2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity

https://algs4.cs.princeton.edu
Two classic sorting algorithms: mergesort and quicksort

Critical components in our computational infrastructure.

**Mergesort.** [this lecture]

**Quicksort.** [next lecture]
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
Mergesort overview

Basic plan.

• Divide array into two halves.
• Recursively sort left half.
• Recursively sort right half.
• Merge two sorted halves.

<table>
<thead>
<tr>
<th>input</th>
<th>MERGESORTEXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left half</td>
<td>EEGMORRSETEXAMPLE</td>
</tr>
<tr>
<td>sort right half</td>
<td>EEGMORRSSALEELMPTX</td>
</tr>
<tr>
<td>merge results</td>
<td>AEEEEEGLMMOPRSTX</td>
</tr>
</tbody>
</table>
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 \(a[hi]\).
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()` when merging two sorted subarrays, each of length \( \frac{n}{2} \), into a sorted array of length \( n \)?

A. \(~\frac{1}{4} n\) to \(~\frac{1}{2} n\)

B. \(~\frac{1}{2} n\)

C. \(~\frac{1}{2} n\) to \(~n\)

D. \(~n\)
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);
    }
}

avoid allocating arrays within recursive function calls

<table>
<thead>
<tr>
<th>lo</th>
<th>mid</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mergesort: trace

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

$a[]$

<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>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>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>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>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>
</tbody>
</table>

result after recursive call
Which subarray lengths will arise when mergesorting 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

https://www.toptal.com/developers/sorting-algorithms/merge-sort
Mergesort: animation

50 reverse-sorted items

https://www.toptal.com/developers/sorting-algorithms/merge-sort
Mergesort: empirical analysis

Running time estimates:

• Laptop executes $10^8$ compares/second.
• Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th></th>
<th>insertion sort $(n^2)$</th>
<th>mergesort $(n \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>computer</td>
<td>thousand</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 \( \leq n \log_2 n \) 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:

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

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

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

proposition holds even when \( n \) is not a power of 2
(but analysis cleaner in this case)
**Proposition.** If \( D(n) \) satisfies \( D(n) = 2 \, D(n/2) + n \) for \( n > 1 \), with \( D(1) = 0 \), then \( D(n) = n \log_2 n \).

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

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

**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);  // solve two problems of half the size
    f(n/2);
    linear(n);  // do $\Theta(n)$ work
    linear(n);
}
```

**Famous examples.** FFT, closest pair, hidden–line removal, Kendall–tau distance, …
**Mergesort analysis: memory**

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

**Pf.** The length of the `aux[]` array is $n$, to handle the last merge.

---

Two sorted subarrays

```
A B C D E F G H I J M N 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
```

effectively negligible

---

**Def.** A sorting algorithm is **in-place** if it uses $\Theta(\log n)$ extra space (or less).

**Ex.** Insertion sort and selection sort.

**Challenge 1 (not hard).** Get by with an `aux[]` array of length $\sim \frac{1}{2} n$ (instead of $n$).

**Challenge 2 (very hard).** In-place merge. [Kronrod 1969]
Consider the following modified version of mergesort.
How much total memory is allocated over all recursive calls?

A. $\Theta(n)$
B. $\Theta(n \log n)$
C. $\Theta(n^2)$
D. $\Theta(2^n)$

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

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);
}
```

makes mergesort about 20% faster
2.2 MERGESORT

- mergesort
- bottom-up mergesort
- sorting complexity
Bottom-up mergesort

Basic plan.

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

<table>
<thead>
<tr>
<th>sz = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, aux, 0, 0, 1)</td>
</tr>
<tr>
<td>merge(a, aux, 2, 2, 3)</td>
</tr>
<tr>
<td>merge(a, aux, 4, 4, 5)</td>
</tr>
<tr>
<td>merge(a, aux, 6, 6, 7)</td>
</tr>
<tr>
<td>merge(a, aux, 8, 8, 9)</td>
</tr>
<tr>
<td>merge(a, aux, 10, 10, 11)</td>
</tr>
<tr>
<td>merge(a, aux, 12, 12, 13)</td>
</tr>
<tr>
<td>merge(a, aux, 14, 14, 15)</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>
**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));
    }
}
```

**Proposition.** At most $n \log_2 n$ compares; $\Theta(n)$ extra space.

**Bottom line.** Simple and non-recursive version of mergesort.
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.
Idea. Exploit pre-existing order by identifying naturally occurring runs.

Tradeoff. Fewer passes vs. extra compares per pass to identify runs.
Timsort (2002)

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

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-earned measure of memory efficiency.

...
Proving that Android’s, Java’s and Python’s sorting algorithm is broken (and showing how to fix it)

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)

<table>
<thead>
<tr>
<th>Details</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong> Bug</td>
<td>Carine Pivoteau wrote:</td>
</tr>
<tr>
<td><strong>Status:</strong> RESOLVED</td>
<td>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 (<a href="http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf">http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf</a>]. 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: <a href="https://arxiv.org/pdf/1805.08612.pdf">https://arxiv.org/pdf/1805.08612.pdf</a>. 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.</td>
</tr>
<tr>
<td><strong>Priority:</strong> P3</td>
<td></td>
</tr>
<tr>
<td><strong>Resolution:</strong> Fixed</td>
<td></td>
</tr>
<tr>
<td><strong>Affects Version/s:</strong> None</td>
<td></td>
</tr>
<tr>
<td><strong>Fix Version/s:</strong> 11</td>
<td></td>
</tr>
<tr>
<td><strong>Component/s:</strong> core-libs</td>
<td></td>
</tr>
<tr>
<td><strong>Labels:</strong> None</td>
<td></td>
</tr>
<tr>
<td><strong>Subcomponent:</strong> java.util: collections</td>
<td></td>
</tr>
<tr>
<td><strong>Introduced In Version:</strong> 6</td>
<td></td>
</tr>
<tr>
<td><strong>Resolved In Build:</strong> b20</td>
<td></td>
</tr>
</tbody>
</table>

[https://bugs.openjdk.java.net/browse/JDK-8203864](https://bugs.openjdk.java.net/browse/JDK-8203864)
### Sorting summary

<table>
<thead>
<tr>
<th>in-place?</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>✔</td>
<td>$n$</td>
<td>$\frac{1}{4}n^2$</td>
<td>$\frac{1}{2}n^2$</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td>$\frac{1}{2}n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$\Theta(n \log n)$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
</tr>
<tr>
<td>?</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
</tr>
</tbody>
</table>

number of compares to sort an array of $n$ elements
2.2 **Mergesort**

- mergesort
- bottom-up mergesort
- sorting complexity
## Computational complexity

A framework to study efficiency of algorithms for solving a particular problem $X$.

<table>
<thead>
<tr>
<th>term</th>
<th>description</th>
<th>example ($X =$ sorting)</th>
</tr>
</thead>
<tbody>
<tr>
<td>model of computation</td>
<td>specifies memory and primitive operations</td>
<td>comparison tree</td>
</tr>
<tr>
<td>cost model</td>
<td>primitive operation counts</td>
<td># compares</td>
</tr>
<tr>
<td>upper bound</td>
<td>cost guarantee provided by some algorithm for a problem</td>
<td>$\sim n \log_2 n$</td>
</tr>
<tr>
<td>lower bound</td>
<td>proven limit on cost guarantee for all algorithms for a problem</td>
<td>?</td>
</tr>
<tr>
<td>optimal algorithm</td>
<td>algorithm with best possible cost guarantee for a problem</td>
<td>?</td>
</tr>
</tbody>
</table>

- Lower bound $\sim$ upper bound

- Can gain knowledge about input only through pairwise compares (e.g., Java's Comparable framework)
Comparison tree (for 3 distinct keys $a$, $b$, and $c$)

- $a < b$
  - yes
  - $b < c$
    - yes
    - $a < c$
      - yes
      - $a < b$
        - yes
        - $a b c$
      - no
      - $a c b$
    - no
    - $a c b$
  - no
  - no
  - $b c a$

- $b < c$
  - yes
  - $a < c$
    - yes
    - $b c a$
    - no
    - $b a c$
  - no
  - $c b a$

height of pruned comparison tree = worst-case number of compares

code between compares (e.g., sequence of exchanges)

one (and only one) reachable leaf corresponds to each each possible ordering
**Proposition.** In the worst case, any compare–based sorting algorithm must make at least 
\( \log_2(n!) \sim n \log_2 n \) compares.

**Pf.**

- Assume array consists of \( n \) distinct values \( a_1 \) through \( a_n \).
- \( n! \) different orderings \( \Rightarrow n! \) reachable leaves.
- Worst–case number of compares = height \( h \) of pruned comparison tree.
- Binary tree of height \( h \) has \( \leq 2^h \) leaves.
Compare-based lower bound for sorting

**Proposition.** In the worst case, any compare–based sorting algorithm must make at least

\[ \log_2(n!) \sim n \log_2 n \] compares.

**Pf.**

- Assume array consists of \( n \) distinct values \( a_1 \) through \( a_n \).
- \( n! \) different orderings \( \Rightarrow \) \( n! \) reachable leaves.
- Worst-case number of compares = height \( h \) of pruned comparison tree.
- Binary tree of height \( h \) has \( \leq 2^h \) leaves.

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

\[ \Rightarrow h \geq \log_2(n!)
\]

\[ \sim n \log_2 n \]

*Stirling’s formula*
Computational complexity

A framework to study efficiency of algorithms for solving a particular problem $X$.

<table>
<thead>
<tr>
<th>term</th>
<th>description</th>
<th>example (X = sorting)</th>
</tr>
</thead>
<tbody>
<tr>
<td>model of computation</td>
<td>specifies memory and primitive operations</td>
<td>comparison tree</td>
</tr>
<tr>
<td>cost model</td>
<td>primitive operation counts</td>
<td># compares</td>
</tr>
<tr>
<td>upper bound</td>
<td>cost guarantee provided by some algorithm for a problem</td>
<td>$\sim n \log_2 n$</td>
</tr>
<tr>
<td>lower bound</td>
<td>proven limit on cost guarantee for all algorithms for a problem</td>
<td>$\sim n \log_2 n$</td>
</tr>
<tr>
<td>optimal algorithm</td>
<td>algorithm with best possible cost guarantee for a problem</td>
<td>mergesort</td>
</tr>
</tbody>
</table>

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

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

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

**Lesson.** Use theory as a guide.

**Ex.** Design sorting algorithm that makes \( \frac{1}{2} n \log_2 n \) compares in worst case?

**Ex.** Design sorting algorithm that makes \( \Theta(n \log n) \) compares and uses \( \Theta(1) \) extra space.
Q. Why doesn’t this Skittles sorter violate the sorting lower bound?

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

Lower bound may not hold if the algorithm can exploit:

- 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 small number of distinct keys. [next lecture]

- The representation of the keys.
  Ex: radix sorts do not make any key compares; they access the data via individual characters/digits.
### Asymptotic notations

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
</tr>
</thead>
</table>
| tilde (~) | leading term | \( \sim \frac{1}{2} n^2 \) | \( \frac{1}{2} n^2 \)  
| | | | \( \frac{1}{2} n^2 + 3n + 22 \)  
| | | | \( \frac{1}{2} n^2 + n \log_2 n \)  
| big Theta (\( \Theta \)) | order of growth | \( \Theta(n^2) \) | \( \frac{1}{2} n^2 \)  
| | | | \( 7n^2 + n^{\frac{1}{2}} \)  
| | | | \( 5n^2 - 3n \)  
| big O (\( O \)) | upper bound | \( O(n^2) \) | \( 10n^2 \)  
| | | | \( 22n \)  
| | | | \( \log_2 n \)  
| big Omega (\( \Omega \)) | lower bound | \( \Omega(n^2) \) | \( \frac{1}{2} n^2 \)  
| | | | \( n^3 + 3n \)  
| | | | \( 2^n \)  

**Warning:** many programmers misuse \( O \) to mean \( \Theta \).
Mergesort quiz 5

Which of the following correctly describes the function $f(n) = 3n^2 + 30n$?

A. $\sim n^2$
B. $\Theta(n)$
C. $O(n^3)$
D. All of the above.
E. None of the above.
Interviewer. Give a formal description of the sorting lower bound for sorting arrays of \( n \) elements.
Mergesort. Makes $\Theta(n \log n)$ compares (and array accesses) in the worst case.

Sorting lower bound. No compare-based sorting algorithm makes fewer than $\Theta(n \log n)$ compares in the worst case.

Divide-and-conquer. Divide a problem into two (or more) subproblems; solve each subproblem independently; combine results.
## Credits

<table>
<thead>
<tr>
<th>Image/Video</th>
<th>Source</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jon von Neumann</td>
<td>IAS / Alan Richards</td>
<td></td>
</tr>
<tr>
<td>Tim Peters</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>Theory vs. Practice</td>
<td>Ela Sjolie</td>
<td></td>
</tr>
<tr>
<td>Skittles Sorting Machine</td>
<td>Rolf R. Bakke</td>
<td></td>
</tr>
<tr>
<td>Fast Skittles Sorting Machine</td>
<td>Kazumichi Moriyama</td>
<td></td>
</tr>
<tr>
<td>Impossible Stamp</td>
<td>Adobe Stock</td>
<td>education license</td>
</tr>
<tr>
<td>Divide-and-Conquer</td>
<td>wallpapercrafter.com</td>
<td></td>
</tr>
<tr>
<td>Mergesort Instructions</td>
<td>IDEA</td>
<td>CC BY-NC-SA 4.0</td>
</tr>
</tbody>
</table>
A final thought