Lecture 10-11: Sorting Basics and Comparators

- Introduction
- Selection sort
- Insertion sort
- Comparators

Some slides adopted from Algorithms 4th Edition or COS226
Why study sorting?

- It’s more common than you think: e.g., sorting flights by price, contacts by last name, files by size, emails by day sent, neighborhoods by zipcode, etc.

- Good example of how to compare the performance of different algorithms for the same problem.

- Some sorting algorithms relate to data structures.

- Sorting your data will often be a good starting point when solving other problems (keep that in mind for interviews).
Definitions

- **Sorting**: the process of arranging $n$ items of a collection in non-decreasing order (e.g., numerically or alphabetically).

- **Key**: assuming that an item consists of multiple components, the key is the property based on which we sort items.

- Examples: items could be books and potential keys are the title or the author which can be sorted alphabetically or the ISBN which can be sorted numerically.
How many different algorithms for sorting can there be?

- Adaptive heapsort
- Bitonic sorter
- Block sort
- Bubble sort
- Cascade mergesort
- Cocktail sort
- Comb sort
- Flashsort
- Gnome sort
- Heapsort
- Insertion sort
- Mergesort
- Odd-even sort
- Pancake sort
- Quicksort
- Radixsort
- Selection sort
- Shell sort
- Spaghetti sort
- Treesort
- ...
Rules of the game - Comparing

- We will be sorting arrays of $n$ items, where each item contains a key. In Java, objects are responsible in telling us how to *naturally* compare their keys.

- Let’s say we want to sort an array of objects of type $T$.

- Our class $T$ should implement the `Comparable` interface (more by the end of this lecture). We will need to implement the `compareTo` method to satisfy a total order.

[https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)
Total order

- Sorting is well defined if and only if there is total order.

- Total order: a binary relation \( \leq \) that satisfies:
  - Reflexivity: for all \( v \), \( v \leq v \).
  - Totality (strongly connected): for all \( v \) and \( w \), if both \( v \leq w \) or \( w \leq v \).
  - Transitivity: for all \( v \) and \( w \), if both \( v \leq w \) or \( w \leq x \) then \( v \leq x \).
  - Antisymmetry: for all \( v \) and \( w \), if both \( v \leq w \) and \( w \leq v \) then \( v = w \).

- For example, standard numerical order for numbers, lexicographical order for strings, chronological order for dates, etc.
INTRODUCTION

Rules of the game

- We will be sorting arrays of $n$ items, where each item contains a key.
- In Java, objects are responsible in telling us how to naturally compare their keys.
- This is achieved by making our class $T$ implement the Comparable interface (more on this by the end of the lecture). We will need to implement `compareTo` to satisfy a total order:

  ```java
  public int compareTo(T that)
  ```

- Implement it so that $v$.compareTo($w$):
  - Returns $>0$ if $v$ is greater than $w$.
  - Returns $<0$ if $v$ is smaller than $w$.
  - Returns $0$ if $v$ is equal to $w$.
- We can now use the same sorting algorithm to sort collections of different data types.
- Java classes such as `Integer`, `Double`, `String`, `File` all implement `Comparable`.

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
Two useful abstractions

- Let’s assume we want to sort an array of comparable objects $T$.
  - $T$ is a bounded generic that implements the interface `Comparable`.
- We will refer to data only through `comparisons` and `exchanges`.
- **Comparisons**: Is $v$ less than $w$?
  
  ```java
  v.compareTo(w) < 0;
  ```
- **Exchanges**: swap item in array $a[]$ at index $i$ with the one at index $j$.
  ```java
  T temp = a[i];
  a[i]=a[j];
  a[j]=temp;
  ```
Rules of the game - Cost model

- **Sorting cost model**: we count *compares* and *exchanges*. If a sorting algorithm does not use exchanges, we count *array accesses*.

- There are other types of sorting algorithms where they are not based on comparisons (e.g., radixsort). We will not see these in CS62 but stay tuned for CS140.
Extra memory: often as important as running time. Sorting algorithms are divided into two categories:

- **In place**: use constant or logarithmic extra memory, beyond the memory needed to store the items to be sorted.
- **Not in place**: use linear auxiliary memory.
Rules of the game - Stability

- **Stable**: sorting algorithms that sort repeated elements in the same order that they appear in the input.

Lecture 10-11: Sorting Basics and Comparators

- Introduction
- Selection sort
- Insertion sort
- Comparators
Selection sort

- Divide the array in two parts: a *sorted subarray* on the left and an *unsorted* on the right.

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.

3 44 38 5 47 1 36 26
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

```
1  44  38  5  47  3  36  26
```

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

Repeat:

• Find the smallest element in the unsorted subarray.
• Exchange it with the leftmost unsorted element.
• Move subarray boundaries one element to the right.
Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Repeat:
- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.

1 3 5 26 36 38 47 44
Selection sort

Repeat:
- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.
Selection sort

Repeat:

- Find the smallest element in the unsorted subarray.
- Exchange it with the leftmost unsorted element.
- Move subarray boundaries one element to the right.
Selection sort

```java
public static <E extends Comparable<E>> void selectionSort(E[] a) {
}
```
Selection sort

```java
public static <E extends Comparable<E>> void selectionSort(E[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        int min = i;
        for (int j = i+1; j < n; j++) {
            if (a[j].compareTo(a[min]) < 0) {
                min = j;
            }
        }
        E temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

- **Invariants**: At the end of each iteration $i$:
  - the array $a$ is sorted in ascending order for the first $i+1$ elements $a[0...i]$
  - no entry in $a[i+1...n-1]$ is smaller than any entry in $a[0...i]$
Selection sort: mathematical analysis for worst-case

```java
public static <E extends Comparable<E>> void selectionSort(E[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        int min = i;
        for (int j = i+1; j < n; j++) {
            if (a[j].compareTo(a[min])<0) {
                min = j;
            }
        }
        E temp = a[i];
        a[i]=a[min];
        a[min]=temp;
    }
}
```

- **Comparisons**: \(1 + 2 + \ldots + (n - 2) + (n - 1)\sim n^2/2\), that is \(O(n^2)\).

- **Exchanges**: \(n\) or \(O(n)\), making it useful when exchanges are expensive.

- **Running time is** quadratic, even if input is sorted.

- **In-place**, requires almost no additional memory.

- **Not stable**, think of the array \([5\_a, 3, 5\_b, 1]\) which will end up as \([1, 3, 5\_b, 5\_a]\).
Practice Time

› Using selection sort, sort the array with elements [12, 10, 16, 11, 9, 7].
› Visualize your work for every iteration of the algorithm.
### SELECTION SORT

**Answer**

<table>
<thead>
<tr>
<th>1st</th>
<th>Selection Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>
Lecture 16: Sorting Basics I

- Introduction
- Selection sort
- Insertion sort
Insertion sort

- Keep a *partially sorted subarray* on the left and an *unsorted subarray* on the right.

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

| 3 | 5 | 38 | 44 | 47 | 1 | 36 | 26 |

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.

3 5 38 44 1 47 36 26
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
**Insertion Sort**

**Insertion sort**

| 3 | 5 | 1 | 38 | 44 | 47 | 36 | 26 |

- **Repeat:**
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

| 5 | 3 | 1 | 38 | 44 | 47 | 36 | 26 |

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
**Insertion Sort**

Insertion sort

```
| 3 | 1 | 5 | 38 | 44 | 47 | 36 | 26 |
```

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

1 3 5 38 44 47 36 26

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
**Repeat:**

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

- **Repeat:**
  - Examine the next element in the unsorted subarray.
  - Find the location it belongs within the sorted subarray and insert it there.
  - Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Find the location it belongs within the sorted subarray and insert it there.
- Move subarray boundaries one element to the right.

Insertion sort

1 3 5 26 36 38 44 47
2.1 INSERTION SORT DEMO
In case you didn’t get this...

▶ https://www.youtube.com/watch?v=ROalU379l3U
Insertion sort

```java
public static <E extends Comparable<E>> void insertionSort(E[] a) {
}
```
public static <E extends Comparable<E>> void insertionSort(E[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if(a[j].compareTo(a[j-1])<0){
                E temp = a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
            } else{
                break;
            }
        }
        break;
    }
    ▸ Invariants: At the end of each iteration i:
    ▸ the array a is sorted in ascending order for the first i+1 elements a[0...i]
Insertion sort: mathematical analysis for worst-case

```java
public static <E extends Comparable<E>> void insertionSort(E[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (a[j].compareTo(a[j-1])<0){
                E temp = a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
            }
            else{
                break;
            }
        }
    }
}
```

- **Comparisons:** $0 + 1 + 2 + \ldots + (n - 2) + (n - 1) \sim n^2/2$, that is $O(n^2)$.

- **Exchanges:** $0 + 1 + 2 + \ldots + (n - 2) + (n - 1) \sim n^2/2$, that is $O(n^2)$.

- **Worst-case running time is quadratic.**

- **In-place,** requires almost no additional memory.

- **Stable**
public static <E extends Comparable<E>> void insertionSort(E[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (a[j].compareTo(a[j-1])<0){
                E temp = a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
            }else{
                break;
            }
        }
    }
}

- **Average case:** quadratic for both comparisons and exchanges $\sim n^2/4$ when sorting a randomly ordered array.

- **Best case:** $n - 1$ comparisons and 0 exchanges for an already sorted array.

https://www.toptal.com/developers/sorting-algorithms/insertion-sort
Practice Time

- Using insertion sort, sort the array with elements [12, 10, 16, 11, 9, 7].
- Visualize your work for every iteration of the algorithm.
Answer

Insertion Sort

1st row: 12, 10, 16, 11, 9, 7
2nd row: 10, 12, 16, 11, 9, 7
3rd row: 10, 12, 16, 11, 9, 7
4th row: 10, 11, 12, 16, 9, 7
5th row: 9, 10, 11, 12, 16, 7
last row: 7, 9, 10, 11, 12, 16

Lecture 10-11: Sorting Basics and Comparators

- Introduction
- Selection sort
- Insertion sort
- Comparators
Comparable

- Interface with a single method that we need to implement:
  
  ```java
  public int compareTo(T that)
  ```

- Implement it so that `v.compareTo(w)`:
  - Returns >0 if `v` is greater than `w`.
  - Returns <0 if `v` is smaller than `w`.
  - Returns 0 if `v` is equal to `w`.

- Corresonds to natural ordering.
Sometimes the natural ordering is not the type of ordering we want.

Comparator is an interface which allows us to dictate that kind of ordering we want by implementing the method:

```java
public int compare(T this, T that)
```

Implement it so that `compare(v, w)`:  
- Returns $>0$ if $v$ is greater than $w$.  
- Returns $<0$ if $v$ is smaller than $w$.  
- Returns $0$ if $v$ is equal to $w$.  

The Java Collections Framework

Collections

COMPARATORS

Sorting Collections

- Collections class contains:
  - `public static <T extends Comparable<? super T>> void sort(List<T> list)`
  - Generic methods introduce their own type parameters.
  - Use `extends` with generics, even if the type parameter implements an interface.
  - The class `T` itself or one of its ancestors implements `Comparable`.
- `Collections.sort(list)`
  - Implemented as optimized mergesort
  - If list’s elements do not implement `Comparable`, throw `ClassCastException`.
Alternative Sorting of Collections

- Collections class contains:
  - static <T> void sort(List<T> list, Comparator<? super T> c)
  - Collections.sort(list, someComparator);
  - If list’s elements do not implement Comparable or cannot be compared with Comparator, throw ClassCastException.
Example: Natural sorting for Employees

```java
public class Employee implements Comparable<Employee> {
    private int id;
    private String name;
    private int age;
    private long salary;

    public Employee() {
    }

    public Employee(int id, String name, int age, long salary) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    //getters and setters

    @Override
    public int compareTo(Employee e) {
        if (this.id > e.id) {
            return 1;
        } else if (this.id < e.id) {
            return -1;
        } else {
            return Character.toString(this.name.charAt(0)).compareToIgnoreCase(Character.toString(e.name.charAt(0)));
        }
    }
}
```

https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator
Example: Alternative sorting for Employees

```java
public static Comparator<Employee> nameComparator = new Comparator<Employee>() {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.getName().compareTo(e2.getName());
    }
};

public static Comparator<Employee> idComparator = new Comparator<Employee>() {
    @Override
    public int compare(Employee e1, Employee e2) {
        return Integer.valueOf(e1.getId()).compareTo(Integer.valueOf(e2.getId()));
    }
};

public static void main(String[] args) {
    Employee e1 = new Employee(5, "Yush", 22, 1000);
    Employee e2 = new Employee(8, "Tharun", 24, 25000);
    Employee e3 = new Employee(5, "Yash", 18, 10000);
    List<Employee> list = new ArrayList<Employee>();
    list.add(e1);
    list.add(e2);
    list.add(e3);

    System.out.print("Unsorted list: ");
    System.out.println(list);

    Collections.sort(list); // call @compareTo(o1)
    System.out.print("Naturally sorted list: ");
    System.out.println(list);

    Collections.sort(list, Employee.nameComparator); // call @compare (o1,o2)
    System.out.print("Sorted list based on names: ");
    System.out.println(list);

    Collections.sort(list, Employee.idComparator); // call @compare (o1,o2)
    System.out.print("Sorted list based on IDs: ");
    System.out.println(list);
}
```

https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator
Lecture 10-11: Sorting Basics and Comparators

- Introduction
- Selection sort
- Insertion sort
- Comparators
Readings:

- Recommended Textbook:
  - Chapter 2.1 (pages 244-262)
  - Chapter 2.5 (Pages 338-339)

- Recommended Textbook Website:
  - Elementary sorts: https://algs4.cs.princeton.edu/21elementary/

- Oracle Documentation:
  - Comparable: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
  - Comparator: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Code

- Lecture 10-11 code

Practice Problems:

- 2.1.1-2.1.8