Computer Science Approach to problem solving

- If my boss / supervisor / teacher formulates a problem to be solved urgently, can I write a program to efficiently solve this problem??
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.

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

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

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

**Property:** poly-time is invariant over all (non-quantum) computer models.
Average/Worst-Case Analysis

**Worst case running time.** Obtain bound on largest possible running time of algorithm on any input of a given size $N$.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.
- For probabilistic algorithms, we take the worst average running time.

**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 input distributions.
**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.

**Unix grep**

**Primality testing**
Why It Matters

Big-O Complexity

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(2^n)$
- $O(n!)$

Operations vs. Elements graph showing the growth rates of different complexities.
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>

Note: age of Universe $\sim 10^{10}$ years...
Chapter 2
Basics of Algorithm Analysis
2.2 Asymptotic Order of Growth
Asymptotic Order of Growth

Let $f: \mathbb{N} \to \mathbb{R}^+$ be a function, we define

**Upper bounds.**

$$O(f) = \{ g: \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \ [ g(n) \leq c \cdot f(n) ] \}.$$ 

**Lower bounds.**

$$\Omega(f) = \{ g: \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \ [ g(n) \geq c \cdot f(n) ] \}.$$ 

**Tight bounds.**

$$\Theta(f) = O(f) \cap \Omega(f).$$

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

$$T(n) \in O(n^2), O(n^3), \Omega(n^2), \Omega(n), \text{ and } \Theta(n^2).$$

$$T(n) \not\in O(n), \Omega(n^3), \Theta(n), \text{ or } \Theta(n^3).$$
Abuse of notation. \( T(n) = O(f(n)) \).

- Not transitive:
  - \( f(n) = 5n^3; \ g(n) = 3n^2 \)
  - \( f(n) = O(n^3) \) and \( g(n) = O(n^3) \) but \( f(n) \neq g(n) \).
- Better notation: \( T(n) \in O(f(n)) \).
- Acceptable notation: \( T(n) \) is \( O(f(n)) \). (if scared by \( \in \) !)

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

- Statement doesn't "type-check".
- Precisely, \( f(n)=1 \in O(n \log n) \), therefore "at least one comparison".
- Use \( \Omega \) for lower bounds: "at least \( \Omega(n \log n) \) comparisons".
- "requires at least \( cn \log n \) comparisons for \( c>0 \) and all large enough \( n \)."
**Limit theorems.**

Let \( f, g : \mathbb{N} \to \mathbb{R}^+ \) be functions, such that

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^+,
\]

then \( f \in \Theta(g) \), \( g \in \Theta(f) \), \( \Theta(f) = \Theta(g) \)

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0,
\]

then \( f \in O(g) \), \( f \notin \Omega(g) \), \( O(f) \subsetneq O(g) \), \( \Omega(g) \subsetneq \Omega(f) \)
Let \( f, g : \mathbb{N} \rightarrow \mathbb{R}^+ \) be functions

**Transitivity.**

- If \( f \in O(g) \) and \( g \in O(h) \) then \( f \in O(h) \)  
  since \( O(f) \subset O(g) \subset O(h) \).
- If \( f \in \Omega(g) \) and \( g \in \Omega(h) \) then \( f \in \Omega(h) \)  
  since \( \Omega(f) \subset \Omega(g) \subset \Omega(h) \).
- If \( f \in \Theta(g) \) and \( g \in \Theta(h) \) then \( f \in \Theta(h) \)  
  since \( \Theta(f) \subset \Theta(g) \subset \Theta(h) \).

**Additivity.**

- If \( f \in O(h) \) and \( g \in O(h) \) then \( f + g \in O(h) \)  
  since \( f(n) < c_f h(n) \), \( g(n) < c_g h(n) \) \( \Rightarrow f(n) + g(n) < (c_f + c_g) h(n) \).
- If \( f \in \Omega(h) \) and \( g \in \Omega(h) \) then \( f + g \in \Omega(h) \).
- If \( f \in \Theta(h) \) and \( g \in O(h) \) then \( f + g \in \Theta(h) \).

**Consequence:**

- \( f + g \in O(\max\{f,g\}) \)  
  since \( f + g \leq 2\max\{f,g\} \).
- \( f + g \in \Omega(\max\{f,g\}) \)  
  since \( f + g \geq \max\{f,g\} \).
- \( f + g \in \Theta(\max\{f,g\}) \)  
  since \( \max\{f,g\} \leq f + g \leq 2 \max\{f,g\} \).
Consequence:

- \( f + g \in O(\max\{f,g\}) \).
- \( f + g \in \Omega(\max\{f,g\}) \).
- \( f + g \in \Theta(\max\{f,g\}) \).

```plaintext
max \leftarrow a_i
for i = 2 to n {
    if (a_i > max)
        max \leftarrow a_i
}

min \leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2
for i = 1 to n {
    for j = i+1 to n {
        d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min)
            min \leftarrow d
    }
}

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
    }
}
```
Asymptotic Bounds for Some Common Functions

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

**Polynomial time.** Running time \( \in O(n^d) \) for some constant \( d \) independent of the input size \( n \).

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

**Logarithms.** For every \( x > 0 \), \( \log n \in O(n^x) \).

**Exponentials.** For every \( r > 1 \) and every \( d > 0 \), \( n^d \in O(r^n) \).

---

- Every exponential grows faster than every polynomial.
- \( \log \) grows slower 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 minimum of $n$ numbers $a_1, \ldots, a_n$.

```plaintext
min ← a_1
for i = 2 to n {
    if (a_i < min)
        min ← a_i
}
```
**O(n log n) Time**

**O(n log n) time.** Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

**Sorting.** **Mergesort** and **Heapsort** are sorting algorithms that perform O(n log n) comparisons.

**Closest Points on a line.** Given n numbers x_1, ..., x_n, what is the smallest distance x_i-x_j between any two points?

**O(n log n) solution.** Sort the n numbers. Scan the sorted list in order, identifying the minimum gap between two successive points.
O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms

Median Finding. Given n distinct numbers a₁, ..., aₙ, find i such that

$$|\{ j : a_j < a_i \}| = \lfloor n-1 / 2 \rfloor$$ and $$|\{ j : a_j > a_i \}| = \lceil n-1 / 2 \rceil$$.

<table>
<thead>
<tr>
<th></th>
<th>22</th>
<th>31</th>
<th>44</th>
<th>7</th>
<th>12</th>
<th>19</th>
<th>20</th>
<th>35</th>
<th>3</th>
<th>40</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>19</td>
<td>20</td>
<td>22</td>
<td>27</td>
<td>31</td>
<td>35</td>
<td>40</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>11</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

Straight forward approach. Θ(n log n) arithmetic operations (to sort first).

Fundamental question. Can we improve upon this approach?

Remark. This algorithm is Ω(n log n) and it seems inevitable in general, but this is just an illusion: Θ(n) is actually possible and optimal...
Quadratic Time: $O(n^2)$

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.

```
min ← (x_1 - x_2)^2 + (y_1 - y_2)^2
for i = 1 to n {
    for j = i+1 to n {
        d ← (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min)
            min ← d
    }
}
```

Remark. This algorithm is $\Omega(n^2)$ and it seems inevitable in general, but this is just an illusion: $\Theta(n \log n)$ is actually possible and optimal… see chapter 5.
First Repetition. Given N numbers $a_1, \ldots, a_N$, find the smallest $n$ s. t. there exists $1 \leq i < n$ such that $a_i = a_n$.

Most natural solutions will be $\Theta(n^2)$.

An $O(n \log n)$ algorithm (where $n$ is the location of the first repetition) can be obtained from Red-Black Trees or similar to MergeSort.
Quadratic time. Solve $O(n^2)$ independent sub-puzzles each in constant-time.

$n \times n \times n$ Rubik’s cube. Given a scrambled $n \times n \times n$ cube, put it in solved configuration.

Remark. This algorithm is $\Omega(n^2)$ and it seems inevitable in general, but this is just an illusion: $\Theta(n^2/\log n)$ is actually possible and optimal...
Cubic time. Enumerate all triples of elements.

Matrix multiplication. Given two nxn matrices of numbers A, B, what is their matrix product C?

\[
\begin{bmatrix}
  c_{11} & c_{12} & \cdots & c_{1n} \\
  c_{21} & c_{22} & \cdots & c_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{n1} & c_{n2} & \cdots & c_{nn}
\end{bmatrix}
= 
\begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{bmatrix}
\times
\begin{bmatrix}
  b_{11} & b_{12} & \cdots & b_{1n} \\
  b_{21} & b_{22} & \cdots & b_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{n1} & b_{n2} & \cdots & b_{nn}
\end{bmatrix}
\]

\(O(n^3)\) solution. For each entry \(c_{ij}\) compute as below.

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}
\]

Remark. This algorithm is \(\Omega(n^3)\) and it seems inevitable in general, but this is just an illusion: \(O(n^{2.3728639})\) is actually possible…
Polynomial Time: \( O(n^k) \) Time

**Independent set of size \( k \).** Given a graph, are there \( k \) nodes such that no two are joined by an edge? \( k \) is a constant

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

```
foreach subset \( S \) of \( k \) nodes {
    if (\( S \) is an independent set)
        report \( S \)
}
```

- Check whether \( S \) is an independent set = \( O(k^2) \).
- Number of \( k \) element subsets = \( \binom{n}{k} = \frac{n (n-1) (n-2) \cdots (n-k+1)}{k (k-1) (k-2) \cdots (2) (1)} \leq \frac{n^k}{k!} \)
  - \( \)poly-time for \( k=17 \), but not practical
Exponential Time

Independent set. Given a graph, what is the maximum size of an independent set?

\(O(n^2 \cdot 2^n)\) solution. Enumerate all subsets.

```plaintext
S* ← ∅
foreach subset S of nodes {
    if (S is an independent set and |S|>|S*|)
        update S* ← S
}
```
Chapter 2

Basics of Algorithm Analysis