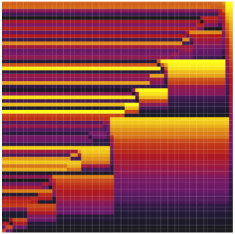
Sorting and Sorting Lower Bounds

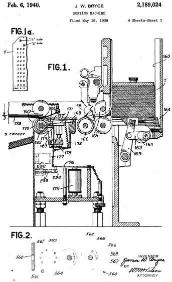
Pat Morin
COMP2402/2002

Carleton University



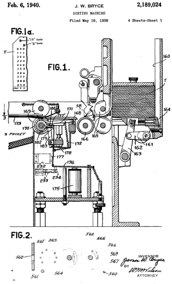
The Merge Sort Algorithm

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



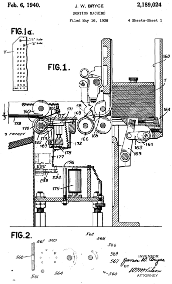
The Merge Sort Algorithm

- Used in J. W. Bryce's Sorting maching in 1938 (U.S. Patent 2189024)
- "Invented" by John von Neumann in 1945



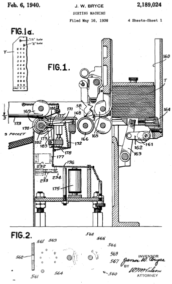
The Merge Sort Algorithm

- Used in J. W. Bryce's Sorting maching in 1938 (U.S. Patent 2189024)
- "Invented" by John von Neumann in 1945
- To sort $\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}-1]$:



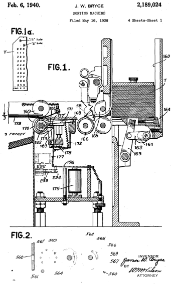
The Merge Sort Algorithm

- Used in J. W. Bryce's Sorting maching in 1938 (U.S. Patent 2189024)
- "Invented" by John von Neumann in 1945
- To sort $a[0], \ldots, a[n-1]$:
  1. sort $a[0], \ldots, a[n/2]$



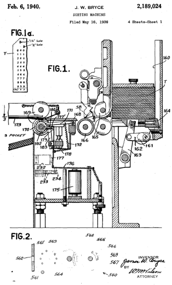
The Merge Sort Algorithm

- Used in J. W. Bryce's Sorting maching in 1938 (U.S. Patent 2189024)
- "Invented" by John von Neumann in 1945
- To sort $\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}-1]$:
  1. sort $\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}/2]$
  2. sort $\mathtt{a}[\mathtt{n}/2+1],\ldots,\mathtt{a}[\mathtt{n}-1]$



The Merge Sort Algorithm

- Used in J. W. Bryce's Sorting maching 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 $\mathtt{a}[0], \ldots, \mathtt{a}[\mathtt{n}-1]$:

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

Mergesort

- To sort $\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}-1]$:
  1. sort $\mathtt{a0}=\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}/2]$ (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 $\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}-1]$:
  1. sort $\mathtt{a0} = \mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}/2]$ (recursively)
  2. sort $\mathtt{a1} = \mathtt{a}[\mathtt{n}/2+1],\ldots,\mathtt{a}[\mathtt{n}-1]$ (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 $\mathtt{a}[0], \ldots, \mathtt{a}[\mathtt{n}-1]$:
  1. sort $\mathtt{a0} = \mathtt{a}[0], \ldots, \mathtt{a}[\mathtt{n}/2]$ (recursively)
  2. sort $\mathtt{a1} = \mathtt{a}[\mathtt{n}/2+1], \ldots, \mathtt{a}[\mathtt{n}-1]$ (recursively)
  3. merge the two sorted sequences
- $\langle 9, 3, 5, 2, 1, 8, 7, 0, 6, 4 \rangle$
- $\langle 1, 2, 3, 5, 9 \rangle \langle 0, 4, 6, 7, 8 \rangle$
- $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle$

Mergesort

```
<T> 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);
}
```

Merging two sorted arrays

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

```
<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

```
<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 $\mathtt{a}[0], \ldots, \mathtt{a}[\mathtt{n}-1]$:
- Let $T(\mathtt{n})$ be the time to run merge sort on an array of length n

---

[1] Cheating a bit here, assuming n is a power of 2.

Analysis of Mergesort

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

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

---

[1] Cheating a bit here, assuming n is a power of 2.

Analysis of Mergesort

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

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

---

[1] Cheating a bit here, assuming n is a power of 2.

Analysis of Mergesort

- Mergesort $\mathtt{a}[0], \ldots, \mathtt{a}[\mathtt{n}-1]$:
  1. sort $\mathtt{a}[0], \ldots, \mathtt{a}[\mathtt{n}/2]$ (recursively)
  2. sort $\mathtt{a}[\mathtt{n}/2+1], \ldots, \mathtt{a}[\mathtt{n}-1]$ (recursively)
  3. merge the two sorted sequences
- Let $T(\mathtt{n})$ be the time to run merge sort on an array of length n
- Step 1 Takes $T(\mathtt{n}/2)$ time
- Step 2 Takes $T(\mathtt{n}/2)$ time
- Step 3 Takes $O(\mathtt{n})$ time

---

[1] Cheating a bit here, assuming n is a power of 2.

Analysis of Mergesort

- Mergesort $\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}-1]$:
  1. sort $\mathtt{a}[0],\ldots,\mathtt{a}[\mathtt{n}/2]$ (recursively)
  2. sort $\mathtt{a}[\mathtt{n}/2+1],\ldots,\mathtt{a}[\mathtt{n}-1]$ (recursively)
  3. merge the two sorted sequences
- Let $T(\mathtt{n})$ be the time to run merge sort on an array of length n
- Step 1 Takes $T(\mathtt{n}/2)$ time
- Step 2 Takes $T(\mathtt{n}/2)$ time
- Step 3 Takes $O(\mathtt{n})$ time
- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$[1]

---

[1] Cheating a bit here, assuming n is a power of 2.

The Mergesort recurrence

- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$

The Mergesort recurrence

- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$
- $T(\mathtt{n}) = O(\mathtt{n}) + 2O(\mathtt{n}/2) + 4T(\mathtt{n}/4)$

The Mergesort recurrence

- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + 4T(\mathtt{n}/4)$

The Mergesort recurrence

- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + 4T(\mathtt{n}/4)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + 4O(\mathtt{n}/4) + 8T(\mathtt{n}/8)$

The Mergesort recurrence

- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + 4T(\mathtt{n}/4)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + O(\mathtt{n}) + 8T(\mathtt{n}/8)$

The Mergesort recurrence

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

The Mergesort recurrence

- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + 4T(\mathtt{n}/4)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + O(\mathtt{n}) + 8T(\mathtt{n}/8)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + O(\mathtt{n}) + \cdots + O(\mathtt{n})$

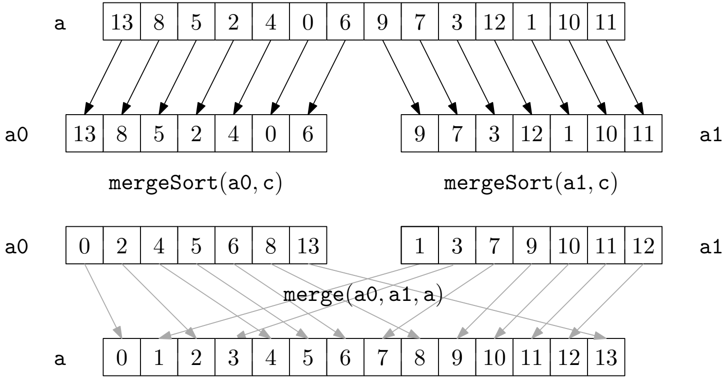
The Mergesort recurrence

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

The Mergesort recurrence

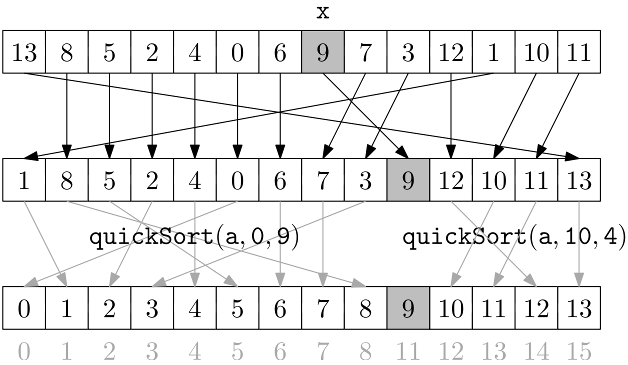
- $T(\mathtt{n}) = O(\mathtt{n}) + 2T(\mathtt{n}/2)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + 4T(\mathtt{n}/4)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + O(\mathtt{n}) + 8T(\mathtt{n}/8)$
- $T(\mathtt{n}) = O(\mathtt{n}) + O(\mathtt{n}) + O(\mathtt{n}) + \cdots + O(\mathtt{n})$
- $T(\mathtt{n}) = O(\mathtt{n}\log\mathtt{n})$
- **Theorem:** The Mergesort algorithm can sort an array of $\mathtt{n}$ items in $O(\mathtt{n}\log\mathtt{n})$ time

Reminder: Mergesort



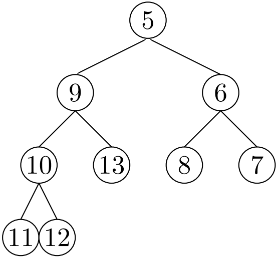
- Mergesort sorts an array of $\mathtt{n}$ elements in $O(\mathtt{n} \log \mathtt{n})$ worst-case time using at most $\mathtt{n} \log \mathtt{n}$ comparisons

Reminder: Quicksort



- Quicksort sorts an array of n elements in $O(n \log n)$ expected time using at most $1.38 n \log n$ expected comparisons

Reminder: Heapsort



5	9	6	10	13	8	7	11	12	4	3	2	1	0
0	1	2	3	4	5	6	7	8	11	12	13	14	15

- Heapsort sorts an array of $\mathtt{n}$ elements in $O(\mathtt{n}\log\mathtt{n})$ worst-case time using at most $2\mathtt{n}\log\mathtt{n}$ comparisons

Comparison-based sorting algorithms

- So far, we have seen 3 sorting algorithms:

Comparison-based sorting algorithms

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

Comparison-based sorting algorithms

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

Comparison-based sorting algorithms

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

Comparison-based sorting algorithms

- So far, we have seen 3 sorting algorithms:
  - Quicksort: $O(\mathtt{n}\log\mathtt{n})$ expected time
  - Heapsort: $O(\mathtt{n}\log\mathtt{n})$ time
  - Mergesort: $O(\mathtt{n}\log\mathtt{n})$ time
- Is there a faster (maybe $O(\mathtt{n})$ time) sorting algorithm?

Comparison-based sorting algorithms

- So far, we have seen 3 sorting algorithms:
  - Quicksort: $O(\mathtt{n}\log\mathtt{n})$ expected time
  - Heapsort: $O(\mathtt{n}\log\mathtt{n})$ time
  - Mergesort: $O(\mathtt{n}\log\mathtt{n})$ time
- Is there a faster (maybe $O(\mathtt{n})$ time) sorting algorithm?
  - Answer: No and yes

Comparison-based sorting algorithms

- Quicksort, Heapsort, and Mergesort are comparison-based

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 $\texttt{a[i]} < \texttt{b[i]}$

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 $\texttt{a[i]} < \texttt{b[i]}$
  - These algorithms can be used to sort any array of `Comparable` items

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 $\texttt{a[i]} < \texttt{b[i]}$
  - These algorithms can be used to sort any array of `Comparable` items
  - But this comes at a price

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 $\mathtt{a[i]} < \mathtt{b[i]}$
  - These algorithms can be used to sort any array of `Comparable` items
  - But this comes at a price
    - Every comparison-based sorting algorithm takes $\Omega(\mathtt{n}\log\mathtt{n})$ time for some input

Comparison trees

- A *comparison tree* is a full binary tree:

Comparison trees

- A *comparison tree* is a full binary tree:
  - each internal node `u` is labelled with a pair `u.i` and `u.j`

Comparison trees

- 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 $\{0, \ldots, n-1\}$

Comparison trees

- 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

Comparison trees

- 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`

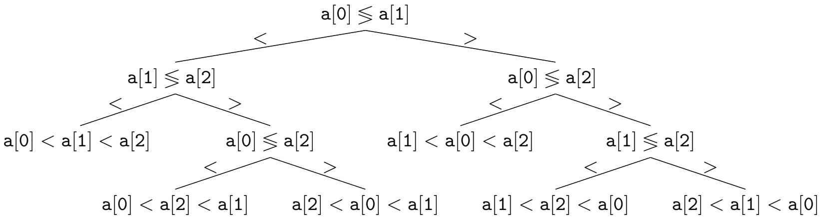
Comparison trees

- 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`

Comparison trees

- 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 $\texttt{u} = \texttt{u.left}$ else $\texttt{u} = \texttt{u.right}$
- The comparison tree *sorts* if, for every input array `a`, the permutation at the leaf for `a` correctly sorts `a`

Comparison tree example



Comparison tree lower bound

- **Lemma:** Every comparison tree that sorts any input of length `n` has at least `n`! leaves

Comparison tree lower bound

- **Lemma:** Every comparison tree that sorts any input of length $\mathtt{n}$ has at least $\mathtt{n}!$ leaves
- **Theorem:** Every comparison tree that sorts any input of length $\mathtt{n}$ has height at least $(\mathtt{n}/2)\log_2(\mathtt{n}/2)$

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 $(\mathtt{n}/2)\log_2(\mathtt{n}/2)$
  - The height of a tree with m leaves is at least $\log_2 m$

Comparison tree lower bound

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

Comparison tree lower bound

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

$$\begin{aligned}
\log_2 \mathtt{n}! &= \log_2(\mathtt{n}) + \log_2(\mathtt{n}-1) + \cdots + \log_2(1) \\
&\geq \log_2(\mathtt{n}) + \cdots + \log_2(\mathtt{n}/2) \\
&\geq \log_2(\mathtt{n}/2) + \cdots + \log_2(\mathtt{n}/2) \\
&= (\mathtt{n}/2)\log_2(\mathtt{n}/2)
\end{aligned}$$

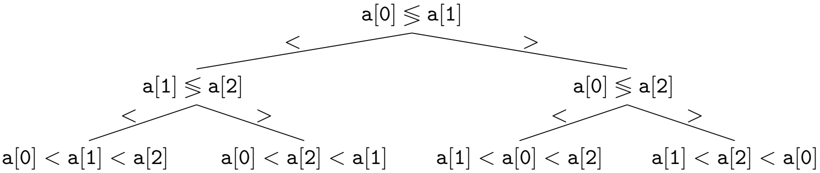
Comparison tree lower bound

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

$$\begin{aligned}
\log_2 \mathtt{n}! &= \log_2(\mathtt{n}) + \log_2(\mathtt{n}-1) + \cdots + \log_2(1) \\
&\geq \log_2(\mathtt{n}) + \cdots + \log_2(\mathtt{n}/2) \\
&\geq \log_2(\mathtt{n}/2) + \cdots + \log_2(\mathtt{n}/2) \\
&= (\mathtt{n}/2)\log_2(\mathtt{n}/2)
\end{aligned}$$

- Lower bound can be improved to $\mathtt{n}\log\mathtt{n} - O(\mathtt{n})$

Comparison tree lower bound



$a[0] \lessgtr a[1]$

$<$ $>$

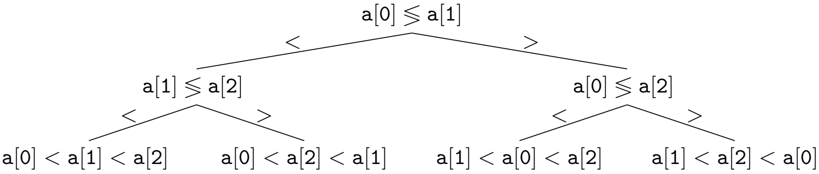
$a[1] \lessgtr a[2]$ $a[0] \lessgtr a[2]$

$<$ $>$ $<$ $>$

$a[0] < a[1] < a[2]$ $a[0] < a[2] < a[1]$ $a[1] < a[0] < a[2]$ $a[1] < a[2] < a[0]$

- Does not sort correctly because

Comparison tree lower bound



$a[0] \lessgtr a[1]$

$<$ $>$

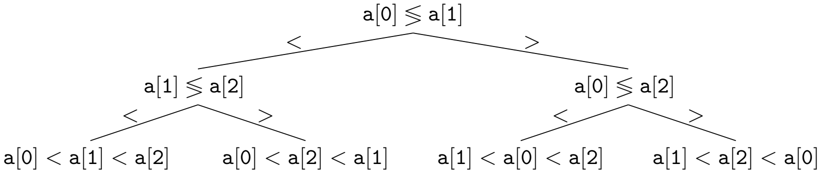
$a[1] \lessgtr a[2]$ $a[0] \lessgtr a[2]$

$<$ $>$ $<$ $>$

$a[0] < a[1] < a[2]$ $a[0] < a[2] < a[1]$ $a[1] < a[0] < a[2]$ $a[1] < a[2] < a[0]$

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

Comparison tree lower bound



- Does not sort correctly because
  - $3! = 3 \cdot 2 \cdot 1 = 6$
  - this tree has only $4 < 6$ leaves

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

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

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 $\Omega(\mathtt{n} \log \mathtt{n})$ comparisons
- **Theorem:** For every comparison-based sorting algorithm $\mathcal{A}$, the expectedd number of comparisons performed by $\mathcal{A}$ while sorting a random permutation is $\Omega(\mathtt{n} \log \mathtt{n})$

Summary

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

Summary

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

Summary

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

Summary

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

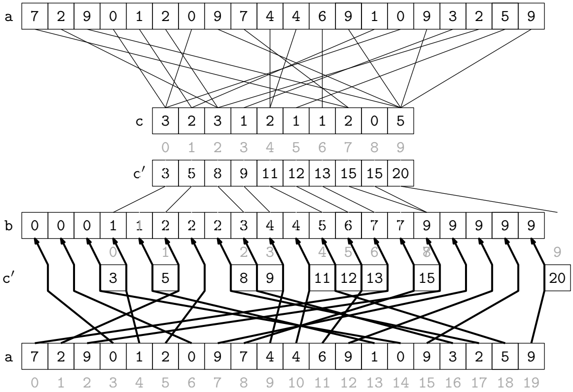
Summary

- Mergesort: runs in $O(\mathtt{n}\log\mathtt{n})$ time
- Any comparison-based sorting algorithm requires $\Omega(\mathtt{n}\log\mathtt{n})$ 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 $\{0,\ldots,\mathtt{k}-1\}$ and sorts them in $O(\mathtt{n}+\mathtt{k})$ time

Counting sort

```
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;
}
```

Counting sort



- **Theorem:** The counting sort algorithm can sort an array $\mathtt{a}$ of $\mathtt{n}$ integers in the range $\{0,\ldots,\mathtt{k}-1\}$ in $O(\mathtt{n}+\mathtt{k})$ 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

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

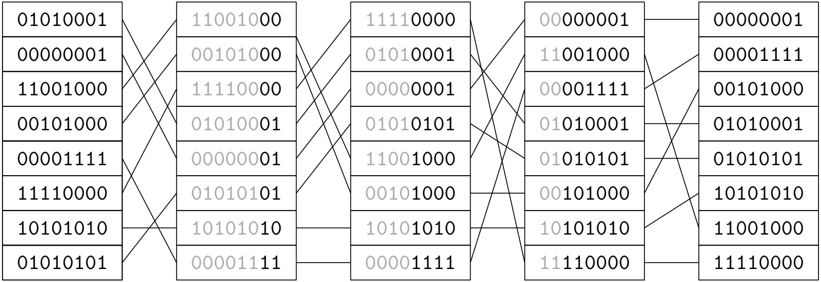
Radix sort

- 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

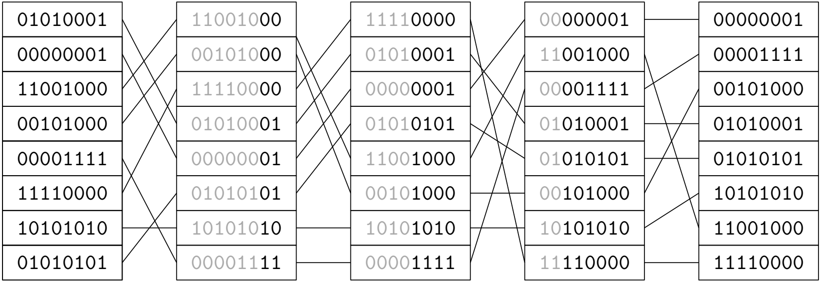
Counting sort



01010001	11001000	11110000	00000001	00000001
00000001	00101000	01010001	11001000	00001111
11001000	11110000	00000001	00001111	00101000
00101000	01010001	01010101	01010001	01010001
00001111	00000001	11001000	01010101	01010101
11110000	01010101	00101000	00101000	10101010
10101010	10101010	10101010	10101010	11001000
01010101	00001111	00001111	11110000	11110000

- **Theorem:** The radix-sort algorithm can sort an array $\mathtt{a}$ of $\mathtt{n}$ $\mathtt{w}$-bit integers in $O(\mathtt{n}+2^{\mathtt{d}})$ time

Counting sort



01010001	11001000	11110000	00000001	00000001
00000001	00101000	01010001	11001000	00001111
11001000	11110000	00000001	00001111	00101000
00101000	01010001	01010101	01010001	01010001
00001111	00000001	11001000	01010101	01010101
11110000	01010101	00101000	00101000	10101010
10101010	10101010	10101010	10101010	11001000
01010101	00001111	00001111	11110000	11110000

- **Theorem:** The radix-sort algorithm can sort an array $\mathtt{a}$ of $\mathtt{n}$ $\mathtt{w}$-bit integers in $O(\mathtt{n} + 2^{\mathtt{d}})$ time
- **Theorem:** The radix-sort algorithm can sort an array $\mathtt{a}$ of $\mathtt{n}$ integers in the range $\{0, \ldots, \mathtt{n}^c - 1\}$ in $O(c\mathtt{n})$ time.

Summary

- Quicksort, Heapsort, and Mergesort can each sort an array of length $\mathtt{n}$ in $O(\mathtt{n}\log\mathtt{n})$ time

Summary

- Quicksort, Heapsort, and Mergesort can each sort an array of length `n` in $O(\mathtt{n} \log \mathtt{n})$ time
    - These work for any `Comparable` data type

Summary

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

Summary

- Quicksort, Heapsort, and Mergesort can each sort an array of length `n` in $O(\mathtt{n}\log\mathtt{n})$ 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 $O(\mathtt{n}\log\mathtt{n})$ 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 $\{0,\ldots,\mathtt{n}^c-1\}$ in $O(c\mathtt{n})$ time (and does no comparisons).