Computational Complexity

This is a study of inherent complexity of computational problems and resources (time, space, communication, etc) needed to solve them.

Topics studied include

- Time complexity, Space complexity
- Determinism vs Non-determinism
- Role of randomness in computation
- Computing exact vs approximate solutions
- Worst case vs average case complexity
- Communication complexity
- Foundations of cryptography, interactive proofs.
- More .....

In this course, we will only study the topic of NP-completeness that involves considerations of time complexity, non-determinism and worst case complexity.

All problems/languages considered henceforth are decidable. The main concern would be time needed to solve them as a function of the input size.

Example $L = \{ 0^k 1^k \mid k \geq 0 \}$.

Algorithm (= TM program) that decides L:

"
- Scan input left to right. If '1' is followed by '0', reject.
- Traverse back and forth, crossing out a '1' for every '0' that is crossed out.
- Accept if all '0's and '1's are crossed out (simultaneously).
- Otherwise reject."

Def Given a TM M and input $x \in \Sigma^*$, running time of M on x is the number of steps M takes till it accepts or rejects.

(Note: We are henceforth dealing with only

those TMs that always halt.)

Def The (worst-case) running time of M as a function of the input size is a function $f: \mathbb{N} \to \mathbb{N}$ s.t.

$f(n)$ = Maximum number of steps M takes on any input $x \in \Sigma^*$ s.t. $|x| = n$. ▨

We make several observations:

① The TM deciding $L = \{0^k 1^k \mid k \geq 0\}$ as described before has runtime of roughly n^2. For inputs of type $0^k 1^k$, $n = 2k$ and the TM has to traverse back-and-forth k times, each traversal needing roughly k steps.

② We generally are not too interested in the precise function $f(n)$ (= runtime), but in upper bounding $f(n)$ in a

"reasonable" manner. We introduce the Big-O notation next towards this end.

(3) The same language could be decided by another TM that runs in less time.

In fact $L = \{0^k 1^k \mid k \geq 0\}$ can be decided by another TM in roughly $n \log_2 n$ time (and still using only one tape). This machine crosses out every alternate '0' and then every alternate '1'. It now knows whether the number of '0's and '1's is even or odd. If the two parities differ, it rejects. Otherwise it repeats this procedure on the '0's and '1's that remain (i.e. not crossed out yet).

④ The runtime could depend on the specific model, i.e. 1-tape TM vs 2-tape TM vs RAM (Random Access Memory) model.

E.g. $L = \{0^k 1^k \mid k \geq 0\}$ can be decided in time roughly n on a 2-tape TM. The m/c can simply copy the suffix 1^k onto the second tape and then by 1-to-1 comparison match '0's with equally many '1's in a single left to right scan.

We now introduce the Big-O notation and then define the "time complexity classes" keeping in mind observations ②-④ above. By "natural" function, we mean any "standard" function such as n, n^2, $n \log_2 n$, 2^n etc (this is admittedly imprecise terminology,

but let's live with it).

Big-O Notation

Def Let $f(n), g(n) : \mathbb{N} \to \mathbb{N}$ be two functions. We say that $f(n)$ is $O(g(n))$ (Big-O of $g(n)$) if there exists a (large enough) constant C and (large enough) integer n_0 such that

$$f(n) \leq C \cdot g(n) \quad \forall\ n \geq n_0 .$$ ▨

We motivate the definition with several clarifying points.

① We think of $f(n)$ as the runtime of an algorithm and $g(n)$ as a natural function that upper bounds $f(n)$.

E.g. the runtime of the two algorithms for $L = \{0^k 1^k \mid k \geq 0\}$ is $O(n^2)$ and $O(n \log_2 n)$ respectively, and that of the algorithm on (on 1-tape m/c)

2-tape m/c is $O(n)$.

(2) We are mainly interested in upper bounding $f(n)$ for sufficiently large n (i.e. $n \geq n_0$). This is referred to as an "asymptotic upper bound" and makes sense in the world of exploding size of datasets and need for algorithms the work on such large datasets.

(3) Theoretically speaking "constants don't matter" (but of course they do in practice!). Thus the functions $10n$ and $10^8 n$ are both simply $O(n)$. There are two reasons to "ignore" constants (i.e. the constant C in the upper bound $f(n) \leq C \cdot g(n)$, which gets "ignored" by the notation $O(g(n))$):

(a) Suppose there are two algorithms with runtimes:

	$f(n)$	$n=10$	$n=10^3$	$n=10^6$
Alg 1	$1000n$	10^4	10^6	10^9
Alg 2	n^2	10^2	10^6	10^{12}

Clearly, in terms of having low runtime, Algo-2 wins for small values of n (up to 10^6), but eventually (10^6 and there-after) Algo-1 wins and wins more and more convincingly as n increases further and further. This "asymptotic behavior" remains true irrespective of the constant 1000 in the $1000n$ vs n^2 comparison.

(b) It turns out that a strange feature of the TM model is that if there is a TM M that runs in time $f(n)$, then it can be simulated by another TM M' that runs in time $\frac{f(n)}{C}$ where C can be any constant. The idea, roughly, is that M' can partition the input tape into chunks/blocks of size C each and then treat each block as one super-cell containing one super-symbol.

This allows M' to simulate C moves of M by a single (super-) move! Thus, in the TM model, an "algorithm" can be speeded up by any constant factor, making constant factors meaningless (as far as theory is concerned).

— × —

Examples of O-notation

① $5n^2 + 70n + 100$ is $O(n^2)$.

This is because one can take $n_0 = 100$, $C = 7$ so that

$$5n^2 + 70n + 100 \leq 5n^2 + n^2 + n^2$$
$$= 7n^2$$
$$= Cn^2 \quad \forall n \geq n_0.$$

One could also take $n_0 = 1$, $C = 175$ so that

$$5n^2 + 70n + 100$$
$$\leq 5n^2 + 70n^2 + 100n^2 \quad \forall n \geq n_0.$$
$$= 175n^2 = Cn^2$$

(2) Generally, any polynomial function

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

is $O(n^k)$ where n^k is the higest degree.

(3) n^{20} is $O(2^n)$.

$\log_2 n$ is $O(\sqrt{n})$.

$2^{\sqrt{\log_2 n}}$ is $O(n)$.

(4) $\left.\begin{array}{l} \log_{10} n \text{ is } O(\log_2 n) \\ \log_2 n \text{ is } O(\log_{10} n) \end{array}\right\} \because \log_{10} n = \frac{\log_2 n}{\log_2 10}$

Thus all logarithms (with base > 1) are the same up to constant factor. In computer science, $\log n$ refers, by convention, to $\log_2 n$.

(5) It may be useful to remember that

$$\log n \ll 2^{\sqrt{\log n}} \ll \sqrt{n} \ll n \ll n \log n$$
$$\ll n^2 \ll n^3 \cdots \ll 2^n \ll 3^n \cdots \ll 2^{2^n}$$

where $g(n) \ll h(n)$ indicates that $h(n)$ has higher asymptotic growth than $g(n)$.

We are now ready to define the "time complexity classes". In the following, $t(n)$ denotes a natural function (e.g. n, $n \log n$, n^2 etc) s.t. $t(n) \geq n$.

Def A language L is decidable in time $t(n)$ if there is a TM M that

- decides L and
- runs in time at most $O(t(n))$, and
- may have $k \geq 1$ tapes for constant k.

Note that according to the definition, $L = \{0^k 1^k \mid k \geq 0\}$ is decidable in time $O(n)$ since there is a 2-tape TM that decides in time $O(n)$.

Def Let $t(n) \geq n$ be a natural function.

DTIME($t(n)$) is the class of all languages decidable in time $O(t(n))$ (by a multi-tape TM).

Note ① We generally consider $t(n) \geq n$ since a TM needs at least n steps just to read its size n input.

② "D" in the notation DTIME( ) refers to "deterministic" TMs. We will consider non-deterministic TMs later.

—— × ——

We note the following observations and results, but these are not the focus of this course.

① 1-tape vs k-tape machines.

Theorem Any k-tape TM that runs in time $t(n)$ can be simulated by a 1-tape TM

in time $O(t(n)^2)$.

This theorem is easy. It turns out that the quadratic loss in the runtime is inherent and necessary. E.g.

$$L = \{ w \in \{0,1\}^* \mid w \text{ is a palindrome} \}$$

can be decided by a 2-tape TM in time $O(n)$, but on a 1-tape TM, necessarily and provably needs asymptotic time $\geq n^2$ (!)

② k-tape vs 2-tape machines.

Theorem Any k-tape TM ($k \geq 3$) that runs in time $t(n)$ can be simulated by a 2-tape TM in time $O(t(n) \log t(n))$.

This theorem is difficult!

③ The definition of the class $DTIME(t(n))$ allows multi-tape TMs.

④ One can ask whether more time allows TMs to solve harder problems. The answer

is affirmative.

Theorem (Time Hierarchy Theorem)

Let $t(n)$, $t'(n)$ be natural functions such that $t'(n) \gg t(n) \cdot \log t(n)$. Then

$$DTIME(t(n)) \subsetneq DTIME(t'(n)),$$

i.e. there exists a language L that is decided in time $t'(n)$ but not in time $t(n)$.

∎

The theorem is not difficult. The proof relies on the diagonalization method.

— x —

Henceforth, we will not be overly concerned about specific runtimes $t(n)$, but only whether $t(n)$ is polynomial time (n, n^2, n^4, n^{20} etc) or not.