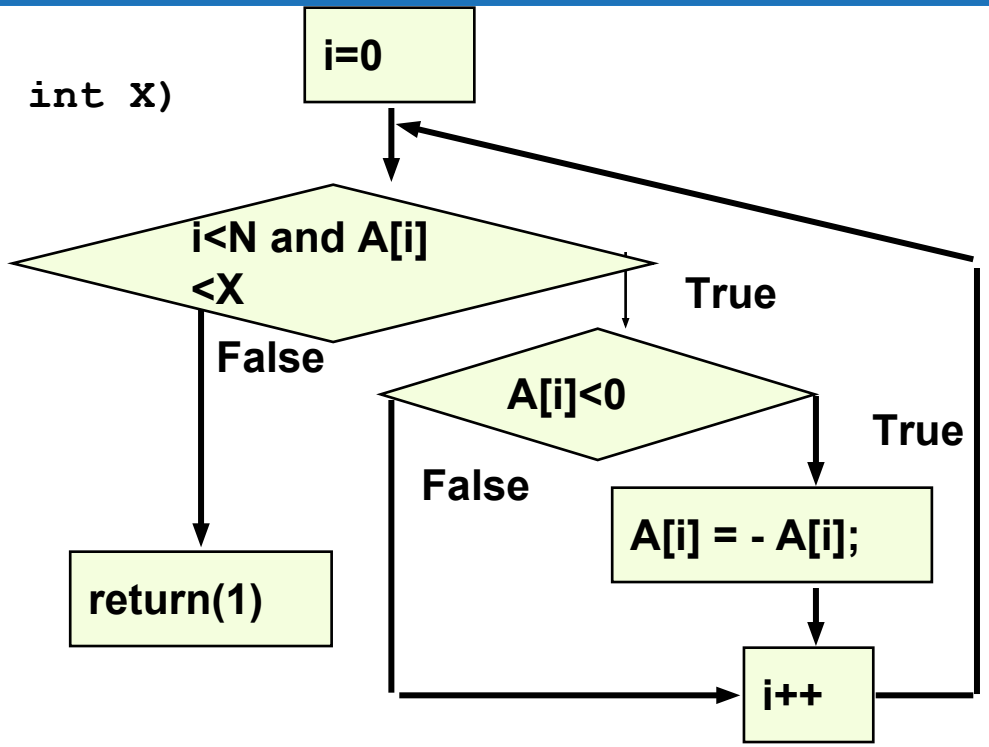
Path Coverage

- Other criteria focus on single elements.
  - However, all tests execute a sequence of elements - a path through the program.
  - Combination of elements matters - interaction sequences are the root of many faults.
- Path coverage requires that all paths through the CFG are covered.
- Coverage = $\frac{\text{Number of Paths Covered}}{\text{Number of Total Paths}}$

Path Coverage

```
int flipSome(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```



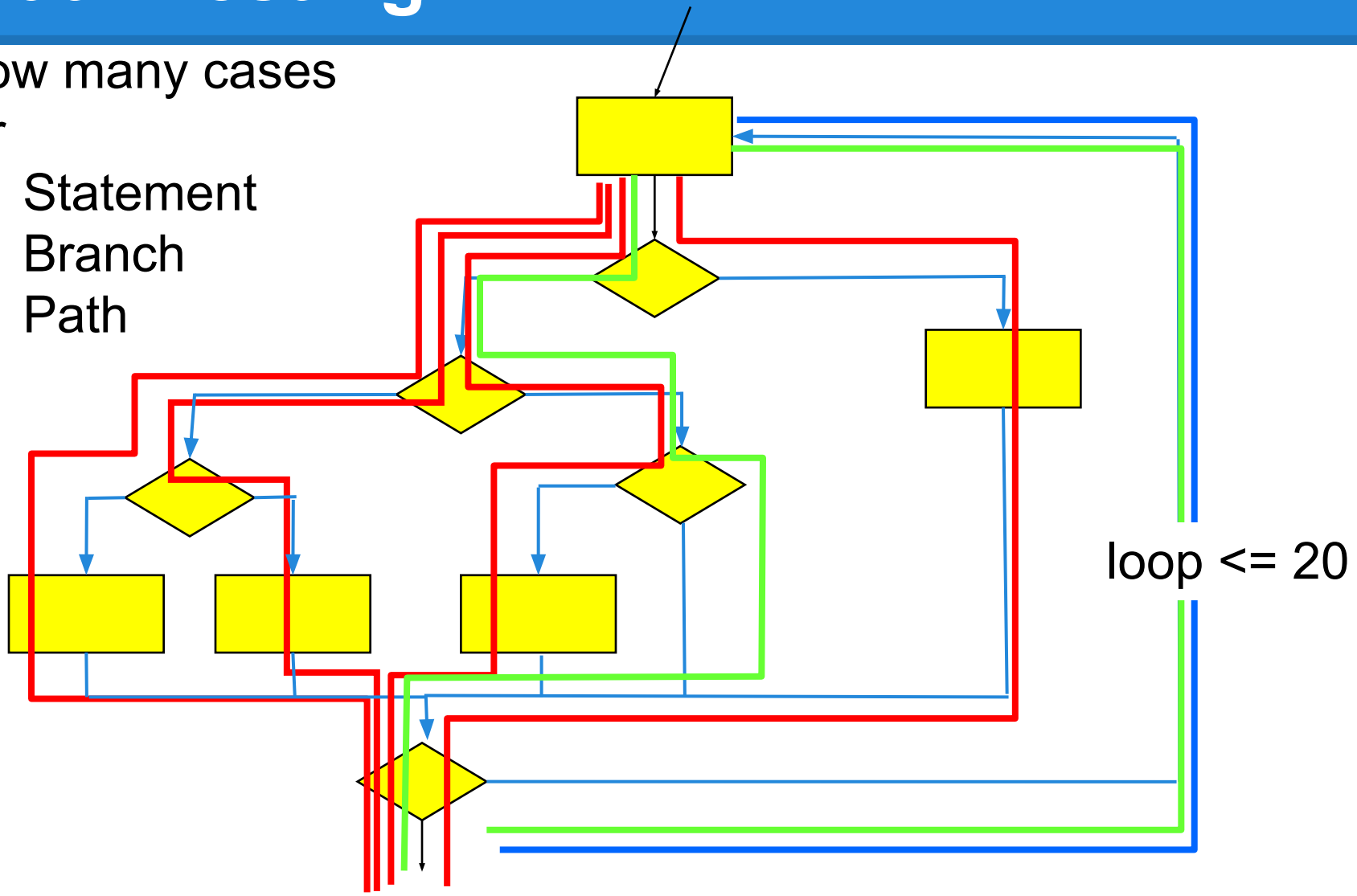
In theory, path coverage is the ultimate coverage metric.
In practice, it is impractical.

- **How many paths does this program have?**

Path Testing

How many cases for

- Statement
- Branch
- Path



Number of Tests

Path coverage for that loop bound requires: **3,656,158,440,062,976** test cases

If you run 1000 tests per second, this will take **116,000 years**.

However, there are ways to get some of the benefits of path coverage without the cost...

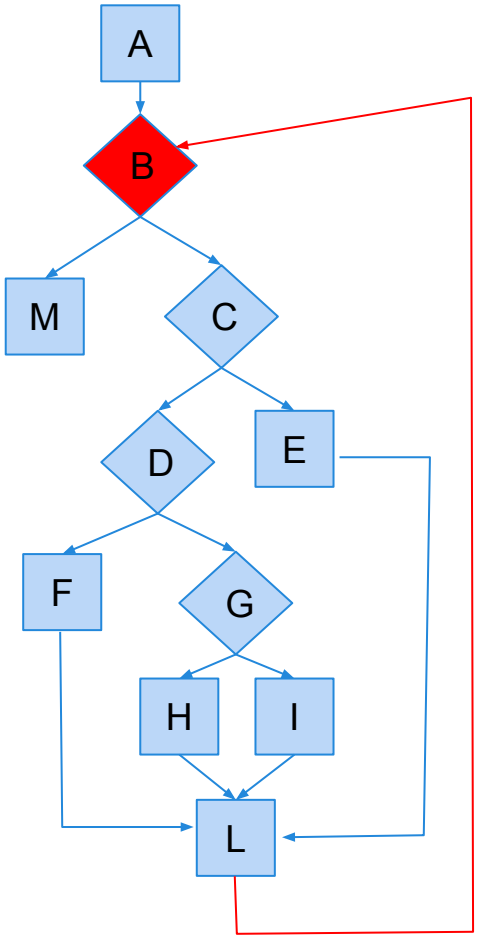
Path Coverage

- Theoretically, the strongest coverage metric.
  - Many faults emerge through sequences of interactions.
- But… Generally impossible to achieve.
  - Loops result in an infinite number of path variations.
  - Even bounding number of loop executions leaves an infeasible number of tests.

Boundary Interior Coverage

- Need to partition the infinite set of paths into a finite number of classes.
- **Boundary Interior Coverage** groups paths that differ only in the subpath they follow when repeating the body of a loop.
  - Executing a loop 20 times is a different path than executing it twice, but the same *subsequences* of statements repeat over and over.

Boundary Interior Coverage



B -> M
B -> C -> E -> L -> B
B -> C -> D -> F -> L -> B
B -> C -> D -> G -> H -> L -> B
B -> C -> D -> G -> I -> L -> B

Number of Paths

- Boundary Interior Coverage removes the problem of infinite loop-based paths.
- However, the number of paths through this code can still be exponential.
  - N non-loop branches results in 2^N paths.
- Additional limitations may need to be imposed on the paths tested.

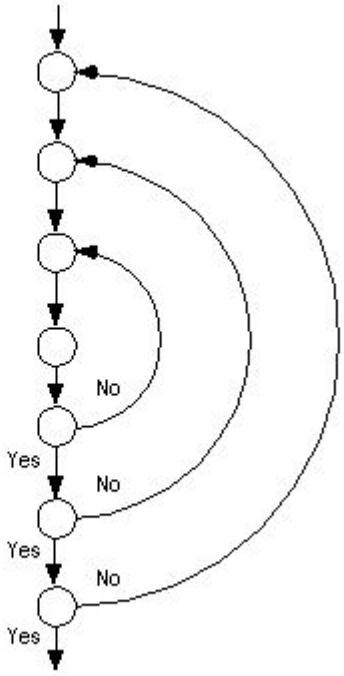
```
if (a)      S1;
if (b)      S2;
if (c)      S3;
...
if (x)      SN;
```

Loop Boundary Coverage

- Focus on problems related to loops.
- Cover *scenarios representative of how loops might be executed*.
- For simple loops, write tests that:
  - Skip the loop entirely.
  - Take exactly one pass through the loop.
  - Take two or more passes through the loop.
  - (optional) Choose an upper bound N, and:
    - M passes, where 2 < M < N
    - (N-1), N, and (N+1) passes

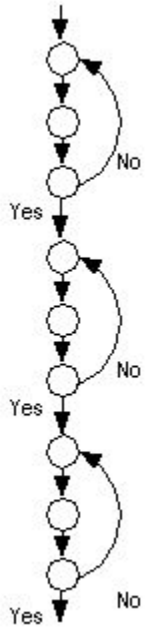
Nested Loops

- Often, loops are nested within other loops.
- For each level, you should execute similar strategies to simple loops.
- In addition:
  - Test innermost loop first with outer loops executed minimum number of times.
  - Move one loops out, keep the inner loop at “typical” iteration numbers, and test this layer as you did the previous layer.
  - Continue until the outermost loop tested.



Concatenated Loops

- One loop executes. The next line of code starts a new loop.
- These are generally independent.
  - Most of the time...
- If not, follow a similar strategy to nested loops.
  - Start with bottom loop, hold higher loops at minimal iteration numbers.
  - Work up towards the top, holding lower loops at “typical” iteration numbers.



Why These Loop Strategies?

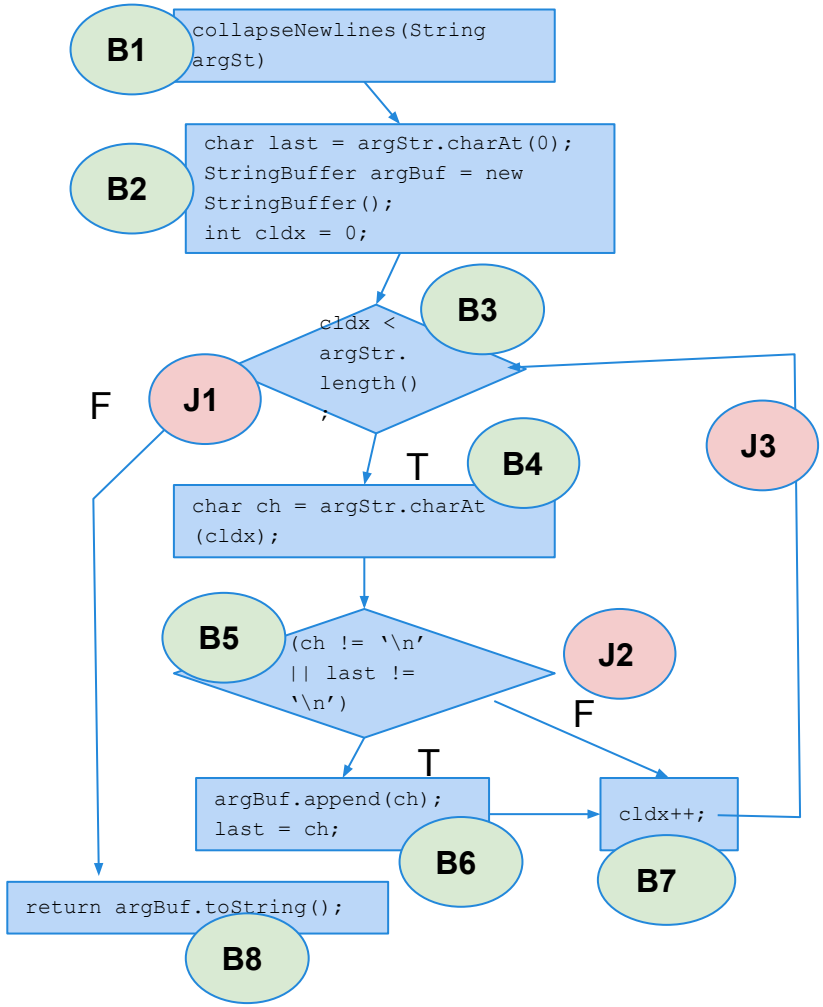
Why do these loop values make sense?

- In proving formal correctness of a loop, we would establish preconditions, postconditions, and invariants that are true on each execution of the loop, then prove that these hold.
  - The loop executes **zero** times when the postconditions are true in advance.
  - The loop invariant is true on loop entry (**one**), then each loop iteration maintains the invariant (**many**).
    - (invariant and !(loop condition) implies postconditions)
- Loop testing strategies echo these cases.

Linear Code Sequences and Jumps

- Often, we want to reason about the subpaths that execution can take.
- A subpath from one branch of control to another is called a LCSAJ.
- The LCSAJs for this example:

From	To	Sequence of Basic Blocks
entry	j1	b1, b2, b3
entry	j2	b1, b2, b3, b4, b5
entry	j3	b1, b2, b3, b4, b5, b6, b7
j1	return	b8
j2	j3	b7
j3	j2	b3, b4, b5
j3	j3	b3, b4, b5, b6, b7



LCSAJ Coverage

- We can require coverage of all sequences of LCSAJs of length *N*.
  - We can string subpaths into paths that connect *N* subpaths.
  - LCSAJ Coverage (N=1) is equivalent to statement coverage.
  - LCSAJ Coverage (N=2) is equivalent to branch coverage
- Higher values of N achieve stronger levels of path coverage.
- Can define a threshold that offers stronger tests while remaining affordable.

Procedure Call Testing

- Metrics covered to this point all look at code *within* a procedure.
- Good for testing individual units of code, but not well-suited for integration testing.
  - i.e., subsystem or system testing, where we bring together units of code and test their combination.
- Should also cover connections between procedures:
  - **calls** and **returns**.

Entry and Exit Testing

- A single procedure may have several entry and exit points.
  - In languages with goto statements, labels allow multiple entry points.
  - Multiple returns mean multiple exit points.
- Write tests to ensure these entry/exit points are entered and exited in the context they are intended to be used.

```
int status (String str){
    if(str.equals(”panic”))
        return 0;
    else if(str.contains(“+”))
        return 1;
    else if(str.contains(“-”))
        return 2;
    else
        return 3;
}
```

- Finds interface errors that statement coverage would not find.

Call Coverage

- A procedure might be called from multiple locations.
- Call coverage requires that a test suite executes all possible method calls.
- Also finds interface errors that statement/branch coverage would not find.

```
void orderPizza (String str){
    if(str.contains(”pepperoni”))
        addTopping(“pepperoni”);
    if(str.contains(“onions”))
        addTopping(“onions”);
    if(str.contains(“mushroom”))
        addTopping(“mushroom”)
}
```

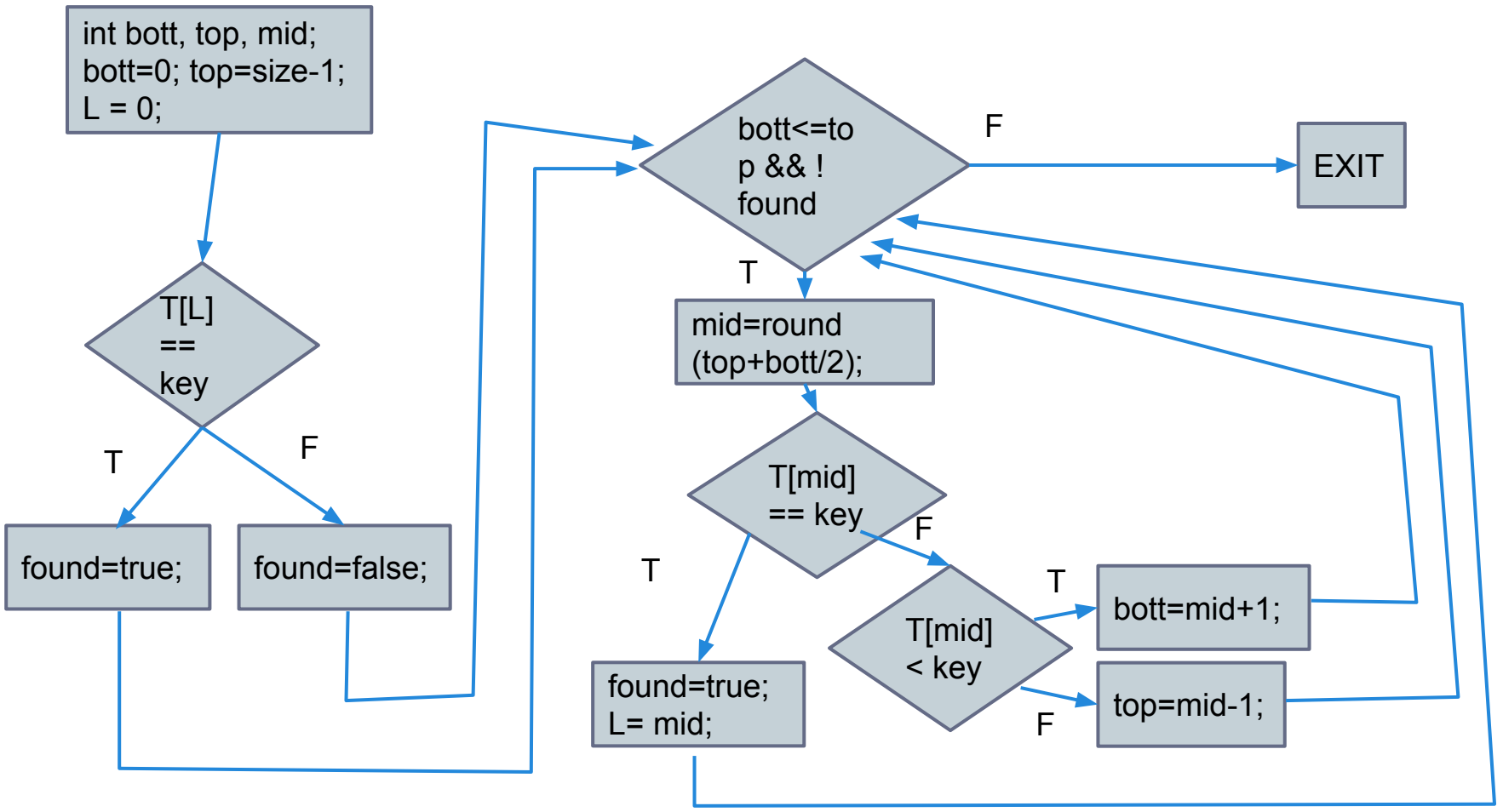
- Challenging for OO systems, where a method call might be bound to different objects at runtime.

Activity: Writing Loop-Covering Tests

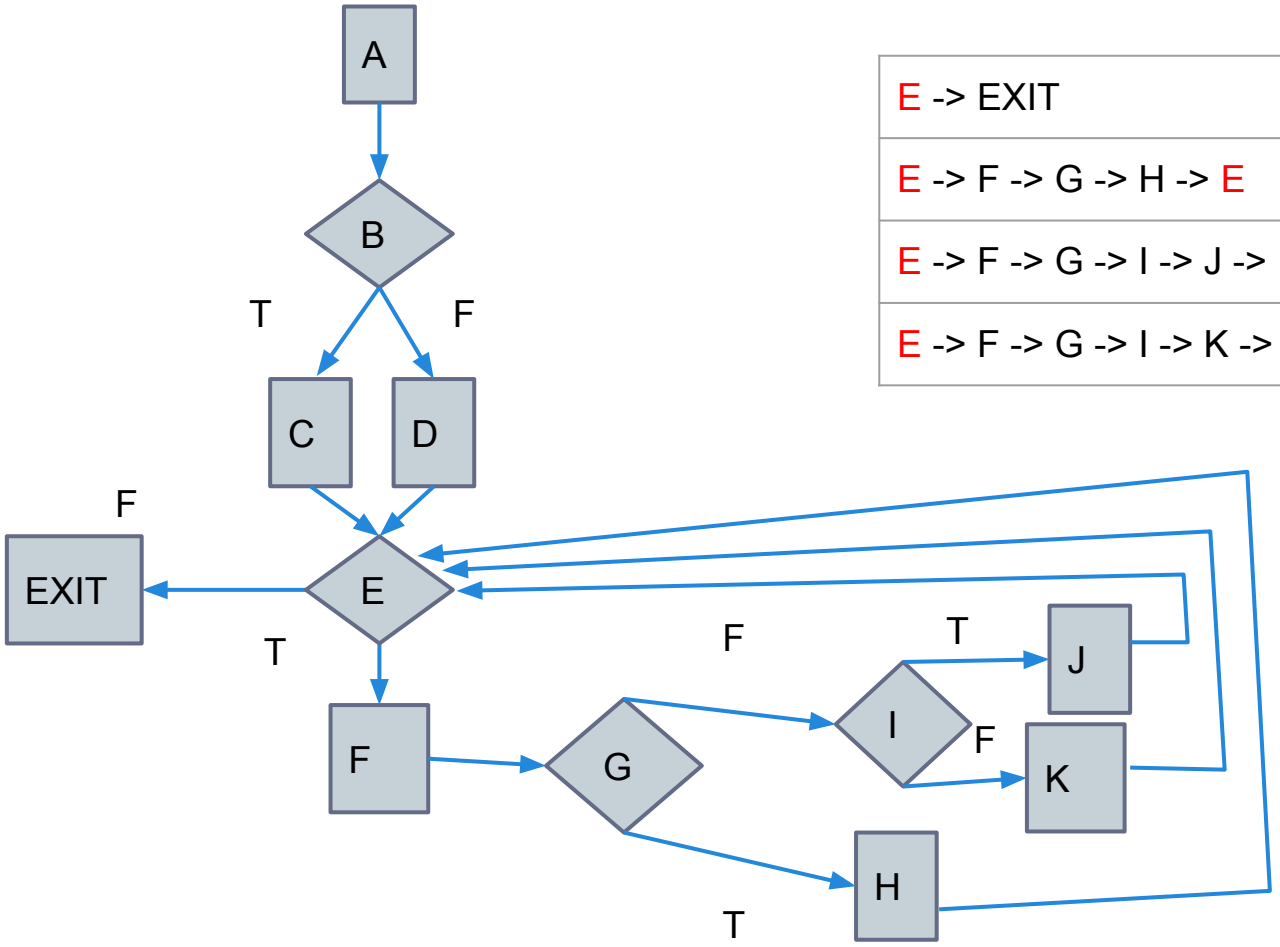
For the binary-search code:

1. Draw the control-flow graph for the method.
2. Identify the subpaths through the loop and draw the unfolded CFG for boundary interior testing.
3. Develop a test suite that achieves loop boundary coverage.

CFG

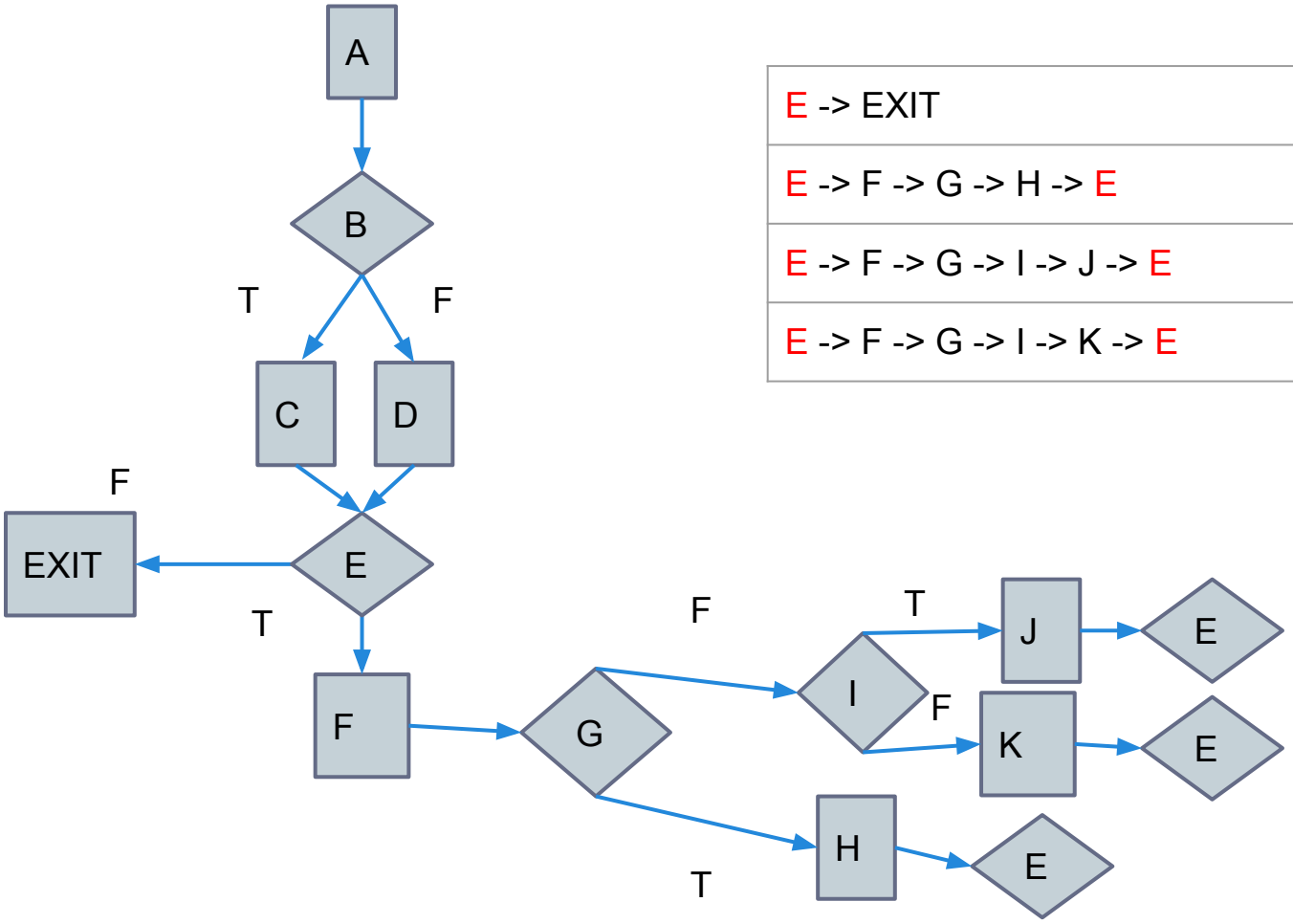

int bott, top, mid;
bott=0; top=size-1;
L = 0;
T[L]
==
key
T
F
found=true;
found=false;
bott<=to
p && !
found
F
EXIT
T
mid=round
(top+bott/2);
T[mid]
== key
T
F
found=true;
L= mid;
T[mid]
< key
T
bott=mid+1;
F
top=mid-1;

CFG



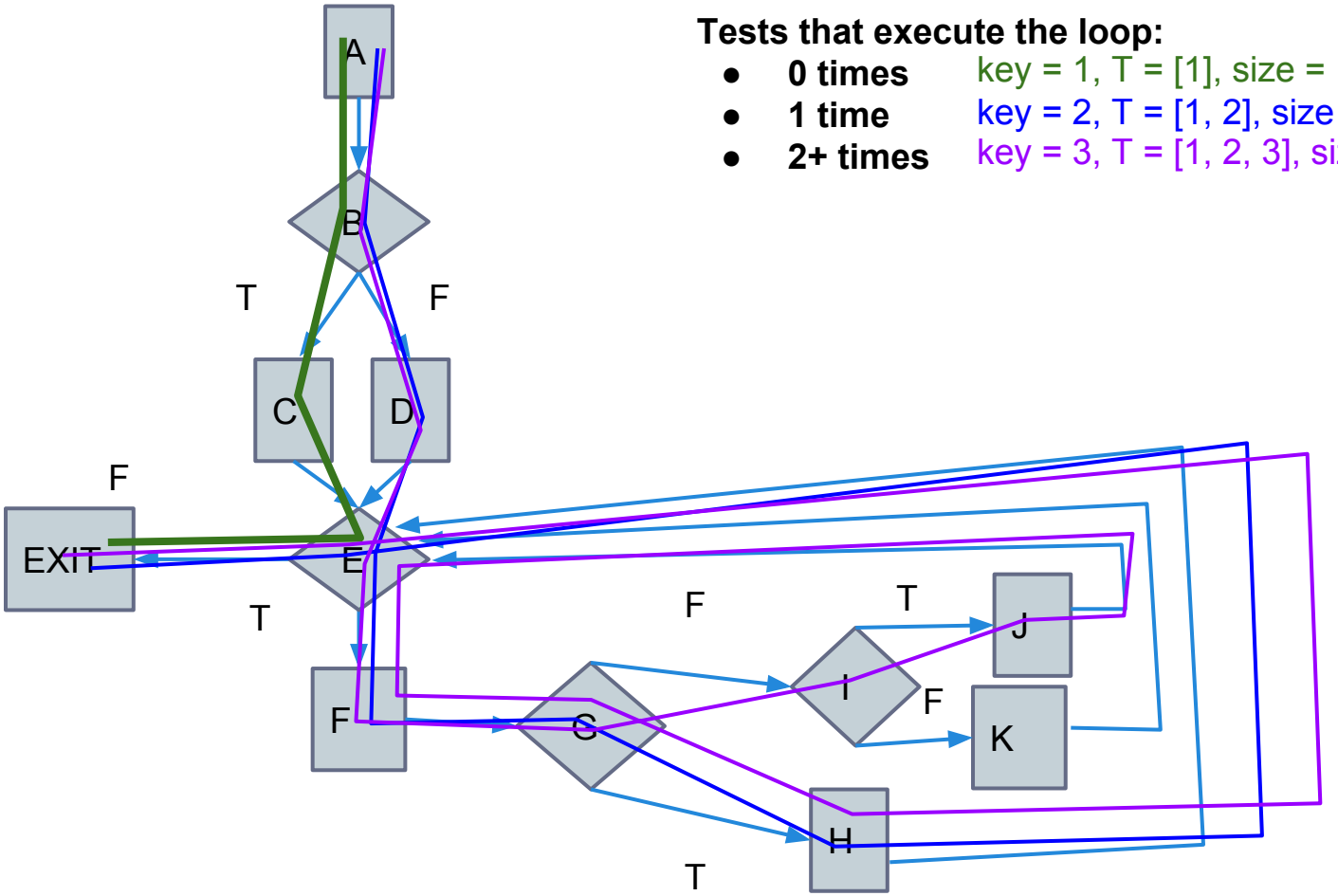
E -> EXIT
E -> F -> G -> H -> E
E -> F -> G -> I -> J -> E
E -> F -> G -> I -> K -> E

CFG



E -> EXIT
E -> F -> G -> H -> E
E -> F -> G -> I -> J -> E
E -> F -> G -> I -> K -> E

CFG



Tests that execute the loop:

- **0 times** key = 1, T = [1], size = 1
- **1 time** key = 2, T = [1, 2], size = 2
- **2+ times** key = 3, T = [1, 2, 3], size = 3

Cyclomatic Testing

- Generally, there are many options for the set of basis subpaths.
- When testing, count the number of independent paths that have already been covered, and add any new subpaths covered by the new test.
  - You can identify allpaths with a set of independent subpaths of size = the cyclomatic complexity.

Uses of Cyclomatic Complexity

- A way to guess “how much testing is enough”.
  - Upper bound on number of tests for branch coverage.
  - Lower bound on number of tests for path coverage.
- Used to refactor code.
  - Components with a complexity > some threshold should be split into smaller modules.
  - Based on the belief that more complex code is more fault-prone.