CSE 12
Sorting and Searching

- Sorting terminology
- Basic sorting algorithms
- Time and space costs of sorting
- Comparisons and swaps
- Comparing primitive types vs. comparing Objects
- The Comparable and Comparator interfaces
Sorting and ordering

• To sort elements of a set... a set of integers, or a set of Strings, or a set of Objects... a *total ordering* must apply to the elements of the set

• The elements of a set are said to be totally ordered by a relation \( \leq \) if the following axioms hold for all elements \( x, y, \) and \( z \) from the set:

  (transitivity:) if \( x \leq y \) and \( y \leq z \), then \( x \leq z \)
  (antisymmetry:) if \( x \leq y \) and \( y \leq x \), then \( x \) equals \( y \)
  (totality:) either \( x \leq y \) or \( y \leq x \) (or both)
Kinds of sorting order

- A *sorting algorithm* takes as input a sequence of elements, and arranges the elements of the sequence in order, according to a total ordering relation.

- **Increasing order** *(also called sorted or ascending order)*: The elements in the sorted sequence are unique (no duplicates) and are ordered such that \( e_1 < e_2 < \ldots < e_n \). For example: \( 2 < 5 < 16 \).

- **Nondecreasing order**: The elements in the sorted sequence may contain duplicates and are sorted such that \( e_1 \leq e_2 \leq \ldots \leq e_n \). For example: \( 2 \leq 5 \leq 5 \leq 21 \).
Kinds of sorting order

- **decreasing order** *(also called reverse sorted or descending order)*: The elements in the sorted sequence are unique (no duplicates) and are ordered such that \( e_1 > e_2 > \ldots > e_n \). For example: 21 > 16 > 5 > 2

- **nonincreasing order**: The elements in the sorted sequence may contain duplicates and are sorted such that \( e_1 \geq e_2 \geq \ldots \geq e_n \). For example: 21 \geq 5 \geq 5 \geq 2
There are many sorting algorithms: sorting is an important operation in many applications.

Some algorithms we will look at: selection sort, insertion sort, quicksort, merge sort, heapsort…

For each, we will want to understand the strategy it uses to sort a sequence, and consider its time and space costs.
Basic sorting algorithms are based on two fundamental operations on elements of the sequence being sorted:

- Comparing two elements
- Swapping two elements

Algorithms differ in how they choose which elements in the sequence to compare, and which elements to swap.

Other operations are performed as well, but these fundamental operations take most of the time.

Therefore, to analyze sorting algorithm time costs, it usually suffices to count the number of comparisons and swaps it takes to sort a sequence of size $n$. 
Sorting algorithm space complexity

- To sort a sequence of length $n$, you obviously need $O(n)$ space to store the sequence

- All the algorithms we will consider use either a constant $O(1)$ amount of additional space (so these are *in-place sorts*), or $O(n)$ additional space

- So, the overall space cost of all these algorithms is $O(n)$
Here is pseudocode for Selection Sort:

**Pseudocode:** selectionSort( comparableType [] array )

1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1

   // invariant: all elements from the first position to the last position of the sorted part are in non-decreasing order
Selection Sort: The Picture

(a) Initial configuration for selection sort. The input array is logically split into an unsorted part and a sorted part (initially empty).

(b) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (first pass).

(c) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (second pass).

(d) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (third pass).
Selection Sort Time Complexity: counting comparisons

• The 'outer loop' iterates $n-1$ times
• In the first iteration, $n-1$ comparisons are done to select the largest element in the unsorted part
• In the next iteration, $n-2$ comparisons are done to select the largest element in the unsorted part
• … in the last iteration, 1 comparison is done to select the largest element in the unsorted part
• So, total number of comparisons is 
  $$1+2+...+(n-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$
Insertion Sort

Here is pseudocode for Insertion Sort:

```
Pseudocode: insertionSort ( primitiveType [] array )
1. while the size of the unsorted part is greater than 0
2. let the target element be the first element in the unsorted part
3. find target’s insertion point in the sorted part
4. insert the target in its final, sorted position
   // invariant: the elements from position 0 to (size of the sorted part – 1)
   // are in nondecreasing order
```

```
Pseudocode finding the insertion point for the target in the sorted part
1. get a copy of the first element in the unsorted part
2. while ( there are elements in the unsorted part to examine AND
       we haven’t found the insertion point for the target )
3. move the element up a position // make room for the target
```
Insertion Sort: The Picture

(a) Initial configuration for insertion sort. The input array is logically split into a sorted part (initially containing one element) and an unsorted part.

(b) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (first pass).

(c) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (second pass).

(d) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (third pass).
Insertion Sort Time Complexity: counting comparisons

- The 'outer loop' iterates $n$ times
- In the first iteration, 0 comparisons are done to select the largest element in the unsorted part
- In the next iteration, 1 comparison is done to select the largest element in the unsorted part
- … in the last iteration, $n-1$ comparisons are done (in the worst case) to select the largest element in the unsorted part
- So, total number of comparisons in the worst case is $0+1+2+...+(n-1) = n(n-1)/2 = n^2/2 - n/2 = O(n^2)$
Time costs based on Swaps

• Instead of comparing element \textit{comparisons}, what if we count the number of element \textit{swaps}?

• \textbf{Selection Sort} always knows \textit{where} the next element is going to go, but it doesn’t know \textit{which} element to move. \textbf{Worst case? $O(n)$ swaps}

• \textbf{Insertion Sort} knows \textit{which} element to insert into the sorted part, but it doesn’t know \textit{where} in the sorted part it will go. \textbf{Worst case? $O(n^2)$ swaps}

• So, counting \textit{both} comparisons and swaps, both of these algorithms are $O(n^2)$ in the worst case
Worst case vs. best case

- When analyzing a sorting algorithm, it can be interesting to ask: What does it do on already sorted input?

- **Selection Sort** will still take $O(n^2)$ comparisons and $O(n)$ swaps

- But **Insertion Sort** will take just $O(n)$ comparisons and zero swaps!

- The best case and worst case time costs are the same for Selection Sort, but Insertion Sort has a best case that is much better than its worst case!
Quick Sort

• The basic strategy behind Quick Sort:

• Pick a “pivot” element of the sequence to be sorted

• “Partitition” the sequence into elements less than the pivot, followed by the pivot, followed by elements greater than the pivot: this puts the pivot element in the correct position in the sequence

• Then continue recursively partitioning each of the unsorted partitions, until the original sequence is sorted
Quick Sort

QuickSort( collection, first, last )

\[
\begin{cases}
\text{collection, if size(collection) } \leq 1 & \text{ // base case} \\
\text{QuickSort(collection, first, pivotPosition } - 1 \text{ )} & \text{ // recursive case — sort left partition} \\
\text{QuickSort(collection, pivotPosition + 1, last )} & \text{ // recursive case — sort right partition}
\end{cases}
\]

if size( collection ) > 1

where

collection — the collection to be sorted
pivotPosition = partition( collection, first, last )
first — the index of the first element of collection
last — the index of the last element of collection
Quick Sort: Implementation

```c
1 void quickSort( int[] array, int first, int last ) {
2     if ( first >= last ) // base case
3         return;
4
5     int pivotPosition = partition( array, first, last );
6
7     // recursive case: sort left partition
8     quickSort( array, first, pivotPosition-1 );
9
10    // recursive case: sort right partition
11    quickSort( array, pivotPosition+1, last );
12 }
```

The code follows the definition very closely...
Quick Sort: An Example

Note 1: picking pivot
Note 2: partition result
Note 3: recursive case
Note 4: base case

array indices
quickSort(array, 0, 7) //original call

selecting the pivot in partition()
after partition(array, 0, 7)
quickSort(array, 0, 2) //recursive case
after partition(array, 0, 2) //recursive case
quickSort(array, 0, 0) //base case
quickSort(array, 2, 2) //base case
quickSort(array, 4, 7) //recursive case
after partition(array, 4, 7)
quickSort(array, 4, 5)
after partition(array, 4, 5)
quickSort(array, 4, 4) //base case
quickSort(array, 5, 5) //base case
quickSort(array, 7, 7) //base case

<= sorted array
Quick Sort: Using A “Good” Pivot

If the pivot “evenly” splits the partition, there will be $\log_2 n$ levels, each of which requires $O(n)$ comparisons: $O(n \log_2 n)$ time complexity.

Space complexity: $O(\log_2 n)$ for the runtime stack activation records.
If the pivot always produces one empty partition and one with \( n - 1 \) elements, there will be \( n \) levels, each of which requires \( O(n) \) comparisons: \( O(n^2) \) time complexity.
Quicksort: picking the pivot

- Which element should Quicksort pick for the pivot?
- Ideally you want to pick the *median* element (the one that has half of the elements smaller and half larger than it), and avoid picking the smallest or largest element
- But finding the median element takes time...
- Faster ways to pick the pivot element:
  - pick the first or last element; but this is a very bad choice on already sorted input!
  - pick the middle element; this *can* be bad, but overall works well in practice
Merge Sort

• Since having partitions of the same size is so important to performance, is there some way we can guarantee equally sized partitions? Yes!

• Idea:
  – Recursively split the partitions into equal chunks until each partition is of size 1 (base case: a sequence of size 1 is already sorted)
  – As the recursion unwinds, *merge* the sorted partitions
Merge Sort

MergeSort(collection, first, last)

\[
\begin{cases}
\text{collection, if size( coll )} \leq 1 \\
\text{MergeSort(collection, first, midpoint } - 1) \\
\text{MergeSort(collection, midpoint, last)} \\
\text{merge ( collection, first, midpoint, last )}
\end{cases}
\]

// base case
// recursive case — sort left partition
// recursive case — sort right partition
// merge left and right partitions

if size( collection ) > 1

where

\begin{align*}
\text{collection} & \quad \text{the collection to be sorted} \\
\text{first} & \quad \text{the index of the first element of coll} \\
\text{midpoint} & \quad (\text{first} + \text{last} + 1) / 2 \\
\text{last} & \quad \text{the index of the last element of coll}
\end{align*}
Merge Sort: An Example

Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
Merge Sort: Time/Space Complexity

- **Time complexity**: Merge Sort guarantees that each split produces partitions of the same size (plus or minus 1), guaranteeing there will be at most \( \log_2 n \) levels.

- There are \( n \) comparisons done on each level to perform the merges at that level.

\[ \Rightarrow O(n \log_2 n) \] comparisons overall.

- **Space complexity**: difficult to make merge sort an in-place sort! Extra space is needed when merging. Still, \( O(n) \) overall space complexity.
Heap Sort: Time/Space Complexity

- **Heap sort** is an efficient sorting algorithm that uses a heap data structure which we will discuss later.

- **Time complexity**: Heap Sort requires $O(n \log_2 n)$ comparisons and swaps in the worst case.

- **Space complexity**: all operations can be done in place: $O(n)$ space complexity.
## Costs of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Cost</th>
<th>Space Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best Case</td>
<td>Worst Case</td>
</tr>
<tr>
<td>Selection sort</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>O(n)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>Quick sort</td>
<td>O(nlog₂n)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>Merge sort</td>
<td>O(nlog₂n)</td>
<td>O(nlog₂n)</td>
</tr>
<tr>
<td>Heap sort</td>
<td>O(nlog₂n)</td>
<td>O(nlog₂n)</td>
</tr>
</tbody>
</table>
Comparing Objects for Ordering

• Every Java object “knows how” to compare itself to another object for equality: call its equals() method, passing the other object as argument

• But some algorithms (e.g. sorting algorithms) and some data structures (e.g. heaps, binary search trees) require that objects be compared to each other for ordering

• How to do that?...
Comparing Objects for Ordering

- In Java, the $\leq$, $\geq$, $==$, $\neq$ operators can be used to compare primitive type values.

- Of these, only $==$ and $\neq$ can be used to compare object references... and they just compare the address stored in the reference, and do not compare for ordering.

- To compare objects for ordering in Java, understand and use the `Comparable<T>` interface.
The Comparable\langle T \rangle \textbf{ Interface} 

\begin{verbatim}
public interface Comparable\langle T \rangle{
    int compareTo( T o );
}
\end{verbatim}

\textbf{obj1} \textbf{.compareTo} (\textbf{obj2}) returns
\begin{itemize}
    \item a negative integer if \textbf{obj1} is “less than” \textbf{obj2}
    \item a positive integer if \textbf{obj1} is “greater than” \textbf{obj2}
    \item zero if \textbf{obj1} is “equal to” \textbf{obj2}
\end{itemize}

<table>
<thead>
<tr>
<th>Test Using \textbf{compareTo} () to Compare Objects</th>
<th>Equivalent Relational Operation to Compare Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{a} \textbf{.compareTo} (\textbf{b}) &lt; 0</td>
<td>\textbf{a} &lt; \textbf{b}</td>
</tr>
<tr>
<td>\textbf{a} \textbf{.compareTo} (\textbf{b}) &gt; 0</td>
<td>\textbf{a} &gt; \textbf{b}</td>
</tr>
<tr>
<td>\textbf{a} \textbf{.compareTo} (\textbf{b}) == 0</td>
<td>\textbf{a} == \textbf{b}</td>
</tr>
<tr>
<td>\textbf{a} \textbf{.compareTo} (\textbf{b}) != 0</td>
<td>\textbf{a} != \textbf{b}</td>
</tr>
<tr>
<td>\textbf{a} \textbf{.compareTo} (\textbf{b}) &lt;= 0</td>
<td>\textbf{a} &lt;= \textbf{b}</td>
</tr>
<tr>
<td>\textbf{a} \textbf{.compareTo} (\textbf{b}) &gt;= 0</td>
<td>\textbf{a} &gt;= \textbf{b}</td>
</tr>
</tbody>
</table>
The `String` class declares that it implements the `Comparable` interface so that an instance of itself can be compared to another `String`.

```java
public class String implements Comparable<String>
```

Note the use of a specific type in the type parameter field. This guarantees that we can only compare Strings to Strings.
Generic Methods

In Java, classes can be generic, and so can methods.

**Declaration**

```java
public <T> Collection<T> copy( Collection<T> c )
```

**Application**

```java
Collection<String> pets = new Collection<String>();

...  

Collection<String> petsCopy = copy( pets );
```

- **type variable (type parameter)**
- **use of type variable**
- **formal type parameter list**
- **actual type to map to T is inferred from the element type of pets**
Towards a Generic Sorting Method

• Suppose we want to write a sorting method that is as generic as possible: that is, one that can take as argument any array of things that can be compared for ordering.

• What should the type of the parameter be?

• First attempt:

  \(<T> \text{ void sort( T[] )}\)

• Not quite right. This will allow an array of any type \(T\)…

• We need to be sure that whatever type \(T\) ends up being, elements of the array are Comparable to one another
Towards a Generic Sorting Method

- What we need for our generic sorting method is a way to say that instances of the type $T$ are comparable to one another

- **Second attempt:**
  
  ```java
  <T extends Comparable<T>> void sort( T[] a )
  ```

- This means our sort() can sort instances of any type $T$, as long as $T$ implements `Comparable<T>`

- (In Java generics, “$T$ extends $X$” really means “$T$ implements $X$”, if $X$ names an interface, not a class)
Towards a Generic Sorting Method

• Problem: The second attempt means that the class we provide for $T$ must implement `Comparable` itself: It cannot have inherited the `compareTo()` method from a superclass that implemented `Comparable`.

We want our sort method to be able to sort objects of type `SuperType` and `SubType`. The header in attempt 2 would only allow my to sort objects of type `SuperType`. 😞
Using a Bounded Wildcard

- We need a way to say:
  - *either* \( T \) implements \( \text{Comparable} \) itself,
  - *or* some *supertype* of \( T \) implements \( \text{Comparable} \), in which case \( T \) will inherit that class’s \( \text{compareTo}() \) method

- Solution: combine the \( \texttt{?} \) wildcard with the keyword `super` to specify a type *lower bound*: \(<\texttt{? super } T>\) describes a family of types bounded *below* by \( T \).

\[
<T \text{ extends Comparable} \texttt{<? super } T>\texttt{>> void sort(T[] a)}
\]

- This means the method accepts an array of any type \( T \) (the lower bound) that directly implements \( \text{Comparable} \), or has a supertype (ancestor) that directly implements \( \text{Comparable} \).
Non-generic vs. Generic Selection Sort

/**
 * Sort the array elements in ascending order
 * using selection sort.
 * @param a - the array of integers to sort
 * @throws NullPointerException if a is null
 * @throws IndexOutOfBoundsException if <tt>n</tt> is greater than <tt>a.length</tt>
 */
static void selectionSort( int[] a ) {
    if ( a == null )
        throw new NullPointerException();

    n = a.length;

    // while the size of the unsorted part is > 1
    for ( int unsortedSize = n;
        unsortedSize > 1; unsortedSize-- ) {
        // find the position of the largest
        // element in the unsorted section
        int maxPos = 0;
        for (int pos = 1; pos < unsortedSize; pos++)
            if ( a[pos] > a[maxPos] )
                maxPos = pos;
        // postcondition: maxPos is the position
        // of the largest element in the unsorted
        // part of the array

        // Swap largest value with the last value
        // in the unsorted part
        int temp = a[unsortedSize - 1];
        a[unsortedSize - 1] = a[maxPos];
        a[maxPos] = temp;
    }
}

/**
 * Sort the array elements in ascending order
 * using selection sort.
 * @param a - array of Comparable objects to sort
 * @throws NullPointerException if a is null
 * @throws IndexOutOfBoundsException if <tt>n</tt> is greater than <tt>a.length</tt>
 */
static <T extends Comparable<T>>
    void selectionSort( T[] a ) {
    if ( a == null )
        throw new NullPointerException();

    n = a.length;

    // while the size of the unsorted part is > 1
    for ( int unsortedSize = n;
        unsortedSize > 1; unsortedSize-- ) {
        // find the position of the largest
        // element in the unsorted section
        int maxPos = 0;
        for (int pos = 1; pos < unsortedSize; pos++)
            if ( a[pos].compareTo(a[maxPos]) > 0 )
                maxPos = pos;
        // postcondition: maxPos is the position
        // of the largest element in the unsorted
        // part of the array

        // Swap largest value with the last value
        // in the unsorted part
        T temp = a[unsortedSize - 1];
        a[unsortedSize - 1] = a[maxPos];
        a[maxPos] = temp;
    }
}

Only the code in red had to be changed!
Making a Class Comparable

• A class that implements the Comparable interface defines, via its compareTo() instance method, a total ordering on instances of that class.

• This is referred to as the class’s natural ordering, and the compareTo() method of a class is called its natural comparison method.

• The compareTo() method should obey the total order axioms.

• Also, the compareTo() method should be consistent with the class’s equals() method: x.equals(y) if and only if x.compareTo(y) == 0.
Example: Making a Name class Comparable

The natural sort order for names is lexicographical by last name, then first name.

If we override equals() in order to be consistent with compareTo(), we should also override hashCode() to honor the contract for equals()
Making Name Comparable

```java
/**
 * The <tt>Name</tt> class stores a person’s first and last name.
 */
public class Name implements Comparable<Name> {
  private String firstName;
  private String lastName;

  // CONSTRUCTORS, ACCESSORS, AND MUTATORS NOT SHOWN

  /**
   * Determine if this name is equal to the given argument.
   * @param o The <tt>Name</tt> object to which we are to compare this <tt>Name</tt>
   * @return true if both the first and last name of this <tt>Name</tt> match the first and last names of the given argument; otherwise return false.
   */
  public boolean equals( Object o ) {
    if ( o instanceof Name )
      return this.compareTo( (Name)o ) == 0;
    return false;
  }
```
Making Name Comparable

```java
/**
 * Return the hashcode for this <tt>Name</tt>.
 * @return The hashcode for this name.
 */
public int hashCode(){
    return 31 * this.lastName.hashCode() +
           this.firstName.hashCode();
}
```

By the contract for `equals()` in `Object`, if `o1.equals(o2)`, then `o1.hashCode == o2.hashCode`. So, if we override `equals()`, we really should override `hashCode()` as well.
Making Name Comparable

```java
/**
 * Compare this <tt>Name</tt> to the <tt>Name</tt> supplied in the argument.
 * @param o The <tt>Name</tt> to which to compare this <tt>Name</tt>.
 * @return -1 if this name is lexicographically less than the other name.
 * 0 if they are equal
 * 1 if this name is lexicographically greater than the other name
 * @throws <tt>IllegalArgumentException</tt> if <tt>other</tt> is <tt>null</tt>
 */
public int compareTo( Name other ){
    // check null argument
    if ( other == null ) throw new IllegalArgumentException();
    // compare last names
    int result = this.lastName.compareTo( other.lastName() );
    // if different, return result of that comparison
    if (result != 0) return result;
    // else, return result of comparing first names
    return this.firstName.compareTo( other.firstName() );
}
```
Next time

- Intro to trees
- The Priority Queue ADT
- Heaps
- Heapsort

Reading: Gray, Ch 10