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. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
<table>
<thead>
<tr>
<th>Divide</th>
<th>Merge Sort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conquer</td>
<td>Trivial</td>
<td>Separate small and big numbers</td>
</tr>
<tr>
<td>Combine</td>
<td>Recurse</td>
<td>Recurse</td>
</tr>
<tr>
<td></td>
<td>Merge 2 sorted arrays</td>
<td>Trivial</td>
</tr>
</tbody>
</table>
Quicksort Example

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

\[
\begin{array}{cccccccccccccc}
29 & 82 & 75 & 64 & 38 & 45 & 94 & 69 & 25 & 76 & 15 & 92 & 37 & 17 & 85 \\
\end{array}
\]
Assumption  We can choose median of an array of size $n$ in $O(n)$ time.
Assumption We can choose median of an array of size $n$ in $O(n)$ time.
**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$(A, n)$

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

- 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. **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 \(x\) \hspace{1cm} Divide
4. \(A_R \leftarrow\) array of elements in \(A\) that are greater than \(x\) \hspace{1cm} Divide
5. \(B_L \leftarrow\) quicksort(\(A_L\), length of \(A_L\)) \hspace{1cm} Conquer
6. \(B_R \leftarrow\) quicksort(\(A_R\), length of \(A_R\)) \hspace{1cm} Conquer
7. \(t \leftarrow\) number of times \(x\) appear \(A\)
8. **return** concatenation of \(B_L\), \(t\) copies of \(x\), and \(B_R\)
Randomized Algorithm Model

**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 programs is deterministic!
Randomized Algorithm Model

**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
Randomized Algorithm Model

**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

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 \(x\) \quad \text{\textbackslash \textbackslash \text{Divide}}
4: \(A_R \leftarrow\) array of elements in \(A\) that are greater than \(x\) \quad \text{\textbackslash \textbackslash \text{Divide}}
5: \(B_L \leftarrow\) quicksort\((A_L, \text{length of } A_L)\) \quad \text{\textbackslash \textbackslash \text{Conquer}}
6: \(B_R \leftarrow\) quicksort\((A_R, \text{length of } A_R)\) \quad \text{\textbackslash \textbackslash \text{Conquer}}
7: \(t \leftarrow\) number of times \(x\) appear \(A\)
8: \textbf{return} concatenation of \(B_L, t\) copies of \(x\), and \(B_R\)

Lemma \quad 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.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

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

| 64 | 82 | 75 | 29 | 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.

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    j
64  82  75  29  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.

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

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

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

\[ i \quad j \]

\begin{array}{cccccccccccccccc}
17 & 37 & 15 & 29 & 38 & 45 & 94 & 69 & 25 & 76 & 64 & 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.

```
17  37  15  29  38  45  64  69  25  76  94  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.

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.

```
17 37 15 29 38 45 25 64 69 76 94 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.
partition($A, \ell, r$)

1: $p \leftarrow$ random integer between $\ell$ and $r$, swap $A[p]$ and $A[\ell]$
2: $i \leftarrow \ell, j \leftarrow r$
3: while true do
5: if $i = j$ then break
6: swap $A[i]$ and $A[j]$; $i \leftarrow i + 1$
7: while $i < j$ and $A[i] < A[j]$ do $i \leftarrow i + 1$
8: if $i = j$ then break
9: swap $A[i]$ and $A[j]$; $j \leftarrow j - 1$
10: return $i$
In-Place Implementation of Quick-Sort

quicksort$(A, \ell, r)$

1: if $\ell \geq r$ then return
2: $m \leftarrow$ partition$(A, \ell, r)$
3: quicksort$(A, \ell, m - 1)$
4: quicksort$(A, m + 1, r)$

To sort an array $A$ of size $n$, call quicksort$(A, 1, n)$.

Note: We pass the array $A$ by reference, instead of by copying.
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays
To merge two arrays, we need a third array with size equaling the total size of two arrays.

3 8 12 20 32 48

5 7 9 25 29
To merge two arrays, we need a third array with size equaling the total size of two arrays.
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 & 3
\end{array}
\]
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays

3
8
12
20
32
48

5
7
9
25
29

3
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 \\
\end{array}
\]
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays
To merge two arrays, we need a third array with size equaling the total size of two arrays.

```
3 8 12 20 32 48
```

```
5 7 9 25 29
```

```
3 5 7
```
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

```
3  8  12  20  32  48
5  7  9  25  29
3  5  7
```
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays.
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays.

```
3 8 12 20 32 48
```

```
5 7 9 25 29
```

```
3 5 7 8
```
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

\[ \begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 \\
\end{array} \]
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays.

![Diagram of merging two arrays into a third array](image-url)
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. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number