_MIN, INT_MIN <= a+b <= INT_MAX, a+b > INT_MAX
  - Boundary values: a+b = INT_MIN, a+b = INT_MIN – 1, a+b = INT_MAX, a+b = INT_MAX + 1
  - We can also set a = INT_MIN and b = 0 individually, etc.

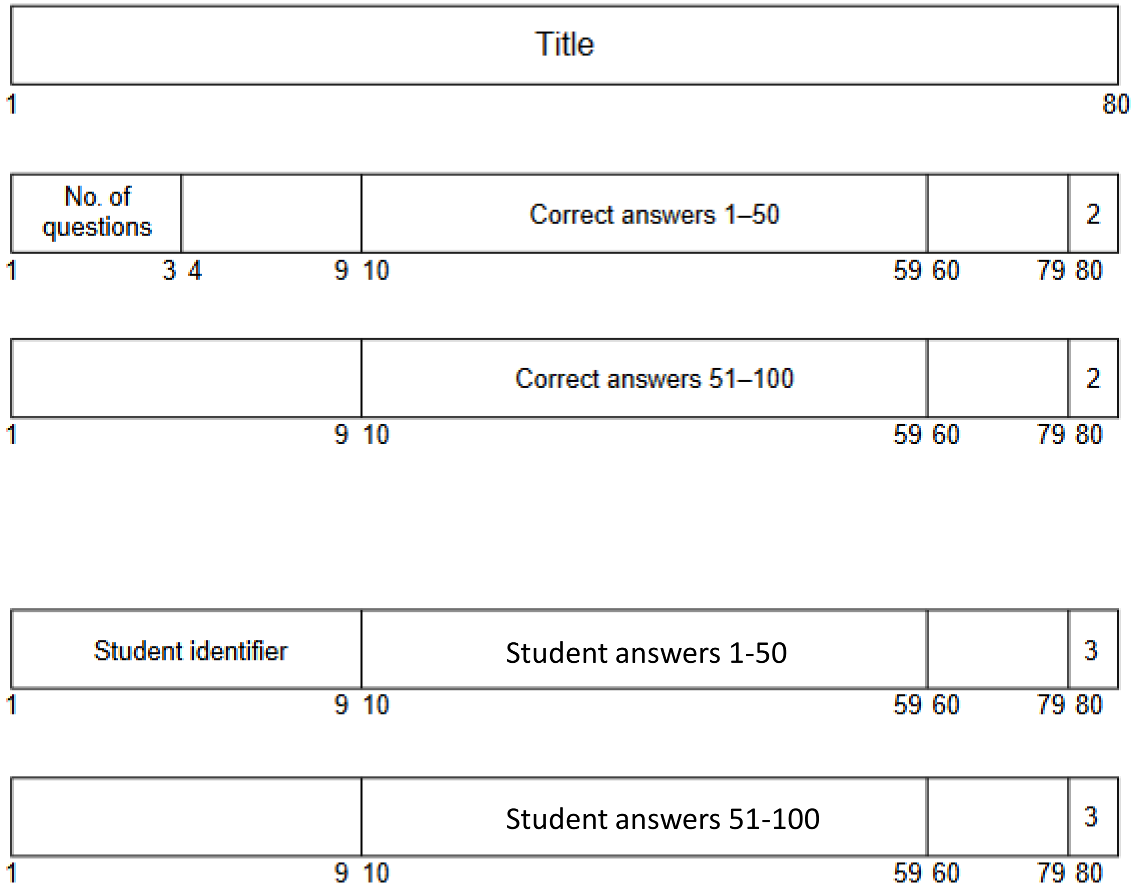
Boundary-value testing example

(Myers, Sandler and Badgett)

- A program grades multiple-choice question solutions
- Each line is <u>80 characters long</u>
- Three parts:
  - First line: Always a title
  - Second part: Correct answers. They are marked with a “2” in the 80^{th} character
  - Third part: Student answers. They are marked with a “3” in the 80^{th} character
- Each line after first contains 50 correct answers (10^{th} to 59^{th} characters) or 50 student answers (at most 999 questions)
- First line contains number of questions in chars 1 to 3
- Each student line starts with a 9-character identifier, up to 200 students
- Output: Students and their grades and ranks, sorted by identifier

Boundary-value testing example

(Myers, Sandler and Badgett)



What are the test cases?

Header/correct answers tests:

1. Empty file
2. Missing title
3. 1-character title
4. 80-character title
5. 0-question exam
6. 1-question exam
7. 50-question exam
8. 51-question exam
9. 999-question exam
10. Number of questions is not a number
11. Number of questions is correct
12. No correct answers
13. Number of correct answers = number of questions + 1
14. Number of correct answers = number of questions - 1

Student answers tests:

15. No students
16. 1 student
17. 200 students
18. 201 students
19. Student answered 1 question but there are 2 correct answers
20. Student answered 2 questions but there is 1 correct answer
21. No student identifier
22. Non-number student identifier
23. Valid student identifier

Report tests:

24. All students have same grade
25. All students have different grade
26. Some students have same grade
27. Student has grade of 0
28. Student has maximum grade
29. Check sort: student has lowest identifier
30. Check sort: student has highest identifier

Boundary-value testing example

- Program that takes (day, month, year) and returns the next date
  - Year from 1 to 3000
- What are the equivalence classes?
  - Month: February, 30 day Months, 31 day Months
  - Day: 1-28, 29, 30, 31
  - Year: 4-year leap years, 100-year non-leap years, 1000-year leap years, other non-leap years
- Choose tests for each of those cases

Boundary-value testing example

- Testing each type of month (year 2023):
  - 1/0, 1/1, 1/31, 1/32, 2/1, 2/28, 2/29, 4/30, 4/31, 12/31
- Testing each type of day (year 2023):
  - 3/15, 3/29, 3/30, 3/31
- Testing each type of year:
  - 2/28/2024, 2/28/2000, 2/28/2100
  - 2/29/2024, 2/29/2000, 2/29/2100
- Overall boundaries:
  - 1/0/1, 1/1/1, 12/31/3000, 12/31/3000, 1/1/3001

Cause-Effect Graphing

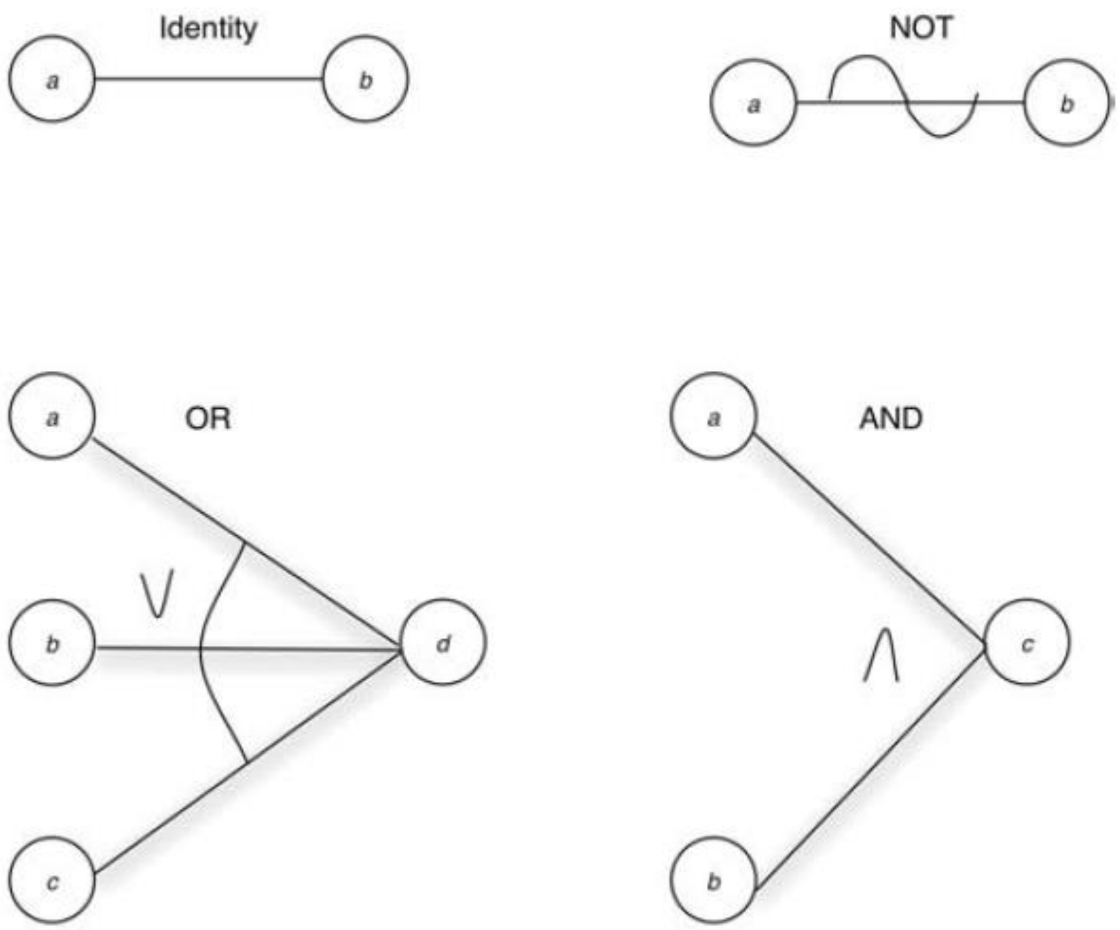
- Equivalence classes/Boundary-value analysis cannot explain how inputs relate to each other
  - We saw a version of this in the password example
  - e.g. if number of questions * number of students > 4,000, OOM error
- First identify all causes and effects in the specification
- Then draw a cause-effect graph for the program
- Cause-effect graph helps us derive test cases

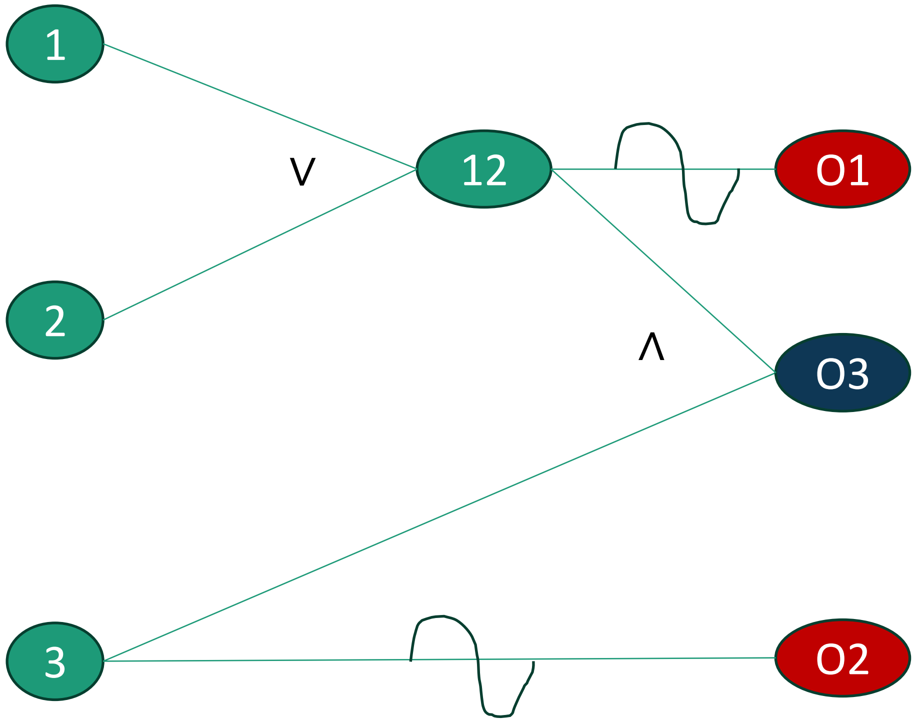
Cause-Effect Graphing

- Example Specification:

> To load a save file, we first check if it is valid. The first character must be 1 “A” (Autosave) or 2 “M” (Manual save), and the 3 second character must be a digit (save number). If the first character is wrong, O1 output “Save error”. If the second character is wrong, output O2 “Save number error”. If both are correct, O3 load the save file.

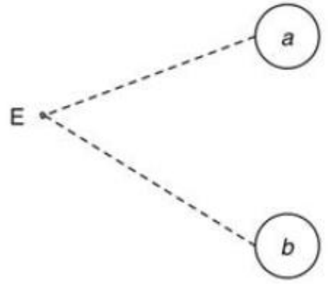
Cause-Effect Graphing



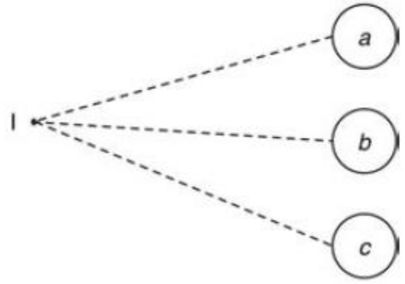


Cause-Effect Graphing

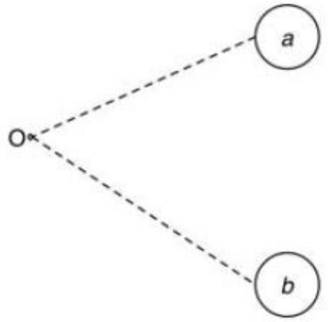
Exclusive: a and b are never both true



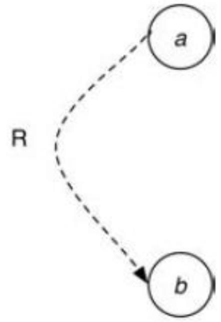
Inclusive: a, b, c are never all false

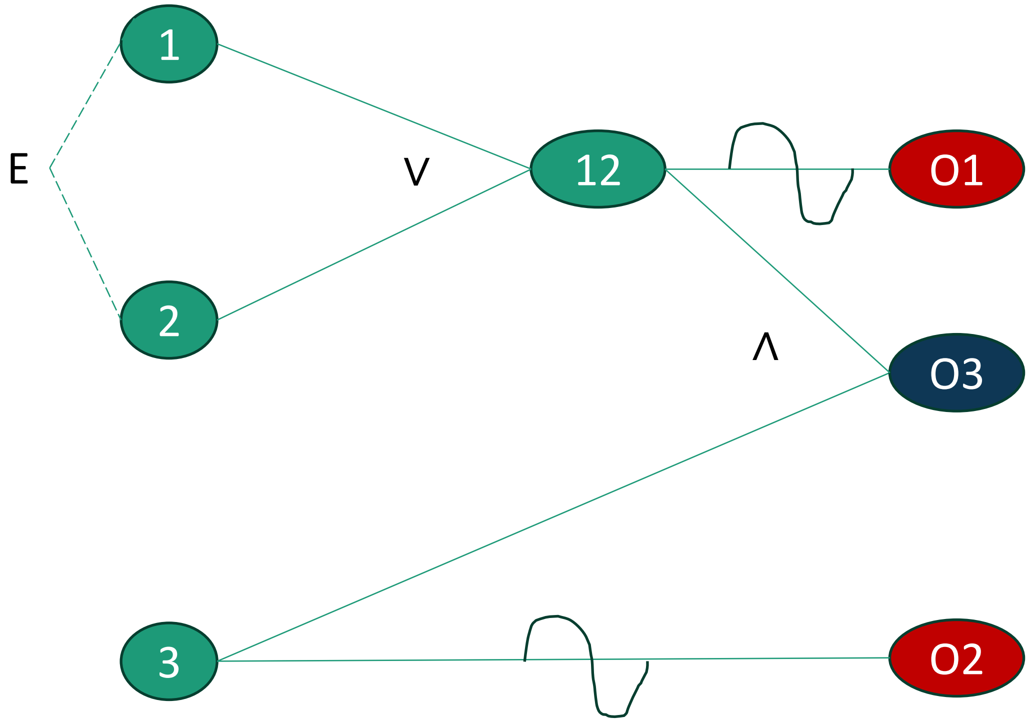


One and only one of a and b are true



a requires b: if a is true, b must be true





Cause-Effect Graphing

- Derive the test cases from the cause-effect graph
- Procedure:

1. Choose an effect and set it to T.
2. Backtrace through the graph finding all combinations that cause the chosen effect to be T.
   - Apply reduction strategies (next slide) to eliminate redundant combinations
3. Repeat step 1 until all effects are covered.

Cause-Effect Graphing

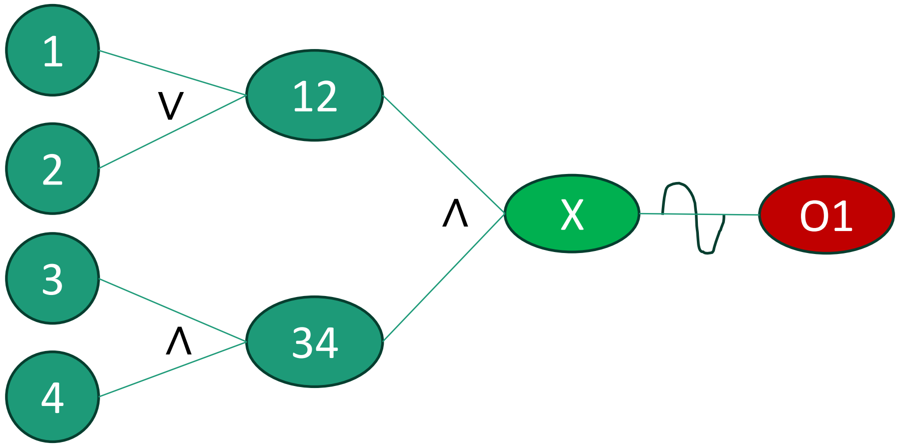
- Reduction strategies for back-tracing:

1. While tracing back an *OR* node where the output is T, only set one output to 1 (e.g. FFT, FTF, TFF)
2. While tracing back an *AND* node where the output is F, consider all cases (e.g. FF, FT, TF)
   a) Terminate backtracing (find only 1 case) for any T inputs. Continue backtracing through F’s.
   b) Terminate backtracing (find only 1 case) for all inputs if all inputs are F.

Cause-Effect Graphing

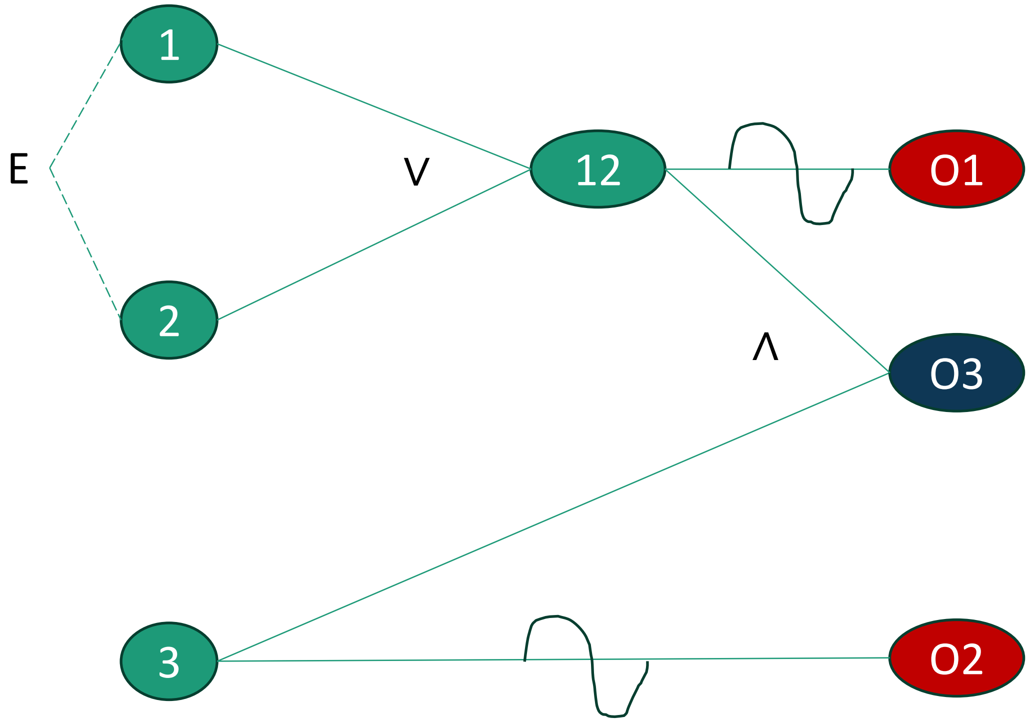
- Example of reduction strategies

1. Set O1 to T, so X = F
2. Three cases for X = F:
    1. 12 = F, 34 = F. Rule 2b: Stop backtracing; find only one case. (1 = F, 2 = F, 3 = T, 4 = F)
    2. 12 = F, 34 = T. Rule 2a: Backtrace through 12. Since it is OR, there is only one case anyway. (1 = F, 2 = F, 3 = T, 4 = T)
    3. 12 = T, 34 = F. Rule 2a: Backtrace through 34. Since it is AND, there are 3 cases.
        - (1 = T, 2 = F, 3 = T, 4 = F), (1 = T, 2 = F, 3 = F, 4 = T), (1 = T, 2 = F, 3 = F, 4 = F)



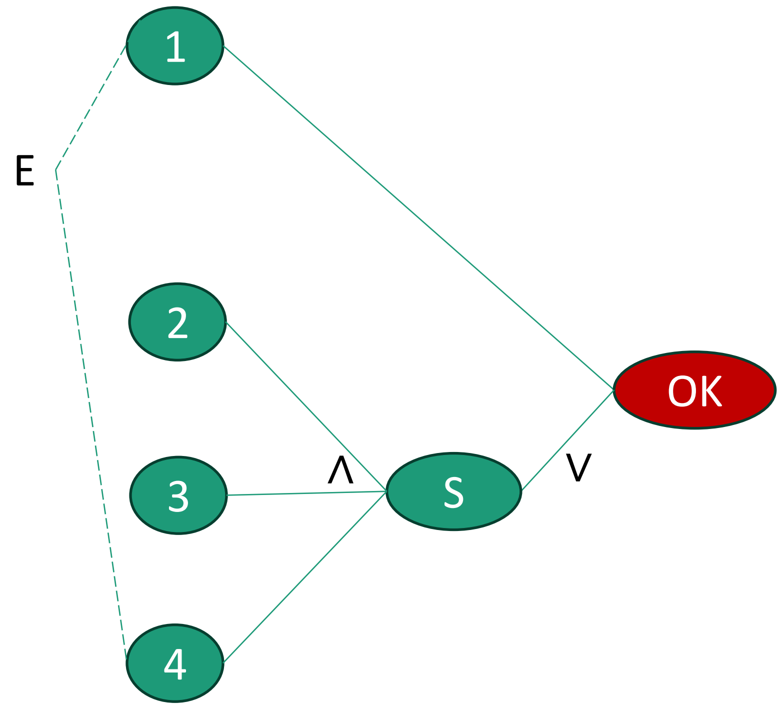
Cause-Effect Graphing

- Save file code, set O3 = T:
- Only one case for O3 = T
  - (12 = T, 3 = T)
- Backtrace through 12 = T:
  - (1 = T, 2 = F, 3 = T)
  - (1 = F, 2 = T, 3 = T)
  - ~~(1 = T, 2 = T, 3 = T)~~



Cause-Effect Graphing

- Password code:
    1. Length > 20
    2. Has special char
    3. Has capital
    4. 10 <= Length <= 20
- Set OK = T
    - 1 = T, S = F
    - 1 = F, S = T -> 2=3=4 = T
    - ~~1 = T, S = T~~
- For 1 = T, S = F
    - Cases for 234: TTF, ~~TFT, FTT~~, TFF, FTF, ~~FFT~~



Cause-Effect Graphing

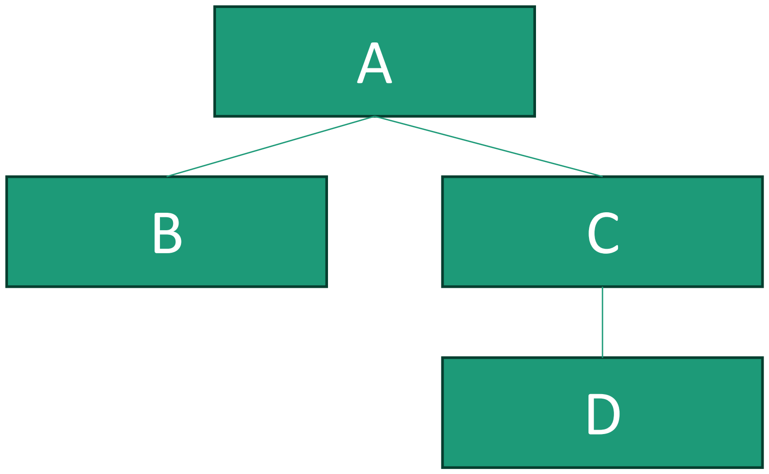
- Effective way to produce logical (and algorithmic) set of test cases without explosion
- Should be combined with boundary value analysis for better test coverage
- Sometimes, the best way is “error guessing”:
  - Identify common errors and generate test cases
  - There is no systematic way to do so: it is based on tester experience

Unit testing (module testing)

- Unit tests focus on a single class
- They should not connect to external databases or services
- To test a class that relies on another class, set the other class as:
  - Mock class: A fake class to examine its values to determine if the test worked. Similar to crash-test dummy.
  - Stub class: A fake class whose properties are fixed by the tester to control the input.
- Example: Test the attack function of the player character
  - Mock class: Check if attack has reduced HP of mock Enemy
  - Stub class: Check if attack hits the enemy if it is close enough and does not hit the enemy if it is far enough

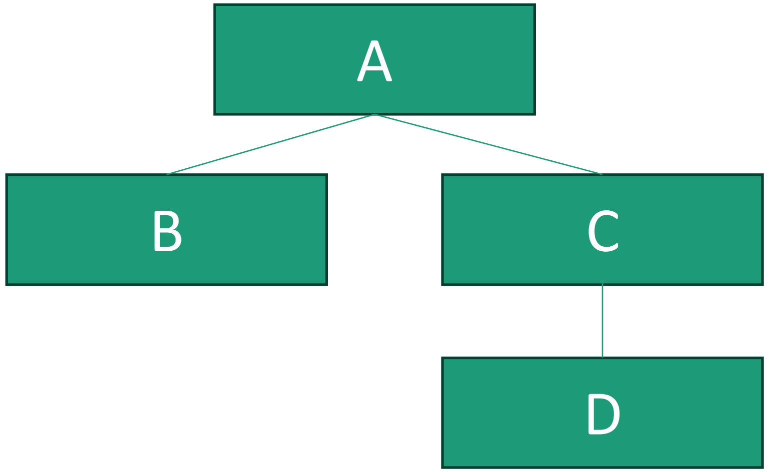
Integration testing

- How do we test several modules that rely on each other?
- Nonincremental approach: test A, B, C, D separately, using mocks to replace classes; then test them all together
- Incremental approach: Test B and D; then test C-D; then test A-B-C-D
  - Mocks are not needed
- Helps to test tightly coupled classes



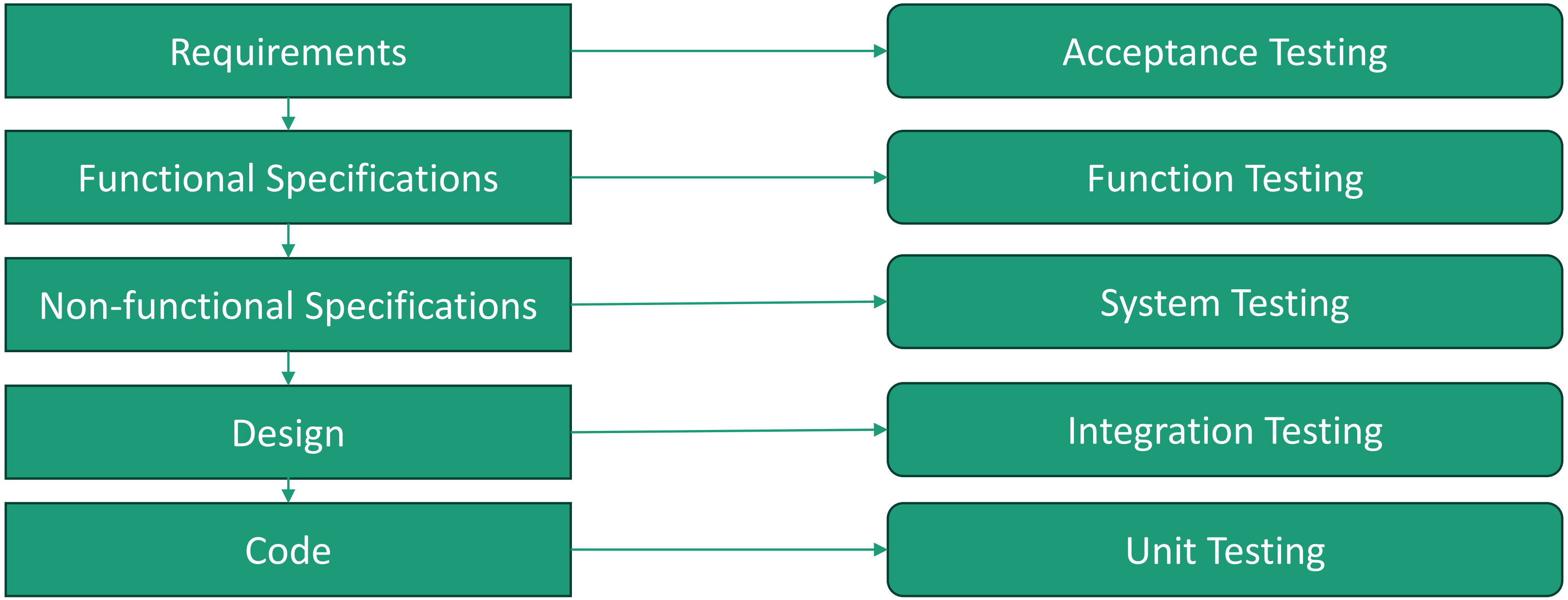
Incremental testing

- Example grading code:
  A. Main code (calls B and C)
  B. Reads and parses input answers and correct answers
  C. Publishes grades and statistics to students
  D. Generates statistics based on grades



1. Test B and D individually
   - D: Input grades, test statistics
2. Test C-D: Test if the grades are published correctly, test if the statistics published by C match those returned by D
3. Test the whole program

Higher-order testing



System testing

- Find issues with the whole system from various non-functional perspectives
- Not based on the functional specifications
- Security, performance, storage, installation, reliability, etc.
- Aim to find issues, not to prove correctness
  - e.g. Stress test a packet filter: raise packet rates until it fails
  - e.g. Security testing: think like an attacker