CSE 373
Data Structures and Algorithms

Lecture 7: Sorting
Why Sorting?

- Practical application
  - People by last name
  - Countries by population
  - Search engine results by relevance

- Fundamental to other algorithms

- Different algorithms have different asymptotic and constant-factor trade-offs
  - No single ‘best’ sort for all scenarios
  - Knowing one way to sort just isn’t enough

- Many to approaches to sorting which can be used for other problems
Problem statement

- There are \( n \) comparable elements in an array and we want to rearrange them to be in increasing order

Pre:
- An array \( A \) of data records
- A value in each data record
- A comparison function
  - \(<, =, >\), compareTo

Post:
- For each distinct position \( i \) and \( j \) of \( A \), if \( i < j \) then \( A[i] \leq A[j] \)
- \( A \) has all the same data it started with
## Sorting Classification

<table>
<thead>
<tr>
<th>In memory sorting</th>
<th>External sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparison sorting</strong>&lt;br&gt;(\Omega(N \log N))</td>
<td><strong>Specialized Sorting</strong></td>
</tr>
<tr>
<td><strong>(O(N^2))</strong></td>
<td><strong>(O(N \log N))</strong></td>
</tr>
<tr>
<td>• Bubble Sort</td>
<td>• Merge Sort</td>
</tr>
<tr>
<td>• Selection Sort</td>
<td>• Quick Sort</td>
</tr>
<tr>
<td>• Insertion Sort</td>
<td>• Heap Sort</td>
</tr>
<tr>
<td>• Shell Sort</td>
<td></td>
</tr>
</tbody>
</table>
Comparison Sorting

Determine order through comparisons on the input data
Bogo sort

- bogo sort: orders a list of values by repetitively shuffling them and checking if they are sorted

- more specifically:
  - scan the list, seeing if it is sorted
  - if not, shuffle the values in the list and repeat

- This sorting algorithm has terrible performance!
  - Can we deduce its runtime?
  - What about best case?
public static void bogoSort(int[] a) {
    while (!isSorted(a)) {
        shuffle(a);
    }
}

// Returns true if array a's elements
// are in sorted order.
public static boolean isSorted(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i+1]) {
            return false;
        }
    }
    return true;
}
// Shuffles an array of ints by randomly swapping each
// element with an element ahead of it in the array.
public static void shuffle(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // pick random number in [i+1, a.length-1] inclusive
        int range = (a.length - 1) - (i + 1) + 1;
        int j = (int)(Math.random() * range + (i + 1));
        swap(a, i, j);
    }
}

// Swaps a[i] with a[j].
private static void swap(int[] a, int i, int j) {
    if (i == j)
        return;

    int temp = a[i];
a[i] = a[j];
a[j] = temp;
}
$O(n^2)$ Comparison Sorting
Bubble sort

- **bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

- more specifically:
  - scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

```
0  1  2  3  4  5
42 35 12  77 101  5
```

No need to swap
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

0 1 2 3 4 5
42 35 12 77 5 101

Largest value correctly placed
public static void bubbleSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = 1; j < a.length - i; j++) {
            // swap adjacent out-of-order elements
            if (a[j-1] > a[j]) {
                swap(a, j-1, j);
            }
        }
    }
}
Bubble sort runtime

Running time (# comparisons) for input size $n$:

\[ \sum_{i=0}^{n-1} \sum_{j=1}^{n-1-i} 1 = \sum_{i=0}^{n-1} (n - 1 - i) \]

\[ = n \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i \]

\[ = n^2 - n - \frac{(n - 1)n}{2} \]

\[ = \Theta(n^2) \]

- number of actual swaps performed depends on the data; out-of-order data performs many swaps
Selection sort

- **selection sort**: orders a list of values by repetitively putting a particular value into its final position

- **more specifically**:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places
Selection sort example

Scan right starting with 3.  
1 is the smallest. Exchange 1 and 3.

Scan right starting with 9.  
2 is the smallest. Exchange 9 and 2.

Scan right starting with 6.  
3 is the smallest. Exchange 6 and 3.

Scan right starting with 6.  
6 is the smallest. Exchange 6 and 6.
## Selection sort example 2

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>27</td>
<td>63</td>
<td>1</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; pass</td>
<td>1</td>
<td>63</td>
<td>27</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt; pass</td>
<td>1</td>
<td>9</td>
<td>27</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt; pass</td>
<td>1</td>
<td>9</td>
<td>14</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>27</td>
<td>63</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selection sort code

```java
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        // find index of smallest element
        int minIndex = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[minIndex]) {
                minIndex = j;
            }
        }
        // swap smallest element with a[i]
        swap(a, i, minIndex);
    }
}
```
Selection sort runtime

- Running time for input size $n$:
  - In practice, a bit faster than bubble sort. Why?

\[
\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n - 1 - (i + 1) + 1)
\]

\[
= \sum_{i=0}^{n-1} (n - i - 1)
\]

\[
= n \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1
\]

\[
= n^2 - \frac{(n - 1)n}{2} - n
\]

\[
= \Theta(n^2)
\]
Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions
Insertion sort

- Simple sorting algorithm.
  - $n-1$ passes over the array
  - At the end of pass $i$, the elements that occupied $A[0]...A[i]$ originally are still in those spots and in sorted order.

\[
\begin{array}{cccccccc}
2 & 15 & 8 & 1 & 17 & 10 & 12 & 5 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
2 & 8 & 15 & 1 & 17 & 10 & 12 & 5 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 8 & 15 & 17 & 10 & 12 & 5 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
Insertion sort example

3 is sorted.

3 and 9 are sorted.
Shift 9 to the right. Insert 6.

3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.

1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
Insertion sort runtime

- worst case: reverse-ordered elements in array.
  \[ \sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n - 1) = \frac{(n - 1)n}{2} \]
  \[ = \Theta(n^2) \]
- best case: array is in sorted ascending order.
  \[ \sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n) \]
- average case: each element is about halfway in order.
  \[ \sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} (1 + 2 + 3 \ldots + (n - 1)) = \frac{(n - 1)n}{4} \]
  \[ = \Theta(n^2) \]
Comparing sorts

- We've seen "simple" sorting algorithms so far, such as selection sort and insertion sort.
- They all use nested loops and perform approximately $n^2$ comparisons.
- They are relatively inefficient.
Consider the following array of int values.

[22, 11, 34, -5, 3, 40, 9, 16, 6]

(a) Write the contents of the array after 3 passes of the outermost loop of bubble sort.

(b) Write the contents of the array after 5 passes of the outermost loop of insertion sort.

(c) Write the contents of the array after 4 passes of the outermost loop of selection sort.