Lecture 3: Analysis of Algorithms (CS483 - 001)

Amarda Shehu

Spring 2017
Outline of Today’s Class

Sorting in Linear Time

1. Counting Sort
2. Radix Sort
We can sort faster than $O(n \cdot \log n)$ 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)**

1. for $i \leftarrow 1$ to $k$ do
2.  $C_i \leftarrow 0$
3. for $j \leftarrow 1$ to $n$ do
4.  $C[A[j]] \leftarrow C[A[j]] + 1$
   $\triangleright C[i] = |\{\text{key} = i\}|$
5. for $i \leftarrow 2$ to $k$ do
6.  $C[i] \leftarrow C[i] + C[i - 1]$
   $\triangleright C[i] = |\{\text{key} \leq i\}|$
7. for $j \leftarrow n$ to 1 do
Counting Sort: Trace

A: 4 1 3 4 3
B: 
C: 1 2 3 4
Counting Sort: Trace

\[ A: \begin{array}{ccccc} 
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 
\end{array} \]

\[ B: \]

\[ C: \begin{array}{cccc} 
1 & 2 & 3 & 4 \\
0 & 0 & 0 & 0 
\end{array} \]

\[ \text{for } i \leftarrow 1 \text{ to } k \]
\[ \text{do } C[i] \leftarrow 0 \]
Counting Sort: Trace

\[ A: \begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 \\
\end{array} \]

\[ B: \]

\[ C: \begin{array}{cccc}
1 & 2 & 3 & 4 \\
0 & 0 & 0 & 1 \\
\end{array} \]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}|\
\]
Counting Sort: Trace

\[ A: \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\hline
4 & 1 & 3 & 4 & 3
\end{array} \]

\[ B: \begin{array}{cccccc}
\hline
\end{array} \]

\[ C: \begin{array}{cccccc}
1 & 2 & 3 & 4 \\
\hline
1 & 0 & 0 & 1
\end{array} \]

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

\[
\begin{array}{ccccc}
& 1 & 2 & 3 & 4 & 5 \\
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
C: & 1 & 0 & 1 & 2 & 4 \\
\end{array}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}|
\]
Counting Sort: Trace

\[\text{for } j \leftarrow 1 \text{ to } n \]
\[\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key } = i\}|\]
Counting Sort: Trace

\[ A: \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 \\
\end{array} \]

\[ B: \begin{array}{cccccc}
1 & 2 & 3 & 4 \\
\end{array} \]

\[ C: \begin{array}{cccccc}
1 & 2 & 3 & 4 \\
1 & 0 & 2 & 2 \\
\end{array} \]

\[ C': \begin{array}{cccccc}
1 & 1 & 2 & 2 \\
\end{array} \]

\[
\text{for } i \leftarrow 2 \text{ to } k \\
\text{do } C[i] \leftarrow C[i] + C[i-1] \\
\triangleright C[i] = |\{\text{key } \leq i\}| 
\]
Counting Sort: Trace

for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$
      ▶ $C[i] = \lvert \{\text{key} \leq i\}\rvert$
Counting Sort: Trace

\[
\begin{array}{c|c|c|c|c|c} 
\text{A:} & 4 & 1 & 3 & 4 & 3 \\
\text{B:} & & & & & \\
\text{C:} & 1 & 0 & 2 & 2 \\
\text{C':} & 1 & 1 & 3 & 5 \\
\end{array}
\]

\[
\text{for } i \leftarrow 2 \ \text{to } k \\
\text{do } C[i] \leftarrow C[i] + C[i-1] \quad \triangleright C[i] = |\{\text{key} \leq i\}|
\]
Counting Sort: Trace

for $j \leftarrow n \text{ downto } 1$

\[\begin{align*}
& \text{do } B[C[A[j]]] \leftarrow A[j] \\
& \phantom{\text{do } } C[A[j]] \leftarrow C[A[j]] - 1
\end{align*}\]
Counting Sort: Trace

for $j \leftarrow n$ down to 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
Counting Sort: Trace

\[ \text{for } j \leftarrow n \text{ downto } 1 \]
\[ \text{do } B[C[A[j]]] \leftarrow A[j] \]
\[ C[A[j]] \leftarrow C[A[j]] - 1 \]
Counting Sort: Trace

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

$A$: 4 1 3 4 3

$B$: 1 3 3 4

$C$: 1 1 1 4

$C'$: 0 1 1 4
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

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

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

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

\[ \Theta(n) \quad \{ \quad \text{for } j \leftarrow n \text{ down to } 1 \quad \quad \text{do } B[C[A[j]]] \leftarrow A[j] \quad \} \]

\[ \Theta(n + k) \]

\[ C[A[j]] \leftarrow C[A[j]] - 1 \]
If $k \in O(n)$, then counting sort takes $O(n)$ time.

- But sorting takes $\Omega(n \cdot \log n)$ time!
- Where is the contradiction?
If $k \in O(n)$, then counting sort takes $O(n)$ time.

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

- *Comparison sorting* takes $\Omega(n \cdot \log n)$
- Counting sort is *not* a comparison sort
- Not a single comparison occurs in counting sort
If \( k \in O(n) \), then counting sort takes \( O(n) \) time.

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

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

329 720 720 329
457 355 329 355
657 436 436 436
839 457 839 457
436 657 355 657
720 329 457 720
355 839 657 839
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
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-$2^r$

\[ \begin{array}{cccc}
8 & 8 & 8 & 8 \\
\end{array} \]

**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:

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

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

- Increasing $r$ means fewer passes, but as $r \gg \log n$, the time grows exponentially.
Radix Sort Runs in Linear Time: Choosing $r$

\[ T(n, b) \in \theta\left(\frac{b}{r}(n + 2^r)\right) \]

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

- For numbers in the range 0 to $n^d - 1$, we have that $b = d \cdot \log n$
- Hence, radix sort runs in $\theta(d \cdot n)$ time
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 $\geq 2000$ 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