2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
Two classic sorting algorithms: mergesort and quicksort

Critical components in the world’s computational infrastructure.
- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [this lecture]

Quicksort. [next lecture]
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
Mergesort

Basic plan.

- Divide array into two halves.
- **Recursively** sort each half.
- Merge two halves.

---

**Example**

<table>
<thead>
<tr>
<th>input</th>
<th>M E R G E S O R T E X A M P L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left</td>
<td>E E G M O R R S</td>
</tr>
<tr>
<td>half</td>
<td>T E X A M P L E</td>
</tr>
<tr>
<td>sort right</td>
<td>E E G M O R R S</td>
</tr>
<tr>
<td>half</td>
<td>A E E L M P T X</td>
</tr>
<tr>
<td>merge results</td>
<td>A E E E E E G L M M O P R R S T X</td>
</tr>
</tbody>
</table>

Mergesort overview
Abstract in-place merge demo

**Goal.** Given two sorted subarrays \( a[lo] \) to \( a[mid] \) and \( a[mid+1] \) to \( a[hi] \), replace with sorted subarray \( a[lo] \) to \( a[hi] \).
Mergesort: Transylvanian–Saxon folk dance

http://www.youtube.com/watch?v=XaqR3G_NVoo
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++)
    {
        if  (i > mid) a[k] = aux[j++];
        else if (j > hi) a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else a[k] = aux[i++];
    }
}
Mergesort quiz 1

How many calls does merge() make to less() in order to merge two sorted subarrays, each of length $n/2$, into a sorted array of length $n$?

A. $\sim \frac{1}{4} n$ to $\sim \frac{1}{2} n$

B. $\sim \frac{1}{2} n$

C. $\sim \frac{1}{2} n$ to $\sim n$

D. $\sim n$
public class Merge
{
    private static void merge(...)
    {
    } /* as before */

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
    {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a)
    {
        Comparable[] aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
Mergesort: trace

```plaintext
Trace of merge results for top-down mergesort

merge(a, aux, 0, 0, 1)
merge(a, aux, 2, 2, 3)
merge(a, aux, 0, 1, 3)
merge(a, aux, 4, 4, 5)
merge(a, aux, 6, 6, 7)
merge(a, aux, 4, 5, 7)
merge(a, aux, 0, 3, 7)
merge(a, aux, 8, 8, 9)
merge(a, aux, 10, 10, 11)
merge(a, aux, 8, 9, 11)
merge(a, aux, 12, 12, 13)
merge(a, aux, 14, 14, 15)
merge(a, aux, 12, 13, 15)
merge(a, aux, 8, 11, 15)
merge(a, aux, 0, 7, 15)
```

result after recursive call
Which of the following subarray lengths will occur when running mergesort on an array of length 12?

A. { 1, 2, 3, 4, 6, 8, 12 }
B. { 1, 2, 3, 6, 12 }
C. { 1, 2, 4, 8, 12 }
D. { 1, 3, 6, 9, 12 }
Mergesort: animation

50 random items

http://www.sorting-algorithms.com/merge-sort

- ▲ algorithm position
- in order
- current subarray
- not in order
Mergesort: animation

50 reverse-sorted items

http://www.sorting-algorithms.com/merge-sort
Running time estimates:
- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>computer</th>
<th>insertion sort (n^2)</th>
<th>mergesort (n log n)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>

Bottom line. Good algorithms are better than supercomputers.
Mergesort analysis: number of compares

**Proposition.** Mergesort uses \( \leq n \lg n \) compares to sort an array of length \( n \).

**Pf sketch.** The number of compares \( C(n) \) to mergesort an array of length \( n \) satisfies the recurrence:

\[
C(n) \leq C\left(\lceil \frac{n}{2} \rceil \right) + C\left(\lfloor \frac{n}{2} \rfloor \right) + n - 1 \quad \text{for } n > 1, \text{ with } C(1) = 0.
\]

We solve this simpler recurrence, and assume \( n \) is a power of 2:

\[
D(n) = 2D(n/2) + n, \text{ for } n > 1, \text{ with } D(1) = 0.
\]

(result holds for all \( n \) (analysis cleaner in this case))
**Divide-and-conquer recurrence**

**Proposition.** If $D(n)$ satisfies $D(n) = 2D(n/2) + n$ for $n > 1$, with $D(1) = 0$, then $D(n) = n \lg n$.

**Pf by picture.** [assuming $n$ is a power of 2]

\[
\begin{align*}
T(n) &= n \lg n
\end{align*}
\]
Mergesort analysis: number of array accesses

Proposition. Mergesort uses \( \leq 6n \lg n \) array accesses to sort an array of length \( n \).

Pf sketch. The number of array accesses \( A(n) \) satisfies the recurrence:

\[
A(n) \leq A(\lceil n/2 \rceil) + A(\lfloor n/2 \rfloor) + 6n \quad \text{for } n > 1, \text{ with } A(1) = 0.
\]

Key point. Any algorithm with the following structure takes \( n \log n \) time:

```java
public static void f(int n)
{
    if (n == 0) return;
    f(n/2); \rightarrow \text{solve two problems}
    f(n/2); \rightarrow \text{of half the size}
    linear(n); \rightarrow \text{do a linear amount of work}
}
```

Notable examples. FFT, hidden-line removal, Kendall-tau distance, ...
Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to \( n \).

Pf. The array \( \text{aux}[] \) needs to be of length \( n \) for the last merge.

![two sorted subarrays](image)

merged result

![merged result](image)

Def. A sorting algorithm is **in-place** if it uses \( \leq c \log n \) extra memory.

Ex. Insertion sort and selection sort.

Challenge 1 (not hard). Use \( \text{aux}[] \) array of length \( \sim \frac{1}{2} n \) instead of \( n \).

Challenge 2 (very hard). In-place merge. [Kronrod 1969]
Mergesort quiz 3

Is our implementation of mergesort **stable**?

A. Yes.

B. No, but it can be easily modified to be stable.

C. No, mergesort is inherently unstable.

D. *I don’t remember what stability means.*

---

A sorting algorithm is stable if it preserves the relative order of equal keys.

**Input:** C A₁ B A₂ A₃

**Sorted:** A₃ A₁ A₂ B C

*not stable*
**Stability: mergesort**

**Proposition.** Mergesort is stable.

```java
public class Merge {
    private static void merge(...) {
        /* as before */
    }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a) {
        /* as before */
    }
}
```

**Pf.** Suffices to verify that merge operation is stable.
Stability: mergesort

**Proposition.** Merge operation is **stable**.

```java
private static void merge(...) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++)
        {  
            if  (i > mid) a[k] = aux[j++];
            else if (j > hi) a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else a[k] = aux[i++];
        }
}
```

---

**Pf.** Takes from left subarray if equal keys.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>B</td>
<td>D</td>
<td>A4</td>
<td>A5</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 10 \) items.

```java
private static void sort(...) {
    if (hi <= lo + CUTOFF - 1) {
        Insertion.sort(a, lo, hi);
        return;
    }

    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```
Mergesort with cutoff to insertion sort: visualization

first subarray
second subarray
first merge
first half sorted
second half sorted
result
Mergesort: practical improvements

Stop if already sorted.
- Is largest item in first half \( \leq \) smallest item in second half?
- Helps for partially ordered arrays.

```java
private static void sort(...) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```
Mergesort: practical improvements

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```java
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid) aux[k] = a[j++];
        else if (j > hi) aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++];
        else aux[k] = a[i++];
    }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(aux, a, lo, mid);
    sort(aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

merge from a[] to aux[]

switch roles of aux[] and a[]
Java 6 system sort

Basic algorithm in Arrays.sort() for sorting objects = mergesort.

- Cutoff to insertion sort = 7.
- Stop-if-already-sorted test.
- Eliminate-the-copy-to-the-auxiliary-array trick.

http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/tip/src/share/classes/java/util/Arrays.java
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ....

<table>
<thead>
<tr>
<th>sz = 1</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>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, aux, 0, 0, 1)</td>
<td>E M R G E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge(a, aux, 2, 2, 3)</td>
<td>E M G R E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge(a, aux, 4, 4, 5)</td>
<td>E M G R E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge(a, aux, 6, 6, 7)</td>
<td>E M G R E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge(a, aux, 8, 8, 9)</td>
<td>E M G R E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge(a, aux, 10, 10, 11)</td>
<td>E M G R E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge(a, aux, 12, 12, 13)</td>
<td>E M G R E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge(a, aux, 14, 14, 15)</td>
<td>E M G R E S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, aux, 0, 1, 3)</td>
</tr>
<tr>
<td>merge(a, aux, 4, 5, 7)</td>
</tr>
<tr>
<td>merge(a, aux, 8, 9, 11)</td>
</tr>
<tr>
<td>merge(a, aux, 12, 13, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, aux, 0, 3, 7)</td>
</tr>
<tr>
<td>merge(a, aux, 8, 11, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, aux, 0, 7, 15)</td>
</tr>
</tbody>
</table>
public class MergeBU
{
    private static void merge(...)
    { /* as before */ }

    public static void sort(Comparable[] a)
    {
        int n = a.length;
        Comparable[] aux = new Comparable[n];
        for (int sz = 1; sz < n; sz = sz+sz)
            for (int lo = 0; lo < n-sz; lo += sz+sz)
                merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, n-1));
    }
}
Mergesort: visualizations

top-down mergesort (cutoff = 12)  

bottom-up mergesort (cutoff = 12)
Which is faster in practice: top-down mergesort or bottom-up mergesort? You may assume $n$ is a power of 2.

A. Top-down (recursive) mergesort.
B. Bottom-up (non-recursive) mergesort.
C. No observable difference.
D. I don't know.
Natural mergesort

**Idea.** Exploit pre-existing order by identifying naturally occurring runs.

```plaintext
input

1  5  10  16  3  4  23  9  13  2  7  8  12  14

first run

1  5  10  16  3  4  23  9  13  2  7  8  12  14

second run

1  5  10  16  3  4  23  9  13  2  7  8  12  14

merge two runs

1  3  4  5  10  16  23  9  13  2  7  8  12  14
```

**Tradeoff.** Fewer passes vs. extra compares per pass to identify runs.
Timsort

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Intro
-----
This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than \( \lg(n!) \) comparisons needed, and as few as \( n-1 \)), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

... 

Consequence. Linear time on many arrays with pre-existing order.

Now widely used. Python, Java 7–9, GNU Octave, Android, ....

http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/java/util/Arrays.java
Proving that Android’s, Java’s and Python’s sorting algorithm is broken (and showing how to fix it)

February 24, 2015

Tim Peters developed the **Timsort hybrid sorting algorithm** in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun’s JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it
# Sorting summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$n$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔️ ✔️</td>
<td>$n$</td>
<td>$\frac{1}{4} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>use for small $n$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td>$n \log_3 n$</td>
<td>?</td>
<td>$c n^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</td>
<td>$\frac{1}{2} n \lg n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>$n \log n$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td>$n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>improves mergesort when pre-existing order</td>
</tr>
<tr>
<td>?</td>
<td>✔️ ✔️</td>
<td>$n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
Computational complexity. Framework to study efficiency of algorithms for solving a particular problem $X$.

Model of computation. Allowable operations.
Cost model. Operation counts.
Upper bound. Cost guarantee provided by some algorithm for $X$.
Lower bound. Proven limit on cost guarantee of all algorithms for $X$.
Optimal algorithm. Algorithm with best possible cost guarantee for $X$.

<table>
<thead>
<tr>
<th>model of computation</th>
<th>decision tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost model</td>
<td># compares</td>
</tr>
<tr>
<td>upper bound</td>
<td>$\sim n \log n$ from mergesort</td>
</tr>
<tr>
<td>lower bound</td>
<td>?</td>
</tr>
<tr>
<td>optimal algorithm</td>
<td>?</td>
</tr>
</tbody>
</table>

Complexity of sorting
Decision tree (for 3 distinct keys a, b, and c)

height of tree = worst-case number of compares

each leaf corresponds to one (and only one) ordering; (at least) one leaf for each possible ordering
**Proposition.** Any compare-based sorting algorithm must make at least \( \lg (n!) \sim n \lg n \) compares in the worst-case.

**Pf.**

- Assume array consists of \( n \) distinct values \( a_1 \) through \( a_n \).
- Worst case dictated by height \( h \) of decision tree.
- Binary tree of height \( h \) has at most \( 2^h \) leaves.
- \( n! \) different orderings \( \Rightarrow \) at least \( n! \) leaves.

![Decision Tree Diagram](diagram.png)
Compare-based lower bound for sorting

**Proposition.** Any compare-based sorting algorithm must make at least 
\[ \lg (n!) \sim n \lg n \] compares in the worst-case.

**Pf.**
- Assume array consists of \( n \) distinct values \( a_1 \) through \( a_n \).
- Worst case dictated by height \( h \) of decision tree.
- Binary tree of height \( h \) has at most \( 2^h \) leaves.
- \( n! \) different orderings \( \implies \) at least \( n! \) leaves.

\[
2^h \geq \text{# leaves} \geq n!
\]

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

*Stirling’s formula*
Complexity of sorting

Model of computation. Allowable operations.
Cost model. Operation count(s).
Upper bound. Cost guarantee provided by some algorithm for $X$.
Lower bound. Proven limit on cost guarantee of all algorithms for $X$.
Optimal algorithm. Algorithm with best possible cost guarantee for $X$.

<table>
<thead>
<tr>
<th>model of computation</th>
<th>decision tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost model</td>
<td># compares</td>
</tr>
<tr>
<td>upper bound</td>
<td>$\sim n \lg n$</td>
</tr>
<tr>
<td>lower bound</td>
<td>$\sim n \lg n$</td>
</tr>
<tr>
<td>optimal algorithm</td>
<td>mergesort</td>
</tr>
</tbody>
</table>

complexity of sorting

First goal of algorithm design: optimal algorithms.
Complexity results in context

**Compares?** Mergesort is optimal with respect to number compares.

**Space?** Mergesort is not optimal with respect to space usage.

**Lessons.** Use theory as a guide.

**Ex.** Design sorting algorithm that guarantees $\sim \frac{1}{2} n \log n$ compares?

**Ex.** Design sorting algorithm that is both time- and space-optimal?
Q. Why doesn’t this Skittles machine violate the sorting lower bound?

https://www.youtube.com/watch?v=tSEHDBSyVo
Lower bound may not hold if the algorithm can take advantage of:

- The initial order of the input array.
  Ex: insertion sort requires only a linear number of compares on partially sorted arrays.

- The distribution of key values.
  Ex: 3-way quicksort requires only a linear number of compares on arrays with a constant number of distinct keys. [stay tuned]

- The representation of the keys.
  Ex: radix sorts require no key compares — they access the data via character/digit compares. [stay tuned]
## Commonly Used Notations in the Theory of Algorithms

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tilde</strong></td>
<td>leading term</td>
<td>$\sim \frac{1}{2} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\frac{1}{2} n^2 + 22n \log n + 3n$</td>
</tr>
<tr>
<td><strong>Big Theta</strong></td>
<td>order of growth</td>
<td>$\Theta(n^2)$</td>
<td>$\frac{1}{2} n^2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$10n^2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$5n^2 + 22n \log n + 3n$</td>
</tr>
<tr>
<td><strong>Big O</strong></td>
<td>upper bound</td>
<td>$O(n^2)$</td>
<td>$10n^2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$100n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$22n \log n + 3n$</td>
</tr>
<tr>
<td><strong>Big Omega</strong></td>
<td>lower bound</td>
<td>$\Omega(n^2)$</td>
<td>$\frac{1}{2} n^2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n^5$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n^3 + 22n \log n + 3n$</td>
</tr>
</tbody>
</table>
**Sorting Lower Bound**

**Interviewer.** Give a formal description of the sorting lower bound for sorting an array of $n$ elements.

**Cornell student.** I use `Arrays.sort()`.

**Yale student.** Any sorting algorithm takes at least $n \log n$ time.

**Harvard student.** Any compare-based sorting algorithm must make at least $O(n \log n)$ compares.

**Princeton student.** To sort an arbitrary array of $n$ items, any compare-based sorting algorithm must make $\Omega(n \log n)$ compares in the worst case.
2.2 MERGESORT

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
**Problem.** Given a singly linked list, rearrange its nodes in sorter order.

**Version 1.** Linearithmic time, linear extra space.

**Version 2.** Linearithmic time, logarithmic (or constant) extra space.

---

**Input**

```
5♣ -> 6♣ -> 2♣ -> 7♣ -> 3♣ -> 4♣ -> null
```

**Sorted**

```
2♣ -> 3♣ -> 4♣ -> 5♣ -> 6♣ -> 7♣ -> null
```