CS 6363-001: Summer 2020

Lecture #1

Course material in several places:
1. www.utdallas.edu/~chandra
   under CS6363 - Computer Algorithms
2. e-learning site
   (Click on my webpage & e-learning site)

Material is of 3 types:
1. Printed lecture notes (from previous semesters)
   at my website
2. Audio/video classes
   (in scanned files at my webpage, elearning
    And recorded files in Course room
     after the class is over
3. HW solutions: my webpage.
4. Announcements etc.: my webpage

Grading:
1. 6 HW that will be graded
   (total weight of all HW: 10%)
2. 3 exams: Exam I: 25%  Exam III: 35%
   Exam II: 30%.
3. There may be a makeup for Exam I (only)
   for 10%.

For HW only: Group formation is recommended.
You need to send email to me with
cc to all members; Size: no less
than 3, no more than 5.
One submission/group.
Since this is an ONLINE class, all HW, exams will be submitted in pdf files (one per HW, one for each exam) by scanning. [No other format please!] all submission on e-learning.

All classes will be in Course room of e-learning for this course. Please familiarize yourself with this, if you are not familiar with this.

Content of this Course

Book: Introduction to Algorithms, by Cormen, Leiserson, Rivest, and Stein
MIT Press (3rd Edition)
+ for specific sections other books (will be mentioned at that time)

TA: TBA

0. Order Notation (Ch3, L#2, HW#1)
1. Recurrence Relations (Ch4, L#3, HW#2)
2. Divide-and-Conquer Algorithms
   (Ch9, Ch4, Ch33, + ... Left # 4, 5, 6, FFT note)

Exam I:

4. Greedy Algorithms (Ch 16, L #7, HW #4)

5. Dynamic Programming (Ch 15, L #8, Add. Lect) HW #5

Exam II

6. Graph Algorithms
   a) Spanning Tree (Ch 23, Ch 21, L # 9)
   b) Single Origin Shortest Path (Ch 24, L # 10, 11)

 c) All pairs Shortest Paths (Ch 25, L # 10)
   Data Str
   d) Maximum Flow (Ch 26, Ch 22, L # 12)

7. NP-Completeness (Ch 34, L # 14) HW # 7.
   (not to be turned in)
What is included & what is excluded

I am assuming that all of you know what algorithms are. If not, think of them as extremely precise recipes (as in cooking). At each step and in every situation (state) that we might find ourselves in, an algorithm tells you what the next step is — no ambiguities or lack of instructions.

So once an algorithm is specified, a "machine" (or "moron") can carry out instructions.

There are many different ways to "classify" algorithms.

1. Deterministic vs Randomized (Probabilistic)

In deterministic algorithms, each step is clearly specified with only one possible action. There are no "Coin-tossers", or use of "Random" numbers or probabilistic choice. These are used in Randomized algorithms.

Example of Randomized algorithm:
2. **Sequential** vs Parallel & Distributed algorithm

   One "operation" at a time.

3. **Offline** vs Online

   All data given at the beginning.

---

**Analysis of algorithms**

Most important characteristic of an algorithm is that it must produce correct results for all input instances of a problem. We are not interested in invalid algorithms. Often, this makes proofs of correctness an integral part of algorithm development.

The next question that arises is: How do we compare two valid algorithms for the same problem? [We assume that both use the same set of basic operations]. This is where **order notation** comes in.
Keep in mind that we are interested in valid algorithms that perform well for large size instances. Suppose we have two valid algorithms A and B and their performance is measured so that the measure is independent of machines they are run on. So we measure performance of an algorithm on the basis of the number of operations needed to solve a problem of "size" n as a function of n. [We realize that some operations are more time consuming than others and the time may also depend on the "size" of the input values used.] This is called "time complexity" of an algorithm. The amount of "space" used by an algorithm also plays a role in comparing algorithms. [However, in this course, we will mainly consider time complexity]. With all this preamble, we get a "graph" of time vs size for an algorithm.

\[
\begin{array}{c|c}
\text{Size} & \text{Time} \\
\hline
\end{array}
\]

It will be assumed this is an increasing function from now on.
However, there is one more difficulty. It has been observed that for different inputs of the same size, an algorithm takes varying amounts of time. An example will make this point clear.

Consider an input: An unsorted array of numbers.

Suppose we want a sorted version of the above. One way to do this is to do the following.

Compare each element with the next one and if there is an inversion, exchange the two.

Two possible inputs:

1) Already sorted input: Alg. takes \( n - 1 \) comparisons

2) Reverse order input: Alg. takes \( \frac{n(n-1)}{2} \) comparisons

Hence, there is a "range" of "time" values for same size input. There are 2 approaches to handle this: Worst case, Average case.

Average case: what weights on input instance to use? Analysis much harder.

Choice in this course.
Added benefit of choosing "Worst Case":

Algorithm user can only be "pleasantly" surprised.

So from now on when we say that an algorithm A takes an amount of "time" f(n) for an input of "size" n, we mean f(n) is the **Worst-Case** time taken by algorithm for input of size n.

Suppose Alg. A takes f(n) \( \{ \) for the same problem. Alg. B takes g(n) \( \} \).

There are several possible "pictures":

(I) Alg. A is "better" than alg. B.

(II) For large values of n, Alg. A is better than B.

(III) No "clear" winner.

There is one more issue before considering order notation.
Suppose \( g(n) = (1+\varepsilon) f(n) \) where \( \varepsilon > 0 \) is small.

"Although" we normally would prefer alg that take \( f(n) \), this "time" advantage may be lost due to "other" factors. To make this stronger, we need Order Notation which compares "asymptotic" performances (as \( n \to \infty \)).

These are mainly comparisons of \( f(n), g(n) \) (and thereby comparison of algorithms).

It will be assumed that
\[
\begin{align*}
&f(n) > 0 \quad \forall n > 0 \quad \lim_{n \to \infty} f(n) = \infty \quad f(n) \uparrow \text{ in } n \\
g(n) > 0 \quad \lim_{n \to \infty} g(n) = \infty \quad g(n) \uparrow \text{ in } n
\end{align*}
\]
from now on.

I am using the notation in your book.
There are five symbols used.

\( \Theta, \Omega, O, \omega \)

Bigoh, Big \( \Theta \), little \( \omega \)

Omega, oh \( \Omega \).

We will go into their definitions and usage now.
At the outset, we mention that these five
are neither mutually exclusive (i.e. there are pairs
of functions $f(n)$, $g(n)$ such that more than one
of these symbols hold) nor are they collectively
exhaustive (i.e. there are pairs of functions such
that none of these hold) though this latter
property is rare for functions that represent
time complexity of practical algorithms.

Definitions. For each of these definitions we
use the ratio $h(n) = \frac{f(n)}{g(n)}$ [Note: Since $g(n) > 0$, this is well defined.]

\[
\begin{align*}
  f(n) &= O(g(n)) \\
  h(n) &\leq c g(n) \\
  n &= n_0 \\
  f(n) &= \Omega(g(n)) \\
  h(n) &\geq c g(n) \\
  n &= n_0 \\
  f(n) &= \Theta(g(n)) \\
  h(n) &\leq c_1 g(n) \quad \text{and} \quad h(n) \geq c_2 g(n) \\
  n &= n_0 \\
  f(n) &= o(g(n)) \iff \lim_{n \to \infty} h(n) = 0 \\
  f(n) &= \omega(g(n)) \iff \lim_{n \to \infty} h(n) = \infty \quad \text{In both these cases, limit is defined to exist.}
\end{align*}
\]
Practical use (most of the time) but not always

\[
\lim_{n \to \infty} h(n) = c < \infty \implies f(n) = \Theta(g(n)) \\
\text{(One way)} \quad \text{Big} \\
\lim_{n \to a} h(n) = c > 0 \implies f(n) = \Omega(g(n)) \\
\text{(One way)}
\]

When \(\lim_{n \to \infty} h(n)\) does not exist, (i.e. \(h(n)\) is oscillating or is difficult to compute) we need to use the prior def. for \(O, \Omega, \Theta\).

Now some examples:

Example 1: \(f(n) = an^2 + bn + c; \quad 0 < a < \infty\)
\(g(n) = n^2\)

Our first attempt is to find \(\lim_{n \to \infty} h(n)\) if it exists (even if \(\lim_{n \to \infty} = \infty\), it exists!!)

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{an^2 + bn + c}{n^2}
\]
\[
= \lim_{n \to \infty} \left[ a + \frac{b}{n} + \frac{c}{n^2} \right] = \begin{cases} 
\infty & \text{for } a > 0 \\
0 & \text{for } a < 0 \\
0 & \text{for } a = 0 
\end{cases}
\]

\[\therefore f(n) = \Theta(g(n))\]

Asymptotically \(f(n)\) and \(g(n)\) differ by a multiplicative constant.
Example 2: \[
\begin{align*}
\lim_{n \to \infty} \frac{f(n)}{g(n)} &= \lim_{n \to \infty} \frac{10^n \cdot n}{n^2} \\
&= \lim_{n \to \infty} \frac{10^n}{n} = 0.
\end{align*}
\]

\[f(n) = o\left(g(n)\right)\] little

This example illustrates some weakness of this asymptotic theory. Most practical problems, algorithm that takes \(f(n)\) time is **not** preferable to one that takes \(g(n)\) time.

Example 3: \[
\begin{align*}
f(n) &= 2^n, \quad g(n) = n!
\end{align*}
\]

\[
\begin{align*}
\lim_{n \to \infty} \frac{f(n)}{g(n)} &= \lim_{n \to \infty} \frac{2^n}{n!} = \lim_{n \to \infty} \left[ \frac{2}{2^{n-1}} \cdots \frac{2}{2} \cdot \frac{2}{1} \right] \\
\frac{f(n)}{g(n)} &> 0
\end{align*}
\]

\[\implies \frac{f(n)}{g(n)} \geq 0\]

\[\therefore f(n) = o\left(g(n)\right)\] little