Sorting

Chapter 7
Quick Sort

- One of the most popular fast sorting algorithms
- Quick sort overcomes the drawback of merge sort of creating an additional array
- Generally, quick sort is the most efficient algorithm for large arrays
Quick Sort

- Pick any element in the array (call it the pivot)
- Place the pivot in its correct position in the array
- Move all smaller elements to the left
- Move all bigger elements to the right
- Sort the left and right sides recursively
Quick Sort Example

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>5</th>
<th>1</th>
<th>3</th>
<th>20</th>
<th>13</th>
<th>7</th>
<th>2</th>
<th>12</th>
</tr>
</thead>
</table>
Quick Sort Example

Select the pivot

8  10  5  1  3  20  13  7  2  12

pivot
Quick Sort Example

Partition the array

8 10 5 1 3 20 13 7 2 12

pivot
Quick Sort Example

Recursively sort both sides

5 1 3 7 2 8 10 20 13 12
Selecting the Pivot

- What would be a good pivot?
- What would be a bad pivot?
- What is the ideal pivot?
- Naïve selection: First element in the list
- A good selection: A random element
- A better selection: Median-of-three
# Median-of-three

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>5</th>
<th>1</th>
<th>3</th>
<th>20</th>
<th>13</th>
<th>7</th>
<th>2</th>
<th>12</th>
</tr>
</thead>
</table>

pivot

<table>
<thead>
<tr>
<th>46</th>
<th>15</th>
<th>10</th>
<th>6</th>
<th>9</th>
<th>15</th>
<th>2</th>
<th>5</th>
<th>18</th>
<th>12</th>
</tr>
</thead>
</table>

pivot

<table>
<thead>
<tr>
<th>15</th>
<th>48</th>
<th>29</th>
<th>18</th>
<th>1</th>
<th>19</th>
<th>33</th>
<th>23</th>
<th>27</th>
<th>41</th>
</tr>
</thead>
</table>

pivot
Partitioning

- The key idea about Quick Sort is to make an in-place partitioning
- Take the pivot out of the way
- Move bigger elements to the right
- Move smaller elements to the left
- Replace the pivot at its place
Partitioning Example

8  10  5  1  11  20  13  7  22  12

pivot
Partitioning Example

8 10 15 1 12 20 13 7 22 11

pivot
## Partitioning Example

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>15</th>
<th>1</th>
<th>12</th>
<th>20</th>
<th>13</th>
<th>7</th>
<th>22</th>
<th>11</th>
</tr>
</thead>
</table>

- **i**
- **j**
- **pivot**
### Partitioning Example

| 8 | 10 | 15 | 1 | 12 | 20 | 13 | 7 | 22 | 11 |

- **i**
- **j**
- **pivot**
Partitioning Example

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>10</th>
<th>15</th>
<th>1</th>
<th>12</th>
<th>20</th>
<th>13</th>
<th>7</th>
<th>22</th>
<th>11</th>
</tr>
</thead>
</table>

$i$, $j$, pivot
### Partitioning Example

<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>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
<td>15</td>
<td>1</td>
<td>12</td>
<td>20</td>
<td>13</td>
<td>7</td>
<td>22</td>
<td>11</td>
</tr>
</tbody>
</table>

- **i**
- **j**
- **pivot**
Partitioning Example

8  10  15  1  12  20  13  7  22  11

i

j  pivot
Partitioning Example

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>15</th>
<th>1</th>
<th>12</th>
<th>20</th>
<th>13</th>
<th>7</th>
<th>22</th>
<th>11</th>
</tr>
</thead>
</table>

Partitioning Example

i

j

pivot
Partitioning Example

The diagram illustrates a partitioning example with an array of numbers: 8, 10, 15, 1, 12, 20, 13, 7, 22, 11. The pivot element is 11. The indices i and j are used to show the partitioning process. The array is rearranged such that all elements less than the pivot are to its left, and all elements greater than the pivot are to its right.
Partitioning Example

| 8 | 10 | 7 | 1 | 12 | 20 | 13 | 15 | 22 | 11 |

i

j

pivot
Partitioning Example

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>7</th>
<th>1</th>
<th>12</th>
<th>20</th>
<th>13</th>
<th>15</th>
<th>22</th>
<th>11</th>
</tr>
</thead>
</table>

- $i$
- $j$
- pivot
Partitioning Example
# Partitioning Example

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>7</th>
<th>1</th>
<th>12</th>
<th>20</th>
<th>13</th>
<th>15</th>
<th>22</th>
<th>11</th>
</tr>
</thead>
</table>

Partitioning example with pivot element 11.

Indices: $i$ for the left pointer, $j$ for the right pointer.

- $i$: Initially at the start of the array.
- $j$: Initially at the end of the array.
- **pivot**: Element 11.

The algorithm aims to place all elements less than the pivot to the left of $i$ and all elements greater than or equal to the pivot to the right of $j$. The process continues recursively until the entire array is partitioned.
Partitioning Example

```
| 8 | 10 | 7 | 1 | 12 | 20 | 13 | 15 | 22 | 11 |
```

- **i**
- **j**
- **pivot**
Partitioning Example

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>7</th>
<th>1</th>
<th>12</th>
<th>20</th>
<th>13</th>
<th>15</th>
<th>22</th>
<th>11</th>
</tr>
</thead>
</table>

i

j

pivot
Partitioning Example

| 8 | 10 | 7 | 1 | 12 | 20 | 13 | 15 | 22 | 11 |

pivot

i \quad j
Partitioning Example

8  10  7  1  12  20  13  15  22  11

pivot

i

j
Partitioning Example

8 10 7 1 12 20 13 15 22 11

i and j are reversed!
Partitioning Example

![Partitioning Example Diagram]
Analysis of Quick Sort

- Cost of the partitioning step
  - \( O(n) \): One scan over the list

- Worst case
  - The sizes of the two sublists are 0 and \( n-1 \)
  - \( O(n^2) \)

- Best case
  - The sizes of the two sublists are almost equal
  - \( O(n \log n) \)

- Average case
  - None of the lists is excessively large or small
  - \( O(n \log n) \)
## Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case</th>
<th>Best-case</th>
<th>Average</th>
<th>Stable*</th>
<th>In-place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bubble</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*A sorting algorithm is said to be stable if equal items remain in the same order after sorting*
# Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case</th>
<th>Best-case</th>
<th>Average</th>
<th>Stable*</th>
<th>In-place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Selection</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bubble</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shell</td>
<td>$O(n^{3/2})$</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Quick</td>
<td>$O(n^2)**$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

*A sorting algorithm is said to be stable if equal items remain in the same order after sorting

** Can be reduced to $O(n \log n)$ with a smart pivot selection algorithm
Lower Bound for Sorting

- Can we create a sorting algorithm with an asymptotic running time that is lower than $O(n \log n)$ in the worst case?

- Assumptions
  - Array elements can be in any order
  - Only comparisons are used to sort elements
Decision Tree (Execution Tree)

Initial state

Comparison

Number of comparisons

Execution terminated
Worst-case

› Deepest part of the tree

› Number of leaf nodes
  › Equal to total number of permutations
  › n!

› Height of the tree
  › log(n!)

log(n!) = \Omega(n \log n)

n \log n is a lower bound for any comparison-based sorting algorithm