1.00 Lecture 33

Sorting 1

Reading for next time: review Big Java 18.1-18.3

Sorting Order

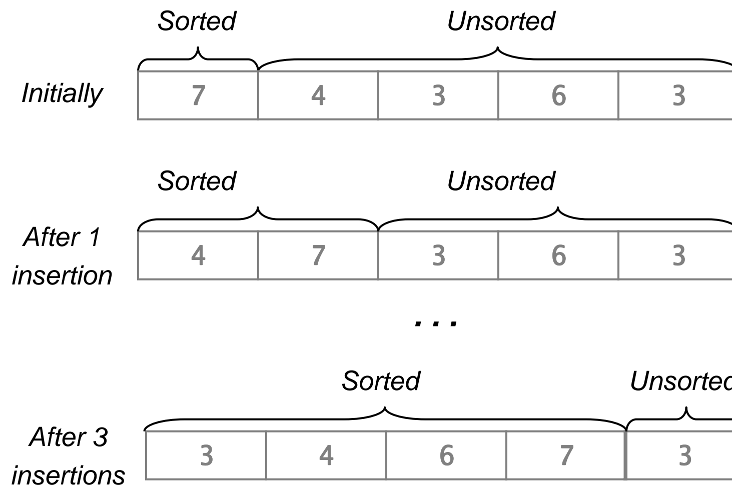
- Sorting implies that the items to be sorted are ordered. We use the same approach that we employed for binary search trees .
- That is, we will sort `Objects`, and whenever we invoke a sort,
  - Either the sort routine can assume that the objects to be sorted belong to a class that implements `Comparable` (method `compareTo`), or
  - We supply a `Comparator` to order the elements to be sorted
    - Comparator interface has a method Compare
- In our examples, we assume the objects implement Comparable

Sort Methods

- We cover three representative methods from a large set:
  - Insertion sort, an 'elementary' method that is slow but ok for small data sets (a few thousand items or less)
  - Quicksort, the fast, standard method for large data sets, but requires some care
  - Counting sort, an even faster method used in a common 'special case'
- We assume the objects we're sorting have two fields:
  - Key, which is the element we sort on (e.g. name)
  - Value, which is the related value we want (e.g., phone number, or email address, or…)

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 elementary sort. It runs slowly, which is ok on small datasets but unacceptable on large ones.
- To save memory and unnecessary copying we sort the array in place.
  - We use the low end of the array to grow the sorted list against the unsorted upper end of the array.



Run the `InsertionSort` Simulation

- **Download `InsertionSort2004.jar`, the simulation that you will use to examine the algorithm, into a convenient folder**
  - **Double-click it in Windows to run it.**
  - **Don't import it into Eclipse.**
- **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 enter a new set of data**

InsertionSort Questions

Use the simulator to explore the following questions :

- 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?

Insertion Sort Code

```
public class InsertionSortTest {

    private static class Item implements Comparable {
        public double key;
        public String value;

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

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

Insertion Sort Code p.2

```
  public static void sort(Comparable[] d) { sort(d, 0, d.length-1);}

  public static void sort(Comparable[] d, int start, int end) {
    Comparable key;
    int i, j;
    for (j= start +1; j <= end; j++) {
      key= d[j];
      for (i= j-1; i >= 0 && key.compareTo(d[i]) < 0; i--)
         d[i+1]= d[i];
      d[i+1]= key;
  }
  public static void main(String[] args) {
    Item b= new Item(3, "Wednesday");
    Item d= new Item(0, "Sunday");
    Item a= new Item(5, "Friday");
    Item c= new Item(2, "Tuesday");
    Item[] days= {a, b, c, d};
    sort(days);
    for (int i=0; i < days.length; i++)
      System.out.println(days[i].key + " " + days[i].value);
} } }
```

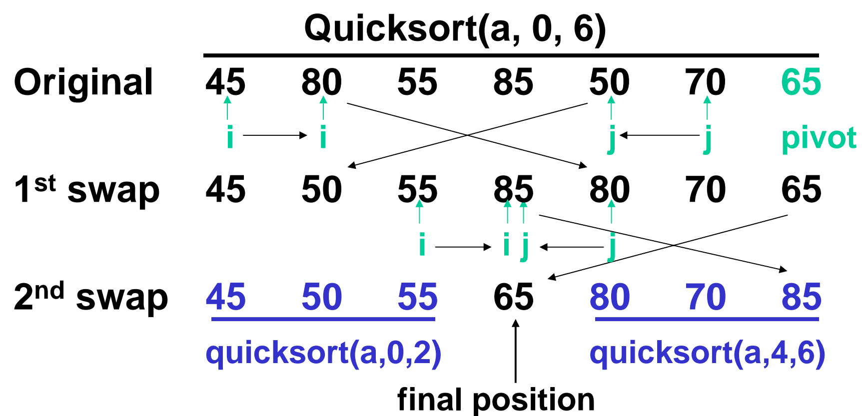
Quicksort overview

- Most efficient general purpose sort, O(n lg n)
- Basic strategy
  - Split array (or vector) of data to be sorted into 2 subarrays so that:
    - Everything in first subarray is smaller than a known value
    - Everything in second subarray is larger than that value
  - Technique is called ‘partitioning’
    - Known value is called the ‘pivot element’
  - Once we’ve partitioned, pivot element will be located in its final position
  - Then we continue splitting the subarrays into smaller subarrays, until the resulting pieces have only one element (recursion!)

Quicksort algorithm

1. Choose an element as pivot. We use right element
2. Start indexes at left and (right-1) elements
3. Move left index until we find an element> pivot
4. Move right index until we find an element < pivot
5. If indexes haven't crossed, swap values and repeat steps 3 and 4
6. If indexes have crossed, swap pivot and left index values
7. Call quicksort on the subarrays to the left and right of the pivot value

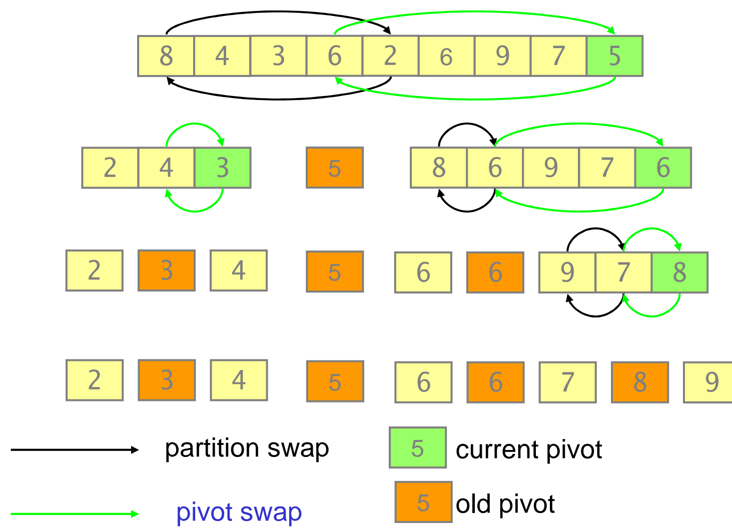
Example



Partitioning

- Partitioning is the key step in quicksort.
- In our 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 >= `pivot`.
- When we find one we scan from the right end using index `high` looking for an element <= `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.

Another Partitioning Example



Quicksort Exercise

- Download and double click the `QuickSort2004.jar` file to run the simulation
  - It works the same way as the InsertionSort simulation.
  - Use the simulator slowly. If you click too quickly it may lose events and show incorrect results. Sorry!
- Quicksort questions:
  - What is the worst case behavior for quicksort?
  - What happens when you quicksort an already-sorted array?

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.
- Other improvements:
  - Convert from recursive to iterative, for efficiency, and always process shortest subarray first (limit stack size, pops, pushes)
  - When subarray is small enough (5-25 elements) use insertion sort; it's faster on small problems

Quicksort main(), exchange

```
import javax.swing.*;
public class QuicksortTest {
    public static void main(String[] args) {
        String input= JOptionPane.showInputDialog("Enter no element");
        int size= Integer.parseInt(input);
        Integer[] sortdata= new Integer[size];
        sortdata[0]= sortdata[size-1]= new Integer(500);
        for (int i=1; i < size-1; i++)
            sortdata[i]= new Integer( (int)(1000*Math.random()));
        sort(sortdata, 0, size-1);
        System.out.println("Done");
        if (size <= 1000)
           for (int i=0; i < size; i++)
               System.out.println(sortdata[i]);
        System.exit(0);
   }
   public static void exchange(Comparable[] a, int i, int j) {
        Comparable o= a[i];                 //  Swaps a[i] and a[j]
        a[i]= a[j];
        a[j]= o;
    }
```

Quicksort, partition

```
   public static int partition(Comparable[] d, int start, int end) {
        Comparable pivot= d[end];          // Partition element
        int low=  start -1;
        int high= end;
        while (true) {
            while ( d[++low].compareTo(pivot) < 0) ;  // Mv indx right
            while ( d[--high].compareTo(pivot) > 0 && high > low); //L
            if (low >= high) break;         // Indexes cross
            exchange(d, low, high);         // Exchange elements
        }
        exchange(d, low, end);              // Exchange pivot, right
        return low;
    }

    public static void sort(Comparable[] d, int start, int end) {
        if (start >= end)
                return;
        int p= partition(d, start, end);
        sort(d, start, p-1);
        sort(d, p+1, end);
    }
 }
```

Quicksort vs Insertionsort

- Demo: sorting random ints:
  - 10,000 elements
  - 1,000,000 elements
- PC speed is about 1GHz or 1 billion ops/second
  - O(n lg n) when n= 1,000,000 is ~20,000,000
  - Each element takes perhaps 40 instructions
  - So quicksort does about 1 billion instructions to sort 1 million elements, which takes about 1 second
  - Insertion sort takes 1,000,000/20, or 50,000 times longer
    - Zzzzzzz…..
- Quicksort with ints rather than Integers or other Objects runs about ten times faster
  - Object overhead (memory allocation)
- No changes needed for quicksort to use Items with keys and values like insertionsort

Exercise

- Finish class StudentTest to sort Students by name
  - We give you a simple Student class (static nested)
    - Data: String name, int year (1-4).
    - Constructor
  - Implement Comparable interface: write compareTo()
    - Use the insertion sort compareTo() example as a guide
    - You can't use < or > to compare Strings
    - Strings implement Comparable, so use that:

```
if (name.compareTo(other.name) < 0)…
```

- We give you most of StudentTest's main() method:
  - We create 5 Student objects, put them in an array for you
  - You must pass the array to the quicksort method
  - We output the results for you

Exercise, p.1

```
public class StudentTest {
   private static class Student implements Comparable{
        private String name;
        private int year;
        public Student(String n, int y) {
                name= n;
                year= y;
        }
        public int compareTo(Object o) {
                // Complete the code
                // Use insertion sort compareTo() as example
        }
   }
```

Exercise, p.2

```
public static void main(String[] args) {
   Student[] sArray= { new Student("Amy  ", 1),
                       new Student("Zane ", 2),
                       new Student("Janet", 3),
                       new Student("Wen  ", 4),
                       new Student("Chris",3) };
   int size= sArray.length;
   // Call the quicksort method here (sort())
   for (int i=0; i < size; i++)
       System.out.println(sArray[i].name+ " " +
              sArray[i].year);
   }
```