Bucket-Sort and Radix-Sort

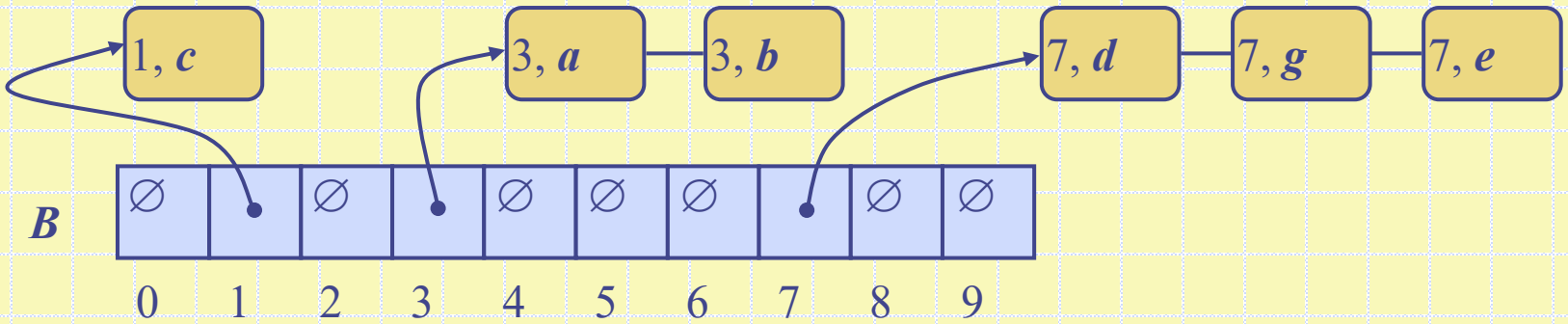
Dr. Aiman Hanna

Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

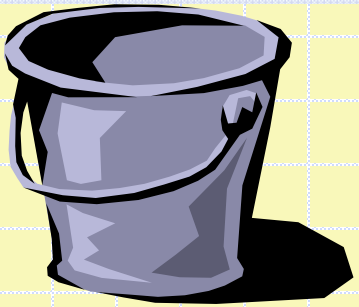
These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Coverage

- Bucket-Sort
- Stable Sorting
- Radix-Sort

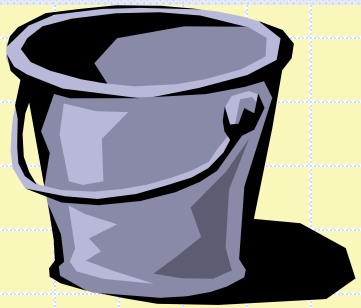


Sorting Lower-Bound



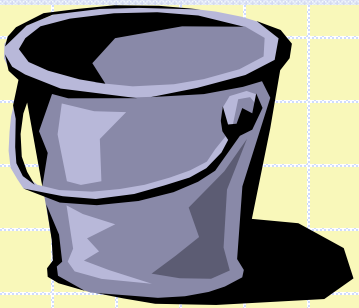
- As previously shown, with comparison-based sorting, $\Omega(n \log n)$ time is necessary to sort a sequence of n elements.
- The following question can then be asked: ***Can any other kinds of sorting algorithms be designed to run asymptotically faster than $O(n \log n)$?***
- Interestingly, such algorithms exist, but they require special assumptions about the input sequence to be sorted.
- Even so, such scenarios often exist in practice, so it worthwhile investigating such algorithms.
- We will next consider the sorting problem of key-value entries, where the keys have a restricted type.

Bucket-Sort



- Let S be a sequence of n entries where the keys are always restricted in the range $[0, N-1]$, for some $N>1$.
- Because of the restrictive assumption about the keys, it is possible to sort without comparisons.
- Bucket-sort does not use comparisons; instead it uses the keys as indices into an auxiliary array B of sequences (buckets).
- The array B has cells that are indexed from 0 to $N-1$.
- An entry with key k is placed in the bucket pointed by $B[k]$.
  - Notice that the bucket pointed by $B[k]$ is hence a sequence of entries that have similar key k.

Bucket-Sort



Phase 1: Empty sequence S by moving each entry (k, v) into its bucket $B[k]$.

Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ back to the end of sequence S.

- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time
  - Hence, total is $O(n + N)$
  - If N is $O(n)$, then we can sort in $O(n)$ time.
- ➔ Bucket-sort can sort in $O(n)$ time; that is clearly lower (better) than $O(n \log n)$.
- You must notice that the range N should not be significantly bigger than n for the algorithm to be efficient.

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 $\neg S.isEmpty()$

$f \leftarrow S.first()$

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

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

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

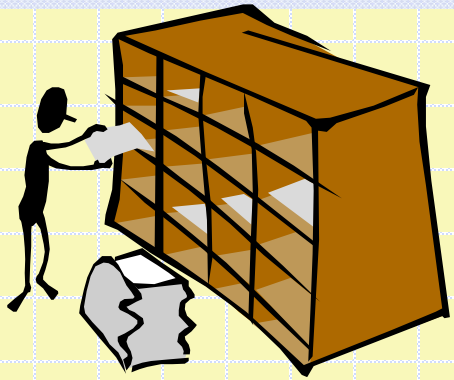
while $\neg B[i].isEmpty()$

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

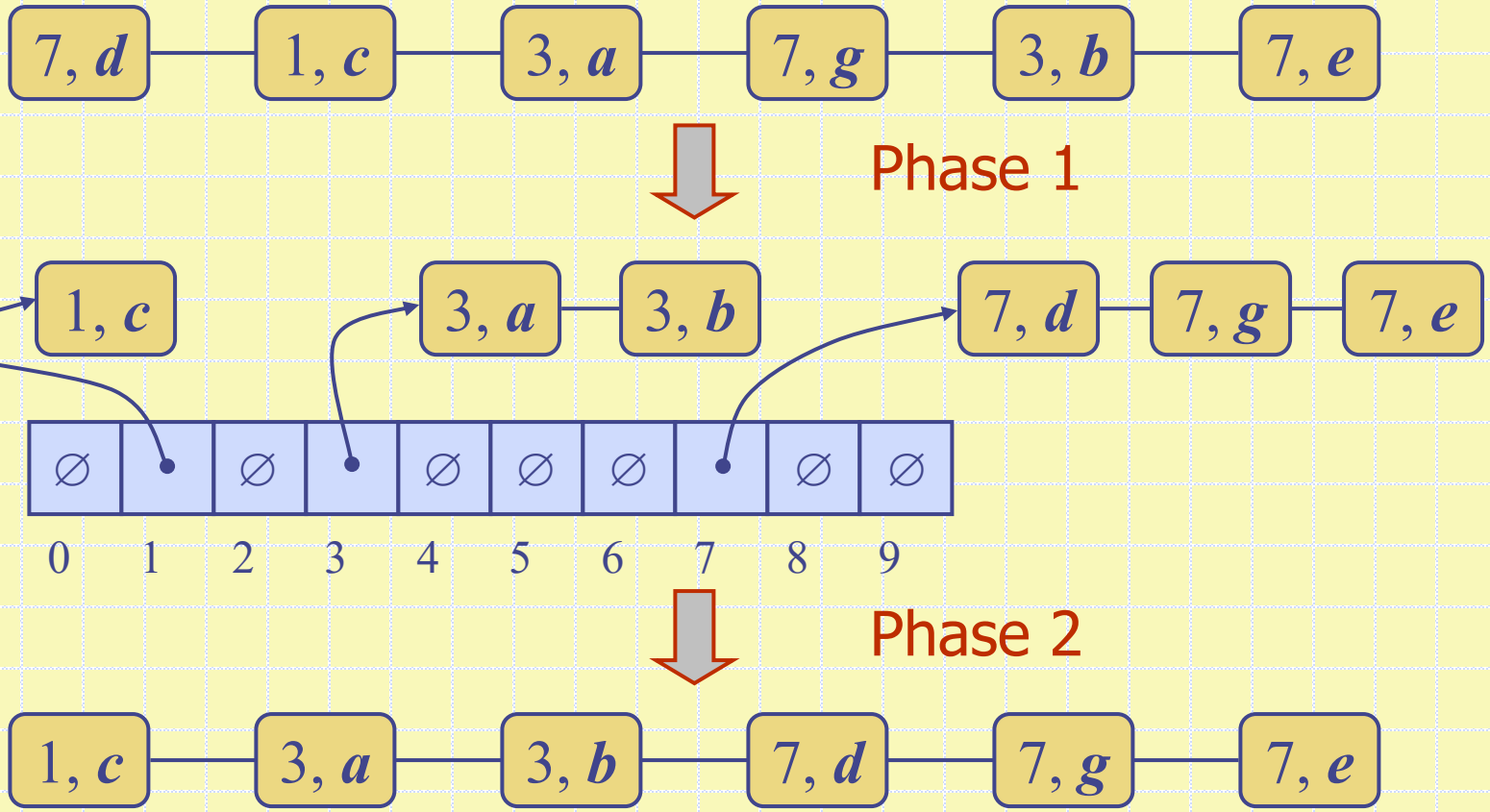
$(k, o) \leftarrow B[i].remove(f)$

$S.addLast((k, o))$

Example



- Key range $[0, 9]$



Properties and Extensions

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

Possible Extensions:

- Integer keys can be in the range $[a - b]$, which is equal in size to $[0 - N-1]$, but not restricted to $[0 - N-1]$
  - Simply put entry (k, v) into bucket $B[k - a]$
  - Example: Keys are 30 to 39. Key 37 is placed in B[37 – 30]; that is B[7].
- String keys can also be possible. For instance if the length of the string is the mean of sorting, and the entries have keys with lengths that are always between a minimum and maximum lengths (e.g., names of the 50 U.S. states), then strings can be used as the keys.

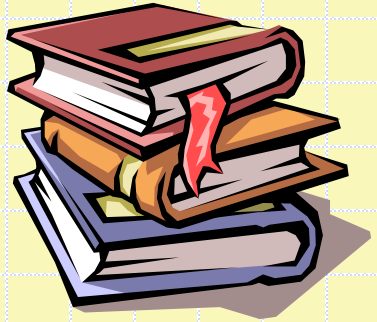
Stable Sorting

- *Stable* Sort Property
  - A sorting algorithm is classified as "stable" if it preserves the original order (in the unsorted sequence) of entries with the same keys when it forms the sorted sequence.
  - In other words, the relative order of any two items with the same key is preserved after the execution of the algorithm.
  - For instance, if three entries (*k2, v3*), (*k2, v9*), and (*k2, v5*) exist in that order in the unsorted sequences, then the final sorted sequence must also have them in that same order (*k2, v3*), (*k2, v9*), then (*k2, v5*).

Stable Sorting

- Our informal description of bucket-sort does not guarantee stability
  - We neither restricted how the entries are removed from the unsorted sequence, nor how they we removed from the bucket entry when forming the sorted sequence.
- Bucket-sort can be made stable however just by:
  - Enforcing the order of removal of the items from the unsorted sequence from first to last,
  - Insert these items at the tail of sequences in the buckets
  - Enforce the removal from the bucket array when forming the sorted sequence to be from first to last (head to tail).
- Generally, stable sorting is important for many applications.

Lexicographic Order



- 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 are a 3-tuple.
- The lexicographic order of two d-tuples is recursively defined as follows

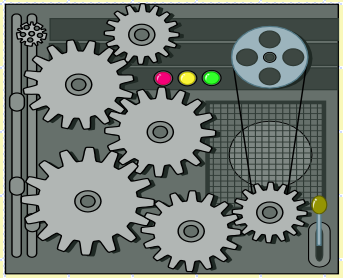
$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$

$$\Leftrightarrow$$

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

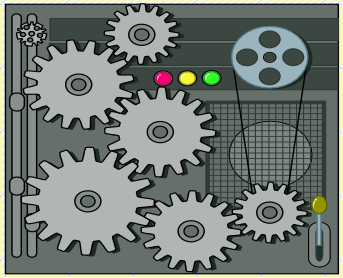
- I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

Radix-Sort



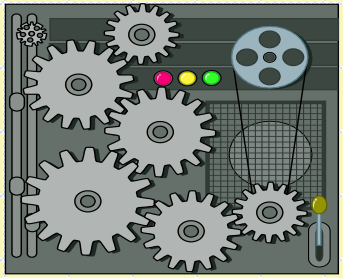
- One of the reasons that stable sorting is so important, is that it allows bucket-sort approach to be applied to more general contexts that sorting integers.
- For instance, suppose that entries have key pairs (x, y), and not just one key. That is, an entry looks as follows: $((x, y), v)$ where the sorting is to be made based on these pairs.
- In such cases, it is natural to define an ordering of these keys using lexicographical convention.
- That is pair $(x_1, y_1) < (x_2, y_2)$ if $x_1 < x_2$ or $x_1 = x_1$ and $y_1 < y_2$.

Radix-Sort



- **Radix-sort** algorithm sorts a sequence S of entries with keys that are pairs (or generally d-tuple).
- Radix-sort performs that simply by applying a stable bucket-sort on the sequence twice (when the keys are pairs. The application is done d times in the general d-tuple case).
- That is, sort first using one of the keys in the pair, then sort again using the other key.
- The question however is that: Given entries with key pairs (x_1, y_1) & (x_2, y_2), should we first sort based on the first component of the pair (these are the x_i keys) then the second component (these are the y_i keys) or vise versa?

Radix-Sort



- **Example:**
  - Let us consider the following sequence S (only the keys are shown) where sorting is needed.

$$S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))$$

- Sorting based on first component then second component will result in the following:
  - After first component sorting, we have:

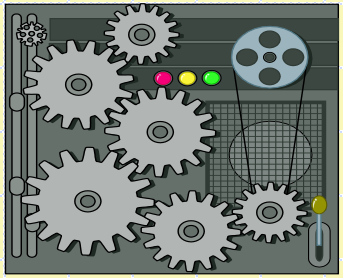
$$S = ((1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2))$$

  - Finally, after second component sorting, we end up with:

$$S = ((1, 2), (2, 2), (3, 2), (2, 3), (3, 3), (1, 5), (2, 5), (1, 7))$$

which is clearly NOT lexicographically sorted!

Radix-Sort



Example (continues ...):

- Now let us sorting based on second component then first component will result in the following:
- Initial sequence

$$S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))$$

  - After first component sorting, we have:

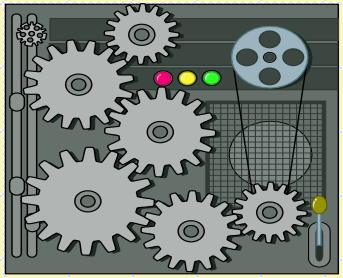
$$S = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7))$$

  - Finally, after second component sorting, we end up with:

$$S = ((1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3))$$

which is indeed lexicographically sorted.

Radix-Sort



- The previous example led us to believe that the sorting should start by the second component then finish by the first one.
- Is that generally the case? Will this always result in the correct behavior?
- *Yes*. By first stably sorting by the second component and then by the first component, we guarantee that if two entries have an equal first component key (when applying the second phase of sorting), then their relative order from the first phase is preserved.
- Thus, the resulting sequence is guaranteed to be lexicographically sorted every time.

Lexicographic-Sort Analysis

- Let C_i be the comparator that compares two tuples by their i-th dimension.
- Let *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 *stableSort*, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*.

Algorithm *lexicographicSort*(S)
Input sequence S of d-tuples
Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1
stableSort(S, C_i)

Example:

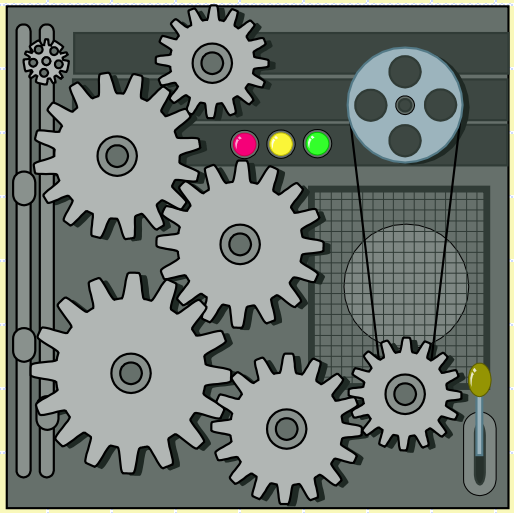
(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

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

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

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

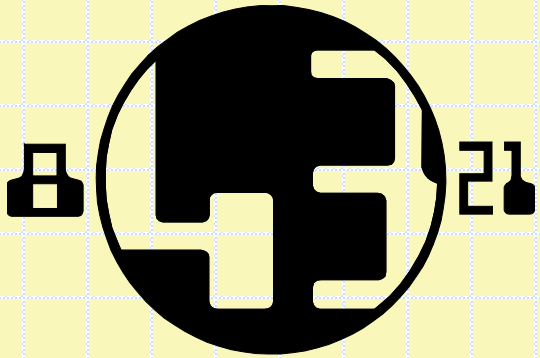
Radix-Sort Analysis



- Again, in the general case, Radix-sort is applicable to d-tuples keys, where the keys in each dimension i are integers in the range $[0, N-1]$ for $N > 1$.
- Radix-sort uses sable bucket-sort for each of the d components.
- Consequently, Radix-sort runs in time $O(d(n+N))$.

Algorithm *radixSort*(S, N)
  Input sequence S of d-tuples such that $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \leq (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
  Output sequence S sorted in lexicographic order
  for $i \leftarrow d$ **downto** 1
    bucketSort(S, N)

Radix-Sort for Binary Numbers



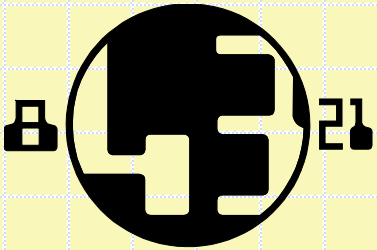
- Consider a sequence of n b-bit integers

$$x = x_{b-1} \ldots x_1 x_0$$

- 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 *binaryRadixSort*(S)
  Input sequence S of b-bit integers
  Output sequence S sorted
  replace each element x of S with the item $(0, x)$
  for $i \leftarrow 0$ **to** $b - 1$
    replace the key k of each item (k, x) of S with bit x_i of x
    bucketSort(S, 2)

Example



- Sorting a sequence of 4-bit integers