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 \( \frac{n}{2} \).
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

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

5.1 Mergesort

5.1 Mergesort

Sorting.

Given \( n \) elements, rearrange in ascending order.

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

Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

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.

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

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

Jon von Neumann (1945)

Merging

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

A Useful Recurrence Relation

Def. \( T(n) \) = number of comparisons to mergesort an input of size \( n \).

Mergesort recurrence.

\[
T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}
\]

Solution. \( T(n) = O(n \log_2 n) \).

Assorted proofs. We describe several ways to prove this recurrence.
Initially we assume \( n \) is a power of 2 and replace \( \leq \) with =.

Proof by Recursion Tree

\[
T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{merging} \end{cases}
\]
Proof by Telescoping

Claim. If \( T(n) \) satisfies this recurrence, then \( T(n) = n \log_2 n \).

\[
T(n) = \begin{cases} 
0 & \text{if } n=1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]

\( \text{assuming } n \text{ is a power of 2} \)

Pf. For \( n > 1 \):

\[
\frac{T(n)}{n} = 2\frac{T(n/2)}{n} + 1
\]

\[
= 2\frac{T(n/2)}{n/2} + 1
\]

\[
= 2\frac{T(n/4)}{n/4} + 1 + 1
\]

\[
\vdots
\]

\[
= 2\frac{T(n/2^k)}{n/2^k} + 1 + \cdots + 1
\]

\[
= \log_2 n
\]

Pf. For \( n > 1 \):

\[
\frac{T(n)}{n} = 2\frac{T(n/2)}{n} + 1
\]

\[
= 2\frac{T(n/2)}{n/2} + 1
\]

\[
= 2\frac{T(n/4)}{n/4} + 1 + 1
\]

\[
\vdots
\]

\[
= 2\frac{T(n/2^k)}{n/2^k} + 1 + \cdots + 1
\]

\[
= \log_2 n
\]

Proof by Induction

Claim. If \( T(n) \) satisfies this recurrence, then \( T(n) = n \log_2 n \).

\[
T(n) = \begin{cases} 
0 & \text{if } n=1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]

\( \text{assuming } n \text{ is a power of 2} \)

Pf. (by induction on \( n \))

- Base case: \( n = 1 \).
- Inductive hypothesis: \( T(n) = n \log_2 n \).
- Goal: show that \( T(2n) = 2n \log_2 (2n) \).

\[
T(2n) = 2T(n) + 2n
\]

\[
= 2n \log_2 n + 2n
\]

\[
= 2n(\log_2 (2n) - 1) + 2n
\]

\[
= 2n \log_2 (2n)
\]

Analysis of Mergesort Recurrence

Claim. If \( T(n) \) satisfies the following recurrence, then \( T(n) \leq n \lceil \log n \rceil \).

\[
T(n) = \begin{cases} 
0 & \text{if } n=1 \\
T\left(\lceil n/2 \rceil \right) + T\left(\lfloor n/2 \rfloor \right) + n & \text{otherwise}
\end{cases}
\]

Pf. (by induction on \( n \))

- Base case: \( n = 1 \).
- Define \( n_1 = \lfloor n/2 \rfloor \), \( n_2 = \lceil n/2 \rceil \).
- Induction step: assume true for \( 1, 2, \ldots, n-1 \).

\[
T(n) \leq T(n_1) + T(n_2) + n
\]

\[
\leq n_1 \lceil \log n_1 \rceil + n_2 \lceil \log n_2 \rceil + n
\]

\[
\leq n_1 \lceil \log n_1 \rceil + n_2 \lceil \log n_2 \rceil + n
\]

\[
= n \lceil \log n \rceil + n
\]

\[
\leq n \lceil \log n \rceil - 1 + n
\]

\[
= n \lceil \log n \rceil
\]

5.3 Counting Inversions
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: a₁, a₂, …, aₙ.
- Songs i and j inverted if i < j, but aᵢ > aⱼ.

Brute force: check all Θ(n²) pairs i and j.

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.
- Divide: separate list into two pieces.

Counting Inversions: Divide-and-Conquer

Divide: O(1).

Songs

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>You</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Inversions: 3-2, 4-2

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

**Divide:** \( O(1) \)

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

**Combine:** ???

**Divide:**
- \( 1 \ 5 \ 4 \ 8 \ 10 \ 2 \ 6 \ 9 \ 12 \ 11 \ 3 \ 7 \)
- \( 1 \ 3 \ 4 \ 8 \ 10 \ 2 \ 6 \ 9 \ 12 \ 11 \ 3 \ 7 \)

5 blue-blue inversions
8 green-green inversions
5-4, 5-2, 4-2, 8-2, 10-2
6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

**Conquer:**

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

Total = 5 + 8 + 9 = 22.

**Sort-and-Count:**

- **Pre-condition:** \([\text{Merge-and-Count}] \) A and B are sorted.
- **Post-condition:** \([\text{Sort-and-Count}] \) L is sorted.

```java
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
        \( r_A, A \) \( \bowtie \) Sort-and-Count(A)
        \( r_B, B \) \( \bowtie \) Sort-and-Count(B)
        \( r, A \) \( \bowtie \) Merge-and-Count(A, B)
    return \( r = r_A + r_B + r \) and the sorted list L
}
```

\[ T(n) \leq T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lceil n/2 \right\rceil \right) + O(n) \Rightarrow T(n) = O(n \log n) \]
5.4 Closest Pair of Points

Closest Pair of Points

Closest pair. Given n points in the plane, find a pair with smallest Euclidean distance between them.

Fundamental geometric primitive.
- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.
- Fast closest pair inspired fast algorithms for these problems.

Brute force. Check all pairs of points p and q with Θ(n^2) comparisons.

1-D version. O(n log n) easy if points are on a line.

Assumption. No two points have same x coordinate.

To make presentation cleaner.

Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.

Obstacle. Impossible to ensure n/4 points in each piece.
Closest Pair of Points

Algorithm.
- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{3}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
- **Combine:** find closest pair with one point in each side, assuming that distance $< \delta$.

Return best of 3 solutions.

$\Omega(n^2)$ seems like
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance $< \delta$.
- Observation: only need to consider points within $\delta$ of line $L$.

Sort points in $2\delta$-strip by their $y$ coordinate.

Only check distances of those within 11 positions in sorted list!

Def. Let $s_i$ be the point in the $2\delta$-strip, with the $i$th smallest $y$ coordinate.

Claim. If $|i - j| \geq 12$, then the distance between $s_i$ and $s_j$ is at least $\delta$.

Pf.
- No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$.

Fact. Still true if we replace 12 with 7.
**Closest Pair Algorithm**

```plaintext
Closest-Pair(p_1, ..., p_n) {
    Compute separation line L such that half the points are on one side and half on the other side.
    δ_1 = Closest-Pair(left half)
    δ_2 = Closest-Pair(right half)
    δ = min(δ_1, δ_2)
    Delete all points further than δ from separation line L
    Sort remaining points by y-coordinate.
    Scan points in y-order and compare distance between each point and next 11 neighbors. If any of these distances is less than δ, update δ.
    return δ.
}
```

**Closest Pair of Points: Analysis**

**Running time.**

\[ T(n) \leq 2T(n/2) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n) \]

**Q.** Can we achieve \( O(n \log n) \)?

**A.** Yes. Don’t sort points in strip from scratch each time.
  - Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
  - Sort by merging two pre-sorted lists.

\[ T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]

**5.5 Integer Multiplication**

**Add.** Given two n-digit integers a and b, compute a + b.
  - \( O(n) \) bit operations.

**Multiply.** Given two n-digit integers a and b, compute a \( \times b \).
  - Brute force solution: \( \Theta(n^2) \) bit operations.

\[ \begin{array}{ccccccccc}
    & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
  * & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
  \hline
  & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array} \]
To multiply two n-digit integers:

- Multiply four \(n/2\)-digit integers.
- Add two \(n/2\)-digit integers, and shift to obtain result.

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in \(O(n^{1.585})\) bit operations.
Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute \( C = AB \).

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

Brute force. \( \Theta(n^3) \) arithmetic operations.

Fundamental question. Can we improve upon brute force?

Matrix Multiplication: Warmup

Divide-and-conquer.
- Divide: partition A and B into \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) blocks.
- Conquer: multiply \( 8 \) \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) recursively.
- Combine: add appropriate products using 4 matrix additions.

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix} =
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

\[
C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21})
\]

\[
C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22})
\]

\[
C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21})
\]

\[
C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22})
\]

\[
T(n) = 8T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^3)
\]

Fast Matrix Multiplication

Key idea. multiply 2-by-2 block matrices with only 7 multiplications.

\[
\begin{bmatrix}
P_1 & P_3 - P_6 - P_5 \\
P_4 & P_5
\end{bmatrix} =
\begin{bmatrix}
P_1 & P_3 \\
P_4 & P_5
\end{bmatrix}
\times
\begin{bmatrix}
P_1 & (B_{13}-B_{23}) \\
(B_{23}-B_{13}) & P_1
\end{bmatrix}
\]

\[
P_2 = (A_{11} + A_{12}) \times B_{23}
\]

\[
P_3 = (A_{21} + A_{22}) \times B_{13}
\]

\[
P_5 = (A_{11} + A_{12}) \times B_{13}
\]

\[
P_5 = (A_{11} + A_{12}) \times B_{13}
\]

\[
P_6 = (A_{22} \times B_{23})
\]

\[
P_7 = (A_{12} \times B_{13})
\]

\[
P_7 = (A_{22} \times B_{23})
\]

\[
P_7 = (A_{12} \times B_{13})
\]

Fast matrix multiplication. (Strassen, 1969)
- Divide: partition A and B into \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) blocks.
- Compute: 14 \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) matrices via 10 matrix additions.
- Conquer: multiply 7 \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) matrices recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.
- Assume \( n \) is a power of 2.
- \( T(n) = \# \) arithmetic operations.

\[
T(n) = 7T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})
\]
Fast Matrix Multiplication in Practice

Implementation issues:
- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around \( n = 128 \).

Common misperception: "Strassen is only a theoretical curiosity."
- Advanced Computation Group at Apple Computer reports 8x speedup on G4 Velocity Engine when \( n \approx 2,500 \).
- Range of instances where it’s useful is a subject of controversy.

Remark. Can “Strassenize” \( Ax=b \), determinant, eigenvalues, and other matrix ops.

Fast Matrix Multiplication in Theory

Best known. \( O(n^{2.376}) \) [Coppersmith-Winograd, 1987.]

Conjecture. \( O(n^{2+\varepsilon}) \) for any \( \varepsilon > 0 \).

Caveat. Theoretical improvements to Strassen are progressively less practical.

Q. Multiply two \( 2 \times 2 \) matrices with only 7 scalar multiplications?
A. Yes! [Strassen, 1969] \( \Theta(n^{\log_27}) = O(n^{1.81}) \)

Q. Multiply two \( 2 \times 2 \) matrices with only 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr, 1971] \( \Theta(n^{\log_26}) = O(n^{1.66}) \)

Q. Two \( 3 \times 3 \) matrices with only 21 scalar multiplications?
A. Also impossible. \( \Theta(n^{\log_27}) = O(n^{1.77}) \)

Q. Two \( 70 \times 70 \) matrices with only 143,640 scalar multiplications?
A. Yes! [Pan, 1980] \( \Theta(n^{\log_226}) = O(n^{1.81}) \)

Decimal wars.
- December, 1979: \( O(n^{2.521812}) \).
- January, 1980: \( O(n^{2.521862}) \).