Here is an example.

$$\begin{array}{c} 576 \\ 494 \\ 194 \\ 296 \\ 278 \\ 176 \\ 954 \end{array} \Longrightarrow \begin{array}{c} 49[4] \\ 19[4] \\ 95[4] \\ 57[6] \\ 29[6] \\ 17[6] \\ 27[8] \end{array} \Longrightarrow \begin{array}{c} 9[5]4 \\ 5[7]6 \\ 1[7]6 \\ 2[7]8 \\ 4[9]4 \\ 1[9]4 \\ 2[9]6 \end{array} \Longrightarrow \begin{array}{c} [1]76 \\ [1]94 \\ [2]78 \\ [2]96 \\ [4]94 \\ [5]76 \\ [9]54 \end{array} \Longrightarrow \begin{array}{c} 176 \\ 194 \\ 278 \\ 296 \\ 494 \\ 576 \\ 954 \end{array}$$

The running time is clearly $\Theta(d(n+k))$ where d is the number of digits, n is the length of the list, and k is the number of values a digit can have. This is usually a constant, but the algorithm's running time will be $\Theta(dn)$ as long as $k \in O(n)$.

Notice that we can be quite flexible in the definition of what a "digit" is. It can be any number in the range from 1 to cn for some constant c, and we will still have an $\Theta(n)$ time algorithm. For example, if we have $d = 2$ and set $k = n$, then we can sort numbers in the range $n * n = n^2$ in $\Theta(n)$ time. In general, this can be used to sort numbers in the range from 1 to n^d in $\Theta(dn)$ time.

At this point you might ask, since a computer integer word typically consists of 32 bits (4 bytes), then doesn't this imply that we can sort any array of integers in $O(n)$ time (by applying radix sort on each of the $d = 4$ bytes)? The answer is yes, subject to this word-length restriction. But you should be careful about attempting to make generalizations when the sizes of the numbers are not bounded. For example, suppose you have n keys and there are no duplicate values. Then it follows that you need at least $B = \lceil \lg n \rceil$ bits to store these values. The number of bytes is $d = \lceil B/8 \rceil$. Thus, if you were to apply radix sort in this situation, the running time would be $\Theta(dn) = \Theta(n \log n)$. So there is no real asymptotic savings here. Furthermore, the locality of reference behavior of Counting Sort (and hence of RadixSort) is not as good as QuickSort. Thus, it is not clear whether it is really faster to use RadixSort over QuickSort. This is at a level of similarity, where it would probably be best to implement both algorithms on your particular machine to determine which is really faster.

Lecture 18: Review for Second Midterm

(Thursday, Apr 2, 1998)

General Remarks: Up to now we have covered the basic techniques for analyzing algorithms (asymptotics, summations, recurrences, induction), have discussed some algorithm design techniques (divide-and-conquer in particular), and have discussed sorting algorithm and related topics. Recall that our goal is to provide you with the necessary tools for designing and analyzing efficient algorithms.

Material from Text: You are only responsible for material that has been covered in class or on class assignments. However it is always a good idea to see the text to get a better insight into some of the topics we have covered. The relevant sections of the text are the following.

- Review Chapts 1: InsertionSort and MergeSort.
- Chapt 7: Heaps, HeapSort. Look at Section 7.5 on priority queues, even though we didn't cover it in class.
- Chapt 8: QuickSort. You are responsible for the partitioning algorithm which we gave in class, not the one in the text. Section 8.2 gives some good intuition on the analysis of QuickSort.
- Chapt 9 (skip 9.4): Lower bounds on sorting, CountingSort, RadixSort.

- Chapt. 10 (skip 10.1): Selection. Read the analysis of the average case of selection. It is similar to the QuickSort analysis.

You are also responsible for anything covered in class.

Cheat Sheets: The exam is closed-book, closed-notes, but you are allowed two sheets of notes (front and back). You should already have the cheat sheet from the first exam with basic definitions of asymptotics, important summations, Master theorem. Also add Stirling's approximation (page 35), and the integration formula for summations (page 50). You should be familiar enough with each algorithm presented in class that you could work out an example by hand, without refering back to your cheat sheet. But it is a good idea to write down a brief description of each algorithm. For example, you might be asked to show the result of BuildHeap on an array, or show how to apply the Partition algorithm used in QuickSort.

Keep track of algorithm running times and their limitations. For example, if you need an efficient stable sorting algorithm, MergeSort is fine, but both HeapSort and QuickSort are not stable. You can sort short integers in $\Theta(n)$ time through CountingSort, but you cannot use this algorithm to sort arbitrary numbers, such as reals.

Sorting issues: We discussed the following issues related to sorting.

Slow Algorithms: BubbleSort, InsertionSort, SelectionSort are all simple $\Theta(n^2)$ algorithm. They are fine for small inputs. They are all in-place sorting algorithms (they use no additional array storage), and BubbleSort and InsertionSort are stable sorting algorithms (if implemented carefully).

MergeSort: A divide-and-conquer $\Theta(n \log n)$ algorithm. It is stable, but requires additional array storage for merging, and so it is not an in-place algorithm.

HeapSort: A $\Theta(n \log n)$ algorithm which uses a clever data structure, called a heap. Heaps are a nice way of implementing a priority queue data structure, allowing insertions, and extracting the maximum in $\Theta(\log n)$ time, where n is the number of active elements in the heap. Remember that a heap can be built in $\Theta(n)$ time. HeapSort is not stable, but it is an in-place sorting algorithm.

QuickSort: The algorithm is based on selecting a pivot value. If chosen randomly, then the expected time is $\Theta(n \log n)$, but the worst-case is $\Theta(n^2)$. However the worst-case occurs so rarely that people usually do not worry about it. This algorithm is not stable, but it is considered an in-place sorting algorithm even though it does require some additional array storage. It implicitly requires storage for the recursion stack, but the expected depth of the recursion is $O(\log n)$, so this is not too bad.

Lower bounds: Assuming comparisons are used, you cannot sort faster than $\Omega(n \log n)$ time. This is because any comparison-based algorithm can be written as a decision tree, and because there are $n!$ possible outcomes to sorting, even a perfectly balanced tree would require height of at least $O(\log n!) = O(n \log n)$.

Counting sort: If you are sorting n small integers (in the range of 1 to k) then this algorithm will sort them in $\Theta(n + k)$ time. Recall that the algorithm is based on using the elements as indices to an array. In this way it circumvents the lower bound argument.

Radix sort: If you are sorting n integers that have been broken into d digits (each of constant size), you can sort them in $O(dn)$ time.

What sort of questions might there be? Some will ask you to about the properties of these sorting algorithms, or asking which algorithm would be most appropriate to use in a certain circumstance. Others will ask you to either reason about the internal operations of the algorithms, or ask you to extend these algorithms for other purposes. Finally, there may be problems asking you to devise algorithms to solve some sort of novel sorting problem.