Lecture 9

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)
  - The smallest *worst-case runtime* we have seen so far: $O(n \log n)$
  - Is $O(n \log n)$ 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
  - 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** j ← 2 to n do
2.   key ← A[j];
3.   i ← j - 1;
4.   **while** i > 0 and A[i] > key do
5.     A[i+1] ← A[i];
6.     i ← i - 1;
7.   endwhile
8.   A[i+1] ← key;
9. **endfor**

**Loop invariant:**
The subarray A[1..j-1] is always sorted.
Reminder: Insertion Sort (from Lecture 1)

**Insertion-Sort** (A)

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

Shift right the entries in $A[1..j-1]$ that are $> key$
Reminder: Insertion Sort (from Lecture 1)

**Insertion-Sort (A)**

1. for j ← 2 to n do
2. key ← A[j];
3. i ← j - 1;
4. while i > 0 and A[i] > key do
5. A[i+1] ← A[i];
6. i ← i - 1;
endwhile
7. A[i+1] ← key;
endfor

*Insert key to the correct location*

*End of iter j: A[1..j] is sorted*
Different Outcomes for Insertion Sort and n=3

Input: \(<a_1, a_2, a_3>\)
Decision Tree for Insertion Sort and n=3
Decision Tree Model for Comparison Sorts

- *Internal node* \((i:j)\): Comparison between elements \(a_i\) and \(a_j\)

- *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 n=3

Input: <9, 4, 6>

output: <4, 6, 9>
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 whenever it compares two elements
  - 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 $h$?
  \[ 2^h \]
- So, we must have:
  \[ 2^h \geq n! \]
Lower Bound for Decision Tree Sorting

**Theorem**: Any comparison sort algorithm requires \( \Omega(n \lg n) \) comparisons in the worst case.

**Proof**: We’ll prove that any decision tree corresponding to a comparison sort algorithm must have height \( \Omega(n \lg n) \)

\[
2^h \geq n! \quad \text{(from previous slide)}
\]

\[
h \geq \lg(n!)
\]

\[
\geq \lg((n/e)^n) \quad \text{(Stirling’s approximation)}
\]

\[
= n \lg n - n \lg e
\]

\[
= \Omega(n \lg n)
\]
Lower Bound for Decision Tree Sorting

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

**Proof**: The $O(n \log n)$ upper bounds on the runtimes for heapsort and merge sort match the $\Omega(n \log n)$ worst-case lower bound from the previous theorem.
Counting sort: No comparisons between elements

Input: $A[1..n]$, where $A[j] \in \{1, 2, \ldots, k\}$

Output: $B[1..n]$, sorted

Auxiliary storage: $C[1..k]$
Counting Sort

```
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 downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1
```

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

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 downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1

Step 1: Initialize all counts to 0

A: 4 1 3 4 3

B: 

C: 0 0 0 0

1 2 3 4
Counting Sort

\begin{itemize}
  \item \textbf{Step 2:} Count the number of occurrences of each value in the input array
\end{itemize}

\begin{verbatim}
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 downto 1 do
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] − 1
\end{verbatim}

\begin{tabular}{c|c|c|c|c|c}
  \hline
  \textbf{j} & 1 & 2 & 3 & 4 & 3 \\
  \hline
  \textbf{A:} & 4 & 1 & 3 & 4 & 3 \\
  \hline
  \textbf{B:} & & & & & \\
  \hline
  \textbf{C:} & 1 & 0 & 2 & 2 & \\
  \hline
\end{tabular}
Counting Sort

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 downto 1 do
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] – 1

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

A: 4 1 3 4 3

B: 

C: 1 1 3 5

A: 1 2 3 4
Counting Sort

\begin{algorithm}
\begin{algorithmic}
\FOR{$i \leftarrow 1 \to k$}
\STATE{$C[i] \leftarrow 0$}
\ENDFOR

\FOR{$j \leftarrow 1 \to n$}
\STATE{$C[A[j]] \leftarrow C[A[j]] + 1$}
\STATE{$// C[i] = |\{\text{key} = i\}|$}
\ENDFOR

\FOR{$i \leftarrow 2 \to k$}
\STATE{$C[i] \leftarrow C[i] + C[i-1]$}
\STATE{$// C[i] = |\{\text{key} \leq i\}|$}
\ENDFOR

\FOR{$j \leftarrow n \ downto \ 1$}
\STATE{$B[C[A[j]]] \leftarrow A[j]$}
\STATE{$C[A[j]] \leftarrow C[A[j]] - 1$}
\ENDFOR
\end{algorithmic}
\end{algorithm}

\textbf{Step 4}: Populate the output array

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

\begin{align*}
\text{A:} & \quad 4 \quad 1 \quad 3 \quad 4 \quad 3 \\
\text{B:} & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
\text{C:} & \quad 1 \quad 1 \quad 2 \quad 5
\end{align*}
Counting Sort

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 downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1

Step 4: Populate the output array

There are C[4] = 5 elts that are ≤ 4

A: 4 1 3 4 3
B: 1 2 3 4 5
C: 1 1 2 4
### Counting Sort

**Step 4**: Populate the output array

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

```
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 downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] – 1
```

A: 4 1 3 4 3
B: 1 2 3 4 5
C: 1 1 1 4
Counting Sort

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 downto 1 do  
    B[C[A[j]]] ← A[j]  
    C[A[j]] ← C[A[j]] - 1  

**Step 4:** Populate the output array

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

![Diagram showing the process of counting sort with example arrays A, B, C.]

| A: 4 1 3 4 3 | j |
| 1 2 3 4 5 |
| B: 3 3 4 4 |
| C: 0 1 1 4 |
Counting Sort

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 downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1

**Step 4:** Populate the output array

There are C[4] = 4 elts that are ≤ 4
Counting Sort

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 downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1

After Count Sort:

A: 4 1 3 4 3

B: 1 2 3 4 5

C: 0 1 1 2

A: 4 1 3 4 3

B: 1 3 3 4 4

C: 0 1 1 2
Counting Sort: Runtime Analysis

for i ← 1 to k do
  C[i] ← 0
  Θ(k)

for j ← 1 to n do
  C[A[j]] ← C[A[j]] + 1
  Θ(n)
  // C[i] = |{key = i}|

for i ← 2 to k do
  C[i] ← C[i] + C[i-1]
  Θ(k)
  // C[i] = |{key ≤ i}|

for j ← n downto 1 do
  B[C[A[j]]] ← A[j]
  Θ(n)
  C[A[j]] ← C[A[j]] – 1

Total runtime: Θ(n+k)

n: size of the input array
k: the range of input values
Counting Sort: Runtime

- Runtime is $\Theta(n+k)$
- If $k = O(n)$, then counting sort takes $\Theta(n)$

**Question**: We proved a lower bound of $\Theta(n\log n)$ before! Where is the fallacy?

**Answer**:
- $\Theta(n\log n)$ lower bound is for comparison-based sorting
- Counting sort is not a comparison sort
- In fact, not a single comparison between elements occurs!
Stable Sorting

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

A: \[4 \ 1 \ 3 \ 4 \ 3\]
B: \[1 \ 3 \ 3 \ 4 \ 4\]

\textit{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
- 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

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

```
3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
```
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 to sort these two cards recursively, remove all the other cards from the machine.

Intermediate piles:

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:
  \[
  S(d) = 10 \ S(d-1) + 1 \quad \text{with} \quad S(1) = 1
  \]

**Reminder**: Recursive call made to each subset with the same most significant digit (MSD)
Hollerith’s MSD-First Radix Sort

**Recurrence:** \( S(d) = 10S(d-1) + 1 \)

\[
S(d) = 10 S(d-1) + 1 \\
= 10 (10 S(d-2) + 1) + 1 \\
= 10 (10 (10 S(d-3) + 1) + 1) + 1 \\
= 10^i S(d-i) + 10^{i-1} + 10^{i-2} + \ldots + 10^1 + 10^0
\]

Iteration terminates when \( i = d-1 \) with \( S(d-(d-1)) = S(1) = 1 \)

\[
S(d) = \sum_{i=0}^{d-1} 10^i = \frac{10^d - 1}{9} = \frac{1}{9} (10^d - 1) = \frac{1}{9} (10^d - 1)
\]
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 $10^{d-1}$ sorting calls to LSDs

$$P(d) = 9 (S(d) - 10^{d-1}) = 9 ((10^d - 1)/9 - 10^{d-1})$$

$$= (10^d - 1 - 9 \cdot 10^{d-1}) = 10^{d-1} - 1$$

$P(d) = 10^{d-1} - 1$

*Alternative solution*: Solve the recurrence:

$$P(d) = 10P(d-1) + 9$$

$P(1) = 0$
Hollerith’s MSD-First Radix Sort

- Example: To sort 3 digit numbers, in the worst case:
  \[ S(d) = \frac{1}{9} (10^3 - 1) = 111 \text{ sorting passes needed} \]
  \[ P(d) = 10^{d-1} - 1 = 99 \text{ intermediate card piles generated} \]

- 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
- 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
- No intermediate card pile generated

Stable sorting needed!!!
LSD-first Radix Sort: Example

**Step 1:** Sort 1\(^{st}\) digit

3 2 9  
4 5 7  
6 5 7  
8 3 9  
4 3 6  
7 2 0  
3 5 5  

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\(^{nd}\) digit

7 2 0  
3 5 5  
4 3 6  
4 5 7  
6 5 7  
3 2 9  
8 3 9  

3 2 0  
3 2 9  
4 3 6  
8 3 9  
3 5 5  
4 5 7  
8 3 9  

**Step 3:** Sort 3\(^{rd}\) digit

7 2 0  
3 2 9  
4 3 6  
8 3 9  
3 5 5  
4 5 7  
6 5 7  
8 3 9  

3 2 9  
3 5 5  
4 3 6  
4 3 6  
3 5 5  
6 5 7  
7 2 0  
8 3 9
Correctness of Radix Sort (LSD-first)

**Proof by induction:**

*Base case:* \( d=1 \) 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 \( d \).

Two numbers that differ in digit \( d \) are correctly sorted (e.g. 355 and 657).

Two numbers equal in digit \( d \) are put in the same order as the input \( \Rightarrow \) correct order.
Radix Sort: Runtime

- Use counting-sort to sort each digit
  \textit{Reminder:} Counting sort complexity: $\Theta(n+k)$
  \begin{itemize}
    \item $n$: size of input array
    \item $k$: the range of the values
  \end{itemize}

- Radix sort runtime: $\Theta(d(n+k))$
  \begin{itemize}
    \item $d$: # of digits
  \end{itemize}

- How to choose the $d$ and $k$?
We have flexibility in choosing \(d\) and \(k\)

Assume we are trying to sort 32-bit words

- We can define each digit to be 4 bits
- Then, the range for each digit \(k = 2^4 = 16\)
  So, counting sort will take \(\Theta(n+16)\)
- The number of digits \(d = 32/4 = 8\)
- 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
  - Or, we can define each digit to be 8 bits
  - Then, the range for each digit \(k = 2^8 = 256\)
    - So, counting sort will take \(\Theta(n+256)\)
  - The number of digits \(d = 32/8 = 4\)
  - Radix sort runtime: \(\Theta(4(n+256)) = \Theta(n)\)
Radix Sort: Runtime

- Assume we are trying to sort \(b\)-bit words
  - Define each digit to be \(r\) bits
  - Then, the range for each digit \(k = 2^r\)
    - So, counting sort will take \(\Theta(n+2^r)\)
  - The number of digits \(d = b/r\)

Radix sort runtime:

\[
T(n, b) = \frac{b}{r} \left( n + 2^r \right)
\]
Radix Sort: Runtime Analysis

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

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

Or, intuitively:

We want to balance the terms \( \frac{b}{r} \) and \( n + 2^r \)

Choose \( r \approx \log n \)

If we choose \( r \ll \log n \) \( \Rightarrow \) (\( n + 2^r \)) term doesn’t improve

If we choose \( r \gg \log n \) \( \Rightarrow \) (\( n + 2^r \)) increases exponentially
Radix Sort: Runtime Analysis

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

Choose \( r = \lg n \)

\[ T(n, b) = \Theta(bn/\lg n) \]

For numbers in the range from 0 to \( n^d - 1 \), we have:

The number of bits \( b = \lg(n^d) = d \lg n \)

Radix sort runs in \( \Theta(dn) \)
Radix Sort: Conclusions

Choose \( r = \log n \) \[ T(n, b) = \Theta(bn/\log n) \]

- **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**: \( \log n = 20 \) 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.