This lecture talks about testing software.

Testing software is actually very difficult. Many reasons make testing software hard. One of the reasons is if conditions.

Whenever there is a if statement, the program is divided into two paths.

One path is when the condition is true; the other path is when the condition is false.

How often do programs have if conditions?

Statistically, approximately 10% of code has conditions such as if, while, or for.

Now, consider a program with five thousand lines. This program will have approximately 500 branches.

These 500 branches will have two to the power of 500 different paths, if every condition is independent.

As of June 2020, the fastest computer can run 416 peta flops. That is 416 times 10 to the power of 15 operations per second.

If each execution path takes only one operation, this fastest computer can check 10 to the power of 17 execution paths per second.

Checking two to the power of 500 execution path will take approximately ten to the power of 132 seconds.

Scientists estimated that the universe is about 13.8 billion years old. That’s approximately ten to the power of 18 seconds.

Thus, using the fastest computer on earth, it will take much longer than the age of the universe to check all possible execution paths.

Fortunately, in most cases, we do not have to check all possible execution paths.

Often, the execution paths are not completely independent. The number of possible execution paths is actually much smaller than two to the power of the number of if conditions.

For example, if an error condition is true, the program exits. We have seen this many times. A program needs several inputs. If insufficient inputs are given, the program returns exit failure. This if condition will not be mixed with the other if conditions.

Even after excluding the if conditions for program exit, there are still many possible execution paths.

We should first focus on the typical execution paths because the program should do what it is supposed to do.

We should also test the expected exceptional cases. For example, a program wants to open a file for reading. The program should check whether F open succeeds or not. Testing should check the program’s behavior when the input file does not exist.

Some students ask why assignments are graded by running test cases. Are there better way to grade assignments? Can we check whether a program is correct or not without running the program?

Unfortunately, this is not possible. Checking a program’s correctness without running the program is mathematically impossible.

The question is whether it is possible to write a computer program X that can check whether a computer program Y is correct or not.

This question is too complex. Let’s ask a simpler question.

Is it possible to write a computer program X that takes two inputs: a computer program Y and an input test case T. .

The program X answers yes if program Y will eventually stop and no if program Y enters an infinite loop for the test case T. .

This is called the halting problem. Program X needs to answer question program Y will halt or not.

The answer is no. It is not possible writing this program X. .

Please notice that this program X is much simpler than checking the correctness of program Y because program X needs to answer only a simple question: whether the program Y will stop or not.

Why is the halting problem unsolvable? This slide provides a conceptual explanation. The formal proof is more complex.

First, we assume that such a program X is possible.

Next, we are going to create a slightly different program called X prime. X prime is a slight extension of X. .

If X says yes, then X primes enters an infinite loop. If X says no, then X primes says yes.

These two changes are easy and definitely doable.

Since the input of X is a program and a test case, the input of X prime is also a program and a test case.

If we use X prime itself as the input, what will happen?

If X primes stops, then X will say yes.

If X says yes, X primes will enter an infinite loop.

If X primes does not stop, then X will say no. If X says no, X primes says yes.

We reach the situation of self-contradictory. If X prime says itself stops, it enters an infinite loop and does not stop. If X prime does not stop, it will say it stops.

This contradiction occurs because we have made a wrong assumption. Our assumption is that we can create the program X. .

The contradiction occurs because the assumption is wrong.

Thus, we cannot create this program X. .

Next, let’s introduce a concept called continuous integration.

Continuous integration triggers testing whenever you do git push.

Many websites provide the service of continuous integration. Travis Cee eye is one example.

Setting up Travis Cee eye is very simple. You need to add a special file called dot travis dot Y. M. L..

This special file specifies how to run tests.

In this example, the file specifies the language as cee.

Testing the program does not require root access and sets sudo to false.

To run the test, simply execute make test.

You have seen make file many times. This is an example of a hierarchical make file. It will enter three directories called H. W. one, H. W. two, and H. W. four. And run the make file inside each of the directory.

We have seen this type of make file before. By adding minus cee after make, this make file can enter the directories and execute the make files inside the directories.

If you set up continuous integration, your program will be tested every time you git push.

Obviously, you need to write the test cases in advance.

In fact, writing tests before writing code is very common.

This is called test driven development.

Software developers think about how to test different parts of code before they write code.

Consider a program that needs four steps to process data. These four steps are expressed as features one, two, three, and four.

It is very common when multiple people develop different features simultaneously. Imagine that you are responsible creating feature two. You need to work on feature two, while another team works on feature one. You cannot wait for the team for feature one. In this case, you have to write your own test generator creating the data that should come from feature one. You also need to write additional code to check whether feature two produces correct results because feature three is not yet available checking your result.