CSCI 136: Data Structures and Advanced Programming

Lecture 14

Sorting, part II

Instructor: Dan Barowy

Williams

Announcements

- Midterm review session
  Probably Monday evening
  (stay tuned)

Outline

1. Bubblesort implementation
2. Generic sorts
3. Comparator interface
4. Insertion sort
5. Comparable interface

Sorting algorithms

Sorting algorithm

A **sorting algorithm** is a **procedure** for transforming an unordered set of data into an ordered sequence.

A **comparison sorting algorithm** takes as input a set **S** and a binary relation **<** that defines a **strict weak ordering** on **S**.

Strict weak order

A **strict weak order** is a mathematical formalization of the intuitive notion of a **ranking** of a set, some of whose members **may be tied** with each other.

A strict weak order has the following **properties**:

- Irreflexivity: For all **x** in **S**, it is **not the case** that **x < x.**
- Asymmetry: For all **x**, **y** in **S**, where **x ≠ y**, if **x < y** then it is **not the case** that **y < x**.
- Transitivity: For all **x**, **y**, **z** in **S**, where **x ≠ y ≠ z ≠ x**, if **x < y** and **y < z** then **x < z**.
- Transitivity of Incomparability: For all **x**, **y**, **z** in **S**, where **x ≠ y ≠ z ≠ x,** if **x is incomparable** with **y** (neither **x < y** nor **y < x** hold), and **y is incomparable** with **z**, then **x is incomparable** with **z**.

Example order

Example: **lexicographical order** (aka, "dictionary order"):

Given two different sequences of the same length, $a_1a_2...a_k$ and $b_1b_2...b_k$, the first one is smaller than the second one for the lexicographical order, if $a_i<b_i$, for the first i where a_i and b_i differ.

To compare sequences of different lengths, the shorter sequence is padded at the end with "blanks."

Lexicographic order is also totally ordered, which is a stricter order than a weak order (i.e., nothing is incomparable).

In-place sort

An **in-place sort** is a sort that takes an unordered set of elements as an array and **modifies** ("mutates") the original array. Most in-place sorts return `void`.

In principle, in-place sorts can be **faster** than **out-of-place** algorithms, since they do not need to copy data.

Tradeoff: make sure that you don't need the original, unsorted data!

Bubble sort

6 5 3 1 8 7 2 4

Bubble sort

Bubble sort is a **sorting algorithm** in which the largest element **"bubbles up"** during each pass. Bubble sort makes **n-1** passes through the data, performing pairwise comparisons of elements using **<**.

Bubble sort maintains the **invariant** (an always-true logical rule) that the rightmost **n-numSorted** elements are sorted.

I.e., bubble sort builds a sorted order to the right.

Bubble sort complexity

Bubble sort is an **$O(n^2)$** sorting algorithm in the **worst case**. The naive algorithm is also **$O(n^2)$** in the **best case**. With a small modification, bubble sort is **$O(n)$** in the best case (i.e., where the array is already sorted).

(code)

Bubble sort algorithm

```
public static void bubbleSort(int[] data) {
 int n = data.length;
 for (int numSorted = 0; numSorted < n; numSorted++) {
   for (int i = 1; i < n; i++) {
      if (data[i-1] > data[i]) {
        swap(data, i-1, i);
      }
    }
  }
}
```

What if…

… you wanted to sort **arbitrary objects**?

What's **problematic** with our bubble sort implementation?

Comparators

Comparators

We frequently have to sort data that is **more complex** than simple numbers.

For example, suppose we need to sort objects, like a **`People[]`**.

How do we define an order so that we can easily sort this?

`compare` to the rescue.

Comparator interface

The **`Comparator` interface** defines the method **`compare`** that lets us compare **two elements** of the same type.

```
public int compare(T o1, T o2)
```

Returns an `int` **< 0** when `o1` is "less than" `o2`.

Returns an `int` **> 0** when `o2` is "less than" `o1`.

Returns an **0** otherwise.

Insertion sort

6 5 3 1 8 7 2 4

Insertion sort

Insertion sort is a **sorting algorithm** in which the next element is **"inserted"** into a sorted array during each step. Insertion sort makes **n-1** passes through the sorted data, performing pairwise comparisons of elements using **<**.

Insertion sort maintains the **invariant** that the leftmost **n-numSorted** elements are sorted.

I.e., insertion sort builds a sorted order to the left.

Insertion sort algorithm

```
public static void insertionSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0..n-1] are in ascending order
{
    int numSorted = 1;      // number of values in place
    int index;              // general index
    while (numSorted < n)
    {
        // take the first unsorted value
        int temp = data[numSorted];
        // ...and insert it among the sorted:
        for (index = numSorted; index > 0; index--)
        {
            if (temp < data[index-1])
            {
                data[index] = data[index-1];
            } else {
                break;
            }
        }
        // reinsert value
        data[index] = temp;
        numSorted++;
    }
}
```

Insertion sort complexity

Insertion sort is an **$O(n^2)$** sorting algorithm in the **worst case**. Insertion sort is **$O(n)$** in the best case.

Selection sort

(read about this on your own!)

Comparable interface

We frequently have to sort data that is **more complex** than simple numbers.

For example, suppose we need to sort objects, like a **`People[]`**.

How do we define an order so that we can easily sort this?

`compareTo` to the rescue.

Comparable interface

The **`Comparable`** **`interface`** defines the method **`compareTo`** that lets us compare **two elements** of the same type.

```
public int compareTo(T o)
```

Returns an `int` **`< 0`** when `this` is "less than" `o`.

Returns an `int` **`> 0`** when `o` is "less than" `this`.

Returns an **`0`** otherwise.

Recap & Next Class

Today we learned:

More sorting algorithms

Comparators

Next class:

Fast comparison sorts