## CS473-Algorithms I

## Lecture 9 <br> Sorting in Linear Time

View in slide-show mode

## How Fast Can We Sort?

- The algorithms we have seen so far:
- Based on comparison of elements
- We only care about the relative ordering between the elements (not the actual values)
$\square$ The smallest worst-case runtime we have seen so far: O (nlgn)
$\square$ Is O(nlgn) the best we can do?
- Comparison sorts: Only use comparisons to determine the relative order of elements.


## Decision Trees for Comparison Sorts

- Represent a sorting algorithm abstractly in terms of a decision tree
- A binary tree that represents the comparisons between elements in the sorting algorithm
$\square$ Control, data movement, and other aspects are ignored
- One decision tree corresponds to one sorting algorithm and one value of $n$ (input size)


## Reminder: Insertion Sort (from Lecture 1)

## Insertion-Sort (A)

1. for $\mathrm{j} \leftarrow 2$ to n do
2. $\mathrm{key} \leftarrow \mathrm{A}[\mathrm{j}]$;
3. $\mathrm{i} \leftarrow \mathrm{j}-1$;
4. while $\mathrm{i}>0$ and $\mathrm{A}[\mathrm{i}]>$ key
do
5. $\mathrm{A}[\mathrm{i}+1] \leftarrow \mathrm{A}[\mathrm{i}]$;
6. $\quad i \leftarrow i-1$;
endwhile
7. $\mathrm{A}[\mathrm{i}+1] \leftarrow$ key;
endfor


Loop invariant:
The subarray $\mathrm{A}[1 . . \mathrm{j}-1]$
is always sorted


## Reminder: Insertion Sort (from Lecture 1)

## Insertion-Sort (A)

1. for $\mathrm{j} \leftarrow 2$ to n do
2. $\mathrm{key} \leftarrow \mathrm{A}[\mathrm{j}]$;
3. $\mathrm{i} \leftarrow \mathrm{j}-1$;
4. while $\mathrm{i}>0$ and $\mathrm{A}[\mathrm{i}]>$ key do
5. $\mathrm{A}[\mathrm{i}+1] \leftarrow \mathrm{A}[\mathrm{i}]$;
6. $\quad \mathrm{i} \leftarrow \mathrm{i}-1$;
endwhile
7. $\mathrm{A}[\mathrm{i}+1] \leftarrow \mathrm{key}$;

Shift right the entries in A[1..j-1] that are > key
endfor

## Reminder: Insertion Sort (from Lecture 1)

## Insertion-Sort (A)

1. for $\mathrm{j} \leftarrow 2$ to n do
2. $\mathrm{key} \leftarrow \mathrm{A}[\mathrm{j}]$;
3. $\quad i \leftarrow j-1$;
4. while $\mathrm{i}>0$ and $\mathrm{A}[\mathrm{i}]>$ key
do
5. $\mathrm{A}[\mathrm{i}+1] \leftarrow \mathrm{A}[\mathrm{i}]$;
6. $\quad \mathrm{i} \leftarrow \mathrm{i}-1$;

## endwhile

7. $\mathrm{A}[\mathrm{i}+1] \leftarrow \mathrm{key}$;
endfor
Insert key to the correct locatio
End of iter $j$ : $A[1 . . j]$ is sorted

## Different Outcomes for Insertion Sort and $n=3$ Input: $<\mathrm{a}_{1}, \mathrm{a}_{2}, \mathrm{a}_{3}>$



## Decision Tree for Insertion Sort and $n=3$



## Decision Tree Model for Comparison Sorts

- Internal node (i.j): Comparison between elements $\mathrm{a}_{\mathrm{i}}$ and $\mathrm{a}_{\mathrm{j}}$
- Leaf node: An output of the sorting algorithm
- Path from root to a leaf: The execution of the sorting algorithm for a given input
- All possible executions are captured by the decision tree
- All possible outcomes (permutations) are in the leaf nodes


## Decision Tree for Insertion Sort and $\mathrm{n}=3$

Input: <9, 4, 6>


## Decision Tree Model

- A decision tree can model the execution of any comparison sort:
$\square$ One tree for each input size n
$\square$ View the algorithm as splitting whenever it compares two elements
$\square$ The tree contains the comparisons along all possible instruction traces

The running time of the algorithm $=$ the length of the path taken
Worst case running time $=$ height of the tree

## Lower Bound for Comparison Sorts

Let n be the number of elements in the input array.

- What is the min number of leaves in the decision tree?
n ! (because there are $n$ ! permutations of the input array, and all possible outputs must be captured in the leaves)
- What is the max number of leaves in a binary tree of height h?
$2^{\text {h }}$
- So, we must have:

$$
2^{\mathrm{h}} \geq \mathrm{n}!
$$

## Lower Bound for Decision Tree Sorting

## Theorem: Any comparison sort algorithm requires

 $\Omega(\mathrm{nlgn})$ comparisons in the worst case.Proof: We'll prove that any decision tree corresponding to a comparison sort algorithm must have height $\Omega$ (nlgn)

$$
\begin{aligned}
2^{\mathrm{h}} & \geq \mathrm{n}!\quad(\text { from previous slide }) \\
\mathrm{h} & \geq \lg (\mathrm{n}!) \\
& \geq \lg \left((\mathrm{n} / \mathrm{e})^{\mathrm{n}}\right) \quad(\text { Stirling's approximation }) \\
& =\mathrm{nlgn}-\mathrm{n} \operatorname{lge} \\
& =\Omega(\mathrm{nlgn})
\end{aligned}
$$

## Lower Bound for Decision Tree Sorting

Corollary: Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof: The O(nlgn) upper bounds on the runtimes for heapsort and merge sort match the $\Omega$ (nlgn) worst-case lower bound from the previous theorem.

## Sorting in Linear Time

## Counting sort: No comparisons between elements

Input: $\mathrm{A}[1 . . \mathrm{n}]$, where $\mathrm{A}[\mathrm{j}] \in\{1,2, \ldots, \mathrm{k}\}$
Output: B[1 .. n], sorted
Auxiliary storage: $\mathrm{C}[1 . . \mathrm{k}]$

## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& \text { // C[i] = |\{key = i\}| } \\
& \text { for } \mathrm{i} \leftarrow 2 \text { to } \mathrm{k} \text { do } \\
& \mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1] \\
& / / C[i]=|\{k e y \leq i\}| \\
& \text { for } \mathrm{j} \leftarrow \mathrm{n} \text { downto } 1 \text { do } \\
& \mathrm{B}[\mathrm{C}[\mathrm{~A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}] \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]-1
\end{aligned}
$$

## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=\mid\{\text { key }=\mathrm{i}\} \mid
\end{aligned}
$$

## Step 1: Initialize all counts to 0


for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / C[i]=|\{k e y \leq i\}|$

for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do $\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$


## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=\mid\{\text { key }=\mathrm{i}\} \mid
\end{aligned}
$$

## Step 2: Count the number of occurrences

 of each value in the input array
for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / \mathrm{C}[\mathrm{i}]=|\{\mathrm{key} \leq \mathrm{i}\}|$

## B: <br> 

for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do
$\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$


## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=\mid\{\text { key }=\mathrm{i}\} \mid
\end{aligned}
$$

Step 3: Compute the number of elements less than or equal to each value

for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / C[i]=|\{k e y \leq i\}|$
for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do
$\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$


## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=|\{\mathrm{key}=\mathrm{i}\}|
\end{aligned}
$$

for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / C[i]=|\{k e y \leq i\}|$
for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do
$\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$

## Step 4: Populate the output array

There are $C[3]=3$ elts that are $\leq 3$


## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=|\{\mathrm{key}=\mathrm{i}\}|
\end{aligned}
$$

for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / C[i]=|\{k e y \leq i\}|$
for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do
$\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$

Step 4: Populate the output array
There are $C[4]=5$ elts that are $\leq 4$


## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=\mid\{\text { key }=\mathrm{i}\} \mid
\end{aligned}
$$

for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / C[i]=|\{k e y \leq i\}|$
for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do
$\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$

## Step 4: Populate the output array

There are $C[3]=2$ elts that are $\leq 3$


## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=|\{\mathrm{key}=\mathrm{i}\}|
\end{aligned}
$$

for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / C[i]=|\{k e y \leq i\}|$
for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do
$\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$

## Step 4: Populate the output array

There are $C[1]=1$ elts that are $\leq 1$


## Counting Sort

$$
\begin{aligned}
& \text { for } \mathrm{i} \leftarrow 1 \text { to } \mathrm{k} \text { do } \\
& \quad \mathrm{C}[\mathrm{i}] \leftarrow 0 \\
& \text { for } \mathrm{j} \leftarrow 1 \text { to } \mathrm{n} \text { do } \\
& \mathrm{C}[\mathrm{~A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{~A}[\mathrm{j}]]+1 \\
& / / \mathrm{C}[\mathrm{i}]=\mid\{\text { key }=\mathrm{i}\} \mid
\end{aligned}
$$

for $\mathrm{i} \leftarrow 2$ to k do $\mathrm{C}[\mathrm{i}] \leftarrow \mathrm{C}[\mathrm{i}]+\mathrm{C}[\mathrm{i}-1]$
$/ / C[i]=|\{k e y \leq i\}|$
for $\mathrm{j} \leftarrow \mathrm{n}$ downto 1 do
$\mathrm{B}[\mathrm{C}[\mathrm{A}[\mathrm{j}]]] \leftarrow \mathrm{A}[\mathrm{j}]$
$\mathrm{C}[\mathrm{A}[\mathrm{j}]] \leftarrow \mathrm{C}[\mathrm{A}[\mathrm{j}]]-1$

## Step 4: Populate the output array

There are $C[4]=4$ elts that are $\leq 4$


## Counting Sort: Runtime Analysis



## Counting Sort: Runtime

Runtime is $\Theta(\mathrm{n}+\mathrm{k})$

- If $k=O(n)$, then counting sort takes $\Theta(n)$
- Question: We proved a lower bound of $\Theta$ (nlgn) before! Where is the fallacy?
- Answer:
$\square \Theta(\mathrm{nlgn})$ lower bound is for comparison-based sorting
$\square$ Counting sort is not a comparison sort
$\square$ In fact, not a single comparison between elements occurs!


## Stable Sorting

Counting sort is a stable sort: It preserves the input order among equal elements.
$\square$ i.e. The numbers with the same value appear in the output array in the same order as they do in the input array.


Exercise: Which other sorting algorithms have this property?

## Radix Sort

- Origin: Herman Hollerith's card-sorting machine for the 1890 US Census.
- Basic idea: Digit-by-digit sorting
- Two variations:
- Sort from MSD to LSD (bad idea)
- Sort from LSD to MSD (good idea)
- LSD/MSD: Least/most significant digit


## Herman Hollerith (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.

- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines (IBM).


## Hollerith Punched Card

- 12 rows and 24 columns $\square$ coded for age, state of residency, gender, etc.

Punched card: A piece of stiff paper that contains digital information represented by the presence or absence of holes.

## "Modern" IBM card

## One character per column

```
01234567S9月BCDEFGHIJKLHNOFQRSTUUMXYZ INTRODUCTON TO ALGORITHNS 09/24/2001
```




```
    ! !! lun!!!!
```

```
    ! !! lun!!!!
```






```
22|22222222|22222222|2222222|22222222222222222222222222222222|2222|22|2222222222
```

```
22|22222222|22222222|2222222|22222222222222222222222222222222|2222|22|2222222222
```














```
999999999|99999999|99999999|9999999|9|99|999999999999999||999999|999999999999999
```

```
999999999|99999999|99999999|9999999|9|99|999999999999999||999999|999999999999999
```

So, that's why text windows have 80 columns!

## Hollerith Tabulating Machine and Sorter



- Mechanically sorts the cards based on the hole locations.
- Sorting performed for one column at a time
- Human operator needed to load/retrieve/move cards at each stage


## Hollerith's MSD-First Radix Sort

- Sort starting from the most significant digit (MSD)
- Then, sort each of the resulting bins recursively
- At the end, combine the decks in order



## Hollerith's MSD-First Radix Sort

- To sort a subset of cards recursively:
- All the other cards need to be removed from the machine, because the machine can handle only one sorting problem at a time.
- The human operator needs to keep track of the intermediate card piles

| 329 | to sort these two cards | $\begin{array}{lll} 3 & 29 \\ 3 & 5 & 5 \end{array}$ |
| :---: | :---: | :---: |
| 355 | recursively, remove all |  |
| 457 | the other cards from |  |
| 436 | \| |  |
| 657 | intermediate pile |  |
| 720 | $457,436,657,720,839$ |  |
| 839 | 457, 436, 657, 720, 839 |  |

## Hollerith's MSD-First Radix Sort

- MSD-first sorting may require:
-- very large number of sorting passes
-- very large number of intermediate card piles to maintain
- S(d): \# of passes needed to sort d-digit numbers (worst-case)
- Recurrence:

$$
S(d)=10 S(d-1)+1 \quad \text { with } S(1)=1
$$

Reminder: Recursive call made to each subset with the same most significant digit (MSD)

## Hollerith's MSD-First Radix Sort Recurrence: $\mathrm{S}(\mathrm{d})=10 \mathrm{~S}(\mathrm{~d}-1)+1$

$$
\begin{aligned}
\mathrm{S}(\mathrm{~d}) & =10 \mathrm{~S}(\mathrm{~d}-1)+1 \\
& =10(10 \mathrm{~S}(\mathrm{~d}-2)+1)+1 \\
& =10(10(10 \mathrm{~S}(\mathrm{~d}-3)+1)+1)+1 \\
& =10^{\mathrm{i}} \mathrm{~S}(\mathrm{~d}-\mathrm{i})+10^{\mathrm{i}-1}+10^{\mathrm{i}-2}+\ldots+10^{1}+10^{0}
\end{aligned}
$$

Iteration terminates when $\mathrm{i}=\mathrm{d}-1$ with $\mathrm{S}(\mathrm{d}-(\mathrm{d}-1))=\mathrm{S}(1)=1$

$$
S(d)=\sum_{i=0}^{d-1} 10^{i}=\frac{10^{d}-1}{10-1}=\frac{1}{9}\left(10^{d}-1\right) \longmapsto S(d)=\frac{1}{9}\left(10^{d}-1\right)
$$

## Hollerith's MSD-First Radix Sort

P(d): \# of intermediate card piles maintained (worst-case)
Reminder: Each routing pass generates 9 intermediate piles except the sorting passes on least significant digits (LSDs)

There are $10^{\mathrm{d}-1}$ sorting calls to LSDs

$$
\begin{aligned}
\mathrm{P}(\mathrm{~d}) & =9\left(\mathrm{~S}(\mathrm{~d})-10^{\mathrm{d}-1}\right)=9\left(\left(10^{\mathrm{d}}-1\right) / 9-10^{\mathrm{d}-1}\right) \\
& =\left(10^{\mathrm{d}}-1-9 \cdot 10^{\mathrm{d}-1}\right)=10^{\mathrm{d}-1}-1 \\
\mathrm{P}(\mathrm{~d}) & =10^{\mathrm{d}-1}-1
\end{aligned}
$$

Alternative solution: Solve the recurrence:

$$
\begin{aligned}
& \mathrm{P}(\mathrm{~d})=10 \mathrm{P}(\mathrm{~d}-1)+9 \\
& \mathrm{P}(1)=0
\end{aligned}
$$

## Hollerith's MSD-First Radix Sort

- Example: To sort 3 digit numbers, in the worst case: $S(d)=(1 / 9)\left(10^{3}-1\right)=111$ sorting passes needed $\mathrm{P}(\mathrm{d})=10^{\mathrm{d}-1}-1=99$ intermediate card piles generated
- MSD-first approach has more recursive calls and intermediate storage requirement
- Expensive for a "tabulating machine" to sort punched cards
- Overhead of recursive calls in a modern computer


## LSD-First Radix Sort

- Least significant digit (LSD)-first radix sort seems to be a folk invention originated by machine operators.
- It is the counter-intuitive, but the better algorithm.
- Basic algorithm:

Sort numbers on their LSD first
Stable sorting needed!!!
Combine the cards into a single deck in order
Continue this sorting process for the other digits from the LSD to MSD

- Requires only d sorting passes
$\square$ No intermediate card pile generated


## LSD-first Radix Sort: Example

Step 1: Sort $1^{\text {st }}$ digit

| 3 | 2 | 9 |
| :--- | :--- | :--- |
| 4 | 5 | 7 |
| 6 | 5 | 7 |
| 8 | 3 | 9 |
| 4 | 3 | 6 |
| 7 | 2 | 0 |
| 3 | 5 | 5 |$\quad \leadsto$| 7 | 2 | 0 |
| :--- | :--- | :--- | :--- |
| 3 | 5 | 5 |
| 4 | 3 | 6 |
| 4 | 5 | 7 |
| 6 | 5 | 7 |
| 3 | 2 | 9 |
| 8 | 3 | 9 |

Step 2: Sort $2^{\text {nd }}$ digit

| 7 | 2 | 0 |
| :--- | :--- | :--- |
| 3 | 5 | 5 |
| 4 | 3 | 6 |
| 4 | 5 | 7 |
| 6 | 5 | 7 |
| 3 | 2 | 9 |
| 8 | 3 | 9 |$\quad$| 7 |  |  |
| :--- | :--- | :--- | :--- |
| 3 | 2 | 0 |
| 3 | 3 | 9 |
| 8 | 3 | 6 |
| 3 | 5 | 5 |
| 4 | 5 | 7 |
| 6 | 5 | 7 |

Step 3: Sort $3^{\text {rd }}$ digit
\(\left.\begin{array}{|lll}\hline 7 \& 2 \& 0 <br>
3 \& 2 \& 9 <br>
4 \& 3 \& 6 <br>
8 \& 3 \& 9 <br>
3 \& 5 \& 5 <br>
4 \& 5 \& 7 <br>

6 \& 5 \& 7\end{array}\right] \quad\)| 3 | 2 | 9 |
| :--- | :--- | :--- |
| 3 | 5 | 5 |
| 4 | 3 | 6 |
| 4 | 5 | 7 |
| 6 | 5 | 7 |
| 7 | 2 | 0 |
| 8 | 3 | 9 |

## Correctness of Radix Sort (LSD-first)

## Proof by induction: Base case: $\mathrm{d}=1$ is correct (trivial)

 Inductive hyp: Assume the first d-1 digits are sorted correctly Prove that all d digits are sorted correctly after sorting digit d

## Radix Sort: Runtime

- Use counting-sort to sort each digit

Reminder: Counting sort complexity: $\Theta(\mathrm{n}+\mathrm{k})$
n : size of input array
k : the range of the values

- Radix sort runtime: $\Theta(\mathrm{d}(\mathrm{n}+\mathrm{k}))$
d: \# of digits
- How to choose the d and k ?


## Radix Sort: Runtime - Example 1

## We have flexibility in choosing d and k

- Assume we are trying to sort 32-bit words
$\square$ We can define each digit to be 4 bits
$\square$ Then, the range for each digit $\mathrm{k}=2^{4}=16$
So, counting sort will take $\Theta(\mathrm{n}+16)$
- The number of digits $d=32 / 4=8$
$\square$ Radix sort runtime: $\Theta(8(n+16))=\Theta(n)$



## Radix Sort: Runtime - Example 2

## We have flexibility in choosing d and k

- Assume we are trying to sort 32-bit words
$\square$ Or, we can define each digit to be 8 bits
- Then, the range for each digit $\mathrm{k}=2^{8}=256$

So, counting sort will take $\Theta(\mathrm{n}+256)$

- The number of digits $d=32 / 8=4$
- Radix sort runtime: $\Theta(4(\mathrm{n}+256))=\Theta(\mathrm{n})$

| 8 bits | 8 bits | 8 bits | 8 bits |
| :--- | :--- | :--- | :--- |

## Radix Sort: Runtime

- Assume we are trying to sort b-bit words
$\square$ Define each digit to be $r$ bits
- Then, the range for each digit $k=2^{r}$

So, counting sort will take $\Theta\left(\mathrm{n}+2^{\mathrm{r}}\right)$

- The number of digits $d=b / r$

Radix sort runtime:

$$
T(n, b)=\Theta\left(\frac{b}{r}\left(n+2^{r}\right)\right)
$$

## Radix Sort: Runtime Analysis

$$
T(n, b)=\Theta\left(\frac{b}{r}\left(n+2^{r}\right)\right)
$$

Minimize $\mathrm{T}(\mathrm{n}, \mathrm{b})$ by differentiating and setting to 0
Or, intuitively:
We want to balance the terms $(\mathrm{b} / \mathrm{r})$ and $\left(\mathrm{n}+2^{\mathrm{r}}\right)$
Choose $r \approx \lg n$
If we choose $r \ll \operatorname{lgn} \square\left(n+2^{\prime}\right)$ term doesn't improve
If we choose $r \gg \operatorname{lgn} \square\left(n+2^{r}\right)$ increases exponentially

## Radix Sort: Runtime Analysis

$$
T(n, b)=\Theta\left(\frac{b}{r}\left(n+2^{r}\right)\right)
$$

Choose $\mathrm{r}=\operatorname{lgn} \quad \longrightarrow \mathrm{T}(\mathrm{n}, \mathrm{b})=\Theta(\mathrm{bn} / \mathrm{lgn})$

For numbers in the range from 0 to $\mathrm{n}^{\mathrm{d}}-1$, we have:
The number of bits $b=\lg \left(n^{d}\right)=d \operatorname{lgn}$
$\square$ Radix sort runs in $\Theta(\mathrm{dn})$

## Radix Sort: Conclusions

$$
\text { Choose } \mathrm{r}=\operatorname{lgn} \quad \Rightarrow \mathrm{T}(\mathrm{n}, \mathrm{~b})=\Theta(\mathrm{bn} / \operatorname{lgn})
$$

- Example: Compare radix sort with merge sort/heapsort

1 million ( $2^{20}$ ) 32 -bit numbers ( $n=2^{20}, b=32$ )
Radix sort: $\lceil 32 / 20\rceil=2$ passes
Merge sort/heap sort: $\operatorname{lgn}=20$ passes

- Downsides:

Radix sort has little locality of reference (more cache misses)
The version that uses counting sort is not in-place

- On modern processors, a well-tuned quicksort implementation typically runs faster.

