

Introduction to Computation and Problem Solving

Class 33: Sorting

December 6, 2005

**Prof. Steven R. Lerman
and
Dr. V. Judson Harward**

Why Is Sorting Interesting?

- **Sorting is an operation that occurs as part of many larger programs.**
- **There are many ways to sort, and computer scientists have devoted much research to devising efficient sorting algorithms.**
- **Sorting algorithms illustrate a number of important principles of algorithm design, some of them counterintuitive.**
- **We will examine a series of sorting algorithms in this session and try and discover the tradeoffs between them.**

Sorting Order

- **Sorting implies that the items to be sorted are ordered. Such items should implement the Comparable<T> interface by having a method compareTo(T o) that compares object o with current object.**
- **We will sort Objects of generic type <T> that implement the Comparable<T> interface.**
- **Signature is of form**
`public static <T extends Comparable<T>>
 sortMethod(T[] data)`
- **Classes such as Integer and String implement Comparable<T>. For general objects, we need to provide implementation of compareTo().**

3

Sorting Methods

- **There are times when we want to sort only part of a collection.**
- **A Sort must supply two methods, one to sort an entire array of objects and one to sort a portion of an array specified by start and end elements (a true closed interval, not half open):**

```
public static <T extends Comparable<T>>  
    void sort(T[] d, int start, int end);
```

```
public static <T extends Comparable<T>>  
    void sort(Object[] d);
```

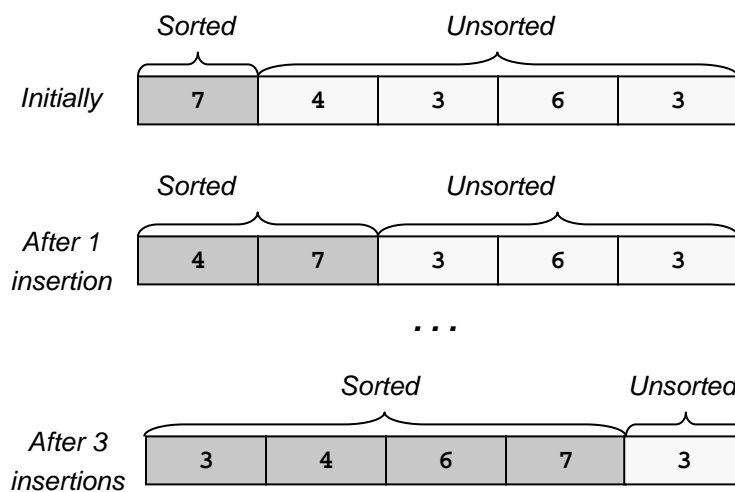
4

Insertion Sort

- Insertion sorting is a formalized version of the way most of us sort cards. In an insertion sort, items are selected one at a time from an unsorted list and inserted into a sorted list.
- It is a good example of an incremental or iterative algorithm. It is simple and intuitive, but we can still do a little to optimize it.
- To save memory and unnecessary copying we sort the array in place.
- We do this by using the low end of the array to grow the sorted list against the unsorted upper end of the array.

5

Insertion Sort Diagram



6

Insertion Sort Code

```
public class Item implements Comparable<Item> {
    public double key;
    public String value;

    public Item(double k, String v) {
        key= k;
        value= v; }

    public int compareTo(Item o) {
        if (key < o.key )
            return -1;
        else if (key > o.key)
            return 1;
        else
            return 0; }
}
```

7

Insertion Sort Code p.2

```
public class InsertionSortMain {
    public static void main(String[] args) {
        Item a= new Item(77.0, "Al");
        Item b= new Item(9.0, "Bob");
        Item c= new Item(22.0, "Cindy");
        Item d= new Item(5.0, "Debra");
        Item[] names= {a, b, c, d};
        insertionSort(names);
        for (int i=0; i < names.length; i++)
            System.out.println(names[i].key + names[i].value); }

    public static <T extends Comparable<T>> void
        insertionSort(T[] data) {
        for (int pos= 1; pos < data.length; pos++) {
            T v= data[pos];
            int i= 0;
            for (i= pos; i>0 && data[i-1].compareTo(v)>0; i--)
                data[i]= data[i-1];
            data[i]= v;
        }
    }
}
```

8

Download Simulations

- Go to the Lecture page on the class web site and download `Sorting.zip`.
- Double click the downloaded file and click the Extract button. Use the file dialog box to choose a place to unpack the simulations we will be using today.

9

Run the `InsertionSort` Simulation

- Navigate to where you unpacked the zip file and double click the `InsertionSort.jar` file in Windows Explorer, not Eclipse. It should bring up the simulation that you will use to examine the algorithm.
- Type in a series of numbers each followed by a return. These are the numbers to sort.
- Click `start` and then `stepInto` to single step through the code.
 - `reset` restarts the current sort.
 - `new` allows you to specify a new sort.

10

InsertionSort Questions

Use the simulator to explore the following questions and let us know when you think you know the answers:

- How many elements have to be moved in the inner `for` loop if you run InsertionSort on an already sorted list? Does it run in $O(1)$, $O(n)$, or $O(n^2)$ time?
- What order of elements will produce the worst performance? Does this case run in $O(1)$, $O(n)$, or $O(n^2)$ time? Why?
- Does the average case have more in common with the best case or the worst case?

11

QuickSort

- Quicksort is a classic and subtle algorithm originally invented by C. A. R. Hoare in 1962.
- There are now endless variations on the basic algorithm, and it is considered the best overall sorting algorithm for industrial-strength applications.

12

QuickSort Strategy

- QuickSort is a divide-and-conquer algorithm. It sorts recursively by performing an operation called *partitioning* on smaller and smaller portions of the array.
- The essential idea of quicksort is to choose an element of the portion of the array to be sorted called the *pivot*. Then the algorithm *partitions* the array with respect to the *pivot*.
- Partitioning means separating the array into two subarrays, the left one containing elements that are less than or equal to the pivot and the right one containing elements greater than or equal to the pivot. The algorithm swaps elements to achieve this condition.

13

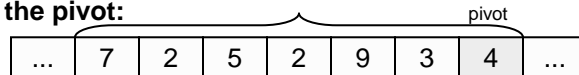
QuickSort Strategy, 2

- Note that these subarrays are not sorted!
- In the version we are using today (but not all QuickSort implementations), we move the pivot between the two partitions.
- Quicksort is then recursively applied to the subarrays.
- The algorithm terminates because subarrays of one element are trivially sorted.

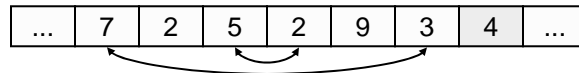
14

QuickSort Single Partition

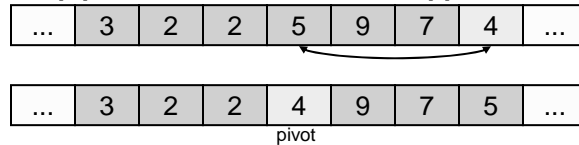
1. Select the last element of the current array segment to be the pivot:



2. Swap elements to fulfill the partition condition:



3. Swap pivot with first element of upper half or partition:



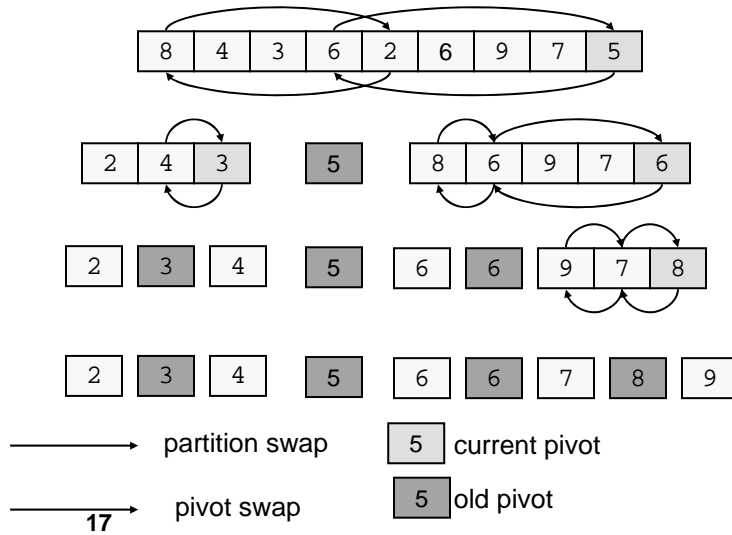
15

Partitioning

- Partitioning is the key step in quicksort.
- In our first version of quicksort, the `pivot` is chosen to be the last element of the (sub)array to be sorted.
- We scan the (sub)array from the left end using index `low` looking for an element \geq the `pivot`.
- When we find one we scan from the right end using index `high` looking for an element \leq `pivot`.
- If `low < high`, we swap them and start scanning for another pair of swappable elements.
- If `low >= high`, we are done and we swap `low` with the `pivot`, which now stands between the two partitions.

16

Recursive Partitioning



Quicksort Simulation

- Double click the `QuickSort.jar` file.
- It works the same way as the `InsertionSort` simulation.

QuickSort Questions

Use the simulator to explore the following questions and let us know when you think you know the answers:

- Why do the `low` and `high` indices stay within the subarray?
- How can we be sure that when the procedure terminates, the subarray is legally partitioned? Can `low` stop at an out of place element without a swap occurring? What if `low` has passed `high`? Why won't `low` be out of place?

19

QuickSort Questions, 2

- What happens when the pivot is the largest or smallest element in the subarray?
- Is quicksort more or less efficient if the pivot is consistently the smallest or the largest element of the (sub)array? What input data could make this happen?
- Why swap `high` and `low` when they are both equal to the pivot? Isn't this unnecessary? What will happen if you try to sort an array where all the elements are equal?

20

A Better Quicksort

- The choice of pivot is crucial to quicksort's performance.
- The ideal pivot is the median of the subarray, that is, the middle member of the sorted array. But we can't find the median without sorting first.
- It turns out that the median of the first, middle and last element of each subarray is a good substitute for the median. It guarantees that each part of the partition will have at least two elements, provided that the array has at least four, but its performance is usually much better. And there are no natural cases that will produce worst case behavior. MedianQuickSort runs in $O(n \log n)$ time.
- Try running MedianQuickSort from `MedianQuickSort.jar`.

21

CountingSort

- InsertionSort and Quicksort both sort by comparing elements. Is there any other way to sort?
- Assume that you are sorting data with a limited set of integer keys that range from 0 to `range`.
- CountingSort sorts the data into a temporary array by taking a "census" of the keys and laying out a directory of where each key must go.

22

Steps in the CountingSort Algorithm

1. Copy the numbers to be sorted to a temporary array.
2. Initialize an array indexed by key values (a histogram of keys) to 0.
3. Iterate over the array to be sorted counting the frequency of each key.
4. Now calculate the cumulative histogram for each key value, k . The first element, $k=0$, is the same as the frequency of key k . The second, $k=1$, is the sum of the frequency for $k=0$ and $k=1$. The third is the sum of the second plus the frequency for $k=2$. And so on.

23

Steps in the CountingSort Algorithm, 2

5. The first element of the cumulative histogram contains the number of elements of the original array with values ≤ 0 . the second, those ≤ 1 . They lay out blocks of values in the sorted array.
6. Starting with the last element in the original array and working back to the first, look up its key in the cumulative histogram to find its destination in the sorted array. It will be the histogram value $- 1$.
7. Decrement the entry in the cumulative histogram so the next key is not stored on top of the first.

24

CountingSort

range = 8

4	3	7	6	4	8	3	5	8
---	---	---	---	---	---	---	---	---

frequencies

0	1	2	3	4	5	6	7	8
0	0	1	2	2	1	1	1	2

cumulative histogram

0	1	2	3	4	5	6	7	8
0	0	1	3	5	6	7	8	10

There are no keys
0, 1

Key 2 occurs 1 time
and should occupy
Position 0.

Key 3 occurs 2 times
and should occupy
positions 1 and 2.

Key 4 occurs 2 times
and should occupy
positions 3 and 4.

Etc.

25

CountingSort Questions

- Double click the `CountingSort.jar` file.
- Note that since CountingSort requires the specification of a range (or two passes over the data), it can't be implemented using our `Sort` interface.
- Experiment with the simulation. Type the numbers to be sorted first. Then enter the range in the range field and press the `start` button. Use `stepinto` to trace the code.
- Does counting sort have a best case or worst case?
- Does it sort in $O(1)$, $O(n)$, or $O(n \log n)$ time? Why?

26