Lecture: Analysis of Algorithms (CS583 - 004)

Amarda Shehu

Spring 2019

1. Outline of Today's Class

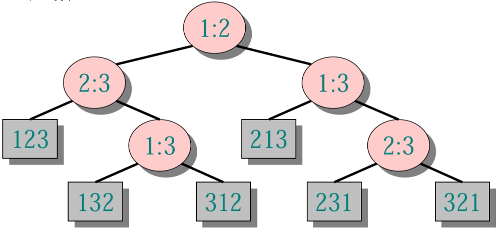
2. Lower Bound on Comparison-based Sorting
   - Decision Trees

How Fast Can We Sort?

- The sorting algorithms we have seen so far are insertion sort, mergesort, heapsort, and quicksort
- All these sorting algorithms are comparison sorts
- They rely on comparisons to determine the relative order of elements
- The best worst-case running time that we have seen for comparison sorting is $O(n \cdot lgn)$
- **Is $O(n \cdot lgn)$ the best we can do?**
- We need to employ decision trees to answer this question

Reason for Employing a Decision Tree

Sort $\langle a_1, a_2, \ldots, a_n \rangle$:

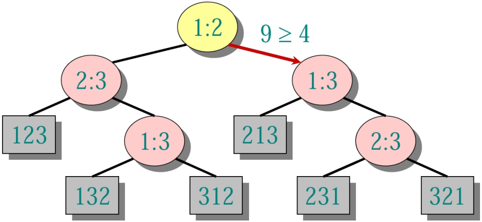


Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

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

Example of a Decision Tree

Sort $\langle a_1, a_2, \ldots, a_n \rangle =< 9, 4, 6 >$:

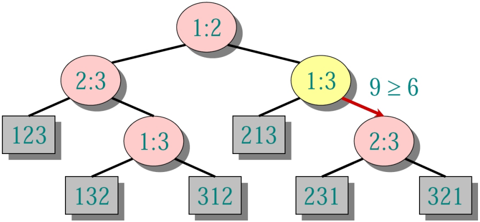


Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

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

Example of a Decision Tree

Sort $\langle a_1, a_2, \ldots, a_n \rangle =< 9, 4, 6 >$:

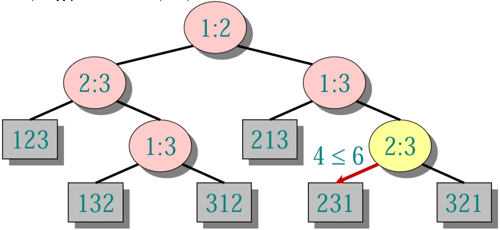


Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$

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

Example of a Decision Tree

Sort $\langle a_1, a_2, \ldots, a_n \rangle = < 9, 4, 6 >$:

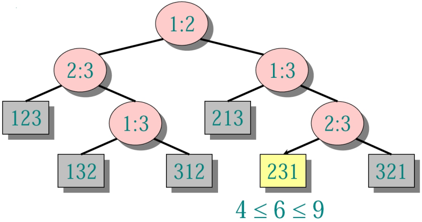


Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

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

Example of a Decision Tree

Sort $\langle a_1, a_2, \ldots, a_n \rangle = < 9, 4, 6 >$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots \pi(n) \rangle$ which establishes the ordering $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$

Decision Tree Model

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

- One tree for each input size n
- View the algorithm as splitting the tree whenever it compares two elements
- The tree contains the comparisons along all possible instruction traces
- The running time of the algorithm is then the length of the actual path taken
- Worst-case running time is the height of tree

Lower Bound for Decision Tree Sorting

Theorem: Any decision tree that can sort n elements must have height $\Omega(n \cdot lgn)$

Proof:

The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations.

A height h binary tree has $\leq 2^h$ leaves

Hence, $n! \leq 2^h$

$$
\begin{aligned}
h &\geq lg(n!) \\
&\geq lg((n/e)^n) \text{ -- Stirling's approximation} \\
&= n \cdot lgn - n \cdot lge \\
&\in \Omega(n \cdot lgn)
\end{aligned}
$$

Corollary: Heapsort and mergesort are asymptotically optimal comparison-based sorting algorithms

Lecture 3: Analysis of Algorithms (CS583 - 004)

Amarda Shehu

Spring 2019

1. Outline of Today's Class

2. Sorting in Linear Time
   - Counting Sort
   - Radix Sort

Sorting in Linear Time

- We can sort faster than $O(n \cdot lgn)$ if we do not compare the items being sorted against each other
- We can do this if we have additional information about the structure of the items
- Examples of Sorting Algorithms that do not compare items
  1. *Counting Sort*
  2. *Radix Sort*
  3. Bucket Sort

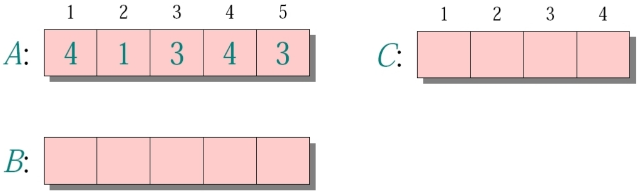
Counting Sort: Basic Idea and Pseudocode

- **Input:** $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots k\}$
- **Output:** $B[1 \ldots n]$ sorted
- **Auxiliary storage:** $C[1 \ldots k]$
- **Note:** all elements are in $\{1, 2, \ldots k\}$
- **Basic Idea:** Count the number of 1's, 2's, ..., k's.

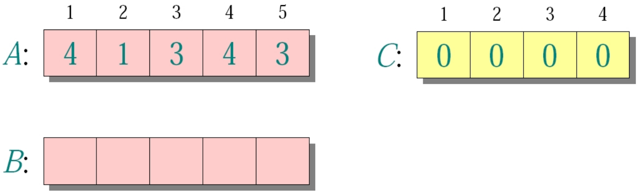
COUNTINGSORT(A, n)

```
for i ← 1 to k do
    C_i ← 0
for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1
    ▷ C[i] = |{key = i}|
for i ← 2 to k do
    C[i] ← C[i] + C[i − 1]
    ▷ C[i] = |{key ≤ i}|
for j ← n to 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j] − 1]
```

Counting Sort: Trace

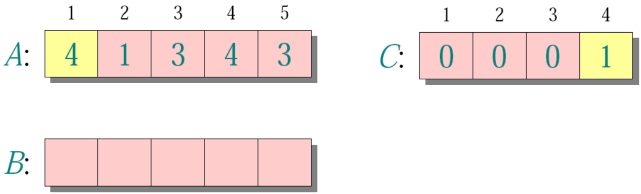


Counting Sort: Trace



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

Counting Sort: Trace



for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\quad \triangleright\ C[i] = |\{\text{key} = i\}|$

Counting Sort: Trace

A:

1	2	3	4	5
4	1	3	4	3

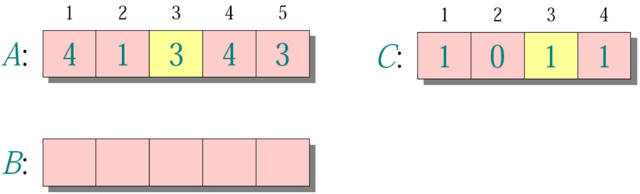
C:

1	2	3	4
1	0	0	1

B:

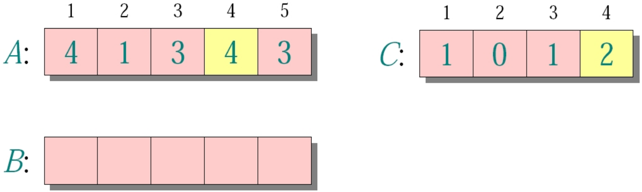
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$

Counting Sort: Trace



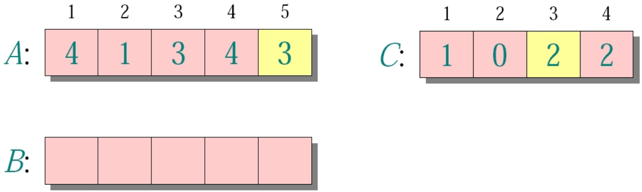
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$

Counting Sort: Trace



for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$

Counting Sort: Trace



for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$

Counting Sort: Trace

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

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

Counting Sort: Trace

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

C':

1	1	3	2

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

Counting Sort: Trace

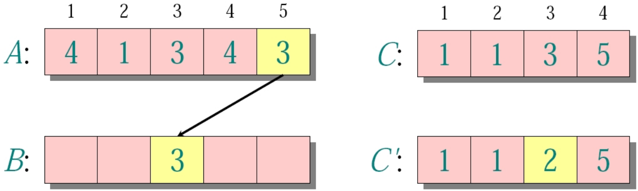
	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

for $i \leftarrow 2$ **to** k

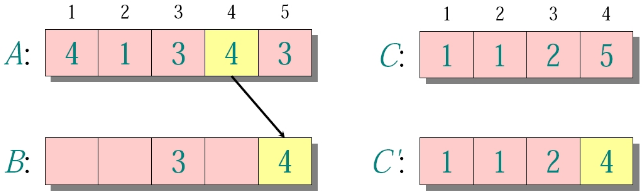
 do $C[i] \leftarrow C[i] + C[i-1]$ $\triangleright$ $C[i] = |\{\text{key} \leq i\}|$

Counting Sort: Trace



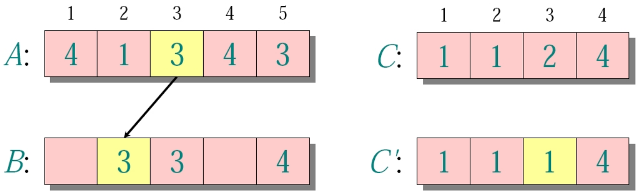
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort: Trace



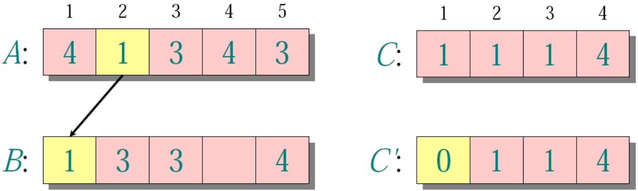
for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort: Trace



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

Counting Sort: Trace

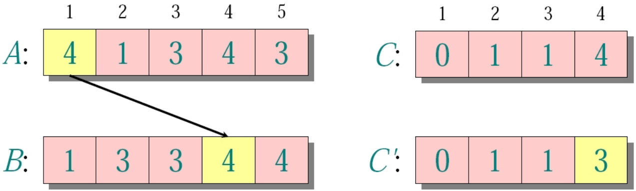


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort: Trace



for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow A[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort: Running Time Analysis

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

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

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

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

$\Theta(n + k)$

Counting Sort: Running Time Analysis

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

- But sorting takes $\Omega(n \cdot lgn)$ time!
- Where is the contradiction?

Counting Sort: Running Time Analysis

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

- But sorting takes $\Omega(n \cdot lgn)$ time!
- Where is the contradiction?

- *Comparison sorting* takes $\Omega(n \cdot lgn)$
- Counting sot is *not* a comparison sort
- Not a single comparison occurs in counting sort

Counting Sort: Running Time Analysis

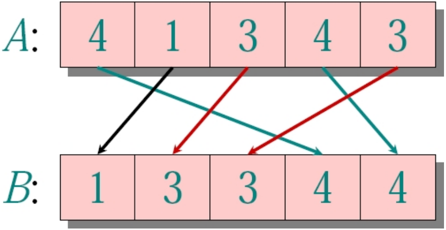
If $k \in O(n)$, then counting sort takes $O(n)$ time.

- But sorting takes $\Omega(n \cdot lgn)$ time!
- Where is the contradiction?

- *Comparison sorting* takes $\Omega(n \cdot lgn)$
- Counting sot is *not* a comparison sort
- Not a single comparison occurs in counting sort

Counting Sort is Stable

Counting sort is a stable sort because it preserves the input order among equal elements.

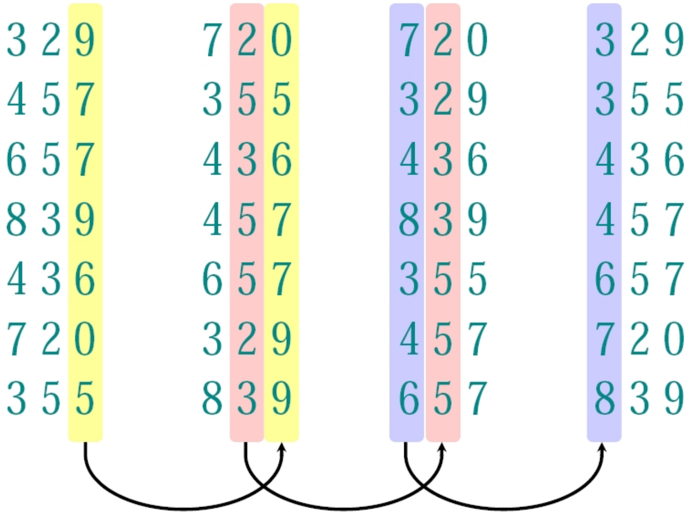


What other sorting algorithms have this property?

Radix Sort

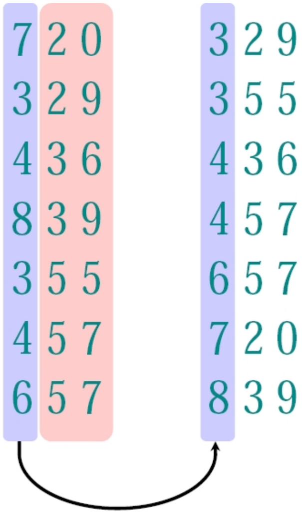
- History: Herman Hollerith's card-sorting machine for the 1890 US Census.
- Radix sort is digit-by-digit sort
- Hollerith's original (wrong) idea was to sort on most significant digit first
- The final (correct) idea was to sort on the least significant digit first with an auxiliary stable sort

Radix Sort in Action



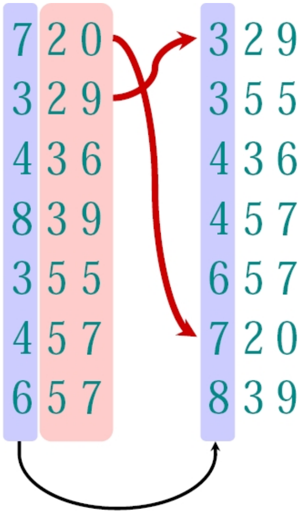
Radix Sort: Correctness

- The proof is by induction on the digit position
- Assume that the numbers are already sorted by their low-order $t-1$ digits
- Sort on digit t



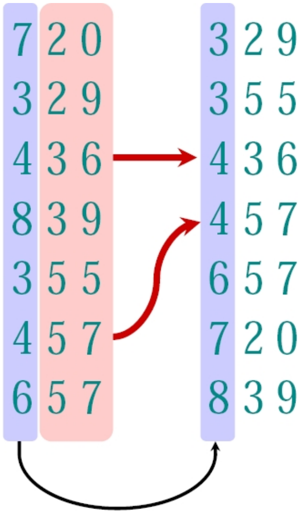
Radix Sort: Correctness

- The proof is by induction on the digit position
- Assume that the numbers are already sorted by their low-order $t - 1$ digits
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted



Radix Sort: Correctness

- The proof is by induction on the digit position
- Assume that the numbers are already 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 - the correct order



Radix Sort: Running Time Analysis

- Assume counting sort is the auxiliary stable sort
- Sort n computer words of b bits each
- Each word can be viewed as having b/r base-2^r

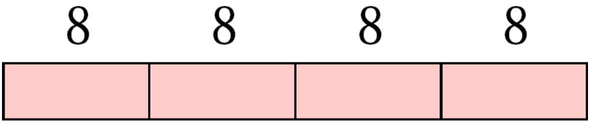


Figure: Example of a 32-bit word

- $r = 8$ means $b/r = 4$ passes of counting sort on base-2^8 digits
- $r = 16$ means $b/r = 2$ passes on base-2^{16} digits
- How many passes should one make?

Radix Sort: Running Time Analysis

Note: Counting sort takes $\theta(n+k)$ time to sort n numbers in the range 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:

$$T(n,b) \in \theta(\frac{b}{r}(n+2^r))$$

Choose r to minimize $T(n,b)$

- Increasing r means fewer passes, but as $r >> lgn$, the time grows exponentially

Radix Sort Runs in Linear Time: Choosing r

$$T(n, b) \in \theta(\frac{b}{r}(n + 2^r))$$

Minimize $T(n, b)$ by differentiating and setting the first derivative to 0. Recall that this is the technique to find minima or maxima for a function.

Alternatively, observe that we do not want $2^r >> n$, and so we can safely choose r to be as large as possible without violating this constraint.

Choosing $r = lgn$ implies that $T(n, b) \in \theta(bn/lgn)$

- For numbers in the range 0 to $n^d - 1$, we have that $b = d \cdot lgn$
- Hence, radix sort runs in $\theta(d \cdot n)$ time

Final Words on Radix Sort and Sorting Algorithms

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

Example: 32-bit numbers

- At most 3 passes when sorting ≥ 2000 numbers
- Mergesort and quicksort do at least $\lceil lg 2000 \rceil$ passes

Not all Rosy:

- Unlike quicksort, radix sort displays little locality of reference
- A well-tuned quicksort does better on modern processors that feature steep memory hierarchies