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 to \( x \)
  - \( G \) elements greater than \( x \)
- Recur: sort \( L \) and \( G \)
- Conquer: join \( L \), \( E \) and \( G \)

Outline and Reading

- Quick-sort (§4.3)
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort (4.3.1)

Execution Example

Pivot selection

7 2 9 4 3 7 8 1

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

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.

1. \( L \), \( E \), \( G \leftarrow \) empty sequences
2. \( x \leftarrow S \).remove\( (p) \)
3. while \( \neg S \).isEmpty
   4. \( y \leftarrow S \).remove\( S \).first\( () \)
5. if \( y < x \)
  6. \( L \).insertLast\( (y) \)
else if \( y = x \)
  7. \( E \).insertLast\( (y) \)
else \( y > x \)
  8. \( G \).insertLast\( (y) \)
9. return \( L \), \( E \), \( G \)
Execution Example (cont.)

Partition, recursive call, pivot selection

2

4 3 1

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

\[ 2 4 3 1 \]

\[ 1 \rightarrow 1 \]

\[ 1 \rightarrow 0 \]

\[ 1 \rightarrow 1 \]

\[ 4 2 \rightarrow 2 1 \]

Execution Example (cont.)

Partition, recursive call, base case

2

4 3 1

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

\[ 2 4 3 1 \]

\[ 1 \rightarrow 1 \]

\[ 1 \rightarrow 0 \]

\[ 4 2 \rightarrow 2 1 \]

Execution Example (cont.)

Recursive call, ..., base case, join

2

4 3 1

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

\[ 2 4 3 1 \]

\[ 1 \rightarrow 1 \]

\[ 1 \rightarrow 0 \]

\[ 4 2 \rightarrow 2 1 \]

Execution Example (cont.)

Recursive call, pivot selection

2

4 3 1

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

\[ 2 4 3 1 \]

\[ 1 \rightarrow 1 \]

\[ 1 \rightarrow 0 \]

\[ 4 2 \rightarrow 2 1 \]

Execution Example (cont.)

Partition, ... recursive call, base case

2

4 3 1

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

\[ 2 4 3 1 \]

\[ 1 \rightarrow 1 \]

\[ 1 \rightarrow 0 \]

\[ 4 2 \rightarrow 2 1 \]

Execution Example (cont.)

Join, join

2

4 3 1

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

\[ 2 4 3 1 \]

\[ 1 \rightarrow 1 \]

\[ 4 2 \rightarrow 2 1 \]
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

$$\sum_{i=0}^{n-1} i$$

Thus, the worst-case running time of quick-sort is $O(n^2)$

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 $1/2$

Comparison-Based Sorting (§ 4.4)

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...

Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort $n$ elements, $x_1, x_2, \ldots, x_n$.

Counting Comparisons

- Let us just count comparisons then.
  - Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree

Decision Tree Height

- The height of this decision tree is a lower bound on the running time
- Every possible input permutation must lead to a separate leaf output.
  - If not, some input of bad orderings would cause good calls:
- The height is at least $\log(n!)$
The Lower Bound

- Any comparison-based sorting algorithms takes at least \( \log(n!) \) time.
- Therefore, any such algorithm takes time at least
  \[
  \log (n!) \geq \log \left( \frac{n}{2} \right) = (n/2) \log (n/2).
  \]
- That is, any comparison-based sorting algorithm must run in \( \Omega(n \log n) \) time.

Bucket-Sort and Radix-Sort

- Key range \([0, 9]\)
- Phase 1: Move items into buckets
  - Phase 1 takes \( O(n) \) time
  - Phase 2 takes \( O(n + N) \) time
- Bucket-sort uses the keys as indices into an auxiliary array \( B \) of sequences (buckets); \( N \) total buckets.
- Phase 1: Empty sequence \( S \) by moving each item \((k, o)\) into its bucket \( B[k] \)
- Phase 2: For \( i = 0, ..., N - 1 \), move the items of bucket \( B[i] \) to the end of sequence \( S \)

Bucket-Sort Properties

- Keys have a fixed range of values.
- Keys are NOT compared.
- bucketSort is a stable sort.
- Stable Sort Property:
  - Any two items with the same key will be in the same relative order after sorting.

Algorithm bucketSort(S, N)

Input: sequence \( S \) of (key, element) items with keys in the range \([0, N - 1]\)
Output: sequence \( S \) sorted by increasing keys
\( B \leftarrow \) array of \( N \) empty sequences
while \( \neg \text{isEmpty}(S) \)
  \( f \leftarrow S\text{first()} \)
  \( (k, o) \leftarrow S\text{remove}(f) \)
  \( B[k].\text{insertLast}(f) \)
for \( i \leftarrow 0 \) to \( N - 1 \)
while \( \neg \text{isEmpty}(B[i]) \)
  \( f \leftarrow B[i]\text{first()} \)
  \( (k, o) \leftarrow B[i]\text{remove}(f) \)
  \( S\text{insertLast}(f) \)
Lexicographic Order

- A $d$-tuple is a sequence of $d$ keys $(k_1, k_2, ..., k_d)$, where key $k_i$ is said to be the $i$-th dimension of the tuple.

Example:

The Cartesian coordinates of a point in space are a 3-tuple $(x_1, x_2, x_3)$.

The lexicographic order of two $d$-tuples is recursively defined as follows:

$(x_1, x_2, ..., x_d) < (y_1, y_2, ..., y_d) \iff x_1 < y_1 \lor x_1 = y_1 \land (x_2, ..., x_d) < (y_2, ..., y_d)$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

Radix-Sort (§ 4.5.2)

- Radix-sort uses bucket-sort to sort each dimension in a stable manner.

Radix-sort is applicable to tuples where the keys in each dimension $i$ are integers in the range $[0, N-1]$.

Radix-sort runs in time $O(d(n+N))$.

Algorithm `radixSort(S, N)`

Input sequence $S$ of $d$-tuples such that $(0, ..., 0) \leq (x_1, ..., x_d)$ and $(x_1, ..., x_d) \leq (N-1, ..., N-1)$ for each tuple $(x_1, ..., x_d)$ in $S$.

Output sequence $S$ sorted in lexicographic order.

for $i \leftarrow d$ downto 1
    bucketSort($S$, $N$, $i$)

Example:

$(7,4,6) (5,1,5) (2,4,6) (2,1,4) (3,2,4)$

$(2,1,4) (3,2,4) (5,1,5) (7,4,6) (2,4,6)$

$(2,1,4) (2,4,6) (3,2,4) (5,1,5) (7,4,6)$

Example

Sorting a sequence of 4-digit integers

Algorithm `base10RadixSort(S)`

Input sequence $S$ of $d$-digit integers

Output sequence $S$ sorted

replace each element $x$ of $S$ with the item $(0, x)$

for $i \leftarrow 0$ to $d-1$
    replace the key $k$ of each item $(k, x)$ of $S$ with digit $x_i$ of $x$

bucketSortS(S, 10)