CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht
Lecture 12
March 7, 2014

Administrative Details

- Lab 4 is due on Monday
  - Any questions?
  - Should have MyVector.java which extends Vector, Student.java, and a bunch of comparators
  - Main method can be in Student.java (or anywhere really)
- Midterm next Wednesday at 1pm in Wege
  - You'll have ~90 minutes to complete exam
  - Review session (opportunity for you to ask me questions)
    - Probably Tuesday, March 11, 9pm – 10pm in TCL 202?
  - Covers book (Ch 1-7, 9), lab, lecture material through next Monday
  - Closed book and notes
  - Study guide posted on Handouts page after class
- Optional lab (Lab 5) next week
  - It will also be posted on the Handouts page on Monday
- Sample exam will be posted on Handouts page too
  - Not my exam! But it gives you a rough idea of what to expect…
  - Most students find my exams to be more challenging than the sample exam

2

Last Time

- Finished discussing searching
- Learned about Comparators
  - You also used them in lab
- Discussed Bubble and Insertion Sort

3

Today's Outline

- Finish up sorting
- Start learning about Lists
  - Focus on singly linked lists today

4

Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Less efficient on large lists than more advanced sorting 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)

5

Insertion Sort

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

6

Selection Sort

- Similar to insertion sort
- Performs **worse** than insertion sort in general (more comparisons required for each step)
- Noted for its simplicity and performance advantages when compared to complicated algorithms
- 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 (starting at the second to last position)

7

Selection Sort

- 11 3 27 5 16
- 11 3 16 5 27
- 11 3 5 16 27
- 5 3 11 16 27
- 3 5 11 16 27

- Time Complexity:
  - $O(n^2)$
- Space Complexity:
  - O(n)

8

Selection Sort

- How would we implement selection sort?
- Can we prove correctness?

9

Proving Correctness

- Show: recSelSort (a, lastIndex) sorts elements a[0]…a[lastIndex].
- Proof:
  - Base case: lastIndex = 0.
  - IH: For a k<lastIndex, recSelSort sorts a[0]…a[k].
  - Prove for lastIndex:

10

Proving Correctness

- After for loop:
  - a[max] is largest element in a[0]…a[lastIndex]
- That value is moved to a[lastIndex]:
  - Rest of elements are now a[0]…a[lastIndex-1].
- Since lastIndex - 1< lastIndex, recSelSort(0, lastIndex-1) sorts a[0]…a[lastIndex-1].
- Thus a[0]…a[lastIndex-1] are in order, and a[lastIndex] > a[lastIndex-1].
- So, a[0]…a[lastIndex] is sorted.

11

Another Proof

- Prove recSelSort(a,n) requires n(n+1)/2 comparisons (and thus is $O(n^2)$).
- Left as an exercise.
  - See course webpage (lecture notes) for solution.

12

Merge Sort

- Considered 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?
  - O(n log n)
- Space Complexity?
  - O(n)

13

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

14

Merge Sort

- How would we implement it?
- [Merge.java, MergeSort.java]

- 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
  - O(n log n)
- Space Complexity?
  - O(n)?
  - Need an extra array of size O(n), so really O(2n)!
  - But O(2n) = O(n)…

15

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

(split lines: log n; merge lines: log n)

merge takes at most n comparisons per line

16

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?

17

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?

18

Quick Sort

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

Merge Sort	Quick Sort
Split list in half	Split list into sublists
Sort halves	Sort sublists
Merge halves	Combine sorted sublists

19

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);
}
```

20

Quick Sort

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

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

Why don't we need to merge?

21

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"

Look at an example…

22

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;
    }
  }
}
```

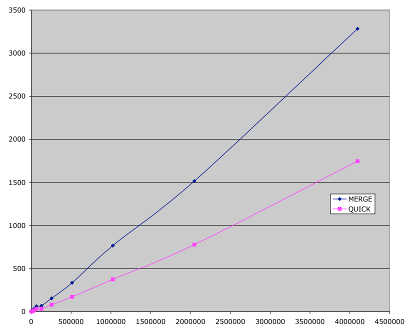
23

Complexity

- Time:
  - Partition is O(n)
  - If partition breaks list exactly in half, same as merge sort, so O(n log n) on average
  - If data is already sorted, partition splits list into groups of 1 and n-1, so $O(n^2)$ in worst case
- Space:
  - O(n) (which is better than O(2n) for merge sort)

24

Merge vs. Quick



25

Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot
- 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)

26

Sorting Wrapup

	Time	Space
Bubble	$O(n^2)$ Best: O(n) - if it stops	O(n)
Insertion	$O(n^2)$ Best: O(n)	O(n)
Selection	$O(n^2)$	O(n)
Merge	O(n log n)	O(2n)
Quick	O(n log n) Worst: $O(n^2)$	O(n)

27