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

```plaintext
quicksort(A, ℓ, r)
1: if ℓ ≥ r then return
2: m ← partition(A, ℓ, r)
3: quicksort(A, ℓ, 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
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 |
```
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
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 showing merging two arrays](image)
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 5
```
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
```
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

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

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

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 & 7 & 8
\end{array}
\]
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.
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  9  12  20  25  29  32  48
```
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
Q: Can we do better than $O(n \log n)$ for sorting?
Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

- To sort, we are only allowed to compare two elements
- We can not use “internal structures” of the elements
Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$. 
Lemma  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \cdots, N\}$. 

**Lemma**  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob “yes/no” questions about $x$. 
**Lemma**  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \ldots, N\}$.
- You can ask Bob “yes/no” questions about $x$.

**Q:** How many questions do you need to ask Bob in order to know $x$?
Lemma  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob “yes/no” questions about $x$.

**Q:** How many questions do you need to ask Bob in order to know $x$?

**A:** $\lceil \log_2 N \rceil$. 
Lemma: The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \log n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob “yes/no” questions about $x$.

Q: How many questions do you need to ask Bob in order to know $x$?

A: $\lceil \log_2 N \rceil$.

![Binary search tree](attachment:image.png)
Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \ldots, n\}$ in his hand.
- You can ask Bob “yes/no” questions about $\pi$. 
Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob “yes/no” questions about $\pi$.

Q: How many questions do you need to ask in order to get the permutation $\pi$?
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \ldots, n\}$ in his hand.
- You can ask Bob “yes/no” questions about $\pi$.

Q: How many questions do you need to ask in order to get the permutation $\pi$?

A: $\log_2 n! = \Theta(n \log n)$
Comparison-Based Sorting Algorithms

**Q:** Can we do better than $O(n \log n)$ for sorting?

**A:** No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form “does $i$ appear before $j$ in $\pi$?”
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \ldots, n\}$ in his hand.
- You can ask Bob questions of the form “does $i$ appear before $j$ in $\pi$?”

Q: How many questions do you need to ask in order to get the permutation $\pi$?
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form “does $i$ appear before $j$ in $\pi$?”

Q: How many questions do you need to ask in order to get the permutation $\pi$?

A: At least $\log_2 n! = \Theta(n \log n)$
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
Selection Problem

**Input:** a set $A$ of $n$ numbers, and $1 \leq i \leq n$

**Output:** the $i$-th smallest number in $A$
Selection Problem

**Input:** a set $A$ of $n$ numbers, and $1 \leq i \leq n$

**Output:** the $i$-th smallest number in $A$

- Sorting solves the problem in time $O(n \lg n)$. 
Selection Problem

**Input:** a set $A$ of $n$ numbers, and $1 \leq i \leq n$

**Output:** the $i$-th smallest number in $A$

- Sorting solves the problem in time $O(n \lg n)$.
- Our goal: $O(n)$ running time
Recall: Quicksort with Median Finder

quicksort\((A, n)\)

1: \textbf{if} \(n \leq 1\) \textbf{then return} \(A\)
2: \(x \leftarrow\) lower median of \(A\)
3: \(A_L \leftarrow\) elements in \(A\) that are less than \(x\) \hspace{1cm} \text{▷ Divide}
4: \(A_R \leftarrow\) elements in \(A\) that are greater than \(x\) \hspace{1cm} \text{▷ Divide}
5: \(B_L \leftarrow\) quicksort\((A_L, A_L.\text{size})\) \hspace{1cm} \text{▷ Conquer}
6: \(B_R \leftarrow\) quicksort\((A_R, A_R.\text{size})\) \hspace{1cm} \text{▷ Conquer}
7: \(t \leftarrow\) number of times \(x\) appear in \(A\)
8: \textbf{return} the array obtained by concatenating \(B_L\), the array containing \(t\) copies of \(x\), and \(B_R\)
Selection Algorithm with Median Finder

selection(A, n, i)

1: if n = 1 then return A
2: x ← lower median of A
3: AL ← elements in A that are less than x  ▶ Divide
4: AR ← elements in A that are greater than x  ▶ Divide
5: if i ≤ AL.size then
6: return selection(AL, AL.size, i)  ▶ Conquer
7: else if i > n − AR.size then
8: return selection(AR, AR.size, i − (n − AR.size))  ▶ Conquer
9: else
10: return x
Selection Algorithm with Median Finder

\textbf{selection}(A, n, i)

1: \textbf{if} $n = 1$ \textbf{then} \textbf{return} $A$
2: $x \leftarrow$ lower median of $A$
3: $A_L \leftarrow$ elements in $A$ that are less than $x$ ▶ Divide
4: $A_R \leftarrow$ elements in $A$ that are greater than $x$ ▶ Divide
5: \textbf{if} $i \leq A_L$.size \textbf{then}
6: \quad \textbf{return} selection($A_L, A_L$.size, $i$) ▶ Conquer
7: \quad \textbf{else if} $i > n - A_R$.size \textbf{then}
8: \quad \textbf{return} selection($A_R, A_R$.size, $i - (n - A_R$.size)) ▶ Conquer
9: \quad \textbf{else}
10: \quad \textbf{return} $x$

- Recurrence for selection: $T(n) = T(n/2) + O(n)$
Selection Algorithm with Median Finder

selection(A, n, i)

1: if n = 1 then return A
2: x ← lower median of A
3: AL ← elements in A that are less than x ▶ Divide
4: AR ← elements in A that are greater than x ▶ Divide
5: if i ≤ AL.size then
6: return selection(AL, AL.size, i) ▶ Conquer
7: else if i > n − AR.size then
8: return selection(AR, AR.size, i − (n − AR.size)) ▶ Conquer
9: else
10: return x

- Recurrence for selection: $T(n) = T(n/2) + O(n)$
- Solving recurrence: $T(n) = O(n)$
Randomized Selection Algorithm

**selection**\((A, n, i)\)

1. **if** \(n = 1\) **then**
   - return \(A\)
2. \(x \leftarrow\) random element of \(A\) (called pivot)
3. \(A_L \leftarrow\) elements in \(A\) that are less than \(x\)
4. \(A_R \leftarrow\) elements in \(A\) that are greater than \(x\)
5. **if** \(i \leq A_L.\text{size}\) **then**
   - return \(\text{selection}(A_L, A_L.\text{size}, i)\)
6. **else if** \(i > n - A_R.\text{size}\) **then**
   - return \(\text{selection}(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))\)
7. **else**
   - return \(x\)
Randomized Selection Algorithm

(selection(A, n, i))

1: if n = 1 then return A
2: x ← random element of A (called pivot)
3: AL ← elements in A that are less than x
4: AR ← elements in A that are greater than x
5: if i ≤ AL.size then
6: return selection(AL, AL.size, i)
7: else if i > n − AR.size then
8: return selection(AR, AR.size, i − (n − AR.size))
9: else
10: return x

expected running time = O(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
<table>
<thead>
<tr>
<th>Polynomial Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: two polynomials of degree $n - 1$</td>
</tr>
<tr>
<td><strong>Output</strong>: product of two polynomials</td>
</tr>
</tbody>
</table>

Example:

\[(3x^3 + 2x^2 + 5x + 4) \times (2x^3 + 3x^2 + 6x + 5) = 6x^6 + 10x^5 + 15x^4 + 12x^3 + 24x^2 + 25x + 8\]
Polynomial Multiplication

Input: two polynomials of degree $n - 1$

Output: product of two polynomials

Example:

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$
Polynomial Multiplication

**Input:** two polynomials of degree $n - 1$

**Output:** product of two polynomials

**Example:**

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$

$$= 6x^6 - 9x^5 + 18x^4 - 15x^3$$

$$+ 4x^5 - 6x^4 + 12x^3 - 10x^2$$

$$- 10x^4 + 15x^3 - 30x^2 + 25x$$

$$+ 8x^3 - 12x^2 + 24x - 20$$

$$= 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20$$
Polynomial Multiplication

**Input:** two polynomials of degree $n - 1$

**Output:** product of two polynomials

Example:

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$

$$= 6x^6 - 9x^5 + 18x^4 - 15x^3$$
$$+ 4x^5 - 6x^4 + 12x^3 - 10x^2$$
$$- 10x^4 + 15x^3 - 30x^2 + 25x$$
$$+ 8x^3 - 12x^2 + 24x - 20$$

$$= 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20$$

**Input:** $(4, -5, 2, 3), (-5, 6, -3, 2)$

**Output:** $(-20, 49, -52, 20, 2, -5, 6)$
Naïve Algorithm

polynomial-multiplication($A, B, n$)

1: let $C[k] \leftarrow 0$ for every $k = 0, 1, 2, \ldots, 2n - 2$
2: for $i \leftarrow 0$ to $n - 1$ do
3:     for $j \leftarrow 0$ to $n - 1$ do
4:         $C[i + j] \leftarrow C[i + j] + A[i] \times B[j]$
5: return $C$

Running time: $O(n^2)$
Naïve Algorithm

**polynomial-multiplication**($A, B, n$)

1: let $C[k] \leftarrow 0$ for every $k = 0, 1, 2, \cdots, 2n - 2$
2: for $i \leftarrow 0$ to $n - 1$ do
3: \hspace{1em} for $j \leftarrow 0$ to $n - 1$ do
4: \hspace{2em} $C[i + j] \leftarrow C[i + j] + A[i] \times B[j]$
5: return $C$

Running time: $O(n^2)$
Divide-and-Conquer for Polynomial Multiplication

\[ p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4) \]
\[ q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5) \]
Divide-and-Conquer for Polynomial Multiplication

\[ p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4) \]
\[ q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5) \]

- \( p(x) \): degree of \( n - 1 \) (assume \( n \) is even)
- \( p(x) = p_H(x)x^{n/2} + p_L(x), \)
- \( p_H(x), p_L(x) \): polynomials of degree \( n/2 - 1 \).
Divide-and-Conquer for Polynomial Multiplication

\[ p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4) \]
\[ q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5) \]

- \( p(x) \): degree of \( n - 1 \) (assume \( n \) is even)
- \( p(x) = p_H(x)x^{n/2} + p_L(x) \),
- \( p_H(x), p_L(x) \): polynomials of degree \( n/2 - 1 \).

\[ pq = (p_H x^{n/2} + p_L)(q_H x^{n/2} + q_L) \]
Divide-and-Conquer for Polynomial Multiplication

\[ p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4) \]
\[ q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5) \]

- \( p(x) \): degree of \( n - 1 \) (assume \( n \) is even)
- \( p(x) = p_H(x)x^{n/2} + p_L(x) \),
- \( p_H(x), p_L(x) \): polynomials of degree \( n/2 - 1 \).

\[
pq = (p_H x^{n/2} + p_L)(q_H x^{n/2} + q_L) = p_H q_H x^n + (p_H q_L + p_L q_H)x^{n/2} + p_L q_L
\]