12th LogicComputation_Week 2_Lesson 4: The Church-Turing Thesis2 key 1

Name ______________________________

Per. ______ Date ________________

Lesson Overview:

This is a short lesson today, but it covers the 'fundamental theorem' of computer science, so it's pretty significant, to say the least!

Fundamental Theorems are funny things. Many different branches of mathematics have them: you already know of the one from Calculus that talks about the relationship between derivatives and integrals, but there's also a Fundamental Theorem of Arithmetic (about primes) and a Fundamental Theorem of Algebra (about roots of a polynomial).

These Fundamental Theorems are the most prominent landmarks in each field. They have a way of capturing the essence of what a subject is all about.

Computer Science has one too. Admittedly, it's not really called "The Fundamental Theorem of Computer Science," it's called the **Church-Turing Thesis**, named after Alan Turing and his PhD advisor and mentor, Alonzo Church. We will get to know it in this lesson.

And here it is:

The Church-Turing Thesis: Any computation that can be carried out by a physical or mechanical process can be translated into an equivalent computation involving a Turing Machine.

Take a step back and think about this. This essentially means that any algorithm can be modeled with a Turing Machine. This also means, ignoring considerations of time and memory limits, that a computer can simulate any physical, mechanical process. *This* (together with the fact that it is a UTM) is why the computer is so successful. *This* is why the possibilities for what a computer can do are endless.

This also means that, while there are improvements to performance that can be made (and are constantly being made) to computers, their nature in terms of what they can and cannot do is fixed. There is no computational advantage (again, ignoring speed and space) to making, say, a trinary computer instead of a binary computer. In fact, there is no computational advantage to having:

1. Multiple read/write heads
2. Multiple tapes
3. Multidimensional tapes

On the other hand, we could also simplify and restrict the TM without affecting it’s computational power:

4. A one-way infinite tape (you wouldn’t need an infinite tape in both directions)
5. Only a binary alphabet
6. Non-erasing: the TM can never rewrite a symbol once it is written on the tape.

The last thing this means that the study of what a computer can and cannot do can be reduced to the study of Turing Machines.

General purpose computers

General purpose computers are the physical instantiation of the universal Turing machine: they are capable of running different algorithms without requiring any hardware modifications. This is possible because modern day microprocessors are based upon the *von Neumann architecture,* which we will learn more about in later lessons. In this model, computer programs and data are stored in the same main memory. This means that the contents of memory can be treated as a machine instruction or data, depending on the context. This is exactly analogous to the tape contents of the UTM which consists of both a program (the original TM) and its data (the tape contents of the original TM).

Alan Turing's work on UTM anticipated the development of general purpose computers, and can be viewed as the invention of software!

The fact that you are reading this text (data) on a device that can run an operating system and web browser (software) is a direct consequence of the general purpose computer concept. Your machine (laptop/smart phone/whatever) was not engineered with the foresight that you would be engaged in this exact task, yet it has the power to do this and many other things besides.

And as impressive as your machine would undoubtedly be to Alan Turing, he could still say with certainty that even though it is faster and more efficient than anything in his era, it can't do anything in principle that a primitive tape-reading Turing Machine could do.

That statement doesn't hold just from 1930 to 2020. Any machine that will ever be built or could ever be built will not be able to perform any computation that a Turing Machine cannot do. That is the Church-Turing Thesis.

I hope you have some appreciation for how significant and astonishing this statement is:

No computer, no as-yet unheard-of technology will ever be able to do any computation that a Turing Machine cannot do.

Alan Turing deservedly gets much of the spotlight for his pioneering work on general computers, but there are others who deserve joint credit for this idea that has shaped so much of our world today.

First, there's the American mathematician Alonzo Church who actually developed his own mathematical model of computation before Turing. His model was called the "lambda calculus" and took a bit more time to catch on, but it's extraordinarily elegant and is a major element of computer science today. The lambda calculus is essentially a way of describing all computations in terms of functions like f(x) or g(x, y, z) but with a different and more general syntax. He envisioned a mathematics created entirely out of nested functions, where even numbers and values like true and false could be defined as functions.

It was part of the collaboration of Church and Turing that they realized that Turing Machines are exactly equivalent to functions written using Church's lambda calculus. So an equally valid way of writing the Church-Turing thesis is:

The Church-Turing Thesis: Any computation that can be carried out by a physical or mechanical process can be translated into an equivalent computation involving a lambda expression

The other pioneers of computer science were Charles Babbage and Ada Lovelace, as we have discussed in a previous lesson.

Turing, Church, Babbage, and Lovelace are visionaries that saw much further into the possibilities of computing than anyone else in their age. Together they shared a vision of a reprogrammable computing machine that would revolutionize our world.

12th LogicComputation_Week 2_Lesson 4: The Church-Turing Thesis2 **ANSWER KEY 1**
key 1

Lesson Overview:

This is a short lesson today, but it covers the 'fundamental theorem' of computer science, so it's pretty significant, to say the least!

Fundamental Theorems are funny things. Many different branches of mathematics have them: you already know of the one from Calculus that talks about the relationship between derivatives and integrals, but there's also a Fundamental Theorem of Arithmetic (about primes) and a Fundamental Theorem of Algebra (about roots of a polynomial).

These Fundamental Theorems are the most prominent landmarks in each field. They have a way of capturing the essence of what a subject is all about.

Computer Science has one too. Admittedly, it's not really called "The Fundamental Theorem of Computer Science," it's called the **Church-Turing Thesis**, named after Alan Turing and his PhD advisor and mentor, Alonzo Church. We will get to know it in this lesson.

And here it is:

The Church-Turing Thesis: Any computation that can be carried out by a physical or mechanical process can be translated into an equivalent computation involving a Turing Machine.

Take a step back and think about this. This essentially means that any algorithm can be modeled with a Turing Machine. This also means, ignoring considerations of time and memory limits, that a computer can simulate any physical, mechanical process. *This* (together with the fact that it is a UTM) is why the computer is so successful. *This* is why the possibilities for what a computer can do are endless.

This also means that, while there are improvements to performance that can be made (and are constantly being made) to computers, their nature in terms of what they can and cannot do is fixed. There is no computational advantage (again, ignoring speed and space) to making, say, a trinary computer instead of a binary computer. In fact, there is no computational advantage to having:

1. Multiple read/write heads
2. Multiple tapes
3. Multidimensional tapes

On the other hand, we could also simplify and restrict the TM without affecting it’s computational power:

4. A one-way infinite tape (you wouldn’t need an infinite tape in both directions)
5. Only a binary alphabet
6. Non-erasing: the TM can never rewrite a symbol once it is written on the tape.

The last thing this means that the study of what a computer can and cannot do can be reduced to the study of Turing Machines.

General purpose computers

General purpose computers are the physical instantiation of the universal Turing machine: they are capable of running different algorithms without requiring any hardware modifications. This is possible because modern day microprocessors are based upon the *von Neumann architecture,* which we will learn more about in later lessons. In this model, computer programs and data are stored in the same main memory. This means that the contents of memory can be treated as a machine instruction or data, depending on the context. This is exactly analogous to the tape contents of the UTM which consists of both a program (the original TM) and its data (the tape contents of the original TM).

Alan Turing's work on UTM anticipated the development of general purpose computers, and can be viewed as the invention of software!

The fact that you are reading this text (data) on a device that can run an operating system and web browser (software) is a direct consequence of the general purpose computer concept. Your machine (laptop/smart phone/whatever) was not engineered with the foresight that you would be engaged in this exact task, yet it has the power to do this and many other things besides.

And as impressive as your machine would undoubtedly be to Alan Turing, he could still say with certainty that even though it is faster and more efficient than anything in his era, it can't do anything in principle that a primitive tape-reading Turing Machine could do.

That statement doesn't hold just from 1930 to 2020. Any machine that will ever be built or could ever be built will not be able to perform any computation that a Turing Machine cannot do. That is the Church-Turing Thesis.

I hope you have some appreciation for how significant and astonishing this statement is:

No computer, no as-yet unheard-of technology will ever be able to do any computation that a Turing Machine cannot do.

Alan Turing deservedly gets much of the spotlight for his pioneering work on general computers, but there are others who deserve joint credit for this idea that has shaped so much of our world today.

First, there's the American mathematician Alonzo Church who actually developed his own mathematical model of computation before Turing. His model was called the "lambda calculus" and took a bit more time to catch on, but it's extraordinarily elegant and is a major element of computer science today. The lambda calculus is essentially a way of describing all computations in terms of functions like f(x) or g(x, y, z) but with a different and more general syntax. He envisioned a mathematics created entirely out of nested functions, where even numbers and values like true and false could be defined as functions.

It was part of the collaboration of Church and Turing that they realized that Turing Machines are exactly equivalent to functions written using Church's lambda calculus. So an equally valid way of writing the Church-Turing thesis is:

The Church-Turing Thesis: Any computation that can be carried out by a physical or mechanical process can be translated into an equivalent computation involving a lambda expression

The other pioneers of computer science were Charles Babbage and Ada Lovelace, as we have discussed in a previous lesson.

Turing, Church, Babbage, and Lovelace are visionaries that saw much further into the possibilities of computing than anyone else in their age. Together they shared a vision of a reprogrammable computing machine that would revolutionize our world.