12th LogicComputation_Week 3_Lesson 1: The Limits of Computation Name _____
key 1 Per. _____ Date _____

Lesson Overview

In this lesson and the next, we start to answer perhaps the important question of computer science: what are computers incapable of? What are the limits of computation?

Can we come to the conclusion that a computer will *never* be able to do X?

Can we conclude that X will be impossible no matter how technology changes, no matter how fast processors become, no matter how much memory they have, no matter how clever we become at programming them, no matter how much artificial intelligence improves?

What could X be, if anything? How can we know for certain what future technology won't be able to do?

These intriguing questions will begin to be answered as we look at a specific issue that came about early on in computer science: The Halting Problem.

Infinite Looping vs Halting

Let's start with a basic truism. *Some programs stop, and other programs do not*.

In other words, some programs start and eventually stop. The programs you have written largely fall into this category. Other programs will continue indefinitely until the power is switched off. An example of this would be your operating system.

Other examples might include some of your past programs that have an error in them. Some of these accidentally ran forever, at least until you reset the virtual machine.

For simplicity, let's replace "program" by "method." The following method eventually stops.

```
public void doSomething(String p){
  int i = 0;
  while(i < 100){
    System.out.println(p);
    i++;
  }
}
```

1. Explain clearly: Why does the previous method eventually halt? (1 pt)

The following method does not halt:

```
public void doSomethingElse(String p){
  int i = 0;
  while(i < 100){
    System.out.println(p);
  }
}
```

2. Explain clearly: Why doesn't the previous method halt? (1 pt)

The following method stops, but it takes a while. (Let's ignore the built-in limits on the size of integers.)

```
public void doSomethingDifferent(String p){
  int i = 0;
  while(i < 10000000000000000000000000000000000000000){
    System.out.println(p);
    i++;
  }
}
```

Humans can recognize that this method will halt. Can a machine recognize that the method will halt?

That's the big question we are focused on in today's lesson. The question is called **The Halting Problem**. Here's the question in detail:

> **The Halting Problem:**
> Is it possible for a program to determine whether any *other* program will eventually finish running or will continue running forever?

So this "halt checking program" we are imagining will need to be able to take two inputs: the first input will be a text representation of a Java method and the second input will be the inputs that are passed in to the method we are checking.

"A method that takes another method as an input" is a very strange thing to think about. How would you even do that?

The haltCheck() method might take a String M that represents the method's code and another String p that represents the input/parameter of the method.

For instance:

String M = “public void doSomethingDifferent(String p){/n/tint i = 0;/n/twhile(i < 10000000000000000000000000000000000000000){/n/t/tSystem.out.println(p);/n/t/ti++;/n/t}/n}”;

The /n and /t symbols conventionally stand for newline and tab. So it looks like a mess, but this perfectly captures everything we had in the last method.

In this example, the method's parameter could really be anything because it just prints that to the screen. So we could have String p = "Hello". And it will print "Hello" to the screen a very large number of times.

The idea of a program taking another program as an input should also be somewhat familiar to you...that's what we talked about last week. Alan Turing's idea was that a Turing Machine along with its input could be written onto the tape of another Turing Machine.

Programs can be the input of another program. This isn't just some hypothetical thought experiment either. It happens all the time on your computer.

Isn't that what BlueJ was doing all along when it ran the programs you typed in?

And, at a much more basic level, your operating system (MacOS, Windows 10, iOS, Android, whatever) is just a program that is running programs that other software developers have written.

Okay, with that out of the way, let's get back to the Halting Problem. What are some strategies for writing a method haltCheck()?

The easiest strategy of just running the input method, starting a timer, and returning "false" after a few hours have passed won't work in general. The last example with the large while loop **does** halt, but not in any reasonable amount of time.

We should instead try to write a method that observes something about the structure of the program itself.

To check if a while loop will finish, we could look at the loop condition (i < 100) and handle different possibilities for each of the comparison symbols <, >, ==, !=, >=, and <= that might be in the method string.

We would also need to know about the initial value of the variable and whether the variable is being incremented, decremented, or not changed in the loop itself.

With all of this information, it is possible to solve the Halting Problem for very simple while loops.

If the haltCheck() method were designed to notice the symbol "<" in the loop condition and the "++" in the loop body after the variable that controls the loop, it could return "true" immediately. That's more or less what you were doing when you were checking the method at a glance. Again, this strategy would only work for very simple while loops.

3. The three methods we were looking at were very easy to analyze because they just involved one while loop and one variable.

Solving the Halting Problem for a more general Java method may be much more difficult. What are some complications you could imagine adding to the method that would make it very difficult to determine whether the method halts? Write a few of your ideas in the space below. (1 pt)

4. If a method does not have any while loops or for loops inside of it, can we conclude that it always halts? Why or why not? (1 pt)

The Halting Problem matters because it would be useful to be able to tell in advance if a program (or method, in our case) was going to run forever. It could be used to generate a warning for software developers who might have a bug in their code.

We'll see in the next lesson that it would have another surprisingly helpful use in addition to that. In fact, a full solution to the halting problem would completely revolutionize our world.

12th LogicComputation_Week 3_Lesson 1: The Limits of Computation **ANSWER KEY 1**
key 1

Lesson Overview

In this lesson and the next, we start to answer perhaps the important question of computer science: what are computers incapable of? What are the limits of computation?

Can we come to the conclusion that a computer will *never* be able to do X?

Can we conclude that X will be impossible no matter how technology changes, no matter how fast processors become, no matter how much memory they have, no matter how clever we become at programming them, no matter how much artificial intelligence improves?

What could X be, if anything? How can we know for certain what future technology won't be able to do?

These intriguing questions will begin to be answered as we look at a specific issue that came about early on in computer science: The Halting Problem.

Infinite Looping vs Halting

Let's start with a basic truism. *Some programs stop, and other programs do not*.

In other words, some programs start and eventually stop. The programs you have written largely fall into this category. Other programs will continue indefinitely until the power is switched off. An example of this would be your operating system.

Other examples might include some of your past programs that have an error in them. Some of these accidentally ran forever, at least until you reset the virtual machine.

For simplicity, let's replace "program" by "method." The following method eventually stops.

```
public void doSomething(String p){
  int i = 0;
  while(i < 100){
    System.out.println(p);
    i++;
  }
}
```

1. Explain clearly: Why does the previous method eventually halt? (1 pt)

   The condition for the while loop to keep running is eventually false because i increments by one each time the loop runs. Eventually i is greater than or equal to 100 so the loop breaks.

The following method does not halt:

```
public void doSomethingElse(String p){
  int i = 0;
  while(i < 100){
    System.out.println(p);
  }
}
```

2. Explain clearly: Why doesn't the previous method halt? (1 pt)

This method does not halt because i is set to 0 and never takes on any other value. The loop condition i < 100 is always true, so the loop never breaks.

The following method stops, but it takes a while. (Let's ignore the built-in limits on the size of integers.)

```
public void doSomethingDifferent(String p){
  int i = 0;
  while(i < 10000000000000000000000000000000000000000){
    System.out.println(p);
    i++;
  }
}
```

Humans can recognize that this method will halt. Can a machine recognize that the method will halt?

That's the big question we are focused on in today's lesson. The question is called **The Halting Problem**. Here's the question in detail:

> **The Halting Problem:**
> Is it possible for a program to determine whether any *other* program will eventually finish running or will continue running forever?

So this "halt checking program" we are imagining will need to be able to take two inputs: the first input will be a text representation of a Java method and the second input will be the inputs that are passed in to the method we are checking.

"A method that takes another method as an input" is a very strange thing to think about. How would you even do that?

The haltCheck() method might take a String M that represents the method's code and another String p that represents the input/parameter of the method.

For instance:

String M = "public void doSomethingDifferent(String p){/n/tint i = 0;/n/twhile(i < 10000000000000000000000000000000000000000){/n/t/tSystem.out.println(p);/n/t/ti++;/n/t}/n}";

The /n and /t symbols conventionally stand for newline and tab. So it looks like a mess, but this perfectly captures everything we had in the last method.

In this example, the method's parameter could really be anything because it just prints that to the screen. So we could have String p = "Hello". And it will print "Hello" to the screen a very large number of times.

The idea of a program taking another program as an input should also be somewhat familiar to you...that's what we talked about last week. Alan Turing's idea was that a Turing Machine along with its input could be written onto the tape of another Turing Machine.

Programs can be the input of another program. This isn't just some hypothetical thought experiment either. It happens all the time on your computer.

Isn't that what BlueJ was doing all along when it ran the programs you typed in?

And, at a much more basic level, your operating system (MacOS, Windows 10, iOS, Android, whatever) is just a program that is running programs that other software developers have written.

Okay, with that out of the way, let's get back to the Halting Problem. What are some strategies for writing a method haltCheck()?

The easiest strategy of just running the input method, starting a timer, and returning "false" after a few hours have passed won't work in general. The last example with the large while loop **does** halt, but not in any reasonable amount of time.

We should instead try to write a method that observes something about the structure of the program itself.

To check if a while loop will finish, we could look at the loop condition (i < 100) and handle different possibilities for each of the comparison symbols <, >, ==, !=, >=, and <= that might be in the method string.

We would also need to know about the initial value of the variable and whether the variable is being incremented, decremented, or not changed in the loop itself.

With all of this information, it is possible to solve the Halting Problem for very simple while loops.

If the haltCheck() method were designed to notice the symbol "<" in the loop condition and the "++" in the loop body after the variable that controls the loop, it could return "true" immediately. That's more or less what you were doing when you were checking the method at a glance. Again, this strategy would only work for very simple while loops.

3. The three methods we were looking at were very easy to analyze because they just involved one while loop and one variable.

Solving the Halting Problem for a more general Java method may be much more difficult. What are some complications you could imagine adding to the method that would make it very difficult to determine whether the method halts? Write a few of your ideas in the space below. (1 pt)

4. If a method does not have any while loops or for loops inside of it, can we conclude that it always halts? Why or why not? (1 pt)

The Halting Problem matters because it would be useful to be able to tell in advance if a program (or method, in our case) was going to run forever. It could be used to generate a warning for software developers who might have a bug in their code.

We'll see in the next lesson that it would have another surprisingly helpful use in addition to that. In fact, a full solution to the halting problem would completely revolutionize our world.