Flowgraphs and Path Testing

Presented By
Trina Saha
Lecturer
City University

Goals of Testing

- First goal is *bug prevention*
  - No code to correct
  - No retesting needed to confirm correction
  - No one is embarrassed, no wreck of schedule
- The second goal is *bug discovery*

Bug Prevention

- *Designing tests* is the best method
- Test-design thinking can discover and eliminate bugs at every stage in software development cycle
- Thus Dave Gelperin and Bill Hetzel advocate "Test, then code"

Bug Discovery

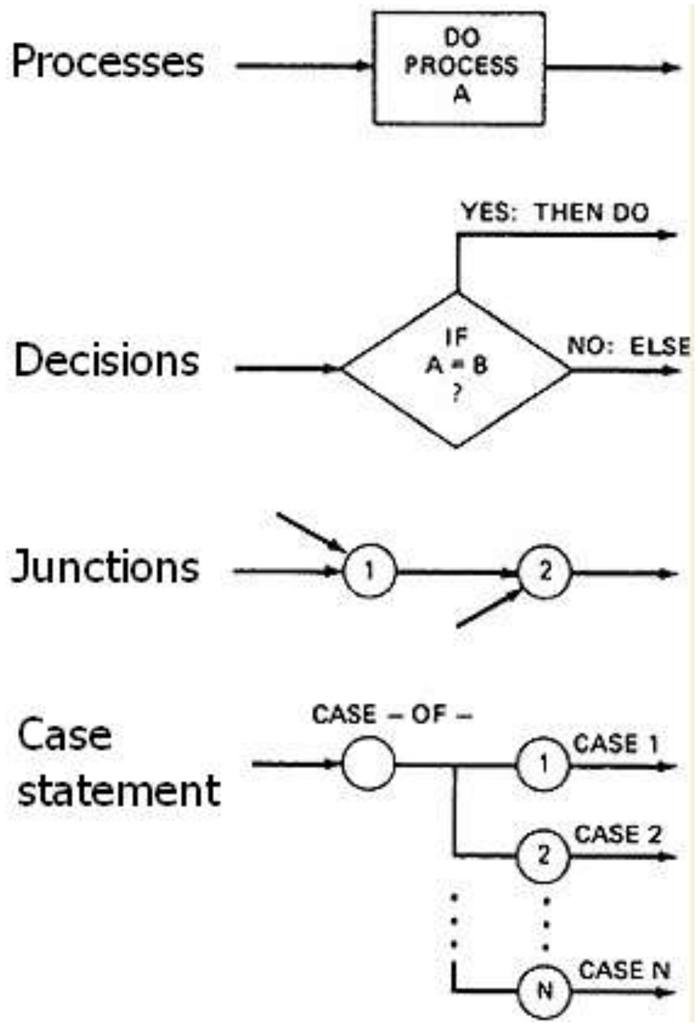
- Bugs are not always obvious
- A test design documents expectations, test procedure, and the results of actual test
- Knowing that a program is incorrect does not imply knowing a bug
- Different bugs can have same manifestations, and
- One bug can have many symptoms

What is Path-testing?

- Path testing
  - A name given to a family of test techniques that judiciously select a set of paths through a program.
- Goal !!
  - achieve some measure of test thoroughness.
- Example
  - Pick enough paths to assure that every source statement has been executed at least once.

Control Flowgraphs

- Control flowgraph / flowgraph is a graphical representation of a programs' control structure
- It uses
  - Process blocks
  - Decisions, and
  - Junctions
- It is similar to earlier flowchart but has some difference



Control Flowgraphs vs. Flowcharts

Control Flowgraph	Flowchart
Do not show the details of what is in a process block	Show the details. May have 100 boxes for 100 steps. There may be higher level of grouping concept, but no fixed rule
Ignores process steps	Flowchart focuses on process steps

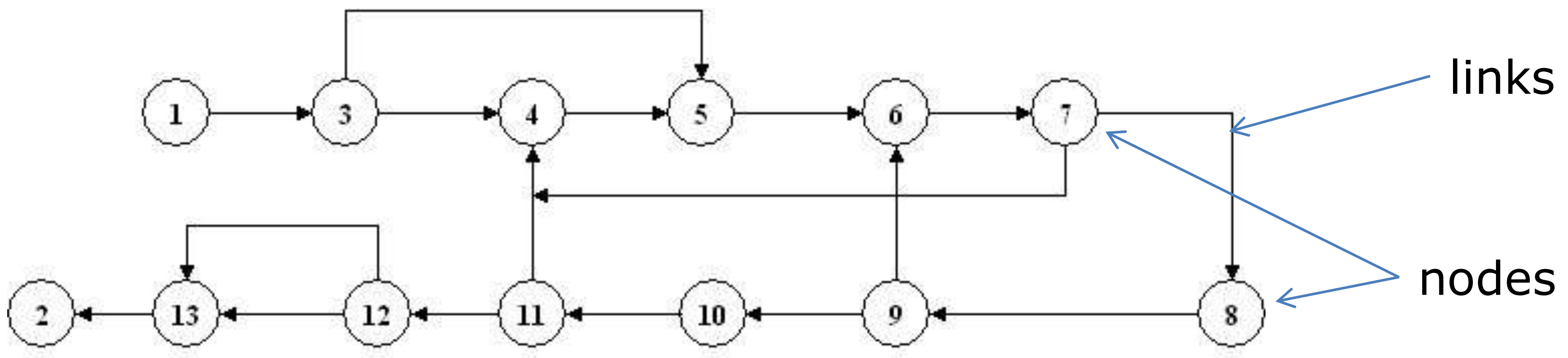


Figure 3.5. Simplified flowgraph notation

Link (7, 4)
Link (12, 13, upper) and (12, 13, lower)
Or simply use a unique lowercase letter for each link in the flowgraph

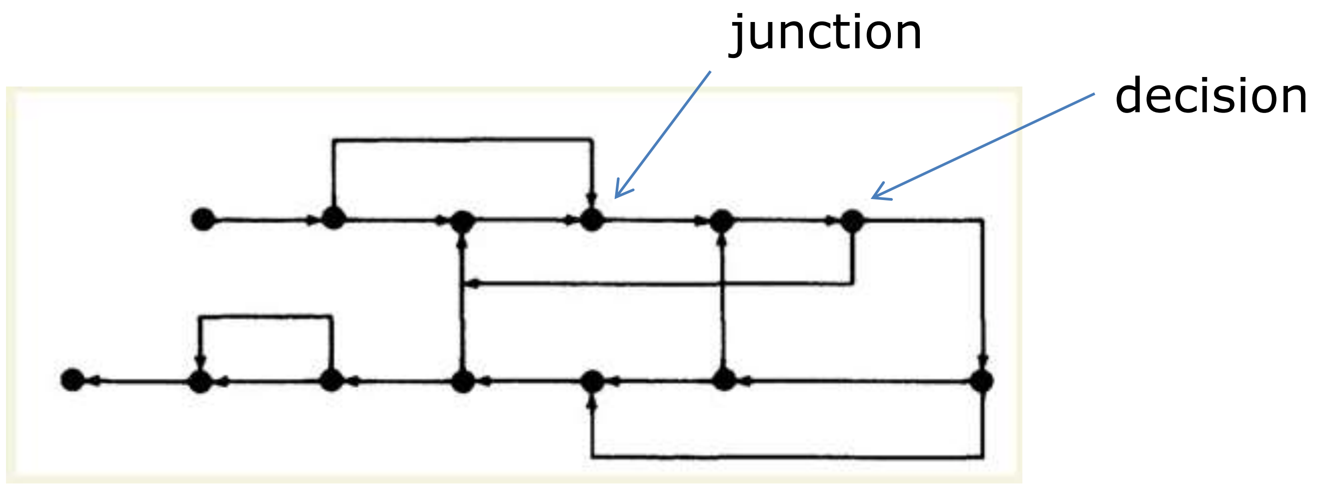
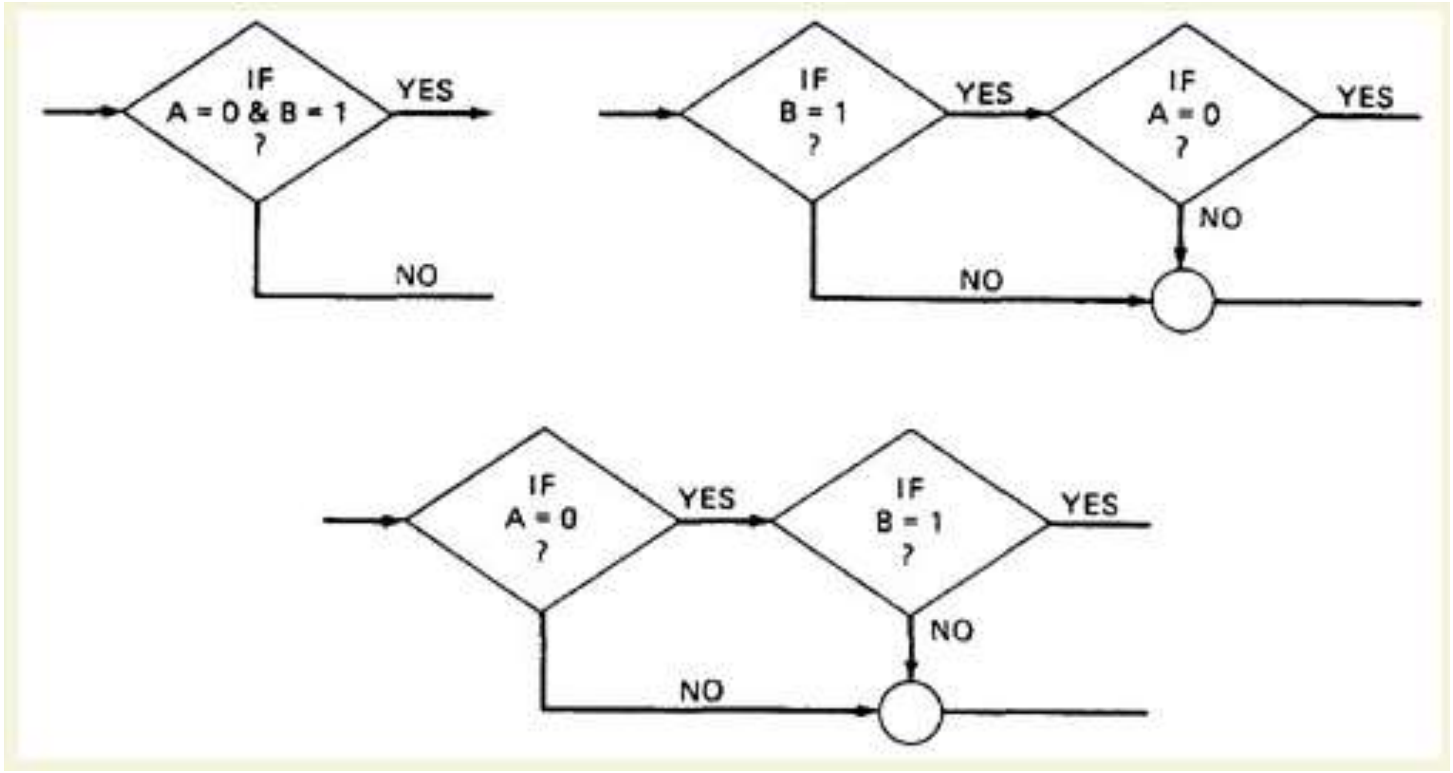


Figure 3.6. Even simpler flowgraph notation

Alternate flowgraphs for same logic



Paths, Nodes and Links

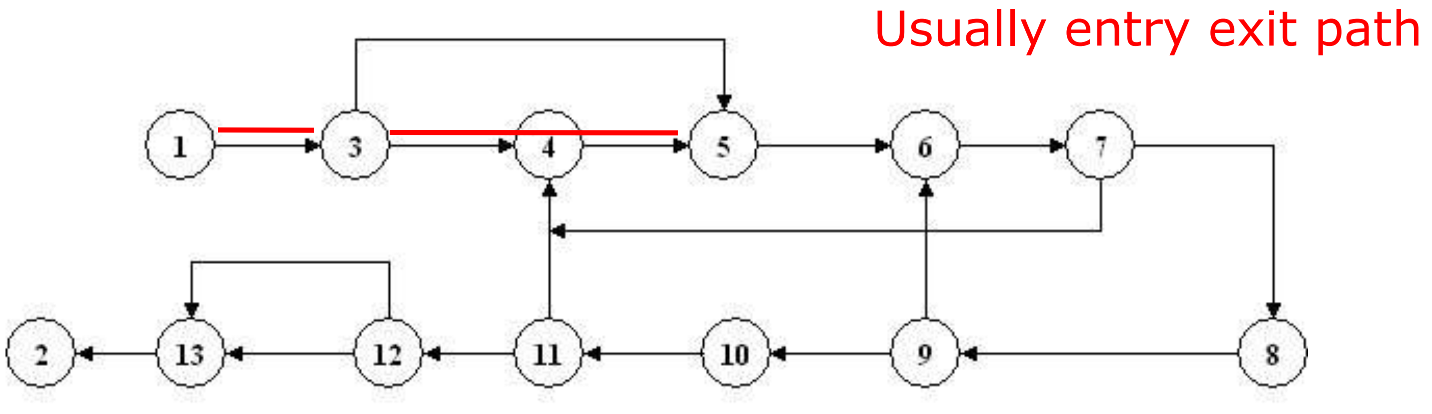


Figure 3.5. Simplified flowgraph notation

Path starts at an entry, junction, decision and ends at another

Path consists of segments

The smallest segment is a link

The length of a path is measure by number of links traversed
Not by the number of statements executed

Name of a path may be "(1, 3, 5, 6, 7, 8, 10, 11, 12, 13, 2)" or
"(1, 3, 4, 5, 6, 7, 4, 5, 6, 7, 8, 10, 11, 12, 13, 2)"

Loop

Fundamental Path Selection Criteria

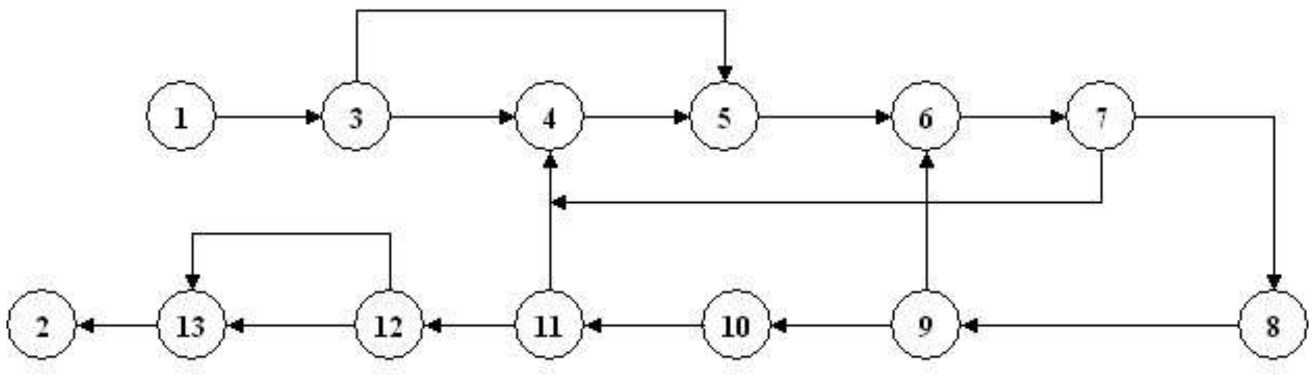


Figure 3.5. Simplified flowgraph notation

- Every decision doubles the number of potential paths
- Every loop multiplies the potential paths
- For a loop, each pass (once, twice, thrice, …) constitutes a different path
- Lavish test might consider all paths, but that would not be a complete test
  - A bug could create unwanted paths or make mandatory paths un-executable

Complete Testing

1. Exercise every path from entry to exit.
2. Exercise every statement/instruction at least once
3. Exercise every branch and case statement, in each direction, at least once

- If perception 1 is followed then perception 2 and 3 automatically followed
- But perception 1 is impractical for most routines. Can only be done with routines that have no loops.
- Perception 2 and 3 appear to be equivalent, but they are not

Static vs. Dynamic Analysis

- Static Analysis
  - Based on examining the source code or structure
  - Can’t determine whether code is reachable or not
  - Previous example, subroutine call with label parameter 100
- Dynamic Analysis
  - Based on code’s behavior while running
  - Can determine whether code is reachable or not

Path Testing Strategies

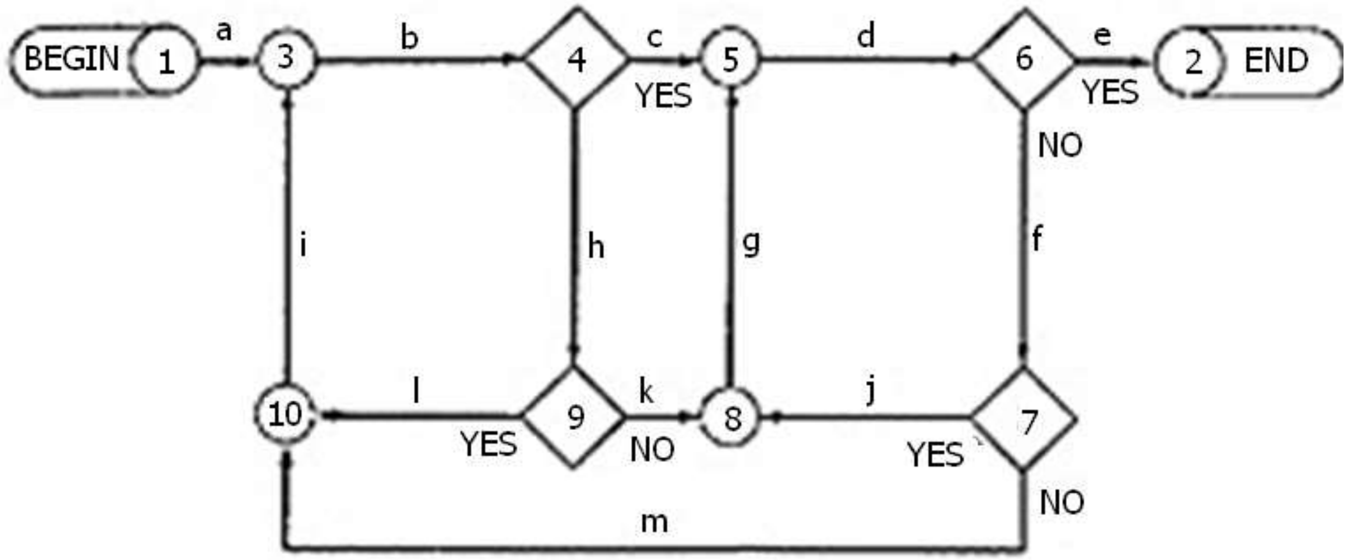
- Path Testing (P_∞)
  - Execute all possible paths i.e. 100% path coverage
  - Impossible to achieve, strongest criteria
- Statement Testing (P_1)
  - Execute all statements at least once
  - 100% statement or node coverage, denoted as C1
  - Weakest criteria
- Branch Testing (P_2)
  - 100% branch or link coverage, denoted as C2

There are infinite number of strategies P_1, P_2, … P_∞

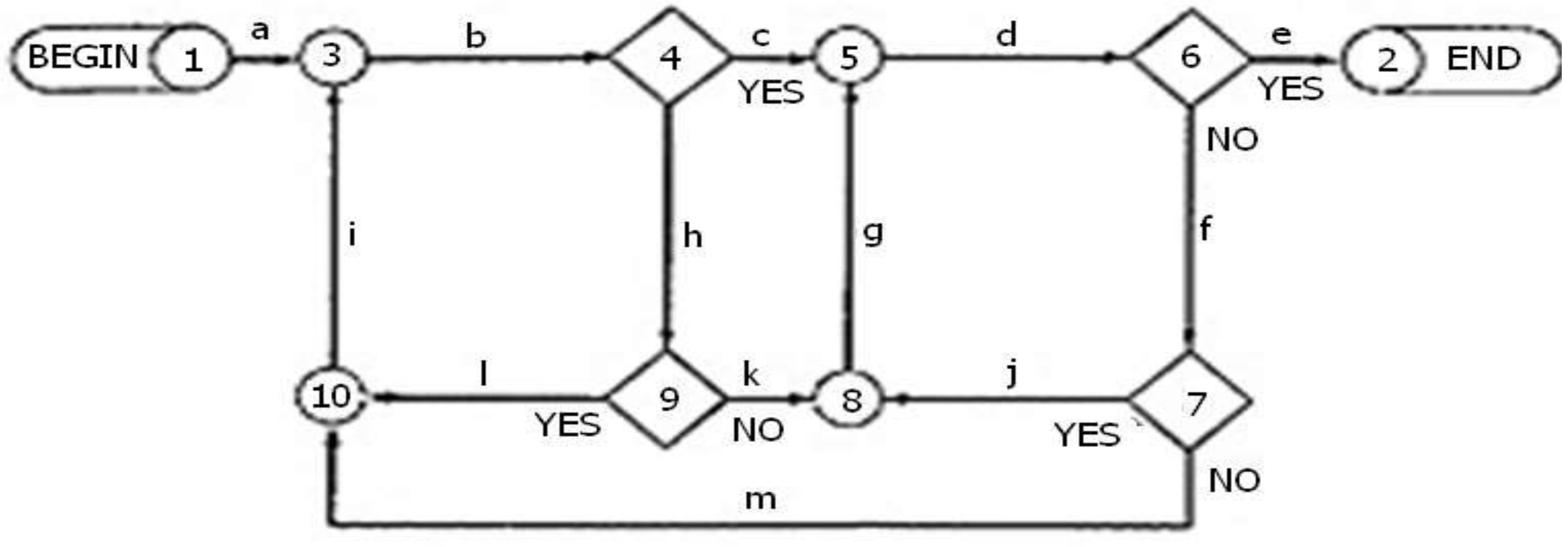
But even that's insufficient to exhaust testing

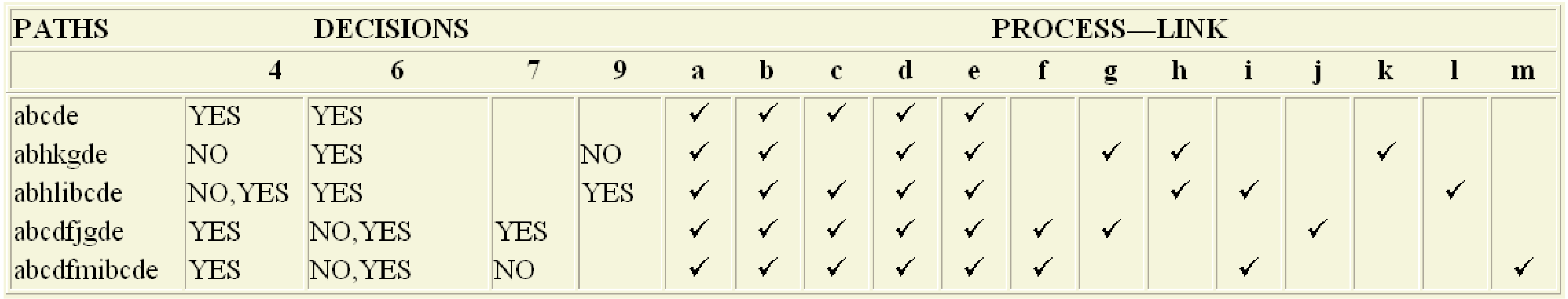
C_1 and C_2 are the minimum requirement

Which paths to select?



- Pick up paths to achieve C1 + C2
- Start at the beginning and take the most obvious path (1, 3, 4, 5, 6, 2) or (*abcde*)
- Take the next most obvious path, *abhkgde*
- Take a simple loop, preferably with previous path, such as *abhlibcde*
- Then take another loop, *abcdfjgde*
- Finally, *abcdfmibcde*





PATHS	DECISIONS				PROCESS—LINK												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES		NO	✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abcdfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abcdfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

- Check the following
  - Does every decision have a YES and a NO in its column? (C2)
  - Has every case of all case statements been marked? (C2)
  - Is every three-way branch (less, equal, greater) covered? (C2)
  - Is every link (process) covered at least once? (C1)

Suggestions to select paths

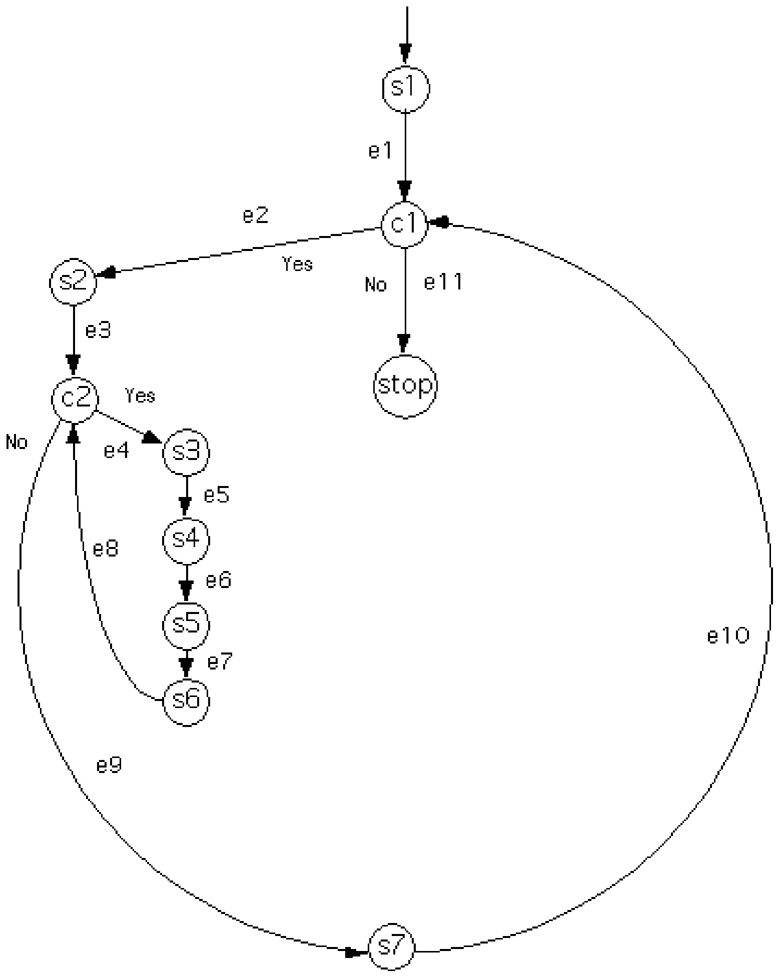
- Pick the simplest, functionally sensible entry/exit path
- Pick additional paths as small variations of previous paths.
- Its better to have small paths, each differing by only one thing, rather paths that cover more but have several changes.
- Cost of extra paths are few more microseconds of computer time.
- Be comfortable with your chosen paths. Play your hunches and give your intuition free reign as long as you achieve C1+C2.
- Don’t follow rules slavishly – except for coverage.

Path Testing Example

```
S1   i := 2
C1   while (i is less than or equal to n) do
S2     j := i - 1
C2     while ((j is greater than or equal to 1) and
                    (A[j] is greater than A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```

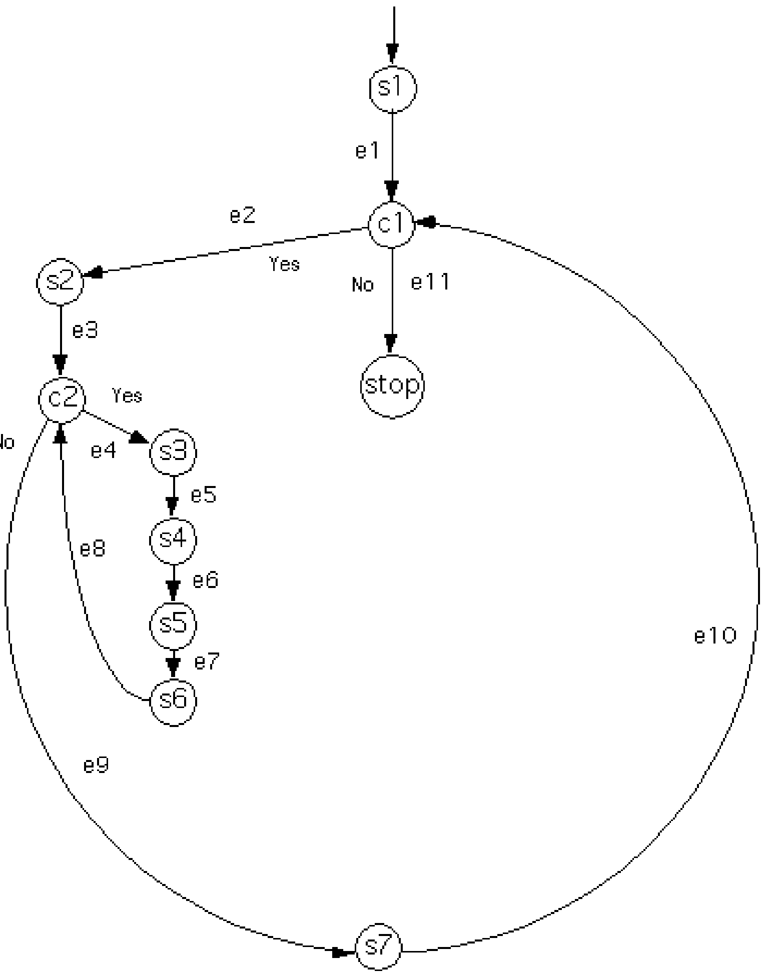
Sort an input array

Cyclomatic Complexity is 3
- Three regions
- N= 10, E = 11, E-N+2 = 3



Design of Tests

```
S1   i := 2
C1   while (i is less than or equal to n) do
S2     j := i - 1
C2     while ((j is greater than or equal to 1) and
                 (A[j] is greater than A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```

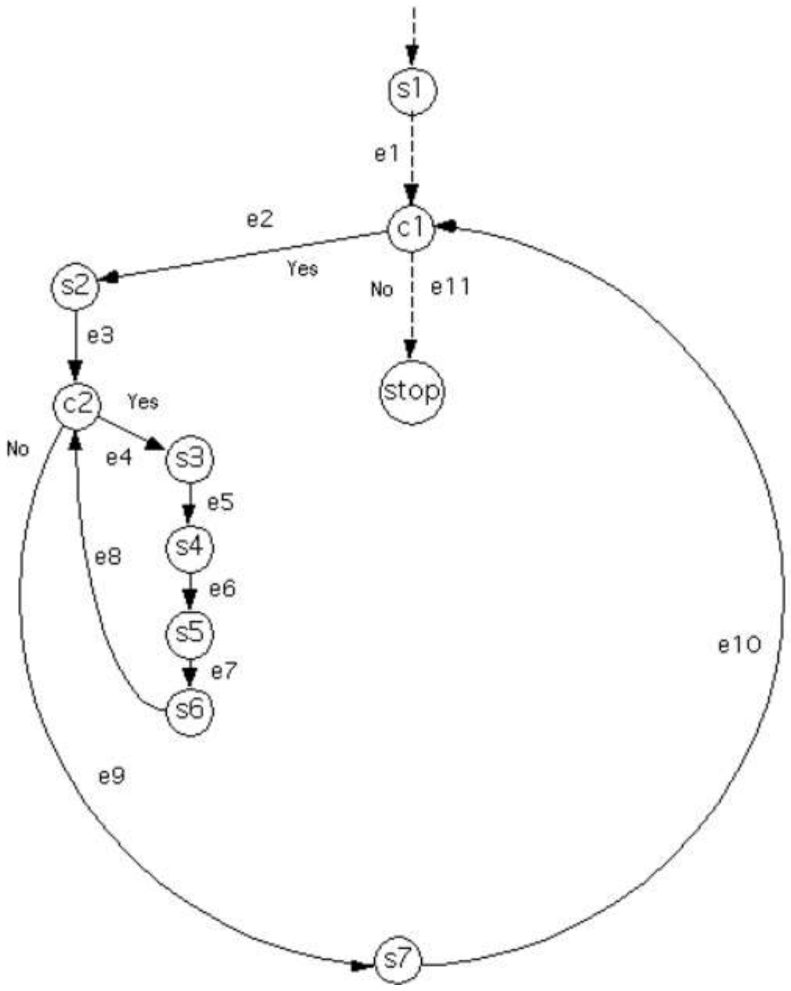


Skip the outer loop

Test #1
Inputs: `n = 1; A[1] = 1`
Expected Outputs: `A[1] = 1`

Design of Tests

```
S1   i := 2
C1   while (i is less than or equal to n) do
S2     j := i - 1
C2     while ((j is greater than or equal to 1) and
                 (A[j] is greater than A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```



Skip the outer loop

Test #1
Inputs: `n = 1; A[1] = 1`
Expected Outputs: `A[1] = 1`

Dashed edges are covered

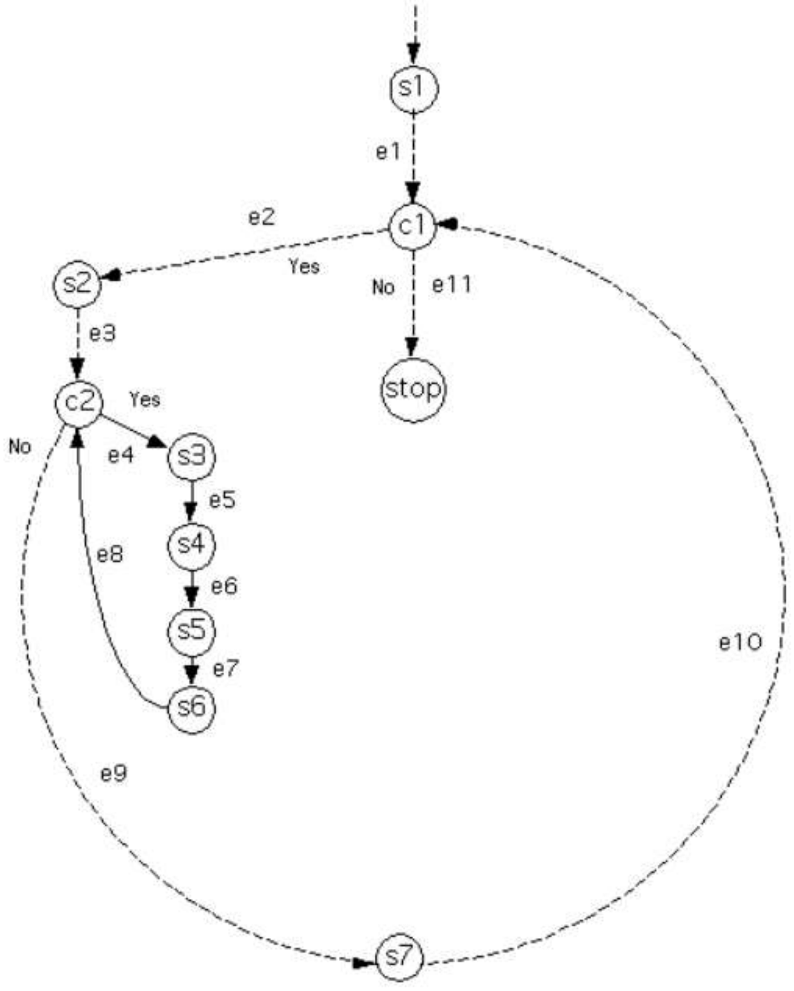
Design of Tests

```
S1   i := 2
C1   while (i is less than or equal to n) do
S2     j := i - 1
C2     while ((j is greater than or equal to 1) and
                  (A[j] is greater than A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```

Outer loop once
skip inner loop

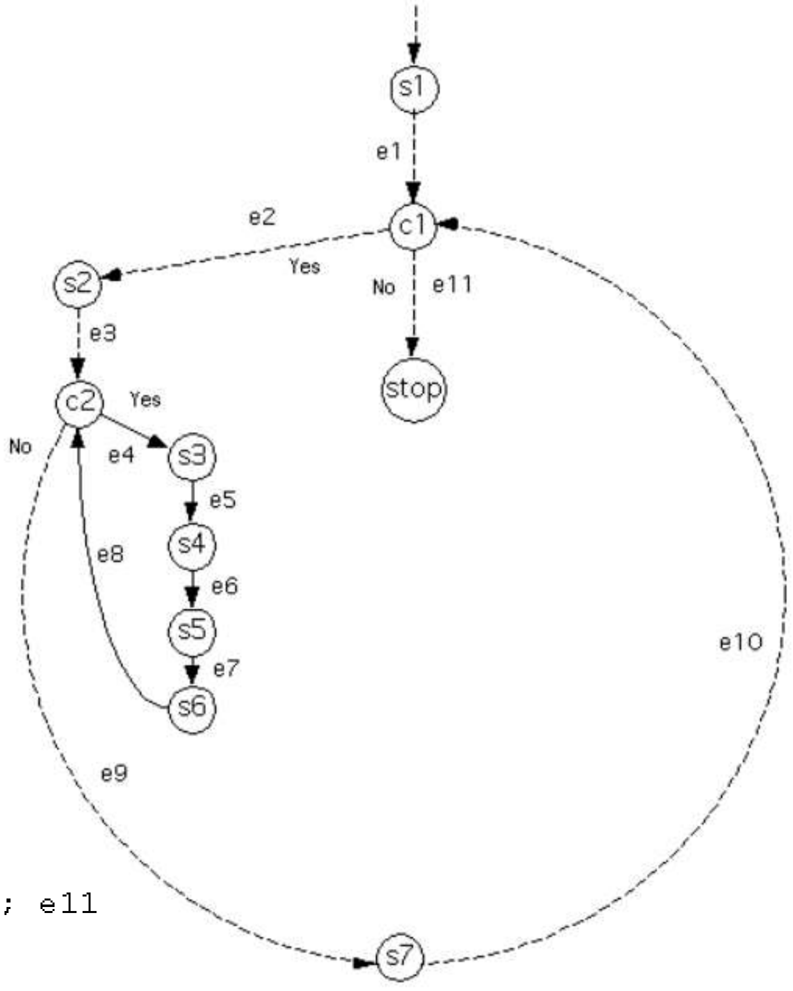
Test #2
Inputs: `n=2; A[1] = 1, A[2] = 2`
Expected Output: `A[1] = 1, A[2] = 2`

```
e1; e2; e3; e9; e10; e11
```



Design of Tests

```
S1   i := 2
C1   while (i is less than or equal to n) do
S2     j := i - 1
C2     while ((j is greater than or equal to 1) and
                  (A[j] is greater than A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```



Inner loop once

Test #3
Inputs: `n=2; A[1] = 2, A[2] = 1`
Expected Output: `A[1] = 1, A[2] = 2`

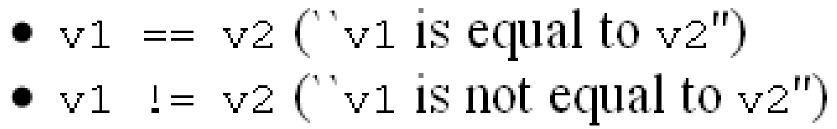
```
e1; e2; e3; e4; e5; e6; e7; e8; e9; e10; e11
```

Condition Testing

- Path testing provides limited coverage
- Condition testing much better coverage but complex
- Three types of conditions
  - Simple conditions
    - A Boolean variable or application
  - Relational Expressions
  - Compound conditions

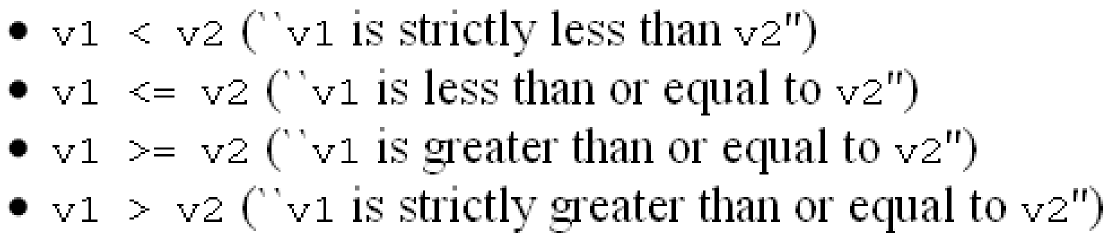
Relational Expressions and Compound Conditions

- Comparisons of two values of same type



  - v1 == v2 (``v1 is equal to v2'')
  - v1 != v2 (``v1 is not equal to v2'')

- If the type is ``ordered'' then additional relational expressions are also possible



  - v1 < v2 (``v1 is strictly less than v2'')
  - v1 <= v2 (``v1 is less than or equal to v2'')
  - v1 >= v2 (``v1 is greater than or equal to v2'')
  - v1 > v2 (``v1 is strictly greater than or equal to v2'')

- Combines simple conditions using logical connectives, such as and (&&), or (||), not (!)

Guidelines for Test Design

- Boolean variables
  - Test for true condition
  - Test for false condition
- Relational Expressions of ordered type
  - Three tests, less than, equal, and greater than
- Relational Expressions of unordered type
  - Two tests, Equal and not equal

Guidelines for Compound Conditions

- If complex condition connects *k* simple conditions with three cases, and *h* simple conditions with two cases, together.
- Then, a total of $3^k 2^h$ cases
- Some might be impossible to achieve, e.g.

  i. `x < y,`
  ii. `y < z,`
  iii. `x < z`

- `(x < y) && (y < z) && (x > z)` impossible to achieve
- So, check only the possible conditions

Application to an Example

```
S1   i := 2
C1   while (i is less than or equal to n) do
S2     j := i - 1
C2     while ((j is greater than or equal to 1) and
                 (A[j] is greater than A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```

First Condition

The test for the outer loop is (or can be written as) `i <= n`

so there are three cases that should be covered by tests

i. `i < n`

ii. `i == n`

iii. `i > n`

Application to an Example

```
S1   i := 2
C1   while (i is less than or equal to n) do
S2     j := i - 1
C2     while ((j is greater than or equal to 1) and
                 (A[j] is greater than A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```

Second Condition

The second condition, the inner loop, is a compound condition

`(j >= 1) && (A[j] > A[j+1])`

so there are $3^2 = 9$ combinations

i. `(j < 1) && (A[j] < A[j+1])`
ii. `(j < 1) && (A[j] == A[j+1])`
iii. `(j < 1) && (A[j] > A[j+1])`
iv. `(j == 1) && (A[j] < A[j+1])`
v. `(j == 1) && (A[j] == A[j+1])`
vi. `(j == 1) && (A[j] > A[j+1])`
vii. `(j > 1) && (A[j] < A[j+1])`
viii. `(j > 1) && (A[j] == A[j+1])`
ix. `(j > 1) && (A[j] > A[j+1])`

Cases Covered by Existing Tests

First Condition

✗ i. `i < n`

✓ ii. `i == n` T2, T3

✓ iii. `i > n` T1

Test #1
Inputs: `n = 1; A[1] = 1`
Expected Outputs: `A[1] = 1`

Test #2
Inputs: `n=2; A[1] = 1, A[2] = 2`
Expected Output: `A[1] = 1, A[2] = 2`

Test #3
Inputs: `n=2; A[1] = 2, A[2] = 1`
Expected Output: `A[1] = 1, A[2] = 2`

Second Condition

so there are $3^2 = 9$ combinations

✓ i. `(j < 1) && (A[j] < A[j+1])`

✓ ii. `(j < 1) && (A[j] == A[j+1])`

✓ iii. `(j < 1) && (A[j] > A[j+1])`

✓ iv. `(j == 1) && (A[j] < A[j+1])` T2

✗ v. `(j == 1) && (A[j] == A[j+1])`

✓ vi. `(j == 1) && (A[j] > A[j+1])`

✗ vii. `(j > 1) && (A[j] < A[j+1])`

✗ viii. `(j > 1) && (A[j] == A[j+1])`

✗ ix. `(j > 1) && (A[j] > A[j+1])`

```
S1   i := 2
C1   while ( i <= n ) do
S2     j := i - 1
C2     while ((j >= 1)&&(A[j] > A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```

Cases Covered by Existing Tests

First Condition

- ✓ i. `i < n`
- ✓ ii. `i == n` T2, T3
- ✓ iii. `i > n` T1

Test #4 covering the fifth case
Inputs: `n = 2; A[1] = 1, A[2] = 1`
Expected Outputs: `A[1] = 1, A[2] = 1`

Test #5 cover the seventh case
Inputs: `n=3; A[1] = 1, A[2] = 2, A[3] = 3`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 3`

Test #6 will cover the eighth case
Inputs: `n=3; A[1] = 1, A[2] = 2, A[3] = 2`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 2`

Test #7 cover the remaining ninth case
Inputs: `n = 4; A[1] = 1, A[2] = 3, A[3] = 2`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 3`

Second Condition

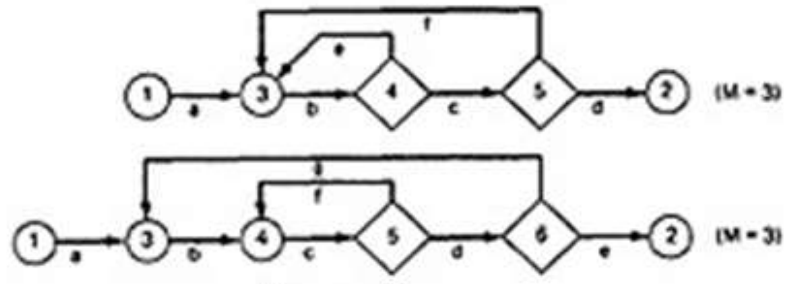
so there are $3^2 = 9$ combinations

- ✓ i. `(j < 1) && (A[j] < A[j+1])`
- ✓ ii. `(j < 1) && (A[j] == A[j+1])`
- ✓ iii. `(j < 1) && (A[j] > A[j+1])`
- ✓ iv. `(j == 1) && (A[j] < A[j+1])` T2
- ✓ v. `(j == 1) && (A[j] == A[j+1])`
- ✓ vi. `(j == 1) && (A[j] > A[j+1])`
- ✓ vii. `(j > 1) && (A[j] < A[j+1])`
- ✓ viii. `(j > 1) && (A[j] == A[j+1])`
- ✓ ix. `(j > 1) && (A[j] > A[j+1])`

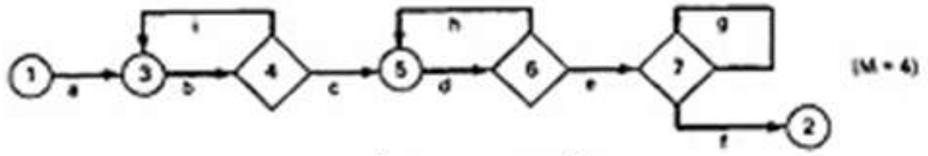
```
S1   i := 2
C1   while ( i <= n ) do
S2     j := i - 1
C2     while ((j >= 1)&&(A[j] > A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```

Loops

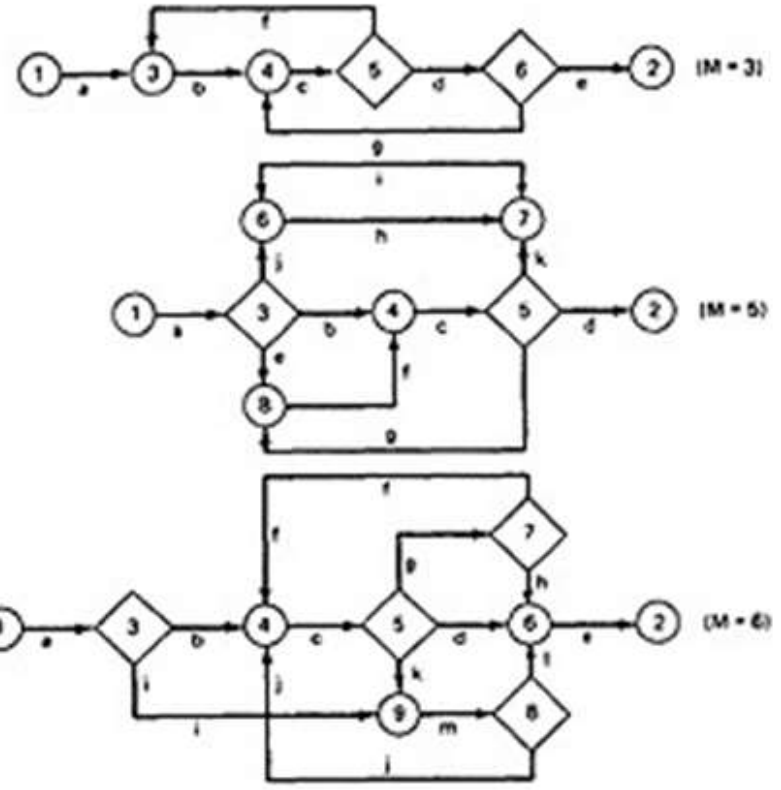
- Three kinds:
  - Nested
  - concatenated
  - horrible



a,b) nested loops



c) concatenated loops



d,e,f) horibble loops

Cases for a single loop

- Case 1: *Single Loop, Zero Minimum, N Maximum, No Excluded Values*
  - Try bypassing the loop (zero iterations). If you can’t, you either have a bug, or zero is not the minimum and you have the wrong case.
  - Could the loop–control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?

Cases for a single loop

- Case 1: *Single Loop, Zero Minimum, N Maximum, No Excluded Values*
  - One pass through the loop.
  - Two passes through the loop for reasons discussed below.
  - A typical number of iterations, unless covered by a previous test.
  - One less than the maximum number of iterations.
  - The maximum number of iterations.
  - Attempt one more than the maximum number of iterations

Cases for a single loop

- Case 1: *Single Loop, Zero Minimum, N Maximum, No Excluded Values*
  - Some data–flow anomalies, such as some initialization problems, can be detected only by two passes through the loop.
  - The problem occurs when data are initialized within the loop and referenced after leaving the loop.
  - If, because of bugs, a variable is defined within the loop but is not referenced or used in the loop, only two traversals of the loop would show the double initialization.

Cases for a single loop

- *Case 2—Single Loop, Nonzero Minimum, No Excluded Values*
  - Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
  - The minimum number of iterations.
  - One more than the minimum number of iterations.
  - Once, unless covered by a previous test.
  - Twice, unless covered by a previous test.
  - A typical value. One less than the maximum value.
  - The maximum number of iterations.
  - Attempt one more than the maximum number of iterations.

Cases for a single loop

- *Case 3—Single Loops with Excluded Values*
  - Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as Cases 1 and 2 above.
  - Let the loop–control variable is 1 to 20, but 7, 8, 9, and 10 are excluded.
  - The two sets of tests are 1–6 and 11–20.
  - Attempt would be <u>0</u>,1,2,4,6,<u>7</u>, for the first range and <u>10</u>,11,15,19,20,<u>21</u>, for the second range.
  - Underlined cases are not supposed to work, but they should be attempted
  - You may try a value within the excluded range, such as 8
  - If two sets of excluded values, then three sets of tests

Cases for a single loop

- *Case 3—Single Loops with Excluded Values*
  - If excluded values are very systematic and easily typified, this approach would entail too many tests.
  - Example: All odd values were excluded.
  - Then I would test the extreme points of the range as if there were no excluded values, for the extreme points, for the excluded values, and also for typical excluded values. For example, if the range is 0 to 20 and odd values are excluded, try – 1,0, 1,2,3,10,11,18,19,20,21,22.

Nested Loops

- If you had five tests for one loop
  - minimum, minimum + 1, typical, maximum – 1, and maximum
  - assuming that one less than the minimum and one more than the maximum were not achievable
- A pair of nested loops would require 25 tests
- Three nested loops would require 125
- This is heavy, reduce number of test cases

Nested Loops

1. Start innermost loop.
   - Set all the outer loops to their minimum values.
   - Test five cases minimum, minimum + 1, typical, maximum – 1, and maximum for the innermost loop
   - Test for out–of–range and excluded values if necessary.
2. If done with outermost loop
   - GOTO step 4, ELSE move out one loop, test five cases with all other loops set to typical values.
3. Continue outward until all loops are covered.
4. Do the five cases for all loops in the nest simultaneously

Nested Loops

- Consequence
  - Twelve tests for a pair of nested loops
  - Sixteen for three nested loops
  - Nineteen for four nested loops
  - Data initialization problems are not identified
- Huang’s twice-through theorem should also be applied for combination of loops
  - Outer loops at the minimum, inner loop five cases
  - Outer loop at one, inner loop five cases
  - Outer loop at two, inner loop five cases
  - Reverse the role of inner and outer loop, do again.

Concatenated Loop

- Two loops are **concatenated** if they are on the same path
- Even if on the same path and they are independent of each other, treat them as individual loops
- Other wise treat them as you would nested loops.

Loop Testing Time

- It can take long testing time if MAX is higher and we attempt (MAX-1, MAX, MAX+1).
  - Three nested loops with limits 10000, 200, 8000, a total of 16,000,000,000 iterations.
  - You can test for 100, 20 and 80
  - Avoid rescaling 2, 4, 8, specially 256, 65536 i.e. binary powers; peculiarity of the numbers may mask a bug

Example Loop Testing

```
S1  i := 2
C1  while ( i <= n ) do
S2    j := i - 1
C2    while ((j >= 1)&&(A[j] > A[j+1])) do
S3      temp := A[j]
S4      A[j] := A[j+1]
S5      A[j+1] := temp
S6      j := j-1
      end while
S7    i := i + 1
    end while
```

- Inner loop iteration is always one less than *i*
- Minimal inner loop iteration is zero, why?
- Inner loop iterates when smaller value moves forward. At any point of time, data is sorted
- Outer loop always *n-1* times

Example Loop Testing

- Let’s sort arrays of length at least 100
- Both inner and outer loop should be tested on inputs of ``moderate'' sizes (say, some size between 40 and 60), as well as on sizes 1, 2, 99, and 100.
  - To keep inner loop ‘typical’ value, use random values
  - For minimum inner loop iteration, use sorted value
  - For maximum inner loop iteration, decreasing order

Example Loop Testing

Test #1
Inputs: `n = 1; A[1] = 1`
Expected Outputs: `A[1] = 1`

Test #2
Inputs: `n=2; A[1] = 1, A[2] = 2`
Expected Output: `A[1] = 1, A[2] = 2`

Test #3
Inputs: `n=2; A[1] = 2, A[2] = 1`
Expected Output: `A[1] = 1, A[2] = 2`

Test #4 covering the fifth case
Inputs: `n = 2; A[1] = 1, A[2] = 1`
Expected Outputs: `A[1] = 1, A[2] = 1`

Test #5 cover the seventh case
Inputs: `n=3; A[1] = 1, A[2] = 2, A[3] = 3`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 3`

Test #6 will cover the eighth case
Inputs: `n=3; A[1] = 1, A[2] = 2, A[3] = 2`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 2`

Test #7 cover the remaining ninth case
Inputs: `n = 4; A[1] = 1, A[2] = 3, A[3] = 2`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 3`

- From path testing and condition testing
  - None use input arrays more than 4
  - So none require execution of inner or outer loop more than three times
  - Worse yet, none of them executed inner loop more than one.
- Thus add one more test (for twice inner loop) to complete testing of loops at the minimal values

Example Loop Testing

Test #1
Inputs: `n = 1; A[1] = 1`
Expected Outputs: `A[1] = 1`

Test #2
Inputs: `n=2; A[1] = 1, A[2] = 2`
Expected Output: `A[1] = 1, A[2] = 2`

Test #3
Inputs: `n=2; A[1] = 2, A[2] = 1`
Expected Output: `A[1] = 1, A[2] = 2`

Test #4 covering the fifth case
Inputs: `n = 2; A[1] = 1, A[2] = 1`
Expected Outputs: `A[1] = 1, A[2] = 1`

Test #5 cover the seventh case
Inputs: `n=3; A[1] = 1, A[2] = 2, A[3] = 3`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 3`

Test #6 will cover the eighth case
Inputs: `n=3; A[1] = 1, A[2] = 2, A[3] = 2`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 2`

Test #7 cover the remaining ninth case
Inputs: `n = 4; A[1] = 1, A[2] = 3, A[3] = 2`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 3`

Test #8 inner loop body is executed twice
Inputs: `n = 3; A[1] = 3, A[2] = 2, A[1] = 1`
Expected Outputs: `A[1] = 1, A[2] = 2, A[3] = 3`

- Thus add one more test (for twice inner loop) to complete testing of loops at the minimal values

- Add other test to ensure that loops are tested at both ‘typical’ and ‘maximal’ values as well.

Example Loop Testing

Test #9
Inputs: `n=51`, `A[i] = i` for `i` between 1 and 50, and `A[51] = 0`
Expected Outputs: `A[i] = [i-1]` for `i` between 1 and 51

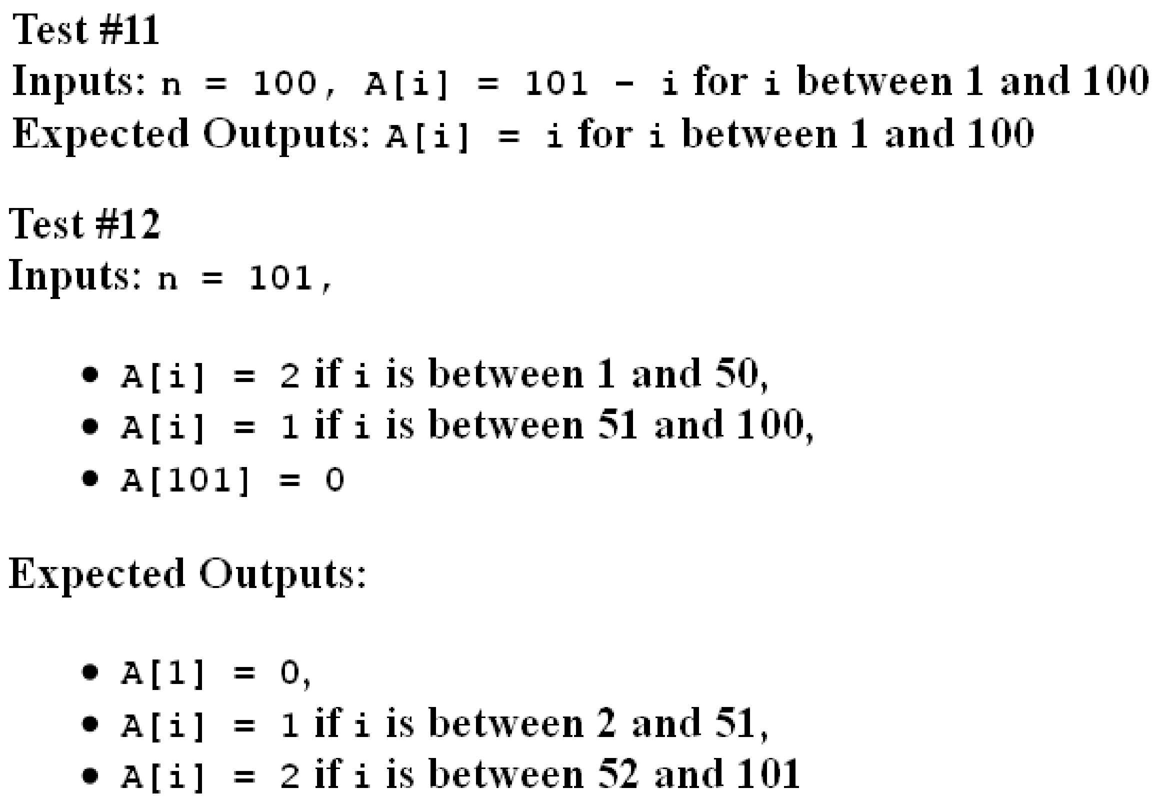
Test #10
Inputs: `n=99`,

- `A[i] = 1` if `i` is between 1 and 98 and is even,
- `A[i] = 2` if `i` is between 1 and 98 and is odd,
- `A[99] = 0`

Expected Output:

- **`A[1] = 0`,**
- **`A[i] = 1` if `i` is between 2 and 50,**
- **`A[i] = 2` if `i` is between 51 and 99**

Example Loop Testing



Test #11
Inputs: `n = 100, A[i] = 101 - i` for `i` between 1 and 100
Expected Outputs: `A[i] = i` for `i` between 1 and 100

Test #12
Inputs: `n = 101,`

- `A[i] = 2` if `i` is between 1 and 50,
- `A[i] = 1` if `i` is between 51 and 100,
- `A[101] = 0`

Expected Outputs:

- `A[1] = 0,`
- `A[i] = 1` if `i` is between 2 and 51,
- `A[i] = 2` if `i` is between 52 and 101

- Finally test the outer loop for 50, 98, 99 and 100 times

Effectiveness of Path Testing

- Approximately 65% of all bugs can be caught
- Path testing is more effective for unstructured than for structured software

Limitations of Path Testing

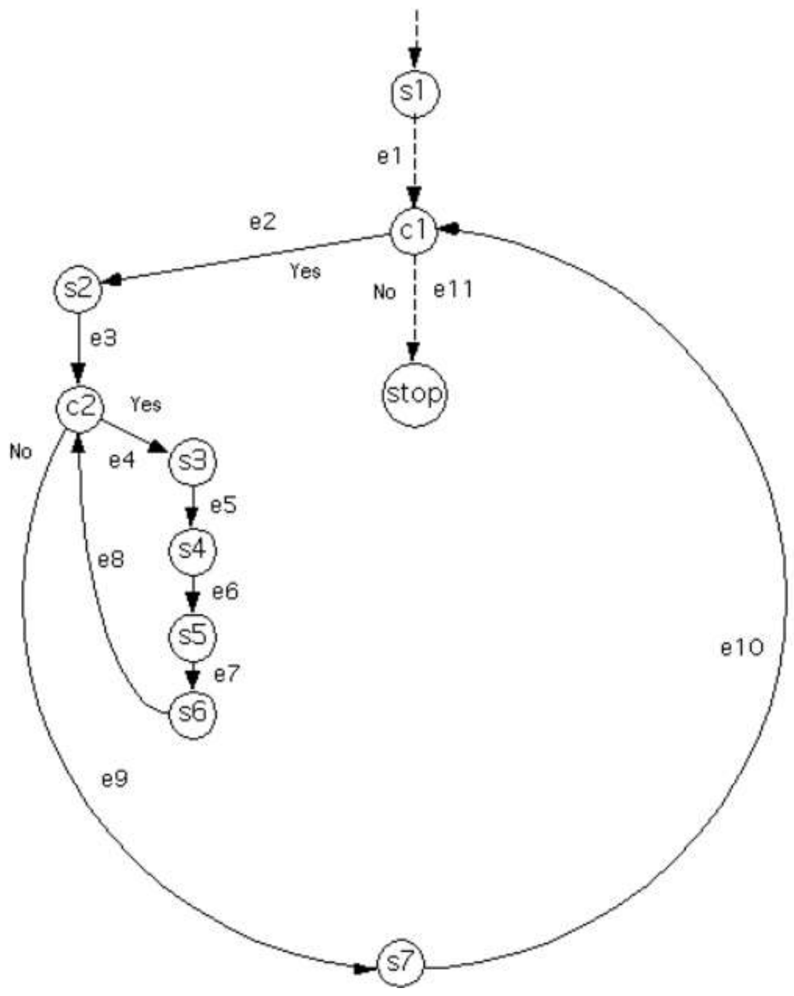
- May not cover if you have bugs, totally wrong or missing functions.
- Interface errors, i.e. interface with other routines
- Database and data–flow errors
  - The routine can pass all of its tests at the unit level, but the possibility that it interferes with or perturbs other routines cannot be determined by unit–level path tests.
- Not all initialization errors are caught by path testing.
- Specification errors can’t be caught.

A Lot of Work?

- Creating flowgraph, selecting covering paths, finding input data, setting up loop cases and combinations.
- But the fact is
  - *The act of careful, complete, systematic, test design will catch as many bugs as the act of testing.*
  - *The test design process, at all levels, is at least as effective at catching bugs as is running the test designed by that process.*
- bugs caught during test design cost less to fix than bugs caught during testing

Tricks to do it

```
S1   i := 2
C1   while ( i <= n ) do
S2     j := i - 1
C2     while ((j >= 1)&&(A[j] > A[j+1])) do
S3       temp := A[j]
S4       A[j] := A[j+1]
S5       A[j+1] := temp
S6       j := j-1
       end while
S7     i := i + 1
     end while
```



- but you can do it from source code
- Make several copies of source code
- Mark the path with a yellow pen

References

- Chapter 3, Software Testing Techniques - Boris Beizer