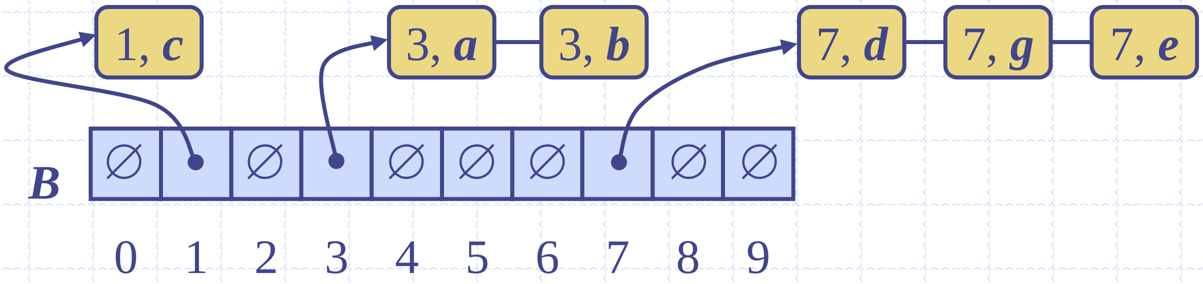
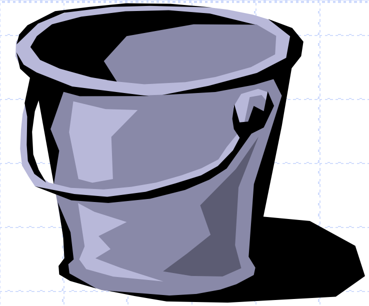
Bucket-Sort and Radix-Sort



Bucket-Sort

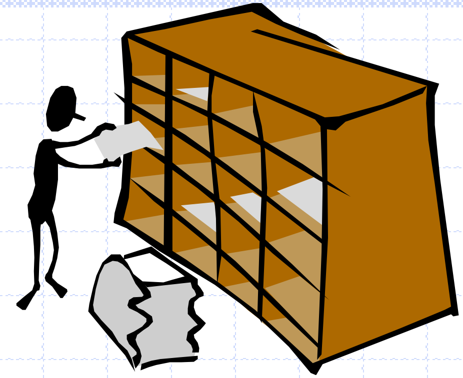


- Let S be a sequence of n (key, element) items with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each item (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the items of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n+N)$ time

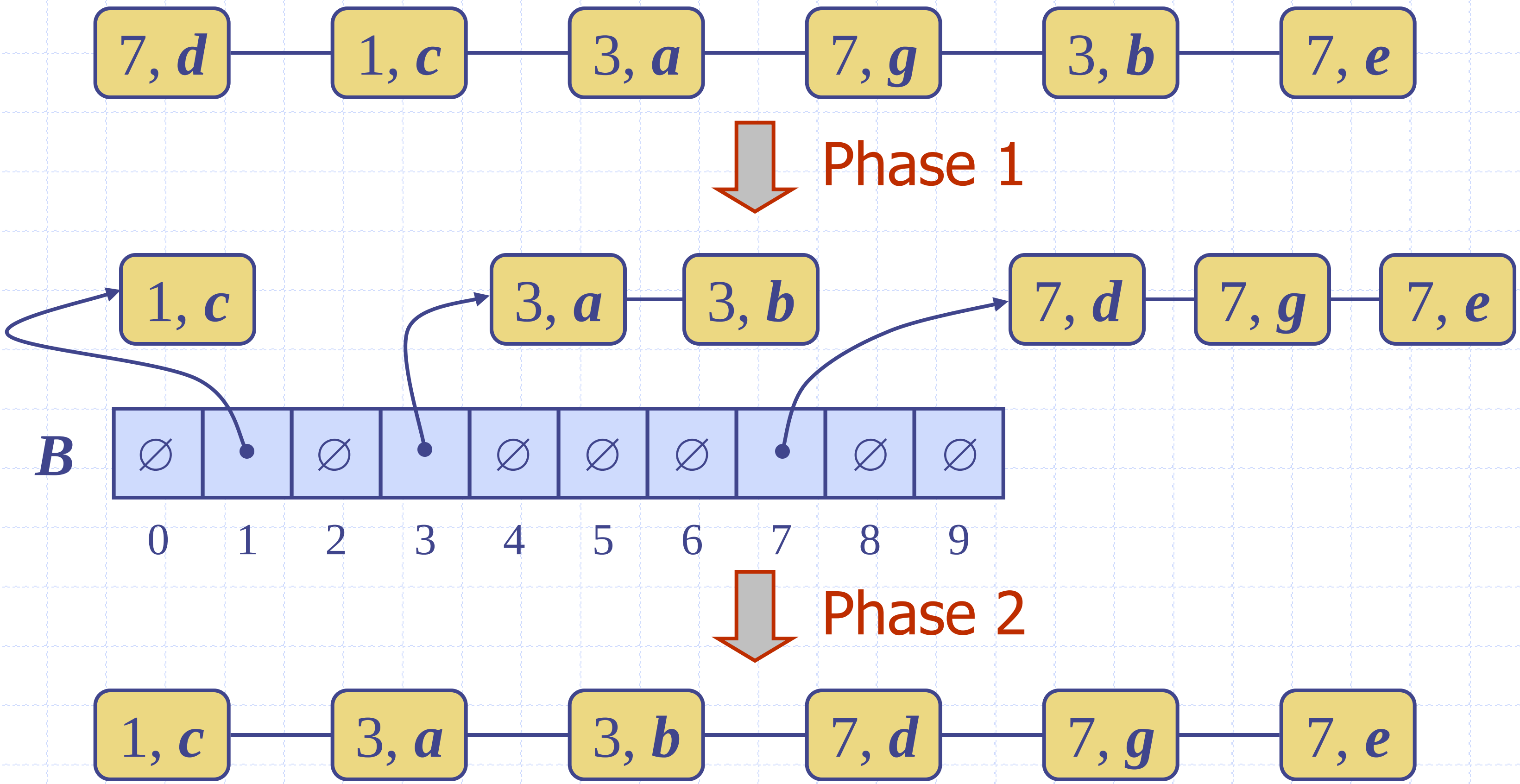
  Bucket-sort takes $O(n + N)$ time

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 $\neg B[i].isEmpty()$
            $f \leftarrow B[i].first()$
            $(k, o) \leftarrow B[i].remove(f)$
            $S.insertLast((k, o))$

Example



- Key range [0, 9]



Properties and Extensions

- Key-type Property
  - The keys are used as integer 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 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 $B[r(k)]$
- Any key-set that can essentially be mapped to a contiguous set of integers

Example Application

- You are hired by a new startup company
- Each person a database (of size 1 million) is uniquely keyed by name or SSN
- Database is continually re-ordered based on a proprietary algorithm
- They ask you to write a program which enables blazingly fast re-sorting of the database by SSN
  - List of people changes slightly each time, but SSN are fixed
  - The order of the people changes drastically (i.e., is not "almost sorted")

Options?

1. Use an O(N^2) algorithm
2. Use an O($N \log N$) algorithm
3. Invent an O(N) algorithm!

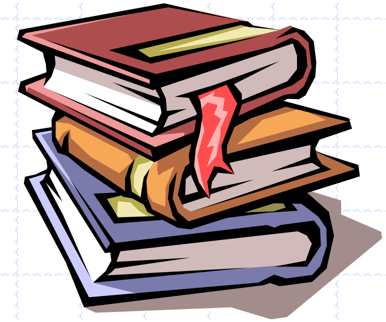
If you accomplish option (3) you get a huge bonus.

What can you do?

Bucket-sort:

a. Map SSN to a compact list of sequential IDs
b. Use compact IDs to bucket sort!

Next topic: Lexicographic Order



- A d-tuple is a sequence of d keys $(k_1, k_2, \dots, k_d)$, 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

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

$$\Leftrightarrow$$

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

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)

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)

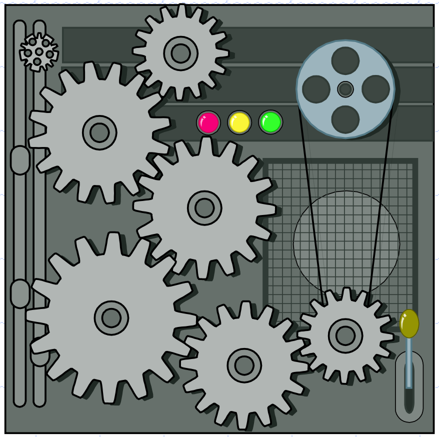
(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

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

Why do you sort starting with the last dimension?

Like numbers/bits: start with LSB to MSB…

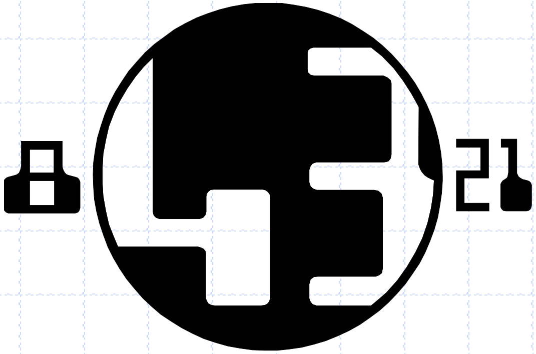
Welcome to Radix-Sort



- 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 $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ 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

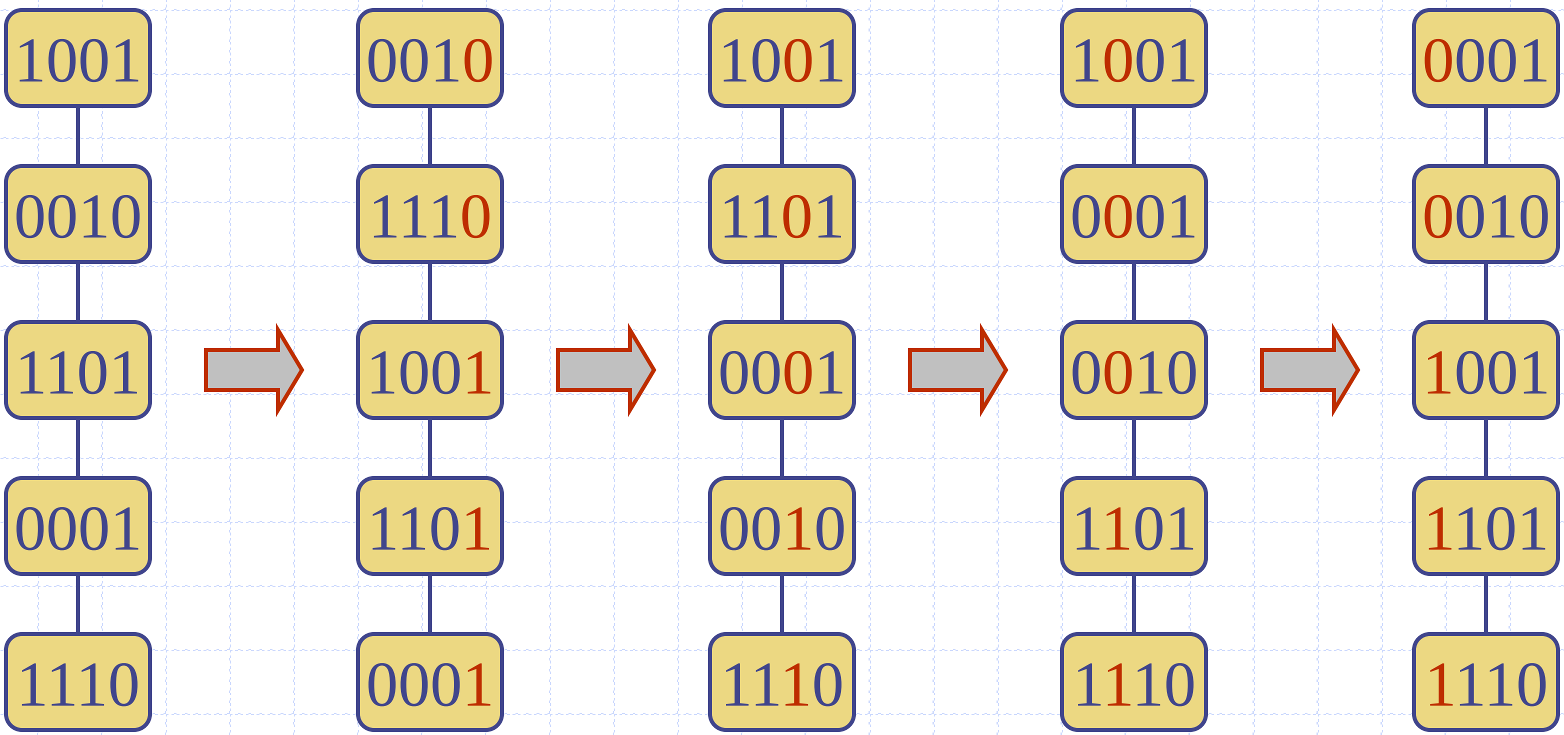


- Consider a sequence of n b-bit integers
  $$x = x_{b-1} \dots x_1 x_0$$
- What is the tuple size?
  - b
- What is the key range?
  - N=2 (i.e., 0 or 1)
- This application of the radix-sort algorithm runs in what time?
  - $O(b(n+2)) = O(n)$ *for b constant*
- For example, we can sort a sequence of 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



Summary of Sorting Algorithms

Algorithm	Time	Notes
bubble-sort	$O(n^2)$	◈ quick and dirty
selection-sort	$O(n^2)$	◈ in-place ◈ slow (good for small inputs)
insertion-sort	$O(n^2)$	◈ in-place, $O(n)$ for almost sorted ◈ slow (good for small inputs)
shell-sort	$O(n^2)$	◈ enhanced insertion sort ◈ can be near $O(n\log^2 n)$
bucket-sort	$O(n + N)$	◈ only for contiguous integer keys [0, N-1] ◈ uses space proportional to key range
radix-sort	$O(d(n + N))$	◈ good for sorting numbers ◈ $O(n)$ for N=2 (binary) and d constant

Summary of Sorting Algorithms

Algorithm	Time	Notes
heap-sort	$O(n \log n)$	◈ in-place, O(n) first results ◈ fast (good for large inputs)
quick-sort	$O(n \log n)$ expected	◈ in-place, randomized ◈ fastest (good for large inputs)
merge-sort	$O(n \log n)$	◈ sequential data access ◈ fast (good for huge inputs)
bucket-sort	$O(n + N)$	◈ only for contiguous integer keys [0, N-1] ◈ uses space proportional to key range
radix-sort	$O(d(n + N))$	◈ good for sorting numbers ◈ O(n) for N=2 (binary) and d constant

Geometry and Sorting

- Sorting is not only useful to organize keys
- Geometric operations and sorting are also tightly linked
- Breaking the $O(n \log n)$ boundary would also benefit geometric operations
  - Or, said another way, there are also geometric arguments to the $O(n \log n)$ boundary

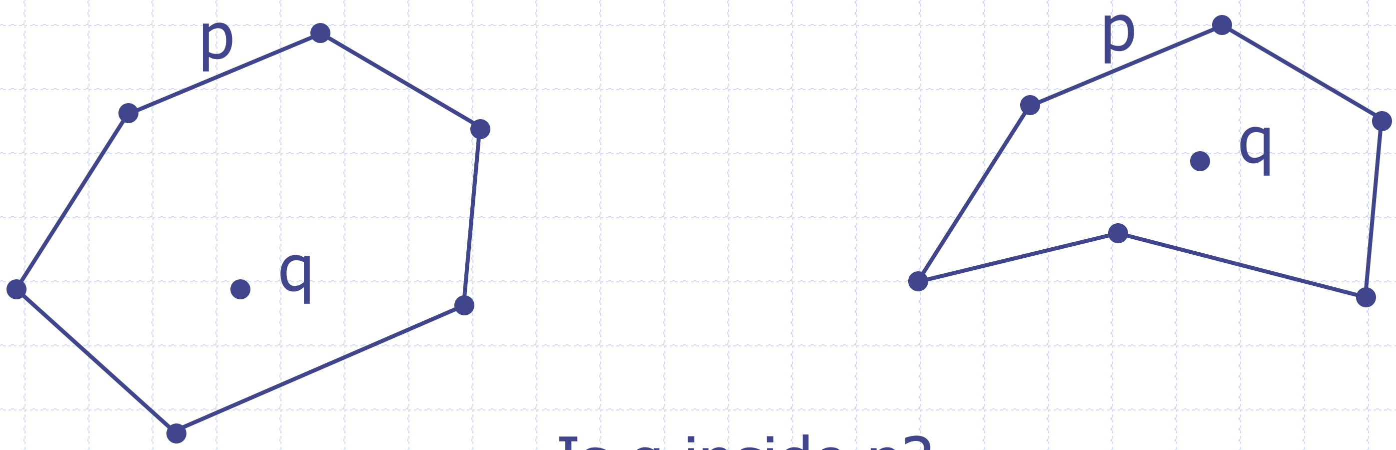
Point-in-Polygon

- Given a set of n-points that form a closed polygon, determine if a point q is "inside" the polygon p

Is q inside p?

Point-in-Polygon

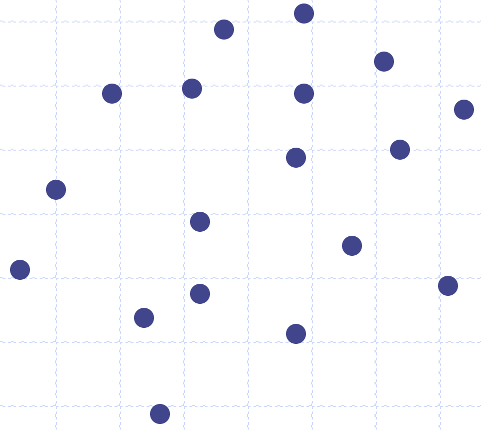
- Straightforward O(n) inclusion algorithm
- For convex polygon case, we can be more efficient…



Is q inside p?

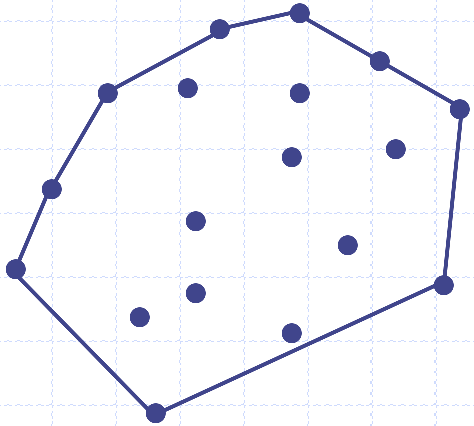
Convex Hull

- Given an array of points P, find the convex hull, e.g., the “shrink wrap”



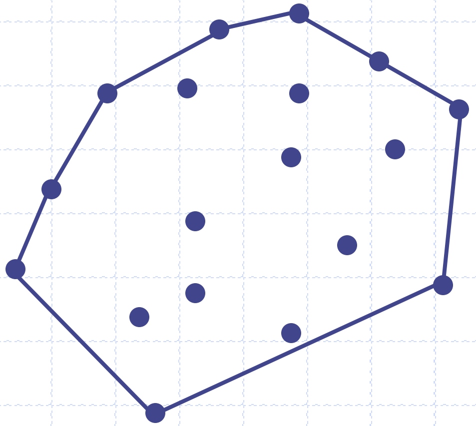
Convex Hull

- Useful for many applications: point inclusion, triangulation, surface reconstruction (in 3D), etc.



Convex Hull

- How do you this? Ideas?
- What is the complexity?



Convex Hull

- Best known solution is O(n log n)
- Corresponds to a two-dimensional sort (in 2D)
  - Thus if could do better than O(n log n), could sort better than O(n log n)