Algorithm Design
Divide and Conquer

Prof. Dr. Brahim Hnich

May 8, 2013

1

1Slides based on those by Kevin Wayne. Copyright ©2005 Pearson-Addison Wesley. All rights reserved.
Outline

- Merge sort
Outline

- Merge sort
- Counting inversions
Outline

- Merge sort
- Counting inversions
- Closest pair of points
Outline

- Merge sort
- Counting inversions
- Closest pair of points
- Integer multiplication
Outline

- Merge sort
- Counting inversions
- Closest pair of points
- Integer multiplication
- Matrix multiplication
Outline

- Merge sort
- Counting inversions
- Closest pair of points
- Integer multiplication
- Matrix multiplication
- Fast Fourier transform
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 $n/2$.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

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

Prof. Dr. Brahim Hnich
Algorithm Design  Divide and Conquer
Divide and conquer

- Divide-and-conquer.
  - Break up problem into several parts.
Divide and conquer

- Divide-and-conquer.
  - Break up problem into several parts.
  - Solve each part recursively.
Divide and conquer

- Divide-and-conquer.
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
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.
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 $\frac{n}{2}$.
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.
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.
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.
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: $O(n^2)$. 

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: \( O(n^2) \).
  - Divide-and-conquer: \( O(n \log n) \).
Merge sort

- Mergesort

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

- Mergesort
  - Divide array into two halves.
Merge sort

- Mergesort
  - Divide array into two halves.
  - Recursively sort each half.
Merge sort

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

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

```
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.

Figure 2.2 To merge sorted lists $A$ and $B$, we repeatedly extract the smaller item from the front of the two lists and append it to the output.
Merge: demo

smallest

A G L O R

smallest

H I M S T

A
Merge: demo

\[
\begin{align*}
\text{smallest} & \quad \text{smallest} \\
A & \quad G & \quad L & \quad O & \quad R \\
H & \quad I & \quad M & \quad S & \quad T \\
A & \quad G & & & & & & & & & & & & & & \\
\end{align*}
\]
Merge: demo

smallest

A G L O R

smallest

H I M S T

A G H
Merge: demo

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

\[
\begin{array}{cccc}
A & G & H & I \\
\end{array}
\]
Merge: demo

A G L O R

smallest

H I M S T

smallest

A G H I L

Algorithm Design  Divide and Conquer
Merge: demo

Algorithm Design
Divide and Conquer
Merge: demo

Algorithm Design Divide and Conquer
Merge: demo

A G L O R

H I M S T

A G H I L M O R

smallest

smallest
Merge: demo

(first half exhausted) A G L O R H I M S T
(smallest) A G H I L M O R S
Merge: demo

first half exhausted

AGLOR

second half exhausted

HIMST

AGHILMORSST
Definition. \( T(n) \) = number of comparisons to mergesort an input of size \( n \).
A Useful recurrence relation

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

Mergesort recurrence.

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

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

Mergesort recurrence.

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

Solution. \( T(n) = O(n\log_2 n) \).
Definition. \( T(n) = \) number of comparisons to mergesort an input of size \( n \).

Mergesort recurrence.

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

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

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

\[ \text{sorting both halves} \quad \underline{\text{merging}} \]

\[ \log_2 n \]

\[ n \]

\[ 2(n/2) \]

\[ 4(n/4) \]

\[ \cdots \]

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

\[ \cdots \]

\[ n/2 \] (2)

\[ n \log_2 n \]
Proof by telescoping

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

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

- sorting both halves
- merging

\( T(n) \) assumes \( n \) is a power of 2

Pf. For \( n > 1 \):

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

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

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

\[
\vdots
\]

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

\[
= \log_2 n
\]
Proof by induction

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

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

\[\text{assumes } n \text{ is a power of 2}\]

**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 $T(n) \leq n \lceil \lg n \rceil$.

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

\[\uparrow\]

\[\log_2 n\]

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 $1, 2, \ldots, n-1$.

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

\[
n_2 = \left\lfloor \frac{n}{2} \right\rfloor \\
\leq \left\lfloor 2^{\lceil \lg n \rceil} / 2 \right\rfloor \\
= 2^{\lceil \lg n \rceil} / 2 \\
\Rightarrow \ \lg n_2 \leq \lceil \lg n \rceil - 1
\]
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 $a_i > a_j$.

- Brute force: check all $n^2$ pairs $i$ and $j$. 
Counting inversions

- Music site tries to match your song preferences with others.
  - You rank $n$ songs

- Songs $i$ and $j$ are inverted if $i < j$, but $a_i > a_j$.

- Brute force: check all $n^2$ pairs $i$ and $j$.
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.

- Songs \( i \) and \( j \) are inverted if \( i < j \), but \( a_i > a_j \)

- Brute force: check all \( n^2 \) pairs \( i \) and \( j \).
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.
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: $1, 2, \ldots, n$
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: $a_1, a_2, \ldots, a_n$
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 $a_i > a_j$

Prof. Dr. Brahim Hnich
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>

Inversions
3-2, 4-2
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: $a_1, a_2, \ldots, a_n$
  - Songs $i$ and $j$ inverted if $i < j$, but $a_i > a_j$

<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><strong>Me</strong></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td><strong>You</strong></td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

- Brute force: check all $n^2$ pairs $i$ and $j$. 
Figure 5.4 Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.
Applications

- Voting theory
Applications

- Voting theory
- Collaborative filtering
Applications

- Voting theory
- Collaborative filtering
- Measuring the "sortedness" of an array
Applications

- Voting theory
- Collaborative filtering
- Measuring the “sortedness” of an array
- Sensitivity analysis of Google’s ranking function
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
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)
A divide and conquer approach

- Divide: separate list into two pieces
A divide and conquer approach

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half
A divide and conquer approach

- 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
A divide and conquer approach

- **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

<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  
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

**Divide**: $O(1)$.

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

**Combine**: ???

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.
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.
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.
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.

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

Count: $O(n)$

Merge: $O(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \implies T(n) = O(n \log n)$$
Merge-and-Count($A,B$)

Maintain a Current pointer into each list, initialized to point to the front elements
Maintain a variable Count for the number of inversions, initialized to 0

While both lists are nonempty:

Let $a_i$ and $b_j$ be the elements pointed to by the Current pointer
Append the smaller of these two to the output list
If $b_j$ is the smaller element then
  Increment Count by the number of elements remaining in $A$
Endif

Advance the Current pointer in the list from which the smaller element was selected.

EndWhile

Once one list is empty, append the remainder of the other list to the output

Return Count and the merged list
Merge and count: demo

3  7  10  14  18  19
\[ \downarrow \]

2  11  16  17  23  25
\[ \downarrow \]

two sorted halves

auxiliary array

Total:
Merge and count: demo

\[ i = 6 \]

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

Total: 6
Merge and count: demo

i = 6

\[
\begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
\end{array}
\]

\[
\begin{array}{cccccc}
2 & 11 & 16 & 17 & 23 & 25 \\
6 &
\end{array}
\]

auxiliary array

Total: 6
Merge and count: demo

i = 5

\[
\begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
\end{array}
\]

\[
\begin{array}{cccccc}
2 & 11 & 16 & 17 & 23 & 25 \\
6 \\
\end{array}
\]

two sorted halves

\[
\begin{array}{cccccc}
2 & 3 & & & & \\
\end{array}
\]

auxiliary array

Total: 6
Merge and count: demo

\[ i = 5 \]

\[
\begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
\end{array}
\]

\[
\begin{array}{cccccc}
2 & 11 & 16 & 17 & 23 & 25 \\
\end{array}
\]

two sorted halves

\[
\begin{array}{ccc}
2 & 3 & 7 \\
\end{array}
\]

auxiliary array

Total: 6
Merge and count: demo

i = 4

\[
\begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
2 & 11 & 16 & 17 & 23 & 25 \\
\end{array}
\]

two sorted halves

\[
\begin{array}{cccc}
2 & 3 & 7 & 10 \\
\end{array}
\]

auxiliary array

Total: 6
Merge and count: demo

\[
i = 3
\]

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

\[
\begin{align*}
2 & & 3 & & 7 & & 10 & & 11
\end{align*}
\]

Two sorted halves

Auxiliary array

Total: 6 + 3
Merge and count: demo

```
i = 3

3  7  10  14  18  19

2  11  16  17  23  25

6  3

two sorted halves

2  3  7  10  11  14

auxiliary array

Total: 6 + 3
```
Merge and count: demo

i = 2

3 7 10 14 18 19

2 11 16 17 23 25

2 3 7 10 11 14 16

two sorted halves

auxiliary array

Total: 6 + 3 + 2
Merge and count: demo

\[ i = 2 \]

\[ \begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
6 & 3 & 2 & 2 & \end{array} \]

\[ \begin{array}{cccccc}
2 & 11 & 16 & 17 & 23 & 25 \\
\end{array} \]

two sorted halves

\[ \begin{array}{cccccc}
2 & 3 & 7 & 10 & 11 & 14 & 16 & 17 \\
\end{array} \]

auxiliary array

Total: \( 6 + 3 + 2 + 2 \)
Merge and count: demo

\[ i = 2 \]

\[
\begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
\hline
6 & 3 & 2 & 2 & & \text{two sorted halves}
\end{array}
\]

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

Total: \[ 6 + 3 + 2 + 2 \]
Merge and count: demo

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ i = 1 \]

\[ \downarrow \]

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>16</td>
<td>17</td>
<td>23</td>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

two sorted halves

\[
\begin{align*}
2 & \\
3 & \\
7 & \\
10 & \\
11 & \\
14 & \\
16 & \\
17 & \\
18 & \\
19 & 
\end{align*}
\]

auxiliary array

Total: \(6 + 3 + 2 + 2\)
**Merge and count: demo**

- **first half exhausted**
- \( i = 0 \)
- \[
\begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
\end{array}
\]
- \[
\begin{array}{cccccc}
2 & 11 & 16 & 17 & 23 & 25 \\
6 & 3 & 2 & 2 \\
\end{array}
\]
- **two sorted halves**
- **auxiliary array**
- Total: \( 6 + 3 + 2 + 2 \)
Merge and count: demo

\[ i = 0 \]

\[ \begin{array}{cccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
6 & 3 & 2 & 2 & 0 \\
\end{array} \]

\[ \begin{array}{cccccc}
2 & 11 & 16 & 17 & 23 & 25 \\
\end{array} \]

two sorted halves

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

auxiliary array

Total: 6 + 3 + 2 + 2 + 0
Merge and count: demo

```
<table>
<thead>
<tr>
<th>3</th>
<th>7</th>
<th>10</th>
<th>14</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>2</th>
<th>11</th>
<th>16</th>
<th>17</th>
<th>23</th>
<th>25</th>
</tr>
</thead>
</table>

i = 0

two sorted halves

```
<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>7</th>
<th>10</th>
<th>11</th>
<th>14</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>23</th>
<th>25</th>
</tr>
</thead>
</table>
```

auxiliary array

Total: $6 + 3 + 2 + 2 + 0 + 0$
### Merge and count: demo

#### Algorithm Design: Divide and Conquer

- **Initial Array:**
  - $i = 0$
  - $3 \ 7 \ 10 \ 14 \ 18 \ 19$

- **Two Sorted Halves:**
  - $2 \ 11 \ 16 \ 17 \ 23 \ 25$
  - $6 \ 3 \ 2 \ 2 \ 0 \ 0$

- **Auxiliary Array:**
  - $2 \ 3 \ 7 \ 10 \ 11 \ 14 \ 16 \ 17 \ 18 \ 19 \ 23 \ 25$

- **Total**:
  - $6 + 3 + 2 + 2 + 0 + 0 = 13$
Sort-and-Count($L$)

If the list has one element then
   there are no inversions
Else
   Divide the list into two halves:
      $A$ contains the first $[n/2]$ elements
      $B$ contains the remaining $[n/2]$ elements
   ($r_A, A$) = Sort-and-Count($A$)
   ($r_B, B$) = Sort-and-Count($B$)
   ($r, L$) = Merge-and-Count($A, B$)
Endif

Return $r = r_A + r_B + r$, and the sorted list $L$
Closest pair. Given $n$ points in the plane, find a pair with smallest Euclidean distance between them.
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.
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 $O(n^2)$ comparisons.
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 $O(n^2)$ comparisons.

1-D version. $O(n \log n)$ easy if points are on a line.
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 $O(n^2)$ comparisons.

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

Assumption. No two points have same $x$ coordinate.
Closest pair of points: algorithm

**Divide:** draw vertical line $L$ so that roughly $\frac{n}{2}$ points on each side.
Closest pair of points: algorithm

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

**Conquer:** find closest pair in each side recursively.

**Combine:** find closest pair with one point in each side.
Closest pair of points: algorithm

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

**Conquer:** find closest pair in each side recursively.

**Combine:** find closest pair with one point in each side.

**Result:** Return best of 3 solutions.
Closest pair of points: algorithm

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

**Conquer**: find closest pair in each side recursively.

**Combine**: find closest pair with one point in each side.

**Result**: Return best of 3 solutions.
Find closest pair with one point in each side. Let $\delta$ be the min of these distances.

$\delta = \min(12, 21)$
Find closest pair with one point in each side. Let $\delta$ be the min of these distances.

- Observation: only need to consider points within distance $\delta$ of line $L$. 

- $\delta$
Find closest pair with one point in each side. Let $\delta$ be the min of these distances.

- Observation: only need to consider points within distance $\delta$ of line $L$.
- Sort points in $2\delta$-strip by their $y$ coordinate.
Find closest pair with one point in each side. Let \( \delta \) be the min of these distances.

- Observation: only need to consider points within distance \( \delta \) of line \( L \).
- Sort points in \( 2\delta \)-strip by their \( y \) coordinate
- Only check distances of those within 11 positions in sorted list!
Definition. Let $s_i$ be the point in the $2\delta$-strip, with the $i^{th}$ smallest $y$-coordinate.
Definition. Let $s_i$ be the point in the $2\delta$-strip, with the $i^{th}$ smallest $y$-coordinate.
Claim. If $|ij| \geq 12$, then the distance between $s_i$ and $s_j$ is at least $\delta$. 
Claim. If $|ij| \geq 12$, then the distance between $s_i$ and $s_j$ is at least $\delta$.

Proof. (sketch)
Claim. If $|ij| \geq 12$, then the distance between $s_i$ and $s_j$ is at least $\delta$.

Proof. (sketch)

- No two points lie in same $\frac{1}{2}\delta$ by-$\frac{1}{2}\delta$ box.
Closest pair of points: analysis

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

Proof. (sketch)

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

Proof. (sketch)

- 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(p₁, ..., pₙ) {
    \textbf{Compute} separation line L such that half the points are on one side and half on the other side.

    δ₁ = Closest-Pair(left half)
    δ₂ = Closest-Pair(right half)
    δ = \min(δ₁, δ₂)

    \textbf{Delete} all points further than δ from separation line L

    \textbf{Sort} remaining points by y-coordinate.

    \textbf{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.

    \textbf{return} δ.
}
Running time:

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

Can we achieve \( O(n \log n) \)?

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 \textit{merging} two pre-sorted lists.

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

Brute force solution: \( O(n^2) \) bit operations.
Multiplication: attempt 1

To multiply two $n$-digit integers: Divide and conquer

- Multiply four $\frac{n}{2}$-digit integers.
- Add two $\frac{n}{2}$-digit integers, and shift to obtain result.

\[
x = 2^{n/2} \cdot x_1 + x_0 \\
y = 2^{n/2} \cdot y_1 + y_0 \\
xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_0 y_0 + x_1 y_0 + x_0 y_1) + x_0 y_0
\]

\[T(n) = 4T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)\]

\[
\uparrow
\]

assumes $n$ is a power of 2
Karatsuba Multiplication

To multiply two $n$-digit integers: Divide and conquer

- Add two $\frac{n}{2}$-digit integers.
- Multiply three $\frac{n}{2}$-digit integers.
- Add, subtract, and shift $\frac{n}{2}$-digit integers to obtain result.

\[
\begin{align*}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
x y &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
&= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) + x_0 y_0
\end{align*}
\]
Karatsuba Multiplication

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

\[
T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil) + \begin{cases} 
\Theta(n) & \text{add, subtract, shift} \\
\text{recursive calls} & \end{cases}
\]

\[
\log_2 3 
\]

\[
\Rightarrow T(n) = O(n^{ \log_2 3 }) = O(n^{1.585})
\]
Karatsuba: Recursion Tree

\[ T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
3T(n/2) + n & \text{otherwise} 
\end{cases} \]

\[ T(n) = \sum_{k=0}^{\log_3 n-1} n \left(\frac{3}{2}\right)^k = n \frac{\left(\frac{3}{2}\right)^{\log_3 n} - 1}{\frac{3}{2} - 1} = 3n^{\log_3 3} - 2 \]

---

**Diagram:**

- **Base Case:** $T(2)$
- **Recursion:** $T(n/2)$
- **Overall Formula:** $T(n / 2^k)$

**Levels:**

- Level 1: $T(2)$
- Level 2: $T(2/2) = T(1)$
- Level 3: $T(2/4) = T(1/2)$
- Level 4: $T(2/8) = T(1/4)$

**Recurrence Relation:**

- $T(n) = 3T(n/2) + n$ for $n > 2$
Definition. Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$
Matrix multiplication

**Definition.** Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$

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

$$\begin{pmatrix}
  c_{11} & c_{12} & \cdots & c_{1n} \\
  c_{21} & c_{22} & \cdots & c_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{n1} & c_{n2} & \cdots & c_{nn}
\end{pmatrix}
\begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\times
\begin{pmatrix}
  b_{11} & b_{12} & \cdots & b_{1n} \\
  b_{21} & b_{22} & \cdots & b_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{n1} & b_{n2} & \cdots & b_{nn}
\end{pmatrix}$$
Matrix multiplication

**Definition.** Given two $n$-by-$n$ matrices $A$ and $B$, compute

$C = AB$

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

**Brute force.** $O(n^3)$ arithmetic operations.
Matrix multiplication

**Definition.** Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$

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

**Brute force.** $O(n^3)$ arithmetic operations.

**Fundamental question.** Can we improve upon brute force?
Divide and conquer. Given two $n$-by-$n$ matrices $A$ and $B$, compute

$$C = AB$$
Matrix multiplication: Warmup

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

- Divide: partition \( A \) and \( B \) into \( \frac{n}{2} \)-by-\( \frac{n}{2} \) blocks.
Matrix multiplication: Warmup

**Divide and conquer.** Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$

- **Divide:** partition $A$ and $B$ into $\frac{n}{2}$-by-$\frac{n}{2}$ blocks.
- **Conquer:** multiply $8 \frac{n}{2}$-by-$\frac{n}{2}$ recursively.
Matrix multiplication: Warmup

**Divide and conquer.** Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$

- Divide: partition $A$ and $B$ into $\frac{n}{2}$-by-$\frac{n}{2}$ blocks.
- Conquer: multiply $8 \frac{n}{2}$-by-$\frac{n}{2}$ recursively.
- Combine: add appropriate products using 4 matrix additions
**Matrix multiplication: Warmup**

**Divide and conquer.** Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$

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

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

\[
\begin{align*}
C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\
C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\
C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\
C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22})
\end{align*}
\]

\[
T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \implies T(n) = \Theta(n^3)
\]
Key idea. multiply 2-by-2 block matrices with only 7 multiplications
Key idea. multiply 2-by-2 block matrices with only 7 multiplications

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

\[
\begin{align*}
C_{11} &= P_3 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_3 + P_1 - P_3 - P_7
\end{align*}
\]

\[
\begin{align*}
P_1 &= A_{11} \times (B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{12}) \times B_{22} \\
P_3 &= (A_{21} + A_{22}) \times B_{11} \\
P_4 &= A_{22} \times (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12})
\end{align*}
\]
Key idea. Multiply 2-by-2 block matrices with only 7 multiplications.

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

\[
\begin{align*}
C_{11} &= P_3 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{align*}
\]

7 multiplications.
Matrix multiplication: Key idea

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

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

\[
\begin{align*}
C_{11} & = P_3 + P_4 - P_2 + P_6 \\
C_{12} & = P_1 + P_2 \\
C_{21} & = P_3 + P_4 \\
C_{22} & = P_3 + P_1 - P_3 - P_7
\end{align*}
\]

7 multiplications.

18 = 10 + 8 additions (or subtractions).
Fast matrix multiplication. (Strassen, 1969)
Fast matrix multiplication. (Strassen, 1969)

- Divide: partition $A$ and $B$ into $\frac{n}{2}$-by-$\frac{n}{2}$ blocks.
Fast matrix multiplication. (Strassen, 1969)

- Divide: partition $A$ and $B$ into $\frac{n}{2}$-by-$\frac{n}{2}$ blocks.
- Compute: 14 $\frac{n}{2}$-by-$\frac{n}{2}$ matrices via 10 matrix additions.
Fast matrix multiplication. (Strassen, 1969)

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

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

- **Divide:** partition $A$ and $B$ into $\frac{n}{2}$-by-$\frac{n}{2}$ blocks.
- **Compute:** 14 $\frac{n}{2}$-by-$\frac{n}{2}$ matrices via 10 matrix additions.
- **Conquer:** multiply 7 $\frac{n}{2}$-by-$\frac{n}{2}$ recursively.
- **Combine:** 7 products into 4 terms using 8 matrix additions.

\[
T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log_27}) = O(n^{2.81})
\]
Consider the recurrence

\[ T(n) = aT\left(\frac{n}{b}\right) + f(n) \]

Then

- If \( f(n) \ll n^{\lg_b a} \) then \( T(n) = O(n^{\lg_b a}) \)
- If \( f(n) \approx n^{\lg_b a} \) then \( T(n) = O(n^{\lg_b a \lg n}) \)
- If \( f(n) \gg n^{\lg_b a} \) and \( \lim_{n \to \infty} \frac{a f\left(\frac{n}{b}\right)}{f(n)} < 1 \) then \( T(n) = O(f(n)) \)
Problem set

We will solve these in class:

- Solved exercise 1 at page 242 from the course’s textbook
- Solved exercise 2 at page 244 from the course’s textbook
- Exercise 3 at page 246 from the course’s textbook
- Computing $x^n$
- Given $n$ coins where $n - 1$ of them are identical and weigh the same and exactly one coin weighs less. Minimize the number of weighing to find the coin with less weight.
- Assume array $A$ is sorted, does there exist an $i$ such that $A[i] = i$?
- Find the median in a list of numbers
- Maximum contiguous sub-array problem
- Computing Convex Hulls