Sorting (Chapter 7)

Assumptions:

- Sorting in main memory.
- Arrays contain only integers which allow us to use $>$ and $<$ operations.
- Smallest item first, largest item last.

What do you do if you need to sort large and complex records, but want to avoid moving the records?

You will see $O(n \log n)$ sorting algorithms and show that any general-purpose sorting algorithm requires $\Omega(n \log n)$ comparisons.

Insertion Sort

Invariant: on pass P, elements 1 through P are sorted.
P goes from 1 to n.
What is the worst case for this algorithm?

What is the best case?

How many iterations of the outer loop in insertion sort?

How many iterations of the inner loop?

What is the total work of insertion sort?

Shellsort

Shellsort works by making sure that every k-th element is sorted relative to each other. (See Fig. 7.3)

Choice of k is called the increment for the algorithm, and the big questions is: what is the sequence of increments to choose?

Heapsort: Build a heap of n elements, and then perform n DELETE_MIN operations.

What is the total time for this algorithm?

What do we do with the elements deleted from the heap?

Mergesort

Mergesort runs in $O(n \log n)$ worst-case time.
It is an example of a recursive, divide and conquer algorithm.

The basic idea is to merge two sorted lists.
Because the lists are *already* sorted, this can be done in one pass through the input.

For example, we take two sorted lists, each of length 4, and create a new sort list of length 8.

How can we guarantee that the lists input to the mergesort are already sorted?

For an example, see the example in Section 7.6.
The code for mergesort is in Fig. 7.9.

Can you code this nonrecursively?

Quicksort

For very small data sets ($n \leq 20$), quicksort is not as fast as insertion sort. But quicksort is very fast for large files with an average running time of $O(n \log n)$.

Basic algorithm for sorting an array S:

1. If the number of elements in S is small, then sort S using an insertion sort.
2. Randomly pick an element v in S called the *pivot*.
3. Partition S into the elements smaller than v (S_1) and the elements greater than v (S_2).
4. Return quicksort(S_1), v, quicksort(S_2).

Design decisions: How do you choose a "good" pivot? How is partition implemented?

lower bound for sorting

Below we show that any comparison-based sorting algorithm requires at least $\Omega(n \log n)$ time in the worst case.

To carry out the proof, we will use the notion of a decision tree.
The nodes of the decision tree are sets of possible states and the edges of the tree represent yes or no questions. Let's look at Fig. 7.17 for an example.

Bucket Sort

Despite the general lower bound, linear time sorting is possible in certain special cases.

If each element of the input is smaller than some integer m, we can do a bucket sort.

Assume all elements to be sorted are distinct.

Algorithm:

1. Create an array called C of size n.
2. Insert each element of the input A_i into $C(A_i)$.
3. Scan the array C printing out a sorted list.

What happens if we take away the assumption that each of the elements of the input is distinct?

Radix Sort

Radix sort (covered in Section 3.2) is a generalization of bucket sort.

Radix handles the case where n is so large that a bucket sort is impractical.

The radix sort works by doing a bucket sort on the input numbers *one digit at a time*.

The algorithm starts with the least-significant digit first. Let's look at an example in Figs. 3.24 – 3.26. Note that in the second pass, numbers enter the bucket sorted by their order in the first pass.

So for example, in bucket 2, 125 goes in before 27.

After the final pass, the numbers are read out of the bucket in order.