Asymptotics

Lecture 2

Akshar Varma

5th July, 2023

CS3000 Algorithms and Data

Analysis of Algorithms

The Why. Do the What? But How?

Common Running Times: Example Time

Asymptotic Notation

Common orders

The Limits of Computation

1. Analysis of Algorithms

“As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise — by what course of calculation can these results be arrived at by the machine in the shortest time?” – Charles Babbage, Passages from the Life of a Philosopher

Analysis of Algorithms

- *Life (both time and space) is finite.*
- Algorithms should not be devour too much time or space.
- Otherwise, why bother? The time spent creating/coding them is a waste.
- Question: How to “measure” time or space that an algorithm needs?
- First idea:
  1. Run on a few inputs
  2. Measure the time taken
  3. Measure memory used
  4. Solved. Profit?
- Alternative: *do something else*

2. The Why. Do the What? But How?

“Mathematics and computer science are the two unnatural sciences, the things that are man-made. We get to set up the rules so it doesn’t matter the way the universe works – we create our own universe” – Donald Knuth

Empirical Measurements and Lessons from its Failure

- Let us try the first idea and perform some empirical measurements.
- Failure!!
- Need something machine independent.
- Specifically, constant factors don't matter as much as change due to input size.
- Care about *asymptotic* behavior: how resource requirement *grows* as input $n \to \infty$.
- Because we care about scalability. Most algorithms might do well for small input.
- We want to characterize which work well on large inputs as well.
- Finally, we focus on worst case behavior:
  1. Different algorithms may work better/worse on certain inputs.
  2. Worst case provides a uniform bound to compare against.
  3. Also, we don't need any consensus on what is meant by the "average" input.
- Asymptotic notation formalizes all this intuition in a mathematically rigorous way.

Comparing Runtimes on Two Machines

n (input size)	Machine A (in nanoseconds)	(in seconds)	Machine B (in nanoseconds)	(in seconds)
16	1	0.0	27726	0.0000277
63	3	0.0	41431	0.0000414
250	12	0.0	55215	0.0000552
1000	50	0.0	69078	0.0000691
⋮	⋮	⋮	⋮	⋮
100000	5000	0.0000050	115129	0.0001151
400000	20000	0.0000200	128992	0.0001290
1600000	80000	0.0000800	142855	0.0001429
⋮	⋮	⋮	⋮	⋮
10e12	500000000000	500	299336	0.0002993
200e12	10000000000000	10000	329293	0.0003293
63839e12	3191950000000000	3191950	386951	0.0003870

10000 s $\approx$ 2.7 h and 3191950 s $>$ 1 year.

Empirical Measurements and Lessons from its Failure

- Let us try the first idea and perform some empirical measurements.
- Failure!!
- Need something machine independent.
- Specifically, constant factors don't matter as much as change due to input size.
- Care about *asymptotic* behavior: how resource requirement *grows* as input $n \to \infty$.
- Because we care about scalability. Most algorithms might do well for small input.
- We want to characterize which work well on large inputs as well.
- Finally, we focus on worst case behavior:
  1. Different algorithms may work better/worse on certain inputs.
  2. Worst case provides a uniform bound to compare against.
  3. Also, we don't need any consensus on what is meant by the "average" input.
- Asymptotic notation formalizes all this intuition in a mathematically rigorous way.

Core Ideas

- Assume that $f(n), g(n) > 0$ for large n.
- We want to be able to say $f(n) \sim g(n)$.

  which means $f(n)$ *basically/for our purposes/essentially* behaves like $g(n)$.
- In this case, *basically/for our purposes/essentially/*, means
  1. Ignore constants: $f(n) \sim g(n) \implies f(n) \sim a \cdot g(n)$ for any positive a.
  2. Only large n: the $\sim$ relation is *basically* $\approx$ for large *enough* n.
- The large values idea is always formalized as:
  there exists some n_0 such that for all $n \geq n_0$ blah blah blah holds.
- The $\sim$ symbol is the asymptotic cousin of $=$ we are familiar with when it comes to numbers.
- But this similarity to $=$ is going to break in some ways in the next slides.

3. Common Running Times: Example Time

Linear Time

- Any algorithm which takes time proportional to the input size n (Cn okay).
- Abstract example: Make a single pass through the input.
- Also: making k passes through the input (if k is constant vs. n).
- Concrete example: Finding minimum (and/or maximum) of a given array.

Logarithmic Time

- We don't even need to look at the whole data! Wuhoo!
- Time proportional to: $C \lg n = C' \ln n = C'' \log n$ (properties of logarithms).
- Consider: Find maximum in (unsorted) list. Cannot be done in logarithmic time.
- Why? Since you don't look at all the input, I will be your adversary and make the unseen input have the maximum value.
- Concrete example of logarithmic time: Binary search in sorted array.
- Generally true of logarithmic time: each step throws away some fraction of data.

Constant Time

- Time taken c doesn't depend on n, the input size! What?
- Tends to be very simple operations; these build up to a more complicated algorithm.
- Examples:
  1. return first character of string;
  2. perform one addition ($x + y$);
  3. index into an array (return max/min from a sorted array).
- Useful algorithms don't have constant time algorithms.

Quadratic Time

- Time taken is $c \cdot n^2$ for some constant c.

  Note: $c \cdot n^2 + d \cdot n \leq (c + d)n^2$ is also quadratic time.
- Shows up when we try all pairs.
- Worse than linear, but mostly manageable. However, better is often possible.
- Examples: Naive sorts like insertion sort, selection sort, bubble sort.
- Example: Find closest pair of points in $2d$ space by trying all pairs.
- We'll later use Divide and Conquer to get $n \log n$ algorithms for the above examples.

Loglinear Time/Linearithmic Time

- Time taken is $c \cdot n \log n$ for some constant c.
- Very commonly shows up when quadratic algorithms are improved using divide and conquer.
- Example: sorting algorithms like mergesort, heapsort, quicksort.
- Example: Find closest pair of points in $2d$ space using divide and conquer.

Polynomial Time

- Time taken is $c \cdot n^d$ for some fixed constant d.
- Superset of everything we saw until now.
- Considered “practical”, “efficient” in the theory of algorithms world.
- Generally true, but any $d > 3$ or so *starts* to become bad in practice.
- Usually, when polynomial time algorithm are found, d might be subsequently improved.
- Notable example of this: checking if a number if prime.
- History of the AKS polynomial time primality checking algorithm:
  1. 2002: d is 12.
  2. 2004: Brought down to 10.5 and then down to 7.5.
  3. 2005: Brought down to 6.

Exponential and Factorial Time

- Exponential Time is $c \cdot 2^n$.
- Terrible! For $n > 58$, time is longer than age of the universe.
- Usually shows up when all combinations are tried in a brute-force manner
- Notable example: Solving Travelling Salesman using Dynamic Programming.

- Factorial Time is $c \cdot n!$.
- Terrible! Starts approaching age of universe with n in the 20s.
- Usually shows up when all re-orderings are tried.
- Notable example: Solving Travelling Salesman via brute force.

Simple Exercises

Try to determine the class that an algorithm for the following belongs to:

- Count occurrences of the word “foo” in a file. $n =$ length of file.
  Linear
- Brute force crack a pass code made using $\{0, 1, \ldots, 9\}$. $n =$ length of code.
  10^n
- Check for plagiarism by comparing each pair of submissions. $n =$ number of submissions.
  n^2
- Play a guessing game with `YES/NO` questions.
  Each answer reducing possible solutions by half.
  $n =$ all possible answers from which guess must be made.
  $\lg n$
- Brute force crack a pattern code on n points.
  A pattern code is a specific ordering through (potentially) all n points.
  $n!$

4. Asymptotic Notation

The Five Types of Relationships

Notation	Description	Intuition	Relation
$O(\cdot)$	not worse than	it won't take longer than	$\preccurlyeq$
$\Omega(\cdot)$	not better than	it takes at least so much	$\succcurlyeq$
$\Theta(\cdot)$	Exact analysis	this is how long it takes	$\asymp$
$o(\cdot)$	Strict upper bound	it takes *strictly less* than	$\prec$
$\omega(\cdot)$	Strict lower bound	it takes *strictly more* than	$\succ$

$$\exists k > 0,\ \exists n_0,\ \forall n > n_0 \text{ such that:} \quad |f(n)| \leq k \cdot g(n)$$

$$\limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Big-Omega Notation: $f(n) = \Omega(g(n))$ f bounded below by g (asymptotically)

$$\exists k > 0,\ \exists n_0,\ \forall n > n_0 \text{ such that:} \qquad f(n) \geq k \cdot g(n)$$

$$\limsup_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

Theta Notation: $f(n) = \Theta(g(n))$ f bounded above and below by g (asymptotically)

$\exists k_1,\ k_2 > 0,\ \exists n_0,\ \forall n > n_0$ such that: $\quad k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$

$$\limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty \text{ and } \limsup_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

$$\forall k > 0,\ \exists n_0,\ \forall n > n_0 \text{ such that:} \quad |f(n)| < k \cdot g(n)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

$$\forall k > 0,\ \exists n_0,\ \forall n > n_0 \text{ such that:} \quad f(n) > k \cdot g(n)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

5. Common orders

“The greatest shortcoming of the human race is our inability to understand the exponential function.”

— Albert Allen Bartlett

Common Time Complexities

$$c \prec \alpha(n) \prec \log^* n \prec \log\log n \prec \log n \prec \mathsf{poly}(\log n) \asymp (\log n)^a$$
$$\prec n^{0<b<1} \prec n \prec n \log^* n \prec n \log n \prec n\mathsf{poly}(\log n) \prec n^2 \prec n^3 \prec \mathsf{poly}(n) \prec$$
$$\preccurlyeq_{\text{hmm..}} 2^{\mathsf{poly}(\log n)} \prec_{\text{give up?}} 2^{o(n)} \prec_{\text{yep. I'm out.}} 2^{O(n)} \prec 2^{\mathsf{poly}(n)} \prec n! \prec 2^{2^{\mathsf{poly}(n)}}$$

The ones you should just grok; $a, b, c > 0$ and $d > 1$

$$c \prec (\log n)^b \prec n^a \prec d^n \prec n!$$

Exponentials $\geq$ Polynomials [Optional: Not necessary, but is neat]

Theorem (Exponentials grow faster than all Polynomials)

For any $a > 1, b > 0$ we have that $\exists c > 0$ and some n_0 such that $\forall n \geq n_0$: $a^n \geq c \cdot n^b$.

Proof.

1. First, prove that $2^n \geq n$ for all $n \geq 0, n \in \mathbb{Z}$ (using induction).

$$2^{n+1} = 2 \cdot 2^n \geq 2 \cdot n \geq n + 1, \text{ for all } n \geq 1$$

2. $2^n = \left(2^{n/c}\right)^c \geq (n/c)^c$ for any $c > 0$.
3. For any $b > 0$ and $n \geq (b+1)^{b+1} = n_0$,

$$2^n \geq (n/(b+1))^{b+1} = n/(b+1)^{b+1} \cdot n^b \geq n^b$$

4. Let $a > 1$, then: $a^n = \left(2^{\lg a}\right)^n = (2)^{n \cdot \lg a} \geq \cdot (n \cdot \lg a)^b = (\lg a)^b \cdot n^b$
5. Thus, for any $a > 1, b > 0$ we have a $c > 0$ with $n_0 = (b+1)^{b+1}$ such that $a^n \geq c \cdot n^b$.

Polynomials $\geq$ Logarithms [Optional: Not necessary, but is neat]

Theorem (Polynomials grow faster than all Logarithms)

For any $a, b > 0$ we have that $\exists c > 0$ and some n_0 such that $\forall n \geq n_0$: $\quad n^a \geq c \cdot (\lg n)^b$.

Proof.

1. If $a > b$, then this follows from the fact that $\log_d x \leq x$ for all $d, x > 0$ ($c = 1$ works).
2. Consider $a < b$

$$\log_a n \leq n \implies \left(\frac{\lg n}{\lg a}\right)^b \leq n^b \implies \frac{(\lg n)^b}{n^{b-a} \cdot \lg a} \leq n^a$$

3. Note: $\lim_{n \to \infty} n^{b-a} \to \infty$.
4. Since $\lg a$ is finite, $\exists n_0$ such that $n^{b-a} > \lg a$ for all $n \geq n_0$.
5. Choosing n_0 accordingly and setting $c = 1$ completes the proof.

■

Factorials $\geq$ Exponentials (via Sterling's Approximation)

[Optional: Not necessary, but is neat]

$$\ln n! = \sum_{j=1}^{n} \ln j$$

$$\sum_{j=1}^{n} \ln j \approx \int_{1}^{n} \ln x dx = n \ln n - n + 1$$

$$n! \sim \left(\frac{n}{e}\right)^n \cdot e$$

$$\left(e^2\right)^n \prec n^n \implies e^n \prec \left(\frac{n}{e}\right)^n$$

Asymptotic Notation Summary

$o(\cdot)$	$O(\cdot)$	$\Theta(\cdot)$	$\Omega(\cdot)$	$\omega(\cdot)$
$\prec$	$\preccurlyeq$	$\asymp$	$\succcurlyeq$	$\succ$

The ones you should just grok; $a, b, c > 0$ and $d > 1$

$$c \prec (\log n)^b \prec n^a \prec d^n \prec n!$$

6. The Limits of Computation

“The greatest shortcoming of the human race is our inability to understand the exponential function.”

— Albert Allen Bartlett

Limits of Computation

- We've seen how we analyze the how long algorithms take and notation to quantify it.
- The landscape there ended at 2^{2^n} since that is already very impractical.
- However, if we wanted to keep going? How far can we go?
- What if time didn't matter, only whether something is possible or not?
- Turns out there are things we cannot figure out.
- First, we simplify things: only look at `YES/NO` (mathematical) problems.
- Instead of sorting a list, return `True/False` if the list is sorted or not.
- Instead of finding index of x in array, return `True/False` if x is in array.
- Given a program and some input, will the program ever halt or not?

Undecidability of the Halting Problem

Theorem (Halting Problem is Undecidable)

No general algorithm exists which can solve the following problem:
Given the description of an arbitrary program and a finite input, decide whether the program will eventually halt or will it run forever.

Proof.

- Let us assume that algorithm H determines if P with input X halts or not.
- Consider the following program T:
  1. Run H with T as input.
  2. If H returns "Halts", then go into infinite loop.
  3. Otherwise: Halt.
- Note that T is designed in a way that falsifies the output of H.
- Thus, no such H can exist which answers the halting problem correctly for all programs and inputs.

■

Landscape of All Mathematical Problems