CS304 - Formal Languages and Automata Theory 4^{th} **Semester**

Lecture 1

Instructor: Prof. Prateek Vishnoi *Indian Institute of Technology, Mandi*

Why "Theory of Computation or FLAT" ??

- We have read CS212, we looked at many interesting problems like searching, sorting, graphs etc. This course looks same, however it is not.
- This course focuses on different objectives. Goal here is not on how effectively we can solve a problem. Here we focuses on broader question

"What are the problems for which we have an algorithm??"

- Wait a second! Is there any problem for which we can't have an algorithm ?? My intuition says that if we can think of a problem, we can solve it(mayn't be efficiently).
- Unfortunately, the above intuition is wrong. Not just that, not even 1% of the problems are computable(have an algorithm), which is completely non-intuitive.

This question gives birth to the "Theory of Computation"

One confusion!

- There are infinitely many problems with arbitrary size of input and output.
- Example 1 : Sort the list of numbers.
- Example 2 : Searching a number in the list.
- In the above two examples, input size and output size are different. First example has a list as an input and output. However, in example 2, input is a tuple of number and list and output is a number.
- This means that different problems may have different size of inputs and outputs.

How we will study all the computational problems on a common platform ??

Need of an hour!

We need a common framework where we can study all the computational problems with common input and outputs. We want to grasp such framework where all the computational problems are there.

Transition to a new framework

- Think about any computational problem from CS212. If we carefully see, we can observe that every computational problem is a function mapping input to outputs.

$$f : I \longrightarrow O$$

- With every computational problem there is a unique function associated with it.
- Algorithm or Program of any computational problem is a finite length description to map all inputs to outputs of that computational problem.
- Example :
  Input : A number n.
  Output is 1 (if n is prime) otherwise 0.

$$f : n \longrightarrow \{0, 1\}$$

  Above problem only defines the function associated with it, doesn't tell anything about its algorithm. One algorithm to the above problem might be to check whether there exist any number k between 2 to $(n-1)$ such that n/k has a remainder 0. If yes then return 0, otherwise 1. So, clear this thing in your mind that defining a function(or a problem) is a different thing and giving an algorithm is different.

Set membership problem

Input: Set S and element a.
Output: 1 if $a \in S$, and 0 if $a \notin S$.

Graph of a function

$$f : I \longrightarrow O$$

Graph of a function is a tuple defined as :

$$graph(f) = \{(a, b) | f(a) = b\}$$

First Claim

> - ***If there is an algorithm to compute f then there is an algorithm to solve the set-membership problem for the $graph(f)$.***

Proof:

- Assume we have an algorithm $\mathcal{A}$ to compute f.
- For (a, b) given as input(for set-membership problem), compute $f(a)$ using algorithm $\mathcal{A}$.
- Now, if $b = f(a) \implies (a, b) \in graph(f)$

- Now, we have an algorithm for deciding whether $(a, b) \in graph(f)$ or not.
- Correctness can be verified easily.

■

Second Claim

- ***If there is an algorithm to solve the set-membership problem for the*** $graph(f)$ ***then there is an algorithm to compute*** f***.***

Proof: Not at all difficult, expecting this from your side. ■

- Bottom line is, now we got the liberty from claim 1 and claim 2 that instead of analysing the computability of function f(which is little difficult), we can analyse the set memebership problem of $graph(f)$. This would reflect the same computability about the function f.

Final Framework for the course

We will talk about the set memebership problem throughout the course rather than function f. Let's define all the things formally.

- Symbols : $a, b, 0, 1 \ldots$
- Alphabet : Finite set of alphabets.
- $\Sigma = \{0, 1\}$
- Σ^* represents set of all binary strings including ϵ (called null string).
- Formal Language : A formal language $\mathcal{L}$ over alphabet Σ is a subset of Σ^*.

Few examples of formal languages are :

1) $L_1 = \{0, 001, 010\}$

2) $L_2 = \{x \in \Sigma^* | x$ has equal number of 0's and 1's$\}$

So for now, we will only be concerned with the computability of the set membership of formal languages.