Sorting is one of the fundamental problems in Computer Science

- Provides a chance to look at a variety of algorithms and techniques for analysis.

Choosing the right sorting algorithm will depend on the characteristics of your problem.
All data elements can be stored in memory at the same time

- If we need to sort a very large data set, things get trickier.

Data stored in an array, indexed from $0 \ldots n - 1$, where $n$ is the number of elements

Each element has a key value (accessed with a $\text{key}()$ method)

We can compare keys for $<$, $>$, $=$

For illustration, we will use arrays of integers – though often keys will be strings, dates, or other things that can be compared.
A sorting algorithm is *Stable* if the relative order of duplicates is preserved.

The order of duplicates matters if the *keys* are duplicated, but the *records* are not.

- Bank accounts, student transcripts, payroll records, etc.

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Bob</td>
</tr>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>2</td>
<td>Ed</td>
</tr>
<tr>
<td>1</td>
<td>Amy</td>
</tr>
<tr>
<td>1</td>
<td>Sue</td>
</tr>
<tr>
<td>2</td>
<td>Al</td>
</tr>
<tr>
<td>2</td>
<td>Bud</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amy</td>
</tr>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>1</td>
<td>Sue</td>
</tr>
<tr>
<td>2</td>
<td>Ed</td>
</tr>
<tr>
<td>2</td>
<td>Al</td>
</tr>
<tr>
<td>3</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Bud</td>
</tr>
</tbody>
</table>

A *non*-Stable sort
Separate list into sorted portion, and unsorted portion

Initially, sorted portion contains first element in the list, unsorted portion is the rest of the list

(A list of one element is always sorted)

Repeatedly insert an element from the unsorted list into the sorted list, until the list is sorted
For Insertion Sort

- Running time $\propto$ # of comparisons
- Worst Case:
12-5: $\Theta()$ For Insertion Sort

- Running time $\propto$ # of comparisons
- Worst Case: Inverse sorted list

# of comparisons:
Running time $\propto$ # of comparisons

Worst Case: Inverse sorted list

# of comparisons:

$$\sum_{i=1}^{n-1} i \in \Theta(n^2)$$
Running time $\propto$ # of comparisons

Best Case:
12-8: \( \Theta() \) For Insertion Sort

- Running time \( \propto \) # of comparisons
- Best Case: Sorted List

# of comparisons:
For Insertion Sort

- Running time $\propto$ # of comparisons
- Best Case: Sorted List

# of comparisons: $n - 1$
12-10: Bubble Sort

6. Scan list from the last index to index 0, swapping the smallest element to the front of the list.
6. Scan the list from the last index to index 1, swapping the second smallest element to index 1.
6. Scan the list from the last index to index 2, swapping the third smallest element to index 2.
   ...
6. Swap the second largest element into position \( (n - 2) \)
Running time $\propto$ # of comparisons

Number of Comparisons:
Running time $\propto$ # of comparisons

Number of Comparisons:

$$\sum_{i=1}^{n-1} i \in \Theta(n^2)$$
12-13: Selection Sort

- Scan through the list, and find the smallest element
- Swap smallest element into position 0
- Scan through the list, and find the second smallest element
- Swap second smallest element into position 1
- ... 
- Scan through the list, and find the second largest element
- Swap smallest largest into position $n - 2$
Running time $\propto$ # of comparisons

Number of Comparisons:
12-15: $\Theta()$ for Selection Sort

- Running time $\propto$ # of comparisons
- Number of Comparisons:

$$\sum_{i=1}^{n-1} i \in \Theta(n^2)$$
Insertion sort is fast if a list is “almost sorted”

How can we use this?

△ Do some work to make the list “almost sorted”
△ Run insertion sort to finish sorting the list

Only helps if work required to make list “almost sorted” is less than $n^2$
6 Sort \( \frac{n}{2} \) sublists of length 2, using insertion sort
6 Sort \( \frac{n}{4} \) sublists of length 4, using insertion sort
6 Sort \( \frac{n}{8} \) sublists of length 8, using insertion sort
\ldots
6 Sort 2 sublists of length \( \frac{n}{2} \), using insertion sort
6 Sort 1 sublist of length \( n \), using insertion sort
Shell sort runs several insertion sorts, using increments

- Code on monitor uses “Shell’s Increments”:
  \[ \{n/2, n/4, \ldots, 4, 2, 1\} \]

Problem with Shell’s Increments:

- Various sorts do not interact much
- If all large elements are stored in odd indices, and small elements are stored in even indices, what happens?
12-19: Other Increments

6 Shell’s Increments: \( \{n/2, n/4, \ldots 4, 2, 1\} \)
   △ Running time: \( O(n^2) \)

6 “/3” increments: \( \{n/3, n/9, \ldots , 9, 3, 1\} \)
   △ Running time: \( O(n^{3/2}) \)

6 Hibbard’s Increments: \( \{2^k - 1, 2^{k-1} - 1, \ldots 7, 3, 1\} \)
   △ Running time: \( O(n^{3/2}) \)
Is Insertion sort stable?
Is Bubble Sort stable?
Is Selection Sort stable?
Is Shell Sort stable?
12-21: Stability

- Is Insertion sort stable? Yes!
- Is Bubble Sort stable? Yes!
- Is Selection Sort stable? Yes!
- Is Shell Sort stable? No!

Note that Insertion, Selection and Bubble Sort are stable, as coded in the text and in lecture. It is possible to modify the algorithms so that they still work, but are not stable.