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.

![Diagram showing intervals](image)
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.
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.

simplex method
Unix grep
Chapter 5
Divide and Conquer
Divide-and-Conquer

Divide-and-conquer.
- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.
- Break up problem of size n into two equal parts of size \( \frac{1}{2}n \).
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

Consequence.
- Brute force: \( n^2 \).
- Divide-and-conquer: \( n \log n \).

Divide et impera.
Veni, vidi, vici.
- Julius Caesar
5.1 Mergesort
**Sorting**

*Sorting.* Given $n$ elements, rearrange in ascending order.

**Obvious sorting applications.**
- List files in a directory.
- Organize an MP3 library.
- List names in phone book.
- Display Google PageRank results.

**Easier once sorted.**
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.

**Non-obvious sorting applications.**
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.

...
Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

\[
\begin{array}{llllllllllll}
A & L & G & O & R & I & T & H & M & S \\
A & L & G & O & R & I & T & H & M & S \\
A & G & L & O & R & H & I & M & S & T \\
A & G & H & I & L & M & O & R & S & T
\end{array}
\]

divide\quad O(1)

sort\quad 2T(n/2)

merge\quad O(n)

Jon von Neumann (1945)
Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?
- Linear number of comparisons.
- Use temporary array.

Challenge for the bored. In-place merge. [Kronrud, 1969]

using only a constant amount of extra storage
5.3 Counting Inversions
Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

**Similarity metric:** number of inversions between two rankings.

- My rank: 1, 2, ..., n.
- Your rank: $a_1, a_2, ..., a_n$.
- Songs i and j inverted if $i < j$, but $a_i > a_j$.

### Songs

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>You</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

**Inversions**

3-2, 4-2

**Brute force:** check all $\Theta(n^2)$ pairs i and j.
Applications

Applications.

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.
- **Divide**: separate list into two pieces.

```
1  5  4  8  10  2  6  9  12  11  3  7
```

Divide: \(O(1)\).
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide:** separate list into two pieces.
- **Conquer:** recursively count inversions in each half.

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>11</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
</table>

5 blue-blue inversions

5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

**Divide:** O(1).

**Conquer:** 2T(n / 2)
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.
- **Divide**: separate list into two pieces.
- **Conquer**: recursively count inversions in each half.
- **Combine**: count inversions where \( a_i \) and \( a_j \) are in different halves, and return sum of three quantities.

\[
\begin{array}{cccccccccccc}
1 & 5 & 4 & 8 & 10 & 2 & 6 & 9 & 12 & 11 & 3 & 7 \\
\end{array}
\]

Divide: \( O(1) \).

\[
\begin{array}{cccccccccccc}
1 & 5 & 4 & 8 & 10 & 2 & 6 & 9 & 12 & 11 & 3 & 7 \\
\end{array}
\]

5 blue-blue inversions

8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Conquer: \( 2T(n / 2) \)

Combine: ???

\[
\begin{array}{cccccccccccc}
\end{array}
\]

Total = 5 + 8 + 9 = 22.
Counting Inversions: Combine

Combine: count blue-green inversions
  - Assume each half is sorted.
  - Count inversions where \(a_i\) and \(a_j\) are in different halves.
  - Merge two sorted halves into sorted whole.

\[
\begin{array}{cccccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
\hline
2 & 11 & 16 & 17 & 23 & 25 \\
\end{array}
\]

13 blue-green inversions: \(6 + 3 + 2 + 2 + 0 + 0\)

Count: \(O(n)\)

\[
\begin{array}{cccccccc}
2 & 3 & 7 & 10 & 11 & 14 & 16 & 17 & 18 & 19 & 23 & 25 \\
\end{array}
\]

Merge: \(O(n)\)

\[
T(n) \leq T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lceil n/2 \right\rceil \right) + O(n) \implies T(n) = O(n \log n)
\]
Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.
Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
    if list L has one element
        return 0 and the list L

    Divide the list into two halves A and B
    (r_A, A) ← Sort-and-Count(A)
    (r_B, B) ← Sort-and-Count(B)
    (r, L) ← Merge-and-Count(A, B)

    return r = r_A + r_B + r and the sorted list L
}
```