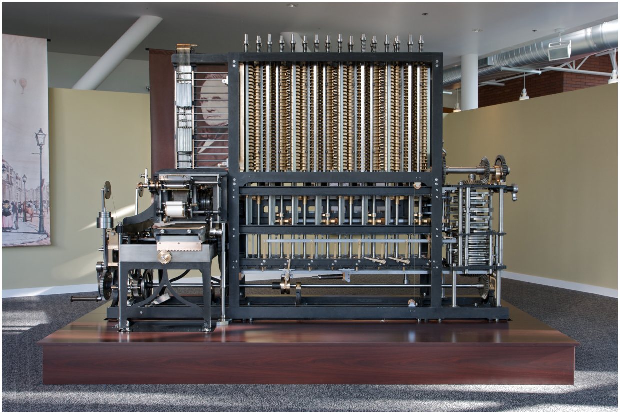
Lecture 8: Order of Growth Classifications

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Indian Institute of Technology Madras

August 25, 2014

Visionary Thinking



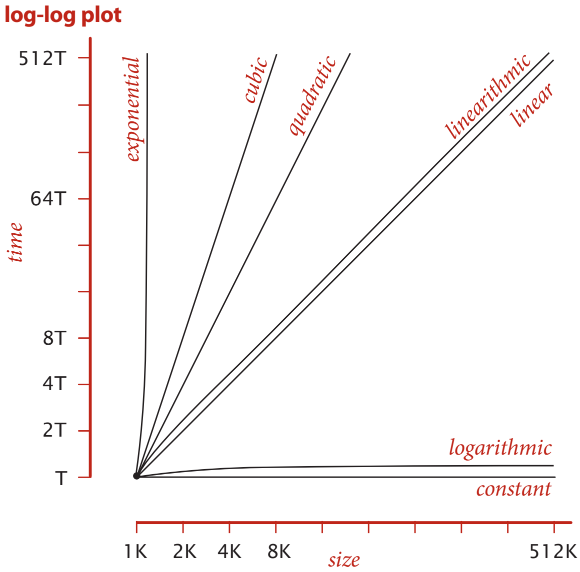
"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise — by what course of calculation can these results be arrived at by the machine in the shortest time?"

— *Charles Babbage (1864)*

Image Courtesy: Wikipedia

Common Order of Growth Classifications

Reference: Robert Sedgewick



Common Order of Growth Classifications

Reference: Robert Sedgewick

order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

Order of Growth: Why do we obsess?

Reference: Robert Sedgewick

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Order of Growth: Why do we obsess?

Assumption: Processor that can execute 10^6 high-level instructions per second

Size	n	$n \lg n$	n^2	$n^2 \lg n$	n^3	1.5^n	2^n	$n!$
10	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
30	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{19} y
50	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 y	10^{51} y
100	< 1 s	< 1 s	< 1 s	< 1 s	1 s	12,892 y	10^{17} y	∞
10^3	< 1 s	< 1 s	1 s	10 s	17 min	∞	∞	∞
10^4	< 1 s	< 1 s	2 min	23 min	12 d	∞	∞	∞
10^5	< 1 s	2 s	3 h	2 d	32 y	∞	∞	∞
10^6	1 s	20 s	12 d	231 d	31,710 y	∞	∞	∞
10^9	17 min	9 h	31,710 y	10^6 y	10^{14} y	∞	∞	∞
10^{12}	12 d	2 y	10^{11} y	10^{13} y	10^{23} y	∞	∞	∞

Adapted from Table 2.1 of *Algorithm Design* by Kleinberg and Tardos

Order of Growth: Why do we obsess?

Assumption: Processor that can execute 10^9 high-level instructions per second

Size	n	$n \lg n$	n^2	$n^2 \lg n$	n^3	1.5^n	2^n	$n!$
10	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
30	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	2 s	10^{16} y
50	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	14 d	10^{48} y
100	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	13 y	10^{14} y	∞
10^3	< 1 s	< 1 s	< 1 s	< 1 s	1 s	∞	∞	∞
10^4	< 1 s	< 1 s	< 1 s	2 s	17 min	∞	∞	∞
10^5	< 1 s	< 1 s	10 s	3 min	12 d	∞	∞	∞
10^6	< 1 s	< 1 s	17 min	6 h	32 y	∞	∞	∞
10^9	1 s	30 s	32 y	949 y	10^{11} y	∞	∞	∞
10^{12}	17 min	12 h	10^8 y	10^{10} y	10^{20} y	∞	∞	∞

Moral of the Story

Why build better algorithms? Can't we use a super supercomputer instead?

> *"A faster algorithm running on a slower computer will always win for sufficiently large instances! Usually, problems don't have to get that large before the faster algorithm wins."*
>
> – *Steve Skiena*

What is a fast algorithm?

Consider the following problems:

1. **Genome Assembly Problem.** Find the shortest common super-string of a set of sequences (*reads*): given strings $\{s_1, s_2, \ldots, s_n\}$; find the super-string T such that every s_i is a sub-string of T
2. **Alignment Problems.** How do you align two protein sequences? structures? graphs?
3. **Parameter Estimation Problem.** Given a set of data $\mathcal{D}$, find the set of model parameters Θ that minimises model error $\mathcal{E}$, with respect to $\mathcal{D}$. Assume that $\theta_i \in \{10^{-4}, 10^3\}$.
4. **8 Queens Problem.** How do you place 8 queens on a chessboard, so that no two queens threaten one another?

Brute Force

- Involves checking every possible solution to a problem
- Typically takes exponential time ($\sim 2^n$ or even $\sim n!$)
- Often useless in practice, esp. for large problems

Aside: See http://en.wikipedia.org/wiki/Four_color_theorem

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+lg\ n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Worst-case analysis

- Running time **guarantees** for *any input* of size n!
- Captures efficiency, in practice

Other analyses

- Probabilistic (expected running time of a randomised algo)
- Amortized complexity (worst-case running time for any sequence of n operations)
- Average-case (expected running time for a random input of size n)
- ...

Big-O Notation

Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n0$.

This definition is often referred to as the "big-O" notation, for it is sometimes pronounced as "$f(n)$ is big-O of $g(n)$."

Actually, $O(g(n))$ is a set of functions, but notation is usually (mis-)written as $f(n) = O(g(n))$, rather than $f(n) \in O(g(n))$.

Asymptotic Notations

$f(n) = \Theta(g(n))$ — Big Theta

- $f(n)$ and $g(n)$ have the *same* order of magnitude
- Tight bound: classify algorithms
- e.g. $n^2, 100n^2, n^2 + 10lg\, n \in \Theta(n^2)$

$f(n) = O(g(n))$ — Big O

- order of magnitude of $f(n)$ is *less than or equal to* $g(n)$
- Upper bound, e.g. $\Theta(n^2)$ and smaller
- e.g. $100n^2, 100n, nlg\, n + 10n \in O(n^2)$

$f(n) = \Omega(g(n))$ — Big Omega

- order of magnitude of $f(n)$ is *greater or equal to* $g(n)$
- Lower bound, e.g. $\Theta(n^2)$ and larger
- e.g. $n^5, 100n^2, n^3 + n^2lg\, n \in \Omega(n^2)$

Asymptotic Notations

Remember, $O(g(n))$ is a set of functions, but notation is usually written as $T(n) = O(g(n))$, rather than $T(n) \in O(g(n))$

We will stick to ~ (tilde) notation in this course ...

Leading Term Approximation

- Gives an approximate idea of performance
- $10n^2$, $10n^2 + 100n\, lg\, n$, $10n^2 + 10n + 1000 \rightarrow \sim 10n^2$

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Memory

- One must also be aware of memory usage of any program
- 32-bit machines used 4 byte pointers
- 64-bit machines use 8 byte pointers
- `double` takes up 8 bytes
- `bool` takes up only one byte
- Objects and other *complex* items have overheads
- 2D arrays (matrices) take up little over $8N^2$ bytes

Self-assessment Exercise

- Choose a simple problem that has more than one algorithm
- Discuss how algorithmic complexity varies from a naïve to a more sophisticated implementation
- Outcome
  - Understand algorithmic complexity better
  - Inspiration!
  - Technical writing practice!