CSE 431/531: Analysis of Algorithms

Divide-and-Conquer

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
Greedy algorithm: design efficient algorithms
- Greedy algorithm: design efficient algorithms
- Divide-and-conquer: design more efficient algorithms
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
merge-sort\((A, n)\)

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

- Divide: trivial
- Conquer: 4, 5
- Combine: 6
Running Time for Merge-Sort

- Each level takes running time $O(n)$
- There are $O(\lg n)$ levels
- Running time $= O(n \lg n)$
- Better than insertion sort
$T(n)$ = running time for sorting $n$ numbers, then

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

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

- With some tolerance of informality:

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

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

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

- 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. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Self-Balancing Binary Search Trees
8. 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 $A[i] > A[j]$. 

Example:

10
8
15
9
12

10 8 15 9 12

4 inversions (for convenience, using numbers, not 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:** an 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: an sequence $A$ of $n$ numbers
Output: number of inversions in $A$

Example:

| 10 | 8 | 15 | 9 | 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:** an 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>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>15</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<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:** an sequence $A$ of $n$ numbers

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

Example:

```
10  8  15  9  12
  8  9   10  12  15
```

4 inversions (for convenience, using numbers, not 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:** an sequence $A$ of $n$ numbers

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

Example:

<table>
<thead>
<tr>
<th>10</th>
<th>8</th>
<th>15</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

- 4 inversions (for convenience, using numbers, not indices): (10, 8), (10, 9), (15, 9), (15, 12)
count-inversions$(A, n)$

1. $c \leftarrow 0$
2. for every $i \leftarrow 1$ to $n - 1$
3. for every $j \leftarrow i + 1$ to $n$
4. if $A[i] > A[j]$ then $c \leftarrow c + 1$
5. return $c$
Divide-and-Conquer

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

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

\[ m = \left| \{(i, j) : B[i] > C[j]\} \right| \]

**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

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

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

\[ m = \left| \{(i, j) : B[i] > C[j]\} \right| \]

**Lemma** If both \( B \) and \( C \) are sorted, then we can compute \( m \) in \( O(n) \) time!
Counting Inversions between \( B \) and \( C \)

Count pairs \( i, j \) such that \( B[i] > C[j] \):

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

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

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

$B$: \[\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}\]  \hspace{1cm} \text{total} = 0

$C$: \[\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}\]
Counting Inversions between $B$ and $C$

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

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

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

$\text{total} = 0$

$+0$

$3$
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
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 $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$\text{total} = 0$

3 5

+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

$\text{total} = 0$

$+0$

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}$

$\text{total} = 0$

$B$: \begin{array}{cccccc}
3 & 5 & 7 \\
\end{array}$

$C$: \begin{array}{cccccc}
+0 \\
\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

+0 +2

3 5 7 8

Total = 2
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

$+0$  $+2$

$B$: total = 2

$C$:
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

$\text{total} = 2$

$+0 \quad +2$

3 5 7 8 9
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 = 2

+0 +2

3 5 7 8 9
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 

$total = 5$ 

$B$: 3 5 7 8 9 12 

$C$: +0 +2 +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

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

Total = 5

+0 +2 +3
Counting Inversions between $B$ and $C$

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

$B$: 

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

\[\text{total} = 8\]

$C$: 

\[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]\n
\[+0 \quad +2 \quad +3 \quad +3\]

\[
\begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 \\
\end{array}
\]
Counting Inversions between $B$ and $C$

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

$B$: 

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

$C$: 

\[
\begin{array}{cccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]

\[
\begin{array}{cccccc}
+0 & +2 & +3 & +3 & & \\
3 & 5 & 7 & 8 & 9 & 12 & 20 \\
\end{array}
\]

$\text{total}= 8$
Counting Inversions between $B$ and $C$

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

$B$: 3 8 12 20 32 48  
      3 8 12 20 32 48  
          +0 +2 +3 +3 

$C$: 5 7 9 25 29  
      5 7 9 25 29  

$B$: 3 5 7 8 9 12 20 25  
      3 5 7 8 9 12 20 25  

$C$: 5 7 9 25 29  
      5 7 9 25 29  

$B$: 3 8 12 20 32 48  
      3 8 12 20 32 48  

$C$: 5 7 9 25 29  
      5 7 9 25 29  

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

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

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

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

$\begin{array}{cccccc} & +0 & +2 & +3 & +3 & \\ 3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 \end{array}$

$\text{total} = 8$
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

$+0 +2 +3 +3$

$B$: 3 5 7 8 9 12 20 25 29

$C$: 5 7 9 25 29

$+0 +2 +3 +3$

$\text{total} = 8$
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}

$B$: \begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 \\
\end{array}

\text{total} = 8
Counting Inversions between $B$ and $C$

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

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

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

$\text{total} = 13$
Counting Inversions between $B$ and $C$

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

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

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

\[
\begin{array}{cccccc}
+0 & +2 & +3 & +3 & +5 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 & 32 \\
\end{array}
\]

total = 13
Counting Inversions between $B$ and $C$

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

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

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

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

\[
\begin{array}{cccccccccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 & 32 & 48 \\
\end{array}
\]

$total = 18$
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

$\text{total} = 18$

+0  +2  +3  +3  +5  +5
Count Inversions between $B$ and $C$

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

merge-and-count($B, C, n_1, n_2$)

1. $count \leftarrow 0$
2. $A \leftarrow []; i \leftarrow 1; j \leftarrow 1$
3. while $i \leq n_1$ or $j \leq n_2$
4. 
   - if $j > n_2$ or ($i \leq n_1$ and $B[i] \leq C[j]$) then
   - append $B[i]$ to $A$; $i \leftarrow i + 1$
   - $count \leftarrow count + (j - 1)$
   - else
   - append $C[j]$ to $A$; $j \leftarrow j + 1$
5. 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 $A$:

```plaintext
sort-and-count(A, n)

1. if $n = 1$ then
2. return $(A, 0)$
3. else
4. $(B, m_1) \leftarrow \text{sort-and-count}(A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor)$
5. $(C, m_2) \leftarrow \text{sort-and-count}(A[\lceil n/2 \rceil + 1..n], \lceil n/2 \rceil)$
6. $(A, m_3) \leftarrow \text{merge-and-count}(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
7. return $(A, m_1 + m_2 + m_3)$
```
A procedure that returns the sorted array of $A$ and counts the number of inversions in $A$:

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

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

- **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. $(B, m_1) \leftarrow$ sort-and-count$(A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor)$
5. $(C, m_2) \leftarrow$ sort-and-count$(A[\lceil n/2 \rceil + 1..n], \lceil n/2 \rceil)$
6. $(A, m_3) \leftarrow$ merge-and-count$(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
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. $(B, m_1) \leftarrow \text{sort-and-count}(A[1..\lfloor n/2 \rfloor], \lceil n/2 \rceil)$
5. $(C, m_2) \leftarrow \text{sort-and-count}(A[\lfloor n/2 \rfloor + 1..n], \lceil n/2 \rceil)$
6. $(A, m_3) \leftarrow \text{merge-and-count}(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
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 Other Classic Algorithms using Divide-and-Conquer
6 Solving Recurrences
7 Self-Balancing Binary Search Trees
8 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. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Self-Balancing Binary Search Trees
8. Computing $n$-th Fibonacci Number
<table>
<thead>
<tr>
<th>Divide</th>
<th>Merge Sort</th>
<th>Conquer</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial</td>
<td>Merge 2 sorted arrays</td>
<td>Recurse</td>
<td>Separate small and big numbers</td>
</tr>
<tr>
<td>Conquer</td>
<td></td>
<td></td>
<td>Recurse</td>
</tr>
<tr>
<td>Combine</td>
<td></td>
<td></td>
<td>Trivial</td>
</tr>
</tbody>
</table>
**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.

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>82</td>
<td>75</td>
<td><strong>64</strong></td>
<td>38</td>
<td>45</td>
<td>94</td>
<td>69</td>
<td>25</td>
<td>76</td>
<td>15</td>
<td>92</td>
<td>37</td>
</tr>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td><strong>64</strong></td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
</tr>
</tbody>
</table>
**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.
### Quicksort

**quicksort**(A, n)

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

quicksort\((A, n)\)

1. if \(n \leq 1\) then 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. \(B_L \leftarrow\) quicksort\((A_L, A_L\text{.size})\) \(\|\) Conquer
6. \(B_R \leftarrow\) quicksort\((A_R, A_R\text{.size})\) \(\|\) Conquer
7. \(t \leftarrow\) number of times \(x\) appear \(A\)
8. return the array obtained by concatenating \(B_L\), the array containing \(t\) copies of \(x\), and \(B_R\)

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

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

1. if *n* ≤ 1 then return *A*
2. *x* ← lower median of *A*
3. *A_L* ← elements in *A* that are less than *x* \ Divide
4. *A_R* ← elements in *A* that are greater than *x* \ Divide
5. *B_L* ← quicksort(*A_L, A_L*.size) \ Conquer
6. *B_R* ← quicksort(*A_R, A_R*.size) \ Conquer
7. *t* ← number of times *x* appear in *A*
8. return the array obtained by concatenating *B_L*, the array containing *t* copies of *x*, and *B_R*

- Recurrence: \( T(n) \leq 2T(n/2) + O(n) \)
- Running time = \( O(n \log n) \)
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 Using A Random Pivot

**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. \(A_L \leftarrow\) elements in \(A\) that are less than \(x\)
4. \(A_R \leftarrow\) elements in \(A\) that are greater than \(x\)
5. \(B_L \leftarrow\) quicksort\((A_L, A_L.\text{size})\)
6. \(B_R \leftarrow\) quicksort\((A_R, A_R.\text{size})\)
7. \(t \leftarrow\) number of times \(x\) appear \(A\)
8. return the array obtained by concatenating \(B_L\), the array containing \(t\) 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 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
**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: make the assumption
Quicksort Using A Random Pivot

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. $A_L \leftarrow$ elements in $A$ that are less than $x$ \ Divide
4. $A_R \leftarrow$ elements in $A$ that are greater than $x$ \ Divide
5. $B_L \leftarrow$ quicksort($A_L, A_L$ size) \ Conquer
6. $B_R \leftarrow$ quicksort($A_R, A_R$ size) \ Conquer
7. $t \leftarrow$ number of times $x$ appear $A$
8. return the array obtained by concatenating $B_L$, the array containing $t$ copies of $x$, and $B_R$

When we talk about randomized algorithm in the future, we show that 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.
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.

\[ i \quad j \]

| 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.

\[ i \quad j \]

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

i

j
```
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.

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

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

\[ \begin{array}{cccccccccc}
17 & 37 & 15 & 29 & 38 & 45 & 25 & 69 & 64 & 76 & 94 & 92 & 75 & 82 & 85 \\
\end{array} \]
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.

```
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.

```
i, j
```

```
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.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

partition$(A, \ell, r)$

1. $p \leftarrow$ random integer between $\ell$ and $r$
2. swap $A[p]$ and $A[\ell]$
3. $i \leftarrow \ell$, $j \leftarrow r$
4. while $i < j$ do
5.   while $i < j$ and $A[i] \leq A[j]$ do $j \leftarrow j - 1$
6.   swap $A[i]$ and $A[j]$
7.   while $i < j$ and $A[i] \leq A[j]$ do $i \leftarrow i + 1$
8.   swap $A[i]$ and $A[j]$
9. return $i$
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

quicksort\((A, \ell, r)\)

1. if \(\ell \geq r\) return
2. \(p \leftarrow \text{partition}(A, \ell, r)\)
3. \(q \leftarrow p - 1; \text{while } A[q] = A[p] \text{ and } q \geq \ell \text{ do: } q \leftarrow q - 1\)
4. quicksort\((A, \ell, q)\)
5. \(q \leftarrow p + 1; \text{while } A[q] = A[p] \text{ and } q \leq r \text{ do: } q \leftarrow q + 1\)
6. quicksort\((A, q, r)\)

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.
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.

<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>12</th>
<th>20</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>
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
  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.

```
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

```
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
```
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.

```
3  8  12  20  32  48
5  7  9  25  29
3  5  7  8  9  12  20  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
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. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Self-Balancing Binary Search Trees
8. 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 cannot use “internal structures” of the elements
**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 $x$?

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

- $x = 1$?
- $x \leq 2$?
- $x = 3$?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
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, $x \in \{1, 2, 3, \ldots, N\}$. 

\[
\begin{align*}
&x = 1? \\
&x \leq 2? \\
&x = 3? \\
&1 2 3 4
\end{align*}
\]
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 $x$. 
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, $x \in \{1, 2, 3, \ldots, N\}$.
- You can ask Bob “yes/no” questions about $x$.

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

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

```
x = 1?
  x = 1?
    1
  x = 2?
    2
x = 3?
  x = 3?
    3
  x = 4?
    4
x \leq 2?
```

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 $\{1, 2, 3, \ldots, n\}$ 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 $\pi$?
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: $\log_2 n! = \Theta(n \log n)$
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: 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 $\pi$?

At least $\log_2 n! = \Theta(n \log n)$
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 $\pi$?

A: At least $\log_2 n! = \Theta(n \log n)$
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. Self-Balancing Binary Search Trees
8. 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. if $n \leq 1$ then return $A$
2. $x \leftarrow$ lower median of $A$
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. $B_L \leftarrow$ quicksort($A_L, A_L$.size)
6. $B_R \leftarrow$ quicksort($A_R, A_R$.size)
7. $t \leftarrow$ number of times $x$ appear in $A$
8. return the array obtained by concatenating $B_L$, the array containing $t$ copies of $x$, and $B_R$
Selection Algorithm with Median Finder

\[
\text{selection}(A, n, i)
\]

1. if \( n = 1 \) then return \( A \)
2. \( x \leftarrow \text{lower median of } A \)
3. \( A_L \leftarrow \text{elements in } A \text{ that are less than } x \)  \hspace{1cm} \text{\textbackslash\textbackslash Divide}
4. \( A_R \leftarrow \text{elements in } A \text{ that are greater than } x \)  \hspace{1cm} \text{\textbackslash\textbackslash Divide}
5. if \( i \leq A_L.\text{size} \) then
6. \hspace{1cm} return \( \text{selection}(A_L, A_L.\text{size}, i) \)  \hspace{1cm} \text{\textbackslash\textbackslash Conquer}
7. elseif \( i > n - A_R.\text{size} \) then
8. \hspace{1cm} return \( \text{select}(A_R, A_R.\text{size}, i - (n - A_R.\text{size})) \)  \hspace{1cm} \text{\textbackslash\textbackslash Conquer}
9. else return \( x \)
### Selection Algorithm with Median Finder

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

1. if \(n = 1\) then 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. if \(i \leq A_L\.size\) then
6. return \(\text{selection}(A_L, A_L\.size, i)\) \\ Conquer
7. elseif \(i > n \− A_R\.size\) then
8. return \(\text{select}(A_R, A_R\.size, i \− (n \− A_R\.size))\) \\ Conquer
9. else return \(x\)

- Recurrence for selection: \(T(n) = T(n/2) + O(n)\)
selection \( (A, n, i) \)

1. if \( n = 1 \) then return \( A \)
2. \( x \leftarrow \) lower median of \( A \)
3. \( A_L \leftarrow \) elements in \( A \) that are less than \( x \) \hspace{1cm} \( \backslash \backslash \) Divide
4. \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \) \hspace{1cm} \( \backslash \backslash \) Divide
5. if \( i \leq A_L.\text{size} \) then
6. \hspace{1cm} return selection \( (A_L, A_L.\text{size}, i) \) \hspace{1cm} \( \backslash \backslash \) Conquer
7. elseif \( i > n - A_R.\text{size} \) then
8. \hspace{1cm} return select \( (A_R, A_R.\text{size}, i - (n - A_R.\text{size})) \) \hspace{1cm} \( \backslash \backslash \) Conquer
9. else return \( x \)

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

\texttt{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
6. \hspace{1em} return \( \text{selection}(A_L, A_L.\text{size}, i) \)
7. elseif \( i > n - A_R.\text{size} \) then
8. \hspace{1em} return \( \text{select}(A_R, A_R.\text{size}, i - (n - A_R.\text{size})) \)
9. else return \( x \)
Randomized Selection Algorithm

\[
\text{selection}(A, n, i)
\]

1. if \( n = 1 \) then return \( A \)
2. \( x \leftarrow \text{random element of } A \) (called pivot)
3. \( A_L \leftarrow \text{elements in } A \text{ that are less than } x \)
4. \( A_R \leftarrow \text{elements in } A \text{ that are greater than } x \)
5. if \( i \leq A_L.\text{size} \) then
   6. return \( \text{selection}(A_L, A_L.\text{size}, i) \)
5. elseif \( i > n - A_R.\text{size} \) then
   7. return \( \text{select}(A_R, A_R.\text{size}, i - (n - A_R.\text{size})) \)
9. else 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. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Self-Balancing Binary Search Trees
8. Computing $n$-th Fibonacci Number
Polynomial Multiplication

**Input:** two polynomials of degree \( n - 1 \)

**Output:** product of two polynomials
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 $C[k] = 0$ for every $k = 0, 1, 2, \cdots, 2n - 2$
2. for $i \leftarrow 0$ to $n - 1$
3. \hspace{1em} for $j \leftarrow 0$ to $n - 1$
4. \hspace{2em} $C[i + j] \leftarrow C[i + j] + A[i] \times B[j]$
5. return $C$

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

**polynomial-multiplication**\(^{(A, B, n)}\)

1. let \(C[k] = 0\) for every \(k = 0, 1, 2, \cdots, 2n - 2\)
2. for \(i \leftarrow 0\) to \(n - 1\)
3. \hspace{1em} for \(j \leftarrow 0\) to \(n - 1\)
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

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

- \( p(x) \): 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 \( n/2 - 1 \).

\[
pq = (p_Hx^{n/2} + p_L)(q_Hx^{n/2} + q_L)
\]
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 \( n/2 - 1 \).

\[ pq = (p_Hx^{n/2} + p_L)(q_Hx^{n/2} + q_L) \]
\[ = p_Hq_Hx^n + (p_Hq_L + p_Lq_H)x^{n/2} + p_Lq_L \]
\[ pq = \left( p_H x^{n/2} + p_L \right) \left( q_H x^{n/2} + q_L \right) \]
\[ = p_H q_H x^n + \left( p_H q_L + p_L q_H \right) x^{n/2} + p_L q_L \]
Divide-and-Conquer for Polynomial Multiplication

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

\[ \text{multiply}(p, q) = \text{multiply}(p_H, q_H) \times x^n \]
\[ + (\text{multiply}(p_H, q_L) + \text{multiply}(p_L, q_H)) \times x^{n/2} \]
\[ + \text{multiply}(p_L, q_L) \]
Divide-and-Conquer for Polynomial Multiplication

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

\[ \text{multiply}(p, q) = \text{multiply}(p_H, q_H) \times x^n \]
\[ + (\text{multiply}(p_H, q_L) + \text{multiply}(p_L, q_H)) \times x^{n/2} \]
\[ + \text{multiply}(p_L, q_L) \]

- Recurrence: \( T(n) = 4T(n/2) + O(n) \)
pq = \left( p_H x^{n/2} + p_L \right) \left( q_H x^{n/2} + q_L \right)
= p_H q_H x^n + \left( p_H q_L + p_L q_H \right) x^{n/2} + p_L q_L

\text{multiply}(p, q) = \text{multiply}(p_H, q_H) \times x^n
+ \left( \text{multiply}(p_H, q_L) + \text{multiply}(p_L, q_H) \right) \times x^{n/2}
+ \text{multiply}(p_L, q_L)

\bullet \text{ Recurrence: } T(n) = 4T(n/2) + O(n)
\bullet \quad T(n) = O(n^2)
Reduce Number from 4 to 3
Reduce Number from 4 to 3

\[ pq = (p_H x^{n/2} + p_L)(q_H x^{n/2} + q_L) \]
\[ = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L \]
Reduce Number from 4 to 3

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

\[ p_H q_L + p_L q_H = (p_H + p_L)(q_H + q_L) - p_H q_H - p_L q_L \]
Divide-and-Conquer for Polynomial Multiplication

\[ \begin{align*}
H &= \text{multiply}(p_H, q_H) \\
L &= \text{multiply}(p_L, q_L) \\
(p, q) &= r_H \times x^n + \left(\text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L\right) \times x^{n/2} + r_L
\end{align*} \]

Solving Recurrence:

\[ T(n) = 3T\left(\frac{n}{2}\right) + O(n) \]

\[ T(n) = O(n \log_2 3) = O(n^{1.585}) \]
$r_H = \text{multiply}(p_H, q_H)$

$r_L = \text{multiply}(p_L, q_L)$
Divide-and-Conquer for Polynomial Multiplication

\[ r_H = \text{multiply}(p_H, q_H) \]
\[ r_L = \text{multiply}(p_L, q_L) \]

\[ \text{multiply}(p, q) = r_H \times x^n \]
\[ + (\text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L) \times x^{n/2} \]
\[ + r_L \]
Divide-and-Conquer for Polynomial Multiplication

\[
\begin{align*}
r_H &= \text{multiply}(p_H, q_H) \\
r_L &= \text{multiply}(p_L, q_L) \\
\text{multiply}(p, q) &= r_H \times x^n + (\text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L) \times x^{n/2} + r_L
\end{align*}
\]

- Solving Recurrence: \( T(n) = 3T(n/2) + O(n) \)
Divide-and-Conquer for Polynomial Multiplication

\[ r_H = \text{multiply}(p_H, q_H) \]
\[ r_L = \text{multiply}(p_L, q_L) \]
\[
multiply(p, q) = r_H \times x^n \]
\[ + \left( \text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L \right) \times x^{n/2} + r_L \]

- **Solving Recurrence:** \( T(n) = 3T(n/2) + O(n) \)
- \( T(n) = O(n^{\lg_2 3}) = O(n^{1.585}) \)
Assumption \( n \) is a power of 2. Arrays are 0-indexed.

\[
multiply(A, B, n)
\]

1. if \( n = 1 \) then return \((A[0]B[0])\)
2. \( A_L \leftarrow A[0 .. n/2 - 1], A_H \leftarrow A[n/2 .. n - 1] \)
3. \( B_L \leftarrow B[0 .. n/2 - 1], B_H \leftarrow B[n/2 .. n - 1] \)
4. \( C_L \leftarrow multiply(A_L, B_L, n/2) \)
5. \( C_H \leftarrow multiply(A_H, B_H, n/2) \)
6. \( C_M \leftarrow multiply(A_L + A_H, B_L + B_H, n/2) \)
7. \( C \leftarrow \text{array of } (2n - 1) \text{ 0’s} \)
8. for \( i \leftarrow 0 \) to \( n - 2 \) do
   9. \( C[i] \leftarrow C[i] + C_L[i] \)
   10. \( C[i + n] \leftarrow C[i + n] + C_H[i] \)
11. \( C[i + n/2] \leftarrow C[i + n/2] + C_M[i] - C_L[i] - C_H[i] \)
12. return \( C \)
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. Self-Balancing Binary Search Trees
8. Computing $n$-th Fibonacci Number
• Closest pair
• Convex hull
• Matrix multiplication
• FFT (Fast Fourier Transform): polynomial multiplication in $O(n \lg n)$ time
Closest Pair

**Input:** \( n \) points in plane: \((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\)

**Output:** the pair of points that are closest
Closest Pair

**Input:** \( n \) points in plane: \((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\)

**Output:** the pair of points that are closest
Closest Pair

**Input:** \( n \) points in plane: \((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\)

**Output:** the pair of points that are closest

- Trivial algorithm: \( O(n^2) \) running time
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line.

![Diagram of points divided by a vertical line]
**Divide-and-Conquer Algorithm for Closest Pair**

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
**Divide-and-Conquer Algorithm for Closest Pair**

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half
Divide-and-Conquer Algorithm for Closest Pair

Each box contains at most one pair
For each point, only need to consider $O(1)$ boxes nearby
time for combine = $O(n)$ (many technicalities omitted)

Recurrence:
$$T(n) = 2T(n/2) + O(n)$$

Running time:
$$O(n \lg n)$$
Each box contains at most one pair
Divide-and-Conquer Algorithm for Closest Pair

- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
Divide-and-Conquer Algorithm for Closest Pair

- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine $= O(n)$ (many technicalities omitted)
Divide-and-Conquer Algorithm for Closest Pair

- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- Time for combine $= O(n)$ (many technicalities omitted)
- Recurrence: $T(n) = 2T(n/2) + O(n)$
Each box contains at most one pair
For each point, only need to consider $O(1)$ boxes nearby
time for combine $= O(n)$ (many technicalities omitted)
Recurrence: $T(n) = 2T(n/2) + O(n)$
Running time: $O(n \log n)$
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \lg n)$-Time Algorithm for Convex Hull
Strassen’s Algorithm for Matrix Multiplication

Matrix Multiplication

**Input:** two $n \times n$ matrices $A$ and $B$

**Output:** $C = AB$
**Strassen’s Algorithm for Matrix Multiplication**

**Matrix Multiplication**

- **Input:** two $n \times n$ matrices $A$ and $B$
- **Output:** $C = AB$

**Naive Algorithm: matrix-multiplication($A$, $B$, $n$)**

1. for $i \leftarrow 1$ to $n$
2.     for $j \leftarrow 1$ to $n$
3.         $C[i, j] \leftarrow 0$
4.     for $k \leftarrow 1$ to $n$
5.         $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
6. return $C$

**Running time:** $O(n^3)$
Strassen’s Algorithm for Matrix Multiplication

Matrix Multiplication

**Input:** two \( n \times n \) matrices \( A \) and \( B \)

**Output:** \( C = AB \)

Naive Algorithm: \texttt{matrix-multiplication}(A, B, n)

1. for \( i \leftarrow 1 \) to \( n \)
2.  for \( j \leftarrow 1 \) to \( n \)
3.   \( C[i, j] \leftarrow 0 \)
4.  for \( k \leftarrow 1 \) to \( n \)
5.    \( C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j] \)
6. return \( C \)

- running time = \( O(n^3) \)
Try to Use Divide-and-Conquer

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}_{n/2}
\quad B = \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}_{n/2}
\]

\[
C = \begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

- \text{matrix\_multiplication}(A, B) recursively calls \text{matrix\_multiplication}(A_{11}, B_{11}), \text{matrix\_multiplication}(A_{12}, B_{21}), ...
Try to Use Divide-and-Conquer

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]

\[
C = \begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

- matrix_multiplication\((A, B)\) recursively calls
  - matrix_multiplication\((A_{11}, B_{11})\),
  - matrix_multiplication\((A_{12}, B_{21})\),
  ...

- Recurrence for running time: \(T(n) = 8T(n/2) + O(n^2)\)
- \(T(n) = O(n^3)\)
Strassen’s Algorithm

- $T(n) = 8T(n/2) + O(n^2)$
- Strassen’s Algorithm: improve the number of multiplications from 8 to 7!
- New recurrence: $T(n) = 7T(n/2) + O(n^2)$
Strassen’s Algorithm

- \( T(n) = 8T(n/2) + O(n^2) \)

- Strassen’s Algorithm: improve the number of multiplications from 8 to 7!

- New recurrence: \( T(n) = 7T(n/2) + O(n^2) \)

- Solving Recurrence \( T(n) = O(n^{\log_2 7}) = O(n^{2.808}) \)
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. Self-Balancing Binary Search Trees
8. Computing $n$-th Fibonacci Number
Methods for Solving Recurrences

- The recursion-tree method
- The master theorem
Recursion-Tree Method

- \( T(n) = 2T(n/2) + O(n) \)
$T(n) = 2T(n/2) + O(n)$

Each level takes running time $O(n)$.

There are $O(lg n)$ levels.

Running time = $O(n \cdot lg n)$.
Recursion-Tree Method

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

Each level takes running time \( O(n) \)
Recursion-Tree Method

- \[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) \]

Each level takes running time \( O(n) \)
- There are \( O(\lg n) \) levels
Recursion-Tree Method

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

Each level takes running time \( O(n) \)
- There are \( O(\log n) \) levels
- Running time = \( O(n \log n) \)
Recursion-Tree Method

\[ T(n) = 3T(n/2) + O(n) \]
Recursion-Tree Method

\[ T(n) = 3T(n/2) + O(n) \]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)
Recursion-Tree Method

\[ T(n) = 3T(n/2) + O(n) \]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n)$

- Total running time at level $i$? 
Recursion-Tree Method

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

- Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n \)
Recursion-Tree Method

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

- Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n \)

- Index of last level?
**Recursion-Tree Method**

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

- **Total running time at level \( i \)?** \( \frac{n}{2^i} \times 3^i = \left( \frac{3}{2} \right)^i n \)

- **Index of last level?** \( \lg_2 n \)
Recursion-Tree Method

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

- Total running time at level \(i\)? \(\frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n\)
- Index of last level? \(\lg_2 n\)
- Total running time?
Recursion-Tree Method

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

![](image)

- Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left( \frac{3}{2} \right)^i n \)
- Index of last level? \( \lg_2 n \)
- Total running time?

\[
\sum_{i=0}^{\lg_2 n} \left( \frac{3}{2} \right)^i n = O \left( n \left( \frac{3}{2} \right)^{\lg_2 n} \right) = O(3^{\lg_2 n}) = O(n^{\lg_2 3}).
\]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$
Recursion-Tree Method

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

[Diagram of a recursion tree with nodes labeled with \((n/2)^2\) and \((n/4)^2\) at various levels, illustrating the division and multiplication by 3 in the recurrence relation.]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$

- Total running time at level $i$?
Recursion-Tree Method

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

\[ \begin{array}{c}
\text{n}^2 \\
(n/2)^2 \\
(n/4)^2 \\
(n/8)^2 \\
\vdots \\
(n/2^i)^2 \end{array} \]

- Total running time at level \( i \): \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)
Recursion-Tree Method

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

- Total running time at level \( i \)? \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)
- Index of last level?
Recursion-Tree Method

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

- Total running time at level \( i \)? \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)

- Index of last level? \( \lg_2 n \)
Recursion-Tree Method

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

Diagram:

- Total running time at level \( i \)? \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)
- Index of last level? \( \lg_2 n \)
- Total running time?
- \( T(n) = 3T(n/2) + O(n^2) \)

![Recursion Tree Diagram]

- Total running time at level \( i \)? \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)
- Index of last level? \( \lg_2 n \)
- Total running time?

\[
\sum_{i=0}^{\lg_2 n} \left( \frac{3}{4} \right)^i n^2 = \]
Recursion-Tree Method

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

- Total running time at level \( i \)? \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)
- Index of last level? \( \lg_2 n \)
- Total running time?

\[
\sum_{i=0}^{\lg_2 n} \left( \frac{3}{4} \right)^i n^2 = O(n^2).
\]
**Master Theorem**

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = 2T(n/2) + O(n) )</td>
<td></td>
<td></td>
<td></td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n) )</td>
<td></td>
<td></td>
<td></td>
<td>( O(n^{\log_2 3}) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n^2) )</td>
<td></td>
<td></td>
<td></td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>

**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,
## Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T(n) = 2T(n/2) + O(n))</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>(O(n \lg n))</td>
</tr>
<tr>
<td>(T(n) = 3T(n/2) + O(n))</td>
<td></td>
<td></td>
<td></td>
<td>(O(n^{\lg_2 3}))</td>
</tr>
<tr>
<td>(T(n) = 3T(n/2) + O(n^2))</td>
<td></td>
<td></td>
<td></td>
<td>(O(n^2))</td>
</tr>
</tbody>
</table>

**Theorem**  
\(T(n) = aT(n/b) + O(n^c)\), where \(a \geq 1, b > 1, c \geq 0\) are constants. Then,
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = 2T(n/2) + O(n) )</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>( O(n \lg n) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n) )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>( O(n^{\lg_2 3}) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n^2) )</td>
<td></td>
<td></td>
<td></td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>

**Theorem** \( T(n) = aT(n/b) + O(n^c), \) where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,
### Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

#### Theorem

$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,
### Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem**  

$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
\text{if } c < \lg_b a \\
\text{if } c = \lg_b a \\
\text{if } c > \lg_b a
\end{cases}$$
**Theorem**  

\( T(n) = aT(n/b) + O(n^c), \) where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
?? & \text{if } c < \log_b a \\
?? & \text{if } c = \log_b a \\
?? & \text{if } c > \log_b a 
\end{cases}
\]
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n \log_2^3)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

\[
T(n) = \begin{cases} 
  O(n^{\log_b a}) & \text{if } c < \log_b a \\
  & \text{if } c = \log_b a \\
  & \text{if } c > \log_b a 
\end{cases}
\]
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
?? & \text{if } c = \lg_b a \\
?? & \text{if } c > \lg_b a 
\end{cases}$$
**Master Theorem**

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem**  
$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}$$
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
?? & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]
**Master Theorem**

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem**  
$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}$$
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Which Case?
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2.
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } c < \log_b a \\
O(n^c \log n) & \text{if } c = \log_b a \\
O(n^c) & \text{if } c > \log_b a 
\end{cases}
\]

- **Ex:** \( T(n) = 4T(n/2) + O(n^2) \). **Case 2.** \( T(n) = O(n^2 \log n) \)
Theorem \( T(n) = aT(n/b) + O(n^c), \) where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg b \ a}) & \text{if } c < \lg b \ a \\
O(n^c \lg n) & \text{if } c = \lg b \ a \\
O(n^c) & \text{if } c > \lg b \ a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2). \) Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n). \) Which Case?
Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } c < \log_b a \\
O(n^c \log n) & \text{if } c = \log_b a \\
O(n^c) & \text{if } c > \log_b a 
\end{cases}$$

- Ex: $T(n) = 4T(n/2) + O(n^2)$. Case 2. $T(n) = O(n^2 \log n)$
- Ex: $T(n) = 3T(n/2) + O(n)$. Case 1.
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } c < \log_b a \\
O(n^c \log n) & \text{if } c = \log_b a \\
O(n^c) & \text{if } c > \log_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \log n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\log_2 3}) \)
- Ex: \( T(n) = T(n/2) + O(1) \). Which Case?
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1 \), \( b > 1 \), \( c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a
\end{cases}
\]

- **Ex:** \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- **Ex:** \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- **Ex:** \( T(n) = T(n/2) + O(1) \). Case 2.
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } c < \log_b a \\
O(n^c \log n) & \text{if } c = \log_b a \\
O(n^c) & \text{if } c > \log_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \log n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\log_2 3}) \)
- Ex: \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\log n) \)
Theorem: \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{1\lg_2 3}) \)
- Ex: \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\lg n) \)
- Ex: \( T(n) = 2T(n/2) + O(n^2) \). Which Case?
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- **Ex:** \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- **Ex:** \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- **Ex:** \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\lg n) \)
- **Ex:** \( T(n) = 2T(n/2) + O(n^2) \). Case 3.
Theorem  \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } c < \log_b a \\
O(n^c \log n) & \text{if } c = \log_b a \\
O(n^c) & \text{if } c > \log_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \log n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\log_2 3}) \)
- Ex: \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\log n) \)
- Ex: \( T(n) = 2T(n/2) + O(n^2) \). Case 3. \( T(n) = O(n^2) \)
Proof of Master Theorem Using Recursion Tree

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

1 node

\[ n^c \]

\[ a \text{ nodes} \]

\[ (n/b)^c \]

\[ a^2 \text{ nodes} \]

\[ \left(\frac{n}{b^2}\right)^c \quad \left(\frac{n}{b^2}\right)^c \]

\[ a^3 \text{ nodes} \]

\[ \left(\frac{n}{b^3}\right)^c \quad \left(\frac{n}{b^3}\right)^c \quad \left(\frac{n}{b^3}\right)^c \quad \left(\frac{n}{b^3}\right)^c \quad \left(\frac{n}{b^3}\right)^c \quad \left(\frac{n}{b^3}\right)^c \]

\[ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \]

\[ c < \log_b a: \text{ bottom-level dominates:} \]

\[ \left(\frac{n}{b}\right)^c \]

\[ \frac{\log n}{\log b} \quad \frac{\log n}{\log b} \quad \frac{\log n}{\log b} \quad \frac{\log n}{\log b} \]

\[ \ldots \quad \ldots \quad \ldots \quad \ldots \]

\[ c > \log_b a: \text{ top-level dominates:} \]

\[ O(n^c) \]
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT(n/b) + O(n^c) \]
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT(n/b) + O(n^c) \]

- 1 node
- \( a \) nodes
- \( a^2 \) nodes
- \( a^3 \) nodes

\[ \left( \frac{n}{b^3} \right)^c \] \[ \left( \frac{n}{b^2} \right)^c \] \[ \left( \frac{n}{b} \right)^c \]

\( c < \log_b a \) : bottom-level dominates: \( \left( \frac{a}{b^c} \right)^{\log_b n} n^c = n^{\log_b a} \)
\[ T(n) = aT(n/b) + O(n^c) \]

- **c < \lg_b a**: bottom-level dominates: \( \left( \frac{a}{b^c} \right)^{\lg_b n} n^c = n^{\lg_b a} \)
- **c = \lg_b a**: all levels have same time: \( n^c \log_b n = O(n^c \log n) \)
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT(n/b) + O(n^c) \]

- **c < \lg_b a**: bottom-level dominates: \( (\frac{a}{b^c})^{\lg_b n} n^c = n^{\lg_b a} \)
- **c = \lg_b a**: all levels have same time: \( n^c \lg_b n = O(n^c \lg n) \)
- **c > \lg_b a**: top-level dominates: \( O(n^c) \)
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. Self-Balancing Binary Search Trees
8. Computing $n$-th Fibonacci Number
Elements are organized in a binary-tree structure
Each element (node) is associated with a \textbf{key} value

- if node $u$ is in the left sub-tree of node $v$, then $u.key \leq v.key$
- if node $u$ is the right sub-tree of node $v$, then $u.key \geq v.key$

BST: numbers denote keys

BST: in-order traversal gives a sorted list of keys

BST: numbers denote keys
Binary Search Tree (BST)

- Elements are organized in a binary-tree structure
- Each element (node) is associated with a key value

- If node $u$ is in the left sub-tree of node $v$, then $u.key \leq v.key$
- If node $u$ is the right sub-tree of node $v$, then $u.key \geq v.key$
- In-order traversal of tree gives a sorted list of keys

BST: numbers denote keys

![BST Diagram](attachment:image.png)
Operations on Binary Search Tree $T$

- **insert**: insert an element to $T$
Operations on Binary Search Tree $T$

- **insert**: insert an element to $T$
- **delete**: delete an element from $T$
Operations on Binary Search Tree $T$

- **insert**: insert an element to $T$
- **delete**: delete an element from $T$
- **count-less-than**: return the number of elements in $T$ with key values smaller than a given value
Operations on Binary Search Tree $T$

- **insert**: insert an element to $T$
- **delete**: delete an element from $T$
- **count-less-than**: return the number of elements in $T$ with key values smaller than a given value
- check existence, return element with $i$-th smallest key value,
  ...
Counting Inversions Via Binary Search Tree (BST)

count-inversions\((A, n)\)

1. \( T \leftarrow \) empty BST
2. \( c \leftarrow 0 \)
3. for \( i \leftarrow n \) downto 1
4. \( c \leftarrow c + T.\text{count-less-than}(A[i]) \)
5. \( T.\text{insert}(A[i]) \)
6. return \( c \)

running time = \( n \times (\text{time for } \text{count-less-than} + \text{time for } \text{insert}) \)
count-inversions($A, n$)

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
   4. $c \leftarrow c + T$.count-less-than($A[i]$)
   5. $T$.insert($A[i]$)
4. return $c$

running time $=$

$n \times (\text{time for count + time for insert})$
Counting Inversions Via Binary Search Tree (BST)

count-inversions\((A, n)\)

1. \(T \leftarrow\) empty BST
2. \(c \leftarrow 0\)
3. for \(i \leftarrow n\) downto 1
4. \(c \leftarrow c + T\.\text{count-less-than}(A[i])\)
5. \(T\.\text{insert}(A[i])\)
6. return \(c\)

running time = 
\(n \times (\text{time for count} + \text{time for insert})\)
Counting Inversions Via Binary Search Tree (BST)

**count-inversions** ($A, n$)

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. $c \leftarrow c + T$.count-less-than($A[i]$)
5. $T$.insert($A[i]$)
6. return $c$

Running time =

$n \times (\text{time for count} + \text{time for insert})$

tree elements

```
15  3  16  12  32  7
```

count-less-than(7) = 0
count-inversions\((A, n)\)

1. \(T \leftarrow \) empty BST
2. \(c \leftarrow 0\)
3. for \(i \leftarrow n\) downto 1
4. \(c \leftarrow c + T.\text{count-less-than}(A[i])\)
5. \(T.\text{insert}(A[i])\)
6. return \(c\)

running time =

\(n \times (\text{time for count} + \text{time for insert})\)
Counting Inversions Via Binary Search Tree (BST)

**count-inversions**(A, n)

1. \( T \leftarrow \text{empty BST} \)
2. \( c \leftarrow 0 \)
3. for \( i \leftarrow n \) downto 1
4. \[ c \leftarrow c + T.\text{count-less-than}(A[i]) \]
5. \( T.\text{insert}(A[i]) \)
6. return \( c \)

running time = 
\( n \times (\text{time for count} + \text{time for insert}) \)
Counting Inversions Via Binary Search Tree (BST)

**Algorithm:**

Given an array $A$ of length $n$

1. Initialize empty BST $T$
2. Initialize $c = 0$
3. For $i$ from $n$ down to $1$
   - $c = c + T$.count-less-than($A[i]$)
5. **Return** $c$

**Running Time:**

$n \times (\text{time for count} + \text{time for insert})$

**Example:**

```
<table>
<thead>
<tr>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
```

- count-less-than(7) = 0
- count-less-than(32) = 1
- Insert(7)
- Insert(32)
count-inversions($A, n$)

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. $c \leftarrow c + T.\text{count-less-than}(A[i])$
5. $T.\text{insert}(A[i])$
6. return $c$

running time =

$n \times (\text{time for count} + \text{time for insert})$
Counting Inversions Via Binary Search Tree (BST)

count-inversions(A, n)

1. \( T \leftarrow \text{empty BST} \)
2. \( c \leftarrow 0 \)
3. for \( i \leftarrow n \) downto 1
4. \( c \leftarrow c + T.\text{count-less-than}(A[i]) \)
5. \( T.\text{insert}(A[i]) \)
6. return \( c \)

running time =
\( n \times (\text{time for count} + \text{time for insert}) \)
Counting Inversions Via Binary Search Tree (BST)

**count-inversions**($A$, $n$)

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. \hspace{1em} $c \leftarrow c + T$.count-less-than($A[i]$)
5. \hspace{1em} $T$.insert($A[i]$)
6. return $c$

running time = $n \times$ (time for count + time for insert)

tree elements

15 3 16 12 32 7

$T$.insert(7)
$T$.insert(32)
$T$.insert(12)

count-less-than(7) = 0
count-less-than(32) = 1
count-less-than(12) = 1
count-less-than(16) = 2
count-inversions($A, n$)

1. $T \leftarrow \text{empty BST}$
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto $1$
4. \hspace{1em} $c \leftarrow c + T.\text{count-less-than}(A[i])$
5. \hspace{1em} $T.\text{insert}(A[i])$
6. return $c$

running time =

$n \times (\text{time for count} + \text{time for insert})$
Counting Inversions Via Binary Search Tree (BST)

count-inversions\((A, n)\)

1. \(T \leftarrow \) empty BST
2. \(c \leftarrow 0\)
3. for \(i \leftarrow n\) downto 1
4. \(c \leftarrow c + T.\text{count-less-than}(A[i])\)
5. \(T.\text{insert}(A[i])\)
6. return \(c\)

running time = \(n \times (\text{time for count} + \text{time for insert})\)

tree elements

\[
\begin{array}{c}
15 & 3 & 16 & 12 & 32 & 7 \\
\end{array}
\]

count-less-than(7) = 0
insert(7)
count-less-than(32) = 1
insert(32)
count-less-than(12) = 1
insert(12)
count-less-than(16) = 2
insert(16)
count-less-than(3) = 0
count-inversions($A, n$)

1. $T \leftarrow \text{empty BST}$
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. \hspace{1em} $c \leftarrow c + T.\text{count-less-than}(A[i])$
5. \hspace{1em} $T.\text{insert}(A[i])$
6. return $c$

running time =

$n \times (\text{time for count} + \text{time for insert})$
count-inversions(A, n)

1. \( T \leftarrow \) empty BST
2. \( c \leftarrow 0 \)
3. for \( i \leftarrow n \) downto 1
4. \( c \leftarrow c + T.\text{count-less-than}(A[i]) \)
5. \( T.\text{insert}(A[i]) \)
6. return \( c \)

running time = 
\[ n \times (\text{time for count} + \text{time for insert}) \]
**Counting Inversions Via Binary Search Tree (BST)**

**count-invensions(A, n)**

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. \[ c \leftarrow c + T.\text{count-less-than}(A[i]) \]
5. $T.\text{insert}(A[i])$
6. return $c$

running time = 
$n \times (\text{time for count} + \text{time for insert})$

count-less-than(7) = 0
insert(7)
count-less-than(32) = 1
insert(32)
count-less-than(12) = 1
insert(12)
count-less-than(16) = 2
insert(16)
count-less-than(3) = 0
insert(3)
count-less-than(15) = 3
insert(15)
Counting Inversions Via Binary Search Tree (BST)

**count-inversions**$(A, n)$

1. $T \leftarrow$ empty BST
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. \hspace{0.5cm} $c \leftarrow c + T$.count-less-than$(A[i])$
5. \hspace{0.5cm} $T$.insert$(A[i])$
6. return $c$

running time =
$n \times ($time for count + time for insert$)$

count-less-than$(7) = 0$
count-less-than$(32) = 1$
count-less-than$(12) = 1$
count-less-than$(16) = 2$
count-less-than$(3) = 0$
count-less-than$(15) = 3$
c = 0 + 1 + 1 + 2 + 0 + 3 = 7$
Binary Search Tree: Insertion

BST: numbers denote keys
Binary Search Tree: Insertion
Binary Search Tree: Insertion

Diagram of a binary search tree with nodes 1, 3, 5, 6, 7, 8, 10, 13, 14.
Binary Search Tree: Insertion
Binary Search Tree: Insertion

Diagram of a binary search tree with numbers 1, 3, 6, 4, 7, 10, 14, 8, and 5 indicated. The tree structure shows the insertion of a new node at position 5.
Binary Search Tree: Insertion

```
8
|--- 3
|   |--- 1
|   |   |--- 4
|   |--- 6
|   |   |--- 7
|   |--- 10
|       |--- 14
|   |   |--- 13
|   |--- 5
```
recursive-insert($v, key$)

1. if $v = \text{nil}$ then
2. $u \leftarrow \text{new node with } u.left = u.right = \text{nil}$
3. $u.key \leftarrow key$
4. return $u$
5. if $key < v.key$ then
6. $v.left \leftarrow \text{recursive-insert($v.left, key$)}$
7. else
8. $v.right \leftarrow \text{recursive-insert($v.right, key$)}$
9. return $v$

insert($key$)

1. $root \leftarrow \text{recursive-insert($root, key$)}$
Binary Search Tree: Deletion

no right child
Binary Search Tree: Deletion

no right child
Binary Search Tree: Deletion
Binary Search Tree: Deletion
Binary Search Tree: Deletion
Binary Search Tree: Deletion
Binary Search Tree: Deletion
Binary Search Tree: Deletion

Diagram of a binary search tree with nodes labeled 1 to 20. Node 7 is highlighted, indicating a deletion operation.
recursive-delete($v$)

1. if $v.right = nil$ then return ($v.left, v$)
2. ($v.right, del) \leftarrow \text{recursive-delete}(v.right)$
3. return ($v, del$)

- recursive-delete($v$) deletes the element in the sub-tree rooted at $v$ with the largest key value
recursive-delete($v$)

1. if $v.right = \text{nil}$ then return ($v.left, v$)
2. ($v.right, del) \leftarrow \text{recursive-delete}(v.right)$
3. return ($v, del$)

- $\text{recursive-delete}(v)$ deletes the element in the sub-tree rooted at $v$ with the largest key value
- returns: the new root and the deleted node
recursive-delete($v$)

1. if $v.right = nil$ then return $(v.left, v)$
2. $(v.right, del) \leftarrow$ recursive-delete($v.right$)
3. return $(v, del)$

- recursive-delete($v$) deletes the element in the sub-tree rooted at $v$ with the largest key value
- returns: the new root and the deleted node

delete($v$)  \hspace{1cm} \text{\textbackslash \textbackslash returns the new root after deletion}

1. if $v.left = nil$ then return $v.right$
2. $(r, del) \leftarrow$ recursive-delete($v.left$)
3. $r.key \leftarrow del.key$
4. return $r$
recursive-delete($v$)

1. if $v.right = \text{nil}$ then return $(v.left, v)$
2. $(v.right, del) \leftarrow \text{recursive-delete}(v.right)$
3. return $(v, del)$

delete($v$) \quad \text{returns the new root after deletion}

1. if $v.left = \text{nil}$ then return $v.right$
2. $(r, del) \leftarrow \text{recursive-delete}(v.left)$
3. $r.key \leftarrow del.key$
4. return $r$
**recursive-delete**(v)

1. if \( v.right = \text{nil} \) then return \((v.left, v)\)
2. \((v.right, \text{del}) \leftarrow \text{recursive-delete}(v.right)\)
3. return \((v, \text{del})\)

**delete**(v) \hspace{1cm} \(\text{\\ returns the new root after deletion}\)

1. if \( v.left = \text{nil} \) then return \(v.right\)
2. \((r, \text{del}) \leftarrow \text{recursive-delete}(v.left)\)
3. \(r.key \leftarrow \text{del.key}\)
4. return \(r\)

- to remove left-child of \( v \): call \( v.left \leftarrow \text{delete}(v.left)\)
- to remove right-child of \( v \): call \( v.right \leftarrow \text{delete}(v.right)\)
- to remove root: call \( root \leftarrow \text{delete}(root)\)
Binary Search Tree: count-less-than

Need to maintain a "size" property for each node:

v.size = number of nodes in the tree rooted at v
Need to maintain a “size” property for each node
• Need to maintain a “size” property for each node
• \( v.size = \) number of nodes in the tree rooted at \( v \)
Binary Search Tree: count-less-than

- Need to maintain a “size” property for each node
- $v.size =$ number of nodes in the tree rooted at $v$
- Need to maintain a “size” property for each node
- \( v.size \) = number of nodes in the tree rooted at \( v \)

\[
\begin{align*}
\# (\text{elements} < 10) &= 8 \\
3 &\quad 11 \\
1 &\quad 6 &\quad 4 &\quad 7 &\quad 14 &\quad 13 &\quad 10 \\
1 &\quad 1 &\quad 1 &\quad 2 \\
4 &\quad 1 &\quad 1 &\quad 1 &\quad 1 &\quad 1
\end{align*}
\]
Need to maintain a “size” property for each node

\[ v.size = \text{number of nodes in the tree rooted at } v \]

\[ \text{# (elements < 10) = } \]
- Need to maintain a “size” property for each node
- $v.size =$ number of nodes in the tree rooted at $v$

# (elements < 10) = (5+1)
Binary Search Tree: count-less-than

- Need to maintain a “size” property for each node
- $v.size =$ number of nodes in the tree rooted at $v$

\[
\begin{align*}
\# \text{(elements < 10)} &= (5+1) \\
\end{align*}
\]
Need to maintain a “size” property for each node

\[ v.size = \text{number of nodes in the tree rooted at } v \]

\# \text{(elements < 10)} = (5+1)
Need to maintain a “size” property for each node

$v.size = \text{number of nodes in the tree rooted at } v$

$\# (\text{elements } < 10) = (5+1) + 1$
Need to maintain a “size” property for each node

$$v.size = \text{number of nodes in the tree rooted at } v$$

$$(\text{elements } < 10) = (5+1) + 1 = 7$$
Trick: “nil” is a node with size 0.

```
recursive-count(v, value)

1. if v = nil then return 0
2. if value ≤ v.key
3. return recursive-count(v.left, key)
4. else
5. return v.left.size + 1 + recursive-count(v.right, key)
```

```
count-less-than(value)

1. return recursive-count(root, value)
```
Each operation takes time $O(h)$.

$h = \text{height of tree}$

$n = \text{number of nodes in tree}$
Running Time for Each Operation

- Each operation takes time $O(h)$.
- $h =$ height of tree
- $n =$ number of nodes in tree

Q: What is the height of the tree in the best scenario?
Each operation takes time $O(h)$.

$h = \text{height of tree}$

$n = \text{number of nodes in tree}$

**Q:** What is the height of the tree in the best scenario?

**A:** $O(\lg n)$
Running Time for Each Operation

- Each operation takes time $O(h)$.
- $h =$ height of tree
- $n =$ number of nodes in tree

**Q:** What is the height of the tree in the **best** scenario?

**A:** $O(\lg n)$

**Q:** What is the height of the tree in the **worst** scenario?
Each operation takes time $O(h)$.

- $h =$ height of tree
- $n =$ number of nodes in tree

**Q:** What is the height of the tree in the **best** scenario?

**A:** $O(\lg n)$

**Q:** What is the height of the tree in the **worst** scenario?

**A:** $O(n)$
Def. A self-balancing BST is a BST that automatically keeps its height small.
Def. A **self-balancing** BST is a BST that automatically keeps its height small

- AVL tree
- red-black tree
- Splay tree
- Treap
- ...
Def. A self-balancing BST is a BST that automatically keeps its height small

- AVL tree
- red-black tree
- Splay tree
- Treap
- ...
An AVL Tree Is Balanced

**Balanced**: for every node $v$ in the tree, the heights of the left and right sub-trees of $v$ differ by at most 1.
An AVL Tree Is Balanced

Balanced: for every node \( v \) in the tree, the heights of the left and right sub-trees of \( v \) differ by at most 1.
An AVL Tree Is Balanced

Balanced: for every node $v$ in the tree, the heights of the left and right sub-trees of $v$ differ by at most 1.

![AVL Tree Diagram]

not balanced
An AVL Tree Is Balanced

**Balanced**: for every node \( v \) in the tree, the heights of the left and right sub-trees of \( v \) differ by at most 1.

![Balanced AVL Trees Diagram](image)
An AVL Tree Is Balanced

Balanced: for every node $v$ in the tree, the heights of the left and right sub-trees of $v$ differ by at most 1.

Lemma Property guarantees height $= O(\log n)$.

- $f(h)$: minimum size of a balanced tree of height $h$
An AVL Tree Is Balanced

Balanced: for every node \( v \) in the tree, the heights of the left and right sub-trees of \( v \) differ by at most 1.

Lemma  Property guarantees height = \( O(\log n) \).

- \( f(h) \): minimum size of a balanced tree of height \( h \)

- \( f(0) = 0, f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 7 \cdots \)
- $f(h)$: minimum size of a balanced tree of height $h$

\[
f(0) = 0 \]
\[
f(1) = 1 \]
\[
f(h) = f(h - 1) + f(h - 2) + 1 \quad h \geq 2
\]
- $f(h)$: minimum size of a balanced tree of height $h$

\[
f(0) = 0 \quad f(1) = 1 \quad f(h) = f(h - 1) + f(h - 2) + 1 \quad h \geq 2
\]

- $f(h) = 2^{\Theta(h)}$ (i.e., $\lg f(h) = \Theta(h)$)
Depth of AVL tree

- $f(h)$: minimum size of a balanced tree of height $h$
- $f(h) = 2^{\Theta(h)}$
Depth of AVL tree

- \( f(h) \): minimum size of a balanced tree of height \( h \)
- \( f(h) = 2^{\Theta(h)} \)
- If a AVL tree has size \( n \) and height \( h \), then
  \[
n \geq f(h) = 2^{\Theta(h)}
  \]
Depth of AVL tree

- \( f(h) \): minimum size of a balanced tree of height \( h \)
- \( f(h) = 2^{\Theta(h)} \)
- If a AVL tree has size \( n \) and height \( h \), then
  \[
  n \geq f(h) = 2^{\Theta(h)}
  \]
- Thus, \( h \leq \Theta(\log n) \)
An AVL Tree Is Balanced

Balanced: for every node \( v \) in the tree, the heights of the left and right sub-trees of \( v \) differ by at most 1.

How can we maintain the balanced property?
An AVL Tree Is Balanced

Balanced: for every node $v$ in the tree, the heights of the left and right sub-trees of $v$ differ by at most 1.

How can we maintain the balanced property?
Maintain Balance Property After Insertion

A: the deepest node such that the balance property is not satisfied after insertion.

Wlog, we inserted an element to the left-sub-tree of A:

B: the root of left-sub-tree of A.

Case 1: we inserted an element to the left-sub-tree of B.
$A$: the deepest node such that the balance property is not satisfied after insertion
A: the deepest node such that the balance property is not satisfied after insertion

Wlog, we inserted an element to the left-sub-tree of A
Maintain Balance Property After Insertion

- **A**: the deepest node such that the balance property is not satisfied after insertion
- **Wlog**, we inserted an element to the left-sub-tree of **A**
- **B**: the root of left-sub-tree of **A**
Maintain Balance Property After Insertion

- $A$: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of $A$
- $B$: the root of left-sub-tree of $A$
- case 1: we inserted an element to the left-sub-tree of $B$
A: the deepest node such that the balance property is not satisfied after insertion

Wlog, we inserted an element to the left-sub-tree of $A$

$B$: the root of left-sub-tree of $A$

case 1: we inserted an element to the left-sub-tree of $B$
A: the deepest node such that the balance property is not satisfied after insertion

Wlog, we inserted an element to the left-sub-tree of $A$

$B$: the root of left-sub-tree of $A$

case 1: we inserted an element to the left-sub-tree of $B$
Maintain Balance Property After Insertion

- **$A$:** the deepest node such that the balance property is not satisfied after insertion
- **Wlog, we inserted an element to the left-sub-tree of $A$**
- **$B$:** the root of left-sub-tree of $A$
- **case 1:** we inserted an element to the left-sub-tree of $B$

\[
\begin{array}{c}
\text{A} \\
(\text{h + 2}) \\
\text{B} \\
(\text{h + 1}) \\
\text{B}_L \\
\text{B}_R \\
\end{array}
\]
A: the deepest node such that the balance property is not satisfied after insertion

Wlog, we inserted an element to the left-sub-tree of A

B: the root of left-sub-tree of A

case 1: we inserted an element to the left-sub-tree of B
Maintain Balance Property After Insertion

- **A**: the deepest node such that the balance property is not satisfied after insertion
- **Wlog, we inserted an element to the left-sub-tree of A**
- **B**: the root of left-sub-tree of A
- **case 1**: we inserted an element to the left-sub-tree of B
Maintain Balance Property After Insertion

- **A**: the deepest node such that the balance property is not satisfied after insertion
- **Wlog**, we inserted an element to the left-sub-tree of **A**
- **B**: the root of left-sub-tree of **A**
- case 1: we inserted an element to the left-sub-tree of **B**
Maintain Balance Property After Insertion

- \( A \): the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of \( A \)
- \( B \): the root of left-sub-tree of \( A \)
Maintain Balance Property After Insertion

- $A$: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of $A$
- $B$: the root of left-sub-tree of $A$
- case 2: we inserted an element to the right-sub-tree of $B$
Maintain Balance Property After Insertion

- $A$: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of $A$
- $B$: the root of left-sub-tree of $A$
- case 2: we inserted an element to the right-sub-tree of $B$
- $C$: the root of right-sub-tree of $B$
A: the deepest node such that the balance property is not satisfied after insertion
Wlog, we inserted an element to the left-sub-tree of A
B: the root of left-sub-tree of A
case 2: we inserted an element to the right-sub-tree of B
C: the root of right-sub-tree of B
Maintain Balance Property After Insertion

- **A**: the deepest node such that the balance property is not satisfied after insertion
- **Wlog**, we inserted an element to the left-sub-tree of **A**
- **B**: the root of left-sub-tree of **A**
- **case 2**: we inserted an element to the right-sub-tree of **B**
- **C**: the root of right-sub-tree of **B**
Maintain Balance Property After Insertion

- \( A \): the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of \( A \)
- \( B \): the root of left-sub-tree of \( A \)
- case 2: we inserted an element to the right-sub-tree of \( B \)
- \( C \): the root of right-sub-tree of \( B \)
- $A$: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of $A$
- $B$: the root of left-sub-tree of $A$
- case 2: we inserted an element to the right-sub-tree of $B$
- $C$: the root of right-sub-tree of $B$
Maintain Balance Property After Insertion

- **A**: the deepest node such that the balance property is not satisfied after insertion
- **Wlog, we inserted an element to the left-sub-tree of** A
- **B**: the root of left-sub-tree of **A**
- **case 2**: we inserted an element to the right-sub-tree of **B**
- **C**: the root of right-sub-tree of **B**
Maintain Balance Property After Insertion

- **A**: the deepest node such that the balance property is not satisfied after insertion
- **Wlog**, we inserted an element to the left-sub-tree of **A**
- **B**: the root of left-sub-tree of **A**
- case 2: we inserted an element to the right-sub-tree of **B**
- **C**: the root of right-sub-tree of **B**
count-inversions($A, n$)

1. $T \leftarrow \text{empty AVL tree}$
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. $c \leftarrow c + T.\text{count-less-than}(A[i])$
5. $T.\text{insert}(A[i])$
6. return $c$

Each operation (insert, delete, count-less-than, etc.) takes time $O(h) = O(\lg n)$. Running time $= O(n \lg n)$.
count-inversions($A, n$)

1. $T \leftarrow$ empty AVL tree
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. $c \leftarrow c + T$.count-less-than($A[i]$)
5. $T$.insert($A[i]$)
6. return $c$

- Each operation (insert, delete, count-less-than, etc.) takes time $O(h) = O(lg n)$. 
count-inversions(A, n)

1. $T \leftarrow$ empty AVL tree
2. $c \leftarrow 0$
3. for $i \leftarrow n$ downto 1
4. $c \leftarrow c + T$.count-less-than($A[i]$)
5. $T$.insert($A[i]$)
6. return $c$

- Each operation (insert, delete, count-less-than, etc.) takes time $O(h) = O(\lg n)$.
- Running time = $O(n \lg n)$
Fibonacci Numbers

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

**n-th Fibonacci Number**

**Input:** integer $n > 0$

**Output:** $F_n$
Computing $F_n$: Stupid Divide-and-Conquer Algorithm

Fib($n$)

1. if $n = 0$ return 0
2. if $n = 1$ return 1
3. return Fib($n - 1$) + Fib($n - 2$)

Q: Is the running time of the algorithm polynomial or exponential in $n$?
Computing $F_n$ : Stupid Divide-and-Conquer Algorithm

\[ \text{Fib}(n) \]
\begin{enumerate}
\item if $n = 0$ return 0
\item if $n = 1$ return 1
\item return Fib($n - 1$) + Fib($n - 2$)
\end{enumerate}

Q: Is the running time of the algorithm polynomial or exponential in $n$?

A: Exponential
## Computing $F_n$ : Stupid Divide-and-Conquer Algorithm

The Fibonacci function $F(n)$ can be defined recursively as follows:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if $n = 0$ return 0</td>
</tr>
<tr>
<td>2</td>
<td>if $n = 1$ return 1</td>
</tr>
<tr>
<td>3</td>
<td>return $Fib(n - 1) + Fib(n - 2)$</td>
</tr>
</tbody>
</table>

**Q:** Is the running time of the algorithm polynomial or exponential in $n$?

**A:** Exponential

- Running time is at least $\Omega(F_n)$
### Computing \( F_n \): Stupid Divide-and-Conquer Algorithm

#### \( \text{Fib}(n) \)

1. if \( n = 0 \) return 0
2. if \( n = 1 \) return 1
3. return \( \text{Fib}(n - 1) + \text{Fib}(n - 2) \)

**Q:** Is the running time of the algorithm polynomial or exponential in \( n \)?

**A:** Exponential

- Running time is at least \( \Omega(F_n) \)
- \( F_n \) is exponential in \( n \)
Computing $F_n$: Reasonable Algorithm

**Fib($n$)**

1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i \leftarrow 2$ to $n$ do
4.     $F[i] \leftarrow F[i - 1] + F[i - 2]$
5. return $F[n]$

- Dynamic Programming
Computing $F_n$: Reasonable Algorithm

Fib($n$)

1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i \leftarrow 2$ to $n$ do
4.   $F[i] \leftarrow F[i - 1] + F[i - 2]$
5. return $F[n]

- Dynamic Programming
- Running time = ?
Computing $F_n$: Reasonable Algorithm

**Fib($n$)**

1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i \leftarrow 2$ to $n$ do
4.   $F[i] \leftarrow F[i - 1] + F[i - 2]$
5. return $F[n]$

- Dynamic Programming
- Running time $= O(n)$
Computing $F_n$: Even Better Algorithm

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
F_{n-1} \\
F_{n-2}
\end{pmatrix}
\]

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^2
\begin{pmatrix}
F_{n-2} \\
F_{n-3}
\end{pmatrix}
\]

\[
\vdots
\]

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^{n-1}
\begin{pmatrix}
F_1 \\
F_0
\end{pmatrix}
\]
power\( (n) \) 

1. if \( n = 0 \) then return \( \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \) 
2. \( R \leftarrow \text{power}(\lfloor n/2 \rfloor) \) 
3. \( R \leftarrow R \times R \) 
4. if \( n \) is odd then \( R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \) 
5. return \( R \) 

Fib\( (n) \) 

1. if \( n = 0 \) then return 0 
2. \( M \leftarrow \text{power}(n - 1) \) 
3. return \( M[1][1] \)
power($n$)

1. if $n = 0$ then return \[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\]
2. $R \leftarrow \text{power}(\lfloor n/2 \rfloor)$
3. $R \leftarrow R \times R$
4. if $n$ is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5. return $R$

Fib($n$)

1. if $n = 0$ then return 0
2. $M \leftarrow \text{power}(n - 1)$
3. return $M[1][1]$

- Recurrence for running time?
**power\( (n) \)**

1. if \( n = 0 \) then return \[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\]
2. \( R \leftarrow \text{power}(\lfloor n/2 \rfloor) \)
3. \( R \leftarrow R \times R \)
4. if \( n \) is odd then \( R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \)
5. return \( R \)

**Fib\( (n) \)**

1. if \( n = 0 \) then return 0
2. \( M \leftarrow \text{power}(n - 1) \)
3. return \( M[1][1] \)

- Recurrence for running time? \( T(n) = T(n/2) + O(1) \)
### power(n)

1. if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. $R \leftarrow \text{power}(\lfloor n/2 \rfloor)$
3. $R \leftarrow R \times R$
4. if $n$ is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5. return $R$

### Fib(n)

1. if $n = 0$ then return 0
2. $M \leftarrow \text{power}(n - 1)$
3. return $M[1][1]$

**Recurrence for running time?**

- $T(n) = T(n/2) + O(1)$
- $T(n) = O(\lg n)$
Running time $= O(\lg n)$: We Cheated!

We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time. Even printing $F(n)$ requires time much larger than $O(\lg n)$.

Fixing the Problem

To compute $F_n$, we need $O(\lg n)$ basic arithmetic operations on integers.
Running time = $O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?
Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$
Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We cannot add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time.
Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time
- Even printing $F(n)$ requires time much larger than $O(\lg n)$
Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We cannot add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time.
- Even printing $F(n)$ requires time much larger than $O(\lg n)$.

Fixing the Problem

To compute $F_n$, we need $O(\lg n)$ basic arithmetic operations on integers.
Summary: 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
Summary: 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
- Write down recurrence for running time
- Solve recurrence using master theorem
Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, \ldots:
  \[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]
Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, \cdots:
  \[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n) \]

- Integer Multiplication:
  \[ T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\lg_2 3}) \]
Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, ⋅⋅⋅:
  \[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]

- Integer Multiplication:
  \[ T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\log_2 3}) \]

- Matrix Multiplication:
  \[ T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) = O(n^{\log_2 7}) \]
Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, · · ·:
  \[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]

- Integer Multiplication:
  \[ T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\log_2 3}) \]

- Matrix Multiplication:
  \[ T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) = O(n^{\log_2 7}) \]

- Usually, designing better algorithm for “combine” step is key to improve running time