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.
  - Insertion Sort: $O(n^2)$
  - Merge/Heap Sort: $O(n \log n)$
  - QuickSort: $O(n \log n) \text{ (expected)}$
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 \( j \leftarrow 2 \) 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

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. $i \leftarrow i - 1$;
7. $A[i+1] \leftarrow$ key;
endfor

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

already sorted


Reminder: Insertion Sort (from Lecture 1)

**Insertion-Sort** (A)

1. for \( j \leftarrow 2 \) to \( n \) do
2. \( \text{key} \leftarrow A[j]; \)
3. \( i \leftarrow j - 1; \)
4. while \( i > 0 \) and \( A[i] > \text{key} \) do
   5. \( A[i+1] \leftarrow A[i]; \)
   6. \( i \leftarrow i - 1; \)
endwhile
7. \( A[i+1] \leftarrow \text{key}; \)
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: $<a_1, a_2, a_3>$  $n=3$
Decision Tree for Insertion Sort and n=3

Leaf nodes: output of sorting algorithm

Internal Node \( \Theta_i: \Theta_j \)

Comparison operators: \( \leq, > \)

Example branches:
- \( <1, 2, 3> \) at node 1:2
- \( <2, 1, 3> \) at node 1:3
- \( <1, 3, 2> \) at node 2:3
- \( <3, 1, 2> \) at node 1:3
- \( <2, 3, 1> \) at node 2:3
- \( <3, 2, 1> \) at node 2:3
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
Decision Tree

\[ \text{height} = \text{worst-case running time} \]

\[ \theta_i = \theta_i \]

Motivation

Try to find a lower bound on the height
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:
  
  $2^h \geq n!$
Theorem: Any comparison sort algorithm requires \( \Omega(n \log n) \) comparisons in the worst case.

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

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

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

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

\[
= n \log n - n \log e
\]

\[
= \Omega(n \log n)
\]
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.
Sorting in Linear Time

**Counting sort**: No comparisons between elements

*Input*: A[1 .. n], where A[j] ∈ {1, 2, ..., k}

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

*Auxiliary storage*: C[1 .. k]  

Sort n integers where each integer is in the range [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
    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

C: Count array
Counting Sort

Step 1: Initialize all counts to 0

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
     0 0 0 0

j
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 2: Count the number of occurrences of each value in the input array

A: 4 1 3 4 3
B:
C: 1 0 2 2 4
Counting Sort

```plaintext
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 \leq 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
Stable Sorting Algorithms

A: \[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

B: \[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

C: \[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

Before prefix sum

C: \[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

After prefix sum

at the end

B: \[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

B array
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[3] = 3 elts that are ≤ 3

A: 4 1 3 4 3
B: [unfilled]
C: 1 1 2 5


prefix sum of C-array
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

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

- **A**: 4 1 3 4 3
- **B**: 1 2 3 4 5
- **C**: 1 1 1 4

$A(j) = 3$
**Counting Sort**

```plaintext
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 $C[1] = 1$ els that are $≤ 1$
Counting Sort

```plaintext
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 $C[4] = 4$ els that are $\leq 4$

```
A: 4 1 3 4 3
B: 1 3 3 4 5
C: 0 1 1 3
```

Step 1
Counting Sort: Runtime Analysis

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

\(\Theta(n)\)

\(\Theta(k)\)

Is this a linear-time algo?

\(\Theta(n)\)
Yes if \(k = \Theta(n)\)
\(\Theta(n) = \Theta(n+k) = \Theta(n+O(n)) = \Theta(n)\)

Total runtime: \(\Theta(n+k)\)

\(A = [5, 1, 10^6, 4]\)
\(\Theta(n+10^6) = \Theta(n+k)\)

\(n: \) 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 **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
```

**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
Single-digit sorting machine
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

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

| 457, 436, 657, 720, 839 | 3 2 9 |
| 3 5 5 |   |

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

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

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

\[
S(d) = \frac{1}{9}(10^d - 1)
\]
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

\[
P(d) = 9 \left(S(d) - 10^{d-1}\right) = 9 \left(\frac{10^d - 1}{9} - 10^{d-1}\right) = (10^d - 1 - 9 \cdot 10^{d-1}) = 10^{d-1} - 1 \quad P(d) = 10P(d-1) + 9
\]

**Alternative solution:** Solve the recurrence:

\[
P(d) = 10^{d-1} - 1
\]
Hollerith’s MSD-First Radix Sort

Example: To sort 3 digit numbers, in the worst case:

\[ S(d) = \frac{1}{9} (10^3 - 1) = 111 \text{ sorting passes needed} \]
\[ P(d) = 10^{d-1} - 1 \approx 99 \text{ intermediate card piles generated} \]

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

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Step 2:** Sort 2\(^{nd}\) digit

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

7 3-digit number
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 ➔ correct order

Last 2 digits sorted due to ind. hyp.
Radix Sort is a framework for performing multi-digit sorting using a ripple digit sort algorithm. The single-digit sorting algorithm used within Radix Sort (multi-digit sort) should be stable. This confirms the correctness of the Radix Sort. Usually, we use Counting Sort for single digit sort in multi-digit Radix Sort.
How do you define digit - simple -

For example: Some dates < Year, Month, Day >

3-digit keys

< Year, Month, Day >

\[
\begin{align*}
\overline{00-20} & \quad 1-12 & \quad 1-31 \\
\overline{2000-2020} & \quad k_3 = 21 & \quad k_2 = 12 & \quad k_1 = 31
\end{align*}
\]

Each day is an integer in the range \(1-31\)

\[
T(n) = \Theta(n + k_1) + \Theta(n + k_2) + \Theta(n + k_3) = 3n + k_1 + k_2 + k_3
\]
Radix Sort: Runtime

- Use counting-sort to sort each digit
  
  **Reminder**: Counting sort complexity: $\Theta(n+k)$
  
  - $n$: size of input array
  - $k$: the range of the values

- Radix sort runtime: $\Theta(d(n+k))$
  
  - $d$: # of digits

- How to choose the $d$ and $k$?
RADIX SORT framework

RadixSort(A, n, d)

| Sort an array of n d-digit numbers |

for i = 1 to d do

StableSort(A, n) on digit i

D Use counting sort as the stable sort algorithm

CountSort on digit i \( \Theta(n+k) \)

Overall Radix Sort runtime \( \Theta(d(n+k)) \)

How to choose \( d \) and \( k \)

\( n = \# \) of keys

\( d = \# \) of digits

\( k = \) range of each digit
Radix Sort: Runtime – Example 1

- 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 \( \Rightarrow k = 2^4 = 16 \)
  - Then, the range for each digit \( k = 2^4 = 16 \)
    So, counting sort will take \( \Theta(n+16) \) time on each of the 8 digits
- The number of digits \( d = \frac{32}{4} = 8 \)
- 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 $\Theta(n+256)$ time on each 8-bit digit
  - The number of digits $d = 32/8 = 4$
  - Radix sort runtime: $\Theta(4(n+256)) = \Theta(n)$

8 bits 8 bits 8 bits 8 bits

32-bit
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 $\Theta(n+2^r)$ time on each digit
  - The number of digits $d = \frac{b}{r}$

Radix sort runtime:

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

\[ T(n, b) = \left( \frac{b}{r} \right) \frac{\#}{\left( n + 2^r \right)} \cdot \frac{b}{r} = \# \text{ of digit} \]

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

Or, intuitively:

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

Choose \( r \approx \log n \)

Each Count Sort pass will take \( \Theta(n + 2^r) = \Theta(n^{1+ n}) \)

If we choose \( r << \log n \Rightarrow (n + 2^r) \text{ term doesn't improve} \)

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

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

Choose \( r = \log n \)

\[
T(n, b) = \Theta \left( \frac{bn}{\log n} \right)
\]

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

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

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

Example: Compare radix sort with merge sort/heapsort

1 million \((2^{20})\) 32-bit numbers \((n = 2^{20}, b = 32)\)

- Radix sort: \([32/20]\) = 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.