Sorting

For the remaining part of this textbook, we'll discuss the sorting problem, which can be informally defined as putting a given set of items in a particular order.

This is a useful task in its own right, but can also be a subproblem in larger algorithmic problems. Sorting can be applied to problems like duplicate finding (after sorting, equivalent items are adjacent), binary search, and balancing data structures.

The other reason we introduce sorting is that the different sorts provide general ideas about how to approach computational problems. The solution(s) to sorting problems will often involve data structures covered in the earlier parts of this course.

Sorting: Definitions

An **ordering relation** $<$ for keys a, b, and c has the following properties:

- *Law of Trichotomy*: Exactly one of a $<$ b, a = b, b $<$ a is true.
- *Law of Transitivity*: If a $<$ b, and b $<$ c, then a $<$ c.

An ordering relation with the properties above is also known as a **total order**.

A **sort** is a permutation of a sequence of elements that puts the keys into non-decreasing order relative to a given ordering relation, such that x1 ≤ x2 ≤ x3≤ ...≤ xN.

Example: String Length

One example of a ordering relation is the length of strings. To see how the two laws apply:

- *trichotomy*: only one of the following can be true for two strings `a` and `b` -- `len(a)` < `len(b)`, `len(a)` = `len(b)`, or `len(a)` > `len(b)`.

- *transitivity:* if `len(a)` < `len(b)` and `len(b)` < `len(c)`, then clearly `len(a)` < `len(c)`.

Suppose we use the ordering relation above to sort `["cows", "get", "going", "the"]`. Then two valid sorts would be `["the", "get", "cows", "going"]` and `["get", "the", "cows", "going"]`. Note that in this sort, `the` and `get` are equivalent since their lengths are equal.

Ordering Relations in Java

In Java, ordering relations are typically given by the `compareTo` or `compare` methods. For example:

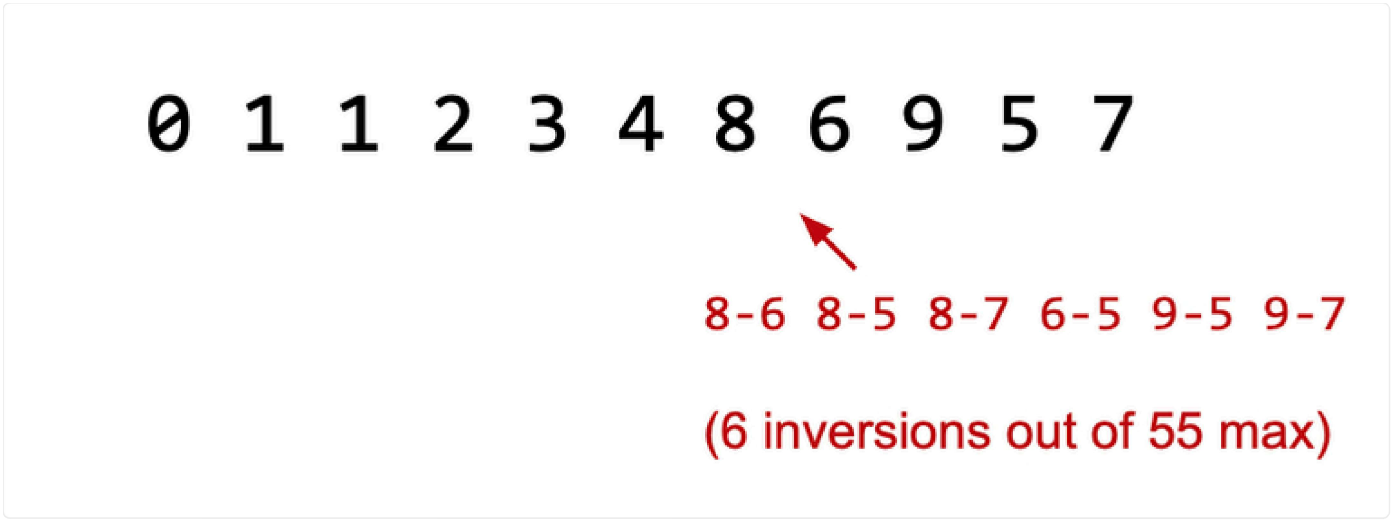
```java
import java.util.Comparator;

public class LengthComparator implements Comparator<String> {
    public int compare(String x, String b) {
        return x.length() - b.length();
    }
}
```

Note by the relation above, `the` and `get` are equal in ordering, but *not* equal by the `.equals()` method.

Inversions

An alternate way to view sorting is as fixing inversions within a sequence of elements. An **inversion** is a pair of elements that are out of order with respect to the defined ordering relation.

For example, in the following sequence of 11 elements, there are 55 possible inversions at most (11 choose 2), and the sequence itself has 6 inversions.



The sequence above has 6 inversions

Sorting, then, can be viewed as follows: given a sequence of elements with Z inversions, perform some sequence of operations to reduce the total number of inversions to zero.

Sorting: Performance

Previously, we have seen characterizations of of the runtime efficiency of an algorithm, also called the **time complexity** of an algorithm. For example, we can say that Dijkstra's has time complexity O(E log V).

Characterizations of the "extra" memory usage of an algorithm is sometimes called the **space complexity** of an algorithm. For example, Dijkstra's has space complexity Θ(V) to store the queue, `distTo`, and `edgeTo` arrays. Note that the graph takes up space Θ(V+E), but we don't count this as part of the space complexity of Dijkstra since the graph is an input to Dijkstra's. In other words, we are only concerned with the *extra* space used by the algorithm.

Previous
29. Basic Sorts

Next
29.2 Selection Sort & Heapsort

Last updated 1 year ago