Chapter 11: Sorting, Sets and Selection

Nancy Amato
Parasol Lab, Dept. CSE, Texas A&M University

Acknowledgement: These slides are adapted from slides provided with Data Structures and Algorithms in C++, Goodrich, Tamassia and Mount (Wiley 2004)

http://parasol.tamu.edu
Outline and Reading

• Merge Sort (§11.1)
• Quick Sort (§11.2)
• Radix Sort and Bucket Sort (§11.3)
• Selection (§11.5)

• Summary of sorting algorithms
Merge Sort
Merge Sort

- Merge sort is based on the divide-and-conquer paradigm. It consists of three steps:
  - **Divide:** partition input sequence $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
  - **Recur:** recursively sort $S_1$ and $S_2$
  - **Conquer:** merge $S_1$ and $S_2$ into a unique sorted sequence

**Algorithm** $mergeSort(S, C)$

**Input** sequence $S$, comparator $C$

**Output** sequence $S$ sorted according to $C$

if $S.size() > 1$

$(S_1, S_2) := partition(S, S.size()/2)$

$S_1 := mergeSort(S_1, C)$

$S_2 := mergeSort(S_2, C)$

$S := merge(S_1, S_2)$

return($S$)
D&C algorithm analysis with recurrence equations

• Divide-and conquer is a general algorithm design paradigm:
  • Divide: divide the input data $S$ into $k$ (disjoint) subsets $S_1, S_2, …, S_k$
  • Recur: solve the subproblems associated with $S_1, S_2, …, S_k$
  • Conquer: combine the solutions for $S_1$ and $S_2$ into a solution for $S$

• The base case for the recursion are subproblems of constant size
• Analysis can be done using recurrence equations (relations)

• When the size of all subproblems is the same (frequently the case) the recurrence equation representing the algorithm is:
  $$T(n) = D(n) + k \cdot T(n/c) + C(n)$$

• Where
  • $D(n)$ is the cost of dividing $S$ into the $k$ subproblems, $S_1, S_2, S_3, …, S_k$
  • There are $k$ subproblems, each of size $n/c$ that will be solved recursively
  • $C(n)$ is the cost of combining the subproblem solutions to get the solution for $S$
Exercise: Recurrence Eqn Setup

Algorithm: transform multiplication of two n-bit integers I and J into multiplication of (n/2)-bit integers and some additions/shifts

Algorithm \textit{Multiply}(I,J)

\textbf{Input} integers I, J (of same size)

\textbf{Output} I*J

If \texttt{I.size()} > 1 {

1. split I and J into high and low order halves: \(I_h, I_l, J_h, J_l\)

2. \(x_1 = I_h \times J_h, x_2 = I_h \times J_l, x_3 = I_l \times J_h, x_4 = I_l \times J_l\)

3. \(Z = x_1 \times 2^n + x_2 \times 2^{n/2} + x_3 \times 2^{n/2} + x_4\)

} \textit{else} \\
\(Z = I \times J\)

Return(Z)

1. Where does recursion happen in this algorithm?
2. Rewrite the step(s) of the algorithm to show this clearly.
Exercise: Recurrence Eqn Setup

Algorithm: transform multiplication of two \( n \)-bit integers \( I \) and \( J \) into multiplication of \((n/2)\)-bit integers and some additions/shifts

<table>
<thead>
<tr>
<th>Algorithm Multiply((I,J))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> integers ( I ), ( J ) (of same size)</td>
</tr>
<tr>
<td><strong>Output</strong> ( I \ast J )</td>
</tr>
<tr>
<td>If ( I \text{.size}() &gt; 1 ) {</td>
</tr>
<tr>
<td>1. split ( I ) and ( J ) into high and low order halves: ( I_h, I_l, J_h, J_l )</td>
</tr>
<tr>
<td>2. ( x_1 = \text{Multiply}(I_h,J_h), x_2 = \text{Multiply}(I_h,J_l), x_3 = \text{Multiply}(I_l,J_h), x_4 = \text{Multiply}(I_l,J_l) )</td>
</tr>
<tr>
<td>3. ( Z = x_1 \ast 2^n + x_2 \ast 2^{n/2} + x_3 \ast 2^{n/2} + x_4 )</td>
</tr>
<tr>
<td>} else</td>
</tr>
<tr>
<td>( Z = I \ast J )</td>
</tr>
<tr>
<td>Return((Z))</td>
</tr>
</tbody>
</table>

3. Assuming that additions and shifts of \( n \)-bit numbers can be done in \( O(n) \) time, describe a recurrence equation showing the running time of this multiplication algorithm.
Exercise: Recurrence Eqn Setup

Algorithm: transform multiplication of two n-bit integers I and J into multiplication of (n/2)-bit integers and some additions/shifts

Algorithm $\textit{Multiply}(I,J)$

Input integers I, J (of same size)

Output $I \times J$

If $I\text{.size()} > 1$

1. split I and J into high and low order halves: $I_h, I_l, J_h, J_l$
2. $x_1 = \text{Multiply}(I_h, J_h)$, $x_2 = \text{Multiply}(I_h, J_l)$, $x_3 = \text{Multiply}(I_l, J_h)$, $x_4 = \text{Multiply}(I_l, J_l)$
3. $Z = x_1 \times 2^n + x_2 \times 2^{n/2} + x_3 \times 2^{n/2} + x_4$

} else

$Z = I \times J$

Return($Z$)

- The recurrence equation for this algorithm is:
  - $T(n) = 4 \ T(n/2) + O(n)$
  - The solution is $O(n^2)$ which is the same as naïve algorithm
Now, back to mergesort…

- The running time of Merge Sort can be expressed by the recurrence equation:

\[ T(n) = 2T(n/2) + M(n) \]

- We need to determine \( M(n) \), the time to merge two sorted sequences each of size \( n/2 \).

**Algorithm** \( mergeSort(S, C) \)

**Input** sequence \( S \), comparator \( C \)

**Output** sequence \( S \) sorted according to \( C \)

\[
\text{if } S.\text{size()} > 1 \{ \\
(S_1, S_2) := \text{partition}(S, S.\text{size()}/2) \\
S_1 := mergeSort(S_1, C) \\
S_2 := mergeSort(S_2, C) \\
S := merge(S_1, S_2) \\
\}
\]

return(S)
Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences \( A \) and \( B \) into a sorted sequence \( S \) containing the union of the elements of \( A \) and \( B \).
- Merging two sorted sequences, each with \( n/2 \) elements and implemented by means of a doubly linked list, takes \( O(n) \) time
  - \( M(n) = O(n) \)

**Algorithm merge**\((A, B)\)**

**Input** sequences \( A \) and \( B \) with \( n/2 \) elements each

**Output** sorted sequence of \( A \) and \( B \)

\( S \leftarrow \) empty sequence

while \( \neg A\.isEmpty() \land \neg B\.isEmpty() \)
  
  if \( A\.first().element() < B\.first().element() \)
    
    \( S\.insertLast(A\.remove(A\.first())) \)
  
  else
    
    \( S\.insertLast(B\.remove(B\.first())) \)

while \( \neg A\.isEmpty() \)
  
  \( S\.insertLast(A\.remove(A\.first())) \)

while \( \neg B\.isEmpty() \)
  
  \( S\.insertLast(B\.remove(B\.first())) \)

return \( S \)
And the complexity of mergesort…

- So, the running time of Merge Sort can be expressed by the recurrence equation:

  \[ T(n) = 2T(n/2) + M(n) = 2T(n/2) + O(n) = O(n\log n) \]

**Algorithm mergeSort(S, C)**

- **Input** sequence \( S \), comparator \( C \)
- **Output** sequence \( S \) sorted according to \( C \)

  ```
  if S.size() > 1 {
    (S₁, S₂) := partition(S, S.size()/2)
    S₁ := mergeSort(S₁, C)
    S₂ := mergeSort(S₂, C)
    S := merge(S₁, S₂)
  }
  return(S)
  ```
Merge Sort Execution Tree (recursive calls)

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - 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  2 | 9  4 → 2  4  7  9

7 | 2 → 2  7  
7 → 7

9 | 4 → 4  9  
9 → 9

2 → 2
4 → 4
```
Execution Example

- Partition

```
| 7 2 9 4 | 3 8 6 1 | 1 2 3 4 6 7 8 9 |
```

```
| 7 2 9 4 | 2 4 7 9 |
```

```
| 7 2 | 2 7 |
```

```
| 9 4 | 4 9 |
```

```
| 3 8 | 3 8 |
```

```
| 6 1 | 1 6 |
```

```
| 7 | 7 |
```

```
| 2 | 2 |
```

```
| 9 | 9 |
```

```
| 4 | 4 |
```

```
| 3 | 3 |
```

```
| 8 | 8 |
```

```
| 6 | 6 |
```

```
| 1 | 1 |
```
Execution Example (cont.)

- Recursive call, partition

[Diagram showing recursive call and partition process]
Execution Example (cont.)

- Recursive call, partition
Execution Example (cont.)

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

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

- Merge
Execution Example (cont.)

- Recursive call, ..., base case, merge
Execution Example (cont.)

- Merge

```
| 7 2 9 4 | 3 8 6 1 | → | 1 2 3 4 6 7 8 9 |

7 2 9 4 | 3 8 6 1 | → | 2 4 7 9 |

7 2 | 9 4 | → | 2 4 7 9 |

7 | 2 | → | 2 7 |

9 4 | → | 4 9 |

3 8 | → | 3 8 |

6 1 | → | 1 6 |
```

```
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1
```
Execution Example (cont.)

- Recursive call, …, merge, merge
Execution Example (cont.)

- Merge

```
7  2  9  4 | 3  8  6  1 → 1  2  3  4  6  7  8  9
3  8  6  1 → 1  3  6  8
```

```
7  | 2  → 2 7
9  | 4  → 4 9
3  | 8  → 3 8
6  | 1  → 1 6
7  → 7  2  → 2  9  → 9  4  → 4  3  → 3  8  → 8  6  → 6  1  → 1
```
Another Analysis of Merge-Sort

- The height $h$ of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The work done at each level is $O(n)$
  - At level $i$, we partition and merge $2^i$ sequences of size $n/2^i$
- Thus, the total running time of merge-sort is $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>#seqs</th>
<th>size</th>
<th>Cost for level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$n/2^i$</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$\log n$</td>
<td>$2^{\log n} = n$</td>
<td>$n/2^{\log n} = 1$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
## Summary of Sorting Algorithms (so far)

<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>Slow, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For small data sets</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC $O(n)$ BC</td>
<td>Slow, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For small data sets</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For large data sets</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For huge data sets</td>
</tr>
</tbody>
</table>
Quick-Sort

7 4 9 6 2 → 2 4 6 7 9

4 2 → 2 4

7 9 → 7 9

2 → 2

9 → 9
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 \( G \)
Analysis of Quick Sort using Recurrence Relations

- **Assumption**: random pivot expected to give equal sized sublists
- The running time of Quick Sort can be expressed as:

\[ T(n) = 2T(n/2) + P(n) \]

- \( T(n) \) - time to run quicksort() on an input of size \( n \)
- \( P(n) \) - time to run partition() on input of size \( n \)

Algorithm `QuickSort(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

**if** \( l \geq r \)

**return**

\( i \leftarrow \) a random integer between \( l \) and \( r \)

\( x \leftarrow S.elemAtRank(i) \)

\( (h, k) \leftarrow Partition(x) \)

`QuickSort(S, l, h - 1)`

`QuickSort(S, k + 1, r)`
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.

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

\[
x \leftarrow S.\text{remove}(p)
\]

\[
\text{while } \neg S.\text{isEmpty}()
\]

\[
y \leftarrow S.\text{remove}(S.\text{first}())
\]

if \( y < x \)

\[
L.\text{insertLast}(y)
\]

else if \( y = x \)

\[
E.\text{insertLast}(y)
\]

else

\[
G.\text{insertLast}(y)
\]

return \( L, E, G \)
So, the expected complexity of Quick Sort

- **Assumption**: random pivot expected to give equal sized sublists
- The running time of Quick Sort can be expressed as:

  \[
  T(n) = 2T(n/2) + P(n) \\
  = 2T(n/2) + O(n) \\
  = O(n\log n)
  \]

**Algorithm QuickSort(S, l, r)**

```plaintext
Input sequence S, ranks l and r
Output sequence S with the elements of rank between l and r rearranged in increasing order
if l \geq r
  return
i \leftarrow a random integer between l and r
x \leftarrow S.elemAtRank(i)
(h, k) \leftarrow Partition(x)
QuickSort(S, l, h - 1)
QuickSort(S, k + 1, r)
```
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

```plaintext
7  4  9  6  2  \rightarrow  2  4  6  7  9
4  2  \rightarrow  2  4
7  9  \rightarrow  7  9
2  \rightarrow  2
9  \rightarrow  9
```
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 $n + (n - 1) + \ldots + 2 + 1 = O(n^2)$
• Alternatively, using recurrence equations, $T(n) = T(n-1) + O(n) = O(n^2)$
Expected Running Time (removing equal split assumption)

- 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:

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
```
Expected Running Time, Cont’d

- **Probabilistic Fact:** The expected number of coin tosses required in order to get $k$ heads is $2^k$ (e.g., it is expected to take 2 tosses to get heads).
- 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 $O(\log n)$
  - 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 $O(n \log n)$

\[ \text{total expected time: } O(n \log n) \]
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

if $l \geq r$
return

$i \leftarrow$ a random integer between $l$ and $r$
$x \leftarrow S\.elemAtRank(i)$
$(h, k) \leftarrow inPlacePartition(x)$

$inPlaceQuickSort(S, l, h - 1)$
$inPlaceQuickSort(S, k + 1, r)$
In-Place Partitioning

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

\[
\begin{array}{ccccccccccccc}
3 & 2 & 5 & 1 & 0 & 7 & 3 & 5 & 9 & 2 & 7 & 9 & 8 & 9 & 7 & 6 & 9
\end{array}
\]
(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 $k$
## Summary of Sorting Algorithms (so far)

<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>Slow, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For small data sets</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC</td>
<td>Slow, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n)$ BC</td>
<td>For small data sets</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For large data sets</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Exp. $O(n \log n)$ AC, BC</td>
<td>Fastest, randomized, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$ WC</td>
<td>For large data sets</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For huge data sets</td>
</tr>
</tbody>
</table>
Selection
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.
  - Also called order statistics, \( i \)-th order statistic is \( i \)-th smallest element
  - Minimum - \( k=1 \) - 1st order statistic
  - Maximum - \( k=n \) - \( n \)-th order statistic
  - Median - \( k=n/2 \)
  - etc
The Selection Problem

• Naïve solution - SORT!
• we can sort the set in $O(n \log n)$ time and then index the $k$-th element.

[Example]

$7 \ 4 \ 9 \ 6 \ 2 \rightarrow 2 \ 4 \ 6 \ 7 \ 9 \ k=3$

• Can we solve the selection problem faster?
The Minimum (or Maximum)

Minimum (A) {
    m = A[1]
    For I=2,n
        M=min(m,A[I])
    Return m
}

• Running Time
  • O(n)
• Is this the best possible?
Quick-Select

- 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 on either $L$ or $G$

- Note: Partition same as Quicksort
Quick-Select Visualization

• 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
```
Exercise

- Best Case - even splits (n/2 and n/2)
- Worst Case - bad splits (1 and n-1)

Derive and solve the recurrence relation corresponding to the best case performance of randomized select.

Derive and solve the recurrence relation corresponding to the worst case performance of randomized select.
Expected Running Time

• 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 $3s/4$

![Diagram]

• A call is good with probability $1/2$
  • $1/2$ of the possible pivots cause good calls:
Expected Running Time, Part 2

- **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,
  - \( T(n) \leq T(3n/4) + bn \) *(expected # of calls before a good call)*
- By Fact #1,
  - \( T(n) \leq T(3n/4) + 2bn \)
- That is, \( T(n) \) is a geometric series:
  - \( T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2n + 2b(3/4)^3n + \ldots \)
- So \( T(n) \) is \( O(n) \).
- We can solve the selection problem in \( O(n) \) expected time.
Deterministic Selection

- We can do selection in $O(n)$ worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
  - Divide $S$ into $n/5$ sets of 5 each
  - Find a median in each set
  - Recursively find the median of the “baby” medians.

- See Exercise C-11.22 for details of analysis.
Bucket-Sort and Radix-Sort
(can we sort in linear time?)
Bucket-Sort

- Let be $S$ be a sequence of $n$ (key, element) items with keys in the range $[0, N - 1]$.
- Bucket-sort uses the keys as indices into an auxiliary array $B$ of sequences (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 $S$.
- Analysis:
  - Phase 1 takes $O(n)$ time.
  - Phase 2 takes $O(n + N)$ time.
Bucket-sort takes $O(n + N)$ time.

**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 $\neg S.isEmpty()$

$f \leftarrow S.first()$

$(k, o) \leftarrow S.remove(f)$

$B[k].insertLast((k, o))$

for $i \leftarrow 0$ to $N - 1$

while $\neg B[i].isEmpty()$

$f \leftarrow B[i].first()$

$(k, o) \leftarrow B[i].remove(f)$

$S.insertLast((k, o))$
Properties and Extensions

**Properties**

- **Key-type**
  - The keys are used as indices into an array and cannot be arbitrary objects
- **Stable Sort**
  - 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 item \((k, o)\) 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 item \((k, o)\) into bucket \(B[r(k)]\)
Example

- Key range [37, 46] – map to buckets [0,9]
Lexicographic Order

• Given a list of 3-tuples:
  \((7,4,6) (5,1,5) (2,4,6) (2,1,4) (5,1,6) (3,2,4)\)

• After sorting, the list is in lexicographical order:
  \((2,1,4) (2,4,6) (3,2,4) (5,1,5) (5,1,6) (7,4,6)\)
Lexicographic Order Formalized

• A $d$-tuple is a sequence of $d$ keys $(k_1, k_2, \ldots, k_d)$, where key $k_i$ is said to be the $i$-th dimension of the tuple
  • Example:
    • The Cartesian coordinates of a point in space is a 3-tuple
  • The lexicographic order of two $d$-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$

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

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.
Exercise: Lexicographic Order

- Given a list of 2-tuples, we can order the tuples lexicographically by applying a stable sorting algorithm two times:
  
  \((3,3) (1,5) (2,5) (1,2) (2,3) (1,7) (3,2) (2,2)\)

- Possible ways of doing it:
  1. Sort first by \(1^{st}\) element of tuple and then by \(2^{nd}\) element of tuple
  2. Sort first by \(2^{nd}\) element of tuple and then by \(1^{st}\) element of tuple

- Show the result of sorting the list using both options
Exercise: Lexicographic Order

(3,3) (1,5) (2,5) (1,2) (2,3) (1,7) (3,2) (2,2)

• Using a stable sort,
  1. Sort first by 1\textsuperscript{st} element of tuple and then by 2\textsuperscript{nd} element of tuple
  2. Sort first by 2\textsuperscript{nd} element of tuple and then by 1\textsuperscript{st} element of tuple

• Option 1:
  • 1\textsuperscript{st} sort: (1,5) (1,2) (1,7) (2,5) (2,3) (2,2) (3,3) (3,2)
  • 2\textsuperscript{nd} sort: (1,2) (2,2) (3,2) (2,3) (3,3) (1,5) (2,5) (1,7) - WRONG

• Option 2:
  • 1\textsuperscript{st} sort: (1,2) (3,2) (2,2) (3,3) (2,3) (1,5) (2,5) (1,7)
  • 2\textsuperscript{nd} sort: (1,2) (1,5) (1,7) (2,2) (2,3) (2,5) (3,2) (3,3) - CORRECT
Lexicographic-Sort

- Let $C_i$ be the comparator that compares two tuples by their $i$-th dimension
- Let $\text{stableSort}(S, C)$ be a stable sorting algorithm that uses comparator $C$
- Lexicographic-sort sorts a sequence of $d$-tuples in lexicographic order by executing $d$ times algorithm $\text{stableSort}$, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of $\text{stableSort}$

Algorithm $\text{lexicographicSort}(S)$

Input sequence $S$ of $d$-tuples
Output sequence $S$ sorted in lexicographic order

for $i \leftarrow d$ downto 1
    $\text{stableSort}(S, C_i)$
Radix-Sort

- 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 $O(d(n + N))$

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

**set** the key $k$ of each item $(k, (x_1, ..., x_d))$ of $S$ to $i$-th dimension $x_i$

$bucketSort(S, N)$
Example: Radix-Sort for Binary Numbers

- Sorting a sequence of 4-bit integers
  - $d=4$, $N=2$ so $O(d(n+N)) = O(4(n+2)) = O(n)$
# 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>Slow, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For small data sets</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$ WC, AC, AC, BC</td>
<td>Slow, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n)$ BC</td>
<td>For small data sets</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For large data sets</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Exp. $O(n \log n)$ AC, BC</td>
<td>Fastest, randomized, in-place</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$ WC</td>
<td>For large data sets</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>Fast, sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For huge data sets</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(d(n+N))$, d #digits, N range of digit values</td>
<td>Fastest, stable only for integers</td>
</tr>
</tbody>
</table>