L04.03
Linear Time sorting

50.004 Introduction to Algorithm
Gemma Roig
(slides adapted from Dr. Simon LUI)
ISTD, SUTD

Objective

- The Ω(nlogn) comparison model sorting algorithm (what we have taught so far)
- The O(n) sorting algorithms (for small integers)
  - counting sort
  - radix sort

Comparison sort

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements

- *e.g.*, merge sort, insertion sort

The best running time that we've seen for comparison sorting is $O(n \log n)$

Is $O(n \log n)$ the best we can do?

Decision trees can help us answer this question

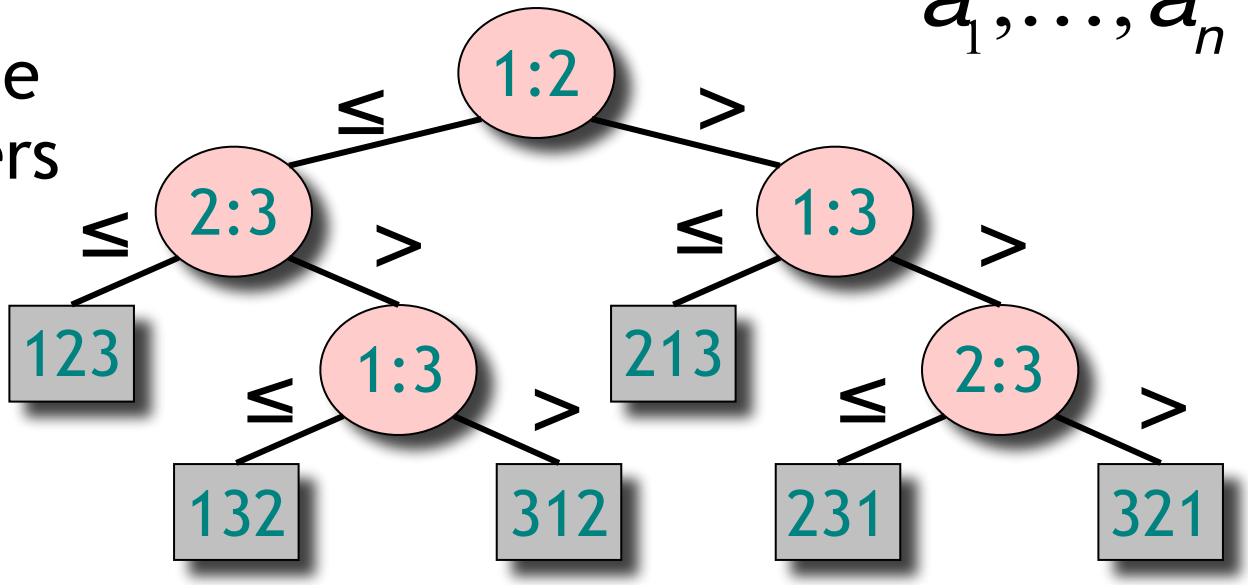
Decision-tree

A given + complete recipe for sorting *n* numbers

- This is an example of sorting 3 numbers

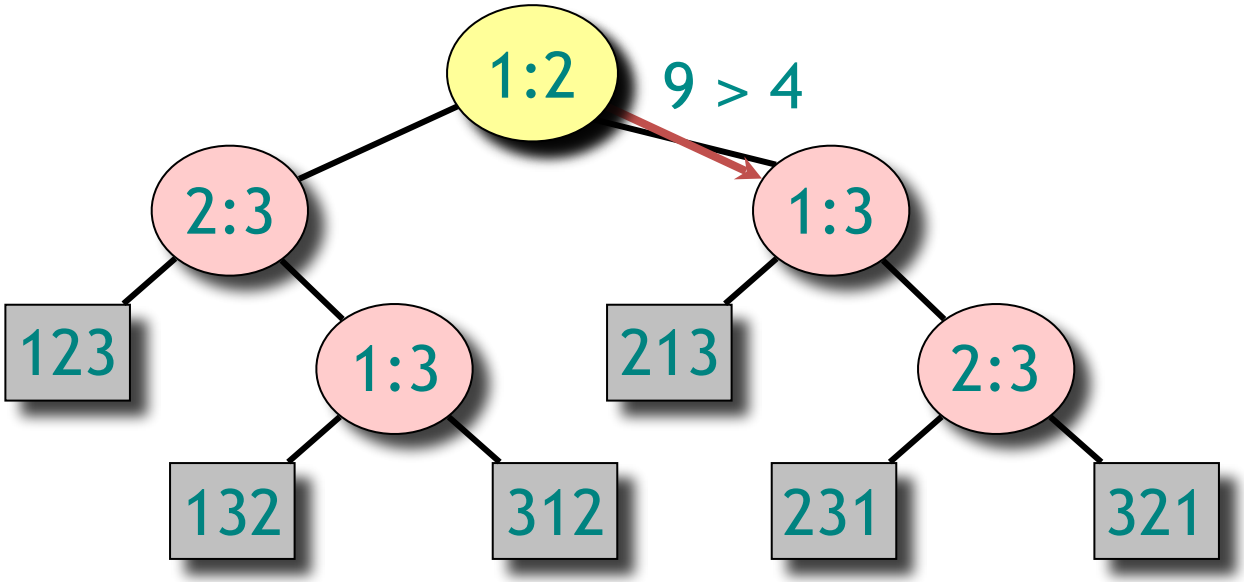
- *i*:*j* means compare *i*th and *j*th number

- Leaves are the results



Decision-tree example

Sort $a_1=9$, $a_2=4$, $a_3=6$

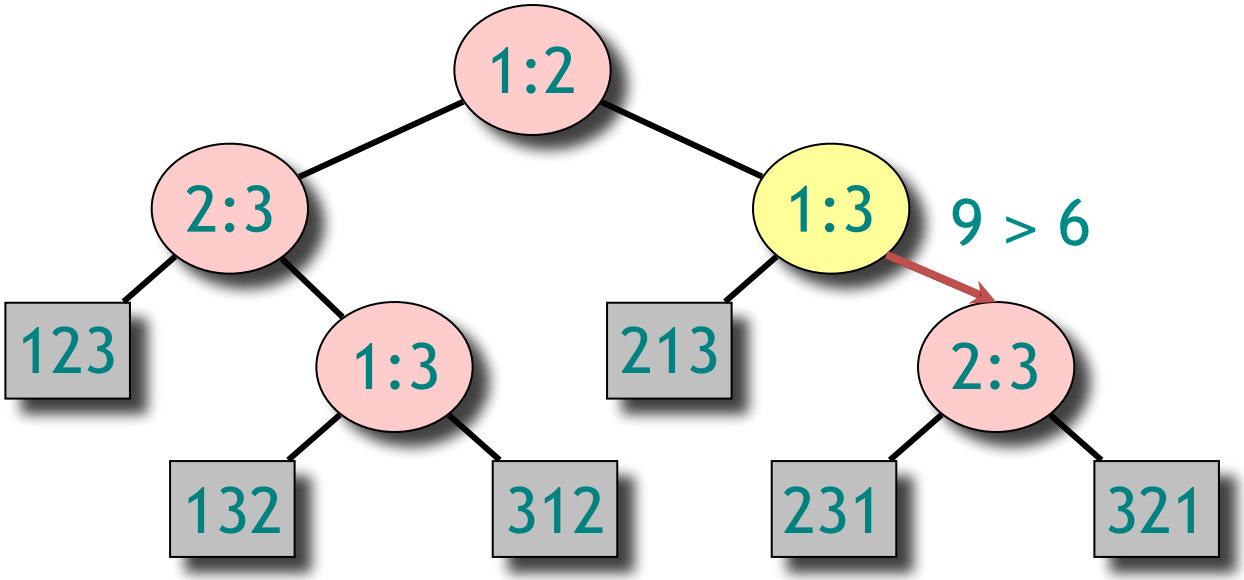


Each internal node is labeled $i:j$ for $i, j \in \{1,\ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$

Decision-tree example

Sort $a_1=9$, $a_2=4$, $a_3=6$

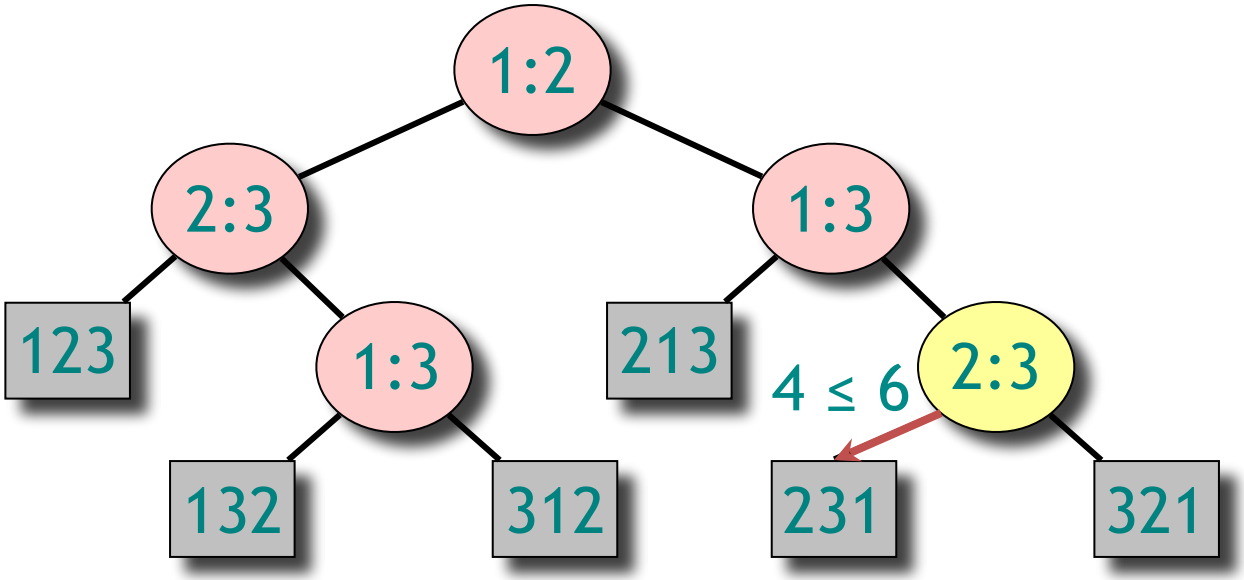


Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if a_i

Decision-tree example

Sort $a_1=9$, $a_2=4$, $a_3=6$

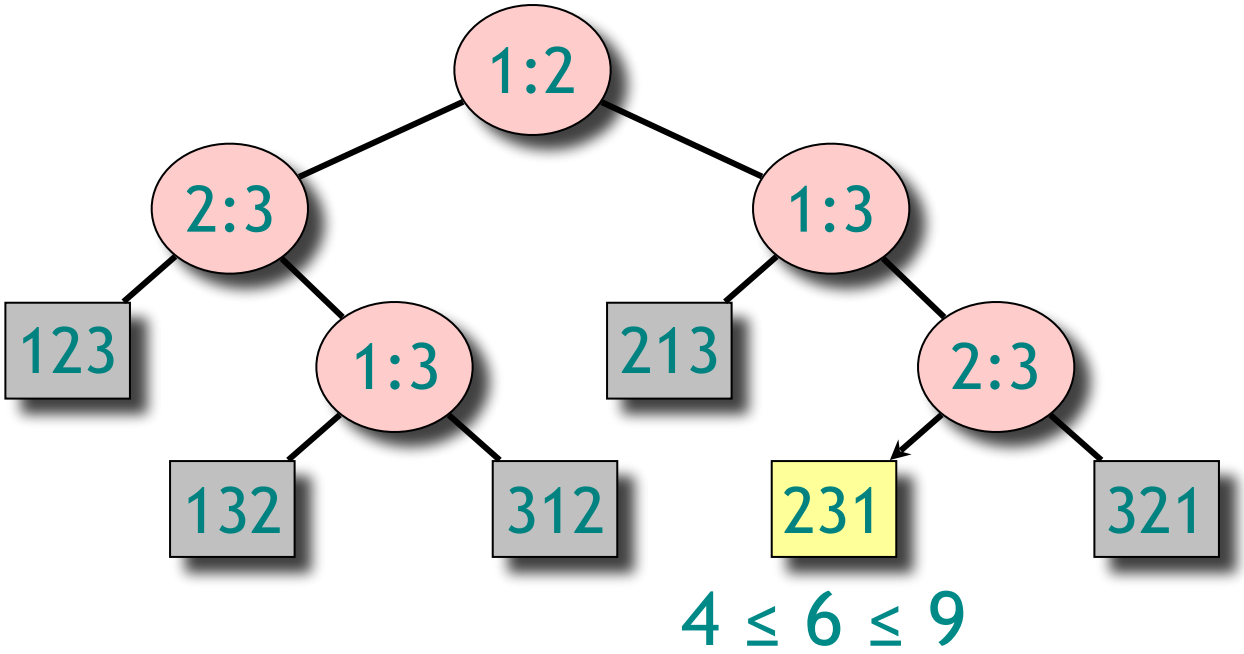


Each internal node is labeled i:j for $i, j \in \{1, \ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$

Decision-tree example

Sort $a_1=9$, $a_2=4$, $a_3=6$



Each leaf contains a permutation $\pi(1), \pi(2),\ldots, \pi(n)$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le$? $\le a_{\pi(n)}$ has been established

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n
- A path from the root to the leaves of the tree represents a trace of comparisons that the algorithm may perform
- The running time of the algorithm = the length of the path taken
- Worst-case running time of the given algorithm = height of tree

Lower bound for decision-tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$

Proof. (Hint: how many leaves are there?)

- The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations = $n!$ possible answers
- A height-h binary tree has $\leq 2^h$ leaves
- Thus

L_n = # of leaves of a decision tree for problem size n

h_n = height of a decision tree for problem size n

$$n! \leq L_n \leq 2^{h_n}$$

$$\Leftrightarrow \log(n!) \leq h_n \Leftrightarrow h_n = \Omega(n \log n)$$

Why it is Ω(nlogn)

- Why Ω(nlogn):

log(n!)
>= n/2 log(n/2) + [(n-1)/2] log2
= n/2 log(n/2) + (n/2) log 2 – ½ log 2
= n/2 log(n) – ½

Also: with Stirling's approx.

n! ~ sqrt(2pi n) (n/e)^n -> log(n!) = Θ (nlogn)

Conclusion: normally, sorting complexity is at least $\Omega(n \log n)$

But there is exception: when k is small.

Linear time sorting

- We sort n keys
- If keys are ("small") integers in $\{0,1,\ldots,k\text{-}1\}$, is it possible to sort them without comparing them?
  - then lower bound of $\Omega(n\log n)$ does not apply
  - will prove that if $k \le n^c$ then we can sort in $O(n)$ time

- Two algorithms: counting sort, radix sort

Counting sort

A[i]: satellite data | key

- Given: an array A[0],…,A[n-1] of *n* keys to be sorted
- The *n* keys are integers in {0,1,…,*k*-1}

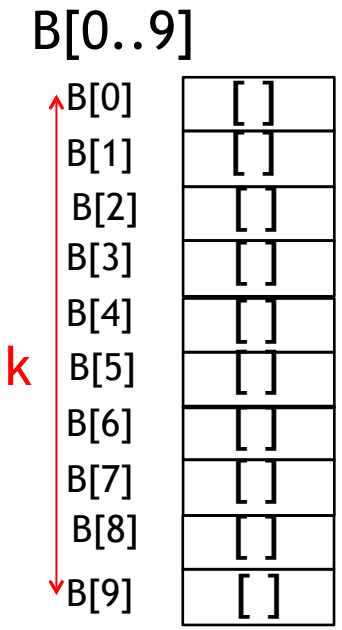
- *B* = array of *k* empty lists (linked, or Python lists)

 - for *j* in range(*n*):
 B[A[*j*]].append(A[*j*])
 - output = []
 - for *i* in range(*k*):
 output.extend(*B*[*i*])

n=5

2	5	9	8	2

keys in 0..9

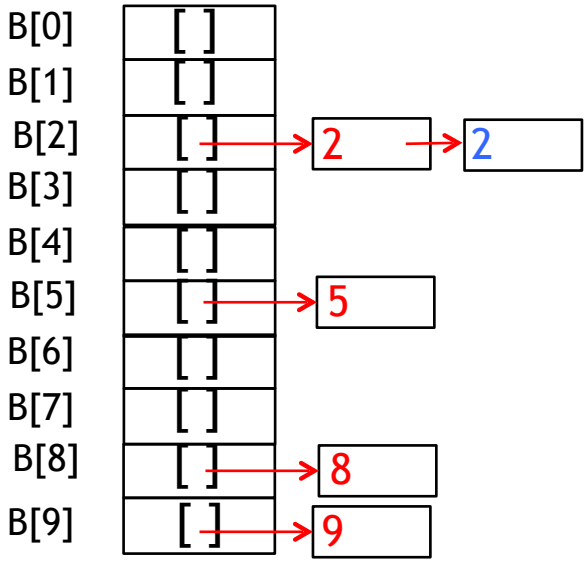


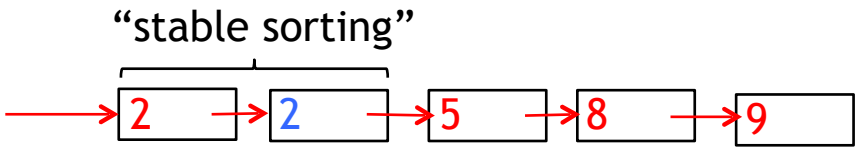
Counting sort

- Given: an array A[0],…,A[n-1] of *n* keys to be sorted
- The *n* keys are integers in {0,1,…,*k*-1}

- *B* = array of *k* empty lists (linked, or Python lists)
  - for *j* in range(*n*):
    B[A[*j*]].append(A[*j*])
  - output = []
  - for *i* in range(*k*):
    output.extend(*B*[*i*])

keys in 0..9





Counting sort

- Given: an array A[0],…,A[n-1] of n keys to be sorted
- The n keys are integers in {0,1,…,k-1}

- B = array of k empty lists (linked, or Python lists) } $O(k)$
- for j in range(n):
  B[A[j]].append(A[j]) } $O(1)$ } $O(n)$
- output = []
- for i in range(k):
  output.extend(B[i]) } $O(\sum_i (1+|B[i]|)) = O(k+n)$

Total time: $\Theta(n+k)$

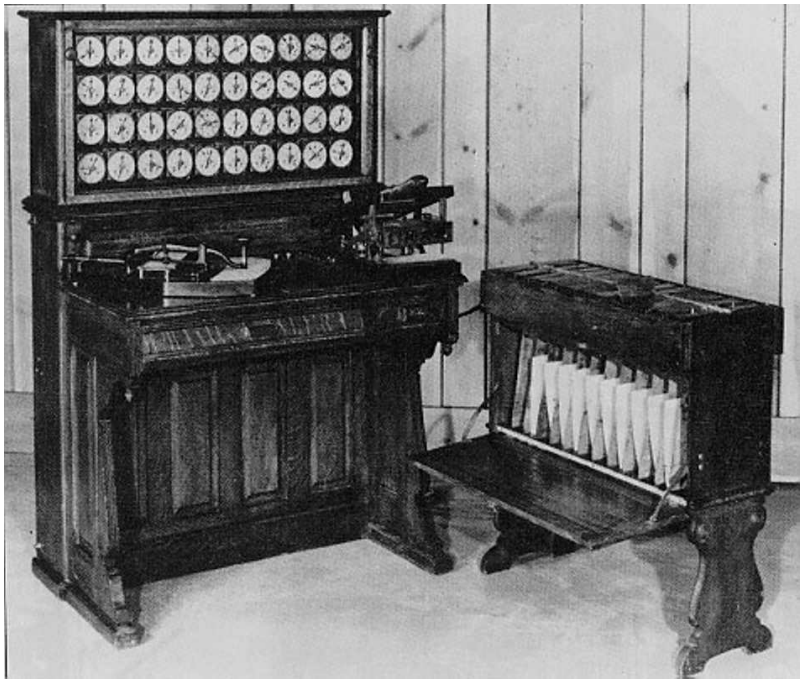
$= \Theta(n)$ if $k = O(n)$

Radix sort

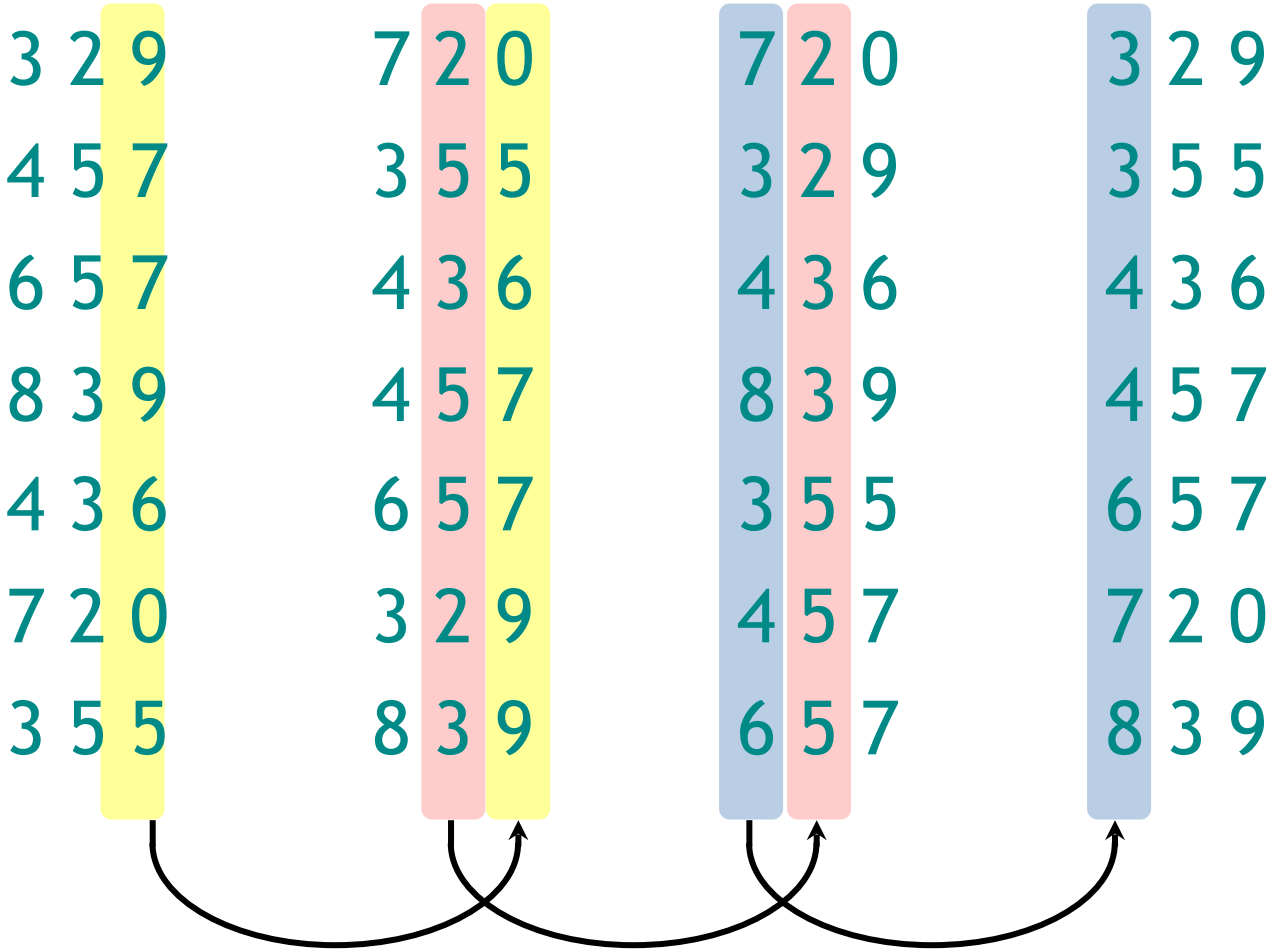
- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census

- Digit-by-digit sort

- Idea: Sort on *least-significant digit first*



Example of radix sort

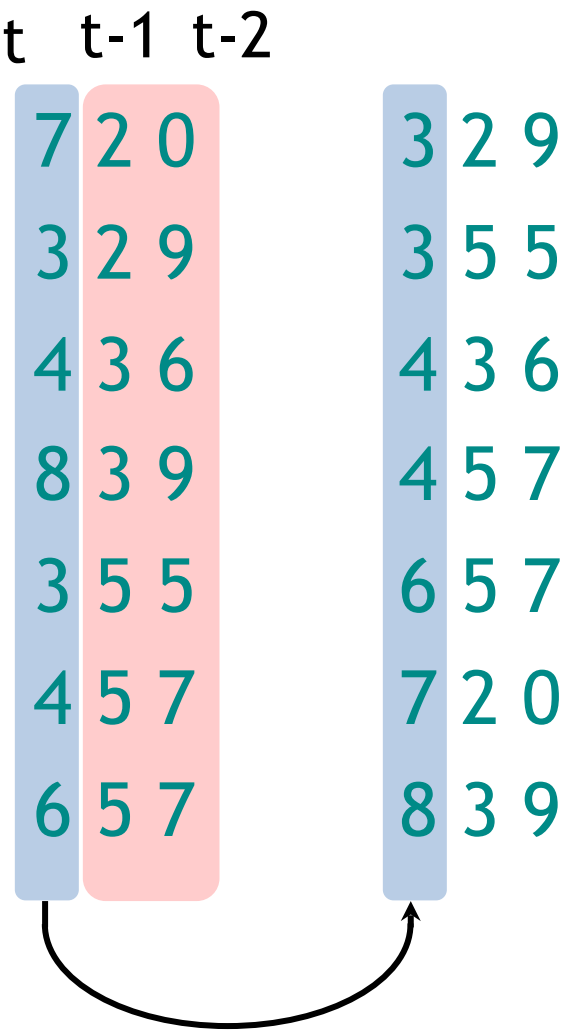


Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits

- Sort on digit t

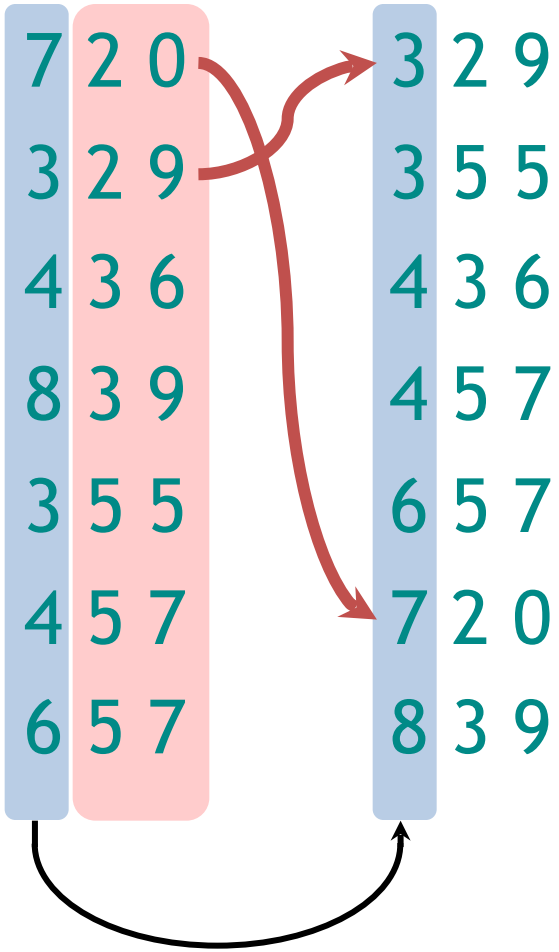


Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits

- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted

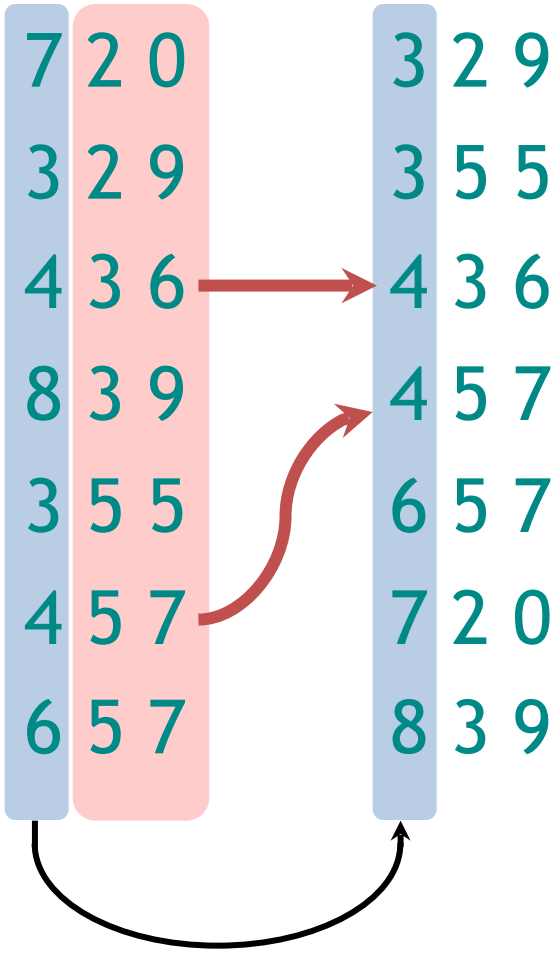


Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits

- Sort on digit t
  - Two numbers that differ in digit t will be correctly sorted
  - Two numbers equal in digit t are put in the same order as the order based on the previous t-1 digits, which will be correctly sorted



Runtime analysis of radix sort

- We are given n integers, each integer $\le M$, in base k
  - each integer has $d = \log_k M$ digits, a digit is in $\{0,1,\ldots,k-1\}$
- Assume counting sort is the auxiliary stable sort

Counting sort: we need

$\Theta(n+k)$ per digit

$\Rightarrow \Theta((n+k)d) = \Theta((n+k)\log_k M)$

$\Rightarrow \Theta(n\log_n M)$ total time if $k = n$

$\Rightarrow \Theta(nc)$ if $M \le n^c$ for some $c > 0$

In practice we expect M to grow as n grows, hence we assume $M < n^c$

$M = 1000$, $d = 3$

base $k = 10$

n

4 3 6

8 3 9

3 5 5

⋮

exercise

- Sort the following numbers by radix sort
- 123, 583, 154, 567, 689, 625, 456

Exercise answer

123	123	123	123
583	583	625	154
154	154	154	456
567	625	456	567
689	456	567	583
625	567	583	625
456	689	689	689

Conclusions

- Ω(nlogn) is the lower bounds for searching and sorting algorithms
- But O(n) is possible if:
  - if the range *M* of possible values grows at most proportionally with the size of the problem (size of problem = *n* = # of items), then we can use counting sort
  - if *M* grows even faster but $O(n^c)$ for some c>1, then we can still keep the linear time result by using radix sort (and choosing optimally the base to represent our integers)