1.2 Five Representative Problems
Interval Scheduling

**Input.** Set of jobs with start times and finish times.

**Goal.** Find maximum cardinality subset of mutually compatible jobs.

Jobs don't overlap
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find maximum weight subset of mutually compatible jobs.
Input. Bipartite graph.
Goal. Find maximum cardinality matching.
Independent Set

Input. Graph.

Goal. Find maximum cardinality independent set. subset of nodes such that no two joined by an edge
Competitive Facility Location

Input. Graph with weight on each each node.

Game. Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

Goal. Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
Five Representative Problems

Variations on a theme: independent set.

Interval scheduling: $n \log n$ greedy algorithm.
Weighted interval scheduling: $n \log n$ dynamic programming algorithm.
Bipartite matching: $n^k$ max-flow based algorithm.
Independent set: NP-complete.
Competitive facility location: PSPACE-complete.
Input. Set of jobs with start times and finish times.

Goal. Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find *maximum weight* subset of mutually compatible jobs.
Bipartite Matching

**Input.** Bipartite graph.

**Goal.** Find *maximum cardinality* matching.
Independent Set

**Input.** Graph.

**Goal.** Find *maximum cardinality* independent set.

A subset of nodes such that no two joined by an edge.
**Competitive Facility Location**

**Input.** Graph with weight on each each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - Francis Sullivan
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 $c > 0$ 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.

$\text{choose } C = 2^d$ for stable matching with $n$ men and $n$ women.
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.
### 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 \( \Theta(n^3) \).
2.4 A Survey of Common Running Times
Linear Time: $O(n)$

Linear time. Running time is at most a constant factor times the size of the input.

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

```plaintext
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```
**Linear Time: \( O(n) \)**

**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 merging two lists](image)

\[
i = 1, \ j = 1
\]

while (both lists are nonempty) {
  \[
  \text{if } (a_i \leq b_j) \text{ append } a_i \text{ to output list and increment } i
  \]
  \[
  \text{else append } b_j \text{ to output list and increment } j
  \]
}

append remainder of nonempty list to output list

**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.

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

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ 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: $O(n^2)$

**Quadratic time.** Enumerate all pairs of elements.

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

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

```plaintext
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.* $\Omega(n^2)$ seems inevitable, but this is just an illusion.
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
    }
}
```
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.

```plaintext
foreach subset S of k nodes {
    check whether S in an independent set
    if (S is an independent set)
        report S is an independent set
}
```

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

poly-time for $k=17$, but not practical
Independent set. Given a graph, what is maximum size of an independent set?

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

```plaintext
S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```
The Plan for the rest of the quarter

See some algorithms. Divide and conquer, greedy, dynamic programming.

NP-completeness. Some problems are so hard, we don’t think they have any polynomial-time algorithm.

Reductions. A lot of these problems (literally thousands), coming from every area of science, engineering, technology, industry, medicine, … are really the same problem in disguise.

More: CSE 421 Algorithms (Winter, Rao)
CSE 431 Complexity (Spring, Lee)
CSE 446 Machine Learning (Winter, Etzioni)