defining efficiency

“Runs fast on typical real-world problem instances”

- Pro: sensible, straight to the point
- Cons:
  - moving target (different computers, architectures, compilers, Moore’s law)
  - highly subjective (how fast is “fast”? what is “typical”?)

Instead, we:

- Give up on detailed timing, focus on scaling.
- Give up on “typical.” Focus on worst-case behavior.
defining efficiency: the RAM model

- RAM = Random Access Machine

- Time $\approx$ # of instructions executed in an ideal assembly language
  - each simple operation (+, *, -, =, if, call) takes one time step
  - each memory access takes one time step

complexity analysis

- Problem size $N$

  - Worst-case complexity: max # steps algorithm takes on any input of size $N$

  - Average-case complexity: average # steps algorithm takes on inputs of size $N$

complexity

- The complexity of an algorithm associates a number $T(N)$, the worst/average-case/best time the algorithm takes, with each problem size $N$.

- Mathematically,
  - $T: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ is a function that maps positive integers giving problem size to positive real numbers giving number of steps
complexity

Problem size $N$

asymptotic growth rates

Given two functions $f, g : \mathbb{N} \to \mathbb{R}_+$

- $f(n)$ 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 it is both $O(g(n))$ and $\Omega(g(n))$

Upper bounds $\leq$

Lower bounds $\geq$

example

Show that $10n^2 - 16n + 100$ is $\Omega(n^2)$
asymptotic bounds for polynomials

\[ p(n) = a_0 + a_1 n + \cdots + a_d n^d \text{ is } \Theta(n^d) \text{ if } a_d > 0 \]

asymptotics of...

\[ \sum_{i=1}^{n} i \]

properties

- transitivity
- additivity

logarithmic vs. polynomial vs. exponential

\[ \log_b(n) = o(n^a) = o(c^n) \]

for all constants \( a, b, \) and \( c \)
scaling

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^{18} years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>n</th>
<th>log_{10} 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>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>4 sec</td>
</tr>
<tr>
<td>50</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>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>very long</td>
</tr>
<tr>
<td>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>
</tr>
<tr>
<td>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>
</tr>
<tr>
<td>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>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,700 years</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

polynomial time = “efficient”

\( P = \text{class of problems solvable by algorithms running in polynomial time, i.e. } O(n^d) \text{ for some constant } d \)

scaling: When input size doubles, running time increases by a constant factor.

vs exponential

interval scheduling

- **Input.** Set of jobs with start times and finish times.
- **Goal.** Find maximum cardinality subset of mutually compatible jobs.

interval scheduling

- **Input.** Set of jobs with start times and finish times.
- **Goal.** Find maximum cardinality subset of mutually compatible jobs.
**weighted interval scheduling**

- **Input.** Set of jobs with start times, finish times, and weights.
- **Goal.** Find maximum weight subset of mutually compatible jobs.

![Weighted Interval Scheduling Diagram]

**bipartite matching**

- **Input.** Bipartite graph.
- **Goal.** Find maximum cardinality matching.

![Bipartite Matching Diagram]

**independent set**

- **Input.** Graph.
- **Goal.** Find maximum cardinality independent set.

![Independent Set Diagram]
representative problems

- Variations on a theme: independent set.

- **Interval scheduling:** $O(n \log n)$ greedy algorithm.

- **Weighted interval scheduling:** $O(n \log n)$ dynamic programming algorithm.

- **Bipartite matching:** $O\left(n^k\right)$ max-flow based algorithm.

- **Independent set:** NP-complete.

- **Competitive facility location:** PSPACE-complete
  
  (see book)