Bucket-Sort and Radix-Sort


© 2014 Goodrich, Tamassia, Goldwasser
Many sorting algorithms are comparison based. They sort by making comparisons between pairs of objects. Examples: 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, $x_1, x_2, \ldots, x_n$. 

Diagram:

- Is $x_i < x_j$?
  - yes
  - no
Let us just count comparisons then. Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree.
The height of the decision tree is a lower bound on the running time

Every input permutation must lead to a separate leaf output

If not, some input \ldots 4 \ldots 5 \ldots would have same output ordering as \ldots 5 \ldots 4 \ldots, which would be wrong

Since there are \( n! = 1 \cdot 2 \cdot \ldots \cdot n \) leaves, the height is at least \( \log(n!) \)
Any comparison-based sorting algorithms takes at least $\log(n!)$ time.

Therefore, any such algorithm takes time at least

$$\log (n!) \geq \log \left( \frac{n}{2} \right)^{\frac{n}{2}} = (n/2) \log (n/2).$$

That is, any comparison-based sorting algorithm must run in $\Omega(n \log n)$ lower bound on its running time.
We showed that the lower bound of sorting with comparison is $\Omega (n \log n)$ time.

Can we do better? Yes, with special assumptions about the input sequence to be sorted.

We will consider the problem of sorting a sequence of entries, each a key-value pair, where the keys have a restricted type

- Bucket-Sort
- Radix-Sort
Let be $S$ be a sequence of $n$ (key, element) entries with integer keys in the range $[0, N-1]$, for some integer $N \geq 2$.

Bucket-sort uses the keys as indices into an auxiliary array $B$ of size $N$ (buckets).

- **Phase 1**: Empty sequence $S$ by moving each entry $(k, o)$ into its bucket $B[k]$.
- **Phase 2**: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of sequence $S$.

**Analysis:**
- Phase 1 takes $O(n)$ time.
- Phase 2 takes $O(n + N)$ time.

Bucket-sort takes $O(n + N)$ time.
Algorithm bucketSort(S):

*Input:* Sequence S of entries with integer keys in the range \([0, N - 1]\)

*Output:* Sequence S sorted in nondecreasing order of the keys

let B be an array of N sequences, each of which is initially empty

for each entry e in S do
  \( k = \) the key of e
  remove e from S
  insert e at the end of bucket (sequence) B[\( k \)]

for \( i = 0 \) to \( N - 1 \) do
  for each entry e in B[\( i \)] do
    remove e from B[\( i \)]
    insert e at the end of S
Bucket-Sort and Radix-Sort

**EXAMPLE**

- **Key range** $[0, 9]$

  - Phase 1:
    - $7, d \rightarrow 1, c \rightarrow 3, a \rightarrow 7, g \rightarrow 3, b \rightarrow 7, e$

  - Phase 2:
    - $1, c \rightarrow 3, a \rightarrow 3, b \rightarrow 7, d \rightarrow 7, g \rightarrow 7, e$

© 2014 Goodrich, Tamassia, Goldwasser
PROPERTIES AND EXTENSIONS

Key-type Property
- The keys are used as indices into an array and cannot be arbitrary objects
- No external comparator

Stable Sort Property
- The relative order of any two items with the same key is preserved after the execution of the algorithm

Extensions
- Integer keys in the range \([a, b]\)
  - Put entry \((k, o)\) into bucket \(B[k - a]\)
- String keys from a set \(D\) of possible strings, where \(D\) has constant size (e.g., names of the 50 U.S. states)
  - Sort \(D\) and compute the rank \(r(k)\) of each string \(k\) of \(D\) in the sorted sequence
  - Put entry \((k, o)\) into bucket \(B[r(k)]\)
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.

- 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.
When sorting key-value pairs, an important issue is how equal keys are handled. Let \( S = ((k_0,v_0), \ldots, (k_{n-1},v_{n-1})) \) be a sequence of such entries.

We say that a sorting algorithm is **stable** if, for any two entries \((k_i,v_i)\) and \((k_j,v_j)\) of \( S \) such that \( k_i = k_j \) and \((k_i,v_i)\) precedes \((k_j,v_j)\) in \( S \) before sorting (that is, \( i < j \)), entry \((k_i,v_i)\) also precedes entry \((k_j,v_j)\) after sorting.

Stability is important for a sorting algorithm because applications may want to preserve the initial order of elements with the same key.

Bucket-sort guarantees stability as long as we ensure that all sequences act as **queues**.
Lexicographic-sort sorts a sequence of \(d\)-tuples in lexicographic order by executing \(d\) times algorithm \(stableSort\), one per dimension.

Lexicographic-sort runs in \(O(dT(n))\) time, where \(T(n)\) is the running time of \(stableSort\).
Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension.

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 $O(d(n+N))$ where $d$ is the dimension of keys, $n$ is the number of data, and keys range is $[0...N-1]$.

Algorithm $\text{radixSort}(S, N)$

- **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

$\text{bucketSort}(S, N)$
Consider a sequence of \( n \) \( b \)-bit integers
\[
x = x_{b-1} \ldots x_1 x_0
\]
We represent each element as a \( b \)-tuple of integers in the range \([0, 1]\) and apply radix-sort with \( N = 2 \)
This application of the radix-sort algorithm runs in \( O(bn) \) time
For example, we can sort a sequence of 32-bit integers in linear time

Algorithm \( \text{binaryRadixSort}(S) \)

\begin{itemize}
  \item \textbf{Input} sequence \( S \) of \( b \)-bit integers
  \item \textbf{Output} sequence \( S \) sorted
  \item replace each element \( x \) of \( S \) with the item \((0, x)\)
  \item for \( i \leftarrow 0 \) to \( b - 1 \)
    \item replace the key \( k \) of each item \((k, x)\) of \( S \) with bit \( x_i \) of \( x \)
  \end{itemize}

\( \text{bucketSort}(S, 2) \)
Sorting a sequence of 4-bit integers

1001 0010 1001 1001 0001 1110
0010 1110 0001 0010 1101 1101
1101 1001 0010 1101 0001 1110
0001 1110 0010 1101 0001 1110

© 2014 Goodrich, Tamassia, Goldwasser
<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>$O(n \log n)$</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>
<tr>
<td>bucket-sort</td>
<td>$O(n+N)$</td>
<td>integer keys of range [0 … N]</td>
</tr>
<tr>
<td>radix-sort</td>
<td>$O(d(n+N))$</td>
<td>d diinteger keys of range [0 … N]</td>
</tr>
</tbody>
</table>
SELECTION PROBLEM


© 2014 Goodrich, Tamassia, Goldwasser
THE SELECTION PROBLEM

- Given an integer $k$ and $n$ elements $x_1, x_2, \ldots, x_n$, taken from a total order, find the $k$-th smallest element in this set.
- Of course, we can sort the set in $O(n \log n)$ time and then index the $k$-th element.
- Can we solve the selection problem faster?

$k=3 \quad 7 \ 4 \ 9 \ 6 \ 2 \rightarrow \ 2 \ 4 \ 6 \ 7 \ 9$
Quick-select is a randomized selection algorithm based on the prune-and-search paradigm:

- **Prune**: pick a random element $x$ (called pivot) and partition $S$ into
  - $L$: elements less than $x$
  - $E$: elements equal $x$
  - $G$: elements greater than $x$
- **Search**: depending on $k$, either answer is in $E$, or we need to recur in either $L$ or $G$

\[
\begin{align*}
|L| &< k \leq |L| + |E| \\
\text{(done)} &
\end{align*}
\]
PARTITION

- We partition an input sequence as in the quick-select algorithm:
  - 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-select takes $O(n)$ time

Algorithm $partition(S, p)$

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

  else { $y > x$ }
    $G.addLast(y)$

return $L, E, G$
An execution of quick-select can be visualized by a recursion path

Each node represents a recursive call of quick-select, and stores $k$ and the remaining sequence.

```
K=5, S=(7 4 9 3 2 6 5 1 8)

K=2, S=(7 4 9 6 5 8)

K=2, S=(7 4 6 5)

K=1, S=(7 6 5)

5
```
Consider a recursive call of quick-select 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.

Good call

Bad call
Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two.

Probabilistic Fact #2: Expectation is a linear function:

\[ E(X + Y) = E(X) + E(Y) \]
\[ E(cX) = cE(X) \]

Let \( T(n) \) denote the expected running time of quick-select.

By Fact #2,

\[ T(n) \leq T\left(\frac{3n}{4}\right) + bn^*(\text{expected \# of calls before a good call}) \]

By Fact #1,

\[ T(n) \leq T\left(\frac{3n}{4}\right) + 2bn \]

That is, \( T(n) \) is a geometric series:

\[ T(n) \leq 2bn + 2b\left(\frac{3}{4}\right)n + 2b\left(\frac{3}{4}\right)^2n + 2b\left(\frac{3}{4}\right)^3n + ... \]

So \( T(n) \) is \( O(n) \).

We can solve the selection problem in \( O(n) \) expected time.