CMPSCI 187: Programming With Data Structures

Lecture #25: Sorting: QuickSort and MergeSort
9 November 2011

Sorting: QuickSort and MergeSort

- Review of O(n^2) Sorting Algorithms
- Divide-And-Conquer Sorting
- QuickSort: Thoughtful Divide, Easy Recombine
- MergeSort: Easy Divide, Thoughtful Recombine
- Why MergeSort is O(n log n)
- Why QuickSort is Kind of O(n log n)
- A Brief Word on Radix Sort

Review of O(n^2) Sorting Algorithms

- We can insert an element into its proper place in an ordered list, taking time O(k) if the list has k elements. **Insertion Sort** creates a sorted list from an arbitrary list by inserting the n elements in turn, taking O(n^2) time in all.

- Finding the smallest element in an unordered list of size k also takes O(k) time. **Selection Sort** takes the smallest, then the next smallest, then the next smallest, and so on, also taking total time O(n^2).

- **Bubble Sort** makes repeated passes over the input, switching adjacent elements if they are out of order -- n - 1 passes of O(n) each for O(n^2) time.

- **Counting Sort** works if the elements are all distinct -- simply compare every element to every other, record how many wins each has, then place the element with i wins in location i of the final array.

Divide-And-Conquer Sorting

- In Binary Search we profited by dividing our search space in half and, with one comparison, reducing our search problem to a similar problem of half the size.

- Sorting can be addressed by dividing our input set into two pieces, recursively solving the two smaller sorting problems, and combining the two resulting sorted lists into a single sorted output list.

- The base case is a list of size 1, which is already sorted. We can hope to get from size n to size 1 with log n levels of recursion. If we spend O(n) time at each level, we get O(n log n) total time, significantly better than O(n^2).

- How we divide and how we recombine will determine two important sorting algorithms, QuickSort and MergeSort.

QuickSort: Thoughtful Divide, Easy Recombine

- If we pick an arbitrary element, called the **pivot**, we can easily divide the other elements into those less than the pivot and those greater than it.

- We move the lesser elements to the left and the greater elements to the right, without worrying about their relative order, then put the pivot between them and recursively sort the two halves. Most of the work of rearranging the elements is done in the method on the next slide.

```
public static <T extends Comparable<? super T>> void
      quickSort (T[ ] data, int min, int max) {
   int index;
   if (max - min > 0) {
      index = findPartition (data, min, max);
      quickSort (data, min, index - 1);
      quickSort (data, index + 1, max);}}
```

The `findPartition` Method

- We return the index of a pivot element, where elements to the left are less than or equal to the pivot and elements to the right are greater than it. This fixes L&C's code on page 228 which is *wrong*.

```
private static <T extends Comparable<? super T>> int
      findPartition (T[ ] data, int min, int max) {
   int left, right;
   T temp, pivot;
   pivot = data [min]; left = min + 1; right = max;
   while (left < right) {
      while (data[left].compareTo(pivot) <= 0 && left < right)
         left++;
      while (data[right].compareTo(pivot) > 0)
         right--;
      if (left < right) {
         temp = data[left];
         data[left] = data[right];
         data[right] = temp;}}
   temp = data[min]; data[min] = data[right]; data[right] = temp;
   return right;}
```

MergeSort: Easy Division, Thoughtful Recombining

- We split the list arbitrarily into two equal halves, recursively sort the two halves, then **merge** the two resulting sorted lists.

- In Project #6 you will create a sorted linked list for each of your `DogTeam` objects, then apply a merging method repeatedly until you have only one list.

- In the array implementation in L&C, we need temporary storage while we read the two sublists off of arrays and put the result into another array. In a linked implementation no extra storage is needed, but for arrays this is why MergeSort can be more expensive than QuickSort.

- Merging is simple -- we look at the next element of each input list, choose the smaller, and move it into the output list. We need some special code to handle the case when we have reached the end of one of the lists.

The Code for MergeSort

- Note that L&C's code on page 230 tries to create an array of Comparables.

```
public static <T extends Comparable<? super T>> void
      mergeSort (T[ ] data, int min, int max) {
   int left, right;
   if (min == max) return;
   int size = max - min + 1; pivot = (min + max)/2;
   T[ ] temp = (T[ ]) (new Object[size]);
   mergeSort (data, min, pivot);
   mergeSort (data, pivot + 1, max);
   for (int i = 0; i < size; i++) temp[i] = data [min + i];
   left = 0; right = pivot - min + 1;
   for (int j = 0; j < size; j++) {
      if (right <= max - min)
         if (left <= pivot - min)
            if (temp[left].compareTo(temp[right]) > 0)
               data[j + min] = temp[right++];
            else data[j + min] = temp[left++];
         else data[j + min] = temp[right++];
      else data[j + min = temp[left++];}}
```

Why MergeSort is O(n log n)

- If we merge two sorted lists, each of which has length at most k, this takes us O(k) time because we do a constant amount of work for each element of either list.

- In a merge sort (roughly speaking), we merge lists of size 1 into lists of size 2, merge those into lists of size 4, then size 8, and so on through log n levels until we have a list of size n. (This is exactly true if n is a power of two, and close to true otherwise.)

- The last merge takes O(n) time. The two merges from size n/4 to size n/2 take O(n/2) time each, for a total of O(n). (I am being sloppy with my big-O arithmetic, in ways that we will ignore until CMPSCI 311.) The four merges from n/8 to n/4 take O(n/4) each, for O(n) total. Each of the log n levels takes O(n) total time, so the overall time is O(n log n).

Why QuickSort is Kind of O(n log n)

- The partition phase of QuickSort divides the list into two pieces. How big the pieces are depends on where the pivot element sits relative to the other elements. If it is near the middle, the pieces are roughly the same size, and if it is near one end, one piece is very small.

- The worst case for QuickSort is when the pivot happens to be the largest or the smallest element, so the larger piece is size n-1. Because we have done O(n) work to find the location of only one element, this will take us O(n^2) in all if it keeps happening. (We are essentially doing a Selection Sort.)

- In CMPSCI 311 we will analyze what happens if the pivot is equally likely to fit anywhere in the list -- we get an **average-case** running time of O(n log n). The idea is that at least half the time, we will reduce the size of the largest piece from k to at most 3k/4, so that in O(log n) phases we will reach the base case. But we have to justify the assumption about the probabilities.

A Brief Word on Radix Sort

- So far all our sorting algorithms have been **comparison-based**, able to work on any type `T` that extends `Comparable<T>`.

- But what about this idea for sorting `String` objects? Make an array with an entry for each possible letter. Divide the input list into one sublist for each *initial* letter, recursively sort each sublist, and combine the results as in QuickSort. We should typically get sublists much smaller than the half-size ones we had before, and the depth of our recursion should be small.

- This would not work so well in the town from the film *Blazing Saddles*, where everyone has the surname “Johnson”.

- This technique was used by the old mechanical card-sorters, that could divide a stack of cards into ten substacks based on any one digit. A trick they discovered was to sort on the *last* digit, then the next-to-last, and so on ending with the first -- this sorts the whole list without needing recursion.