Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$.
Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$

**Can we sort faster?**

For example in $O(n \log \log n)$ time? Or even $O(n)$ time? Yes! we can sometimes sort faster. But in general, no, we cannot.

Example: Sort an array of length $n$ of bits, i.e., every array element is either 0 or 1, in time $O(n)$?

Count number of 0s $n_0$ Write $n_0$ 0s followed by $n - n_0$ 1s

Both operations take time $O(n)$
Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$

**Can we sort faster?**

- For example in $O(n \log \log n)$ time?

---

Dr. Christian Konrad
12: LB for Sorting, Countingsort, Radixsort
Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$

**Can we sort faster?**
- For example in $O(n \log \log n)$ time?
- Or even $O(n)$ time?

Yes! we can sometimes sort faster
But in general, no, we cannot

Example: Sort an array of length $n$ of bits, i.e., every array element is either 0 or 1, in time $O(n)$?

Count number of 0s $n_0$
Write $n_0$ 0s followed by $n - n_0$ 1s
Both operations take time $O(n)$
Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$

**Can we sort faster?**

- For example in $O(n \log \log n)$ time?
- Or even $O(n)$ time?

**Yes!** we can sometimes sort faster
Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$

**Can we sort faster?**
- For example in $O(n \log \log n)$ time?
- Or even $O(n)$ time?

**Yes!** we can sometimes sort faster
But in general, **no**, we cannot
Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$

**Can we sort faster?**
- For example in $O(n \log \log n)$ time?
- Or even $O(n)$ time?

Yes! we can sometimes sort faster
But in general, no, we cannot

**Example:** Sort an array of length $n$ of bits, i.e., every array element is either 0 or 1, in time $O(n)$?
Can we sort faster than $O(n \log n)$ time?

**Recall:** Fastest runtime of any sorting algorithm seen is $O(n \log n)$

**Can we sort faster?**
- For example in $O(n \log \log n)$ time?
- Or even $O(n)$ time?

**Yes!** we can sometimes sort faster
But in general, **no**, we cannot

**Example:** Sort an array of length $n$ of bits, i.e., every array element is either 0 or 1, in time $O(n)$?
- Count number of 0s $n_0$
- Write $n_0$ 0s followed by $n - n_0$ 1s
- Both operations take time $O(n)$
Comparison-based Sorting

Order is determined solely by comparing input elements. All information we obtain is by asking "Is $A[i] \leq A[j]$?", for some $i, j$, in particular, we may not inspect the elements.

Quicksort, mergesort, insertionsort, heapsort are comparison-based sorting algorithms.

Lower Bound for Comparison-based Sorting

We will prove that every comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons. This implies that $O(n \log n)$ is an optimal runtime for comparison-based sorting.
Comparison-based Sorting

Order is determined solely by comparing input elements
Comparison-based Sorting

Order is determined solely by comparing input elements.

All information we obtain is by asking “Is $A[i] \leq A[j]$?”, for some $i, j$, in particular, we may not inspect the elements.
Comparison-based Sorting

- Order is determined solely by comparing input elements
- All information we obtain is by asking “Is $A[i] \leq A[j]$?”, for some $i, j$, in particular, we may not inspect the elements
- Quicksort, mergesort, insertionsort, heapsort are comparison-based sorting algorithms
Comparison-based Sorting

- Order is determined solely by comparing input elements.
- All information we obtain is by asking “Is $A[i] \leq A[j]$?” for some $i, j$, in particular, we may not inspect the elements.
- Quicksort, mergesort, insertion sort, heapsort are comparison-based sorting algorithms.

Lower Bound for Comparison-based Sorting
Comparison-based Sorting

- Order is determined solely by comparing input elements
- All information we obtain is by asking “Is \( A[i] \leq A[j] \)?”, for some \( i, j \), in particular, we may not inspect the elements
- Quicksort, mergesort, insertionsort, heapsort are comparison-based sorting algorithms

Lower Bound for Comparison-based Sorting

- We will prove that every comparison-based sorting algorithm requires \( \Omega(n \log n) \) comparisons
Comparison-based Sorting

Order is determined solely by comparing input elements

All information we obtain is by asking “Is $A[i] \leq A[j]$?” for some $i, j$, in particular, we may not inspect the elements

Quicksort, mergesort, insertionsort, heapsort are comparison-based sorting algorithms

Lower Bound for Comparison-based Sorting

We will prove that every comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons

This implies that $O(n \log n)$ is an optimal runtime for comparison-based sorting
Lower Bound for Comparison-based Sorting

Problem

A: array of length \( n \), all elements are different

We are only allowed to ask: Is \( A[i] < A[j] \), for any \( i, j \in [n] \)

How many questions are needed until we can determine the order of all elements?

Permutations

A bijective function \( \pi : [n] \rightarrow [n] \) is called a permutation

\[
\begin{align*}
\pi(1) &= 3 \\
\pi(2) &= 2 \\
\pi(3) &= 4 \\
\pi(4) &= 1
\end{align*}
\]
Problem

- $A$: array of length $n$, all elements are different
Problem

- $A$: array of length $n$, all elements are different
- We are only allowed to ask: Is $A[i] < A[j]$, for any $i, j \in [n]$
Problem

- $A$: array of length $n$, all elements are different
- We are only allowed to ask: Is $A[i] < A[j]$, for any $i, j \in [n]$
- How many questions are needed until we can determine the order of all elements?
Problem

- $A$: array of length $n$, all elements are different
- We are only allowed to ask: Is $A[i] < A[j]$, for any $i, j \in [n]$
- How many questions are needed until we can determine the order of all elements?

Permutations
Problem
- $A$: array of length $n$, all elements are different
- We are only allowed to ask: Is $A[i] < A[j]$, for any $i, j \in [n]$
- How many questions are needed until we can determine the order of all elements?

Permutations
- A \textit{bijective} function $\pi : [n] \rightarrow [n]$ is called a permutation
Problem

- \( A \) : array of length \( n \), all elements are different
- We are only allowed to ask: Is \( A[i] < A[j] \), for any \( i, j \in [n] \)
- How many questions are needed until we can determine the order of all elements?

Permutations

- A bijective function \( \pi : [n] \rightarrow [n] \) is called a permutation

\[
\begin{align*}
\pi(1) & = 3 \\
\pi(2) & = 2 \\
\pi(3) & = 4 \\
\pi(4) & = 1
\end{align*}
\]
Problem

- $A$ : array of length $n$, all elements are different
- We are only allowed to ask: Is $A[i] < A[j]$, for any $i, j \in [n]$
- How many questions are needed until we can determine the order of all elements?

Permutations

- A *bijective* function $\pi : [n] \rightarrow [n]$ is called a permutation

\[
\begin{align*}
\pi(1) &= 3 \\
\pi(2) &= 2 \\
\pi(3) &= 4 \\
\pi(4) &= 1
\end{align*}
\]

- A reordering of $[n]$
How many permutations are there?

Lemma $|\Pi| = n! = n \cdot (n-1) \cdot 3 \cdot 2 \cdot 1$

Proof. The first element can be mapped to $n$ potential elements. The second can only be mapped to $(n-1)$ elements. etc.
How many permutations are there?
Let $\Pi$ be the set of all permutations on $n$ elements

\textbf{Lemma} \quad |\Pi| = n! = n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1

\textbf{Proof.} The first element can be mapped to $n$ potential elements. The second can only be mapped to $(n-1)$ elements. etc.

Rephrasing our Task:
Find permutation $\pi \in \Pi$ such that:
$A[\pi(1)] < A[\pi(2)] < \cdots < A[\pi(n-1)] < A[\pi(n)]$
How many permutations are there?
Let $\Pi$ be the set of all permutations on $n$ elements

**Lemma**

$$|\Pi| = n! = n \cdot (n-1) \ldots 3 \cdot 2 \cdot 1$$
How many permutations are there?
Let Π be the set of all permutations on \( n \) elements

**Lemma**
\[ |\Pi| = n! = n \cdot (n - 1) \cdots 3 \cdot 2 \cdot 1 \]

**Proof.**
How many permutations are there?
Let $\Pi$ be the set of all permutations on $n$ elements

**Lemma**

$|\Pi| = n! = n \cdot (n - 1) \ldots 3 \cdot 2 \cdot 1$

**Proof.** The first element can be mapped to $n$ potential elements.
How many permutations are there?
Let $\Pi$ be the set of all permutations on $n$ elements

Lemma
\[ |\Pi| = n! = n \cdot (n - 1) \cdots 3 \cdot 2 \cdot 1 \]

Proof. The first element can be mapped to $n$ potential elements. The second can only be mapped to $(n - 1)$ elements.
How many permutations are there?
Let \( \Pi \) be the set of all permutations on \( n \) elements

**Lemma**

\[ |\Pi| = n! = n \cdot (n-1) \ldots 3 \cdot 2 \cdot 1 \]

**Proof.** The first element can be mapped to \( n \) potential elements. The second can only be mapped to \( (n-1) \) elements. etc.
How many permutations are there?
Let \( \Pi \) be the set of all permutations on \( n \) elements

**Lemma**

\[
|\Pi| = n! = n \cdot (n - 1) \ldots 3 \cdot 2 \cdot 1
\]

**Proof.** The first element can be mapped to \( n \) potential elements. The second can only be mapped to \( (n - 1) \) elements. etc.

Rephrasing our Task:
How many permutations are there?
Let $\Pi$ be the set of all permutations on $n$ elements

**Lemma**

$$|\Pi| = n! = n \cdot (n - 1) \cdots 3 \cdot 2 \cdot 1$$

**Proof.** The first element can be mapped to $n$ potential elements. The second can only be mapped to $(n - 1)$ elements. etc.

**Rephrasing our Task:** Find permutation $\pi \in \Pi$ such that:

$$A[\pi(1)] < A[\pi(2)] < \cdots < A[\pi(n - 1)] < A[\pi(n)]$$
Example:

Sort 3 elements by asking queries:

$A_i < A_j$, for $i, j \in [3]$.

How many Queries are needed? (worst case)

Lemma

At least 3 queries are needed to sort 3 elements.

Proof.

Let the three elements be $a, b, c$. Suppose that the first query is $a < b$ and suppose that the answer is yes. (if it is not then relabel the elements $a, b, c$). We are left with 3 scenarios:

1. $a < b < c$
2. $a < c < b$
3. $c < a < b$

Next we either ask $a < c$ or $b < c$.

Suppose that we ask $a < c$. Then, if the answer is yes then we are left with cases 1 and 2 and we need an additional query.

Suppose that we ask $b < c$. Then, if the answer is no then we are left with cases 2 and 3 and we need an additional query.
Example:
Example:

How many Queries are needed? (worst case)
Example:

How many Queries are needed? (worst case)

Lemma

At least 3 queries are needed to sort 3 elements.
Example:

How many Queries are needed? (worst case)

**Lemma**

At least 3 queries are needed to sort 3 elements.

**Proof.**
Decision-tree Model

Example:

How many Queries are needed? (worst case)

Lemma
At least 3 queries are needed to sort 3 elements.

Proof. Let the three elements be $a, b, c$. 
Example:
Sort 3 elements by asking queries: $A[i] < A[j]$, for $i, j \in [3]\,$

**How many Queries are needed?** (worst case)

**Lemma**

At least 3 queries are needed to sort 3 elements.

**Proof.** Let the three elements be $a, b, c$. Suppose that the first query is $a < b$ and suppose that the answer is yes.
Example:

How many Queries are needed? (worst case)

Lemma

At least 3 queries are needed to sort 3 elements.

Proof. Let the three elements be $a, b, c$. Suppose that the first query is $a < b$ and suppose that the answer is yes. (if it is not then relabel the elements $a, b, c$).
Example:
Sort 3 elements by asking queries: \( A[i] < A[j], \) for \( i, j \in [3] \)

**How many Queries are needed?** (worst case)

**Lemma**

*At least 3 queries are needed to sort 3 elements.*

**Proof.** Let the three elements be \( a, b, c. \) Suppose that the first query is \( a < b \) and suppose that the answer is yes. (if it is not then relabel the elements \( a, b, c \)). We are left with 3 scenarios:
Example:

How many Queries are needed? (worst case)

Lemma

At least 3 queries are needed to sort 3 elements.

Proof. Let the three elements be $a, b, c$. Suppose that the first query is $a < b$ and suppose that the answer is yes. (if it is not then relabel the elements $a, b, c$). We are left with 3 scenarios:

1. $a < b < c$
2. $a < c < b$
3. $c < a < b$
Example:

How many Queries are needed? (worst case)

Lemma

At least 3 queries are needed to sort 3 elements.

Proof. Let the three elements be $a, b, c$. Suppose that the first query is $a < b$ and suppose that the answer is yes. (if it is not then relabel the elements $a, b, c$). We are left with 3 scenarios:

1. $a < b < c$
2. $a < c < b$
3. $c < a < b$

Next we either ask $a < c$ or $b < c$.
Example:
Sort 3 elements by asking queries: \( A[i] < A[j], \) for \( i, j \in [3] \)

How many Queries are needed? (worst case)

Lemma
At least 3 queries are needed to sort 3 elements.

Proof. Let the three elements be \( a, b, c. \) Suppose that the first query is \( a < b \) and suppose that the answer is yes. (if it is not then relabel the elements \( a, b, c \)). We are left with 3 scenarios:

1. \( a < b < c \)     2. \( a < c < b \)     3. \( c < a < b \)

Next we either ask \( a < c \) or \( b < c. \) Suppose that we ask \( a < c. \)
Example:

How many Queries are needed? (worst case)

Lemma

At least 3 queries are needed to sort 3 elements.

Proof. Let the three elements be \( a, b, c \). Suppose that the first query is \( a < b \) and suppose that the answer is yes. (if it is not then relabel the elements \( a, b, c \)). We are left with 3 scenarios:

1. \( a < b < c \)
2. \( a < c < b \)
3. \( c < a < b \)

Next we either ask \( a < c \) or \( b < c \). Suppose that we ask \( a < c \). Then, if the answer is yes then we are left with cases 1 and 2 and we need an additional query.
Example:

How many Queries are needed? (worst case)

Lemma

At least 3 queries are needed to sort 3 elements.

Proof. Let the three elements be $a, b, c$. Suppose that the first query is $a < b$ and suppose that the answer is yes. (if it is not then relabel the elements $a, b, c$). We are left with 3 scenarios:

1. $a < b < c$
2. $a < c < b$
3. $c < a < b$

Next we either ask $a < c$ or $b < c$. Suppose that we ask $a < c$. Then, if the answer is yes then we are left with cases 1 and 2 and we need an additional query. Suppose that we ask $b < c$. 
**Example:**

**How many Queries are needed?** (worst case)

**Lemma**

*At least 3 queries are needed to sort 3 elements.*

**Proof.** Let the three elements be $a, b, c$. Suppose that the first query is $a < b$ and suppose that the answer is yes. (if it is not then relabel the elements $a, b, c$). We are left with 3 scenarios:

1. $a < b < c$
2. $a < c < b$
3. $c < a < b$

Next we either ask $a < c$ or $b < c$. Suppose that we ask $a < c$. Then, if the answer is yes then we are left with cases 1 and 2 and we need an additional query. Suppose that we ask $b < c$. Then, if the answer is no then we are left with cases 2 and 3 and we need an additional query.
Every Guessing Strategy is a Decision-tree
Every Guessing Strategy is a Decision-tree

Observe:
- Every leaf is a permutation
Every Guessing Strategy is a Decision-tree

Observe:
- Every leaf is a permutation
- An execution is a root-to-leaf path
Lemma

Any comparison-based sorting algorithm requires \( \Omega(n \log n) \) comparisons.

Proof

Observe that decision-tree is a binary tree. Every potential permutation is a leaf. There are \( n! \) leaves. A binary tree of height \( h \) has no more than \( 2^h \) leaves. Hence:

\[
2^h \geq n!
\]

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

\( \geq \Omega(n \log n) \).

Comment:
Stirling's approximation for \( \log(n!) \) can be used for proving \( \log(n!) = \Omega(n \log n) \).
Lemma

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.
Lemma

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof
Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof Observe that decision-tree is a binary tree.
Lemma

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof Observe that decision-tree is a binary tree. Every potential permutation is a leaf.
Lemma

*Any comparison-based sorting algorithm requires* $\Omega(n \log n)$ *comparisons.*

**Proof** Observe that decision-tree is a binary tree. Every potential permutation is a leaf. There are $n!$ leaves.
Lemma

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

**Proof** Observe that decision-tree is a binary tree. Every potential permutation is a leaf. There are $n!$ leaves. A binary tree of height $h$ has no more than $2^h$ leaves.
Lemma

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof  Observe that decision-tree is a binary tree. Every potential permutation is a leaf. There are $n!$ leaves. A binary tree of height $h$ has no more than $2^h$ leaves. Hence:

$$2^h \geq n!$$
Lemma

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof Observe that decision-tree is a binary tree. Every potential permutation is a leaf. There are $n!$ leaves. A binary tree of height $h$ has no more than $2^h$ leaves. Hence:

\[
2^h \geq n!
\]

\[
h \geq \log(n!)
\]
Lemma

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof Observe that decision-tree is a binary tree. Every potential permutation is a leaf. There are $n!$ leaves. A binary tree of height $h$ has no more than $2^h$ leaves. Hence:

$$
2^h \geq n!
$$

$$
h \geq \log(n!) = \Omega(n \log n)
$$

Comment: Stirling’s approximation for $n!$ can be used for proving $\log(n!) = \Omega(n \log n)$
Counting Sort: Sorting Integers fast

Counting Sort

Input is an array $A$ of integers from \{0, 1, 2, ..., $k$\}, for some integer $k$.

Idea

For each element $x$, count number of elements $< x$.

Put $x$ directly into its position.

Difficulty:

Multiple elements have the same value.
Counting Sort
Input is an array $A$ of integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$

Idea
Counting Sort
Input is an array $A$ of integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$

Idea

- For each element $x$, count number of elements $< x$
Counting Sort: Sorting Integers fast

Counting Sort
Input is an array $A$ of integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$

Idea
- For each element $x$, count number of elements $< x$
- Put $x$ directly into its position
Counting Sort: Sorting Integers fast

Counting Sort
Input is an array $A$ of integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$

Idea
- For each element $x$, count number of elements $< x$
- Put $x$ directly into its position
- **Difficulty:** Multiple elements have the same value
**Algorithm**

**Require:** Array $A$ of $n$ integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$

Let $C[0 \ldots k]$ be a new array with all entries equal to 0

Store output in array $B[0 \ldots n-1]$

```plaintext
for $i = 0, \ldots, n-1$ do  
    {Count how often each element appears}
    $C[A[i]] \leftarrow C[A[i]] + 1$

for $i = 1, \ldots, k$ do  
    {Count how many smaller elements appear}
    $C[i] \leftarrow C[i] + C[i-1]$

for $i = n-1, \ldots, 0$ do
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$

return $B$
```
Require: Array $A$ of $n$ integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$
Let $C[0 \ldots k]$ be a new array with all entries equal to 0
Store output in array $B[0 \ldots n-1]$

for $i = 0, \ldots, n - 1$ do \{Count how often each element appears\}
    $C[A[i]] \leftarrow C[A[i]] + 1$

for $i = 1, \ldots, k$ do \{Count how many smaller elements appear\}
    $C[i] \leftarrow C[i] + C[i - 1]$

for $i = n - 1, \ldots, 0$ do
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$

return $B$

- Last loop processes $A$ from right to left
Algorithm

**Require:** Array $A$ of $n$ integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$

Let $C[0 \ldots k]$ be a new array with all entries equal to 0

Store output in array $B[0 \ldots n - 1]$

for $i = 0, \ldots, n - 1$ do {Count how often each element appears}

\[
C[A[i]] \leftarrow C[A[i]] + 1
\]

for $i = 1, \ldots, k$ do {Count how many smaller elements appear}

\[
C[i] \leftarrow C[i] + C[i - 1]
\]

for $i = n - 1, \ldots, 0$ do

\[
B[C[A[i]] - 1] \leftarrow A[i]
\]

\[
C[A[i]] \leftarrow C[A[i]] - 1
\]

return $B$

- Last loop processes $A$ from right to left
- $C[A[i]]$: Number of *smaller* elements than $A[i]$
**Algorithm**

**Require:** Array $A$ of $n$ integers from $\{0, 1, 2, \ldots, k\}$, for some integer $k$

Let $C[0 \ldots k]$ be a new array with all entries equal to 0

Store output in array $B[0 \ldots n - 1]$

```plaintext
for $i = 0, \ldots, n - 1$ do {Count how often each element appears}
    $C[A[i]] \leftarrow C[A[i]] + 1$

for $i = 1, \ldots, k$ do {Count how many smaller elements appear}
    $C[i] \leftarrow C[i] + C[i - 1]$

for $i = n - 1, \ldots, 0$ do
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$

return $B$
```

- Last loop processes $A$ from right to left
- $C[A[i]]$: Number of *smaller* elements than $A[i]$
- Decrementing $C[A[i]]$: Next element of value $A[i]$ should be left of the current one

Dr. Christian Konrad

12: LB for Sorting, Countingsort, Radixsort
Example: $n = 8$, $k = 5$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
**Example:** \( n = 8, \ k = 5 \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
for i = n-1, \ldots, 0 do
    C[A[i]] ← C[A[i]] - 1
```
Counting Sort: Example

Example: $n = 8, k = 5$

\[
A = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
2 & 5 & 3 & 0 & 2 & 3 & 0 & 3
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 \\
2 & 0 & 2 & 3 & 0 & 1
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 \\
2 & 2 & 4 & 7 & 7 & 8
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{bmatrix}
\]

for $i = n - 1, \ldots, 0$ do

- $B[C[A[i]] - 1] \leftarrow A[i]$
- $C[A[i]] \leftarrow C[A[i]] - 1$
Example: $n = 8$, $k = 5$

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
C & 2 & 2 & 4 & 7 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
B & & & & & & & 3 \\
\end{array}
\]

\[
\text{for } i = n - 1, \ldots, 0 \text{ do} \\
B[C[A[i]] - 1] \leftarrow A[i] \\
C[A[i]] \leftarrow C[A[i]] - 1
\]
**Counting Sort: Example**

**Example:** \( n = 8, \ k = 5 \)

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
C & 2 & 2 & 4 & 6 & 7 & 8 \\
B & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 3 \\
\end{array} \]

\[
\text{for } i = n - 1, \ldots, 0 \text{ do }
\begin{align*}
C[A[i]] & \leftarrow C[A[i]] - 1
\end{align*}
\]
Counting Sort: Example

Example: $n = 8, k = 5$

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
B & 0 & & & & & 3 \\
C & 2 & 2 & 4 & 6 & 7 & 8 \\
\end{array}
\]

for $i = n - 1, \ldots, 0$ do

\[
\begin{align*}
B & [C[A[i]] - 1] \leftarrow A[i] \\
C & [A[i]] \leftarrow C[A[i]] - 1
\end{align*}
\]
Example: $n = 8$, $k = 5$

\begin{align*}
A & \begin{bmatrix}
2 & 5 & 3 & 0 & 2 & 3 & 0 & 3
\end{bmatrix} \\
C & \begin{bmatrix}
2 & 0 & 2 & 3 & 0 & 1
\end{bmatrix} \\
C & \begin{bmatrix}
1 & 2 & 4 & 6 & 7 & 8
\end{bmatrix}
\end{align*}

\begin{align*}
\text{for } i = n - 1, \ldots, 0 \text{ do} \\
C[A[i]] & \leftarrow C[A[i]] - 1
\end{align*}
Counting Sort: Example

Example: \( n = 8, \ k = 5 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
C & 1 & 2 & 4 & 6 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
B & 0 & 3 & 3 \\
\end{array}
\]

\[
\text{for } i = n - 1, \ldots, 0 \text{ do} \\
B[C[A[i]] - 1] \leftarrow A[i] \\
C[A[i]] \leftarrow C[A[i]] - 1 
\]
Counting Sort: Example

Example: $n = 8$, $k = 5$

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
C & 1 & 2 & 4 & 5 & 7 & 8 \\
B & 0 & & & & 3 & 3 & \\
\end{array}
\]

for $i = n - 1, \ldots, 0$ do

\[
\begin{align*}
C[A[i]] & \leftarrow C[A[i]] - 1
\end{align*}
\]
Example: \( n = 8, \ k = 5 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
C & 1 & 2 & 4 & 5 & 7 & 8 \\
B & 0 & 2 & 3 & 3 \\
\end{array}
\]

\[
\text{for } i = n - 1, \ldots, 0 \text{ do} \\
B[C[A[i]] - 1] \leftarrow A[i] \\
C[A[i]] \leftarrow C[A[i]] - 1
\]
## Counting Sort: Example

**Example:** \( n = 8, \ k = 5 \)

| \( A \) | \( 2 \) | \( 5 \) | \( 3 \) | \( 0 \) | \( 2 \) | \( 3 \) | \( 0 \) | \( 3 \) |
|---|---|---|---|---|---|---|---|

| \( C \) | \( 2 \) | \( 0 \) | \( 2 \) | \( 3 \) | \( 0 \) | \( 1 \) |
|---|---|---|---|---|---|

for \( i = n - 1, \ldots, 0 \) do

\[
B[C[A[i]] - 1] \leftarrow A[i] \\
C[A[i]] \leftarrow C[A[i]] - 1
\]

| \( B \) | \( 0 \) | \( 2 \) | \( 3 \) | \( 3 \) |
|---|---|---|---|

Dr. Christian Konrad
Counting Sort: Example

**Example:** \( n = 8, \ k = 5 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
C & 1 & 2 & 3 & 5 & 7 & 8 \\
B & 0 & 0 & 2 & 3 & 3 \\
\end{array}
\]

\[
\text{for } i = n - 1, \ldots, 0 \text{ do} \\
B[C[A[i]] - 1] \leftarrow A[i] \\
C[A[i]] \leftarrow C[A[i]] - 1
\]
Example: \( n = 8, \ k = 5 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
C & 0 & 2 & 3 & 5 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
B & 0 & 0 & 2 & 3 & 3 \\
\end{array}
\]

For \( i = n - 1, \ldots, 0 \) do
- \( B[C[A[i]] - 1] \leftarrow A[i] \)
- \( C[A[i]] \leftarrow C[A[i]] - 1 \)
**Example**: \( n = 8, \ k = 5 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
A & 2 & 5 & 3 & 0 & 2 & 3 & 0 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
C & 2 & 0 & 2 & 3 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
C & 0 & 2 & 2 & 4 & 7 & 7 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
B & 0 & 0 & 2 & 2 & 3 & 3 & 3 & 5 \\
\end{array}
\]

for \( i = n - 1, \ldots, 0 \)

\[
\begin{align*}
C[A[i]] & \leftarrow C[A[i]] - 1
\end{align*}
\]
Analysis: Counting Sort

Runtime:

```
for i = 0, . . . , n − 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, . . . , k do
    C[i] ← C[i] + C[i − 1]
for i = n − 1, . . . , 0 do
    C[A[i]] ← C[A[i]] − 1
```
Analysis: Counting Sort

Runtime:

\[ O(n) + O(k) + O(n) = O(n + k) \]

```
for i = 0, \ldots, n - 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, \ldots, k do
    C[i] ← C[i] + C[i - 1]
for i = n - 1, \ldots, 0 do
    C[A[i]] ← C[A[i]] - 1
```
Analysis: Counting Sort

Runtime:

\[ O(n) + O(k) + O(n) = O(n + k) \]

- Counting Sort has runtime \( O(n) \)
  if \( k = O(n) \)

```plaintext
for i = 0, \ldots, n - 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, \ldots, k do
    C[i] ← C[i] + C[i - 1]
for i = n - 1, \ldots, 0 do
    C[A[i]] ← C[A[i]] - 1
```

Runtime:
\[ O(n) + O(k) + O(n) = O(n + k) \]
Analysis: Counting Sort

Runtime:

\[ O(n) + O(k) + O(n) = O(n + k) \]

- Counting Sort has runtime \( O(n) \) if \( k = O(n) \)
- This beats the lower bound for comparison-based sorting

```plaintext
for i = 0, \ldots, n - 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, \ldots, k do
    C[i] ← C[i] + C[i - 1]
for i = n - 1, \ldots, 0 do
    C[A[i]] ← C[A[i]] - 1
```
Analysis: Counting Sort

Runtime:

\[ O(n) + O(k) + O(n) = O(n + k) \]

- Counting Sort has runtime \( O(n) \) if \( k = O(n) \)
- This beats the lower bound for comparison-based sorting

Stable? In-place?

\[
\begin{align*}
&\text{for } i = 0, \ldots, n - 1 \text{ do} \\
&\quad C[A[i]] \leftarrow C[A[i]] + 1 \\
&\text{for } i = 1, \ldots, k \text{ do} \\
&\quad C[i] \leftarrow C[i] + C[i - 1] \\
&\text{for } i = n - 1, \ldots, 0 \text{ do} \\
&\quad B[C[A[i]] - 1] \leftarrow A[i] \\
&\quad C[A[i]] \leftarrow C[A[i]] - 1
\end{align*}
\]
Analysis: Counting Sort

Runtime:

\[ O(n) + O(k) + O(n) = O(n + k) \]

- Counting Sort has runtime \( O(n) \) if \( k = O(n) \)
- This beats the lower bound for comparison-based sorting

Stable? In-place? Yes, it is stable (important!)

```
for i = 0, \ldots, n - 1 do
    C[A[i]] \leftarrow C[A[i]] + 1
for i = 1, \ldots, k do
    C[i] \leftarrow C[i] + C[i - 1]
for i = n - 1, \ldots, 0 do
    B[C[A[i]] - 1] \leftarrow A[i]
    C[A[i]] \leftarrow C[A[i]] - 1
```
Analysis: Counting Sort

Runtime:

\[ O(n) + O(k) + O(n) = O(n + k) \]

- Counting Sort has runtime \( O(n) \)
  if \( k = O(n) \)
- This beats the lower bound for comparison-based sorting

Stable? In-place? Yes, it is stable (important!) No, not in-place
Analysis: Counting Sort

Runtime:

\[ O(n) + O(k) + O(n) = O(n + k) \]

- Counting Sort has runtime \( O(n) \) if \( k = O(n) \)
- This beats the lower bound for comparison-based sorting

Stable? In-place? Yes, it is stable (important!) No, not in-place

Correctness Loop Invariant
Radix Sort

Input is an array $A$ of $d$ digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$.

Examples

$b = 2, d = 5$. E.g. 01101 (binary numbers)

$b = 10, d = 4$. E.g. 9714

Idea

Iterate through the $d$ digits
Sort according to the current digit
Radix Sort

Input is an array $A$ of $d$ digits integers, each digit is from the set
$\{0, 1, \ldots, b - 1\}$
Radix Sort
Input is an array \( A \) of \( d \) digits integers, each digit is from the set \( \{0, 1, \ldots, b-1\} \)

Examples
Radix Sort
Input is an array \( A \) of \( d \) digits integers, each digit is from the set \( \{0, 1, \ldots, b - 1\} \)

Examples
- \( b = 2, d = 5 \). E.g. 01101 (binary numbers)
Radix Sort
Input is an array $A$ of $d$ digits integers, each digit is from the set \{0, 1, \ldots, b - 1\}

Examples
- $b = 2, d = 5$. E.g. 01101 (binary numbers)
- $b = 10, d = 4$. E.g. 9714
Radix Sort

Input is an array $A$ of $d$ digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Examples

- $b = 2, d = 5$. E.g. 01101 (binary numbers)
- $b = 10, d = 4$. E.g. 9714

Idea
Radix Sort
Input is an array $A$ of $d$ digits integers, each digit is from the set \{0, 1, \ldots, b - 1\}

Examples
- $b = 2$, $d = 5$. E.g. 01101 (binary numbers)
- $b = 10$, $d = 4$. E.g. 9714

Idea
- Iterate through the $d$ digits
Radix Sort

Input is an array $A$ of $d$ digits integers, each digit is from the set \{0, 1, \ldots, b - 1\}

Examples

- $b = 2$, $d = 5$. E.g. 01101 (binary numbers)
- $b = 10$, $d = 4$. E.g. 9714

Idea

- Iterate through the $d$ digits
- Sort according to the current digit
Radix Sort Algorithm

for $i = 1, \ldots, d$ do

Use a stable sort algorithm to sort array $A$ on digit $i$ (least significant digit is digit 1)

Example

329 457 657 839 436 720 355

$\rightarrow$

72 0 35 5 43 6 45 7 65 7 32 9 83 9

$\rightarrow$

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

$\rightarrow$

3 29 3 55 4 36 4 57 6 57 7 20 8 39
Radix Sort Algorithm

\[
\text{for } i = 1, \ldots, d \text{ do }
\]

Use a stable sort algorithm to sort array \( A \) on digit \( i \)

(least significant digit is digit 1)
Radix Sort Algorithm

\[
\text{for } i = 1, \ldots, d \text{ do }
\]
Use a stable sort algorithm to sort array \(A\) on digit \(i\)

(least significant digit is digit 1)

Example
Radix Sort Algorithm

\[
\text{for } i = 1, \ldots, d \text{ do} \\
\quad \text{Use a stable sort algorithm to} \\
\quad \text{sort array } A \text{ on digit } i \\
\]

(least significant digit is digit 1)

Example

329
457
657
839
436
720
355
Radix Sort (2)

Radix Sort Algorithm

```plaintext
for i = 1, ..., d do
    Use a stable sort algorithm to sort array A on digit i

(least significant digit is digit 1)
```

Example

<table>
<thead>
<tr>
<th>329</th>
<th>720</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>355</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
</tr>
<tr>
<td>436</td>
<td>657</td>
</tr>
<tr>
<td>720</td>
<td>329</td>
</tr>
<tr>
<td>355</td>
<td>839</td>
</tr>
</tbody>
</table>
Radix Sort (2)

Radix Sort Algorithm

\[
\text{for } i = 1, \ldots, d \text{ do} \\
\text{Use a stable sort algorithm to sort array } A \text{ on digit } i \\
\text{(least significant digit is digit 1)}
\]

Example

<table>
<thead>
<tr>
<th>329</th>
<th>720</th>
<th>720</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
</tr>
<tr>
<td>436</td>
<td>657</td>
<td>355</td>
</tr>
<tr>
<td>720</td>
<td>329</td>
<td>457</td>
</tr>
<tr>
<td>355</td>
<td>839</td>
<td>657</td>
</tr>
</tbody>
</table>
Radix Sort Algorithm

\[
\text{for } i = 1, \ldots, d \text{ do }
\]
\[
\text{Use a stable sort algorithm to sort array } A \text{ on digit } i
\]

(least significant digit is digit 1)

Example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>329</td>
<td>720</td>
<td>720</td>
<td>329</td>
</tr>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
<td>355</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
<td>457</td>
</tr>
</tbody>
</table>

Dr. Christian Konrad

12: LB for Sorting, Countingsort, Radixsort
Analysis

Lemma
Given $n$ $d$-digit number in which each digit can take on up to $b$ possible values. Radix-sort correctly sorts these numbers in $O(d(n + b))$ time if the stable sort it uses takes $O(n + b)$ time.

Proof
Runtime is obvious. Correctness follows by induction on the columns being sorted.

Observe:
If $d = O(1)$ and $b = O(n)$ then the runtime is $O(n)$!
Analysis

Lemma

Given $n$ $d$-digit number in which each digit can take on up to $b$ possible values. Radix-sort correctly sorts these numbers in $O(d(n + b))$ time if the stable sort it uses takes $O(n + b)$ time.
Analysis

Lemma

Given n d-digit number in which each digit can take on up to b possible values. Radix-sort correctly sorts these numbers in $O(d(n + b))$ time if the stable sort it uses takes $O(n + b)$ time.

Proof
Analysis

**Lemma**

Given $n$ $d$-digit number in which each digit can take on up to $b$ possible values. Radix-sort correctly sorts these numbers in $O(d(n + b))$ time if the stable sort it uses takes $O(n + b)$ time.

**Proof** Runtime is obvious. Correctness follows by induction on the columns being sorted.
Analysis

Lemma

Given $n$ $d$-digit number in which each digit can take on up to $b$ possible values. Radix-sort correctly sorts these numbers in $O(d(n + b))$ time if the stable sort it uses takes $O(n + b)$ time.

Proof Runtime is obvious. Correctness follows by induction on the columns being sorted.

Observe: If $d = O(1)$ and $b = O(n)$ then the runtime is $O(n)!$