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: $n \log n$. 

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

Applications.
- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.
- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- 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.

Jon von Neumann (1945)

```
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
divide  O(1)
sort    2T(n/2)
merge   O(n)
```
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.**

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

**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

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

\[ T(n) \]

\[ T(n/2) \]

\[ T(n/4) \]

\[ T(n/8) \]

\[ T(n/2^k) \]

\[ T(2) \]

\[ \log_2 n \]

\[ \frac{n}{2} \]

\[ n \log_2 n \]
Proof by Telescoping

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

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

assumes $n$ is a power of 2

Pf. For $n > 1$:

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

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

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

\[
\vdots
\]

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

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

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

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

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

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

assumes \( n \) is a power of 2
Analysis of Mergesort Recurrence

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

$$T(n) \leq \begin{cases} 
0 & \text{if } n = 1 \\
\frac{T(\lceil n/2 \rceil)}{\text{solve left half}} + T(\lfloor n/2 \rfloor) + \frac{n}{\text{merging}} & \text{otherwise}
\end{cases}$$

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, ..., $n-1$.

$$T(n) \leq T(n_1) + T(n_2) + n$$
$$\leq n_1 \lceil \lg n_1 \rceil + n_2 \lfloor \lg n_2 \rfloor + n$$
$$\leq n_1 \lceil \lg n_2 \rceil + n_2 \lfloor \lg n_2 \rfloor + n$$
$$= n \lfloor \lg n_2 \rfloor + n$$
$$\leq n(\lceil \lg n \rceil - 1) + n$$
$$= n \lceil \lg n \rceil$$

Similarly,

$$n_2 = \lfloor n/2 \rfloor$$
$$\leq \left\lfloor 2^{\lceil \lg n \rceil} / 2 \right\rfloor$$
$$= 2^{\lceil \lg n \rceil} / 2$$
$$\Rightarrow \lg n_2 \leq \lceil \lg n \rceil - 1$$
Two Exercises

- Using recursion tree to guess a result, and then, applying induction to prove.

(1) \( T(n) = 3 \ T(\frac{n}{4}) + \Theta(n^2) \)

  Use \( cn^2 \) to replace \( \Theta(n^2) \) for \( c > 0 \) in recursion tree
  Apply \( T(n) \leq dn^2 \) for \( d > 0 \), the guess result, in induction prove
  Determine the constraint associated with \( d \) and \( c \)

(2) \( T(n) = T(n/3) + T(2n/3) + O(n) \)

  Use \( c \) to represent the constant factor in \( O(n) \) in recursion tree
  Apply \( T(n) \leq d \ n \lg n \) for \( d > 0 \), the guess result, in induction prove
  Determine the constraint associated with \( d \) and \( c \)
Master Theorem

The master theorem

Let \( a \geq 1 \) and \( b > 1 \) be constants, let \( f(n) \) be a function, and let \( T(n) \) be defined on the nonnegative integers by the recurrence

\[
T(n) = aT(n/b) + f(n),
\]

where we interpret \( n/b \) to mean either \( \lfloor n/b \rfloor \) or \( \lceil n/b \rceil \). Then \( T(n) \) has the following asymptotic bounds:

1. If \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a} \log n) \).
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0 \), and if \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \).

\[
\begin{align*}
T(n) &= 9T(n/3) + n, \quad T(n) = \Theta(n^2); \\
T(n) &= T(2n/3) + 1, \quad T(n) = \Theta(\log n); \\
T(n) &= 8T(n/2) + \Theta(n^2), \quad T(n) = \Theta(n^3); \\
T(n) &= 3T(n/4) + n \log n, \quad T(n) = \Theta(n \log n) \\
T(n) &= 2T(n/2) + \Theta(n), \quad T(n) = \Theta(n \log n); \\
T(n) &= 7T(n/2) + \Theta(n^2), \quad T(n) = \Theta(n^{\log_7 7}).
\end{align*}
\]
Parallel Merge Sort

Merge sort with parallel recursion: $O(n)$, still slow

Parallel multiway merge sort

Merge sort with parallel merge

Merge sort with two layers
   Bottom layer: slow but efficient
   Top layer: fast but inefficient

Multisequence selection
Extra: Parallel Multiway Merge Sort

Map-Shuffle-Reduce in Hadoop

Partition data and assign to m processors
Each processor sorts data based on n samples

Data access: message passing
Extra: Parallel Merge (Sort)

Merge two sorted subsequences: $O(\log^2 n)$ with $O(n / \log^2 n)$ processors: switch to sequence merge sort with sizes are reduced to $O(\log^2 n)$

Data access: PRAM
Parallel random-access memory
EREW or CRCW

Speedup: seq. time / para. time, Efficiency: # of processors (k)/speed up
Cost: # of processors x parallel time, Cost-optimal: efficiency = 1

Other parallel sorts: bitonic, quick, radix, and sample sort

J. Wu and S. Olariu, On Cost-Optimal Merge of Two Intransitive Sorted Sequence, 2003
Extra: Searching

Systematically search the “space” for a solution.
Key: how to divide (-and-eliminate) the solution space.

1. A person is $L$ distance away from a long wall with no end on both sides. A diamond is placed on the wall which can be identified through touching. Design a searching method with a constant bound in moving distance.
1. A fish needs to be steamed between 5 to 18 minutes. Design a fast-searching method to find the best cooking time. Under- and over-cook can be compared via tasting, but not during cooking. (1 minute is the basic unit of time duration. Quality of fish is a quadratic function.)

check golden-section search
Extra: Fibonacci Sequence and Golden Ratio

\[ F_n = F_{n-1} + F_{n-2}, \ F_0 = 0, \ F_1 = 1: \ 0, 1, 1, 2, 3, 5, 8, 13, 21, \]

\[ (a+b)/b = b/a = 1.618... \]

In music, human body, nature, ...

Eye of god
Extended Fibonacci sequence:
2, 4, 6, 10, 16, 26, ...
4, 8, 12, 20, 32, 52, ...
8, 16, 24, 40, 64, 104, ...

Fibonacci sequence in Last Supper:
1, 2, 3, 5, 8, 13

21 x 21 = 34 x 13 (?)

Mathematics is the language in which God has written the universe - Galileo Galilei
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: $1, 2, \ldots, n$.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs $i$ and $j$ inverted if $i < j$, but $a_i > a_j$.

<table>
<thead>
<tr>
<th>Songs</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>

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

Inversions
3-2, 4-2
More Applications of Rankings

Applications.
- Collaborative filtering (preferences/taste info. from many users)
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.

Voting theory: 3-party voting (Condorcet paradox, Arrow's impossibility theorem on voting)

1: A>B>T Based on 1 and 3: A beats B
2: B>T>A Based on 2 and 3: T beats A
3: T>A>B Based on 1 and 2: B beats T

(A: Anderson, B: Biden, T: Trump)
Counting Inversions: Divide-and-Conquer

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

Divide-and-conquer.

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

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

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>

**Divide**: O(1).

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

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
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.

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

<table>
<thead>
<tr>
<th>1 5 4 8 10 2</th>
<th>6 9 12 11 3 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 blue-blue inversions</td>
<td>8 green-green inversions</td>
</tr>
</tbody>
</table>

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.
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.

Assume each half is sorted.

Count inversions where $a_i$ and $a_j$ are in different halves.

Merge two sorted halves into sorted whole.

to maintain sorted invariant

Count: $O(n)$

Merge: $O(n)$

13 blue-green inversions: $6 + 3 + 2 + 2 + 0 + 0$

$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \Rightarrow T(n) = O(n \log n)$
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
}
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.

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

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

Assumption. No two points have same \( x \) coordinate.

\[ \uparrow \]

to make presentation cleaner
Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.
Closest Pair of Points: First Attempt

**Divide.** Sub-divide region into 4 quadrants.

**Obstacle.** Impossible to ensure \( \frac{n}{4} \) points in each piece.

![Diagram showing a region divided into four quadrants with scattered points.]
Closest Pair of Points

Algorithm.

- **Divide**: draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.

- Bonus point: Given a country map, perform two vertical-oriented cuts of the map into three parts: east, middle, and west, such that $|\text{east}| + |\text{west}| = |\text{middle}|$ ( $|x|$ stands for the population of $x$ ).
**Closest Pair of Points**

**Algorithm.**

- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
Closest Pair of Points

Algorithm.
- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
- **Combine:** find closest pair with one point in each side. \( \Rightarrow \text{seems like } \Theta(n^2) \)
- Return best of 3 solutions.
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < \( \delta \).
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 \( L \).
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.

$\delta = \min(12, 21)$
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!

$\delta = \min(12, 21)$
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. (This is independent of \( \delta \) calculated at each recursive call.)
Closest-Pair Algorithm

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.

\[ \delta_1 = \text{Closest-Pair(left half)} \]
\[ \delta_2 = \text{Closest-Pair(right half)} \]
\[ \delta = \min(\delta_1, \delta_2) \]

Delete all points further than \( \delta \) 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 \( \delta \), update \( \delta \).

return \( \delta \).
}

\( O(n \log n) \)

\( 2T(n / 2) \)

\( O(n) \)

\( O(n \log n) \)

\( O(n) \)
Closest Pair of Points: Analysis

Running time.

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

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.

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

Q. Can we do better?

A. Yes, \( O(n) \) using randomized solution (Chapter 13)
Integer Multiplication

$X$ times $Y$: half-and-half, but still $O(n^2)$

\[ xy = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \]
\[ = x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0. \]

**Complexity:**
\[ T(n) \leq 4T(n/2) + cn \]

Reduce 4 calls to 3:
\[ (x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0 \]

**New complexity:**
\[ T(n) \leq 3T(n/2) + cn \]

Hence,
\[ O(n \log_2 3) = O(n^{1.59}) \]