Analyzing Algorithms, continued.

Properties of Big oh

- max rule: \( o(f(n) + g(n)) \) is \( O(\max\{f(n), g(n)\}) \)
- transitivity: \( f(n) \leq O(g(n)) \), \( g(n) \leq O(h(n)) \)
  \[ \Rightarrow f(n) \leq O(h(n)) \]

Further Definitions

- \( f(n) \) is \( \Omega(g(n)) \) "omega" if there exists constants \( c, n_0 \) s.t.
  \[ f(n) \geq c \cdot g(n) \ \forall \ n \geq n_0 \]
- \( f(n) \) is \( \Theta(g(n)) \) "theta" if \( f(n) \) is \( O(g(n)) \) and \( \Omega(g(n)) \)
  - gives an exact (asymptotic) bound.
- \( f(n) \) is \( o(g(n)) \) "little oh" if for any constant \( c > 0 \),
  there exists a constant \( n_0 \) s.t.
  \[ f(n) \leq c \cdot g(n) \ \forall \ n \geq n_0 \]
  equivalently \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \) (for \( f, g : \mathbb{N} \to \mathbb{R}^+ \))

Comparing Algorithms using Asymptotic Analysis

Suppose the worst case run time of

algorithm A is \( O(n^2) \)

algorithm B is \( O(n \log n) \)

Which is better? - We can't know!

This is like \( x \leq 5 \) \( y \leq 10 \), which is smaller?

To compare algorithms, we need tight bounds

\( \Theta(n^2) \) vs \( \Theta(n \log n) \) better.
0 is like $\leq$
0 \ldots < 
\theta \ldots =

One difference: we can compare any 2 numbers $x \leq y$ or $y \leq x$

But there are functions $f, g$ s.t. $f(n)$ is not $O(g(n))$ and $g(n)$ is not $O(f(n))$. Find some.

**Challenge**

Can you find such $f(n)$s using just $+, \times, \text{exponentiation, log}$?
*not allowed to use $\sin, \log, \Gamma$
*not allowed to say $f(n) = \frac{\pi}{2} - n$ even

Ref. G.H. Hardy. See Concrete Math, Graham, Knuth, Patashnik.
Typical run-times and how they compare

\[
\begin{align*}
&\log n \quad \text{binary search} \\
&n \quad \text{find max} \\
&n \log n \quad \text{sorting} \\
&n^2 \quad \text{insertion sort} \\
&n^3 \quad \text{multiplying two } n \times n \text{ matrices} \\
&2^n \quad \text{try all subsets} \\
&n! \quad \text{try all orderings of a set (e.g. Travelling Salesman)}
\end{align*}
\]

Ordering, where \( f(n) \leq g(n) \) means \( f(n) \in O(g(n)) \)

\[
1 \leq \log \log n \leq \log n \leq \log^2 n \leq \sqrt{n} \leq n
\leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!
\]

Also \( n^a \in O(n^b) \) \( b > a > 0 \)

\[
\log^a n \in O(n^b) \quad b > 0
\]

\( n^a \in O(2^n) \)

Skiena Ch. 2 is a good reference

Examples using above:

• \( n \log n \) vs \( n \sqrt{n}/2 \)
  
  From above \( \log n \in O(\sqrt{n}) \)
  
  So \( n \log n \in O(n \sqrt{n}/2) \)

• \( \log(\sqrt{n}) \) vs \( \log n \)
  
  \( \log(\sqrt{n}) = \log n^{1/2} = \frac{1}{2} \log n \in \Theta(\log n) \)
Sometimes we will analyze an algorithm's run-time in terms of several parameters.

**EX1.** Graph with \( n \) vertices and \( m \) edges

\( m = O(n^2) \) but \( O(n+m) \) can be better than \( O(n^2) \)

**EX2.** Input and output size

E.g., Jarvis' March: \( O(n \cdot h) \) \( n = \) input size, \( h = \) output size.

**Defn** \( f, g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^+ \)

\( f(n,m) \) is \( O(g(n,m)) \) if there exists constants \( c, n_0, m_0 \) s.t.

\[ f(n,m) \leq c \cdot g(n,m) \text{ for all } n \geq n_0, m \geq m_0 \]

**Summary** We analyze algorithms by analyzing the asymptotic rate of growth of the worst case run-time. Does it really matter?
As computers get faster it is even more important to have improved technology on several polynomial and exponential time algorithms.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$</th>
<th>$\lg n$</th>
<th>$n$</th>
<th>$n \lg n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$\mu s$</td>
<td>0.003</td>
<td>0.01</td>
<td>0.033</td>
<td>0.1</td>
<td>1</td>
<td>3.63</td>
</tr>
<tr>
<td>20</td>
<td>$\mu s$</td>
<td>0.004</td>
<td>0.02</td>
<td>0.086</td>
<td>0.4</td>
<td>1 ms</td>
<td>77.1</td>
</tr>
<tr>
<td>30</td>
<td>$\mu s$</td>
<td>0.005</td>
<td>0.03</td>
<td>0.147</td>
<td>0.9</td>
<td>1 s</td>
<td>$8.4 \times 10^{15}$ yrs</td>
</tr>
<tr>
<td>40</td>
<td>$\mu s$</td>
<td>0.005</td>
<td>0.04</td>
<td>0.213</td>
<td>1.6</td>
<td>18.3 min</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>$\mu s$</td>
<td>0.006</td>
<td>0.05</td>
<td>0.282</td>
<td>2.5</td>
<td>13 days</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>$\mu s$</td>
<td>0.007</td>
<td>0.1</td>
<td>0.644</td>
<td>10</td>
<td>4 $\times 10^{13}$ yrs</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>$\mu s$</td>
<td>0.010</td>
<td>1.00</td>
<td>9.966</td>
<td>1 ms</td>
<td>100 ms</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>$\mu s$</td>
<td>0.013</td>
<td>10</td>
<td>130</td>
<td>10 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>$\mu s$</td>
<td>0.017</td>
<td>10 ms</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>$\mu s$</td>
<td>0.020</td>
<td>1 ms</td>
<td>19.93</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td>$\mu s$</td>
<td>0.023</td>
<td>0.01 sec</td>
<td>0.23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000,000</td>
<td>$\mu s$</td>
<td>0.027</td>
<td>0.10 sec</td>
<td>2.66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>$\mu s$</td>
<td>0.030</td>
<td>1 sec</td>
<td>29.90</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4: Growth rates of common functions measured in nanoseconds

Size of Largest Problem Instance

<table>
<thead>
<tr>
<th>Solvable in 1 Hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>With present computer</td>
</tr>
<tr>
<td>$n$</td>
</tr>
<tr>
<td>$n^2$</td>
</tr>
<tr>
<td>$n^3$</td>
</tr>
<tr>
<td>$n^5$</td>
</tr>
<tr>
<td>$2^n$</td>
</tr>
<tr>
<td>$3^n$</td>
</tr>
</tbody>
</table>

Figure 1.3 from *Computers and Intractability: A Guide to the Theory of NP-Completeness*, by Garey and Johnson: Effect of improved technology on several polynomial and exponential time algorithms.
Algorithmic Paradigms
1. reductions
2. divide and conquer
3. greedy
4. dynamic programming

Reductions
Often, you can use known algorithms to solve new problems. (Don’t reinvent the wheel.)
Example. 2-SUM and 3-SUM

2-SUM
Input: array \( A[i..n] \) of numbers and target number \( m \)
Find \( i, j \) s.t. \( A[i] + A[j] = m \) (if they exist)
Algorithm 1
\[
\begin{align*}
&\text{for } i = 1..n \\
&\quad \text{for } j = i..n \\
&\quad \quad \text{if } A[i] + A[j] = m \text{ SUCCESS} \\
&\quad \text{end} \\
&\text{end} \\
&\text{FAIL}
\end{align*}
\]
Run time \( O(n^2) \)
Algorithm 2
Sort \( A \).
For each \( i \) do binary search for \( m - A[i] \)
\[
\begin{align*}
&O(n \log n) + O(n \log n) = O(n \log n) \\
&\text{sort} \quad \text{n binary searches}
\end{align*}
\]
Algorithm 3  Improve the 2nd phase

sorted array $A$

$$\begin{array}{c}
12 \\ 3 \\ 5 \\ 11 \quad 12 \quad 20 \quad 22
\end{array}$$

$\uparrow$  \hspace{1cm} $\uparrow$  \hspace{1cm} $\uparrow$

$i$  \hspace{1cm} $j$  \hspace{1cm} $j$

Target: $m = 23$


24 - too big. Decrease $j$.

22 - too small. Increase $i$.

23 - just right!

$i \leftarrow 1$; $j \leftarrow n$
while $i \leq j$

$S \leftarrow A[i] + A[j]$

if $S > m$

$j \leftarrow j - 1$

else if $S < m$

$i \leftarrow i + 1$

else SUCCESS

end

FAIL

Correctness invariant:

if there is a solution

$i^* \leq j^*$  then

$i^* \geq i$,  $j^* \leq j$

EX. Give more details

Run time

$O(n)$ (after sorting)

3-Sum

Input array $A[i..n]$ of numbers

and target number $m$.


(allow $i=j$ etc.)
We can reduce 3-SUM to 2-SUM (multiple calls to it)
So run 2-SUM with target \( m - A[k] \) for each \( k \).
Run-time \( \frac{O(n \cdot n \log n)}{\# k's} = O(n^2 \log n) \) # 2-SUM

Look more closely:
2-SUM was \( \frac{O(n \log n)}{\text{sort}} + \frac{O(n)}{\text{Algorithm 2}} \)
We only need to sort once
This gives \( O(n \log n) + O(n^2) = O(n^2) \).

Is there a faster algorithm for 3-SUM?
For many years people thought NO, but now
there are faster algorithms (2014, 2017).