An Overview

At the end of *CS 3221 Algorithm Analysis*, that we went through last fall, we looked at the following table, summarizing all the problems that we have to deal with in this business. ☺

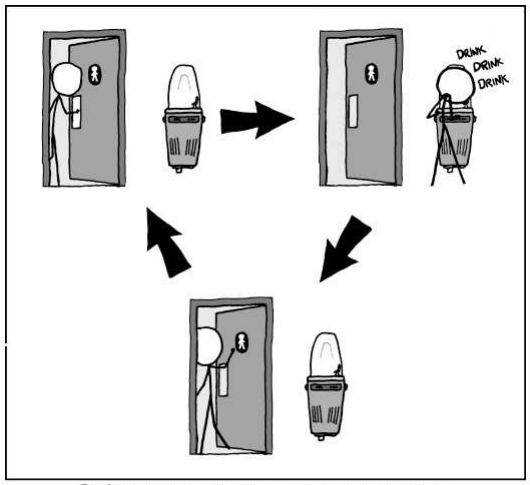
Category	**Time**	**Examples**
Solvable (*CS 2010* $\rightarrow$ *CS 3221*)		
Easy	$O(p(n))$	Sorting, DFS, MST, ...
Uncertain	?	TSP, NP Complete ...
Hard	$\Omega(2^n)$	Hanoi Tower problem
Unsolvable (*CS 3780*)		
	N/A	Halting problem Inc.

We have looked at many "easy" problems in *CS 3221 Algorithm Analysis*, for which we have found out an algorithm with a polynomial, $p(n)$, price tag, and we also briefly talked about those Uncertain and Hard problems.

In this course, we will flip the coin and study *all these Unsolvable problems that no computer is able to solve*, such as the Halting problem. ☹

Halting problem

Can we come up with a program, `H`, that will take two inputs, a program `P` and an input `i`, and decide if, with this input `i`, the program `P` will stop or get into an infinite loop?



I AVOID DRINKING FOUNTAINS OUTSIDE BATHROOMS BECAUSE I'M AFRAID OF GETTING TRAPPED IN A LOOP.

Question: Can you (`H`) tell if he (`P`) will be done with the water (`i`) next time, or keep on going through the door? ☹

On the other side, we will also show that something that looks tough can actually be done ☺.

Language translation

We have to write programs in a programming language, ☺

```
a=b+c;
```

But a computer could only understand something like this. ☹

```
0000 000000000101
0011 000000000110
0001 000000000111
```

Question: How could a computer tell a program is grammatically correct in, e.g., `Java`; and, *if it is*, then translate it into a machine language program that the computer can execute?

Answer: A compiler does all these and more.

Aho and Ullman got a million bucks for this work. Check out the course page on *ACM 2020 Turing award*.

It is different...

As we mentioned, during the last class of *Algorithm Analysis*, to solve problems, we have to understand *what problem* that we are trying to solve? *Can we solve it?* If we can, *how* do we do it, and *how long* does it take? If we can't, *why?*

In most of the CS courses, we talk about *how* to use a computer to solve problems. Our goal there is to find a "better" way, faster and/or smaller, to solve those problems.

Here, in this course, besides cracking more problems, some of them seemingly not as useful, we will also look at the other side of the coin, by studying problems that no computer can solve, no mater how powerful it is, and what we do. ☹

More generally, *What can a computer do, can't do, and 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 fundamental 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 later 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, especially AI ad ML, it has become a matured technology and used everywhere. ☺

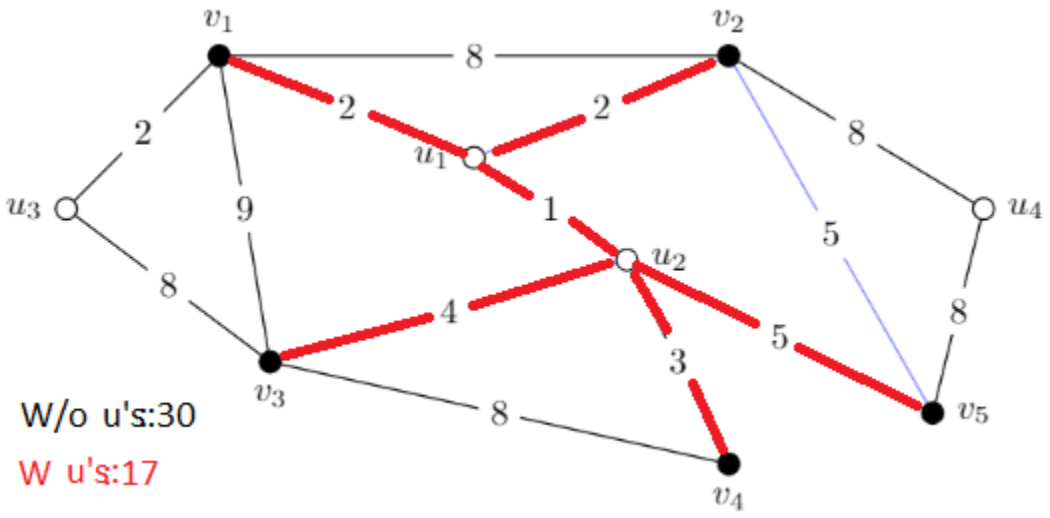
Applications include one in the Orlando International Airport, where FR technology is used to verify the faces of all the passengers.

Check out the course page.

MST and Steiner trees

In *Algorithm analysis*, 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, u_1, u_2, u_3, and u_4, 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 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: How could we know something is not doable?

Answer: The only way is to give a proof. ☹

Messy math or mathy mess?

Mathematics is behind Computer Science, and plays an important role in almost all the computer science courses. For example, we have to use mathematics to construct computing formulas, and figure out the programming logic behind algorithms.

We saw the value of mathematics in proving the correctness of some of the searching and sorting algorithms, and characterizing their running time, in *CS 3221 Algorithm Analysis*.

In this course, to address the ability issue, we have to show that, *beyond any doubt,* not just a reasonable amount as expected of a lawyer ☺, something is true or false. Thus, we have to make a heavy use of argumentative mathematics, or proofs.

Such a need calls for a different set of mathematical skills, and, frankly, it 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, as long as it does something, we don't care how much time it is going to take. ☺

We only care about the *capability* of a machine, but not even a bit on its *efficiency.*

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: There are four memory structures.... We will go through all of them later on.

A different time frame for the homework

As a practical concern, the chapters in this course, particularly, the earlier ones, are rather long. For example, the next one on finite automata contains more than 100 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 need to figure out how to manage your time so that you can get all the works done while the concepts, among many, are still fresh, and warm, in your head. ☺

In particular, you should not wait till the last weekend to start doing your work. I guarantee you that... . ☹

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.

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

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

Automata theory is to define exactly what do we mean by a computer in the above context. *Each of the four automata, with a unique memory structure, is associated with exactly one of the four languages.*

Click the **Topics** button on the syllabus.

Computability

In 1931, Kurt Gödel demonstrated that, within any given branch of mathematics, there would always be something (called *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, represented as a text file, and input, i, H can decide weather P will get into an infinite loop with i. (Page 2)

We will prove later in Chapter 5 why this is the case.

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, project, tests, or exam for any of these programming courses. ☺

It is much more challenging 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 need more time, or we have to work harder, or maybe it is just too challenging for me. ☹

What do you mean *exactly*?

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, faster, with more memory, or both?

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.*

This sort of reasoning leads to the definition of an algorithm in terms of *Turing machine*.

Automata and their uses

To define exactly what a "computer" is, we will start with various mathematic models of computers, referred to as *automata,* in terms of its capability.

We will start 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 (TM)* (1936), which you should have heard of in *CS 2010 Computing Fundamentals.*

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

As a result, *if a problem is not computable in TM, it is not computable at all.* ☺ or ☹

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 than what a monkey does.

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.*

Arithmetic evaluation is *regular,* and almost all the programming languages that we have been using are *context-free.*

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 could not do, in terms of its languages.

Beside talking, we will also walk a little with *JFLAP*, an automata simulator to play with various mathematical machines that we will come up with.

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