An Overview

This course is different from all the other courses that you have taken, or will take, as a Computer Science major.

In most of the CS courses, we talk about how to make a computer usable, and useful, so that we can use it to solve our problems.

Most recently, in the Algorithms course, besides discussing various algorithms, we also discussed how difficult, or easy, it is to solve a problem. For example, to sort a list of $n$ pieces of data, we have to make $\Omega(n)$ key comparisons.

Our goal there is to find a better way, faster and/or smaller, to solve our problems.
A different scope

In this course, besides what problems a computer can solve, we will look at the other side of the coin, to study problems that no computer can solve, no matter how powerful it is, and what we do.

In other words, we now turn to another, more challenging, question: given a problem, whether or not a computer can solve it at all.

More generally, What can a computer do and can’t do?

If it can, how? If it can’t, why?
An example

To have a scoop of ice cream, you can walk to Ice Cream Parlor, downtown Plymouth, in a few minutes.

If you want to have a good one, you can walk to Kellerhaus, Weirs Beach. You have to walk 19 plus miles, thus about five hours at the speed of four miles per hour.

It takes a while, but still doable.

On the other hand, there is no way that you can walk all the way to the Moon, no matter how fast you walk, and what pair of shoes you wear... because of the gravity.
Other examples

There is a difference between what we could not do and what we can’t do right now, but maybe we can down the road.

For example, although it takes a baby three days to recognize her parents’s face, especially that of her mother, it used to drive a computer nuts.

But, with some significant progress in both hardware and software, the success rate of 98% matching in face recognition via a Mastercard pilot study has been reported by Computer-Weekly last year.
MST and Steiner trees

In CS3221, we learned that it takes polynomial time, $O(|E| \log |V|)$ to be exact, to construct a minimum spanning tree in $G(V,E)$.

A related problem is the minimum Steiner tree problem, which is also to optimally interconnect terminals, but you might use extra points, called Steiner points.

The minimum Steiner tree problem finds important applications in chip design.
Getting better and better

The general problem is NP-Complete, but practical and exact algorithms have been found and progress have been made over time.

Back in the 1970’s, we can’t even work with more than ten terminals. In 1985, we can construct a minimum Steiner tree for more than twenty terminals. In early 1990’s, further improved algorithms can handle fifty terminals.

At present time, it takes about an hour to construct a minimum Steiner tree for one thousand terminals, which can cut the total length of the wires used by 20%.

**Question:** Now could we know something is not doable?

**Answer:** The only way is to give a proof.
A different method

In all the computer science courses that we have been taking, mathematics is a very important component. Particularly, we have to use mathematics to construct computing formulas, and, in some cases, to figure out the logic behind the algorithms.

We started to see the value of mathematical proofs in showing the correctness of algorithms in the analysis course.

In this course, to address the ability issue, we have to show, beyond any doubt, something is true or false.

Thus, we have to make a heavy use of argumentative mathematics, or proofs. This calls for a different set of mathematical skills, and, frankly, is quite challenging.
A different concern

In the algorithm/programming courses, we focus on time, thus the most important hardware piece is the processor: the faster the processor is, the less time it will take an algorithm to finish.

When discussing what a computer can and cannot do, timing is no longer the issue, as long as it does something, we don’t care how much time it is going to take.

On the other hand, as we will see, the amount of memory, and its structure, directly determines the capability of a computer.

**Question:** Which one?

**Answer:** Four of them....
A different time frame

As a practical concern, the chapters in this course, particularly, the earlier ones, are rather long. For example, the next one lasts about ninety pages, and will take about three weeks to finish.

It will also be a challenge to finish all the homework in a timely manner, and hand it in at the end of a chapter.

You get to know how to manage your time so that you can get all the works done while the concepts, among many, are still fresh in your head.

In particular, you should not wait till the last weekend to start doing your work.
What will we talk about?

We essentially talk about three things: computability, automata, and formal languages.

*Computability* refers to the ability of a computer, i.e., what a computer can do and cannot do. We will present a collection of problems that a computer can solve and cannot solve.

There are so many computers in our hand, which computer are we talking about?

*Automata theory* is to define exactly what do we mean by a computer in the above context.

To unify all those problems that we are interested, we discuss what *languages* an automata can or cannot accept or generate.
Computability

In 1931, Kurt Gödel demonstrated that within any given branch of mathematics, there would always be some propositions that couldn’t be proved either true or false using the rules and axioms ... of that mathematical branch itself.

This result is then used to conclude that: some problems are unsolvable by any “computers”, since any computer only knows a finite number of rules and axioms because of its finite nature.

An often used example of this nature is Halting Problem, i.e., there does not exist a program, $H$, that, given any program, $P$, and input, $i$, $H$ can decide weather $P$ will get into an infinite loop with $i$. 
A tough job

It is already difficult enough to show that something is solvable: we have to find a solution. If it were easy, there would not need any homework, or exam.

It is much more difficult to prove that something is unsolvable or not computable. Could we conclude that something is unsolvable, if we could not come up with a solution in 2 hours, or 10 days?

We could not, otherwise no instructor will be able to fail you in a course.

Maybe we have to work harder, or we are simply not smart enough.
Similarly, if we send in a program to calculate a function, but the computer does not come back with an answer in 10 days. Could we conclude this function is not calculable? Should we wait for some more time, say 150 years, or is it really not computable at all?

Maybe we should come up with another program, and/or try it in another computer.

Thus, a better question is that what do we mean by saying that a problem is not “computable” by a “computer”?

If we can provide a precise definition of a computer, and prove a problem is NOT solvable by a specific computer, we can conclude that it is unsolvable by that computer.
Automata and their uses

To define exactly what is a “computer”, we will start with various mathematic models of computers, referred to as automata, in terms of its capability, from simpler ones (regular machines) to more complicated ones, four in all, and culminating at a precise model of computation, or algorithm, the Turing Machine (1936), which you should have heard of in CS2010 Fundamental.

In fact, we will show that TM is more powerful than any real computer since it has infinite amount of memory (processor vs. memory).

Notice that in this course, we only care about the capability of a machine but not even a bit on its efficiency.

The big deal is that, if a problem is not computable in TM, then we can conclude that problem is not computable, no matter what computer is used.
Formal languages

A human being is (much) more capable as compared with a monkey, partly because the language that we speak is much more sophisticated.

Thus, a good way to characterize the capability of an automaton is to specify the (formal) language it understands. What we will do in this part is to study what is exactly the language that different classes, four of them, of automata can generate and/or accept.

This might be the more practical part of the theoretical computer science, as it finds wide application in computer engineering, such as compiler construction, hardware design, and artificial intelligence.
A summary

We will define various automata based on the size and structure of their memory, then discuss the equivalence between these automata and different kinds of languages. This is to characterize exactly what “computers” we are talking about.

We will then discuss exactly what a specific computer/automaton can do and cannot do in terms of its languages.

Because of the precise nature of such subjects, we will start with a (p)review of some of the mathematical concepts.