CSCI 136
Data Structures &
Advanced Programming

Lecture 13
Spring 2018
Profs Bill & Jon
Administrative Details

- Lab 5 Posted
  - Sorting with Comparators
- Midterm Wednesday March 14
  - Held in your scheduled Lab (same time and place)
  - Study guide and sample exam
  - Review session
Last Time

• The Comparable Interface
  • Including: how to write a generic static method
  • Generic Linear and Binary Search methods

• “Basic” Sorting
  • Bubble sort
Today’s Outline

• “Basic” Sorting Wrapup
  • Bubble, Insertion, Selection Sorts

• Comparator: interface for flexible sorting

• More Efficient Sorting Algorithms
  • MergeSort
  • QuickSort
Basic Sorting Algorithms

• BubbleSort
  • Swaps consecutive elements of $a[0..k]$ until largest element is at $a[k]$; Decrements $k$ and repeats

• InsertionSort
  • Assumes $a[0..k]$ is sorted and moves $a[k+1]$ across $a[0..k]$ until $a[0..k+1]$ is sorted
  • Increments $k$ and repeats

• SelectionSort
  • Finds largest item in $a[0..k]$ and swaps it with $a[k]$
  • Decrements $k$ and repeats
Sorting Preview: Bubble Sort

- Simple sorting algorithm that works by ascending through the list to be sorted, comparing two items at a time, and swapping them if they are in the wrong order
- Repeated until no swaps are needed
- Gets its name from the way larger elements "bubble" to the end of the list
Bubble Sort

5 1 3 2 9

• First Pass:
  - (5 1 3 2 9) → (1 5 3 2 9)
  - (1 5 3 2 9) → (1 3 5 2 9)
  - (1 3 5 2 9) → (1 3 2 5 9)
  - (1 3 2 5 9) → (1 3 2 5 9)

• Second Pass:
  - (1 3 2 5 9) → (1 3 2 5 9)
  - (1 3 2 5 9) → (1 2 3 5 9)
  - (1 2 3 5 9) → (1 2 3 5 9)

• Third Pass:
  - (1 2 3 5 9) -> (1 2 3 5 9)
  - (1 2 3 5 9) -> (1 2 3 5 9)

• Fourth Pass:
  - (1 2 3 5 9) -> (1 2 3 5 9)

http://www.youtube.com/watch?v=lyZQPjUT5B4
Bubble Sort

• Simple sorting algorithm that works by ascending through the list to be sorted, comparing two items at a time, and swapping them if they are in the wrong order
• Repeated until no swaps are needed
• Gets its name from the way larger elements "bubble" to the end of the list
• Time complexity?
  • $O(n^2)$
• Space complexity?
  • $O(n)$ total (no additional space is required)
Sorting Preview: Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Sorted list in low region of the array
- To-be-sorted part in upper region
- Each time you “grow” your sorted region, you swap it backwards into its sorted location
Sorting Preview: Insertion Sort

Red: sorted region.
Each round, swap the first unsorted item back into sorted region.
Sorting Preview: Insertion Sort

• Less efficient on large lists than more advanced algorithms

• Advantages:
  • Simple to implement and efficient on small lists
  • Efficient on data sets which are already substantially sorted

• Time complexity
  • \( O(n^2) \)

• Space complexity
  • \( O(n) \)
The algorithm works as follows:

- Find the maximum value in the list
- Swap it with the value in the last position
- Repeat the steps above for remainder of the list (ending at the second to last position)
Sorting Preview: Selection Sort

- 11 3 27 5 16  Swap 27 with 16
- 11 3 16 5 27  Swap 16 with 5
- 11 3 5 16 27  Swap 11 with 5
- 5 3 11 16 27  Swap 5 with 3
- 3 5 11 16 27  Done!
Sorting Preview: Selection Sort

• Similar to insertion sort
• Performs worse than insertion sort in general
• Noted for its simplicity and performance advantages when compared to complicated algorithms
• Time Complexity:
  • $O(n^2)$
• Space Complexity:
  • $O(n)$
Basic Sorting Algorithms
(All Run in $O(n^2)$ Time)

- **BubbleSort**
  - Always performs $cn^2$ comparisons and might need to perform $cn^2$ swaps

- **InsertionSort**
  - Might need to perform $cn^2$ comparisons and $cn^2$ swaps

- **SelectionSort**
  - Always performs $cn^2$ comparisons but only $O(n)$ swaps
The “Basic” sorts all use a utility method: swap. How would you implement swap?

```java
private static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```
Aside: Lower Bound Notation

**Definition:** A function $f(n)$ is $\Omega(g(n))$ if for some constant $c > 0$ and all $n \geq n_0$

$$f(n) \geq c \cdot g(n)$$

So, $f(n)$ is $\Omega(g(n))$ exactly when $g(n)$ is $O(f(n))$

The previous slide says that all three sorting algorithms have time complexity

- $O(n^2)$: Never use more than $cn^2$ operations
- $\Omega(n^2)$: Sometimes use at least $cn^2$ operations

When $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$ we write:

$f(n)$ is $\Theta(g(n))$
Comparators

- Limitations with Comparable interface?
  - Comparable permits 1 order between objects
  - What if `compareTo()` isn’t the desired ordering?
  - What if `Comparable` isn’t implemented?

- Solution: Comparators
Comparators (Ch 6.8)

- A comparator is an object that contains a method that is capable of comparing two objects
- Sorting methods can be written to apply a Comparator to two objects when a comparison is to be performed
- Different comparators can be applied to the same data to sort in different orders or on different keys

```java
public interface Comparator <E> {
    // pre: a and b are valid objects
    // post: returns a value <, =, or > than 0 determined by
    // whether a is less than, equal to, or greater than b
    public int compare(E a, E b);
}
```
class Patient {
    protected int age;
    protected String name;
    public Patient (String n, int a) { name = n; age = a; }
    public String getName() { return name; }
    public int getAge() { return age; }
}

class NameComparator implements Comparator <Patient>{
    public int compare(Patient a, Patient b) {
        return a.getName().compareTo(b.getName());
    }
    // Note: No constructor; a “do-nothing” constructor is added by Java
}

public void sort(T a[], Comparator<T> c) {
    ...
    if (c.compare(a[i], a[max]) > 0) {...}
}

sort(patients, new NameComparator());
Comparable vs Comparator

• Comparable Interface for class \( X \)
  • Permits just one order between objects of class \( X \)
  • Class \( X \) must implement a `compareTo` method
  • Changing order requires rewriting `compareTo`
    • And then recompiling class \( X \)

• Comparator Interface
  • Allows creation of “compator classes” for class \( X \)
  • Class \( X \) isn’t changed or recompiled
  • Multiple Comparators for \( X \) can be developed
    • Ex: Sort Strings by length (alphabetically for same-length)
    • Ex: Sort names by last name instead of first name
Selection Sort with Comparator

```java
public static <E> int findPosOfMax(E[] a, int last, Comparator<E> c) {
    int maxPos = 0; // A wild guess
    for (int i = 1; i <= last; i++)
        if (c.compare(a[maxPos], a[i]) < 0)
            maxPos = i;
    return maxPos;
}

public static <E> void selectionSort(E[] a, Comparator<E> c) {
    for (int i = a.length - 1; i > 0; i--)
        int big = findPosOfMin(a, i, c);
        swap(a, i, big);
}
```

- The same array can be sorted in multiple ways by passing different Comparator<E> values to the sort method;
Merge Sort

• A divide and conquer algorithm
• Merge sort works as follows:
  • Base case:
    • If the list is of length 0 or 1, then it is already sorted. Return the sorted list.
  • Divide the unsorted list into two sublists of about half the size of original list.
  • Recursive call:
    • Sort each sublist by re-applying merge sort.
  • Merge the two sublists back into one sorted list.
Merge Sort

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

Transylvanian Merge Sort Folk Dance
Merge Sort

• How would we implement it?
• Pseudocode:

```c
// recursively mergesorts A[from..To] “in place”
void recMergeSortHelper(A[], int from, int to)
    if (from < to)
        // find midpoint
        mid = (from + to)/2
        // sort each half
        recMergeSortHelper(A, from, mid)
        recMergeSortHelper(A, mid+1, to)
        // merge sorted lists
        merge(A, from, to)
```

But `merge` hides a number of important details....