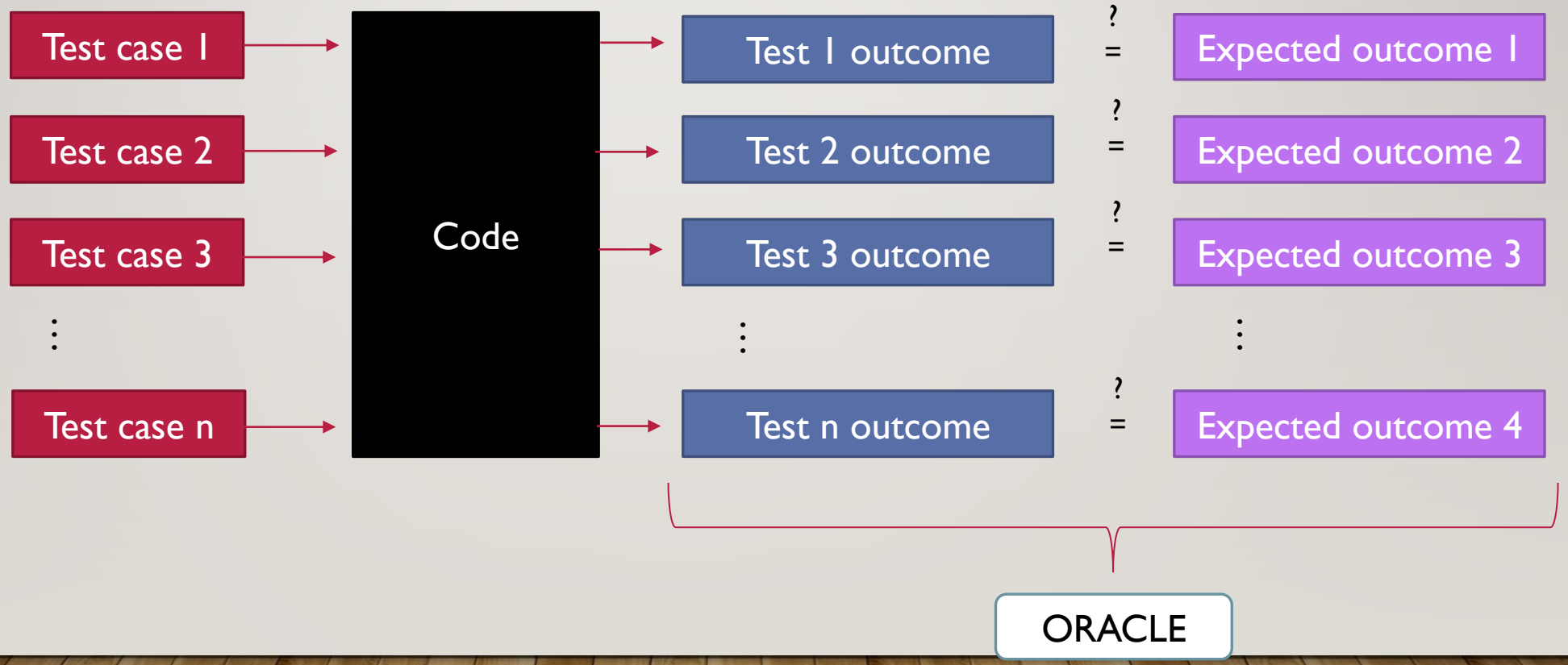
IOT SYSTEM TESTING

WHAT IS SOFTWARE TESTING?

- Determining whether a property is fulfilled by a software

- Test case: a test case is a specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective (Wikipedia)

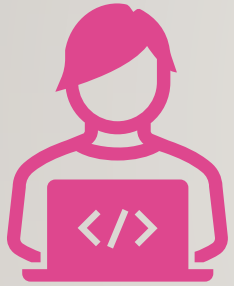
SOFTWARE TESTING



STEPS OF A TEST

- Setup: bringing the software under test to the needed state
- Invocation: running the tetst with the given test data
- Assessment: checking whether the test output is as expected
- Teardown: cleaning up test environment

TESTING



Manual Testing:

Manual providing test cases for verifying correct code functioning

Automatic Testing:

Writing programs that executes the code and inputs test cases to it for verifying correct code functioning.

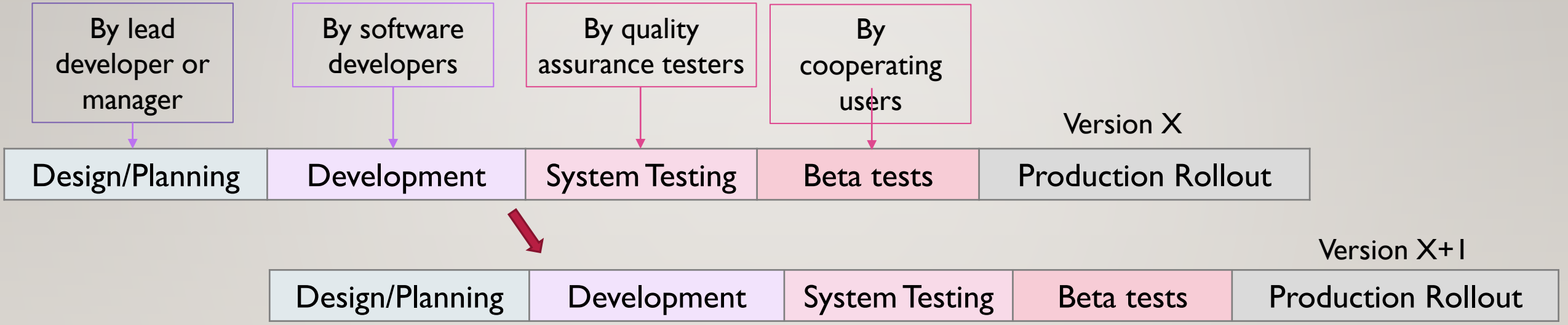
MANUAL TESTING

EXERCISE 1

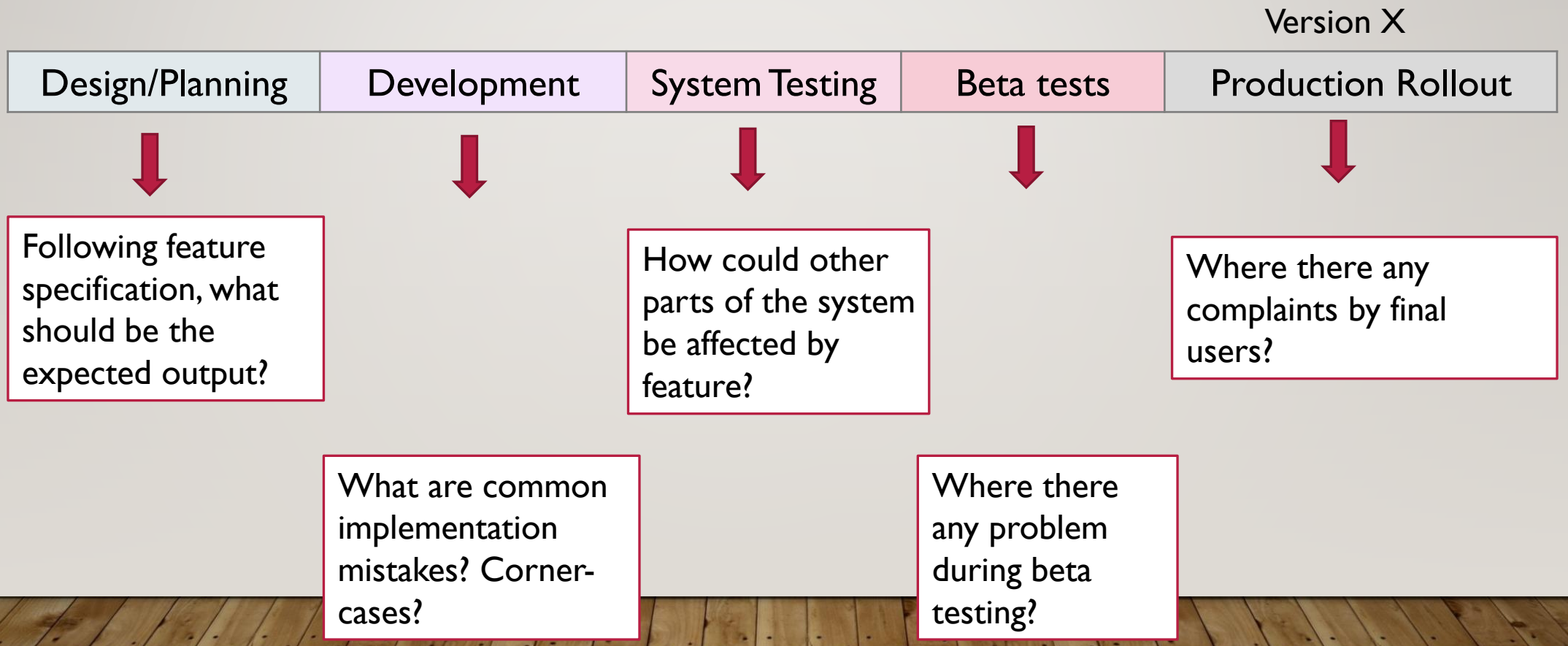
HOW TO SELECT TEST CASES?

- Time is bounded! (You can't input all test cases)
- Popular approaches include:
  - Providing expected user inputs
  - Providing inputs to exceptions by design (branch cond values)
  - Previously reported bug
  - Unexpected values
  - …
  - (any other input sequence you can imagine)
- Is it effective? Can we always be sure all code branches were covered?

SOFTWARE DEVELOPMENT CYCLE IN INDUSTRY



HOW DO TEST CASES ARISE?



TEST CASES SCALABILITY

- Each version release adds test-cases to the pool of test cases to be checked for upcoming versions
- System Testing phase is time-bounded
- Can’t perform manually all test cases in given time

AUTOMATIZED TEST CASE EXECUTION

- Idea:
  - Provide a set of test cases that should be checked
  - Make other computers execute these test cases (behaving as testers/users) and validate the expected output, according to the Oracle.

AUTOMATIZED TEST CASE EXECUTION IN RIOT-OS

EXERCISE 2

LIMITATIONS OF AUTOMATIZED TEST CASES EXECUTION

Since test cases are generated by humans and mostly based on previous knowledge/experience, it is not possible to check for errors that have not yet being discovered.

Definition test-cases takes considerable amount of time – providing the expected output to new test cases can be time-consuming for some features

AUTOMATED TEST CASE GENERATION

- Idea: let the computer generate new test cases, execute and validate them
- How?
  - Generate an input following problem-specific patterns
  - Generate the expected result
- Is it always possible?

AUTOMATED TEST CASE GENERATION

EXERCISE 3

TEST CASE GENERATION

```
int counter = 0;
for(int i = 0; i < 10; i++){
 if(get_button1_state() == 1)
  counter++;
}

if(counter == 10)
 bug();
```

How many test cases do we need to generate to get to „bug”?

What is the percentage of test cases (from the total number of tests cases) that would get to „bug”?

THE COLOR OF THE BOX

Black-box testing

- Test cases are generated without access to source code, and are usually based on software specifications.

White-box testing

- Test cases are generated taking into account source code structure.

EXAMPLE: ANALYZING QUADRATIC EQUATIONS

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Do we need to test **all** combinations of a x b c inputs?
(consider that a float has 2^{32} bits)

```
int analyze_quadratic_expression(float a, float b, float c){
 if(a == 0){
  return info_not_quadratic(b,c);
 }
 else{
  if( b*b > 4*a*c){
   return info_two_real_roots(a,b,c);
  }
  else if(b*b == 4*a*c){
   return info_single_real_root(a,b,c);
  }
  else{
   return info_two_imaginary_roots(a,b,c);
  }
 }
}
```

EXAMPLE: ANALYZING QUADRATIC EQUATIONS

Let's focus on **these branches** only

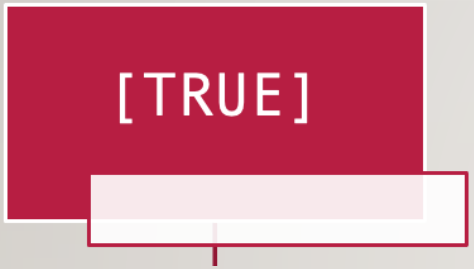
```
int analyze_quadratic_expression(float a, float b, float c){
  if(a == 0){
    return info_not_quadratic(b,c);
  }
  else{
    if(b*b > 4*a*c){
      return info_two_real_roots(a,b,c);
    }
    else if(b*b == 4*a*c){
      return info_single_real_root(a,b,c);
    }
    else{
      return info_two_imaginary_roots(a,b,c);
    }
  }
}
```

EXAMPLE: ANALYZING QUADRATIC EQUATIONS

We only need **4 test cases!**

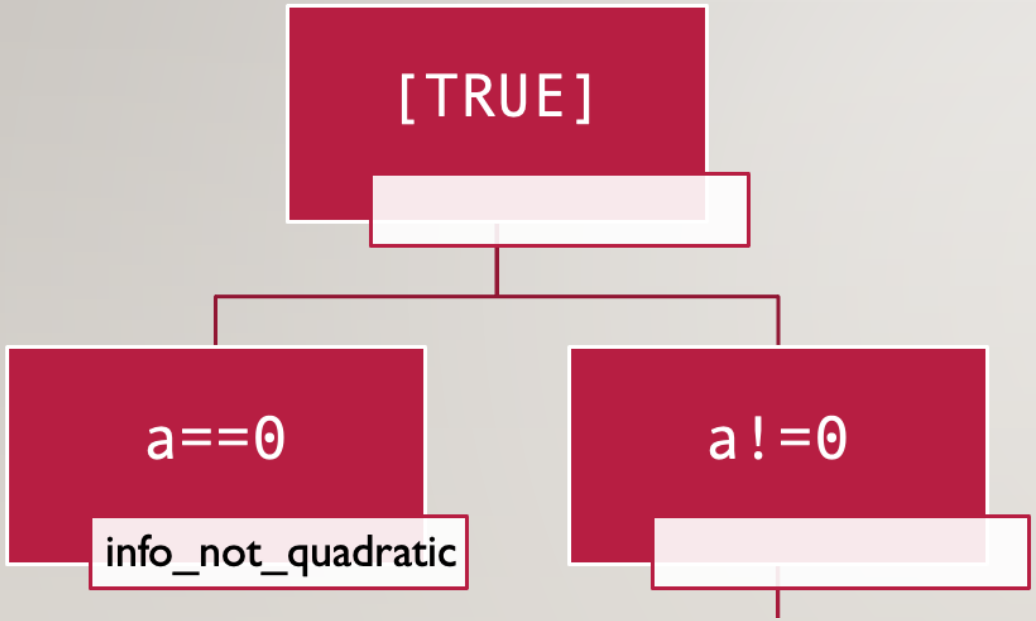
```
int analyze_quadratic_expression(float a, float b, float c){
  if(a == 0){
    return info_not_quadratic(b,c);
  }
  else{
    if(b*b > 4*a*c){
      return info_two_real_roots(a,b,c);
    }
    else if(b*b == 4*a*c){
      return info_single_real_root(a,b,c);
    }
    else{
      return info_two_imaginary_roots(a,b,c);
    }
  }
}
```

SYMBOLIC EXECUTION TREE



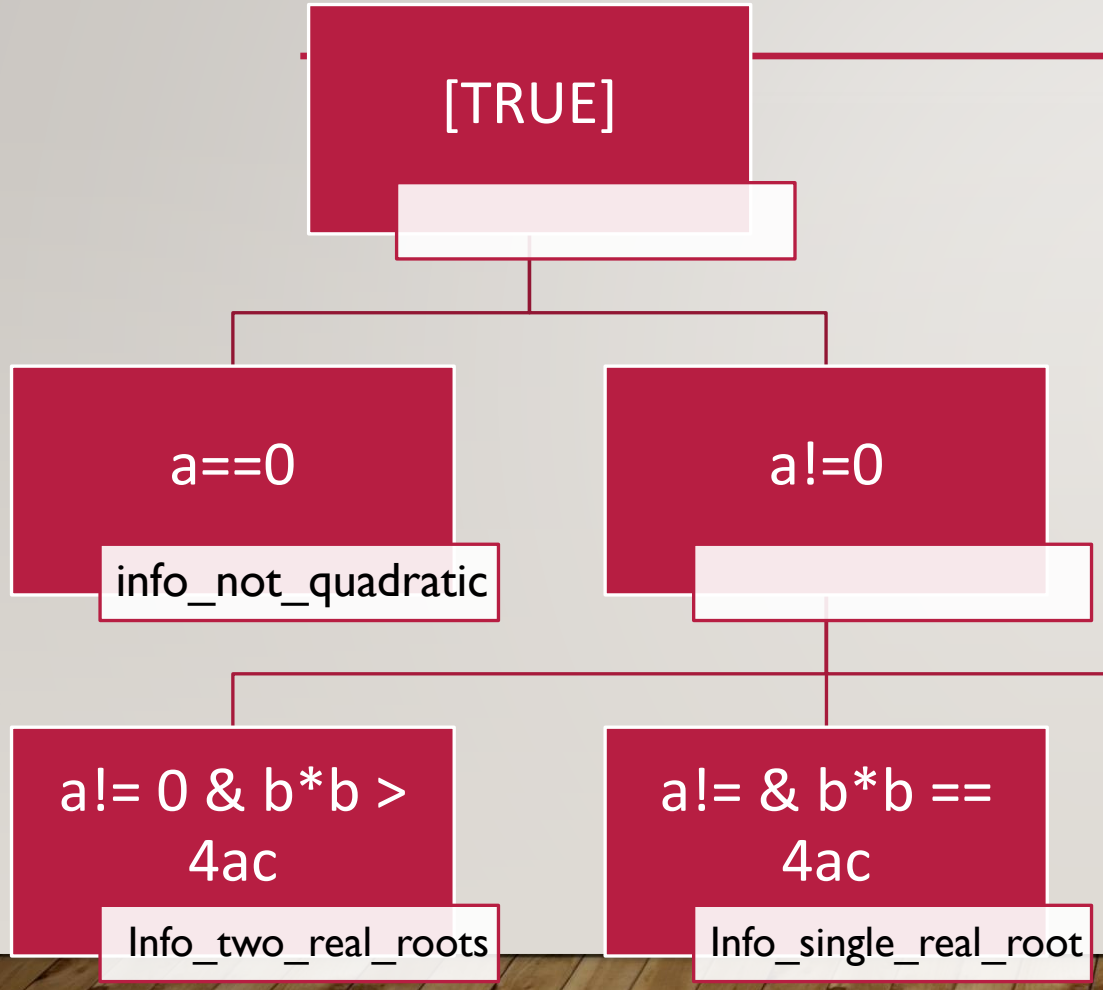

```
int analyze_quadratic_expression(float a, float b, float c){
  if(a == 0){
    return info_not_quadratic(b,c);
  }
  else{
    if(b*b > 4*a*c){
      return info_two_real_roots(a,b,c);
    }
    else if(b*b == 4*a*c){
      return info_single_real_root(a,b,c);
    }
    else{
      return info_two_imaginary_roots(a,b,c);
    }
  }
}
```

SYMBOLIC EXECUTION TREE




```
int analyze_quadratic_expression(float a, float b, float c){
 if(a == 0){
  return info_not_quadratic(b,c);
 }
 else{
  if(b*b > 4*a*c){
   return info_two_real_roots(a,b,c);
  }
  else if(b*b == 4*a*c){
   return info_single_real_root(a,b,c);
  }
  else{
   return info_two_imaginary_roots(a,b,c);
  }
 }
}
```

SYMBOLIC EXECUTION TREE




```
int analyze_quadratic_expression(float a, float b, float c){
 if(a == 0){
  return info_not_quadratic(b,c);
 }
 else{
  if(b*b > 4*a*c){
   return info_two_real_roots(a,b,c);
  }
  else if(b*b == 4*a*c){
   return info_single_real_root(a,b,c);
  }
  else{
                                    ary_roots(a,b,c);
 }
}
```

CONDITIONS AND TEST CASES

TC n.	condition	Expected result
1	a==0	info_not_quadratic(b,c)
2	b*b>4ac, a != 0	info_two_real_roots(a,b,c)
3	b*b==4ac, a != 0	info_single_root(a,b,c)
4	b*b<4ac, a != 0	info_two_imaginary_roots(a,b,c)

```
int analyze_quadratic_expression(float a, float b, float c){
 if(a == 0){
  return info_not_quadratic(b,c);
 }
 else{
  if(b*b > 4*a*c){
   return info_two_real_roots(a,b,c);
  }
  else if(b*b == 4*a*c){
   return info_single_real_root(a,b,c);
  }
  else{
   return info_two_imaginary_roots(a,b,c);
  }
 }
}
```

WHICH TEST CASES?

TC n.	a	b	c	Expected result
1	0	0	0	info_not_quadratic(b,c)
2	1	1	0	info_two_real_roots(a,b,c)
3	1	2	1	info_single_root(a,b,c)
4	1	0	1	info_two_imaginary_roots(a,b,c)

```
int analyze_quadratic_expression(float a, float b, float c){
 if(a == 0){
  return info_not_quadratic(b,c);
 }
 else{
  if(b*b > 4*a*c){
   return info_two_real_roots(a,b,c);
  }
  else if(b*b == 4*a*c){
   return info_single_real_root(a,b,c);
  }
  else{
   return info_two_imaginary_roots(a,b,c);
  }
 }
}
```

STATIC PROGRAM ANALYSIS

STATIC PROGRAM ANALYSIS - DEFINITION

- static program analysis (or static analysis) is the analysis of computer programs performed **without executing them**, in contrast with dynamic program analysis, which is performed on programs during their execution (Wikipedia)

IS THERE AN ERROR?

```
char* read_password(void){
 int counter = 0;
 char input_char = 0;
 int input_password[6] = {0,0,0,0,0,0};
 while(input_char != '#'){
  sleep_until_key_pressed();
  input_char = get_input_char();
  input_password[counter] = input_char;
  counter++;
 }
 return input_password;
}
```

METHODS USED BY STATIC PROGRAM ANALYSIS

- Abstract interpretation: 'executes' the software based on the mathematical properties of each statement and declaration
- Data-flow analysis: to calculate the possible set of values at various points in a computer program.
- Hoare logic: logical rules for reasoning rigorously about the correctness of computer programs
- Model checking: model checking or property checking is a method for checking whether a finite-state model of a system meets a given specification

STATIC ANALYSIS TOOLS FOR C CODE

EXERCISE 4