L14: Sorting

Alex Steiger
CompSci 201: Spring 2024
2/28/2024: LEAP DAY EVE
Announcements, Coming up

• Today, Wednesday 2/28
  • APT 5 (linked list problems) due

• Next Monday 3/4
  • Project P3: DNA (linked list project) due

• Next Wednesday 3/6
  • APT 6 (sorting problems) due

• Then...Spring Break!
Today’s outline

1. Announce Midsemester Survey
   1. Invaluable for staff, especially UTAs ⇒ for you!
   2. Look for Canvas announcement from Violet

2. Sorting in Java: Comparing objects with Comparable and Comparator

3. Efficient sorting algorithms
   1. Insertion sort
   2. Recursive Mergesort
Sorting in Java: Comparable, Comparator
Sorting w/ Java.util: Put elements of Array/List in non-decreasing order

- Arrays.sort / Collections.sort are void – they sort the array/list passed as an argument.
- Default order is non-decreasing (least to greatest).

```java
67    int[] elements = {5, 3, 9, 2, 4, 1};
68    Arrays.sort(elements);
69    System.out.println( Arrays.toString(elements) );
```

- Prints [1, 2, 3, 4, 5, 9]
Java API Sort Algorithms

- `Collections.sort` (for a List)
- `Arrays.sort` (for an Array)

- Both $O(N \log(N))$, nearly linear runtime complexity.
- Sorts in-place, mutates the input rather than return a new List/Array.
- **Stable**, does not reorder elements if not needed (e.g., if two elements are equal).
What can be compared and sorted in Java?

- Objects of a Class that implements `Comparable` interface. Has a `naturalOrder`.
- Requires implementing a `.compareTo()` method

Should return an int:

- `< 0` if `this` comes before the parameter.
- `0` if `this` and the parameter are equal.
- `> 0` if `this` comes after the parameter.

```java
private static class Person implements Comparable<Person> {
    String first;
    String last;
    public Person(String s) {...}
    public String getFirst() { return first; }
    public String getLastName() { return last; }
    public String toString() { return first + " " + last; }
    @Override
    public int compareTo(Person p){
        int diff = last.compareTo(p.last);
        if (diff != 0) return diff;
        return first.compareTo(p.first);
    }
}
```
Strings are Comparable

• What is the equivalent of `<` for Strings?
• Use the `compareTo` method for the natural lexicographic (dictionary/sorted) ordering.

```
jshell> "a".compareTo("b");
$30 ==> -1

jshell> "b".compareTo("b");
$31 ==> 0

jshell> "b".compareTo("a");
$32 ==> 1

jshell> "az".compareTo("cb");
$37 ==> -2
```

Negative for “less than”
Zero for “equal”
Positive for “less than”
Lexicographic, check first character, second if equal, third if still equal, ...
Sorting Comparable objects by naturalOrder

\[\text{sloth, house, owl, ant, mice, kelp}\]

```java
String[] a = {"sloth", "house", "owl", "ant", "mice", "kelp"};
System.out.println(Arrays.toString(a));

String[] copy = Arrays.copyOf(a, a.length);
Arrays.sort(copy);
System.out.println(Arrays.toString(copy));
```

\[\text{ant, house, kelp, mice, owl, sloth}\]

- naturalOrder for Strings is lexicographic (alphabetical or dictionary order)
Comparable for other classes?

All Blob comparing code available here

• Can implement Comparable interface when defining your own class.

```java
public class Blob implements Comparable<Blob> {
    String name;
    String color;
    int size;
}
```

• Must implement a compareTo method

```java
@Override
public int compareTo(Blob other) {
    return this.name.compareTo(other.name);
}
```
Sorting Comparable Objects

• Running code in a main method...

40    System.out.println(myBlobs);

Original: [(bo, blue, 4), (al, red, 2), (cj, green, 1), (di, red, 4)]

42    Collections.sort(myBlobs);
43    System.out.println(myBlobs);

Sorted: [(al, red, 2), (bo, blue, 4), (cj, green, 1), (di, red, 4)]

• Formal guarantee: Element $e_1$ will come before $e_2$ (after sorting) if $e_1.compareTo(e2) < 0$. 
Defining a Comparator

• What if...
  • The class doesn’t implement Comparable?
  • Or you want to sort a different way?

• Create a helper class that implements the Comparator interface.
  • One method: compare: indicates how to compare two objects

• Then pass a Comparator object to your call to sort.
Defining a Comparator<Blob>

```java
import java.util.Comparator;

Separate class:
• implements Comparator<TypeToCompare>,
• and implements a single method compare

public class BlobComparator implements Comparator<Blob> {
  @Override
  public int compare(Blob a, Blob b) {
    int sizeDiff = a.size - b.size;
    if (sizeDiff != 0) {
      return (-1) * sizeDiff;
    }
    return a.compareTo(b);
  }
}
```

Takes 2 parameters, Should return:
• < 0 if a comes before b,
• > 0 if a comes after b,
• 0 if equal in order

Flipping the sign reverses the comparison, large to small

Breaking ties by the natural order
Sorting with a Comparator

• Running code in a main method...

```java
40    System.out.println(myBlobs);
```

Original: `[(bo, blue, 4), (al, red, 2), (cj, green, 1), (di, red, 4)]`

```java
48    Collections.sort(myBlobs, new BlobComparator());
49    System.out.printf(format: "%s
\n", myBlobs);
```

Sorted: `[(bo, blue, 4), (di, red, 4), (al, red, 2), (cj, green, 1)]`

• Element `e1` will come before `e2` (after sorting) if `compare(e1, e2) < 0`. 
Private Inner Comparator

• Can define a Comparator class as a private inner class if only used inside the class.
• Useful for APTs, here is an example:

SimpleSort APT

Problem Statement

Sometimes sorting helps in recognizing patterns. Given an array of strings, write the method `recognize` that returns an array of the same strings, but sorted by length with the shortest strings first and the longest strings last in the returned array. You can create a new array or sort the array parameter `value`, but you must return a sorted array containing the same strings that are in `values`.

In the returned array, strings that are the same length should be sorted in alphabetical order. See the examples for details.

```
public class LengthSort {
   public String[] rearrange(String[] values){
      // you write code here and replace statement below
      return null;
   }
}
```
Private Inner Comparator

- Can define a Comparator class as a private inner class if only used inside the class.
- Useful for APTs, here is an example:

*Given `String[]` values:*
  - Sort first in *non-decreasing order of length*,
  - then sort same-length in *alphabetical order*.

*`[“a”, “b”, “c”, “an”, “be”, “pi”, “test”, “quiz”]`*
Template for Solving **LengthSort** with a Private Inner Comparator

Can see this code here

```java
import java.util.Arrays;
import java.util.Comparator;

public class LengthSort {
    private class LengthSortComp implements Comparator<String> {
        @Override
        public int compare(String a, String b) {
            // Need to modify this to solve the problem
            return a.compareTo(b);
        }
    }

    public String[] rearrange(String[] values) {
        Arrays.sort(values, new LengthSortComp());
        return values;
    }
}
```
Comparable vs. Comparator

- **Comparable** `a`: use `a.compareTo(b)`
  - What is method signature? One parameter
  - Method in class of which object `a` is an instance
  - `a` is `this`, `b` is a parameter

- **Comparator** `c`, use `c.compare(a,b)`
  - Method has two parameters
  - Part of [Comparator](Java API link)

- Both return an int:
  - `< 0 (means a comes before b)
  - `== 0` (means `a` equals `b`)
  - `> 0` (means `a` comes after `b`)
Runtime Complexity of Sort and Comparator?

• `Arrays.sort`, `Collections.sort`, call either `compareTo` (default) or `compare` (if you give a Comparator)...

• $O(N \log(N))$ `compareTo/compares`, on an Array/List of N elements.

• Exists theoretical proof that this many comparisons is **necessary** for any comparison-based sorting.
When is comparing once not constant time?

```java
public class ListComp implements Comparator<List<Integer>> {
    @Override
    public int compare(List<Integer> list1, List<Integer> list2) {
        int minLength = Math.min(list1.size(), list2.size());
        for (int i=0; i<minLength; i++) {
            int diff = list1.get(i) - list2.get(i);
            if (diff != 0) {
                return diff;
            }
        }
        return 0;
    }
}
```

Runtime complexity of this Comparator may depend on the length of the two Lists being compared.

Overall runtime complexity to sort N ArrayLists, each with M elements, is O(MN log(N)) in the worst case with this Comparator.
java.util.Comparator: Convenient Shorthands

- `Comparator.naturalOrder` and `reversed()`

```java
jshell> Comparator<String> c = Comparator.naturalOrder()
c ==> INSTANCE

jshell> c.compare("a","b")
$12 ==> -1

jshell> c.reversed().compare("a","b")
$13 ==> 1
```

- `Comparator.comparing`

```java
jshell> Comparator<String> c = Comparator.comparing(String::length)
c ==> java.util.Comparator$$Lambda$27/0x0000000800b97c/0@2b71fc7e

jshell> c.compare("this", "is")
$15 ==> 1

jshell> c.compare("is", "it")
$16 ==> 0
```

Must be Comparable

Syntax is: `<Type>::<method name>` to sort something of the Type by the result of some getter method that returns something Comparable.
Comparator-generating shorthands

[sloth, house, owl, ant, mice, kelp]

copy = Arrays.copyOf(a, a.length);
Arrays.sort(copy, Comparator.comparing(String::length));
System.out.println(Arrays.toString(copy));

[owl, ant, mice, kelp, sloth, house]

• Why does "owl" come before "ant"?
  • Stable sort respects order of equal keys
Using `.thenComparing` shorthand

[sloth, house, owl, ant, mice, kelp]

```java
Arrays.sort(copy, Comparator.comparing(String::length).
    thenComparing(Comparator.naturalOrder()));
```

[ant, owl, kelp, mice, house, sloth]

- First compare by length
  - if same? Compare naturally
Comparator with “lambdas”

Can also define a comparator with a “lambda expression.”

```java
Integer[] nums = {2, 0, 1};
Comparator<Integer> comp = (a, b) -> (b - a);
Arrays.sort(nums, comp);  // nums is now { 2, 1, 0 }
```
What is printed by the following line of code?

```java
System.out.println("duke".compareTo("devils"));
```

- true
- false
- an integer less than 0
- 0
✓- an integer greater than 0
After sorting, ar will be...

```java
String[] ar = {"bird", "dog", "cat", "snake"};
Comparator<String> comp = Comparator.comparing(String::length);
Arrays.sort(ar, comp);
```

Ans: [dog, cat, bird, snake]
Suppose you have the following list of lists of integers:

[[2, 0, 1], [1, 0, 1], [1, 6]]. After sorting, the list would be ordered as...

```java
public class ListComp implements Comparator<List<Integer>> {
    @Override
    public int compare(List<Integer> list1, List<Integer> list2) {
        int minLength = Math.min(list1.size(), list2.size());
        for (int i = 0; i < minLength; i++) {
            int diff = list1.get(i) - list2.get(i);
            if (diff != 0) {
                return diff;
            }
        }
        return 0;
    }
}
```

Ans: [[1, 0, 1], [1, 6], [2, 0, 1]]
Suppose you have an `ArrayList` `myLists` of `N ArrayLists`, each of size at most `M`. The worst-case runtime complexity to compare any two elements of `myLists` would be....

```java
public class ListComp implements Comparator<List<Integer>> {
    @Override
    public int compare(List<Integer> list1, List<Integer> list2) {
        int minLength = Math.min(list1.size(), list2.size());
        for (int i=0; i<minLength; i++) {
            int diff = list1.get(i) - list2.get(i);
            if (diff != 0) {
                return diff;
            }
        }
        return 0;
    }
}
```

Ans: $O(M)$
Given an Array of N Strings, each of length at most M, the worst case runtime complexity to sort the Array with java.util.Arrays.sort is..

Ans: $O(M \times N \log N)$
Efficient sorting algorithms

See example implementations here
Selection Sort with a Loop Invariant

- Loop invariant: On iteration $i$, the first $i$ elements are the smallest $i$ elements in sorted order.

- On iteration $i$...
  - Find the smallest element from index $i$ onward
    - *(By loop invariant, must be the next smallest element)*
  - Swap that with the element at index $i$

- Algorithm is called **Selection Sort**.
Selection Sort Code and Runtime

```java
public static void selectSort(int[] ar) {
    for (int i=0; i<ar.length; i++) {
        int minDex = i;
        for (int j=i+1; j<ar.length; j++) {
            if (ar[j] < ar[minDex]) {
                minDex = j;
            }
        }
        int temp = ar[i];
        ar[i] = ar[minDex];
        ar[minDex] = temp;
    }
}
```

Nested O(N) loops, overall O(N^2)
Mergesort

High level idea:
- **Base case:** size 1
  - Return list
- **Recursive case:**
  - Mergesort(first half)
  - Mergesort(second half)
  - ...
Mergesort

High level idea:
• Base case: size 1
  • Return list
• Recursive case:
  • Mergesort(first half)
  • Mergesort(second half)
  • Merge the sorted halves
  • Return sorted