CSCI 136
Data Structures &
Advanced Programming

Lecture 13

Fall 2017

Instructors: Bill & Bill

Administrative Details

- Mountain Day Madness!
  - All topics move one lecture earlier until M.D.
  - Problem Set 2 is online
    - Due 11:00pm, Thursday, Oct. 12 (in instructor's mail cubby)
- Reading Period Madness
  - Normal TA hours are in effect
  - Lab 5 will go online this weekend
  - Bring a design document to Lab on Wednesday
    - We'll begin collecting them again

Last Time

- The Comparable Interface
  - Including: how to write a generic static method
  - Generic Linear and Binary Search methods
- Basic Sorting
  - Bubble, Insertion, Selection Sorts

Today's Outline

- Basic Sorting Summary
- Comparator interfaces 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

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

Lower Bound Notation

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

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

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 c n^2 operations
- $\boldsymbol{\Omega}(n^2)$: Sometimes use at least c n^2 operations

When f(n) is O(g(n)) and f(n) is $\boldsymbol{\Omega}$(g(n)) we write

f(n) is $\boldsymbol{\Theta}$(g(n))

Comparators

- Limitations with Comparable interface
  - Only permits one order between objects
  - What if it isn't the desired ordering?
  - What if it 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

```
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);
}
```

Example

Note that Patient does not implement Comparable or Comparator!

```
class Patient {
    protected int age;
    protected String name;
    public Patient (String s, int a) {name = s; 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 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
    - Sort Strings by length (alphabetically for equal-length)

Selection Sort with Comparator

```
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:
  - If the list is of length 0 or 1, then it is already sorted.
  - Divide the unsorted list into two sublists of about half the size of original list.
  - Sort each sublist recursively by re-applying merge sort.
  - Merge the two sublists back into one sorted list.
- Time Complexity?
  - Spoiler Alert! We'll see that it's O(n log n)
- Space Complexity?
  - O(n)

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?
- First pass…

// recursively mergesorts A[from .. To] "in place"

void recMergeSortHelper(A[], int from, int to)

if (from ≤ to)

mid = (from + to)/2

recMergeSortHelper(A, from, mid)

recMergeSortHelper(A, mid+1, to)

merge(A, from, to)

But *merge* hides a number of important details….

Merge Sort

- How would we implement it?
  - Review MergeSort.java
  - Note carefully how temp array is used to reduce copying
  - Make sure the data is in the correct array!
- Time Complexity?
  - Takes at most 2k comparisons to merge two lists of size k
  - Number of splits/merges for list of size n is log n
  - Claim: At most time O(n log n)…We'll see soon...
- Space Complexity?
  - O(n)?
  - Need an extra array, so really O(2n)! But O(2n) = O(n)

Merge Sort = O(n log n)

- [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

log n (split)

log n (merge)

merge takes at most n comparisons per line

Time Complexity is O(n log(n))

- Prove for $n = 2^k$ (true for other n but harder)
- That is, MergeSort for performs at most
  - $n * \log(n) = 2^k * k$ comparisions of elements
- Base case: $k \leq 1$: 0 comparisons: $0 < 1 * 2^1$ ✓
- Induction Step: Suppose true for all integers smaller than k. Let T(k) be # of comparisons for 2^k elements. Then
- $\underline{T(k) \leq} \; 2^k + 2*T(k-1) \leq 2^k + 2(k-1)2^{k-1} \leq \underline{k*2^k}$ ✓

Merge Sort

- Unlike Bubble, Insertion, and Selection sort, Merge sort is a divide and conquer algorithm
  - Bubble, Insertion, Selection sort complexity: $O(n^2)$
  - Merge sort complexity: $O(n \log n)$
- Are there any problems or limitations with Merge sort?
- Why would we ever use any other algorithm for sorting?

Problems with Merge Sort

- Need extra temporary array
  - If data set is large, this could be a problem
- Waste time copying values back and forth between original array and temporary array
- Can we avoid this?

Quick Sort

- Quick sort is designed to behave much like Merge sort, without requiring extra storage space

Merge Sort	Quick Sort
Divide list in half	Partition* list into 2 parts
Sort halves	Sort parts
Merge halves	Join* sorted parts

Recall Merge Sort

```
private static void mergeSortRecursive(Comparable data[],
                        Comparable temp[], int low, int high) {
   int n = high-low+1;
   int middle = low + n/2;
   int i;

   if (n < 2) return;
   // move lower half of data into temporary storage
   for (i = low; i < middle; i++) {
       temp[i] = data[i];
   }
   // sort lower half of array
   mergeSortRecursive(temp,data,low,middle-1);
   // sort upper half of array
   mergeSortRecursive(data,temp,middle,high);
   // merge halves together
   merge(data,temp,low,middle,high);
}
```

Quick Sort

```
// pre: low <= high
// post: data[low..high] in ascending order
public void quickSortRecursive(Comparable data[],
                        int low, int high) {
        int pivot;
        if (low >= high) return;

        /* 1 – split with pivot */
        pivot = partition(data, low, high);
        /* 2 - sort small */
        quickSortRecursive(data, low, pivot-1);
        /* 3 - sort large */
        quickSortRecursive(data, pivot+1, high);
}
```

Partition

1. Put first element (pivot) into sorted position
2. All to the left of "pivot" are smaller and all to the right are larger
3. Return index of "pivot"

Partition by Hungarian Folk Dance

Partition

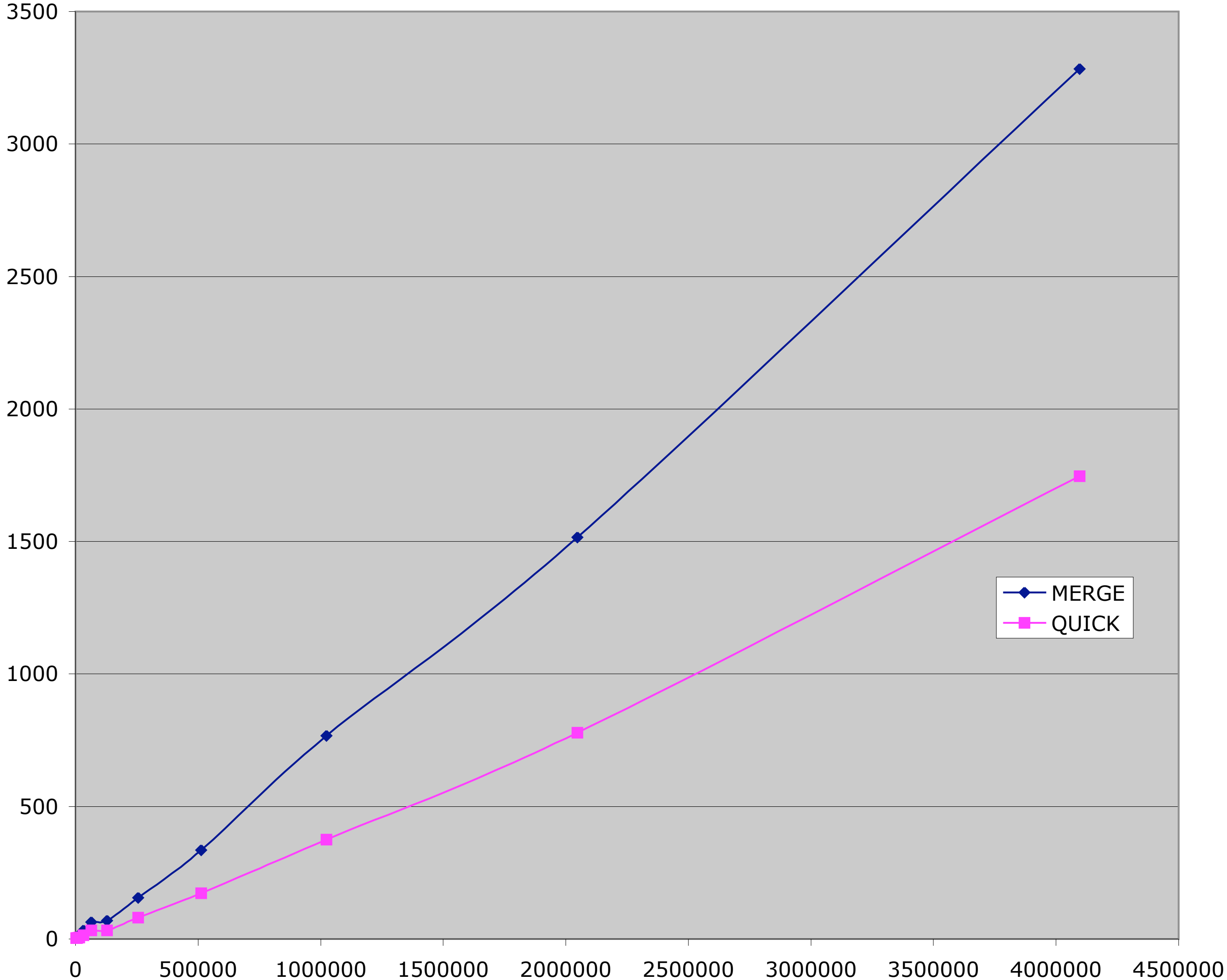
```
int partition(int data[], int left, int right) {
  while (true) {
    while (left < right && data[left] < data[right])
      right--;
    if (left < right) {
      swap(data,left++,right);
    } else {
      return left;
    }

    while (left < right && data[left] < data[right])
      left++;
    if (left < right) {
      swap(data,left,right--);
    } else {
      return right;
    }
  }
}
```

Complexity

- Time:
  - Partition is O(n)
  - If partition breaks list exactly in half, same as merge sort, so O(n log n)
  - If data is already sorted, partition splits list into groups of 1 and n-1, so $O(n^2)$
- Space:
  - O(n) (so is MergSort)
    - In fact, it's n + c compared to 2n + c for MergeSort

Merge vs. Quick



Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot (heuristic!)
- Combine selection sort with quick sort
  - For small n, selection sort is faster
  - Switch to selection sort when elements is <= 7
  - Switch to selection/insertion sort when the list is almost sorted (partitions are very unbalanced)
    - Heuristic!

Sorting Wrapup

	Time	Space
Bubble	Worst: $O(n^2)$ Best: O(n) - if "optimiazed"	O(n) : n + c
Insertion	Worst: $O(n^2)$ Best: O(n)	O(n) : n + c
Selection	Worst = Best: $O(n^2)$	O(n) : n + c
Merge	Worst = Best:: O(n log n)	O(n) : 2n + c
Quick	Average = Best: O(n log n) Worst: $O(n^2)$	O(n) : n + c

More Skill-Testing (Try these at home)

Given the following list of integers:

9 5 6 1 10 15 2 4

1) Sort the list using Bubble sort. Show your work!
2) Sort the list using Insertion sort. . Show your work!
3) Sort the list using Merge sort. . Show your work!
4) Verify the best and worst case time and space complexity for each of these sorting algorithms as well as for selection sort.