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

[https://algs4.cs.princeton.edu](https://algs4.cs.princeton.edu)
Mergesort

Basic plan.

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

<table>
<thead>
<tr>
<th>input</th>
<th>MERGESORTEXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left half</td>
<td>EEGMORRSTEXAMPLE</td>
</tr>
<tr>
<td>sort right half</td>
<td>EEGMORRSSAEELMPTX</td>
</tr>
<tr>
<td>merge results</td>
<td>AEEXEELMMOPRRSTX</td>
</tr>
</tbody>
</table>

Mergesort overview

First Draft of a Report on the EDVAC

John von Neumann
**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 \( a[hi] \).

<table>
<thead>
<tr>
<th>( a[] )</th>
<th>( lo )</th>
<th>( mid )</th>
<th>( mid+1 )</th>
<th>( hi )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*sorted*
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++];
    }
}
How many calls does \texttt{merge()} make to \texttt{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

```
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)
```

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

`result after recursive call`

```markdown
a[]

<table>
<thead>
<tr>
<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>M</td>
<td>E</td>
<td>R</td>
<td>G</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>M</td>
<td>G</td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>E</td>
<td>T</td>
<td>A</td>
<td>X</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>M</td>
<td>P</td>
<td>E</td>
<td>L</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>P</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>P</td>
<td>T</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>T</td>
</tr>
</tbody>
</table>
```
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
Mergesort: animation

50 reverse-sorted items

http://www.sorting-algorithms.com/merge-sort
Mergesort: empirical analysis

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 any array of length \( n \).

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

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

**For simplicity:** Assume \( n \) is a power of 2 and solve this recurrence:

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

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

**Pf by picture.** [assuming \( n \) is a power of 2]

![Divide-and-conquer recurrence diagram](image-url)
Mergesort analysis: number of array accesses

Proposition. Mergesort makes $\Theta(n \log n)$ array accesses.

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

$$A(n) = A(\lceil n/2 \rceil) + A(\lfloor n/2 \rfloor) + \Theta(n) \text{ for } n > 1, \text{ with } A(1) = 0.$$ 

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

```java
public static void f(int n)
{
    if (n == 0) return;
    f(n/2); \hspace{1cm} \text{solve two problems}
    f(n/2); \hspace{1cm} \text{of half the size}
    linear(n); \hspace{1cm} \text{do a linear amount of work}
}
```

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

**Proposition.** Mergesort uses $\Theta(n)$ extra space.

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

---

two sorted subarrays

```
A C D G H I M N U V  B E F J O P Q R S T
```

merged result

```
A B C D E F G H I J M N O P Q R S T U V
```

---

**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]
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.

<table>
<thead>
<tr>
<th>input</th>
<th>C</th>
<th>A₁</th>
<th>B</th>
<th>A₂</th>
<th>A₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td>A₃</td>
<td>A₁</td>
<td>A₂</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

not stable

skipped in lecture (see precept)
Mergesort: practical improvement

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 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
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, ....
Bottom-up mergesort: Java implementation

```java
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));
    }
}
```

**Bottom line.** Simple and non-recursive version of mergesort.
Mergesort: visualizations

- Top-down mergesort (cutoff = 12)
- Bottom-up mergesort (cutoff = 12)
Which is faster in practice for \( n = 2^{20} \), top-down mergesort or bottom-up mergesort?

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

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

**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 \( \log(n) \)) 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 7–11, GNU Octave, Android, ....

https://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 Envisage Written by Stijn de Gouw. $s

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
Timsort bug (May 2018)

Execution error in Java's Timsort

Details

- **Type:** Bug
- **Status:** RESOLVED
- **Priority:** P3
- **Resolution:** Fixed
- **Affects Version/s:** None
- **Fix Version/s:** 11
- **Component/s:** core-libs
- **Labels:** None
- **Subcomponent:** java.util.collections
- **Introduced In Version:** 6
- **Resolved In Build:** b20

Description

Carine Pivoteau wrote:
While working on a proper complexity analysis of the algorithm, we realised that there was an error in the last paper reporting such a bug (http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf). This implies that the correction implemented in the Java source code (changing Timsort stack size) is wrong and that it is still possible to make it break. This is explained in full details in our analysis: https://arxiv.org/pdf/1805.08812.pdf.

We understand that coming upon data that actually causes this error is very unlikely, but we thought you’d still like to know and do something about it. As the authors of the previous article advocated for, we strongly believe that you should consider modifying the algorithm as explained in their article (and as was done in Python) rather than trying to fix the stack size.
<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></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>selection</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>insertion</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>shell</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>merge</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>timsort</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></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>
<tr>
<td>?</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$n \lg \varphi$, where $\varphi = \#$ runs (proved in August 2018)
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
Complexity of sorting

**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>comparison tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost model</td>
<td># compares</td>
</tr>
<tr>
<td>upper bound</td>
<td>$\sim n \lg n$ from mergesort</td>
</tr>
<tr>
<td>lower bound</td>
<td>?</td>
</tr>
<tr>
<td>optimal algorithm</td>
<td>?</td>
</tr>
</tbody>
</table>

lower bound $\sim$ upper bound

Complexity of sorting

can access information only through compares (e.g., Java Comparable framework)
Comparison tree (for 3 distinct keys a, b, and c)

Each reachable leaf corresponds to one (and only one) ordering; exactly one reachable leaf for each possible ordering.

Height of pruned comparison tree = worst-case number of compares.

Code between compares (e.g., sequence of exchanges).
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 number of compares = height \( h \) of pruned comparison tree.
- Binary tree of height \( h \) has \( \leq 2^h \) leaves.
- \( n! \) different orderings \( \Rightarrow \) \( n! \) reachable leaves.
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 number of compares = height \( h \) of pruned comparison tree.
- Binary tree of height \( h \) has \( \leq 2^h \) leaves.
- \( n! \) different orderings \( \Rightarrow \) \( n! \) reachable leaves.

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

\[
\Rightarrow h \geq \lg(n!)
\]

\[
\sim n \lg n
\]

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>comparison 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>

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 $\sim \frac{1}{2} n \lg n$ compares?

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

https://www.youtube.com/watch?v=tSEHDBSxnVo
Complexity results in context (continued)

Lower bound may not hold if the algorithm can take advantage of:

- The initial order of the input array.
  Ex: insertion sort makes only $\Theta(n)$ compares on partially sorted arrays.

- The distribution of key values.
  Ex: 3-way quicksort makes only $\Theta(n)$ compares on arrays with a constant number of distinct keys. [stay tuned]

- The representation of the keys.
  Ex: radix sorts do not make any key compares; they access the data via character/digit compares. [stay tuned]
## Asymptotic notations

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
</tr>
</thead>
<tbody>
<tr>
<td>tilde</td>
<td>leading term</td>
<td>~ ( \frac{1}{2} n^2 )</td>
<td>( \frac{1}{2} n^2 ) ( \frac{1}{2} n^2 + 3n + 22 )</td>
</tr>
<tr>
<td>big Theta</td>
<td>order of growth</td>
<td>( \Theta(n^2) )</td>
<td>( \frac{1}{2} n^2 ) ( 7n^2 + n^{\frac{1}{2}} ) ( 5n^2 - 3n )</td>
</tr>
<tr>
<td>big O</td>
<td>upper bound</td>
<td>( O(n^2) )</td>
<td>( 10n^2 ) ( 22n ) ( \log_2 n )</td>
</tr>
<tr>
<td>big Omega</td>
<td>lower bound</td>
<td>( \Omega(n^2) )</td>
<td>( \frac{1}{2} n^2 ) ( n^3 + 3n ) ( 2^n )</td>
</tr>
</tbody>
</table>
Interviewer. Give a formal description of the sorting lower bound for sorting an array of $n$ elements.
2.2 *Mergesort*

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

Application. Sort list of inodes to garbage collect in Linux kernel.

Version 0. $\Theta(n \log n)$ time, $\Theta(n)$ extra space.

Version 1. $\Theta(n \log n)$ time, $\Theta(\log n)$ extra space.

Version 2. $\Theta(n \log n)$ time, $\Theta(1)$ extra space.