# CSE 221: Algorithms

## Sorting lower bounds and Linear time sorting

Mumit Khan

Computer Science and Engineering
BRAC University

**References**

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
2. Erik Demaine and Charles Leiserson, *6.046J Introduction to Algorithms*. MIT OpenCourseWare, Fall 2005. Available from: `ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/index.htm`

Last modified: October 27, 2009

# Contents

# Contents

# What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

# What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

# What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

# What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

**Question**

Can a sorting algorithm do better than $O(n \lg n)$ in the worst-case?

# What's the best we can do?

- Bubble, selection, insertion, quicksort . . . $O(n^2)$
- Heapsort, mergesort . . . $O(n \lg n)$

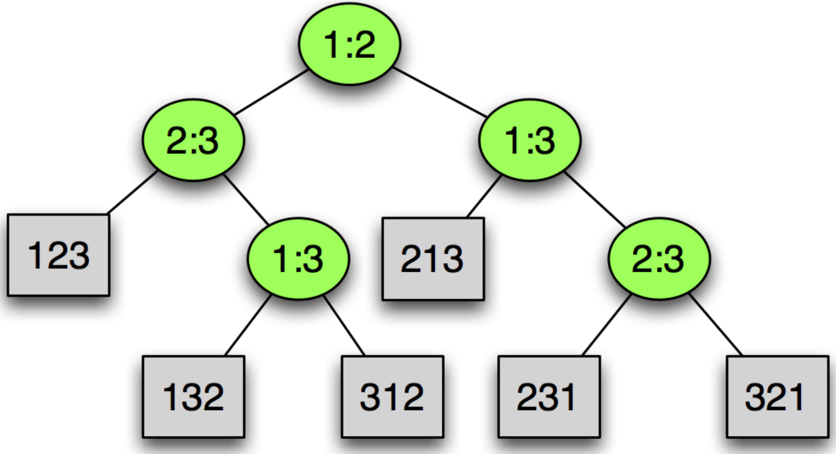
**Question**

Can a sorting algorithm do better than $O(n \lg n)$ in the worst-case?

We can use a decision tree to answer this question.

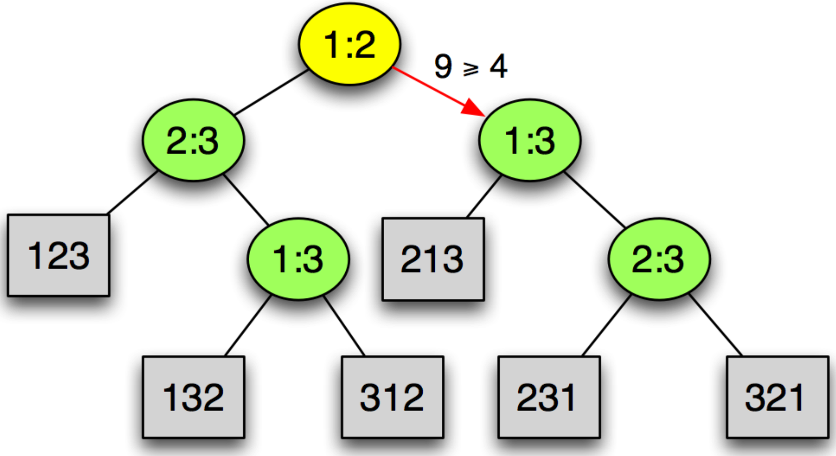
# Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



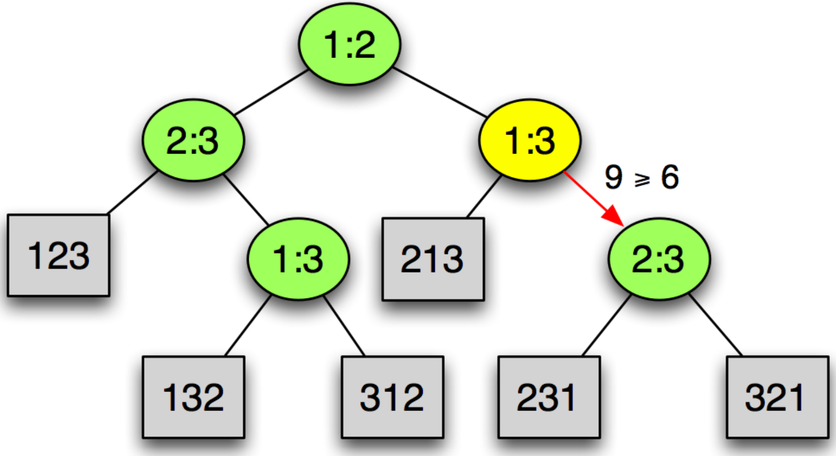
# Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



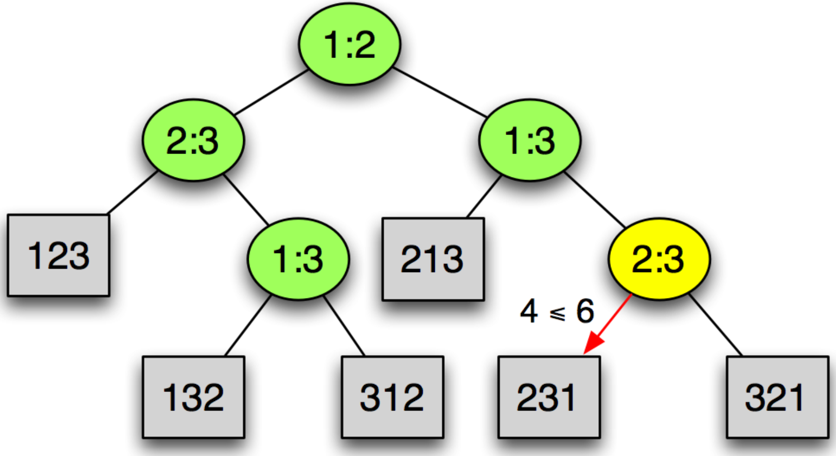
# Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



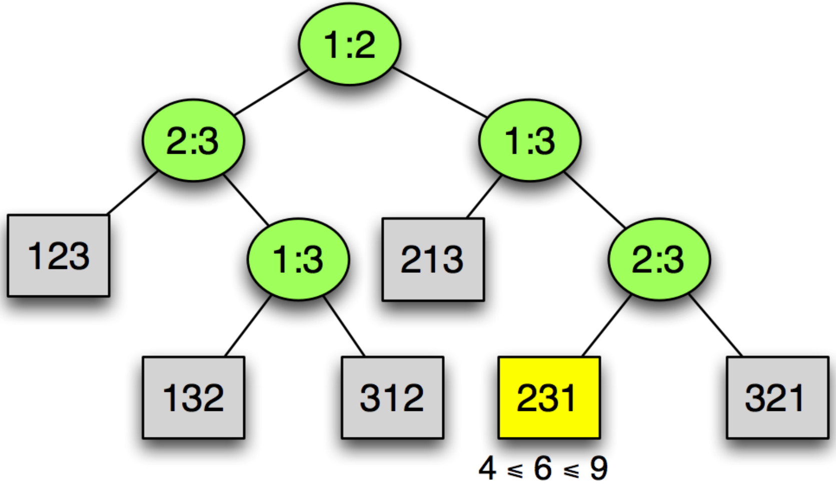
# Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$



# Decision tree for comparison based sorting

Sequence $A = \langle 9, 4, 6 \rangle$ $\Longrightarrow$ Sequence $B = \langle 4, 6, 9 \rangle$



# Lower bound on comparison based sorting

**Question**

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$\begin{aligned} h = \lceil \lg n! \rceil &\geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil \\ &= \Omega(n \lg n) \end{aligned}$$

# Lower bound on comparison based sorting

**Question**

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$
\begin{aligned}
h = \lceil \lg n! \rceil &\geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil \\
&= \Omega(n \lg n)
\end{aligned}
$$

# Lower bound on comparison based sorting

**Question**

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$h = \lceil \lg n! \rceil \geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil$$
$$= \Omega(n \lg n)$$

# Lower bound on comparison based sorting

**Question**

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$h = \lceil \lg n! \rceil \geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil$$
$$= \Omega(n \lg n)$$

# Lower bound on comparison based sorting

## Question

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$h = \lceil \lg n! \rceil \geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil$$
$$= \Omega(n \lg n)$$

# Lower bound on comparison based sorting

**Question**

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$h = \lceil \lg n! \rceil \geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil$$
$$= \Omega(n \lg n)$$

# Lower bound on comparison based sorting

**Question**

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$h = \lceil \lg n! \rceil \geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil$$
$$= \Omega(n \lg n)$$

# Lower bound on comparison based sorting

**Question**

What is the best that we can do with comparison-based sorting?

- $n!$ possible permutations, one of which is the sorted sequence.
- Minimum number of comparisons is the path from root of the decision tree to one of the $n!$ leaves.
- Height of a binary tree with $n!$ leaves is $\lceil \lg n! \rceil$. (Note: by Stirling's approximation $n! \geq (n/e)^n$, where $e$ is Euler's constant.)

$$h = \lceil \lg n! \rceil \geq \lceil \lg((n/e)^n) \rceil = \lceil n \lg n - n \lg e \rceil$$
$$= \Omega(n \lg n)$$

**Theorem**

*The worst-case asymptotic time complexity for any comparison-based sorting algorithm is* $\boxed{\Omega(n \lg n)}$.

# Contents

# Sorting in linear time: Counting sort

**Counting sort**: No comparisons between elements.

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

# Sorting in linear time: Counting sort

**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 algorithm

$\textbf{for } i \leftarrow 1 \textbf{ to } k$

$\quad \textbf{do } C[i] \leftarrow 0$

$\textbf{for } j \leftarrow 1 \textbf{ to } n$

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

$\textbf{for } i \leftarrow 2 \textbf{ to } k$

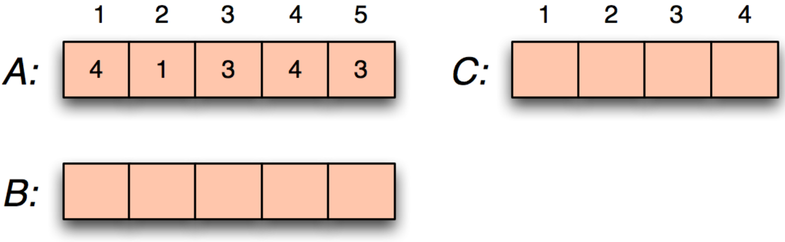
$\quad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \qquad \triangleright\ C[i] = |\{key \le i\}|$

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

$\quad \textbf{do } B[C[A[j]]] \leftarrow A[j]$

$\quad \phantom{\textbf{do }} C[A[j]] \leftarrow C[A[j]] - 1$

# Counting sort example



# Counting sort example: loop 1

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| *A*: | 4 | 1 | 3 | 4 | 3 |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| *C*: | 0 | 0 | 0 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| *B*: | | | | | |

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

# Counting sort example: loop 2

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| A: | 4 | 1 | 3 | 4 | 3 |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| C: | 0 | 0 | 0 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| B: | | | | | |

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

# Counting sort example: loop 2

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| *A:* | 4 | 1 | 3 | 4 | 3 |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| *C:* | 1 | 0 | 0 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| *B:* | | | | | |

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

# Counting sort example: loop 2

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| A: | 4 | 1 | 3 | 4 | 3 |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| C: | 1 | 0 | 1 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| B: | | | | | |

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

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

# Counting sort example: loop 2

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| *A:* | 4 | 1 | 3 | 4 | 3 |

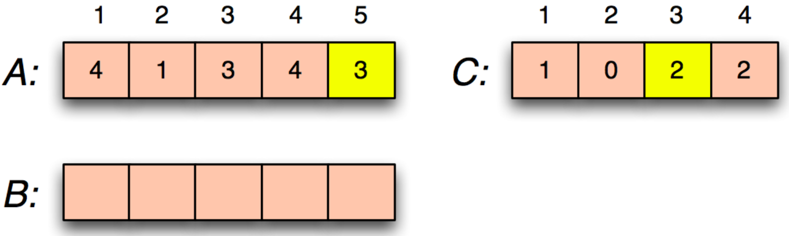
| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| *C:* | 1 | 0 | 1 | 2 |

| | | | | | |
|---|---|---|---|---|---|
| *B:* | | | | | |

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

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

# Counting sort example: loop 2




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

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

# Counting sort example: loop 3

A:

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 4 | 1 | 3 | 4 | 3 |

B:

| | | | | |
|---|---|---|---|---|
| | | | | |

C:

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 1 | 0 | 2 | 2 |

C′:

| | | | |
|---|---|---|---|
| 1 | 1 | 2 | 2 |

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

# Counting sort example: loop 3

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

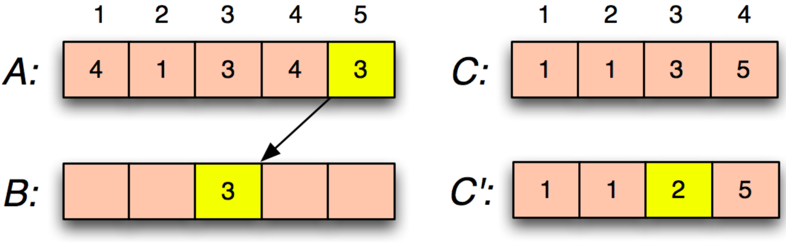
# Counting sort example: loop 3

| | 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 |

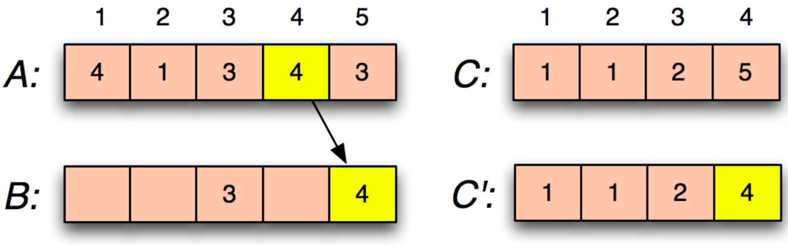
**for** $i \leftarrow 2$ **to** $k$
  **do** $C[i] \leftarrow C[i] + C[i-1]$   $\triangleright\ C[i] = |\{key \leq i\}|$

# Counting sort example: loop 4



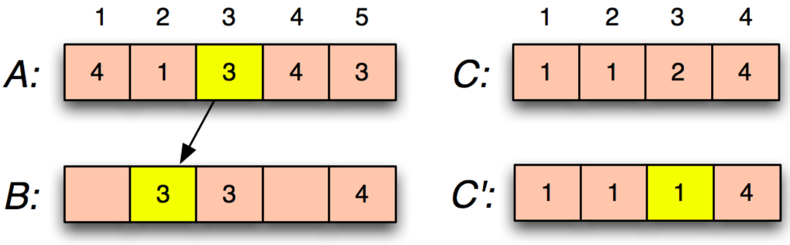

**for** $j \leftarrow n$ **downto** $1$
  **do** $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

# Counting sort example: loop 4



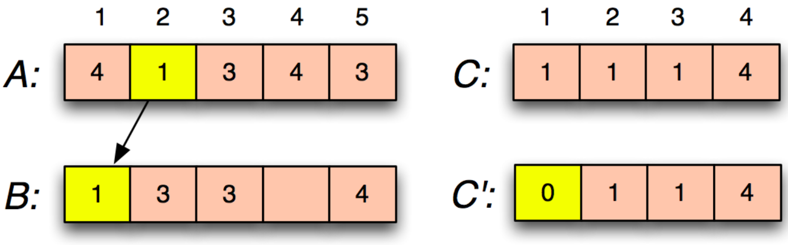

**for** $j \leftarrow n$ **downto** 1
**do** $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

# Counting sort example: loop 4



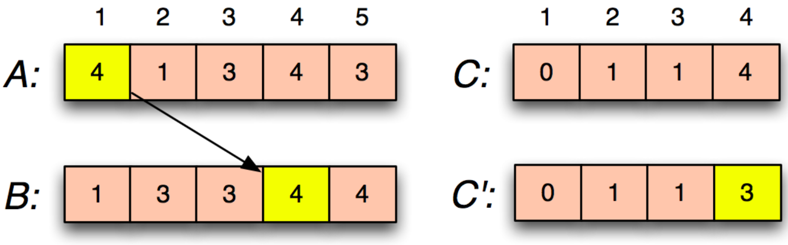

**for** $j \leftarrow n$ **downto** 1
**do** $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

# Counting sort example: loop 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 4




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

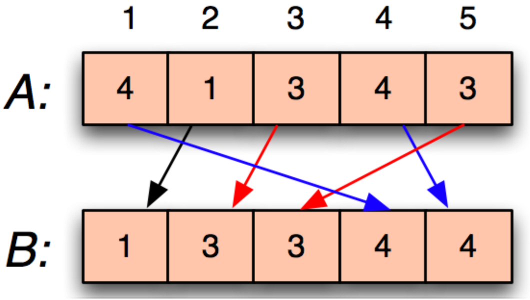
  

# Sort stability

Counting sort is stable, ie., it preserves the relative order of "equal" elements in the input.

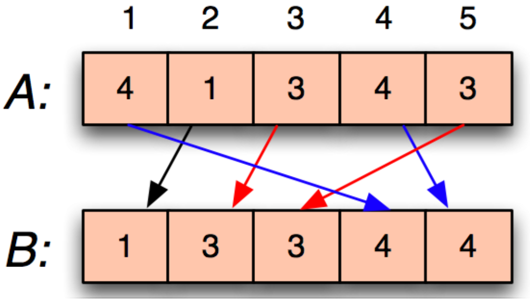
# Sort stability

Counting sort is stable, ie., it preserves the relative order of "equal" elements in the input.



# Sort stability

Counting sort is stable, ie., it preserves the relative order of "equal" elements in the input.

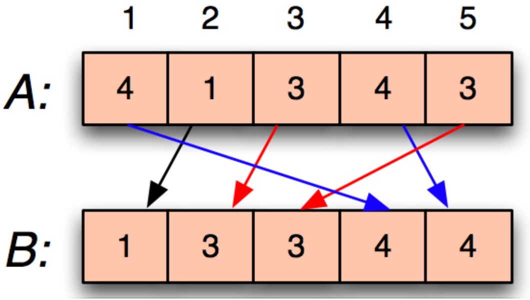



## Questions

- Would it still be stable if we had used **for** $j \leftarrow 1$ **to** $n$ instead of **for** $j \leftarrow n$ **downto** $1$?

# Sort stability

Counting sort is stable, ie., it preserves the relative order of "equal" elements in the input.




## Questions

- Would it still be stable if we had used **for** $j \leftarrow 1$ **to** $n$ instead of **for** $j \leftarrow n$ **downto** $1$?
- What other sort algorithms that you've seen so far are stable?

# Counting sort complexity

$$
\begin{aligned}
&\textbf{for } i \leftarrow 1 \textbf{ to } k \\
&\qquad \textbf{do } C[i] \leftarrow 0 \\
&\textbf{for } j \leftarrow 1 \textbf{ to } n \\
&\qquad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
&\textbf{for } i \leftarrow 2 \textbf{ to } k \\
&\qquad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \\
&\textbf{for } j \leftarrow n \textbf{ downto } 1 \\
&\qquad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\
&\qquad\quad\;\; C[A[j]] \leftarrow C[A[j]] - 1
\end{aligned}
$$

# Counting sort complexity

$$
\Theta(k) \quad \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow 0 \end{array} \right.
$$

$$
\begin{array}{l}
\textbf{for } j \leftarrow 1 \textbf{ to } n \\
\qquad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
\textbf{for } i \leftarrow 2 \textbf{ to } k \\
\qquad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \\
\textbf{for } j \leftarrow n \textbf{ downto } 1 \\
\qquad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\
\qquad \quad\;\; C[A[j]] \leftarrow C[A[j]] - 1
\end{array}
$$

# Counting sort complexity

$$\Theta(k) \quad \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow 0 \end{array} \right.$$

$$\Theta(n) \quad \left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \qquad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right.$$

$$\begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \\ \textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \qquad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\ \qquad \quad\;\; C[A[j]] \leftarrow C[A[j]] - 1 \end{array}$$

# Counting sort complexity

$$\Theta(k) \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow 0 \end{array} \right.$$

$$\Theta(n) \left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \qquad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right.$$

$$\Theta(k) \left\{ \begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \end{array} \right.$$

$$\begin{array}{l} \textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \qquad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\ \qquad\quad\;\; C[A[j]] \leftarrow C[A[j]] - 1 \end{array}$$

# Counting sort complexity

$$\Theta(k) \quad \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow 0 \end{array} \right.$$

$$\Theta(n) \quad \left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \qquad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right.$$

$$\Theta(k) \quad \left\{ \begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \end{array} \right.$$

$$\Theta(n) \quad \left\{ \begin{array}{l} \textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \qquad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\ \qquad \phantom{\textbf{do }} C[A[j]] \leftarrow C[A[j]] - 1 \end{array} \right.$$

# Counting sort complexity

$$
\begin{array}{ll}
\Theta(k) & \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow 0 \end{array} \right. \\
\Theta(n) & \left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \qquad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right. \\
\Theta(k) & \left\{ \begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } k \\ \qquad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \end{array} \right. \\
\Theta(n) & \left\{ \begin{array}{l} \textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \qquad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\ \qquad\quad\; C[A[j]] \leftarrow C[A[j]] - 1 \end{array} \right. \\
\hline
\Theta(n+k) &
\end{array}
$$

# Running time of Counting sort

The worst-case running time of Counting sort is $O(n + k)$.

# Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

## Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.

# Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

## Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?

# Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

## Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

# Running time of Counting sort

The worst-case running time of Counting sort is $O(n+k)$.

## Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

## And the answer is ...

- The $\Omega(n \lg n)$ is for comparison sorting.

# Running time of Counting sort

The worst-case running time of Counting sort is $O(n + k)$.

## Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

## And the answer is ...

- The $\Omega(n \lg n)$ is for comparison sorting.
- Counting sort is **not** a comparison sort.

# Running time of Counting sort

The worst-case running time of Counting sort is $O(n + k)$.

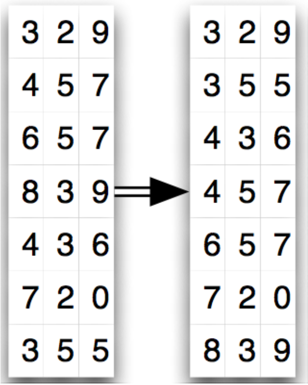
## Observations

- If $k = O(n)$, then the worst case running time is $\Theta(n)$.
- But didn't we just prove that sorting takes $\Omega(n \lg n)$ time?
- So what's wrong with this picture?

## And the answer is ...

- The $\Omega(n \lg n)$ is for comparison sorting.
- Counting sort is **not** a comparison sort.
- In fact, counting sort does not use a single comparison.

# Radix sort




## Radix sort basics

- Digit by digit sort.
- Can be either *most-significant* digit first, or *least-significant* digit first.
- A good way is to stably sort *least-significant* digit first.

# Radix sort in action

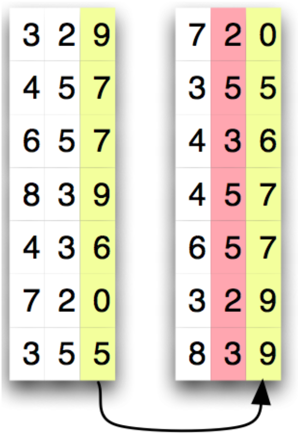
| | | |
|---|---|---|
| 3 | 2 | 9 |
| 4 | 5 | 7 |
| 6 | 5 | 7 |
| 8 | 3 | 9 |
| 4 | 3 | 6 |
| 7 | 2 | 0 |
| 3 | 5 | 5 |

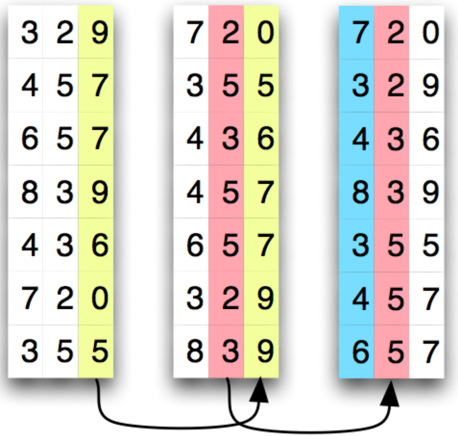
# Radix sort in action

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

# Radix sort in action

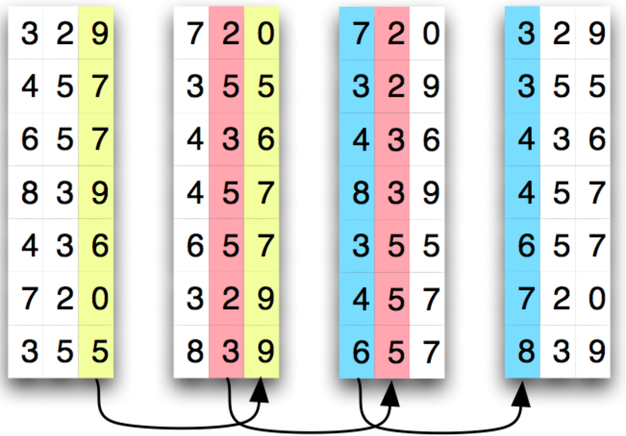


# Radix sort in action

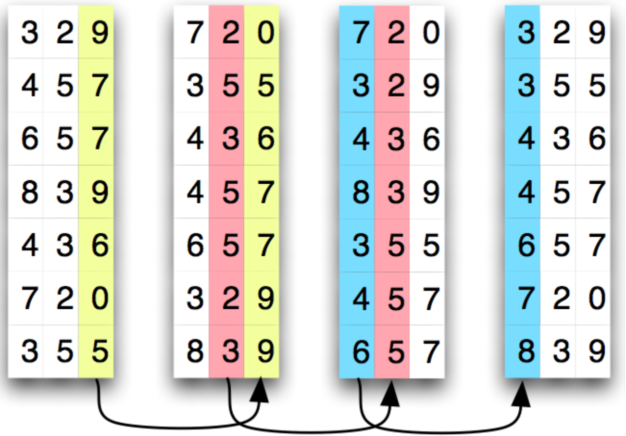


# Radix sort in action



# Radix sort in action




Analysis: For numbers in the range $[0 \,..\, n^d - 1]$, radix sort runs in $\Theta(dn)$ time.

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \,..\, 2^d - 1]$ range.

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{..} 2^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{.\,.} 2^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \,..\, 2^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.

## Questions to ask (and remember)

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{..} 2^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.

## Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{..} 2^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.

## Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?
- Are there sorting algorithms that beat this lower bound?

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \,..\, 2^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.

## Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?
- Are there sorting algorithms that beat this lower bound?
- How can you beat a proven lower bound?

# Conclusion

- Linear-time sorting algorithms beat the $\Omega(n \lg n)$ lower bound of comparison-based sorts by *not* doing any element comparison.
- Counting sort is a $\Theta(n)$ time algorithm if $k = O(n)$, and it is stable.
- Radix sort is a $\Theta(dn)$ algorithm for numbers in $[0 \mathinner{..} 2^d - 1]$ range.
- Radix sort is an excellent algorithm is trivial to implement, and works well for large inputs.
- Radix sort often uses counting sort as the stable auxiliary sorting routine.

## Questions to ask (and remember)

- What is the lower bound of comparison-based sorting algorithms?
- Are there sorting algorithms that beat this lower bound?
- How can you beat a proven lower bound?
- Why do we recommend sorting *least-significant* digits first in radix sort?