Lecture 1

Instructor: Karteek Sreenivasaiah

8th March 2020

Computability Theory

The course CS2410 on Theory of Computation dealt with *Computability Theory*.

Given a model of computation, we studied:

- What languages (Boolean functions) can be computed?
- What languages cannot be computed?
- Proof techniques for all of the above.

There was no analysis of the amount of resources used by the computation models.

Computability Theory

Computability Theory generally allows for unlimited resources.
Example:

- Push Down Automata were allowed a stack of unbounded size.
- Turing machines were allowed unbounded *running time* and unbounded *tape space*

In this course (CS2420):
We will study computation while closely examining the resources used for various functions.

Computational Complexity Theory

Computational Complexity - as the name suggests is about studying the complexity (difficulty) of computing functions.

Two of the main goals[1]of this area are to understand:

- What functions are *hard* to compute?
- Why are some functions hard to compute?

But wait! What does '*hard to compute*' even mean?
As usual, we will have to define this formally.

Let's understand it intuitively first.

[1]There is a widespread misconception that Complexity Theory is about measuring running time of algorithms. It is not!

Computational Complexity Theory

Example

Consider the following two tasks:

- Multiplying two *n* digit numbers (integers in decimal)
- Adding two *n* digit numbers (integers in decimal)

Which do you think is harder to compute by hand?

Computational Complexity Theory

Example

Consider the following two tasks:

- Multiplying two n digit numbers (integers in decimal)
- Adding two n digit numbers (integers in decimal)

Which do you think is harder to compute by hand?

Multiplication? Maybe.
But you might think that because you know the procedure taught in high-school to multiply two integers.

What if there is some other algorithm that makes multiplication easier than addition, and we just haven't figured it out yet?

In Computational Complexity, we are interested in formal proofs to compare hardness of functions.

Course Plan

We explore the basics of computational complexity:

1. Cost of simple conversions between various TM models.
2. The complexity classes P and NP.
3. Several examples of languages in P and NP.
4. The P vs NP problem.
5. Notions of *hardness* and *completeness* for complexity classes.
6. Boolean Formula Satisfiability (SAT) is NP-complete.
7. Polynomial time reductions from SAT to several natural problems to establish their NP-completeness.

Topics (2) to (7) from NPTEL. We will have Q&A on Google Classroom.

Preliminaries and Notation

Let M be a deterministic Turing machine deciding a language $L \subseteq \Sigma^*$.
Let $f : \mathbb{N} \to \mathbb{N}$ be a function.
Let $t(x)$ be the number of steps that M takes on input $x \in \Sigma^*$.

Running time (worst case)

The running time of M is $f(n)$ if and only if:

$$\forall n \in \mathbb{N}, \max_{x \in \Sigma^n} \{t(x)\} = f(n)$$

Preliminaries and Notation

Let M be a deterministic Turing machine deciding a language $L \subseteq \Sigma^*$.
Let $f : \mathbb{N} \to \mathbb{N}$ be a function.
Let $t(x)$ be the number of steps that M takes on input $x \in \Sigma^*$.

Running time (worst case)

The running time of M is $f(n)$ if and only if:

$$\forall n \in \mathbb{N}, \max_{x \in \Sigma^n} \{t(x)\} = f(n)$$

In words:
For each input length n, look at the input x (of length n) on which M takes the longest amount of time. The amount of time M takes on x is the running time for that length.

For each length n, the function $f(n)$ should equal the longest time taken as described above.

Preliminaries and Notation

Let M be a non-deterministic Turing machine deciding $L \subseteq \Sigma^*$. Let $f : \mathbb{N} \to \mathbb{N}$ be a function. Define $t(x)$ as follows:

$$t(x) = \max_{\text{branch } \rho} \left\{ \begin{array}{l} \text{number of steps that } M \text{ takes on input} \\ x \in \Sigma^* \text{ along the branch } \rho \end{array} \right\}$$

In words: $t(x)$ now denotes the worst case across all non-deterministic branches of M.

Preliminaries and Notation

Let M be a non-deterministic Turing machine deciding $L \subseteq \Sigma^*$.
Let $f : \mathbb{N} \to \mathbb{N}$ be a function. Define $t(x)$ as follows:

$$t(x) = \max_{\text{branch } \rho} \left\{ \begin{array}{l} \text{number of steps that } M \text{ takes on input} \\ x \in \Sigma^* \text{ along the branch } \rho \end{array} \right\}$$

In words: $t(x)$ now denotes the worst case across all non-deterministic branches of M.

Running time (worst case)

The running time of M is $f(n)$ if and only if:

$$\forall n \in \mathbb{N}, \max_{x \in \Sigma^n} \{t(x)\} = f(n)$$

Preliminaries and Notation

Typically, we are only interested in *asymptotic* bounds on running time. i.e., we focus on how the running time increases with increase in input length.

You should learn the following asymptotic notation as homework:

- Big-O, little-o
- Big-Ω, little-ω
- Θ

The above can be found in section 7.1 in the text by Sipser.

Time Complexity Classes

DTIME

Let $t : \mathbb{N} \to \mathbb{R}^+$. The complexity class $\text{DTIME}(t(n))$ to be the set of all languages that are decidable by a deterministic single tape Turing machine in time $O(t(n))$.

Time Complexity Classes

NTIME

Let $t : \mathbb{N} \to \mathbb{R}^+$. The complexity class $\mathrm{NTIME}(t(n))$ to be the set of all languages that are decidable by a non-deterministic single tape Turing machine in time $O(t(n))$.

Time Complexity Classes

Example

Consider the following language:

$$A = \{0^n 1^n \mid n \geq 0\}$$

Recall that we saw an algorithm in class which is roughly as follows:

- Scan from left to right to check the pattern is 0^*1^*.
- Cross off the leftmost 0 that is not already crossed off.
- Scan right and cross off the first 1.
- Go all the way left, repeat.
- If all symbols were crossed off, *accept*. Else *reject*.

Time Complexity Classes

It is easy to see that the above algorithm gives:

Theorem

$A \in \mathrm{DTIME}(n^2)$

Exercise: Show that $A \in \mathrm{DTIME}(n \log n)$
(Hint: Cross off half the symbols in each round)

Relationship among models

Recall the conversion from multi-tape TM to single tape from class.
What is the blow-up in running time incurred?

Relationship among models

Recall the conversion from multi-tape TM to single tape from class. What is the blow-up in running time incurred?

Theorem

Every multitape Turing machine with running time $t(n)$ has an equivalent single tape Turing machine with running time $O(t^2(n))$.

Relationship among models

The conversion discussed in class was roughly:

- Write the contents of all the k tapes one after the other.
- Use a delimiter to separate them.
- To simulate each transition:
  - Scan from left to right to "collect" the symbols under each head.
  - Scan from left to right to update each simulated tape's contents.
  - Repeat.

Observation: A TM that runs for at most $t(n)$ time can only use at most $t(n)$ many cells on the tape.

Analysis:

- Each scan in the simulation uses time at most $k \cdot t(n)$.
- There are at most $c \cdot t(n)$ many scans (where c is a constant).
- This gives a total of $c \cdot k \cdot t^2(n) \in O(t^2(n))$.

Relationship among models

Recall the conversion from non-deterministic to deterministic TM. What is the blow-up in running time incurred?

Theorem

A non-deterministic single tape TM running in time $t(n)$ has an equivalent deterministic single tape Turing machine with running time $2^{O(t(n))}$.

Relationship among models

The conversion discussed in class was roughly:
Let b be the most number of non-det choices at any particular transition.
Then the total number of vertices in the non-deterministic tree is $O(b^{t(n)})$.

- We created a 3 tape deterministic machine.
- The machine *explores* the non-deterministic branches.
- The exploration was using a Breadth-first search approach.
- Exploring each branch required starting from root all over again.

Relationship among models

The conversion discussed in class was roughly:
Let b be the most number of non-det choices at any particular transition.
Then the total number of vertices in the non-deterministic tree is $O(b^{t(n)})$.

- We created a 3 tape deterministic machine.
- The machine *explores* the non-deterministic branches.
- The exploration was using a Breadth-first search approach.
- Exploring each branch required starting from root all over again.

Analysis:

- There are total of $b^{t(n)}$ nodes to explore.
- Each node takes at most $O(t(n))$ time to reach.
- Total time taken is $O(t(n) \cdot b^{t(n)}) \in 2^{O(t(n))}$.

Thank you!