CSE 421: Intro Algorithms

2: Analysis

Larry Ruzzo
Elaine presented an introduction to analysis and “big-O” on the whiteboard. Her notes are linked from the 421 web page.

The Powerpoint slides below supplement that (plus a bit of new material, especially “little-o”).
Why big-O: measuring algorithm efficiency
What’s big-O: definition and related concepts
Reasoning with big-O: examples & applications
  - polynomials
  - exponentials
  - logarithms
  - sums
Polynomial Time
Why big-O: measuring algorithm efficiency
Our correct TSP algorithm was incredibly slow
   No matter what computer you have
As a 2\textsuperscript{nd} example, for large problems, mergesort beats insertion sort – \( n \log n \) vs \( n^2 \) matters a lot
   Even tho the alg is more complex & inner loop is slower
   No matter what computer you have
We want a general theory of “efficiency” that is
   Simple
   Objective
   Relatively independent of changing technology
   Measures \textit{algorithm}, not code
   But still \textit{predictive} – “theoretically bad” algorithms should be bad in practice and vice versa (usually)
defining efficiency

“Runs fast on typical real problem instances”

Pro:
  sensible, bottom-line-oriented

Con:
  moving target (diff computers, compilers, Moore’s law)
  highly subjective (how fast is “fast”? What’s “typical”?)
defining efficiency

“Runs fast on a specific suite of benchmarks”

Pro:
  again sensible, bottom-line-oriented

Con:
  all the problems above
  are benchmarks representative
  algorithms can be “tuned” to the well-known benchmarks
  generating/maintaining benchmarks is a burden
  benchmarking a new algorithm is a lot of work
Instead:

a) Give up on detailed timing, focus on scaling
   Nanoseconds matter of course, but we often want to push to bigger problems tomorrow than we can solve today, so an algorithm that scales as $n^2$, say, will very likely beat one that grows as $2^n$ or $n^{10}$ or even $n^3$, even if the later uses fewer nanoseconds for today’s $n$.

b) Give up on “typical,” focus on worst case behavior
   Over all inputs of size $n$, how fast are we on the worst? Removes all debate about “typical” / “average.”

Overall, these yield a big win in terms of technology independence, ease of analysis, robustness
The time complexity of an algorithm associates a number $T(n)$, the worst-case time the algorithm takes, with each problem size $n$.

Mathematically,

$$T: N^+ \rightarrow R$$

i.e., $T$ is a function mapping positive integers (problem sizes) to positive real numbers (number of steps).

"Reals" so, e.g., we can say $\sqrt{n}$ instead of $\lfloor \sqrt{n} \rfloor$

"Positive" so, e.g., log($n$) and $2^n/n$ aren’t problematic
computational complexity

Problem size

Time

T(n)
why worst-case analysis?

Appropriate for time-critical applications
  E.g. avionics, nuclear reactors

Unlike Average-Case, no debate over the right definition
  If worst $\gg$ average, then (a) alg is doing something pretty subtle, & (b) are hard instances really that rare?

Analysis often much easier

Result is often representative of “typical” problem instances

Of course there are exceptions…
Asymptotic growth rate, i.e., characterize growth rate of worst-case run time as a function of problem size, up to a constant factor, e.g. $T(n) = O(n^2)$

Why not try to be more precise?

- Average-case, e.g., is hard to define, analyze
- Technological variations (computer, compiler, OS, …) easily 10x or more
- Being more precise is much more work

A key question is “scale up”: if I can afford this today, how much longer will it take when my business is 2x larger? (E.g. today: $cn^2$, next year: $c(2n)^2 = 4cn^2$ : 4 x longer.) Big-O analysis is adequate to address this.
Big-O: a math notation for an upper bound on the asymptotic growth rate of a function

E.g., if \( f(n) = \text{value of the n}\text{th} \text{ prime} \), \( f(n) = O(n \log n) \)

In CS, commonly used to describe run time of algorithms, usually worst case run time, but could be other run time functions.

E.g., for Quicksort

\[
\begin{align*}
T_{\text{best}}(n) &= O(n) \\
T_{\text{avg}}(n) &= O(n \log n) \\
T_{\text{worst}}(n) &= O(n^2)
\end{align*}
\]
What’s big-O: definition and related concepts
Given two functions $f$ and $g$: $\mathbb{N}^+ \rightarrow \mathbb{R}$

- $f(n)$ is $O(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\leq c \cdot g(n)$

- $f(n)$ is $\Omega(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\geq c \cdot g(n)$

- $f(n)$ is $\Theta(g(n))$ iff there are constants $c_1, c_2 > 0$ so that eventually always $c_1 g(n) \leq f(n) \leq c_2 g(n)$

“Eventually always $P(n)$” means “$\exists n_0$ s.t. $\forall n > n_0$ $P(n)$ is true.” I.e., there can be exceptions, but only for finitely many “small” values of $n$. 

---

**Upper Bounds**

**Lower Bounds**

**Both**
computational complexity

Problem size

Time

$T(n)$
Example: $T(n) = \Theta(n \log_2 n)$
since for all problem sizes $n > n_0$, the worst case run time $T(n)$ is between $n \log_2 n$ and $2n \log_2 n$. 

![Diagram showing computational complexity](image-url)
A typical program with initialization and two nested loops might have runtime something like this
If \( T(n) = n^2 + 30n + 5000 \), then \( T(n) = \Theta(n^2) \), since for all \( n \geq 135 \), we have \( n^2 \leq T(n) \leq 1.5 \ n^2 \).
Reasoning with big-O: examples & applications

polynomials
exponentials
logarithms
sums
Show $10n^2 - 16n + 100$ is $O(n^2)$:

$10n^2 - 16n + 100 \leq 10n^2 + 100$

$= 10n^2 + 10^2$

$\leq 10n^2 + n^2 = 11n^2$ for all $n \geq 10$

$\therefore O(n^2)$ [and also $O(n^3)$, $O(n^4)$, $O(n^{2.5})$, ...]
Show $10n^2 - 16n + 100$ is $\Omega(n^2)$:

$10n^2 - 16n + 100 \geq 10n^2 - 16n$

$\geq 10n^2 - n^2 = 9n^2$ for all $n \geq 16$

$\therefore \Omega(n^2)$ [and also $\Omega(n)$, $\Omega(n^{1.5})$, $\ldots$]

Therefore also $10n^2 - 16n + 100$ is $\Theta(n^2)$

[but not $\Theta(n^{1.999})$ or $\Theta(n^{2.001})$]
asymptotic bounds for polynomials

Polynomials:
\[ p(n) = a_0 + a_1 n + \ldots + a_d n^d \] is \( \Theta(n^d) \) if \( a_d > 0 \)

Proof:
\[
\begin{align*}
p(n) &= a_0 + a_1 n + \ldots + a_d n^d \\
&\leq |a_0| + |a_1| n + \ldots + a_d n^d \\
&\leq |a_0| n^d + |a_1| n^d + \ldots + a_d n^d \\
&= c n^d, \text{ where } c = (|a_0| + |a_1| + \ldots + |a_{d-1}| + a_d)
\end{align*}
\]
\[
\therefore \quad p(n) = O(n^d)
\]

Exercise: show that \( p(n) = \Omega(n^d) \)

Hint: this direction is trickier; focus on the “worst case” where all coefficients except \( a_d \) are negative.
another example of working with $O$-$\Omega$-$\Theta$ notation

Example: For any $a$, and any $b > 0$, $(n+a)^b$ is $\Theta(n^b)$

$(n+a)^b \leq (2n)^b$ for $n \geq |a|$

$= 2^b n^b$

$= cn^b$ for $c = 2^b$

so $(n+a)^b$ is $O(n^b)$

$(n+a)^b \geq (n/2)^b$ for $n \geq 2|a|$ (even if $a < 0$)

$= 2^{-b} n^b$

$= c'n^b$ for $c' = 2^{-b}$

so $(n+a)^b$ is $\Omega(n^b)$
more examples: tricks for sums

Example: $\sum_{1 \leq i \leq n} i = \Theta(n^2)$

Proof:

(a) An upper bound: each term is $\leq$ the max term

$$\sum_{1 \leq i \leq n} i \leq \sum_{1 \leq i \leq n} n = n^2 = O(n^2)$$

(b) A lower bound: each term is $\geq$ the min term

$$\sum_{1 \leq i \leq n} i \geq \sum_{1 \leq i \leq n} 1 = n = \Omega(n)$$

This is valid, but a weak bound.

Better: pick a large subset of large terms

$$\sum_{1 \leq i \leq n} i \geq \sum_{n/2 \leq i \leq n} n/2 \geq \left\lfloor n/2 \right\rfloor^2 = \Omega(n^2)$$

E.g.: for $i = 1..n$ {
  for $j=1$ to $i$ {
    ...
  }
}
Transitivity.

If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Proofs are left as exercises.
For all $r > 1$ (no matter how small) and all $d > 0$, (no matter how large)

$$n^d = O(r^n)$$

In short, every exponential grows faster than every polynomial!

(proof below)
Example: For any $a, b > 1$ \( \log_a n \) is \( \Theta(\log_b n) \)

\[
\log_a b = x \text{ means } a^x = b
\]

\[
a^{\log_a b} = b
\]

\[
(a^{\log_a b})^{\log_b n} = b^{\log_b n} = n
\]

\[
(\log_a b)(\log_b n) = \log_a n
\]

\[
c \log_b n = \log_a n \text{ for the constant } c = \log_a b
\]

So:

\[
\log_b n = \Theta(\log_a n) = \Theta(\log n)
\]

Corollary: base of a log factor is usually irrelevant, asymptotically. E.g. “\( O(n \log n) \)” [but \( n^{\log_2 8} \neq O(n^{\log_8 8}) \)]
Logarithms:
For all $x > 0$, (no matter how small) $\log n = O(n^x)$

log grows slower than every polynomial
domination: little-o

f(n) is \( o(g(n)) \) iff \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \) that is, \( g(n) \) dominates \( f(n) \)

If \( a \leq b \) then \( n^a \) is \( O(n^b) \)

If \( a < b \) then \( n^a \) is \( o(n^b) \)

\( f(n) = O(g(n)) \) vs \( f(n) = o(g(n)) \) are analogs to \( \leq \) vs \( < \)

Note:
if \( f(n) \) is \( \Theta(g(n)) \) then it cannot be \( o(g(n)) \)
working with little-o

\[ n^2 = o(n^3) \text{ [Use algebra]:} \]

\[
\lim_{n \to \infty} \frac{n^2}{n^3} = \lim_{n \to \infty} \frac{1}{n} = 0
\]

\[ n^3 = o(e^n) \text{ [Use L’Hospital’s rule 3 times]:} \]

\[
\lim_{n \to \infty} \frac{n^3}{e^n} = \lim_{n \to \infty} \frac{3n^2}{e^n} = \lim_{n \to \infty} \frac{6n}{e^n} = \lim_{n \to \infty} \frac{6}{e^n} = 0
\]
For all $r > 1$ (no matter how small) and all $d > 0$, (no matter how large)
$n^d = O(r^n)$
$n^d = o(r^n)$, even

Exercise: prove this, using tricks from previous slide

In short, every exponential grows faster than every polynomial!
Given two functions $f(n)$ and $g(n)$, if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 
    c & \text{for some constant } c > 0 \\
    0 & \text{for some constant } c > 0
\end{cases},$$

then

$$\begin{cases} 
    f(n) = \Theta(g(n)) \\
    f(n) = o(g(n)) \quad [\Rightarrow O(g(n))]
\end{cases},$$

respectively.

Inconclusive if the limit doesn’t exist. E.g., no limit for $f/g$ at right, but $g(n) \leq f(n) = O(f(n))$
big-theta, etc. are not always “nice”

\[ f(n) = \begin{cases} 
  n^2, & \text{n even} \\
  n, & \text{n odd} 
\end{cases} \]

\( f(n) \neq \Theta(n^a) \) for any \( a \).

Fortunately, such nasty cases are rare

\( n \log n \neq \Theta(n^a) \) for any \( a \), either, but at least it’s simpler.
“Theorem”: \( \sum_{1 \leq i \leq n} i = O(n) \)

“Proof:” (by induction on \( n \))

basis: \( \sum_{1 \leq i \leq 1} i = 1 = O(1) \)

induction step:

\[
\sum_{1 \leq i \leq n} i = (\sum_{1 \leq i \leq n-1} i) + n
= O(n-1) + n \quad (\text{by ind. hyp.})
= O(n)
\]

Q. Where’s the flaw??

A. Never use “big-O” like this in an induction; instead, explicitly show the implicit constant “c”; in the above “proof,” you’ll see “c” become “c+1”…”

FALSE!
“One-Way Equalities”

2 + 2 is 4
2 + 2 = 4
4 = 2 + 2

2n² + 5 n is O(n³)
2n² + 5 n = O(n³)
O(n³) = 2n² + 5 n

All dogs are mammals
All mammals are dogs

Bottom line:
OK to put big-O in R.H.S. of equality, but not left.

Better, but less common, notation: T(n) ∈ O(f(n)).
I.e., O(f(n)) is the set of all functions that grow no more rapidly than some constant times f.

Replace “=” by “∈” or “⊆” as appropriate: e.g.:
2n² + 5 n ∈ O(n²) ⊆ O(n³)
Polynomial Time
the complexity class P: polynomial time

P: The set of problems solvable by algorithms with running time $O(n^d)$ for some constant $d$

(d is a constant independent of the input size $n$)

*Nice scaling property:* there is a constant $c$ s.t. doubling $n$, time increases only by a factor of $c$.

(E.g., $c \sim 2^d$)

Contrast with exponential: For any constant $c$, there is a $d$ such that $n \rightarrow n+d$ increases time by a factor of more than $c$.

(E.g., $c = 100$ and $d = 7$ for $2^n$ vs $2^{n+7}$)
polynomial vs exponential growth

$2^{2n}$

$2^{n/10}$

$1000n^2$
Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>$&lt; 1$ sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

not only get very big, but do so abruptly, which likely yields erratic performance on small instances.
Next year’s computer will be 2x faster. If I can solve problem of size $n_0$ today, how large a problem can I solve in the same time next year?

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Size Increase</th>
<th>E.g. $T=10^{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>$n_0 \rightarrow 2n_0$</td>
<td>$10^{12} \rightarrow 2 \times 10^{12}$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$n_0 \rightarrow \sqrt{2} n_0$</td>
<td>$10^6 \rightarrow 1.4 \times 10^6$</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>$n_0 \rightarrow 3\sqrt{2} n_0$</td>
<td>$10^4 \rightarrow 1.25 \times 10^4$</td>
</tr>
<tr>
<td>$2^n /10$</td>
<td>$n_0 \rightarrow n_0+10$</td>
<td>$400 \rightarrow 410$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$n_0 \rightarrow n_0+1$</td>
<td>$40 \rightarrow 41$</td>
</tr>
</tbody>
</table>
why “polynomial”?

Point is not that $n^{2000}$ is a nice time bound, or that the differences among $n$ and $2n$ and $n^2$ are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

“My problem is in P” is a starting point for a more detailed analysis

“My problem is not in P” may suggest that you need to shift to a more tractable variant, or otherwise readjust expectations
Big $\mathcal{O}$ is a math notation defining an upper bound on growth rate of a function (typically a function lacking a simple analytic formula.)

In CS, that function is often the worst case run time of some algorithm (as a function of input size, $n$, where “worst case” means max time over all inputs of size $n$.)

BUT, it can also be used for other functions, like best- or average-case time/space/…, so be clear/careful re defn.

Big $\Omega$ is analogous math notation for lower bounds

Big $\Theta$: upper and lower bounds simultaneously

These notations deliberately define growth rate only up to a (hidden) constant factor, essentially because (a) scaling matters more than the constant, and (b) the constant is strongly technology-dependent (language, code, compiler, processor, …) making it much more work to pin down.
So, a typical initial goal for algorithm analysis is to find a reasonably tight, asymptotic, bound on worst case running time as a function of problem size. Usually upper bound is a tight upper bound, i.e., $\Theta$ if possible.

This is rarely the last word, but often helps separate good algorithms from blatantly poor ones – so you can concentrate on the good ones! As one important example, poly time algorithms are almost always preferable to exponential time ones.