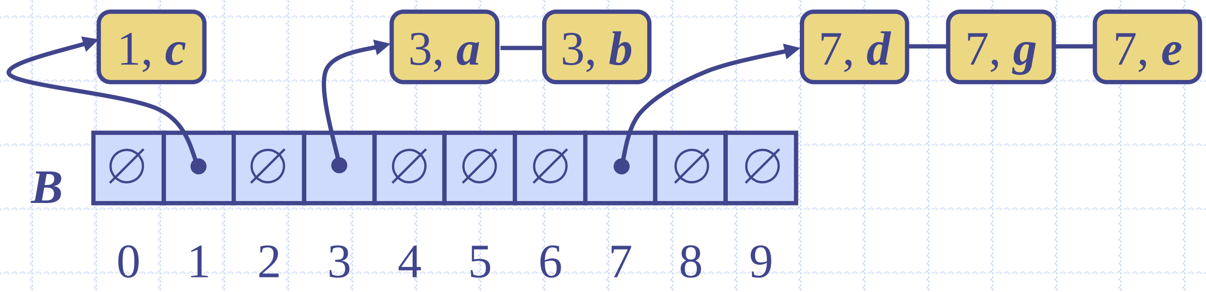
Bucket-Sort and Radix-Sort



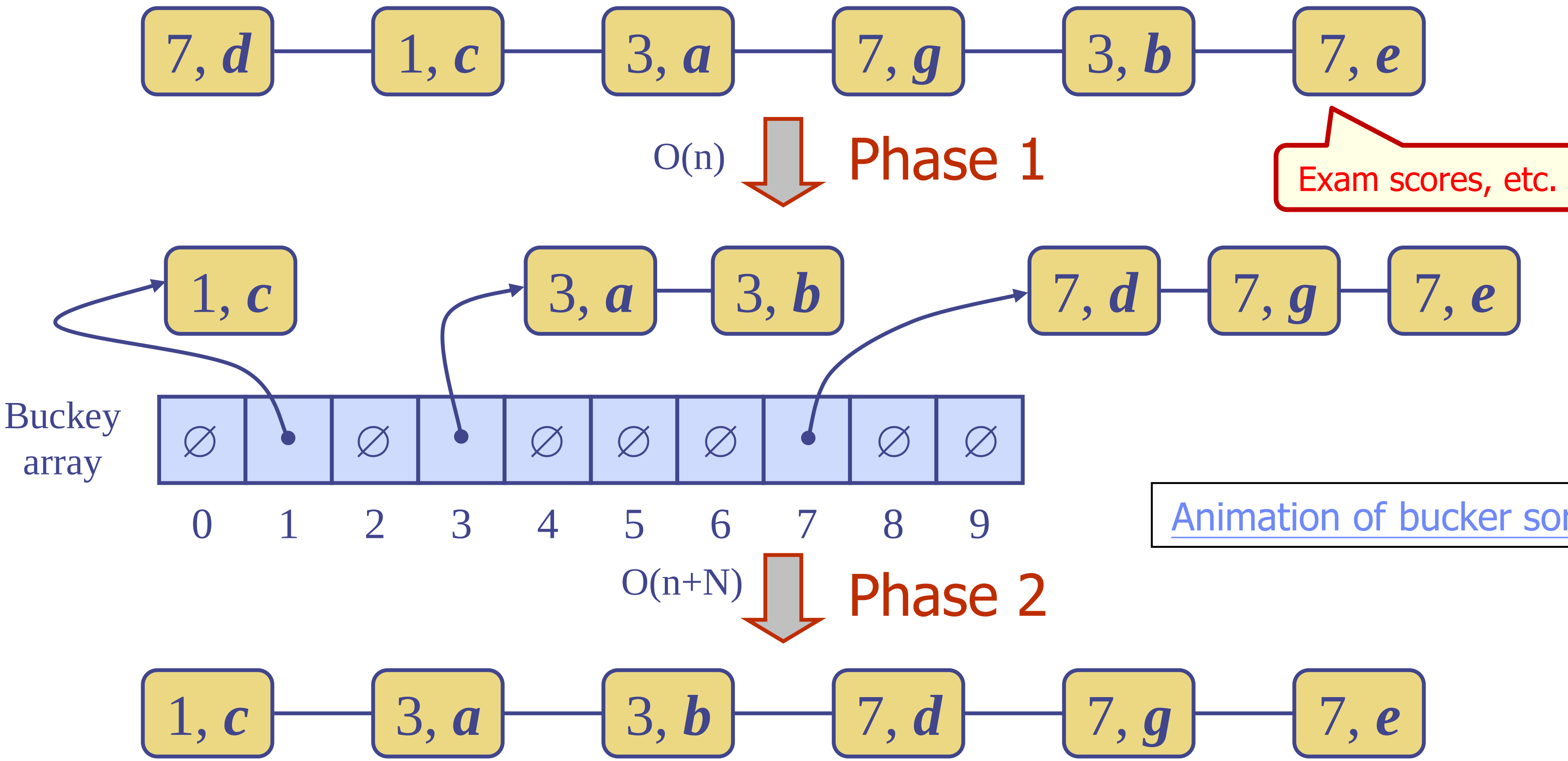
Introduction

- Comparison-based sorting algorithms → $\Omega(n \log n)$
  - Quick-sort, heap-sort, merge-sort
- Can we do better?
  - Yes, if the keys have a restricted type
  - Examples: Bucket-sort and radix-sort
  - Complexity: Linear!

No comparison needed!

Bucket-sort Example

- Key range [0, 9]



[Animation of bucker sort]

Bucket-Sort

- Let be S be a sequence of n (key, element) entries 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 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

Linear time if N << n

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 $\neg S.empty()$
    $(k, o) \leftarrow S.front()$
    $S.eraseFront()$
    $B[k].insertBack((k, o))$
  for $i \leftarrow 0$ **to** $N-1$
    while $\neg B[i].empty()$
      $(k, o) \leftarrow B[i].front()$
      $B[i].eraseFront()$
      $S.insertBack((k, o))$

Counting-sort

- Characteristics of counting-sort
  - A special case of bucket-sort
  - Avoid the use of linked list by counting the number of occurrence of each key

Counting-sort Example

Quiz!

- Input array:

6	0	1	3	2	3	4	2	2

Keys in 0~6

- Count array:

0	1	2	3	4	5	6
1	1	3	2	1	0	1

Use keys as indices

- Position array:

0	1	2	3	4	5	6
0	1	2	5	7	8	8

Cumulative sum of the count array

Starting positions

- Output array:

0	1	2	3	4	5	6	7	8
0	1	2	2	2	3	3	4	6

Animation of count-sort

Properties and Extensions

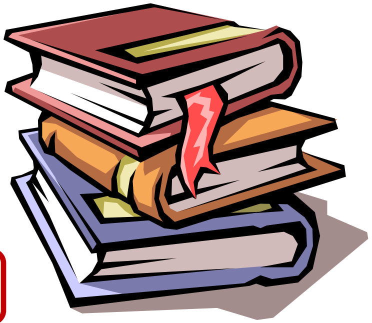
- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

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 $B[r(k)]$

You need to convert keys into buckey array indices.

Lexicographic Order



Dictionary 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.

Lexicographic-Sort

- 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
    $stableSort(S, C_i)$

Another example: Sort dates

Example:

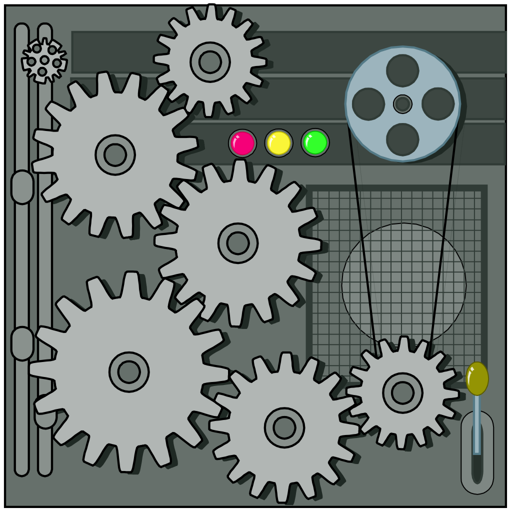
(7, 4, 6) (5, 1, 5) (2, 4, 6) (2, 1, 4) (3, 2, 4)

(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



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

Animation of radix sort

Algorithm ***radixSort***(S, N)

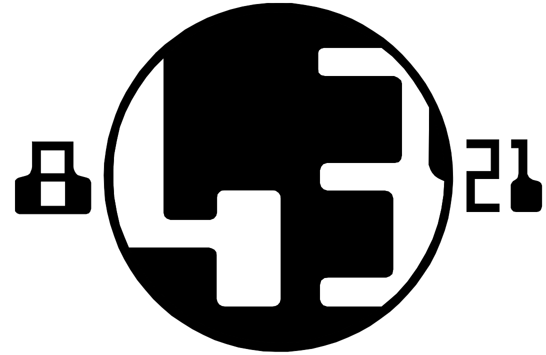
  Input sequence S of d-tuples such that $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S

  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
  $$x = x_{b-1} \dots x_1 x_0$$
- 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 $(0, x)$

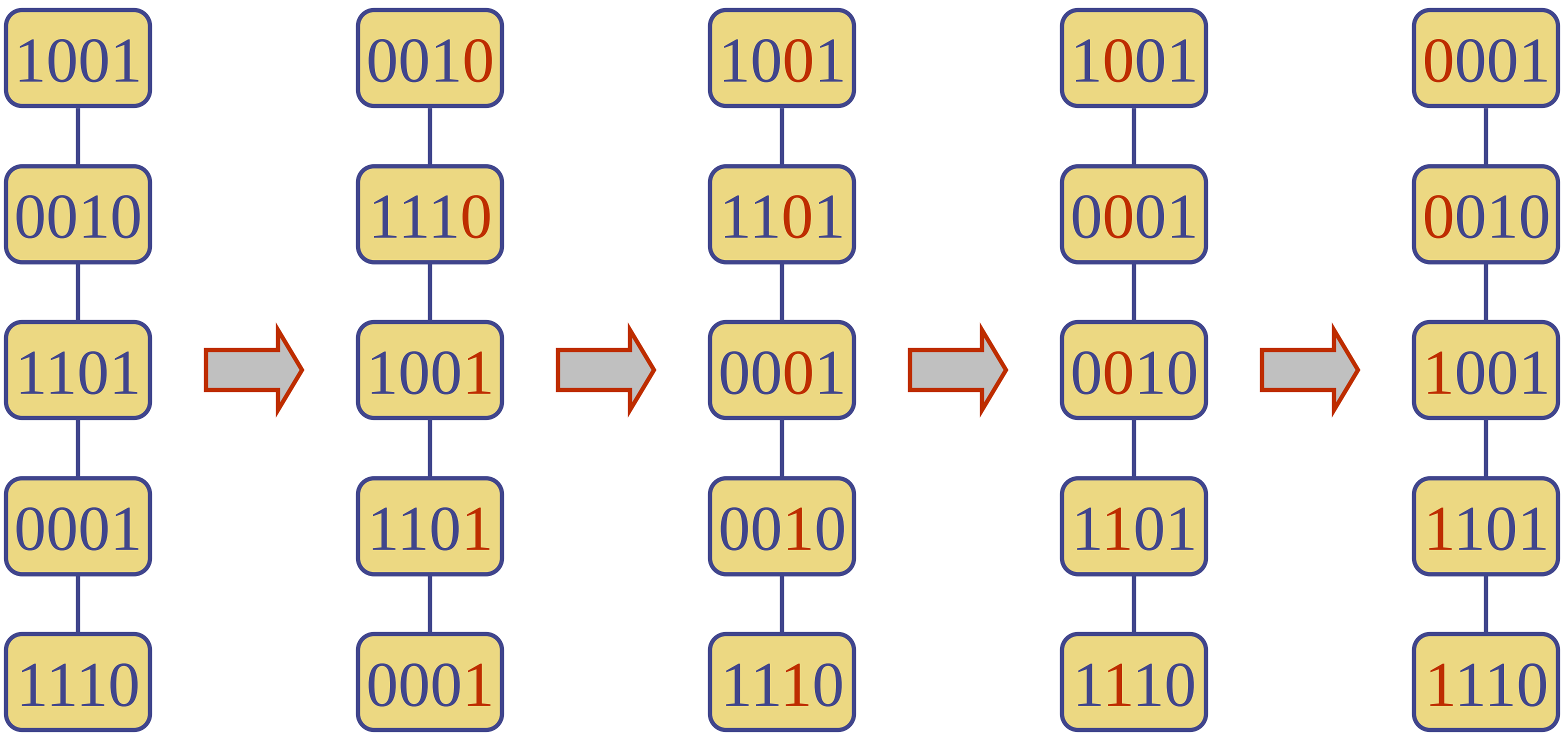
for $i \leftarrow 0$ **to** $b - 1$

replace the key k of each item (k, x) of S with bit x_i of x

bucketSort$(S, 2)$

Radix-sort Example

- Sorting a sequence of 4-bit integers



Another Radix-sort Example

input	digit 1	digit 2	digit 3	digit 4
1233	7650	4721	1145	0774
1145	4721	5522	6161	1145
7650	6161	6732	1233	1233
4721	6732	1233	7235	4265
6732	5522	7235	4265	4536
4265	1233	5336	5336	4721
7235	0774	4536	5522	5336
6161	1145	1145	4536	5522
0774	4265	7650	7650	6161
5336	7235	6161	6732	6732
4536	5336	4265	4721	7235
5522	4536	0774	0774	7650

Quiz!

- Quicksort: O(n log n)
- Radix sort: O(d*(n+10))