Class 40: Sorting, Revisited

Held: Wednesday, 10 November 2010

Summary: We continue our exploration of sorting.

Related Pages:
- EBoard.
- Due: Assignment 7: Sorting Out Sorting, Again.

Notes:
- For those of you who are planning to declare CS majors ... We’d like to encourage you to think about study abroad.
- We hope to have prospectives in class on Friday. Think about what you want to ask them.
- ETHEL concert tonight.
- EC for Thursday extra tomorrow: How did you get placed in Math and Stats and CS?
- Reading for Friday: K&R 7.5-7.5.
- Are there questions on Assignment 7?

Overview:
- Templates for sorting algorithms.
- Review: Some sorting algorithms.
- Choosing a sorting algorithm.
- Lab.

Templates for Sorting Algorithms

As you may recall, we developed a generic template for sorting algorithms.

- You need to define (using #define)
  - TYPE, the type of values that you’re sorting;
  - PREFIX(FUN), a macro that puts a prefix on a function name; and
  - MAYPRECEDEXY, which the routine can use to determine whether or not X may precede Y in the sorted version.
- The declaration of a sorting routine is then simply

```c
void PREFIX(sort)(TYPE values[], int size);
```

K&R take a slightly different perspective. They focus on arrays of pointers (most frequently arrays of strings) and pass in a comparator.
void sort(void *values[], int size, int (*compare)(void *x, void *y));

Is that enough to let us sort arrays of integers or doubles? No.

Can we combine the ideas? Certainly.

void PREFIX(sort)(TYPE values[], int size, int (*compare)(TYPE x, TYPE y));

Is this a good idea? I’ll let you reflect on that question.

Weinman suggests an alternative way of combining the two. We skip the macro technique and pass in the size of objects.

void sort(void *values, int size, int valsize, int (*compare)(void *x, void *y));

Is this a good idea? I’ll also let you reflect on that question.

**Review: Sorting Algorithms**

We’ve sketched out a variety of sorting algorithms.

- **Selection sort:** Repeatedly select the largest (or smallest) remaining element and move it to the appropriate end of the unsorted portion.
- **Insertion sort:** Repeatedly insert elements from the unsorted portion of the array into the proper location of the sorted portion.
- **Merge sort:** Divide into two arrays. Recursively sort each half. Merge back together.
- **Quicksort:** Pick a random pivot. Divide into small and large. Recursively sort each half.
- **Bubble sort:** Step through the array, swapping out-of-place elements.
- **Shell sort:** A variant of insertion sort.

There are also many more.

**Selecting Sorting Algorithms**

Suppose we have a library of correctly implemented sorting routines. How do we decide which one to use?

- An obvious criterion: *Time Efficiency*. We choose the “fastest” algorithm.
- Another obvious criterion: *Space Efficiency*. We choose the algorithm that uses the least additional space.
  - Bye bye merge sort.
- A less obvious criterion: Stability. If A and B are “equal” and A precedes B in the original list, does A precede B in the sorted list?
  - Which sorts seem to be stable?
  - Which sorts seem to be unstable?
- Another less obvious criterion: Range of data. What if I’m sorting by grade?
- It also turns out to be useful to think about the structure of the data
  - Once a day, I sort my huge dataset. Over the next day, I may add ten more things to the dataset.
Once a day, I sort my huge dataset. Over the next day, I may change the values of ten things. (Same assumption.)

Lab

- Continue the lab on pointers to functions.