Yardsticks for Comparing Algorithms

- How do we characterize the "goodness" of an algorithm?
Yardsticks for Comparing Algorithms

- How do we characterize the "goodness" of an algorithm?
  - Accuracy of results
Yardsticks for Comparing Algorithms

- How do we characterize the "goodness" of an algorithm?
  - Accuracy of results
  - Space required to run algorithm (*space complexity*)
Yardsticks for Comparing Algorithms

- How do we characterize the "goodness" of an algorithm?
  - Accuracy of results
  - Space required to run algorithm (*space complexity*)
  - Time required to run algorithm (*time complexity*)
Yardsticks for Comparing Algorithms

- How do we characterize the "goodness" of an algorithm?
  - Accuracy of results
  - Space required to run algorithm (*space complexity*)
  - Time required to run algorithm (*time complexity*)
    - We'll focus mainly on time complexity in this course
Yardsticks for Comparing Algorithms

- How do we characterize the "goodness" of an algorithm?
  - Accuracy of results
  - Space required to run algorithm (*space complexity*)
  - Time required to run algorithm (*time complexity*)
    - We'll focus mainly on time complexity in this course

- **Important:** The *time complexity* of an algorithm is the number of comparisons required in terms of the problem size $N$
Outline

- Motivation
- Search algorithms
  - Linear Search
  - Binary Search
- Sorting Algorithms
  - Selection Sort
  - Bubble Sort
    - Optimizations
- More Examples
- Conclusion

Note: These algorithms can be implemented over arrays or ArrayLists. We use Arrays for our lectures due to simplified syntax
Searching within Collections

- We've now seen three *data structures*:
  - Arrays, ArrayList, HashMap
  - More such data structures to come in CS 261
Searching within Collections

- We've now seen three *data structures*:
  - Arrays, ArrayList, HashMap
  - More such data structures to come in CS 261
    - Sets, trees, stacks, queues, heaps...
Searching within Collections

- We've now seen three *data structures*:
  - Arrays, ArrayList, HashMap
  - More such data structures to come in CS 261
    - Sets, trees, stacks, queues, heaps...

- One of the key operations is to find or *search* for things within collections of data:
We've now seen three *data structures*:

- Arrays, ArrayList, HashMap
- More such data structures to come in CS 261
  - Sets, trees, stacks, queues, heaps...

One of the key operations is to find or *search* for things within collections of data:
Given an array (or ArrayList) of elements and a search-key key,

- Returns the index of the 1st occurrence of the key, or -1 if not found

Let's define linearSearch():

- Visits each item in order and checks against the key
  - Might return part way in the loop if key is found!
Time Complexity: Linear Search

- Linear search algorithm:

```java
public static int linearSearch(int[] list, int key) {
    // Need to return the index of the key, so we
    // don't want to use a for-each loop (index-less)
    for (int i = 0; i < list.length; i++) {
        // found the key! return the index and return early!
        if (key == list[i]) {
            return i;
        }
    }
    // didn't find the key! return -1
    return -1;
}
```

- Suppose `list` contains [90, 33, 64, 23, 9, 34, 100, 56]
  - Length of array is 8
  - How many key-comparisons do we need to find 23, 12, and then 90?
Complexity Cases

- We're concerned with the following cases
  - **Best Case**
    - Scenario in which the algorithm requires the fewest number of comparisons to complete running an algorithm.
  - **Worst Case**
    - Scenario in which the algorithm requires the most number of comparisons to complete running an algorithm.
  - **Average Case**
    - The number of comparisons taken in the likeliest scenario.

- Time Complexity of Linear Search for list of size $N$?
Outline

- Motivation for Search and Sort
- Linear Search
- **Binary Search**
- Selection Sort
- Bubble Sort
  - Optimizations
- More Examples
- Conclusion
Does List Order Help?

- Is there a *smarter* way to search if the list is sorted in ascending order?
  - Think: Lots of things are sorted in life... how does that help you search?
Binary Search
Binary Search

- The idea: Exploit the sorted ordering
Binary Search

- The idea: Exploit the sorted ordering
  - Skip to the middle item in the list, and compare with key...
Binary Search

- The idea: Exploit the sorted ordering
  - Skip to the middle item in the list, and compare with key...
  - If not found, then which half might the key be?
Binary Search

Think contact list: Search for the name Kate
Binary Search

Think contact list: Search for the name Kate

Scroll to the name in the middle of the list.
Binary Search

Think contact list: Search for the name *Kate*

Scroll to the *name* in the middle of the list.

1) Is it *Kate*?
Binary Search

Think contact list: Search for the name **Kate**

Scroll to the name in the middle of the list.

1) Is it **Kate**?
   
   (a) If so, you're done!
Think contact list: Search for the name *Kate*

Scroll to the *name* in the middle of the list.

1) Is it *Kate*?
   (a) If so, you're done!
2) If not, does *Kate precede* the *name* on the page?
   Repeat on first half of the contact list
Think contact list: Search for the name *Kate*

Scroll to the name in the middle of the list.

1) Is it *Kate*?
   
   (a) If so, you're done!

2) If not, does *Kate* precede the name on the page?
   
   Repeat on first half of the contact list

3) Otherwise,
   
   Repeat on second half of the contact list
Binary Search

Think contact list: Search for the name *Kate*

Scroll to the *name* in the middle of the list.

1) Is it *Kate*?
   (a) If so, you're done!
2) If not, does *Kate precede* the *name* on the page?
   Repeat on first half of the contact list
3) Otherwise,
   Repeat on second half of the contact list
4) Repeat until you run find *Kate*, or run out of names
Binary Search

- The idea: Exploit the sorted ordering

Think contact list: Search for the name *Kate*

Scroll to the name in the middle of the list.

1) Is it *Kate*?
   (a) If so, you're done!
2) If not, does *Kate* precede the name on the page?
   Repeat on first half of the contact list
3) Otherwise,
   Repeat on second half of the contact list
4) Repeat until you run find *Kate*, or run out of names
The idea: Exploit the sorted ordering

- Skip to the middle item in the list, and compare with key...

Think contact list: Search for the name **Kate**

Scroll to the name in the middle of the list.

1) Is it **Kate**?
   (a) If so, you're done!
2) If not, does **Kate precede** the name on the page?
   Repeat on first half of the contact list
3) Otherwise,
   Repeat on second half of the contact list
4) Repeat until you run find **Kate**, or run out of names
Binary Search

- The idea: Exploit the sorted ordering
  - Skip to the middle item in the list, and compare with key...
  - If not found, then which half might the key be?

Think contact list: Search for the name **Kate**

Scroll to the **name** in the middle of the list.

1) Is it **Kate**?
   (a) If so, you're done!
2) If not, does **Kate precede** the **name** on the page?
   Repeat on *first* half of the contact list
3) Otherwise,
   Repeat on *second* half of the contact list
4) Repeat until you run find **Kate**, or run out of names
Remember, the list must already be in *ascending order* for binary search to work.
Search for key = 10

- Step 1: Initialize the positions of the "book ends"

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

length: 9

Comparisons: 0
Search for key = 10

- Step 2: Determine the position of the *mid-point* between book ends
  - Compare this element with the key!

```plaintext
- list
  - length: 9
  - 3 4 6 7 9 10 12 13 15
  - left (0)
  - mid (4)
    From: (0+8)/2
  - right (8)
```

Comparisons: 1
Step 3: In this case, \texttt{key} > \texttt{list[mid]}

- If \texttt{key} is indeed in the list, it \textit{must} be in the right half

\begin{itemize}
  \item [3] \texttt{key} = 10
\end{itemize}

Comparisons: 1
Step 3: In this case, `key > list[mid]`

- If `key` is indeed in the list, it *must* be in the right half
- Eliminate left half by moving left "book end" over

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

length: 9

Comparisons: 1
Search for key = 10

- Step 2 (repeated): Determine the position of the *mid-point* between book ends
  - Compare this element with the key!

Comparisons: 2
Search for key = 10

- Step 3 (repeated): In this case, \texttt{key} < \texttt{list[mid]}
  - If \texttt{key} is indeed in the list, it \textit{must} be in the left half

Comparisons: 2
Step 3 (repeated): In this case, $key < list[mid]$

- If $key$ is indeed in the list, it must be in the left half
- Eliminate right half by moving right "book end" over

Comparisons: 2
Step 2: Determine the position of the *mid-point* between book ends

• Compare this element with the key!

**list**

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

*Comparisons: 3*
Step 3 (repeated): In this case, key == list[mid]

- Return mid (5), the position in which the key was found
  - (Method terminates.)

Search for key = 10

Comparisons: 3
Another Example

- What if the key wasn't in the list?
  - Do a trace for key = 5
public static int binarySearch(int[] list, int key) {
    int left = 0;
    int right = list.length - 1;

    while (left <= right) {
        // compute midpoint
        int mid = (left + right) / 2;
        if (key == list[mid]) {
            return mid;  // found the key! Return early
        } else if (key > list[mid]) {
            left = mid + 1;  // shift 'left' edge to the right
        } else {
            right = mid - 1;  // shift 'right' edge to the left
        }
    }
    // didn't find the key! return -1
    return -1;
}
Let's try to analyze the best and worst case scenarios for binary search:

- **Best case is observed when...?**
  - Estimated number of comparisons?

- **Worst case is observed when...?**
  - Estimated number of comparisons?
  - Intuition: You have N elements in the list
  - The list size shrinks by half after each comparison
  - Let's visualize the worst case on the board

*Average case? (Next Lab)*
Binary Search Summary

- Let's compare to Linear Search

<table>
<thead>
<tr>
<th>Worst Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>List size: $n$</td>
</tr>
<tr>
<td>Linear Search: $n$</td>
</tr>
<tr>
<td>Binary Search: $\log_2(n)$</td>
</tr>
</tbody>
</table>

![Graph comparing Linear Search and Binary Search](image)
Binary Search Summary

- Let's compare to Linear Search

<table>
<thead>
<tr>
<th>Worst Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>List size: ( n )</td>
</tr>
<tr>
<td>Linear Search: ( n )</td>
</tr>
<tr>
<td>Binary Search: ( \log_2(n) )</td>
</tr>
</tbody>
</table>

| 1 | 1 | 1 |

Graph showing comparison of Linear Search and Binary Search for worst case time complexity.
Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: ( n )</th>
<th>Linear Search: ( n )</th>
<th>Binary Search: ( \log_2(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>
Binary Search Summary

- Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: $n$</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
</tbody>
</table>
Binary Search Summary

Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: $n$</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
</tbody>
</table>
Binary Search Summary

- Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: ( n )</th>
<th>Linear Search: ( n )</th>
<th>Binary Search: ( \log_2(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
</tbody>
</table>
### Binary Search Summary

- Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: $n$</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
</tbody>
</table>

### Worst Case Time Complexity

![Graph showing comparisons for Linear Search and Binary Search]
Binary Search Summary

Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: $n$</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
</tbody>
</table>
Binary Search Summary

Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: $n$</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100,000,000</td>
<td>27</td>
</tr>
</tbody>
</table>

Worst Case Time Complexity

![Graph showing linear and binary search comparisons](image-url)
Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: ( n )</th>
<th>Linear Search: ( n )</th>
<th>Binary Search: ( \log_2(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100,000,000</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>
Binary Search Summary

- Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: $n$</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100,000,000</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>

- Why would we ever use linear search?
Binary Search Summary

- Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size: $n$</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100,000,000</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>

- Why would we ever use linear search?
  - Think phone book again. What if I only had a phone number, but I wanted to find the name to whom it belongs.
Is This What Google Does?

Puget Sound - Wikipedia, the free encyclopedia

Puget Sound /ˈpuːʒət sɔːnd/ is a sound along the northwestern coast of the U.S. state of Washington, an inlet of the Pacific Ocean, and part of the Salish Sea.

Sound - Puget Sound region - Salish Sea - Puget Sound AVA

University of Puget Sound

www.pugetsound.edu/ University of Puget Sound
Admissions, student information, faculty and staff, alumni, news, events, athletics.
Located in Tacoma, WA.
Admission - Departments & Programs - Academics - Employment

Puget Sound Energy

https://www.pse.com/ Puget Sound Energy
A regulated utility, providing electric and natural gas service to the Puget Sound region.
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
- More Array Examples
- Conclusion

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
"Make the Common Case Fast!"

- Why do you sort things?
  - Your wardrobe...
  - Hand of playing cards...
  - Contacts in your phone...
  - Organizing books on your shelf...
"Make the Common Case Fast!"

- Why do you sort things?
  - Your wardrobe...
  - Hand of playing cards...
  - Contacts in your phone...
  - Organizing books on your shelf...

- Core CS (tenet) tenet: "Make the common case fast!"
  - Search is one of the most frequently-used operations in life
    - And certainly in computing
Problem: Given an array of integers, put the list in *ascending* order, that is:

\[
\text{list}[0] \leqslant \text{list}[1] \leqslant \text{list}[2] \leqslant \ldots \leqslant \text{list}[\text{list.length}-1]
\]
Sorting

- Problem: Given an array of integers, put the list in *ascending* order, that is:

\[
\text{list[0] \leq list[1] \leq list[2] \leq \ldots \leq list[list.length-1]}
\]
Selection Sort

- Observe: at any point, a list is partitioned into:
  - A sorted sublist (blue) and an unsorted sublist (red)
    - Partitioned at index $i$ (which points to the first item in the unsorted sublist)

- Algorithm: While $i$ has not passed the end of the list:
  - Find the index of the smallest item in the unsorted sublist
  - Swap it with the first item in the unsorted sublist at index $i$
  - Increment $i$ (increasing sorted sublist)
Selection Sort (Step-By-Step)

- **minIdx**
  - 19 13 12 7

- **i**
  - Index of the first element in unsorted sublist

- **minIdx**
  - Index of smallest item in unsorted sublist

- The item currently being compared

- Sorted
- Unsorted
Selection Sort (Step-By-Step)

- **minIdx**
  - 19 13 12 7
  - \( i \) = 0

- **Sorted**

- **Unsorted**
  - 19 13 12 7
  - \( j \) = \( \minIdx \) + 1

- **Index of the first element in unsorted sublist**: \( i \)
- **Index of smallest item in unsorted sublist**: \( \minIdx \)
- **The item currently being compared**
Selection Sort (Step-By-Step)

- minIdx
  - i
- Sorted
  - Unssorted
- Index of the first element in unsorted sublist
- Index of smallest item in unsorted sublist
- The item currently being compared
Selection Sort (Step-By-Step)

Index of smallest item in unsorted sublist

Index of the first element in unsorted sublist

The item currently being compared
Selection Sort (Step-By-Step)

1. The item currently being compared is 7.
2. The index of the first element in the unsorted sublist is i = 0.
3. The index of the smallest item in the unsorted sublist is minIdx = 0.
4. The item currently being compared is 7.
5. The item is moved to the sorted sublist.

Sorted: [19, 13, 12, 7]
Unsorted: []
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist
- **Sorted**: Blue
- **Unsorted**: Red
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared
Selection Sort (Step-By-Step)

1. **minIdx**
   - 19 13 12 7
   - \(i\)

2. **minIdx**
   - 19 13 12 7
   - \(i\) \(j\)

3. **minIdx**
   - 19 13 12 7
   - \(i\) \(j\)

4. **minIdx**
   - 19 13 12 7
   - \(i\) \(j\)

**MinIdx**

- Index of smallest item in unsorted sublist

**i**

- Index of the first element in unsorted sublist

**Sorted**

- Blue

**Unsorted**

- Red

**The item currently being compared**

- Green
Selection Sort (Step-By-Step)

- **minIdx**
  - 19 13 12 7
  - i

- **minIdx**
  - 19 13 12 7
  - i j

- **minIdx**
  - 19 13 12 7
  - i j

- **minIdx**
  - 7 13 12 19
  - i j

- **sorted**
  - Index of the first element in unsorted sublist

- **minIdx**
  - Index of smallest item in unsorted sublist

- **item compared**
  - The item currently being compared
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared

0. Start with the first element.
1. Compare each element to the current minimum.
2. If a new minimum is found, update minIdx.
3. Move to the next element and repeat until the end.
4. Repeat the process for each sublist until sorted.
Selection Sort (Step-By-Step)

- **minIdx**
  - Initial: 19 13 12 7
  - After 1st iteration: 7 13 12 19
  - After 2nd iteration: 7 13 12 19
  - After 3rd iteration: 7 13 12 19

- **i**: Index of the first element in unsorted sublist
- **j**: Index of smallest item in unsorted sublist
- **Sorted**: The item currently being compared
- **Unsorted**: The item currently being compared

The algorithm works by repeatedly finding the minimum element from the unsorted part of the list and putting it at the beginning. This process is repeated until the entire list is sorted.
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist.
- **Sorted**: Elements that have been sorted.
- **Unsorted**: Elements that have not been sorted.
- **i**: Index of the first element in unsorted sublist.
- **j**: The item currently being compared.

1. **Initial State**
   - Unsorted array: 19, 13, 12, 7
   - Sorted: Empty

2. **First Iteration**
   - Find the minimum index: 7
   - Swap 7 with 19
   - New array: 7, 13, 12, 19

3. **Second Iteration**
   - Find the minimum index: 7
   - Swap 7 with 19
   - New array: 13, 12, 19, 7

4. **Third Iteration**
   - Find the minimum index: 7
   - Swap 7 with 12
   - New array: 13, 12, 19, 7
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist
- **Sorted**: Blue
- **Unsorted**: Red
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared

1. **Step 1**:
   - Unsorted: 19 13 12 7
   - Sorted: 
   - minIdx: 19

2. **Step 2**:
   - Unsorted: 7 13 12 19
   - Sorted: 19
   - minIdx: 7

3. **Step 3**:
   - Unsorted: 7 13 12 19
   - Sorted: 7 19
   - minIdx: 7

4. **Step 4**:
   - Unsorted: 7 13 12 19
   - Sorted: 7 13 12 19
   - minIdx: 7

The process continues until the list is fully sorted.
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist
- **Sorted**: Blue
- **Unsorted**: Red
- **i**: Index of the first element in unsorted sublist
- **minIdx**: Index of smallest item in unsorted sublist
- **The item currently being compared**: Empty

**Steps:**
1. Initialize `minIdx` to the index of the first element.
2. Check each element from the unsorted sublist.
3. If a smaller element is found, update `minIdx`.
4. Swap the smallest element found with the first element of the unsorted sublist.
5. Increment the index `i`.
6. Repeat steps 2-5 until the entire list is sorted.
Selection Sort (Step-By-Step)

- **minIdx**
  - Initially, `minIdx` is set to the index of the first element in the unsorted sublist.
  - As the algorithm progresses, `minIdx` is updated to the index of the smallest element in the unsorted sublist.

- **i**
  - `i` is the index of the first element in the unsorted sublist.

- **j**
  - `j` is the index of the element currently being compared.

- **Sorted**
  - Elements in the sorted sublist.

- **Unsorted**
  - Elements in the unsorted sublist.

- **The item currently being compared**
  - The element at index `j` is being compared against the element at index `minIdx`.

- **Index of the first element in unsorted sublist**
  - `i` is the index of the first element in the unsorted sublist.

- **Index of smallest item in unsorted sublist**
  - `minIdx` is the index of the smallest item in the unsorted sublist.
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist
- **Sorted**: Unsorted
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared

```
minIdx

19 13 12 7
i

minIdx

19 13 12 7
i j

minIdx

19 13 12 7
i j

minIdx

19 13 12 7
i j

minIdx

19 13 12 7
i j

minIdx

19 13 12 7
i j

minIdx

7 13 12 19
i

minIdx

7 13 12 19
i j

minIdx

7 12 13 19
i j
```

**i**

- Index of the first element in unsorted sublist

**minIdx**

- Index of smallest item in unsorted sublist

**The item currently being compared**
Selection Sort (Step-By-Step)

1. \( \text{minIdx} \): Index of the first element in unsorted sublist
2. \( i \): Index of the first element in unsorted sublist
3. \( j \): Index of the smallest item in unsorted sublist
4. \( \text{minIdx} \): Index of smallest item in unsorted sublist
5. \( \) The item currently being compared

Sorter:

- \( \text{Sorted} \)
- \( \text{Unsorted} \)
Selection Sort (Step-By-Step)

- **minIdx**
  - Initial array: 19 13 12 7
  - Index of the first element in unsorted sublist: i

- Find the minimum value in the current sublist
  - Current sublist: 19 13 12 7
  - minIdx changes to the index of the minimum value

- Swap the minimum value with the first element of the sublist
  - Sorted: 7 13 12 19
  - Unsorted: 19 13 12

- Increment the index i
  - i = 1
  - j = i + 1

- Repeat the process for the next sublist
  - Current sublist: 7 13 12
  - minIdx changes to the index of the minimum value

- Swap the minimum value with the first element of the sublist
  - Sorted: 7 12 13 19
  - Unsorted: 19

- Increment the index i
  - i = 3

- Repeat the process for the next sublist
  - Current sublist: 7 19

- Swap the minimum value with the first element of the sublist
  - Sorted: 7 12 13 19
  - Unsorted: 19

- The item currently being compared

**Key Points**
- **Sorted**: Elements that have been sorted.
- **Unsorted**: Elements that are yet to be sorted.
- **minIdx**: Index of smallest item in unsorted sublist.
- **Index of the first element in unsorted sublist**: i
- **The item currently being compared**: j
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared

Sorted

Unsorted

CSCI 161: Introduction to Computer Science - 9 - Searching and Sorting Algorithms
Selection Sort (Step-By-Step)

minIdx

| 19 | 13 | 12 | 7 |

i

minIdx

| 19 | 13 | 12 | 7 |

i j

minIdx

| 19 | 13 | 12 | 7 |

i j

minIdx

| 19 | 13 | 12 | 7 |

i j

minIdx

| 7 | 13 | 12 | 19 |

i

minIdx

| 7 | 13 | 12 | 19 |

i j

minIdx

| 7 | 12 | 13 | 19 |

i

minIdx

| 7 | 12 | 13 | 19 |

i j

minIdx

| 7 | 12 | 13 | 19 |

i j

minIdx

| 7 | 12 | 13 | 19 |

i j

minIdx

| 7 | 12 | 13 | 19 |

i j

minIdx

| 7 | 12 | 13 | 19 |

i j

sorted: **Sorted**
unsorted: **Unsorted**
i: Index of the first element in unsorted sublist
minIdx: Index of smallest item in unsorted sublist

| i | j |

The item currently being compared
Selection Sort (Step-By-Step)

Sorted

Unsorted

The item currently being compared

Index of smallest item in unsorted sublist

Index of the first element in unsorted sublist

The item currently being compared
Selection Sort (Step-By-Step)

- **minIdx**: Index of smallest item in unsorted sublist
- **Sorted**: Green box
- **Unsorted**: Red box
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared

```
minIdx
19 13 12 7
i

minIdx
19 13 12 7
i  j

minIdx
19 13 12 7
i  j

minIdx
19 13 12 7
i  j

minIdx
19 13 12 7
i  j

minIdx
7 13 12 19
i

minIdx
7 13 12 19
i  j

minIdx
7 13 12 19
i  j

minIdx
7 13 12 19
i  j

minIdx
7 13 12 19
i  j

minIdx
7 12 13 19
i

minIdx
7 12 13 19
i  j

minIdx
7 12 13 19
i  j

minIdx
7 12 13 19
i  j

minIdx
7 12 13 19
i  j
```
Selection Sort (Step-By-Step)

- **minIdx**
  - Index of smallest item in unsorted sublist
- **Sorted**
  - Items that have been sorted
- **Unsorted**
  - Items that remain unsorted
- **i**
  - Index of the first element in unsorted sublist
- **j**
  - The item currently being compared

Diagram:

1. Initialize `minIdx` with the index of the first element `i`.
2. Compare `j` with `minIdx`.
3. If `j` is smaller, update `minIdx` to `j`.
4. Swap `minIdx` with the current element `j`.
5. Increment `i` and repeat until the end of the list.

Example:

```
Original: 7 12 13 19
Sorted:   7 12 13 19
```
Selection Sort (Step-By-Step)

- **minIdx**
  - Index of the first element in unsorted sublist
- **minIdx**
  - Index of smallest item in unsorted sublist
- **i**
  - Index of the first element in unsorted sublist
- **j**
  - The item currently being compared
- **Sorted**
- **Unsorted**
Selection Sort (Step-By-Step)

- **minIdx**
  - Index of smallest item in unsorted sublist

- **Sorted**
  - Blue

- **Unsorted**
  - Red

- **i**
  - Index of the first element in unsorted sublist

- **j**
  - The item currently being compared

- **minIdx**
  - Index of smallest item in unsorted sublist

- **Unsorted**
  - Red

- **Sorted**
  - Blue
Selection Sort (Step-By-Step)

- **minIdx**
  - 19 13 12 7
  - i

- **minIdx**
  - 19 13 12 7
  - i j

- **minIdx**
  - 19 13 12 7
  - i j

- **minIdx**
  - 7 13 12 19
  - i

- **minIdx**
  - 7 13 12 19
  - i j

- **minIdx**
  - 7 13 12 19
  - i j

- **minIdx**
  - 7 13 12 19
  - i j

- **minIdx**
  - 7 12 13 19
  - i

- **minIdx**
  - 7 12 13 19
  - i j

- **minIdx**
  - 7 12 13 19
  - i j

- **minIdx**
  - 7 12 13 19
  - i j

- **minIdx**
  - 7 12 13 19
  - i j

- **minIdx**
  - 7 12 13 19
  - i

- **Done!**

- **Sorted**
- **Unsorted**

- **i**
  - Index of the first element in unsorted sublist

- **minIdx**
  - Index of smallest item in unsorted sublist

- **The item currently being compared**
Selection Sort (Cont.)

```java
public static void selectionSort(int[] list) {
    //i is the index of the first item of the unsorted sublist
    for (int i = 0; i < list.length; i++) {
        //
    }
}
```

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
public static void selectionSort(int[] list) {
    // i is the index of the first item of the unsorted sublist
    for (int i = 0; i < list.length; i++) {
        int minIdx = i; // index of minimum item found so far in unsorted sublist
    }
}

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
public static void selectionSort(int[] list) {
    // i is the index of the first item of the unsorted sublist
    for (int i = 0; i < list.length; i++) {
        int minIdx = i; // index of minimum item found so far in unsorted sublist
        for (int j = i+1; j < list.length; j++) {
            if (list[j] < list[minIdx]) {
                // found a smaller item at list[j], so update minIdx to j
                minIdx = j;
            }
        }
    }
    // Check out this link for animation (note animation builds list backwards): http://cs.pugetsound.edu/~aasmith/sorters/
public static void selectionSort(int[] list) {
    //i is the index of the first item of the unsorted sublist
    for (int i = 0; i < list.length; i++) {
        int minIdx = i; //index of minimum item found so far in unsorted sublist
        for (int j = i+1; j < list.length; j++) {
            if (list[j] < list[minIdx]) {
                //found a smaller item at list[j], so update minIdx to j
                minIdx = j;
            }
        }
        //swap the two items at index i and minIdx
        int temp = list[i];
        list[i] = list[minIdx];
        list[minIdx] = temp;
    }
}

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
What if Data Was Sorted Already?

- **minIdx**
  - 7
  - 12
  - 13
  - 19

- i
- Sorted sublist
- Unsorted sublist
- Index of the first element in unsorted sublist
- Index of smallest item so far in unsorted sublist
- The item currently being compared
(What if Data Was Sorted Already?)

- **minIdx**
  - 7 12 13 19
- **i**
  - 7 12 13 19
  - i j

- **Sorted sublist**
- **Unsorted sublist**
- **i** Index of the first element in unsorted sublist
- **minIdx** Index of smallest item so far in unsorted sublist
- **The item currently being compared**
(What if Data Was Sorted Already?)

minIdx

7 12 13 19

i

minIdx

7 12 13 19

i  j

Sorted sublist

Unsorted sublist

i  Index of the first element in unsorted sublist

minIdx  Index of smallest item so far in unsorted sublist

The item currently being compared
(What if Data Was Sorted Already?)

- **minIdx**
  - 7 12 13 19

- **i**, **j**

- **Sorted sublist**
- **Unsorted sublist**
- **minIdx**
  - Index of the first element in unsorted sublist
- **minIdx**
  - Index of smallest item so far in unsorted sublist
- **The item currently being compared**
(What if Data Was Sorted Already?)

- minIdx
- $i$
- $j$
- Sorted sublist
- Index of the first element in unsorted sublist
- unIdx
- Index of smallest item so far in unsorted sublist
- The item currently being compared
(What if Data Was Sorted Already?)

- **minIdx**
  - 7 12 13 19
  - i

- **minIdx**
  - 7 12 13 19
  - i  j

- **minIdx**
  - 7 12 13 19
  - i  j

- **Sorted sublist**
  - [7, 12, 13, 19]

- **Unsorted sublist**
  - [i, j]

- **minIdx**
  - Index of smallest item so far in unsorted sublist

- **i**
  - Index of the first element in unsorted sublist

- **The item currently being compared**
(What if Data Was Sorted Already?)

- **minIdx**: Index of the first element in unsorted sublist
- **i**: Index of smallest item so far in unsorted sublist
- **j**: The item currently being compared

```
minIdx

7 12 13 19

i

minIdx

7 12 13 19

i  j

minIdx

7 12 13 19

i  j
```
(What if Data Was Sorted Already?)

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>13</th>
<th>12</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Sorted sublist**
- **Unsorted sublist**
- **minIdx**: Index of smallest item so far in unsorted sublist
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared
(What if Data Was Sorted Already?)

**minIdx**

| 7 | 12 | 13 | 19 |

i

**minIdx**

| 7 | 12 | 13 | 19 |

i  j

**minIdx**

| 7 | 12 | 13 | 19 |

i  j

**minIdx**

| 7 | 13 | 12 | 19 |

j

- **Sorted sublist**
- **Unsorted sublist**
- **i** Index of the first element in unsorted sublist
- **minIdx** Index of smallest item so far in unsorted sublist
- **j** The item currently being compared
(What if Data Was Sorted Already?)

minIdx

7 12 13 19

minIdx

7 12 13 19

i

minIdx

7 12 13 19

i

j

minIdx

7 12 13 19

i

j

minIdx

7 12 13 19

i

j

Sorted sublist

Unsorted sublist

i

Index of the first element in unsorted sublist

minIdx

Index of smallest item so far in unsorted sublist

The item currently being compared
(What if Data Was Sorted Already?)

- minIdx
  - 7 12 13 19
    - i
- minIdx
  - 7 12 13 19
    - i  j
- minIdx
  - 7 12 13 19
    - i  j
- minIdx
  - 7 12 13 19
    - i  j

- Sorted sublist
- Unsorted sublist
- i
  - Index of the first element in unsorted sublist
- minIdx
  - Index of smallest item so far in unsorted sublist
- The item currently being compared
(What if Data Was Sorted Already?)

---

**minIdx**

![Diagram showing elements 7, 12, 13, and 19 with indices i and j.](image)

---

**minIdx**

![Diagram showing elements 7, 12, 13, and 19 with indices i and j.](image)

---

**minIdx**

![Diagram showing elements 7, 12, 13, and 19 with indices i and j.](image)

---

**minIdx**

![Diagram showing elements 7, 12, 13, and 19 with indices i and j.](image)

---

**minIdx**

![Diagram showing elements 7, 12, 13, and 19 with indices i and j.](image)
(What if Data Was Sorted Already?)

- **minIdx**: Index of the first element in unsorted sublist
- **minIdx**: Index of smallest item so far in unsorted sublist
- **Sorted sublist**
- **Unsorted sublist**
- **i**: Index of the first element in unsorted sublist
- **j**: The item currently being compared
What if Data Was Sorted Already?

- **minIdx**: Index of the first element in unsorted sublist
- **Sorted sublist**: The item currently being compared
- **Unsorted sublist**: Index of smallest item so far in unsorted sublist
- **The item currently being compared**

The diagram illustrates the process of searching or sorting an array, with elements highlighted to show the progression of the algorithm.
(What if Data Was Sorted Already?)

```
minIdx

7 12 13 19
i

minIdx

7 12 13 19
i j

minIdx

7 12 13 19
i j

minIdx

7 12 13 19
i j

Sorted sublist

Unsorted sublist

Index of the first element in unsorted sublist

Index of smallest item so far in unsorted sublist

The item currently being compared
```
(What if Data Was Sorted Already?)

Sorted sublist

Unsorted sublist

Index of the first element in unsorted sublist

Index of smallest item so far in unsorted sublist

The item currently being compared
(What if Data Was Sorted Already?)

- **minIdx**
  - **7 12 13 19**
  - **i**

- **minIdx**
  - **7 12 13 19**
  - **i j**

- **minIdx**
  - **7 12 13 19**
  - **i j**

- **minIdx**
  - **7 12 13 19**
  - **i j**

- **minIdx**
  - **7 12 13 19**
  - **j**

**Sorted sublist**
- Index of the first element in unsorted sublist

**minIdx**
- Index of smallest item so far in unsorted sublist

**The item currently being compared**

**Unsorted sublist**
**What if Data Was Sorted Already?**

- **minIdx**
  - Index of the first element in unsorted sublist
  - Index of smallest item so far in unsorted sublist
  - The item currently being compared

- **Sorted sublist**
  - **Unsorted sublist**

- **i**
  - Index of the first element in unsorted sublist

- **j**
  - The item currently being compared
(What if Data Was Sorted Already?)

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minIdx</th>
<th>7</th>
<th>12</th>
<th>13</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Sorted sublist**
- **Unsorted sublist**
- **Index of the first element in unsorted sublist**
- **minIdx** Index of smallest item so far in unsorted sublist
- **The item currently being compared**
(What if Data Was Sorted Already?)

Sorted sublist

Unsorted sublist

Index of the first element in unsorted sublist

Index of smallest item so far in unsorted sublist

The item currently being compared
(What if Data Was Sorted Already?)

**Sorted sublist**

**Unsorted sublist**

**minIdx**

Index of the first element in unsorted sublist

**minIdx**

Index of smallest item so far in unsorted sublist

**The item currently being compared**

---

CSCI 161: Introduction to Computer Science - 9 - Searching and Sorting Algorithms
(What if Data Was Sorted Already?)

- **minIdx**: Index of smallest item so far in unsorted sublist
- **Sorted sublist**: The item currently being compared
- **Unsorted sublist**: Index of the first element in unsorted sublist
- **The item currently being compared**:
(What if Data Was Sorted Already?)

- `minIdx` is the index of the first element in the unsorted sublist.
- `minIdx` is the index of the smallest item so far in the unsorted sublist.
- The item currently being compared.
- Sorted sublist
- Unsorted sublist
- Index of the first element in the unsorted sublist
- The item currently being compared.
(What if Data Was Sorted Already?)

- Sorted sublist
- Index of the first element in unsorted sublist
- minIdx
- Index of smallest item so far in unsorted sublist
- The item currently being compared
(What if Data Was Sorted Already?)

```plaintext
minIdx

7 12 13 19  
  i

minIdx

7 12 13 19  
  i  j

minIdx

7 12 13 19  
  i

minIdx

7 12 13 19  
  i  j

minIdx

7 12 13 19  
  i

minIdx

7 12 13 19  
  i  j

minIdx

7 12 13 19  
  i

minIdx

7 12 13 19  
  i

minIdx

7 12 13 19  
  i

minIdx

7 12 13 19  
  i  j
```

Sorted sublist

Unsorted sublist

Index of the first element in unsorted sublist

Index of smallest item so far in unsorted sublist

The item currently being compared

Done!
Selection Sort Complexity

- Selection sort
  - Does it matter if the list was already sorted or unsorted?
    - Nope. This is in contrast to an algorithm like `linearSearch()`
  - Always takes the same number of comparisons no matter what data is stored inside the list.
  - Complexity: \( \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \)

- Selection sort has a *consistent* time-complexity
  - *i.e.*, Best/worst/average-case complexities are the same
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
- More Examples
- Conclusion

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
Bubble Sort

- A list has: unsorted sublist (red) and a sorted sublist (blue)
  - Counter \( i \) tracks the number of elements that are in sorted sublist
    - The sorted sublist is empty initially (so, initially \( i = 0 \))
    - Compare each pair of adjacent items (at \( j \) and \( j-1 \)) in unsorted sublist
      - Swap if they're out of order
  - Continue until \( i \) increments to the size of the list
Bubble Sort Algorithm

- Algorithm: Assume list is defined as a field

```java
public static void bubbleSort(int[] list) {
    //counts the number of elements that are in the sorted sublist
    for (int i = 0; i < list.length; i++) {
    }
}
```
Bubble Sort Algorithm

- Algorithm: Assume list is defined as a field

```java
public static void bubbleSort(int[] list) {
    //counts the number of elements that are in the sorted sublist
    for (int i = 0; i < list.length; i++) {
        for (int j = 1; j < list.length - i; j++) {
            if (list[j-1] > list[j]) {
                //swap the two adjacent items if out of order (bubble up)
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
            }
        }
    }
}
```
public static void bubbleSort(int[] list) {
    //counts the number of elements that are in the sorted sublist
    for (int i = 0; i < list.length; i++) {

    }

} Stop when j falls off the unsorted sublist.
(Everything to the right of j is already sorted!)
public static void bubbleSort(int[] list) {
    //counts the number of elements that are in the sorted sublist
    for (int i = 0; i < list.length; i++) {
        for (int j = 1; j < list.length - i; j++) {
            if (list[j - 1] > list[j]) {
                //swap the two adjacent items if out of order (bubble up)
                int temp = list[j - 1];
                list[j - 1] = list[j];
                list[j] = temp;
            }
        }
    }
}

Stop when j falls off the unsorted sublist.
(Everything to the right of j is already sorted!)
Bubble Sort Example (Step-by-Step)

A comparison made between these items
Bubble Sort Example (Step-by-Step)

i = 0

A comparison made between these items
Bubble Sort Example (Step-by-Step)

\[ i = 0 \]

\[
\begin{array}{cccc}
19 & 12 & 13 & 7 \\
\end{array}
\]

A comparison made between these items
Bubble Sort Example (Step-by-Step)

i = 0

19 12 13 7

j-1 j

12 19 13 7

j-1 j

A comparison made between these items
Bubble Sort Example (Step-by-Step)

\[ i = 0 \]

\[
\begin{array}{cccc}
19 & 12 & 13 & 7 \\
\end{array}
\]

A comparison made between these items
Bubble Sort Example (Step-by-Step)

\[ i = 0 \]

\[
\begin{array}{cccc}
\text{j-1} & \text{j} & \text{19} & \text{12} & \text{13} & \text{7} \\
\text{12} & \text{19} & \text{13} & \text{7} \\
\text{12} & \text{19} & \text{13} & \text{7} \\
\text{12} & \text{13} & \text{19} & \text{7} \\
\end{array}
\]

A comparison made between these items
Bubble Sort Example (Step-by-Step)

\[ i = 0 \]

\[ \begin{array}{cccc}
19 & 12 & 13 & 7 \\
j-1 & j & & \\
\end{array} \]

\[ \begin{array}{cccc}
12 & 19 & 13 & 7 \\
j-1 & j & & \\
\end{array} \]

\[ \begin{array}{cccc}
12 & 19 & 13 & 7 \\
j-1 & j & & \\
\end{array} \]

\[ \begin{array}{cccc}
12 & 13 & 19 & 7 \\
j-1 & j & & \\
\end{array} \]

\[ \begin{array}{cccc}
12 & 13 & 19 & 7 \\
j-1 & j & & \\
\end{array} \]

\[ \begin{array}{cccc}
12 & 13 & 7 & 19 \\
j-1 & j & & \\
\end{array} \]

A comparison made between these items
Bubble Sort Example (Step-by-Step)

i = 0

A comparison made between these items
Bubble Sort Example (Step-by-Step)

A comparison made between these items
Bubble Sort Example (Step-by-Step)

i = 0

i = 1

A comparison made between these items

CSCI 161: Introduction to Computer Science - 9 - Searching and Sorting Algorithms
Bubble Sort Example (Step-by-Step)

i = 0

19 12 13 7
j-1 j

12 19 13 7
j-1 j

12 13 19 7
j-1 j

12 13 19 7
j-1 j

12 13 7 19
j-1 j

12 7 13 19
j-1 j

j (stop!)

i = 1

12 13 7 19
j-1 j

12 13 7 19
j-1 j

12 7 13 19
j-1 j

A comparison made between these items
Bubble Sort Example (Step-by-Step)

A comparison made between these items
Bubble Sort Example (Step-by-Step)

Bubble Sort Example (Step-by-Step)

A comparison made between these items
Bubble Sort Example (Step-by-Step)

i = 0

19 12 13 7

j-1  j

12 19 13 7

j-1  j

12 13 19 7

j-1  j

12 13 19 7

j-1  j

12 13 7 19

j (stop!)

i = 1

12 13 7 19

j-1  j

12 13 7 19

j-1  j

12 7 13 19

j-1  j

12 7 13 19

j-1  j

A comparison made between these items

i = 2

12 7 13 19

j-1  j

12 7 13 19

j-1  j

7 12 13 19

j (stop!)
Bubble Sort Example (Step-by-Step)

i = 0

19 12 13 7

j-1  j

12 19 13 7

j-1  j

12 13 19 7

j-1  j

12 13 19 7

j-1  j

12 13 7 19

j-1  j

12 13 7 19

j-1  j

12 7 13 19

j-1  j

12 7 13 19

j-1  j

12 7 13 19

j-1  j

7 12 13 19

j-1  j

A comparison made between these items

(stop!)
Bubble Sort Example (Step-by-Step)

1. Initial state:
   
   \[
   \begin{array}{cccc}
   i = 0 & 19 & 12 & 13 & 7 \\
   j-1 & j & & & \\
   \end{array}
   \]

2. Iteration 1:
   - Compare 19 and 12, swap
   - No need to compare 19 and 13
   - No need to compare 19 and 7

   \[
   \begin{array}{cccc}
   i = 1 & 12 & 13 & 7 & 19 \\
   j-1 & j & & & \\
   \end{array}
   \]

3. Iteration 2:
   - Compare 12 and 13, no swap
   - Compare 13 and 7, swap
   - No need to compare 13 and 19

   \[
   \begin{array}{cccc}
   i = 2 & 12 & 7 & 13 & 19 \\
   j-1 & j & & & \\
   \end{array}
   \]

4. Iteration 3:
   - No need to compare 12 and 7
   - Compare 7 and 12, no swap
   - Compare 12 and 13, no swap
   - No need to compare 12 and 19

   \[
   \begin{array}{cccc}
   i = 3 & 7 & 12 & 13 & 19 \\
   j-1 & j & & & \\
   \end{array}
   \]

A comparison made between these items

(stop!)
Bubble Sort Example (Step-by-Step)

1. i = 0
   - Original array: [19, 12, 13, 7]
   - After one pass: [12, 13, 7, 19]
   - (stop!)

2. i = 1
   - After two passes: [7, 12, 13, 19]
   - (stop!)

3. i = 2
   - After three passes: [7, 12, 13, 19]
   - (stop!)

A comparison made between these items.
Bubble Sort Example (Step-by-Step)

i = 0

19 12 13 7
j-1 j

i = 1

12 13 7 19
j-1 j

i = 2

12 7 13 19
j-1 j

i = 3

7 12 13 19
j-1 j

i = 4

7 12 13 19
j-1 j

A comparison made between these items
Bubble Sort Example (Step-by-Step)

Complexity Analysis:
Best: $n(n-1)/2$
Worst: $n(n-1)/2$
Average: $n(n-1)/2$
Another Example

Complexity Analysis:
Best: \( n(n-1)/2 \)
Worst: \( n(n-1)/2 \)
Average: \( n(n-1)/2 \)

A comparison made between these items
Another Example

i = 0

<table>
<thead>
<tr>
<th>19</th>
<th>7</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Complexity Analysis:
Best: $n(n-1)/2$
Worst: $n(n-1)/2$
Average: $n(n-1)/2$
Another Example

i = 0

A comparison made between these items

Complexity Analysis:
Best: \( n(n-1)/2 \)
Worst: \( n(n-1)/2 \)
Average: \( n(n-1)/2 \)
Another Example

A comparison made between these items

**Complexity Analysis:**
- Best: $n(n-1)/2$
- Worst: $n(n-1)/2$
- Average: $n(n-1)/2$
Another Example

\[ i = 0 \]

\[
\begin{array}{cccc}
19 & 7 & 12 & 13 \\
\end{array}
\]

A comparison made between these items

Complexity Analysis:
Best: \( n(n-1)/2 \)
Worst: \( n(n-1)/2 \)
Average: \( n(n-1)/2 \)
Another Example

i = 0

\[
i = 0
\]

\[
\begin{array}{cccc}
19 & 7 & 12 & 13 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 19 & 12 & 13 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 19 & 12 & 13 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 12 & 19 & 13 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 12 & 19 & 13 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 12 & 19 & 13 \\
\end{array}
\]

A comparison made between these items

**Complexity Analysis:**
Best: $n(n-1)/2$
Worst: $n(n-1)/2$
Average: $n(n-1)/2$
Another Example

\[ i = 0 \]

19 7 12 13

j-1 j

7 19 12 13

j-1 j

7 12 19 13

j-1 j

7 12 19 13

j-1 j

7 12 13 19

A comparison made between these items

**Complexity Analysis:**
Best: \( n(n-1)/2 \)
Worst: \( n(n-1)/2 \)
Average: \( n(n-1)/2 \)
Another Example

i = 0

A comparison made between these items

Complexity Analysis:
Best: n(n-1)/2
Worst: n(n-1)/2
Average: n(n-1)/2
Another Example

Complexity Analysis:
Best: \( \frac{n(n-1)}{2} \)
Worst: \( \frac{n(n-1)}{2} \)
Average: \( \frac{n(n-1)}{2} \)
### Another Example

<table>
<thead>
<tr>
<th>i = 0</th>
<th>i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram for i = 0" /></td>
<td><img src="image2.png" alt="Diagram for i = 1" /></td>
</tr>
</tbody>
</table>

**Complexity Analysis:**
- Best: $n(n-1)/2$
- Worst: $n(n-1)/2$
- Average: $n(n-1)/2$

A comparison made between these items.
Another Example

A comparison made between these items

Complexity Analysis:
Best: $n(n-1)/2$
Worst: $n(n-1)/2$
Average: $n(n-1)/2$
Another Example

<table>
<thead>
<tr>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Complexity Analysis:
Best: $n(n-1)/2$
Worst: $n(n-1)/2$
Average: $n(n-1)/2$

A comparison made between these items
Another Example

### Complexity Analysis:

- **Best:** $n(n-1)/2$
- **Worst:** $n(n-1)/2$
- **Average:** $n(n-1)/2$
Another Example

<table>
<thead>
<tr>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
<th>i = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>j-1</td>
<td>j</td>
<td>j-1</td>
<td>j-1</td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
</tbody>
</table>

A comparison made between these items

**Complexity Analysis:**
Best: \( n(n-1)/2 \)
Worst: \( n(n-1)/2 \)
Average: \( n(n-1)/2 \)
Another Example

Complexity Analysis:
Best: \( n(n-1)/2 \)
Worst: \( n(n-1)/2 \)
Average: \( n(n-1)/2 \)
Another Example

Complexity Analysis:
Best: $n(n-1)/2$
Worst: $n(n-1)/2$
Average: $n(n-1)/2$
Couldn't We Have Stopped Sooner?

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

i = 0

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

```
\text{Check out this link for animation:}
\text{http://cs.pugetsound.edu/~aasmith/sorters/}
```
Couldn't We Have Stopped Sooner?

i = 0

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

Check out this link for animation: http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

i = 0

j-1 j
19 7 12 13

j-1 j
7 19 12 13

j-1 j
7 12 19 13

j-1 j
7 12 19 13

j-1 j
7 12 13 19

j-1 j
7 12 13 19

j (stop!)

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

Check out this link for animation: http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

i = 0

i = 1

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

```
i = 0

19 7 12 13
  j-1  j

7 19 12 13
  j-1  j

7 12 19 13
  j-1  j

7 12 19 13
  j-1  j

7 12 13 19
  j-1  j

7 12 13 19
  j-1  j

```

```
i = 1

7 12 13 19
  j-1  j

7 12 13 19
  j-1  j

7 12 13 19
  j-1  j

7 12 13 19
  j-1  j

7 12 13 19
  j-1  j

(j stop!)

```

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Couldn't We Have Stopped Sooner?

Check out this link for animation: http://cs.pugetsound.edu/~aasmith/sorters/

Should end the algorithm now!

(How do we know the list was already sorted?)
Can We Do Better? (Cont.)

THIS IS THE BEST CASE!

(Only one pass of $i$ is needed when list is already sorted)
Can We Do Better? (Cont.)

\[
i = 0 \\
\begin{array}{llll}
7 & 12 & 13 & 19 \\
7 & 12 & 13 & 19
\end{array}
\]

\[
i = 1
\]

THIS IS THE BEST CASE!

(Only one pass of \( i \) is needed when list is already sorted)
Can We Do Better? (Cont.)

This is the best case!

(Only one pass of $i$ is needed when list is already sorted)
Can We Do Better? (Cont.)

\[ i = 0 \]

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

\[ i = 1 \]

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

THIS IS THE BEST CASE!

(Only one pass of \( i \) is needed when list is already sorted)
Can We Do Better? (Cont.)

\[ i = 0 \]

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

\[ i = 1 \]

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

THIS IS THE BEST CASE!

(Only one pass of \( i \) is needed when list is already sorted)
Can We Do Better? (Cont.)

\[
i = 0
\]

\[
\begin{array}{c}
7 \\
12 \\
13 \\
19 \\
\end{array}
\]

\[
i = 1
\]

\[
\begin{array}{c}
7 \\
12 \\
13 \\
19 \\
\end{array}
\]

\[
\begin{array}{c}
7 \\
12 \\
13 \\
19 \\
\end{array}
\]

\[
\begin{array}{c}
7 \\
12 \\
13 \\
19 \\
\end{array}
\]

\[
\begin{array}{c}
7 \\
12 \\
13 \\
19 \\
\end{array}
\]

\[
\begin{array}{c}
7 \\
12 \\
13 \\
19 \\
\end{array}
\]

\[
\begin{array}{c}
7 \\
12 \\
13 \\
19 \\
\end{array}
\]

**Done!**

THIS IS THE BEST CASE!

(Only one pass of \(i\) is needed when list is already sorted)
Bubble Sort (Optimize It!)

- Optimization: If a pass did not require swaps, the list is already sorted.
  - How do I know if a swap has been made? How do I stop the algorithm if a swap was not made in a pass?
  - Here's the old code:

```java
public static void bubbleSort(int[] list) {
    for (int i = 0; i < list.length; i++) {
        for (int j = 1; j < list.length - i; j++) {

            // Need to bubble list[j-1] up
            if (list[j-1] > list[j]) {
                // Swap the two adjacent items if out of order
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
            }
        }
    }
}
```
Bubble Sort Algorithm (Optimized)

- If a pass did not require swaps, the list is already sorted.
  - Optimizations made in red (below)

```java
public static void bubbleSort(int[] list) {
    boolean swapOccurred = true; // Why initialize to true?
    for (int i = 0; swapOccurred && i < list.length; i++) {
        swapOccurred = false; // Assume no swaps will happen

        for (int j = 1; j < list.length - i; j++) {
            if (list[j-1] > list[j]) { // Need to bubble list[j-1] up
                // Swap the two adjacent items if out of order
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
                swapOccurred = true; //we swapped elements, more passes needed
            }
        }
    }
}
```

Best case time complexity NOW?
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
  - Shaker Sort
- More Examples
- Conclusion

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
(Cocktail) Shaker Sort

- **Shaker Sort** is a further optimization on Bubble Sort

- Recall: One badly placed item screws up Bubble Sort's optimization
  - (e.g., a small item toward the end of the list)

- Shaker sort fixes this by alternating bubble sort left and right each pass
Shaker Sort Example
Shaker Sort Example

\[
\begin{align*}
\text{i} &= 0 \\
1 &\quad 5 &\quad 7 &\quad 0 \\
\text{j-1} &\quad \text{j}
\end{align*}
\]
Shaker Sort Example

i = 0

1 5 7 0

j-1 j

1 5 7 0

j-1 j
Shaker Sort Example

\[ i = 0 \]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\text{j-1} & \text{j} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\text{j-1} & \text{j} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\text{j-1} & \text{j} & & \\
\end{array}
\]
Shaker Sort Example

\( i = 0 \)

1 5 7 0

1 5 7 0

1 5 7 0

1 5 0 7
Shaker Sort Example

i = 0

1 5 7 0
j-1 j

1 5 7 0
j-1 j

1 5 7 0
j-1 j

1 5 0 7

1 5 0 7
j-1 j
### Shaker Sort Example

#### i = 0

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### i = 1

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>j+1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The arrows indicate the direction of the sort.
Shaker Sort Example

\[ i = 0 \]

\[ i = 1 \]

1 5 7 0
\[ j-1 \quad j \]

1 5 7 0
\[ j-1 \quad j \]

1 5 0 7

1 0 5 7
\[ j \quad j+1 \]

CSCI 161: Introduction to Computer Science - 9 - Searching and Sorting Algorithms
Shaker Sort Example

\[ i = 0 \]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 0 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j & j+1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 0 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j & j+1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 0 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j & j+1 \\
\end{array}
\]
Shaker Sort Example

\[ i = 0 \]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\text{j-1} & \text{j} & \text{j-1} & \text{j} \\
1 & 5 & 7 & 0 \\
\text{j-1} & \text{j} \\
1 & 5 & 0 & 7 \\
\text{j} & \text{j+1} \\
1 & 5 & 0 & 7 \\
\text{j-1} & \text{j} \\
\end{array}
\]

\[ i = 1 \]

\[
\begin{array}{cccc}
1 & 5 & 0 & 7 \\
\text{j} & \text{j+1} \\
1 & 0 & 5 & 7 \\
\text{j} & \text{j+1} \\
0 & 1 & 5 & 7 \\
\text{j-1} & \text{j} \\
0 & 1 & 5 & 7 \\
\text{j-1} & \text{j} \\
\end{array}
\]

\[ i = 2 \]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j-1} & \text{j} \\
0 & 1 & 5 & 7 \\
\text{j-1} & \text{j} \\
\end{array}
\]
Shaker Sort Example

\( i = 0 \)

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\text{j-1} & j & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\text{j-1} & j & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\text{j-1} & j & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 0 & 7 \\
\text{j} & \text{j+1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 5 & 7 \\
\text{j} & \text{j+1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\text{j} & \text{j-1} & & \\
\end{array}
\]
Shaker Sort Example

\[
\begin{align*}
&i = 0 \\
&1 &5 &7 &0 \\
&j-1 &j &j-1 &j \\
&1 &5 &7 &0 \\
&j-1 &j &j-1 &j \\
&1 &5 &7 &0 \\
&j-1 &j &j-1 &j \\
&1 &5 &0 &7 \\
&j &j+1 &j &j+1 \\
&1 &0 &5 &7 \\
&j &j+1 &j &j+1 \\
&1 &0 &5 &7 \\
&j &j+1 &j &j+1 \\
&0 &1 &5 &7 \\
&j-1 &j &j-1 &j \\
&0 &1 &5 &7 \\
&j-1 &j &j-1 &j \\
&0 &1 &5 &7 \\
&j-1 &j &j-1 &j \\
&0 &1 &5 &7 \\
&j-1 &j &j-1 &j \\
&0 &1 &5 &7
\end{align*}
\]
Shaker Sort Example

### i = 0

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### i = 1

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>0</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>j+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>j+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>j+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(No swaps! Done)

### i = 2

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j-1</td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
- More Examples
- Conclusion
Write a method `reverseList(int[] list)` that reverses the contents of an integer array, and returns it.

- And analyze its complexity
More Practice...

- Write the following methods and determine their time complexity.
  - Write a method `inAscendingOrder(int[] list)` that determines whether an integer array, named `list`, is in increasing order.
  - Write a method `printPairs(ArrayList<String> names)` that prints every pair of names exactly once in an ArrayList of Strings. For instance:

```java
ArrayList<String> names

Output
Aidan, Halle
Aidan, Marisa
Aidan, Troy
Halle, Marisa
Halle, Troy
Marisa, Troy
```
More Practice: Prime Numbers

- Write the following methods:

  ```java
  public boolean isPrime(int n)
  ```

  ```java
  public ArrayList<Integer> findPrimes(ArrayList<Integer> list)
  ```

  - Given an array list of integers, return an array list of prime numbers from that list.
Complexity of `isPrime()`?

- **Best case?**
  - What would cause the algorithm to terminate quickly?

- **Worst case?**
  - What would cause algorithm to incur the most iterations?

- **Average case?**
  - Is the common number prime or not?

```java
/** Tests whether the given number is prime. *
 * @param N   a positive number
 * @return true if N is prime, and false otherwise *
 */
public boolean isPrime(int N) {
    int divisor = 2;
    while (N % divisor != 0) {
        //try each divisor...
        divisor++;
    }
    return divisor == N;
}
```
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
- More Examples
- Conclusion
Summary of Time Complexity

- **Constant Time**: Number of steps/comparisons is independent of N
  - Example: 1 (best case in linear search)

- **Logarithmic Time**: Strive for this when possible
  - Example: log₂(N) (binary search, finding logarithms)

- **Linear Time**: Very good!
  - Example: N

- **Quadratic Time**: Not bad.... depends on exponent of leading term.
  - Examples: N³ (matrix-matrix mult.), N² (bubble and selection sort)

- **Exponential Time**: Avoid like plague
  - Examples: 2^N (traveling salesman, finding power sets)

- **Unbounded**: No guarantee algorithm will ever stop running!
  - Examples: BogoSort, roll die until 1 shows up
In Conclusion...

- Saw several classic CS algorithms...
- Searching a large collection is one of the most commonly-used operations
  - The most important thing:
    - Making the common case fast
    - Therefore, we sort to make search fast!

- Time complexity:
  - Time taken by an algorithm is quantified as number of comparisons required for a given problem size.
  - Not the only "yardstick"

- More sophisticated object behaviors (Next time)
Hwk 8
  • Optional Pitch is due this Friday! (By email)

Tomorrow's Lab:
  • Install IntelliJ https://www.jetbrains.com/idea/

Last time…
  • Enum classes for streamlining constants
  • Non-static vs. static
    - The static main method

Today:
  • A brief tour of IntelliJ
  • Start complexity analysis
Lab 11 Post-Mortem:
  • Thoughts on IntelliJ? Pros and cons?

Last time…
  • Time-Complexity Analysis
    • Linear Search

Today:
  • Binary Search Algorithm
Administtrivia 4/27

- Breathe: Last full week of classes!
- Hwk 8 due Wed, May 6
  - Finish it early!
  - You have all the tools to finish it right now!

- Last time...
  - Lab 11 post-mortem
  - Binary Search

- Today...
  - Impact of logarithmic-time complexity
  - Selection sort