CSE 431/531: Algorithm Analysis and Design (Spring 2024)
Divide-and-Conquer

Lecturer: Kelin Luo
Department of Computer Science and Engineering
University at Buffalo
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Solving Recurrences
6. Other Classic Algorithms using Divide-and-Conquer
7. Computing $n$-th Fibonacci Number
Greedy Algorithm

- mainly for combinatorial optimization problems
- trivial algorithm runs in exponential time
- greedy algorithm gives an efficient algorithm
- main focus of analysis: correctness of algorithm
Greedy Algorithm
- mainly for combinatorial optimization problems
- trivial algorithm runs in exponential time
- greedy algorithm gives an efficient algorithm
- main focus of analysis: correctness of algorithm

Divide-and-Conquer
- not necessarily for combinatorial optimization problems
- trivial algorithm already runs in polynomial time
- divide-and-conquer gives a more efficient algorithm
- main focus of analysis: running time
Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance
Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

**Running time analysis**

- recursive programs: recurrence
merge-sort\((A, n)\)

1: \textbf{if} \ n = 1 \ \textbf{then}
2: \quad \textbf{return} \ A
3: \textbf{else}
4: \quad B \leftarrow \text{merge-sort}(A[1..\lfloor n/2\rfloor], \lfloor n/2\rfloor)
5: \quad C \leftarrow \text{merge-sort}(A[\lceil n/2 \rceil + 1..n], \lceil n/2 \rceil)
6: \quad \textbf{return} \ \text{merge}(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)
**merge-sort(A, n)**

1: if $n = 1$ then
2: return $A$
3: else
4: $B \leftarrow \text{merge-sort}(A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor)$
5: $C \leftarrow \text{merge-sort}(A[\lfloor n/2 \rfloor + 1..n], \lfloor n/2 \rfloor)$
6: return merge($B, C, \lfloor n/2 \rfloor, \lfloor n/2 \rfloor$)

- Divide: trivial
- Conquer: 4, 5
- Combine: 6
merge-sort()
merge-sort()
merge-sort()
merge-sort()
merge-sort()
merge-sort()
merge-sort()
merge-sort()
Running Time for Merge-Sort

- Each level takes running time $O(n)$
- There are $O(\log n)$ levels
- Running time $= O(n \log n)$
- Better than insertion sort
Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a + b)/2 \rfloor$: $A[a, q]$ and $A[q + 1, b]$; or $A[a, q - 1]$ and $A[q, b]$?
Running Time for Merge-Sort

Implementation

- Divide \( A[a, b] \) by \( q = \lfloor (a + b)/2 \rfloor \): \( A[a, q] \) and \( A[q + 1, b] \); or \( A[a, q - 1] \) and \( A[q, b] \)?

- Speed-up: avoid the constant copying from one layer to another and backward
Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a + b)/2 \rfloor$: $A[a, q]$ and $A[q + 1, b]$; or $A[a, q - 1]$ and $A[q, b]$?

- Speed-up: avoid the constant copying from one layer to another and backward

- Speed-up: stop the dividing process when the sequence sizes fall below constant
Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a + b)/2 \rfloor$: $A[a, q]$ and $A[q + 1, b]$; or $A[a, q - 1]$ and $A[q, b]$?
- Speed-up: avoid the constant copying from one layer to another and backward
- Speed-up: stop the dividing process when the sequence sizes fall below constant
Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a + b)/2 \rfloor$: $A[a, q]$ and $A[q + 1, b]$; or $A[a, q - 1]$ and $A[q, b]$?
- Speed-up: avoid the constant copying from one layer to another and backward
- Speed-up: stop the dividing process when the sequence sizes fall below constant

Stable sorting algorithm

- Stable sorting algorithm has the property that equal items will appear in the final sorted list in the same relative order that they appeared in the initial input.
Running Time for Merge-Sort Using Recurrence

- \( T(n) = \text{running time for sorting } n \text{ numbers}, \text{then} \)

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2
\end{cases}
\]
Running Time for Merge-Sort Using Recurrence

- \( T(n) = \) running time for sorting \( n \) numbers, then

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

- With some tolerance of informality:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
2T(n/2) + O(n) & \text{if } n \geq 2 
\end{cases}
\]
- \( T(n) = \) running time for sorting \( n \) numbers, then

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

- With some tolerance of informality:

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

- Even simpler: \( T(n) = 2T(n/2) + O(n) \). (Implicit assumption: \( T(n) = O(1) \) if \( n \) is at most some constant.)
Running Time for Merge-Sort Using Recurrence

- \( T(n) = \) running time for sorting \( n \) numbers, then

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

- With some tolerance of informality:

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

- Even simpler: \( T(n) = 2T(n/2) + O(n) \). (Implicit assumption: \( T(n) = O(1) \) if \( n \) is at most some constant.)

- Solving this recurrence, we have \( T(n) = O(n \lg n) \) (we shall show how later)
Outline

1 Divide-and-Conquer

2 Counting Inversions

3 Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem

4 Polynomial Multiplication

5 Solving Recurrences

6 Other Classic Algorithms using Divide-and-Conquer

7 Computing $n$-th Fibonacci Number
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$. 

**Counting Inversions**

**Input:** an sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$

**Example:**

4 inversions (for convenience, using numbers, no indices):

$(10, 8)$, $(10, 9)$, $(15, 9)$, $(15, 12)$
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

**Counting Inversions**

**Input:** a sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$
Def. Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

**Input**: a sequence $A$ of $n$ numbers

**Output**: number of inversions in $A$

Example:

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>8</th>
<th>15</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
</table>
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

**Counting Inversions**

**Input:** a sequence $A$ of $n$ numbers  
**Output:** number of inversions in $A$

**Example:**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>15</td>
<td>9</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

**Counting Inversions**

**Input:** a sequence $A$ of $n$ numbers  
**Output:** number of inversions in $A$

**Example:**
```
10  8  15  9  12
  
8   9  10  12  15
```
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

**Counting Inversions**

**Input:** a sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$

**Example:**

4 inversions (for convenience, using numbers, not indices):

- $(10, 8)$
- $(10, 9)$
- $(15, 9)$
- $(15, 12)$
Naive Algorithm for Counting Inversions

\[
\text{count-inversions}(A, n)
\]

1: \( c \leftarrow 0 \)
2: \textbf{for} every \( i \leftarrow 1 \) to \( n - 1 \) \textbf{do}
3: \quad \textbf{for} every \( j \leftarrow i + 1 \) to \( n \) \textbf{do}
4: \quad \quad \textbf{if} \ A[i] > A[j] \textbf{ then} \( c \leftarrow c + 1 \)
5: \quad \textbf{return} \( c \)
Divide-and-Conquer

A: $B$ $C$

- $p = \lfloor n/2 \rfloor$, $B = A[1..p]$, $C = A[p+1..n]$  
- $\#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m$
  \[ m = \left| \{(i, j) : B[i] > C[j]\} \right| \]

Q: How fast can we compute $m$, via trivial algorithm?

A: $O(n^2)$

- Can not improve the $O(n^2)$ time for counting inversions.
Divide-and-Conquer

$p = \lfloor n/2 \rfloor, B = A[1..p], C = A[p + 1..n]$

$\#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m$

$m = \left| \{(i, j) : B[i] > C[j]\} \right|$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: \[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
\]

$C$: \[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]

\[
\text{total} = 0
\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$:  3  8  12  20  32  48  

$C$:  5  7  9  25  29  

$\text{total} = 0$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$$
\begin{array}{c}
B: \\
\begin{array}{ccccccc}
3 & 8 & 12 & 20 & 32 & 48 & \\
\end{array} \\
\end{array}
\quad total = 0

\begin{array}{c}
C: \\
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 & \\
\end{array} \\
\end{array}

+0

3
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$$
\begin{array}{cccccc}
B: & 3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
$$

$$
\begin{array}{cccc}
C: & 5 & 7 & 9 & 25 & 29 \\
\end{array}
$$

$\text{total}= 0$

$+0$

3
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

total = 0

+0

3 5
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$+$ 0

3 5

$\text{total} = 0$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

\[
\begin{align*}
B: & \quad 3 \quad 8 \quad 12 \quad 20 \quad 32 \quad 48 \\
C: & \quad 5 \quad 7 \quad 9 \quad 25 \quad 29 \\
\end{align*}
\]

\[
\begin{align*}
\text{total} = 0
\end{align*}
\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: \[3 \ 8 \ 12 \ 20 \ 32 \ 48\]

$C$: \[5 \ 7 \ 9 \ 25 \ 29\]

$\text{total} = 0$

$+0$

\[3 \ 5 \ 7\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 
\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
\]

$C$: 
\[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]

\[\text{total} = 2\]

$+0$ $+2$

\[
\begin{array}{cccc}
3 & 5 & 7 & 8 \\
\end{array}
\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: \[\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}\]  

$C$: \[\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}\]  

total = 2

\[\begin{array}{cccccc}
3 & 5 & 7 & 8 \\
\end{array}\]  

+0  +2
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: \[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
\]

$C$: \[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]

$\text{total} = 2$

$+0$ \quad $+2$

$\begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 \\
\end{array}$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:  

$B$: \begin{align*} &3 &8 &12 &20 &32 &48 
\end{align*}$

$C$: \begin{align*} &5 &7 &9 &25 &29 
\end{align*}$

\[\text{total} = 2\]

+0 \quad +2

\begin{align*} &3 &5 &7 &8 &9 
\end{align*}
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: \[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48
\end{array}
\]

$C$: \[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29
\end{array}
\]

$\text{total} = 5$

$+$ $0$ $+$ $2$ $+$ $3$

\[
\begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 & 12
\end{array}
\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$\text{total} = 5$

+0  +2  +3

3  5  7  8  9  12
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: \[\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}\] 

$C$: \[\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}\]

\[\begin{array}{cccccc}
+0 & +2 & +3 & +3 \\
\end{array}\]

\[\begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 \\
\end{array}\]

Total = 8
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48  

$C$: 5 7 9 25 29  

$\text{total} = 8$

3 5 7 8 9 12 20
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$3 + 0 + 2 + 3 + 3 = \text{total}= 8$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$\text{total} = 8$

$+0$ $+2$ $+3$ $+3$

3 5 7 8 9 12 20 25
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: 3 8 12 20 32 48 

$C$: 5 7 9 25 29 

$+0$ $+2$ $+3$ $+3$

3 5 7 8 9 12 20 25 29 

total = 8
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48  

$C$: 5 7 9 25 29  

$\text{total} = 8$

+0 +2 +3 +3

3 5 7 8 9 12 20 25 29
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48  

$C$: 5 7 9 25 29  

$\text{total} = 13$

\begin{align*}
+0 & \quad +2 & \quad +3 & \quad +3 & \quad +5 \\
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 & 32
\end{align*}
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: 3 8 12 20 32 48  

$C$: 5 7 9 25 29  

$\text{total} = 13$

+0  +2  +3  +3  +5  

3 5 7 8 9 12 20 25 29 32
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$+$0  $+$2  $+$3  $+$3  $+$5  $+$5

3 5 7 8 9 12 20 25 29 32 48

total = 18
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$\text{total} = 18$
Count Inversions between $B$ and $C$

- Procedure that merges $B$ and $C$ and counts inversions between $B$ and $C$ at the same time

```plaintext
merge-and-count($B, C, n_1, n_2$)
1: count $\leftarrow 0$;
2: $A \leftarrow$ array of size $n_1 + n_2$; $i \leftarrow 1$; $j \leftarrow 1$
3: while $i \leq n_1$ or $j \leq n_2$ do
4:   if $j > n_2$ or ($i \leq n_1$ and $B[i] \leq C[j]$) then
5:     $A[i + j - 1] \leftarrow B[i]$; $i \leftarrow i + 1$
6:     count $\leftarrow$ count + ($j - 1$)
7:   else
8:     $A[i + j - 1] \leftarrow C[j]$; $j \leftarrow j + 1$
9: return $(A, count)$
```
Sort and Count Inversions in $A$

- A procedure that returns the sorted array of $A$ and counts the number of inversions in $A$:

```
sort-and-count($A$, $n$)
1: if $n = 1$ then
2: return ($A$, 0)
3: else
4: ($B$, $m_1$) ← sort-and-count($A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$)
5: ($C$, $m_2$) ← sort-and-count($A[\lceil n/2 \rceil + 1..n], \lceil n/2 \rceil$)
6: ($A$, $m_3$) ← merge-and-count($B$, $C$, $\lceil n/2 \rceil$, $\lceil n/2 \rceil$)
7: return ($A$, $m_1 + m_2 + m_3$)
```
Sort and Count Inversions in $A$

- A procedure that returns the sorted array of $A$ and counts the number of inversions in $A$:

<table>
<thead>
<tr>
<th>sort-and-count($A$, $n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: if $n = 1$ then</td>
</tr>
<tr>
<td>2: return $(A, 0)$</td>
</tr>
<tr>
<td>3: else</td>
</tr>
<tr>
<td>4: $(B, m_1) \leftarrow$</td>
</tr>
<tr>
<td>5: $(C, m_2) \leftarrow$</td>
</tr>
<tr>
<td>6: $(A, m_3) \leftarrow$</td>
</tr>
<tr>
<td>7: return $(A, m_1 + m_2 + m_3)$</td>
</tr>
</tbody>
</table>

- Divide: trivial
- Conquer: 4, 5
- Combine: 6, 7
sort-and-count\((A, n)\)

1: if \(n = 1\) then
2: return \((A, 0)\)
3: else
4: \((B, m_1) \leftarrow \text{sort-and-count}\left(A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor\right)\)
5: \((C, m_2) \leftarrow \text{sort-and-count}\left(A[\lceil n/2 \rceil + 1..n], \lfloor n/2 \rfloor\right)\)
6: \((A, m_3) \leftarrow \text{merge-and-count}(B, C, \lceil n/2 \rceil, \lfloor n/2 \rfloor)\)
7: return \((A, m_1 + m_2 + m_3)\)

- Recurrence for the running time: \(T(n) = 2T(n/2) + O(n)\)
sort-and-count($A, n$)

1: if $n = 1$ then
2: return $(A, 0)$
3: else
4: $(B, m_1) \leftarrow$ sort-and-count($A[1..\lceil n/2 \rceil], \lfloor n/2 \rfloor$)
5: $(C, m_2) \leftarrow$ sort-and-count($A[\lfloor n/2 \rfloor + 1..n], \lfloor n/2 \rfloor$)
6: $(A, m_3) \leftarrow$ merge-and-count($B, C, \lfloor n/2 \rfloor, \lfloor n/2 \rfloor$)
7: return $(A, m_1 + m_2 + m_3)$

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$
- Running time $= O(n \log n)$
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Solving Recurrences
6. Other Classic Algorithms using Divide-and-Conquer
7. Computing $n$-th Fibonacci Number
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Solving Recurrences
6. Other Classic Algorithms using Divide-and-Conquer
7. Computing $n$-th Fibonacci Number
### Quicksort vs Merge-Sort

<table>
<thead>
<tr>
<th>Divide</th>
<th>Merge Sort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial</td>
<td>Recurse</td>
<td>Separate small and big numbers</td>
</tr>
<tr>
<td>Combine</td>
<td>Merge 2 sorted arrays</td>
<td>Recurse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Trivial</td>
</tr>
</tbody>
</table>
Quicksort Example

**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

| 29 | 82 | 75 | 64 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

| 29 | 82 | 75 | 64 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
### Assumption

We can choose median of an array of size $n$ in $O(n)$ time.

<table>
<thead>
<tr>
<th>29</th>
<th>82</th>
<th>75</th>
<th>64</th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>37</th>
<th>17</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
</tbody>
</table>
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

<table>
<thead>
<tr>
<th>29</th>
<th>82</th>
<th>75</th>
<th>64</th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>37</th>
<th>17</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
</tbody>
</table>
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

<table>
<thead>
<tr>
<th>29</th>
<th>82</th>
<th>75</th>
<th>64</th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>37</th>
<th>17</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>17</td>
<td>29</td>
<td>38</td>
<td>45</td>
<td>37</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
</tbody>
</table>
Quicksort

\[ \text{quicksort}(A, n) \]

1. \textbf{if} \( n \leq 1 \) \textbf{then return} \( A \)
2. \( x \leftarrow \) lower median of \( A \)
3. \( A_L \leftarrow \) array of elements in \( A \) that are less than \( x \) \hspace{1cm} \| \hspace{1cm} \text{Divide}
4. \( A_R \leftarrow \) array of elements in \( A \) that are greater than \( x \) \hspace{1cm} \| \hspace{1cm} \text{Divide}
5. \( B_L \leftarrow \) quicksort(\( A_L \), length of \( A_L \)) \hspace{1cm} \| \hspace{1cm} \text{Conquer}
6. \( B_R \leftarrow \) quicksort(\( A_R \), length of \( A_R \)) \hspace{1cm} \| \hspace{1cm} \text{Conquer}
7. \( t \leftarrow \) number of times \( x \) appear in \( A \)
8. \textbf{return} concatenation of \( B_L \), \( t \) copies of \( x \), and \( B_R \)
Quicksort

quicksort\( (A, n) \)

1: \textbf{if} \( n \leq 1 \) \textbf{then return} \( A \)
2: \( x \leftarrow \) lower median of \( A \)
3: \( A_L \leftarrow \) array of elements in \( A \) that are less than \( x \) \hspace{1cm} \| \hspace{1cm} \text{Divide}
4: \( A_R \leftarrow \) array of elements in \( A \) that are greater than \( x \) \hspace{1cm} \| \hspace{1cm} \text{Divide}
5: \( B_L \leftarrow \) quicksort\( (A_L, \text{length of} \ A_L) \) \hspace{1cm} \| \hspace{1cm} \text{Conquer}
6: \( B_R \leftarrow \) quicksort\( (A_R, \text{length of} \ A_R) \) \hspace{1cm} \| \hspace{1cm} \text{Conquer}
7: \( t \leftarrow \) number of times \( x \) appear \( A \)
8: \textbf{return} concatenation of \( B_L \), \( t \) copies of \( x \), and \( B_R \)

- Recurrence \( T(n) \leq 2T(n/2) + O(n) \)
Quicksort

**quicksort**\((A, n)\)

1. **if** \(n \leq 1\) **then return** \(A\)
2. \(x \leftarrow\) lower median of \(A\)
3. \(A_L \leftarrow\) array of elements in \(A\) that are less than \(x\)  
   \hspace{2cm} \text{// Divide}
4. \(A_R \leftarrow\) array of elements in \(A\) that are greater than \(x\)  
   \hspace{2cm} \text{// Divide}
5. \(B_L \leftarrow\) quicksort\((A_L, \text{length of } A_L)\)  
   \hspace{2cm} \text{// Conquer}
6. \(B_R \leftarrow\) quicksort\((A_R, \text{length of } A_R)\)  
   \hspace{2cm} \text{// Conquer}
7. \(t \leftarrow\) number of times \(x\) appear \(A\)
8. **return** concatenation of \(B_L\), \(t\) copies of \(x\), and \(B_R\)

- **Recurrence** \(T(n) \leq 2T(n/2) + O(n)\)
- **Running time** = \(O(n \lg n)\)
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

**Q:** How to remove this assumption?
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

**Q:** How to remove this assumption?

**A:**
1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)
Assumption  We can choose median of an array of size $n$ in $O(n)$ time.

Q: How to remove this assumption?

A:
1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)
2. Choose a pivot randomly and pretend it is the median (it is practical)
quicksort\((A, n)\)

1. \textbf{if} \(n \leq 1\) \textbf{then return} \(A\)
2. \(x \leftarrow \) a random element of \(A\) (\(x\) is called a pivot)
3. \(A_L \leftarrow\) array of elements in \(A\) that are less than \(x\) \hfill Divide
4. \(A_R \leftarrow\) array of elements in \(A\) that are greater than \(x\) \hfill Divide
5. \(B_L \leftarrow\) quicksort\((A_L, \text{length of } A_L)\) \hfill Conquer
6. \(B_R \leftarrow\) quicksort\((A_R, \text{length of } A_R)\) \hfill Conquer
7. \(t \leftarrow\) number of times \(x\) appear \(A\)
8. \textbf{return} concatenation of \(B_L\), \(t\) copies of \(x\), and \(B_R\)
Assumption There is a procedure to produce a random real number in $[0, 1]$.

Q: Can computers really produce random numbers?
**Assumption**  
There is a procedure to produce a random real number in \([0, 1]\).

**Q:** Can computers really produce random numbers?  

**A:** No! The execution of a computer program is deterministic!
Assumption: There is a procedure to produce a random real number in [0, 1].

Q: Can computers really produce random numbers?

A: No! The execution of a computer programs is deterministic!

In practice: use pseudo-random-generator, a deterministic algorithm returning numbers that “look like” random
**Assumption**  There is a procedure to produce a random real number in $[0, 1]$.

**Q:** Can computers really produce random numbers?

**A:** No! The execution of a computer programs is deterministic!

- In practice: use *pseudo-random-generator*, a deterministic algorithm returning numbers that “look like” random
- In theory: assume they can.
Quicksort Using A Random Pivot

\textbf{quicksort}(A, n)

1: if $n \leq 1$ then return $A$

2: $x \leftarrow$ a random element of $A$ ($x$ is called a pivot)

3: $A_L \leftarrow$ array of elements in $A$ that are less than $x$

4: $A_R \leftarrow$ array of elements in $A$ that are greater than $x$

5: $B_L \leftarrow$ quicksort($A_L$, length of $A_L$)

6: $B_R \leftarrow$ quicksort($A_R$, length of $A_R$)

7: $t \leftarrow$ number of times $x$ appear $A$

8: return concatenation of $B_L$, $t$ copies of $x$, and $B_R$

\textbf{Lemma}  The expected running time of the algorithm is $O(n \lg n)$. 
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

| 29 | 82 | 75 | 64 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm**: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm**: an algorithm that only uses “small” extra space.

![Array Partition Example](image)

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
QuickSort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

\[ i \quad j \]

\[
\begin{array}{ccccccccccccccc}
17 & 64 & 75 & 29 & 38 & 45 & 94 & 69 & 25 & 76 & 15 & 92 & 37 & 82 & 85 \\
\end{array}
\]
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

```
17  37  75  29  38  45  94  69  25  76  15  92  64  82  85
```

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.

<table>
<thead>
<tr>
<th>17</th>
<th>37</th>
<th>75</th>
<th>29</th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>64</th>
<th>82</th>
<th>85</th>
</tr>
</thead>
</table>
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm:** an algorithm that only uses “small” extra space.

```
17 37 64 29 38 45 94 69 25 76 15 92 75 82 85
```

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an "In-Place" Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm**: an algorithm that only uses “small” extra space.

\[ \begin{array}{cccccccccccccccc}
17 & 37 & 15 & 29 & 38 & 45 & 64 & 69 & 25 & 76 & 94 & 92 & 75 & 82 & 85 \\
\end{array} \]
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm**: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.

<table>
<thead>
<tr>
<th>17</th>
<th>37</th>
<th>15</th>
<th>29</th>
<th>38</th>
<th>45</th>
<th>25</th>
<th>69</th>
<th>64</th>
<th>76</th>
<th>94</th>
<th>92</th>
<th>75</th>
<th>82</th>
<th>85</th>
</tr>
</thead>
</table>
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.

```
17  37  15  29  38  45  25  64  69  76  94  92  75  82  85
```
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

- To partition the array into two parts, we only need $O(1)$ extra space.