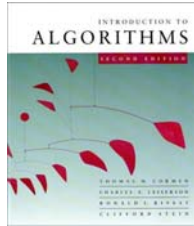




CS 5633 -- Spring 2010



Sorting

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk



How fast can we 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., insertion sort, merge sort, quicksort, heapsort.

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

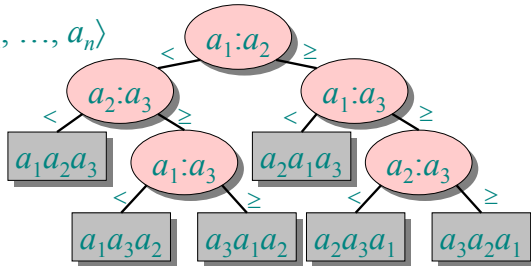
Is $O(n \log n)$ the best we can do?

Decision trees can help us answer this question.



Decision-tree example (Insertion sort)

Sort $\langle a_1, a_2, \dots, a_n \rangle$



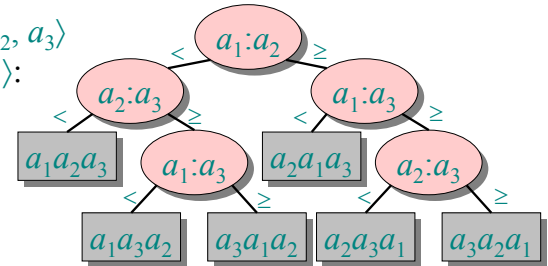
Each internal node is labeled $a_i:a_j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



Decision-tree example (Insertion sort)

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



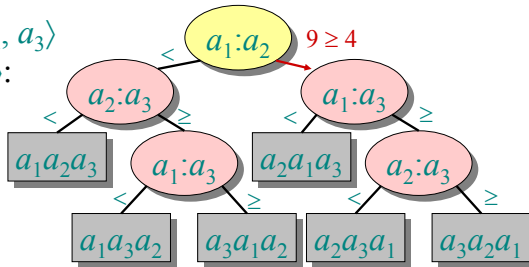
Each internal node is labeled $a_i:a_j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



Decision-tree example (Insertion sort)

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $a_i:a_j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

2/2/10

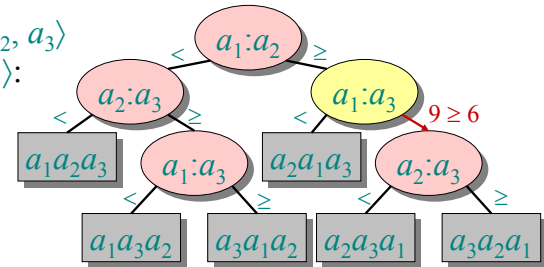
CS 5633 Analysis of Algorithms

5



Decision-tree example (Insertion sort)

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $a_i:a_j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

2/2/10

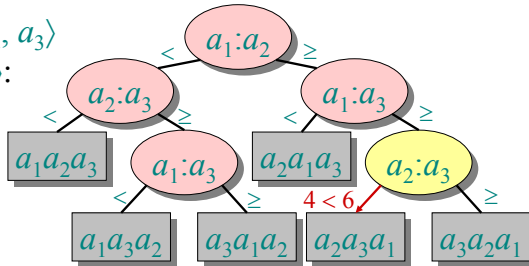
CS 5633 Analysis of Algorithms

6



Decision-tree example (Insertion sort)

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $a_i:a_j$ for $i, j \in \{1, 2, \dots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

2/2/10

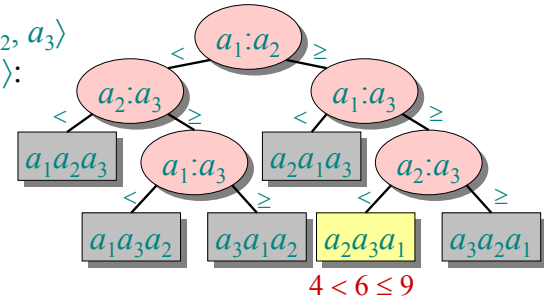
CS 5633 Analysis of Algorithms

7



Decision-tree example (Insertion sort)

Sort $\langle a_1, a_2, a_3 \rangle$
 $= \langle 9, 4, 6 \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ has been established.

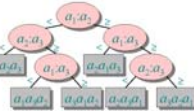
2/2/10

CS 5633 Analysis of Algorithms

8



Decision-tree model

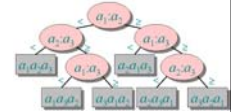


A decision tree models the execution of any comparison sorting algorithm:

- One tree per input size n .
- The tree contains **all** possible comparisons (= if-branches) that could be executed for **any** input of size n .
- The tree contains **all** comparisons along **all** possible instruction traces (= control flows) for **all** inputs of size n .
- For one input, only one path to a leaf is executed.
- Running time = length of the path taken.
- Worst-case running time = height of tree.



Lower bound for comparison sorting



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

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

$$\begin{aligned} \therefore h &\geq \log(n!) && (\log \text{ is mono. increasing}) \\ &\geq \log\left(\frac{n^n}{e^n}\right) && (\text{Stirling's formula}) \\ &= n \log n - n \log e \\ \Rightarrow h &\in \Omega(n \log n) \quad \square \end{aligned}$$



Lower bound for comparison sorting

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. \square



Sorting in linear time

Counting sort: No comparisons between elements.

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

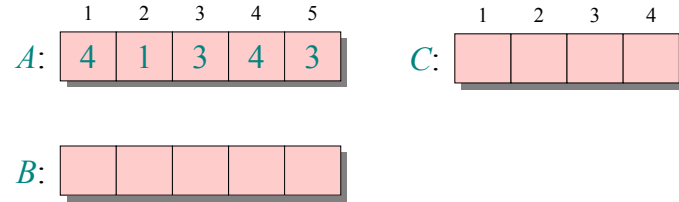


Counting sort

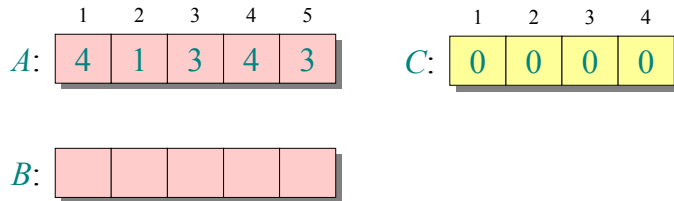
1. for $i \leftarrow 1$ to k
do $C[i] \leftarrow 0$
2. for $j \leftarrow 1$ to n
do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleleft C[i] = |\{\text{key} = i\}|$
3. for $i \leftarrow 2$ to k
do $C[i] \leftarrow C[i] + C[i-1] \quad \triangleleft C[i] = |\{\text{key} \leq i\}|$
4. for $j \leftarrow n$ downto 1
do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$



Counting-sort example



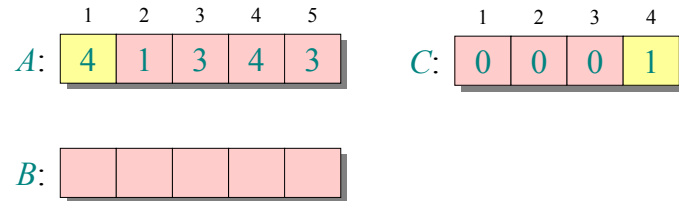
Loop 1



1. for $i \leftarrow 1$ to k
do $C[i] \leftarrow 0$



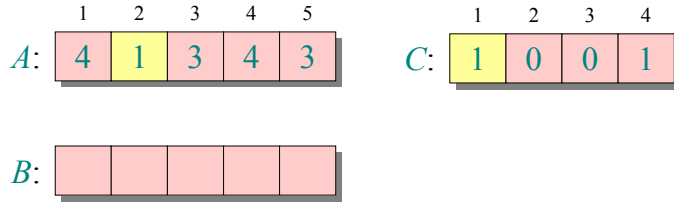
Loop 2



2. for $j \leftarrow 1$ to n
do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleleft C[i] = |\{\text{key} = i\}|$



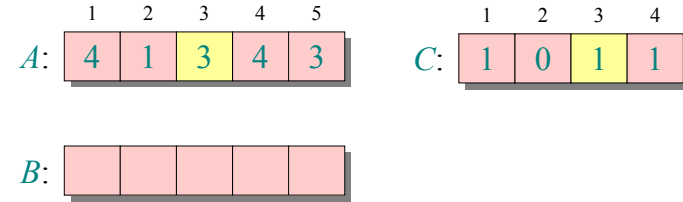
Loop 2



2. for $j \leftarrow 1$ to n
 do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleleft C[i] = |\{\text{key} = i\}|$



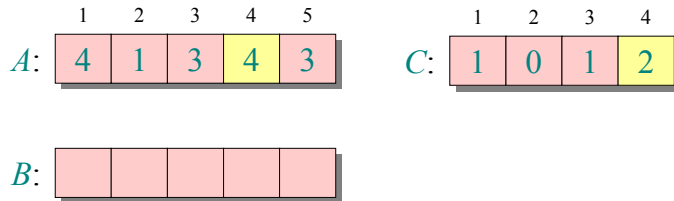
Loop 2



2. for $j \leftarrow 1$ to n
 do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleleft C[i] = |\{\text{key} = i\}|$



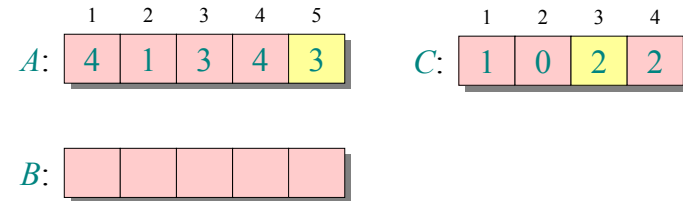
Loop 2



2. for $j \leftarrow 1$ to n
 do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleleft C[i] = |\{\text{key} = i\}|$



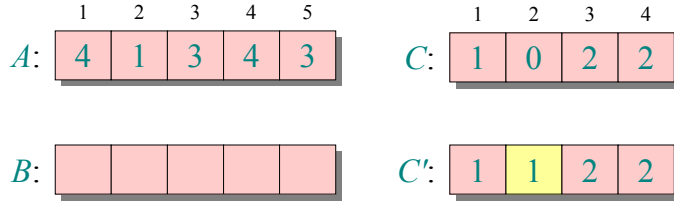
Loop 2



2. for $j \leftarrow 1$ to n
 do $C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleleft C[i] = |\{\text{key} = i\}|$



Loop 3



3. for $i \leftarrow 2$ to k
 do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleleft C[i] = |\{\text{key} \leq i\}|$

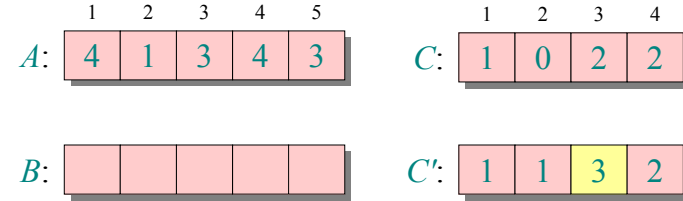
2/2/10

CS 5633 Analysis of Algorithms

21



Loop 3



3. for $i \leftarrow 2$ to k
 do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleleft C[i] = |\{\text{key} \leq i\}|$

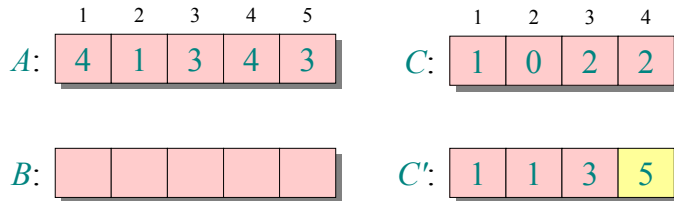
2/2/10

CS 5633 Analysis of Algorithms

22



Loop 3



3. for $i \leftarrow 2$ to k
 do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleleft C[i] = |\{\text{key} \leq i\}|$

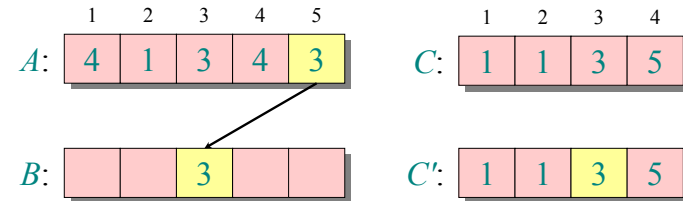
2/2/10

CS 5633 Analysis of Algorithms

23



Loop 4



4. for $j \leftarrow n$ downto 1
 do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

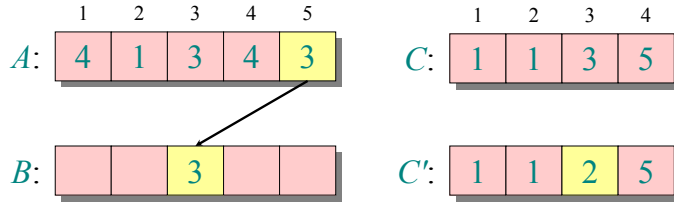
2/2/10

CS 5633 Analysis of Algorithms

24



Loop 4



```

4. for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 

```

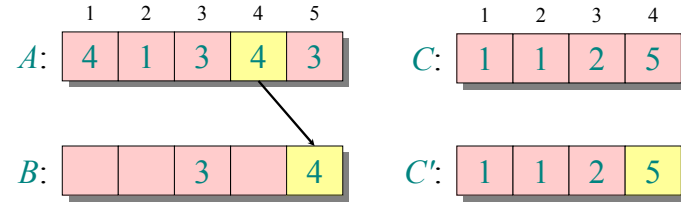
2/2/10

CS 5633 Analysis of Algorithms

25



Loop 4



```

4. for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 

```

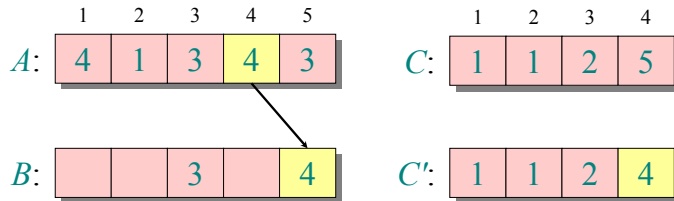
2/2/10

CS 5633 Analysis of Algorithms

26



Loop 4



```

4. for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 

```

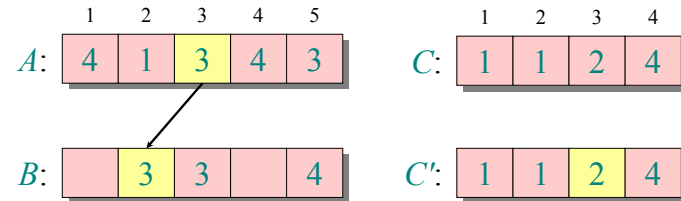
2/2/10

CS 5633 Analysis of Algorithms

27



Loop 4



```

4. for  $j \leftarrow n$  downto 1
  do  $B[C[A[j]]] \leftarrow A[j]$ 
      $C[A[j]] \leftarrow C[A[j]] - 1$ 

```

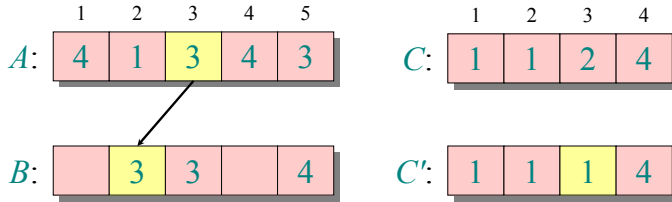
2/2/10

CS 5633 Analysis of Algorithms

28



Loop 4



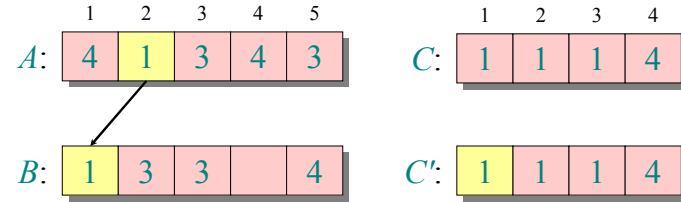
```

4. for j ← n downto 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1

```



Loop 4



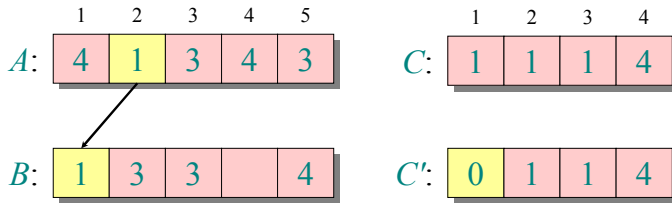
```

4. for j ← n downto 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1

```



Loop 4



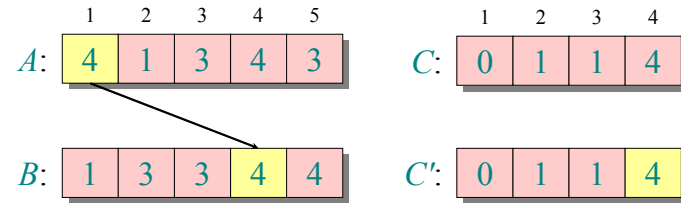
```

4. for j ← n downto 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1

```



Loop 4



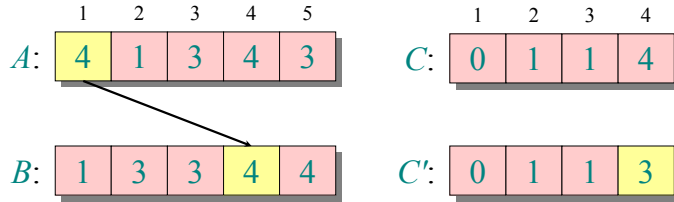
```

4. for j ← n downto 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1

```




Loop 4



```

4. for j ← n downto 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1

```



Analysis

$\Theta(k)$ { 1. for $i \leftarrow 1$ to k
do $C[i] \leftarrow 0$

$\Theta(n)$ { 2. for $j \leftarrow 1$ to n
do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ { 3. for $i \leftarrow 2$ to k
do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ { 4. for $j \leftarrow n$ downto 1
do $B[C[A[j]]] \leftarrow A[j]$
 $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$



Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \log n)$ time!
- Where's the fallacy?

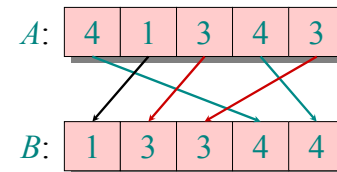
Answer:

- **Comparison sorting** takes $\Omega(n \log n)$ time.
- Counting sort is not a **comparison sort**.
- 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.



Exercise: What other sorts have this property?

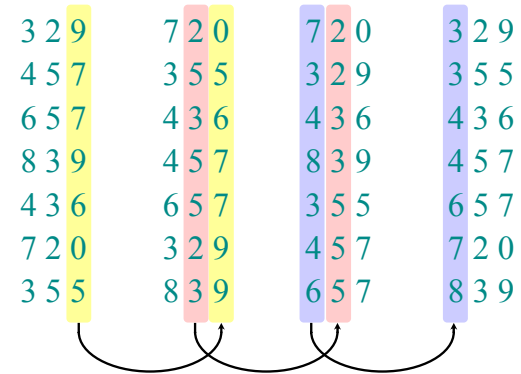


Radix sort

- **Origin:** Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix .)
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on **least-significant digit first** with an auxiliary **stable** sorting algorithm (like counting sort).



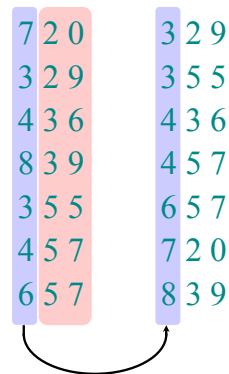
Operation of radix sort



Correctness of radix sort

Induction on digit position

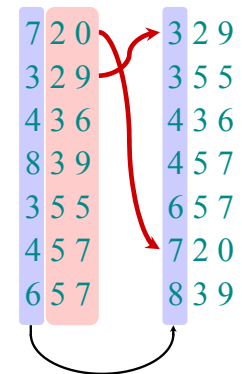
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t



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.

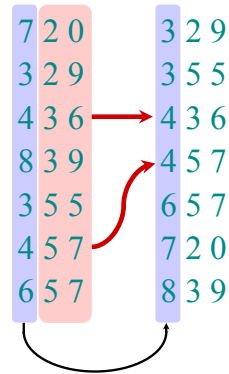




Correctness of radix sort

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 are correctly sorted.
 - Two numbers equal in digit t are put in the same order as the input \Rightarrow correct order.



2/2/10

CS 5633 Analysis of Algorithms

41

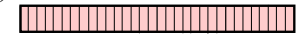


Analysis of radix sort

- Sort n computer words of b bits each.
- View each word as having b/r base- 2^r digits.

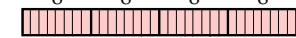
Example: 32-bit word ($b=32$)

$r = 1$: 32 base-2 digits



$\Rightarrow b/r = 32$ passes of counting sort on base-2 digits

$r = 8$: 32/8 base- 2^8 digits



$\Rightarrow b/r = 4$ passes of counting sort on base- 2^8 digits

$r = 16$: 32/16 base- 2^{16} digits



$\Rightarrow b/r = 2$ passes of counting sort on base- 2^{16} digits

2/2/10

CS 5633 Analysis of Algorithms

42



Analysis of radix sort

- Sort n computer words of b bits each.
- View each word as having b/r base- 2^r digits.
- Assume counting sort is the auxiliary stable sort.
- Make b/r passes of counting sort on base- 2^r digits

How many passes should we make?

2/2/10

CS 5633 Analysis of Algorithms

43



Analysis (continued)

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

- If each b -bit word is broken into r -bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time.
- Since there are b/r passes, we have

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

- Choose r to minimize $T(n, b)$:
Increasing r means fewer passes, but as $r \gg \log n$, the time grows exponentially.

2/2/10

CS 5633 Analysis of Algorithms

44



Choosing r

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

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \log n$ implies $T(n, b) = \Theta(bn/\log n)$.



Radix Sort with optimized r

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.

The runtime of radix sort is: $T(n, b) = \Theta(bn/\log n)$.

- Example:
For numbers in the range from 0 to $n^d - 1$, we have $b = d \log n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.
- Notice that counting sort runs in $O(n+k)$ time, where all numbers are in the range 1 through k .



Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quicksort do at least $\lceil \log 2000 \rceil = 11$ passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.



Appendix: Punched-card technology

- [Herman Hollerith \(1860-1929\)](#)
- [Punched cards](#)
- [Hollerith's tabulating system](#)
- [Operation of the sorter](#)
- [Origin of radix sort](#)
- ["Modern" IBM card](#)

Return to last slide viewed.





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.

2/2/10

CS 5633 Analysis of Algorithms

49



Punched cards

- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.



Replica of punch card from the 1900 U.S. census. [Howells 2000]

2/2/10

CS 5633 Analysis of Algorithms

50



Hollerith's tabulating system

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

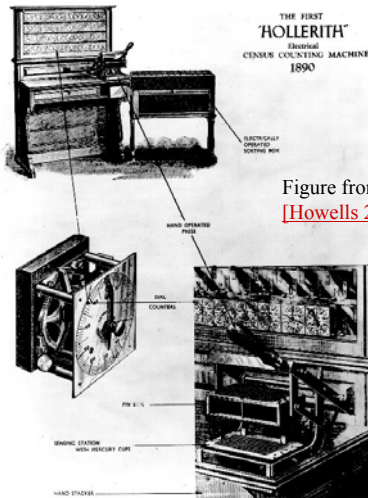


Figure from [Howells 2000].

2/2/10

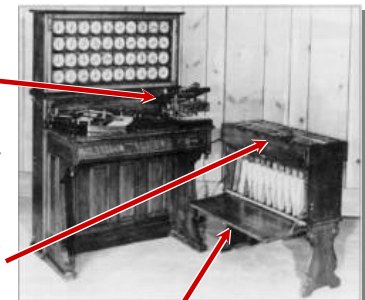
CS 5633 Analysis of Algorithms

51



Operation of the sorter

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into the bin and closes the lid.
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.



Hollerith Tabulator, Pantograph, Press, and Sorter

2/2/10

CS 5633 Analysis of Algorithms

52

