The Merge Sort Algorithm

- Used in J. W. Bryce’s Sorting machine in 1938 (U.S. Patent 2189024)

To sort \( a[0], \ldots, a[n-1] \):

1. sort \( a[0], \ldots, a[n/2] \)
2. sort \( a[n/2+1], \ldots, a[n-1] \)
3. merge the two sorted sequences
The Merge Sort Algorithm

- Used in J. W. Bryce’s Sorting machine in 1938 (U.S. Patent 2189024)
- “Invented” by John von Neumann in 1945
The Merge Sort Algorithm

- Used in J. W. Bryce’s Sorting machine in 1938 (U.S. Patent 2189024)
- “Invented” by John von Neumann in 1945
- To sort $a[0], \ldots, a[n-1]$: 

begin
    if $n = 1$ then return $a[0]$; 
    else 
        $l := \lfloor n/2 \rfloor$;
        $r := n - l$;
        $L := \text{mergeSort}(a[0], \ldots, a[l-1])$;
        $R := \text{mergeSort}(a[l], \ldots, a[r-1])$;
        return $\text{merge}(L, R)$; 
end
The Merge Sort Algorithm

- Used in J. W. Bryce’s Sorting machine in 1938 (U.S. Patent 2189024)
- “Invented” by John von Neumann in 1945
- To sort \( a[0], \ldots, a[n-1] \):
  1. sort \( a[0], \ldots, a[n/2] \)
The Merge Sort Algorithm

- Used in J. W. Bryce’s Sorting machine in 1938 (U.S. Patent 2189024)
- “Invented” by John von Neumann in 1945
- To sort $a[0], \ldots, a[n-1]$:
  1. sort $a[0], \ldots, a[n/2]$
  2. sort $a[n/2 + 1], \ldots, a[n - 1]$
The Merge Sort Algorithm

- Used in J. W. Bryce’s Sorting machine in 1938 (U.S. Patent 2189024)
- “Invented” by John von Neumann in 1945
- To sort $a[0], \ldots, a[n − 1]$:  
  1. sort $a[0], \ldots, a[n/2]$  
  2. sort $a[n/2 + 1], \ldots, a[n − 1]$  
  3. merge the two sorted sequences
Mergesort

To sort $a[0], \ldots, a[n - 1]$:

- $\langle 9, 3, 5, 2, 1, 8, 7, 0, 6, 4 \rangle$
- $\langle 9, 3, 5, 2, 1 \rangle \langle 8, 7, 0, 6, 4 \rangle$
Mergesort

- To sort $a[0], \ldots, a[n-1]$: 
  1. sort $a_0 = a[0], \ldots, a[n/2]$ (recursively)

- $\langle 9, 3, 5, 2, 1, 8, 7, 0, 6, 4 \rangle$
- $\langle 1, 2, 3, 5, 9 \rangle \langle 8, 7, 0, 6, 4 \rangle$
Mergesort

To sort $a_0, \ldots, a_{n-1}$:

1. sort $a_0 = a[0], \ldots, a[n/2]$ (recursively)
2. sort $a_1 = a[n/2 + 1], \ldots, a[n - 1]$ (recursively)

$\langle 9, 3, 5, 2, 1, 8, 7, 0, 6, 4 \rangle$
$\langle 1, 2, 3, 5, 9 \rangle \langle 0, 4, 6, 7, 8 \rangle$
Mergesort

To sort \(a[0], \ldots, a[n-1]\):

1. sort \(a0 = a[0], \ldots, a[n/2]\) (recursively)
2. sort \(a1 = a[n/2 + 1], \ldots, a[n-1]\) (recursively)
3. merge the two sorted sequences

\[
\langle 9, 3, 5, 2, 1, 8, 7, 0, 6, 4 \rangle
\]
\[
\langle 1, 2, 3, 5, 9 \rangle \langle 0, 4, 6, 7, 8 \rangle
\]
\[
\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle
\]
void mergeSort(T[] a, Comparator<T> c) {
    if (a.length <= 1) return;
    T[] a0 = Arrays.copyOfRange(a, 0, a.length / 2);
    T[] a1 = Arrays.copyOfRange(a, a.length / 2, a.length);
    mergeSort(a0, c);
    mergeSort(a1, c);
    merge(a0, a1, a, c);
}
To merge two sorted arrays (or lists) a and b we scan them sequentially.

```java
<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {
    int i0 = 0, i1 = 0;
    for (int i = 0; i < a.length; i++) {
        if (i0 == a0.length)
            a[i] = a1[i1++];
        else if (i1 == a1.length)
            a[i] = a0[i0++];
        else if (compare(a0[i0], a1[i1]) < 0)
            a[i] = a0[i0++];
        else
            a[i] = a1[i1++];
    }
}
```
Merging two sorted arrays

To merge two sorted arrays (or lists) $a$ and $b$ we scan them sequentially

```java
<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {
    int i0 = 0, i1 = 0;
    for (int i = 0; i < a.length; i++) {
        if (i0 == a0.length)
            a[i] = a1[i1++];
        else if (i1 == a1.length)
            a[i] = a0[i0++];
        else if (compare(a0[i0], a1[i1]) < 0)
            a[i] = a0[i0++];
        else
            a[i] = a1[i1++];
    }
}
```

Takes $O(n)$ time
Analysis of Mergesort

- Mergesort $a[0], \ldots, a[n - 1]$:

- Let $T(n)$ be the time to run merge sort on an array of length $n$

\footnote{Cheating a bit here, assuming $n$ is a power of 2.}
Analysis of Mergesort

- Mergesort $a[0], \ldots, a[n-1]$:  
  1. sort $a[0], \ldots, a[n/2]$ (recursively)

- Let $T(n)$ be the time to run merge sort on an array of length $n$
- Step 1 Takes $T(n/2)$ time

---

\[^1\text{Cheating a bit here, assuming } n \text{ is a power of 2.}\]
Analysis of Mergesort

- Mergesort $a[0], \ldots, a[n-1]$:  
  1. sort $a[0], \ldots, a[n/2]$ (recursively)  
  2. sort $a[n/2+1], \ldots, a[n-1]$ (recursively)

- Let $T(n)$ be the time to run merge sort on an array of length $n$
- Step 1 Takes $T(n/2)$ time
- Step 2 Takes $T(n/2)$ time

\footnote{Cheating a bit here, assuming $n$ is a power of 2.}
Analysis of Mergesort

Mergesort \( a[0], \ldots, a[n - 1] \):
1. sort \( a[0], \ldots, a[n/2] \) (recursively)
2. sort \( a[n/2 + 1], \ldots, a[n - 1] \) (recursively)
3. merge the two sorted sequences

Let \( T(n) \) be the time to run merge sort on an array of length \( n \)

- Step 1 Takes \( T(n/2) \) time
- Step 2 Takes \( T(n/2) \) time
- Step 3 Takes \( O(n) \) time

---

\(^1\text{Cheating a bit here, assuming } n \text{ is a power of 2.}\)
Analysis of Mergesort

- Mergesort $a[0], \ldots, a[n-1]$:
  1. sort $a[0], \ldots, a[n/2]$ (recursively)
  2. sort $a[n/2+1], \ldots, a[n-1]$ (recursively)
  3. merge the two sorted sequences

- Let $T(n)$ be the time to run merge sort on an array of length $n$
- Step 1 Takes $T(n/2)$ time
- Step 2 Takes $T(n/2)$ time
- Step 3 Takes $O(n)$ time
- $T(n) = O(n) + 2T(n/2)^1$

---

$^1$Cheating a bit here, assuming $n$ is a power of 2.
The Mergesort recurrence

\[ T(n) = O(n) + 2T(n/2) \]
The Mergesort recurrence

- \( T(n) = O(n) + 2T(n/2) \)
- \( T(n) = O(n) + 2O(n/2) + 4T(n/4) \)

Theorem:
The Mergesort algorithm can sort an array of \( n \) items in \( O(n \log n) \) time.
The Mergesort recurrence

- \( T(n) = O(n) + 2T(n/2) \)
- \( T(n) = O(n) + O(n) + 4T(n/4) \)

Theorem: The Mergesort algorithm can sort an array of \( n \) items in \( O(n \log n) \) time.
The Mergesort recurrence

- \( T(n) = O(n) + 2T(n/2) \)
- \( T(n) = O(n) + O(n) + 4T(n/4) \)
- \( T(n) = O(n) + O(n) + 4O(n/4) + 8T(n/8) \)
The Mergesort recurrence

\[ T(n) = O(n) + 2T(n/2) \]
\[ T(n) = O(n) + O(n) + 4T(n/4) \]
\[ T(n) = O(n) + O(n) + O(n) + 8T(n/8) \]

Theorem:
The Mergesort algorithm can sort an array of \( n \) items in \( O(n \log n) \) time.
The Mergesort recurrence

- $T(n) = O(n) + 2T(n/2)$
- $T(n) = O(n) + O(n) + 4T(n/4)$
- $T(n) = O(n) + O(n) + O(n) + 8T(n/8)$
- $T(n) = O(n) + O(n) + O(n) + \cdots + nO(1)$

Theorem: The Mergesort algorithm can sort an array of $n$ items in $O(n \log n)$ time.
The Mergesort recurrence

- \( T(n) = O(n) + 2T(n/2) \)
- \( T(n) = O(n) + O(n) + 4T(n/4) \)
- \( T(n) = O(n) + O(n) + O(n) + 8T(n/8) \)
- \( T(n) = O(n) + O(n) + O(n) + \cdots + O(n) \)
The Mergesort recurrence

- \( T(n) = O(n) + 2T(n/2) \)
- \( T(n) = O(n) + O(n) + 4T(n/4) \)
- \( T(n) = O(n) + O(n) + O(n) + 8T(n/8) \)
- \( T(n) = O(n) + O(n) + O(n) + \cdots + O(n) \)
- \( T(n) = O(n \log n) \)
The Mergesort recurrence

- $T(n) = O(n) + 2T(n/2)$
- $T(n) = O(n) + O(n) + 4T(n/4)$
- $T(n) = O(n) + O(n) + O(n) + 8T(n/8)$
- $T(n) = O(n) + O(n) + O(n) + \cdots + O(n)$
- $T(n) = O(n \log n)$

**Theorem:** The Mergesort algorithm can sort an array of $n$ items in $O(n \log n)$ time.
Mergesort sorts an array of \( n \) elements in \( O(n \log n) \) worst-case time using at most \( n \log n \) comparisons.
QuickSort sorts an array of $n$ elements in $O(n \log n)$ expected time using at most $1.38n \log n$ expected comparisons.
Heapsort sorts an array of $n$ elements in $O(n \log n)$ worst-case time using at most $2n \log n$ comparisons.
Comparison-based sorting algorithms

- So far, we have seen 3 sorting algorithms:
  - Quicksort: $O(n \log n)$ expected time
  - Heapsort: $O(n \log n)$ time
  - Mergesort: $O(n \log n)$ time

- Is there a faster (maybe $O(n)$ time) sorting algorithm?
- Answer: No and yes
So far, we have seen 3 sorting algorithms:

- Quicksort: $O(n \log n)$ expected time
Comparison-based sorting algorithms

So far, we have seen 3 sorting algorithms:

- Quicksort: $O(n \log n)$ expected time
- Heapsort: $O(n \log n)$ time

Is there a faster (maybe $O(n)$ time) sorting algorithm?

Answer: No and yes
So far, we have seen 3 sorting algorithms:

- Quicksort: $O(n \log n)$ expected time
- Heapsort: $O(n \log n)$ time
- Mergesort: $O(n \log n)$ time
So far, we have seen 3 sorting algorithms:

- Quicksort: $O(n \log n)$ expected time
- Heapsort: $O(n \log n)$ time
- Mergesort: $O(n \log n)$ time

Is there a faster (maybe $O(n)$ time) sorting algorithm?
So far, we have seen 3 sorting algorithms:

- Quicksort: $O(n \log n)$ expected time
- Heapsort: $O(n \log n)$ time
- Mergesort: $O(n \log n)$ time

Is there a faster (maybe $O(n)$ time) sorting algorithm?

Answer: No and yes
Comparison-based sorting algorithms

- Quicksort, Heapsort, and Mergesort are comparison-based

These algorithms can be used to sort any array of Comparable items, but this comes at a price.

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ time for some input.
Comparison-based sorting algorithms

- Quicksort, Heapsort, and Mergesort are comparison-based
  - All branching in the algorithm is based on the results of comparisons of the form $a[i] < b[i]$
Comparison-based sorting algorithms

- Quicksort, Heapsort, and Mergesort are comparison-based
  - All branching in the algorithm is based on the results of comparisons of the form $a[i] < b[i]$
  - These algorithms can be used to sort any array of Comparable items
QuickSort, Heapsort, and Mergesort are comparison-based

- All branching in the algorithm is based on the results of comparisons of the form $a[i] < b[i]$
- These algorithms can be used to sort any array of Comparable items
- But this comes at a price
Quicksort, Heapsort, and Mergesort are comparison-based

- All branching in the algorithm is based on the results of comparisons of the form $a[i] < b[i]$
- These algorithms can be used to sort any array of Comparable items
- But this comes at a price
  - Every comparison-based sorting algorithm takes $\Omega(n \log n)$ time for some input
A comparison tree is a full binary tree:
A comparison tree is a full binary tree:
- each internal node $u$ is labelled with a pair $u.i$ and $u.j$
A comparison tree is a full binary tree:
- each internal node $u$ is labelled with a pair $u.i$ and $u.j$
- each leaf is labelled with a permutation of $\{0, \ldots, n-1\}$
A *comparison tree* is a full binary tree:

- each internal node $u$ is labelled with a pair $u.i$ and $u.j$
- each leaf is labelled with a permutation of $\{0, \ldots, n-1\}$

For an array $a$ we can *run* the comparison tree
A *comparison tree* is a full binary tree:
- each internal node $u$ is labelled with a pair $u.i$ and $u.j$
- each leaf is labelled with a permutation of $\{0, \ldots, n-1\}$

For an array $a$ we can *run* the comparison tree
- $u$ is the root

The comparison tree *sorts* if, for every input array $a$, the permutation at the leaf for $a$ correctly sorts $a$
A *comparison tree* is a full binary tree:
- each internal node $u$ is labelled with a pair $u.i$ and $u.j$
- each leaf is labelled with a permutation of $\{0, \ldots, n-1\}$

For an array $a$ we can *run* the comparison tree:
- $u$ is the root
- while $u$ is not a leaf

The comparison tree *sorts* if, for every input array $a$, the permutation at the leaf for $a$ correctly sorts $a$. 
A comparison tree is a full binary tree:
- each internal node $u$ is labelled with a pair $u.i$ and $u.j$
- each leaf is labelled with a permutation of $\{0, \ldots, n-1\}$

For an array $a$ we can run the comparison tree
- $u$ is the root
- while $u$ is not a leaf
  - if $a[u.i] < a[u.j]$ then $u = u.left$ else $u = u.right$

The comparison tree sorts if, for every input array $a$, the permutation at the leaf for $a$ correctly sorts $a$
Comparison tree example

### a[0] ≤ a[1]

- **a[1] ≤ a[2]**
  - **a[0] < a[1] < a[2]**
  - **a[0] ≤ a[2]**
  - **a[1] < a[0] < a[2]**
  - **a[1] ≤ a[2]**

- **a[0] < a[2] < a[1]**
  - **a[0] < a[2]**
  - **a[2] < a[0]**
  - **a[2] < a[1]**
  - **a[2] < a[1] < a[0]**

- **a[0] < a[1]**
  - **a[0] < a[1] < a[2]**
  - **a[0] < a[2]**
  - **a[2] < a[1]**
  - **a[2] < a[1] < a[0]**

- **a[1] < a[0]**
  - **a[1] < a[0] < a[2]**
  - **a[1] < a[2]**
  - **a[2] < a[0]**
  - **a[2] < a[1] < a[0]**

- **a[0] < a[2]**
  - **a[0] < a[2] < a[1]**
  - **a[0] < a[2]**
  - **a[2] < a[1]**
  - **a[2] < a[1] < a[0]**

Pat Morin  COMP2402/2002  Sorting and Sorting Lower Bounds
Lemma: Every comparison tree that sorts any input of length $n$ has at least $n!$ leaves.
Lemma: Every comparison tree that sorts any input of length \( n \) has at least \( n! \) leaves.

Theorem: Every comparison tree that sorts any input of length \( n \) has height at least \((n/2) \log_2(n/2)\).
Comparison tree lower bound

- **Lemma:** Every comparison tree that sorts any input of length \( n \) has at least \( n! \) leaves.

- **Theorem:** Every comparison tree that sorts any input of length \( n \) has height at least \( (n/2) \log_2(n/2) \).
  - The height of a tree with \( m \) leaves is at least \( \log_2 m \).
Lemma: Every comparison tree that sorts any input of length $n$ has at least $n!$ leaves.

Theorem: Every comparison tree that sorts any input of length $n$ has height at least $(n/2) \log_2 (n/2)$.

- The height of a tree with $m$ leaves is at least $\log_2 m$.
- The height of a tree with $n!$ leaves is at least $\log_2 n!$.
Comparison tree lower bound

- **Lemma:** Every comparison tree that sorts any input of length \( n \) has at least \( n! \) leaves.

- **Theorem:** Every comparison tree that sorts any input of length \( n \) has height at least \( (n/2) \log_2 (n/2) \)
  - The height of a tree with \( m \) leaves is at least \( \log_2 m \)
  - The height of a tree with \( n! \) leaves is at least \( \log_2 n! \)

\[
\log_2 n! = \log_2(n) + \log_2(n-1) + \cdots + \log_2(1) \\
\geq \log_2(n) + \cdots + \log_2(n/2) \\
\geq \log_2(n/2) + \cdots + \log_2(n/2) \\
= \left(\frac{n}{2}\right) \log_2 \left(\frac{n}{2}\right)
\]
Lemma: Every comparison tree that sorts any input of length \( n \) has at least \( n! \) leaves.

Theorem: Every comparison tree that sorts any input of length \( n \) has height at least \( (n/2) \log_2 (n/2) \).

- The height of a tree with \( m \) leaves is at least \( \log_2 m \).
- The height of a tree with \( n! \) leaves is at least \( \log_2 n! \).

\[
\log_2 n! = \log_2(n) + \log_2(n-1) + \cdots + \log_2(1) \\
\geq \log_2(n) + \cdots + \log_2(n/2) \\
\geq \log_2(n/2) + \cdots + \log_2(n/2) \\
= (n/2) \log_2(n/2)
\]

Lower bound can be improved to \( n \log n - O(n) \).
Comparison tree lower bound

Does not sort correctly because

\[ a[0] \leq a[2] \]
\[ a[1] < a[0] \leq a[2] \]
\[ a[1] < a[2] < a[0] \]
\[ a[0] < a[2] < a[1] \]
\[ a[0] < a[1] < a[2] \]
\[ a[1] < a[0] < a[2] \]
\[ a[1] < a[2] < a[0] \]

\[ 3! = 3 \cdot 2 \cdot 1 = 6 \]

This tree has only 4 leaves.
Comparison tree lower bound

▶ Does not sort correctly because

▶ $3! = 3 \cdot 2 \cdot 1 = 6$
Does not sort correctly because

- $3! = 3 \cdot 2 \cdot 1 = 6$
- this tree has only 4 < 6 leaves
Every deterministic comparison-based sorting algorithm $A$ that can sort every array of $n$ elements defines a comparison tree $T_A$ that sorts.
Every deterministic comparison-based sorting algorithm $A$ that can sort every array of $n$ elements defines a comparison tree $T_A$ that sorts.

The height of $T_A$ is equal to the (worst-case) number of comparisons that $A$ performs.

Theorem: For every deterministic comparison-based sorting algorithm $A$, there exists an input such that $A$ requires $\Omega(n \log n)$ comparisons.

Theorem: For every comparison-based sorting algorithm $A$, the expected number of comparisons performed by $A$ while sorting a random permutation is $\Omega(n \log n)$. 
Comparison-based sorting and comparison trees

- Every deterministic comparison-based sorting algorithm $\mathcal{A}$ that can sort every array of $n$ elements defines a comparison tree $T_{\mathcal{A}}$ that sorts.
- The height of $T_{\mathcal{A}}$ is equal to the (worst-case) number of comparisons that $\mathcal{A}$ performs.
- **Theorem:** For every deterministic comparison-based sorting algorithm $\mathcal{A}$, there exists an input such that $\mathcal{A}$ requires $\Omega(n \log n)$ comparisons.
- **Theorem:** For every comparison-based sorting algorithm $\mathcal{A}$, the expected number of comparisons performed by $\mathcal{A}$ while sorting a random permutation is $\Omega(n \log n)$. 
Summary

- Mergesort: runs in $O(n \log n)$ time

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

- Mergesort, Quicksort, and Heapsort are optimal comparison-based sorting algorithms

In-class problem:

- Design an algorithm that takes an array $a$ of $n$ integers in the range $\{0, \ldots, k-1\}$ and sorts them in $O(n + k)$ time
Mergesort: runs in $O(n \log n)$ time

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time.
Mergesort: runs in $O(n \log n)$ time

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

Mergesort, Quicksort, and Heapsort are optimal comparison-based sorting algorithms
Mergesort: runs in $O(n \log n)$ time
Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time
Mergesort, Quicksort, and Heapsort are optimal comparison-based sorting algorithms
In-class problem:
Mergesort: runs in $O(n \log n)$ time

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

Mergesort, Quicksort, and Heapsort are optimal comparison-based sorting algorithms

In-class problem:
- Design an algorithm that takes an array $a$ of $n$ integers in the range $\{0, \ldots, k-1\}$ and sorts them in $O(n + k)$ time
Counting sort

```java
int[] countingSort(int[] a, int k) {
    int c[] = new int[k];
    for (int i = 0; i < a.length; i++)
        c[a[i]]++;
    for (int i = 1; i < k; i++)
        c[i] += c[i - 1];
    int b[] = new int[a.length];
    for (int i = a.length - 1; i >= 0; i--)
        b[--c[a[i]]] = a[i];
    return b;
}
```
Theorem: The counting sort algorithm can sort an array $a$ of $n$ integers in the range $\{0, \ldots, k - 1\}$ in $O(n + k)$ time.
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
  - integers have \( w \) bits
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
  - integers have $w$ bits
  - “digit” has $d$ bits
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
  - integers have $w$ bits
  - “digit” has $d$ bits
  - uses $w/d$ passes of counting-sort

Correctness depends on fact that counting sort is stable

if $a[i] = a[j]$ and $i < j$ then $a[i]$ appears before $a[j]$ in the output
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
  - integers have $w$ bits
  - “digit” has $d$ bits
  - uses $w/d$ passes of counting-sort
- Starts by sorting least-significant digits first
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
  - integers have $w$ bits
  - “digit” has $d$ bits
  - uses $w/d$ passes of counting-sort
- Starts by sorting least-significant digits first
  - works up to most significant digits

Correctness depends on fact that counting sort is stable

if $a[i] = a[j]$ and $i < j$ then $a[i]$ appears before $a[j]$ in the output
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
  - integers have $w$ bits
  - “digit” has $d$ bits
  - uses $w/d$ passes of counting-sort
- Starts by sorting least-significant digits first
  - works up to most significant digits
- Correctness depends on fact that counting sort is stable
Radix sort

- Radix-sort uses the counting sort algorithm to sort integers one “digit” at a time
  - integers have $w$ bits
  - “digit” has $d$ bits
  - uses $w/d$ passes of counting-sort
- Starts by sorting least-significant digits first
  - works up to most significant digits
- Correctness depends on fact that counting sort is stable
  - if $a[i] = a[j]$ and $i < j$ then $a[i]$ appears before $a[j]$ in the output
Theorem: The radix-sort algorithm can sort an array $a$ of $n$ $w$-bit integers in $O(n + 2^d)$ time.
Theorem: The radix-sort algorithm can sort an array $a$ of $n$ $w$-bit integers in $O(n + 2^d)$ time.

Theorem: The radix-sort algorithm can sort an array $a$ of $n$ integers in the range $\{0, \ldots, n^c - 1\}$ in $O(cn)$ time.
Quick sort, Heapsort, and Mergesort can each sort an array of length $n$ in $O(n \log n)$ time.
Quicksort, Heapsort, and Mergesort can each sort an array of length \( n \) in \( O(n \log n) \) time.
- These work for any Comparable data type.
Quicksort, Heapsort, and Mergesort can each sort an array of length $n$ in $O(n \log n)$ time

- These work for any Comparable data type
- Quicksort and Heapsort are *in-place* but do more comparisons
QuickSort, Heapsort, and Mergesort can each sort an array of length $n$ in $O(n \log n)$ time

- These work for any Comparable data type
- Quicksort and Heapsort are *in-place* but do more comparisons
- Mergesort requires an auxiliary array
Summary

- Quicksort, Heapsort, and Mergesort can each sort an array of length $n$ in $O(n \log n)$ time
  - These work for any Comparable data type
  - Quicksort and Heapsort are in-place but do more comparisons
  - Mergesort requires an auxiliary array

- Radix-sort can sort an array $a$ of $n$ integers in the range $\{0, \ldots, n^c - 1\}$ in $O(cn)$ time (and does no comparisons).