Chapter 2
Basics of Algorithm Analysis
2.1 Computational Tractability

Time complexity

Space complexity (not discussed, but later in this class)
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
Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
  - Typically takes $2^N$ time or worse for inputs of size $N$.
  - Unacceptable in practice.

Undecidable problems. Wang tiles (or Wang dominoes)

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor $C$.

There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $c N^d$ steps.

Def. An algorithm is poly-time if the above scaling property holds.

Choose $C = 2^d$
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.
- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
Worst-Case Polynomial-Time

**Def.** An algorithm is **efficient** if its running time is polynomial.

**Justification:** It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

**Exceptions.**

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

\[\text{simplex method}\]
Why It Matters

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>
2.2 Asymptotic Order of Growth
Asymptotic Order of Growth

**Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

**Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

**Tight bounds.** $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

**Ex:** $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.
Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Not transitive:
  - $f(n) = 5n^3; \ g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

- Statement doesn't "type-check."
- Use $\Omega$ for lower bounds.
Properties

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 $g = \Theta(h)$ then $f + g = \Theta(h)$.
Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant $d$ independent of the input size $n$.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

- can avoid specifying the base

Logarithms. For every $x > 0$, $\log n = O(n^x)$.

- log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

- every exponential grows faster than every polynomial
2.4 A Survey of Common Running Times
Linear Time: $O(n)$

Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

```
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```
Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

![Diagram of merge process]

### Claim
Merging two lists of size $n$ takes $O(n)$ time.

### Pf.
After each comparison, the length of output list increases by 1.
O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms. Also referred to as linearithmic time

Sorting. Mergesort, heapsort, quicksort are sorting algorithms that perform O(n log n) comparisons (on average for quicksort).

Largest empty interval. Given n time-stamps $x_1, ..., x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane \((x_1, y_1), \ldots, (x_n, y_n)\), find the pair that is closest.

\(O(n^2)\) solution. Try all pairs of points.

\[
\begin{align*}
\min &\leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2 \\
\text{for} \ i &= 1 \ \text{to} \ n \ \{ \\
\quad \text{for} \ j &= i+1 \ \text{to} \ n \ \{ \\
\quad \quad d &\leftarrow (x_i - x_j)^2 + (y_i - y_j)^2 \\
\quad \quad \text{if} \ (d < \min) \\
\quad \quad \quad \min &\leftarrow d \\
\quad \} \\
\} \\
\end{align*}
\]

Remark. \(\Omega(n^2)\) seems inevitable, but this is just an illusion. 

---

\[\text{don't need to take square roots}\]
Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```
Independent I (dominating D) set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge (all nodes not in the set have a neighbor in the set)?

$O(n^k)$ solution. Enumerate all subsets of $k$ nodes.

```
foreach subset S of k nodes {
    check whether S is I (D)
    if (S is I (D))
        report S is I (D)
}
```

- Check whether $S$ is an independent set = $O(k^2)$.
- Number of $k$ element subsets = \[ \binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots21} \leq \frac{n^k}{k!} \]
- $O(k^2 n^k / k!) = O(n^k)$.

Poly-time for $k=17$, but not practical.
Exponential Time

Independent/Dominating set: maximum I (minimum D) Given a graph, what is maximum (minimum) size of an independent/dominating set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

\[
S^* \leftarrow \emptyset \\
\text{foreach subset } S \text{ of nodes } \{ \\
\quad \text{check whether } S \text{ is I (D)} \\
\quad \text{if (S is largest I (smallest D) seen so far)} \\
\quad\quad \text{update } S^* \leftarrow S \\
\}\n\]
Five Representative Problems (in Chapter 1)

**Interval Scheduling (IS)**
Find the max. number of non-overlapping activities (greedy solution)

**Weighted Interval Scheduling (WIS)**
Find a set of max. weighted non-overlapping activities (dynamic programming)

**Bipartite Matching (BM)**
Find a matching of maximum size in a given bipartite graph (network flow)

**Independent Set (IS)**
*NP-complete*: IS, WIS and BM as special cases, no polynomial solution

**Competitive Facility Allocation (CFA)**
Two player game on location and profit (based on closeness): harder than NP-complete
*PSPACE-complete*: unlike IS, CFA is hard to verify for a given solution
Player 1 first, followed by Player 2. Can Player 2 guarantee a reward of 20 (or 25)?