Quick-Sort
Quick-Sort

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- **Divide**: pick a random element \( x \) (called pivot) and partition \( S \) into
  - \( L \) elements less than \( x \)
  - \( E \) elements equal \( x \)
  - \( G \) elements greater than \( x \)
- **Recur**: sort \( L \) and \( G \)
- **Conquer**: join \( L, E \) and \( G \)
Importance of Partitioning

After partitioning

- What can you say about the position of the pivot?
Importance of Partitioning

- After partitioning
  - What can you say about the position of the pivot?
    - The pivot is at the correct spot
  - Also, two smaller subproblems
    - Not including the pivot
Partition

- Partition an input sequence:
  - remove each element \( y \) from \( S \) and
  - insert \( y \) into \( L, E \) or \( G \), depending on the result of the comparison with the pivot \( x \)

- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes \( O(1) \) time

- Partition step of quick-sort takes \( O(n) \) time

Algorithm \( \text{partition}(S, p) \)

- Input sequence \( S \), position \( p \) of pivot
- Output subsequences \( L, E, G \) of the elements of \( S \) less than, equal to, or greater than the pivot, resp.

\[
L, E, G \leftarrow \text{empty sequences}
\]

\[
x \leftarrow S.remove(p)
\]

while \( \neg S.isEmpty() \)

\[
y \leftarrow S.remove(S.first())
\]

if \( y < x \)

\[
L.addLast(y)
\]

else if \( y = x \)

\[
E.addLast(y)
\]

else \{ \( y > x \) \}

\[
G.addLast(y)
\]

return \( L, E, G \)
Partition the list recursively
Merge the lists and the pivot
Worst-case Time Complexity

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of $L$ and $G$ has size $n - 1$ and the other has size 0.
- The running time is proportional to the sum $n + (n - 1) + \ldots + 2 + 1$.
- Thus, the worst-case running time of quick-sort is $O(n^2)$.

<table>
<thead>
<tr>
<th>depth</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n - 1$</td>
<td>1</td>
</tr>
</tbody>
</table>
Expected Time Complexity

- \( \mathcal{O}(n \log n) \)

- Proof in the book
  - And skipped slides at the end
In-place Quick Sort

- O(1) extra space
- Same basic algorithm
  - Partition based on a pivot
  - Quick Sort on the two partitions
- Partitioning uses O(1) extra space
  - Left and right indices to scan for elements on the “wrong side”:
    - Smaller elements that are on the right side
    - Larger element that are on the left side
<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>67</td>
<td>23</td>
</tr>
<tr>
<td>87</td>
<td>98</td>
<td>43</td>
</tr>
<tr>
<td>23</td>
<td>43</td>
<td>56</td>
</tr>
<tr>
<td>left</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>87</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>87</td>
<td>23</td>
<td>98</td>
</tr>
<tr>
<td>23</td>
<td>98</td>
<td>43</td>
</tr>
<tr>
<td>98</td>
<td>43</td>
<td>56</td>
</tr>
<tr>
<td>left</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td>left</td>
<td>right</td>
<td>pivot</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>23</td>
<td>98</td>
<td>43</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>43</td>
<td>98</td>
<td>43</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td>67</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td>23</td>
<td>98</td>
<td>67</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>23</td>
</tr>
<tr>
<td>Left</td>
<td>Right</td>
<td>Pivot</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>23</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>23</td>
</tr>
<tr>
<td>left</td>
<td>right</td>
<td>pivot</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>23</td>
</tr>
<tr>
<td>34</td>
<td>67</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>23</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>23</td>
</tr>
<tr>
<td>34</td>
<td>43</td>
<td>23</td>
</tr>
</tbody>
</table>
In-Place Quick-Sort

Algorithm \texttt{inPlaceQuickSort}(S, start, end)

\textbf{Input} sequence \(S\), \texttt{start} and \texttt{end} indices

\textbf{Output} sequence \(S\) sorted between \texttt{start} and \texttt{end}

\textbf{if} \texttt{start} \(\geq\) \texttt{end} \textbf{ return}

\( \text{left} \leftarrow \text{start} \)

\( \text{right} \leftarrow \text{end} - 1 \) // before pivot

\( \text{pivot} \leftarrow S[\text{end}] \) // pivot is the last element

\textbf{while} \texttt{left} \(\leq\) \texttt{right} // still have elements

\textbf{while} (\texttt{left} \(\leq\) \texttt{right} \& \& \(S[\text{left}] < \text{pivot}\)) // find element larger than pivot

\texttt{left}++

\textbf{while} (\texttt{left} \(\leq\) \texttt{right} \& \& \(S[\text{right}] > \text{pivot}\)) // find element smaller than pivot

\texttt{right}--

\textbf{if} (\texttt{left} \(\leq\) \texttt{right}) // put the two elements in the correct partitions

\( \text{swap } S[\text{left}] \text{ and } S[\text{right}]; \texttt{left}++; \texttt{right}--; \)

\( \text{Swap } S[\text{end}] \text{ and } S[\text{left}] \) // put pivot at the correct spot

\( \texttt{inPlaceQuickSort}\!(S, \text{start}, \text{left} - 1) \)

\( \texttt{inPlaceQuickSort}\!(S, \text{left} + 1, \text{end}) \)
Selection of Pivots

- Last element (or first element)
  - If the list is partially sorted
    - might be the smallest/largest element
  - the worst-case scenario

- Ideas?
Selection of Pivots

- Last element (or first element)
  - If the list is partially sorted
    - might be the smallest/largest element
  - the worst-case scenario

- Random element
  - But calling random() has time overhead

- Median-of-three
  - Median of first, last, and middle elements
### Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>- in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- slow (good for small inputs)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>- in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- slow (good for small inputs)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \log n)$</td>
<td>- in-place, randomized</td>
</tr>
<tr>
<td></td>
<td>expected</td>
<td>- fastest (good for large inputs)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>- in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>- sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- fast (good for huge inputs)</td>
</tr>
</tbody>
</table>
Skipping the rest
Expected Running Time

Consider a recursive call of quick-sort on a sequence of size $s$

- **Good call**: the sizes of $L$ and $G$ are each less than $3s/4$
- **Bad call**: one of $L$ and $G$ has size greater than $3s/4$

A call is **good** with probability $1/2$

- $1/2$ of the possible pivots cause good calls:
Expected Running Time, Part 2

- **Probabilistic Fact:** The expected number of coin tosses required in order to get $k$ heads is $2^k$

- For a node of depth $i$, we expect:
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$

- Therefore, we have:
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$

- The amount or work done at the nodes of the same depth is $O(n)$

- Thus, the expected running time of quick-sort is $O(n \log n)$

\[ \text{Expected height} = O(\log n) \]
\[ \text{time per level} = O(n) \]
\[ \text{total expected time:} \quad O(n \log n) \]
Quick-Sort Tree

- An execution depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1

```
7 4 9 6 2 → 2 4 6 7 9

4 2 → 2 4
7 9 → 7 9
2 → 2
9 → 9
```
Execution Example

Pivot selection

```
7 2 9 4 3 7 6 1
```

Diagram showing the execution example of quick-sort with pivot selection.
Execution Example (cont.)

Partition, recursive call, pivot selection

7 2 9 4 3 7 6 1

2 4 3 1
Execution Example (cont.)

Partition, recursive call, base case

\[ 7 2 9 4 3 7 6 1 \]

\[ 2 4 3 1 \]

\[ 1 \rightarrow 1 \]
Execution Example (cont.)

Recursive call, ..., base case, join

```
7 2 9 4 3 7 6 1
```

```
2 4 3 1 → 1 2 3 4
```

```
1 → 1
```

```
4 3 → 3 4
```

```
4 → 4
```

```
1 → 1
```

```
4 3 → 3 4
```

```
4 → 4
```
Execution Example (cont.)

Recursive call, pivot selection

7 2 9 4 3 7 6 1

2 4 3 1 \rightarrow 1 2 3 4

1 \rightarrow 1

4 3 \rightarrow 3 4

4 \rightarrow 4

7 9 7
Execution Example (cont.)

Partition, ..., recursive call, base case
Execution Example (cont.)

Join, join

\[ 7 \ 2 \ 9 \ 4 \ 3 \ 7 \ 6 \ 1 \rightarrow 1 \ 2 \ 3 \ 4 \ 6 \ 7 \ 7 \ 9 \]

\[ 2 \ 4 \ 3 \ 1 \rightarrow 1 \ 2 \ 3 \ 4 \]

\[ 1 \rightarrow 1 \]

\[ 4 \ 3 \rightarrow 3 \ 4 \]

\[ 4 \rightarrow 4 \]

\[ 7 \ 9 \ 7 \rightarrow 7 \ 7 \ 9 \]

\[ 9 \rightarrow 9 \]
In-Place Partitioning

Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

\[
\begin{array}{c}
\text{j} \\
3 \ 2 \ 5 \ 1 \ 0 \ 7 \ 3 \ 5 \ 9 \ 2 \ 7 \ 9 \ 8 \ 9 \ 7 \ 6 \ 9 \\
\text{k}
\end{array}
\]

(pivot = 6)

Repeat until j and k cross:
- Scan j to the right until finding an element \( \geq x \).
- Scan k to the left until finding an element \( < x \).
- Swap elements at indices j and k