CS2020: Data Structures and Algorithms (Accelerated)

Discussion Group Problems for Week 5

For: February 6/17, 2011

Problem 1. (Radix Sort Redux.)

We did not have time in lecture to discuss radix sort, so we will look at an example of Radis Sort here in the Discussion Group. The basic idea of Radix Sort is to sort a set of numbers one digit at a time. Assume, for the moment, that we are trying to sort base-10 numbers. We can think of a base-10 number as consisting of a set of digits. For example, the number 5763 contains four digits: 3 is the 1's digit, 6 is the 10's digit, 7 is the 100's digit, and 5 is the 1000's digit.

The basic idea behind radix-sort is to sort a number one digit at a time. For example, imagine I am trying to sort the following list:

456
372
252

We could sort this list by any of the three digits. If we sort it by the 1's digit, then we get:

372
252
456

If we sort the original list by the 10's digit, then we get:

456
252
372

If we sort the original list by the 100's digit, then we get:

252
372
456

Notice that when we sort by some digit, there may be two or more entries that have the some value. In this case, we require our sorting algorithm to be *stable*, i.e., in the output, entries with the same value should appear in the same order as in the input.

For radix-sort, we sort the set of numbers in the input by one digit at a time, beginning with the 1's digit, then the 10's digit, then the 100's digit, etc. (Notice that this may seem backwards from what you would have expected. Think about and discuss why this is the right order in which to perform the digit-by-digit sorting.) Thus, for a d digit number, it will require d invocations of sort to execute a radix-sort.

Problem 1.a. Execute a radix-sort on the following numbers:

352
452
723
854
943
256
982
921

Problem 1.b. Why is it important that the sorting algorithm used for each digit be stable? Think about what happens in the above example if the sorting algorithm is *not* stable.

Problem 1.c. In this example, we are sorting base-10 numbers, where each digit ranges from 0 to 9. (Of course, radix-sort can be used with digits of any size.) Assume you are sorting n base-10 numbers of d-digits each. Moreover, assume you are using CountingSort for sorting each of the digits. What is the asymptotic running time of RadixSort? How much extra storage space is needed (in asymptotic terms)?

Problem 2. (Improving the SkipList.)

In lecture on tuesday, we discussed a strategy for implementing the Select function in a SkipList. Recall that the Select function takes an integer k, and returns the k^{th} smallest element in the SkipList. Modify the SkipList code distributed in class to support the Select function.

Problem 3. (Almost sorted...)

Professor Bitdiddle, in the end, decided to sort the exams. He was carrying the large pile of exams back to his office when, in his absent-minded fashion, he ran into Professor Acker and accidently dropped all the papers. After gathering them all up, he cried in distress, "Oh no! I'll have to resort everything!"

Professor Acker, however, realized that most of the exams remained in the right order. In fact, each exam was within k slots of its correct position. (For example, if some paticular exam would have been at position ℓ in the correctly sorted list, then it is at position greater than $(\ell - k)$ and earlier than $(\ell + k)$ in the almost sorted list.) Professor Acker suggests that the list can be sorted much more quickly via InsertionSort. In this problem, we show that Professor Acker is correct (as usual). For this problem, assume that the array $A[1..n]$ is an array of n distinct elements, and that $k < n/2$.

Problem 3.a. First, we define an *inversion.* If $i < j$ and $A[i] > A[j]$, then we say that the pair (i, j) is an *inversion.* What permutation of the array $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

Problem 3.b. Show that in a list that is almost sorted as described above (i.e., each element is within k slots of its proper position, InsertionSort runs on $O(nk)$ time. *Hint:* First show that InsertionSort runs in time $O(n + I)$, where I is the number of inversions in A.

Problem 3.c. Show, using the decision-tree technique, that sorting such an almost-sorted list via a comaprison-sort takes time at least $O(n \log k)$.

Problem 3.d. (Optional, hard.) Devise an algorithm for sorting such an almost-sorted list in time $O(n \log k)$. (*Hint*: think about how to efficiently merge t lists. One good solution uses a heap.)