# CSC320 – Introduction

## Questions

- What problems can a computer solve?
    - Given a computer program, find the bugs or determine if it's buggy.
    - Given a computer program, determine whether it halts.
    - Given a computer program, does it ever access more than 1000 memory cells?
    - Given a computer program, does it ever uses a certain register?
    - Given a computer program, does it have correct Java syntax?
    - Given a computer program, does it ever output "yes"?

- What problems can be solved efficiently (i.e. polynomial time)
    - Can we schedule exams over a two week period so that no student is scheduled to take two finals at the same time? (scheduling)
    - Can we pair up students with co-op jobs so that there is no (student, employer) who are not paired up but would prefer each other? (Stable marriage)
    - In a network, can we route calls from the origins to their destinations so that no two share a common link? (disjoint paths)
    - (Public key cryptography) Can we have a public key cryptography system which is hard to break? Easy to generate keys for?
    - Can we filter out bots by creating a test that only humans can solve quickly?

- What is a problem?
- What is an encoding of a problem?
- What is a computer?
- What can humans compute?(Turing)

- The Turing machine model
    - Does RAM make a difference?
    - Do we need an infinite tape?
    - What if we replace it with a stack? (Context-free grammars/pushdown automaton)
    - Two stacks?
    - What if we get rid of the tape and have finite memory (Regular languages/finite automata)

- Scientific American article:
    - What does it mean to compute? Give me a description which will also allow me to know what can't be computed.
    - What is the simplest model of computation you can think of? (Finite automaton.)

# Bounding the Power of the Machine

This course is different from probably every course you have taken. In every other course, you learn algorithms or programming methods, hardware design and so on – methods for finding solutions and implementations. In this course, we will learn what can't be done: what problems you will never succeed in programming solutions to, what problems for which you are unlikely to find solutions which take a reasonable amount of time and space, and what problems will not yield to certain standard design techniques.

This course is sometimes met with the question: why do we need to know this? The first response to this question is that to know limits means to know when to give up or when to use more sophisticated techniques. In real life, rather than in homework, one may encounter problems which one would like to solve. It is nice to abandon a problem for a reason other than that you were unable to solve it, to be able to say that's not possible or not likely to be possible; that we need to change the problem or simply give up. It may seem that negative results in computer science are inferior because they do not have immediate application. In recent years, the need for determining the limits of computing has become critical and marketable. Consider the need for on-line privacy, security, and cryptography. Each of these depends on knowing what a computer can't do in a reasonable amount of time. For example the security of the RSA encryption scheme is related to our inability to factor large numbers quickly. One notion of privacy is that we don't want to allow a computer to go on a fishing expedition, i.e., search all files to find individuals with certain characteristics, but we do want to allow targeted lookup of a small number of individuals. Cynthia Dwork of Microsoft has developed secure data bases which require too much space or time to search widely yet accommodate directed searches. We have also seen a need to distinguish human users from bots for the prevention of spam. For example, an email address is sometimes described on a web page with a gif rather than plain text, as that is thought to be readable by people but not by machines. For this to work, we need to know that a pattern recognition problem which is easy for people to solve is beyond the resources of a computer. The problem of distinguishing computers from humans dates back to the 1950s when people like Turing proposed the question of whether computers could think. Turing proposed the Turing test: Can a person having a dialogue with a computer and a person distinguish one from the other? Recently, Manuel Blum and Louis van Ahn at Carnegie Mellon attempted to formally describe a class of problems which can be used to distinguish machines from human beings. Unlike the Turing test, this theory distinguishes the human from the machine on the basis of speed of response. This work is the basis for the widely-used human-identification system known as CAPTCHA. The limits of what is computable given limited resources has entered even the social sciences such as economics and psychology. For example, when economists model the efforts of agents to maximize their utility, they now sometimes

take into account the fact that agents have limited computational power in making their decisions, and therefore may not be able to exactly determine their optimal strategy.

The problem of what is computable and what is not actually has a long history and dates back to the early 1900s before the invention of the modern computer. In fact, we will go back and reconstruct an older result, from 1870, of Cantor, who discovered that some kinds of infinity are “bigger” than others. At the turn of the 20th century, people began to question the foundations of mathematics. In the 1930's Gödel’s incompleteness result showed that within every theory of mathematics, there are mathematical statements which are true but can’t be proven according to the theory.

In the 1920s and 30s, several mathematicians attempted to formalize the notion of computability, which they understand in terms of what people do when they use pen and paper and follow a finitely described procedure (or algorithm.) Surprisingly, all the models suggested turned out to be equivalent.

## A Brief Overview

The first half of the course will introduce weaker models of computation, such as finite automata (and regular languages) and pushdown automata (and context free grammars). These provide useful paradigms for programming, and so this section concerns a set of techniques which could equally well have been taught in an algorithms course but are traditionally taught here. Another goal of this section is to provide an introduction to formal proofs about strings and these models. These techniques are useful for formal verification methods of hardware and software systems. This will be followed by an introduction to context-free grammars and pushdown automata, an intermediate model which is also of fundamental in the design of parsers used in language processors such as compilers.