CSE 431 – Theory of Computation
Lecture 9 – Tues, Apr 29
Lecturer: James Lee
Scribed by: Devin Field

# Complexity Theory

Before now, we have simply looked at if a Turing Machine can exist to solve a problem without any regard to the time involved in doing so. We are now going to look at the resource consumption of TMs, specifically, the time it takes to solve a problem.

**Example:**

$A = \{0^k 1^k : k \geq 0\}$

A is a decidable language of a series of 0's followed by the same number of 1's

M = "1: Make sure the input looks like $0^*1^*$

2: Iteratively cross off first a '0' and then a '1'

3: Finally check that the tape is 'XXXX…XXX_' "

**Note:** _ is the symbol used to indicate a blank spot on the tape

How long does this take to run?

If it has n-bits, it intuitively runs in $O(n^2)$ steps

**Definition:** The <u>time complexity</u> of the TM M is a function $f: \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the max number of steps M takes on input of size n.

**Definition:** $TIME(f(n)) = \{L : L \text{ can be decided by a TM with time complexity } O(f(n))\}$

Ex: $A \in TIME(n^2)$

# Big O-Notation

$f, g: \mathbb{N} \to \mathbb{R}_+$

$f(n) = O(g(n))$ if there are $c, n_0 \geq 0$ such that $f(n) \leq c * g(n)$ for $n \geq n_0$

**Example:** $100n^4 + 7n^3 + n + 1000 = O(n^4)$

Order of magnitude: $O(\log(n)) \leq O(n^c) \leq O(2^n)$

We don't have to worry about base of the logs because $\forall\, a, b \to O(log_b\, n) = O(log_a n)$

$f(n) = 2^{O(n)}$ means $\exists\, c, n_0\ f(n) \leq 2^{cn}$ for $n \geq n_0$

**Definition:** $f(n) = O(g(n))\ if\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Can we find a faster algorithm for A than $O(n^2)$?

Divide the number of zeroes and ones by two each step and then check the parity

Ex: 0 0 0 0 0 1 1 1 1 1

X0X0XX1X1X

XXX0XXXX1X

XXXXXXXXXX

Step 1: Check input → $O(n)$

Step 2: Loop over the input until it is all X's → O(log($n$))

Step 2.a: Loop across the input, cross out every other 1 and 0 → $O(n)$

Step 3: Check if all X's → $O(n)$

The time complexity of this machine is $O(n + n * \log(n) + n) = O(n * log(n))$

This means that $A \in TIME(n * \log(n))$

But what if we had a two tape TM?

Then we can solve it in $O(n)$ time!

First, verify that the input is all 0's followed by all 1's

Iterate from left to right on the input:

If it is a 0, move the lower tape to the right

If it is a 1, move the lower tape to the left

If you try to move off the left side of the tape, Reject

If you aren't in the first spot after the entire input, Reject

Else, Accept

This leads to the "Extended Church-Turing Thesis": "All reasonable computation models can simulate each other with only polynomial slow down."

**Theorem:** If $t(n) \geq n$, then every multi-tape TM with runtime t(n) can be simulated by a single-tape TM in time $O(t(n)^2)$

**Proof (Sketch):**

Recall how we simulate a multi-tape TM on a single-tape TM

Single-tape: INPUT#TAPE 1#Tape 2#.....#Tape n

In order to calculate a single step of a multi-tape:

1 pass to read all heads

1 pass to move all the tape heads

This means that a single tape TM can simulate a single step of a multi-tape TM in $O(L)$ where L is the current length of the tape

$L \leq O(t(n))$

Therefore, there are going to be $O\big(t(n)\big)$ steps all taking a maximum of $O\big(t(n)\big)$ time

$O\big(t(n)\big) * O\big(t(n)\big) = O(t(n)^2)$

The "Extended Church-Turing Thesis" is largely believed to be false now because of Quantum Computers

Quantum Computers can factor a polynomial in $O(n^c)$

Regular computers take $O(2^{n^{1/3}})$

# P vs NP

P is all the problems that can be solved in polynomial time
$P = \bigcup_{k\geq 0} TIME(n^k)$ → All the algorithms that are "efficient"

**Non-deterministic Computation:**
In non-det computation, the TM can execute multiple choices at the same time. At each choice, it "splits" into multiple executions, allowing for all options to be explored quickly. The TM accepts the input if <u>any</u> of the paths accept.

**Definition:** A <u>Hamiltonian Path</u> is a path that visits each node in a graph exactly one.

$HAMPATH = \{< G, s, t >: G$ is a graph that has a directed Hamiltonian path from $s \to t\}$
This is an efficient problem to solve using a non-det computer

**Definition:** A non-det TM is a <u>decider</u> if it halts on every computation path.

**Definition:** The non-det time complexity of N is $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the max height of the computation tree on any input of size n.

**Definition:** $NTIME(f(n)) = \{L: L$ can be decided by a non-det TM with complexity $O(f(n))\}$

$HAMPATH \in NTIME(n * log(n))$

Where does the $\log(n)$ come from?
The max number of states that can be jumped to from a single node at each point is a constant.
We call this constant b
If I want to jump to n different states, it actually takes $\log(n)$ steps

**Definition:** $NP = \bigcup_{k\geq 0} NTIME(n^k)$

**"Stupid" Theorem:** $NTIME(t(n)) \leq TIME(2^{O(t(n))})$
A non-det. computer can be simulated on a reg. computer
It can simply encode every path and then try them in order
One path takes $O(t(n))$ on a 3 tape machine
Tape 1: The input
Tape 2: The current path to take
Tape 3: The work tape
The max number of paths possible is $b^{t(n)}$
$O(t(n) * b^{t(n)}) = 2^{O(t(n))}$
Therefore a single tape TM = $2^{O(t(n))^2} = 2^{O(t(n))}$