Quick-Sort

Quick-Sort
Outline and Reading

- Quick-sort (§4.3)
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort (4.3.1)
- In-place quick-sort (§4.8)
- Summary of sorting algorithms
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 $G$
Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element $y$ from $S$ and
  - We 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

- Thus, the 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.

$L$, $E$, $G \leftarrow$ empty sequences

$x \leftarrow S.remove(p)$

while $\neg S.isEmpty()$

  $y \leftarrow S.remove(S.first())$

  if $y < x$
    $L.insertLast(y)$
  else if $y = x$
    $E.insertLast(y)$
  else
    $G.insertLast(y)$

return $L$, $E$, $G$
Quick-Sort Tree

An execution of quick-sort is 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
Execution Example (cont.)

Partition, recursive call, pivot selection

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

```
2 4 3 1
```

```
2 4 3 1
```

```
1 3 8 6
```

```
1 3 8 6
```

```
2
```

```
2
```
Execution Example (cont.)

Partition, recursive call, base case

![Diagram of quicksort execution example]
Execution Example (cont.)

Recursive call, ..., base case, join
Execution Example (cont.)

Recursive call, pivot selection

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

```
2 4 3 1 → 1 2 3 4
```

```
1 → 1
```

```
4 3 → 3 4
```

```
4 → 4
```

```
7 9 7
```

```
```

```
```

Quick-Sort 10
Execution Example (cont.)

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

Join, join

\[
\begin{array}{c}
7 & 2 & 9 & 4 & 3 & 7 & 6 & 1 \\
\end{array}
\rightarrow
\begin{array}{c}
1 & 2 & 3 & 4 & 6 & 7 & 7 & 9 \\
\end{array}
\]
**Worst-case Running Time**

- 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>

**Diagram:**

- The diagram illustrates the recursive nature of the quick-sort algorithm, showing the depth versus time for different splits.
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:

012345678910111213141516

---

**Bad pivots** | **Good pivots** | **Bad pivots**
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 of work done at the nodes of the same depth is $O(n)$

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

Total expected time: $O(n \log n)$
In-Place Quick-Sort

Quick-sort can be implemented to run in-place.

In the partition step, we use replace operations to rearrange the elements of the input sequence such that:

- the elements less than the pivot have rank less than $h$
- the elements equal to the pivot have rank between $h$ and $k$
- the elements greater than the pivot have rank greater than $k$

The recursive calls consider:

- elements with rank less than $h$
- elements with rank greater than $k$

---

Algorithm $inPlaceQuickSort(S, l, r)$

**Input** sequence $S$, ranks $l$ and $r$

**Output** sequence $S$ with the elements of rank between $l$ and $r$ rearranged in increasing order

1. if $l \geq r$
   - return
2. $i \leftarrow$ a random integer between $l$ and $r$
3. $x \leftarrow S\.elemAtRank(i)$
4. $(h, k) \leftarrow inPlacePartition(x)$
5. $inPlaceQuickSort(S, l, h - 1)$
6. $inPlaceQuickSort(S, k + 1, r)$
In-Place Partitioning

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

$$3 \ 2 \ 5 \ 1 \ 0 \ 7 \ 3 \ 5 \ 9 \ 2 \ 7 \ 9 \ 8 \ 9 \ 7 \ \boxed{6} \ 9$$

(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 $k$
## 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, slow (good for small inputs)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \log n)$ expected</td>
<td>in-place, randomized, fastest (good for large inputs)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>sequential data access, fast (good for huge inputs)</td>
</tr>
</tbody>
</table>