CMPSCI 187: Programming With Data Structures

Lecture #33: Simple Sorting Methods
David Mix Barrington
30 November 2012

Simple Sorting Methods

- The Sorting Problem
- A Sorting Test Harness
- Selection Sort
- Analyzing Selection Sort
- Bubble Sort
- Insertion Sort
- Analyzing Insertion Sort

The Sorting Problem

- We've seen that searching, for example, can be performed much more quickly on a sorted list than on an unsorted one. When we have data in a spreadsheet, sorted using one definition of "smaller", we often want to see what it looks like sorted by some other definition. (Similarly in Project #5 the word-frequency pairs could be sorted alphabetically or by frequency.) It's thus interesting to look at algorithms for **sorting** a list of objects.

- We'll present a number of sorting algorithms in this lecture and the next. All will be **comparison-based**, meaning that the only operation they perform on the items is the `compareTo` method that all `Comparable` objects must have.

- We'll look at the asymptotic running time of each sort, and also at other considerations that might lead us to prefer one algorithm over another. These include memory usage, requirements on access to the data, and behavior in the average case, best case, or typical case.

A Sorting Test Harness

- DJW present a driver program called a test harness that can be used to demonstrate each sorting method and evaluate the number of **comparisons** and **swaps** that it uses on a randomly chosen input. Swaps are our basic means of moving items around within the space they originally occupy.

- The harness will sort sequences of 50 integers, each randomly chosen to be any number in the range from 00 to 99 with equal probability. We will thus usually have sets of equal elements in the sequence. Since the sorting algorithms are comparison-based, they would take the same number of comparisons and swaps on any set of data of the same **order type**.

- The harness has a method `initValues` to populate the array, a method `isSorted` to test whether the array is already sorted, a method `swap` that switches the values at two given indices, and a method `printValues` that gives the 50 values in a table of five rows with 10 values each.

Selection Sort

- A natural way to sort is to find the element that should come first, put it in the first position, and continue by finding the element that should come next.

- We find the next element by simple linear search, then swap it into its correct position. This avoids using memory for both the old and new list.

- This would also be easy to code recursively.

```
static int minIndex(int startIndex, int endIndex) {
   int indexOfMin = startIndex;
   for (int index = startIndex + 1; index < endIndex; index++)
      if (values[index] < values[indexOfMin])
         indexOfMin = index;
   return indexOfMin;}

static void selectionSort( ) {
   int endIndex = SIZE - 1;
   for (int current = 0; current < endIndex; current++)
      swap(current, minIndex(current, endIndex));}
```

Analyzing Selection Sort

- The minIndex method with parameters x and y takes O(y - x) time in the worst case, when the item to be swapped into the front happens to be in the last position. In this case it compares item x against y - x - 1 other elements.

- The total time can't be more than $O(n^2)$, since we have a main loop that we go through n-1 times, and O(y - x) is at worst O(n). But we can figure out the exact number of comparisons in the worst case. (By the way, it turns out that "the worst case" is when the input is initially exactly backwards.) This number is (n-1) + (n-2) + ... + 2 + 1 which evaluates to n(n-1)/2, the n'th "**triangle number**". (Perhaps the easiest way to see this is to add this sum to 1 + 2 + ... + (n-1) term by term, to get that double the sum is n-1 terms each of which is n.

- If swaps are much more expensive than comparisons, selection sort is actually rather good, because it uses only n-1 swaps in the worst case.

Bubble Sort

- Bubble sort is a variant of selection sort where we find the next element in sorted order by first comparing the last two elements, then comparing the lesser of those two with the next to next to last, then comparing the smallest so far with the one before that, and so on. The minimum element in the range we check "bubbles" to the top, and some other elements also move.

- We again use n(n-1)/2 swaps in the worst case. But they are **adjacent swaps** -- we could implement bubble sort on a linked list as well as on an array.

```
static void bubbleUp (int startIndex, int endIndex) {
   for (int index = endIndex; index > startIndex; index--)
      if (values[index] < values[index - 1])
         swap(index, index - 1);}

static void bubbleSort( ) {
   int current = 0;
   while (current < (SIZE - 1)) {
      bubbleUp(current, SIZE - 1);
      current++;}}
```

Insertion Sort

- Another natural way to sort is to gradually build up a sorted list by successively inserting new elements. This is a bit like our reheaping up -- we have a hole at the end of the list, and we move it up by swapping until the new element can go into it.

```
static void insertElement(int startIndex, int endIndex) {
   boolean finished = false;
   int current = endIndex;
   boolean moreToSearch = true;
   while (moreToSearch && !finished) {
      if (values[current] < values[current - 1]) {
         swap(current, current - 1);
         current--;
         moreToSearch = (current != startIndex);}
      else finished = true;}}

static void insertionSort( ) {
   for (int count = 1; count < SIZE; count++)
      insertElement (0, count);}
```

Analyzing Insertion Sort

- The insertion step with parameters x and y also takes y - x - 1 comparisons in the worst case -- it takes that many swaps as well. Again the worst case is a backwards list. The total number of comparisons is again n(n-1)/2, and the total running time is $O(n^2)$.

- Insertion sort has a good best-case behavior -- if the list is already sorted it still does n insertion steps, but each one then takes no swaps and only one comparison, so the total time is O(n).

- One measure of how far from being sorted a list might be is the number of **reversals** -- the number of pairs of elements that are in the wrong relative order. Both bubble sort and insertion sort perform a number of swaps that is equal to the number of reversals, since each adjacent swap fixes exactly one reversal. But if the number of reversals is r, insertion sort takes only O(n+r) time, which is very good if the list is relatively sorted.