1 Sorting (Intro)

- Why sort information?
- Binary search depends on sorted information, and if haven’t built a tree structure (we’ve stored our information in a sequence), we need some way to sort it before we search it
- Many things in real-life are sorted (and are too large to be done manually or in an ad hoc fashion), and we don’t even realize it – phone books, textbook indices, email (by email software), physical mail (at the post office)
- Typically, we sort once (or a few times), but search many times, so the time we spend sorting is worthwhile (the phone company sorts the phone book once, millions of people search it millions of times)

2 Terminology

- Sorting indicates that we impose some ordering property upon the set of items we wish to sort, but the ordering property can be almost anything we want (even a trivial one that indicates that we wish to consider all items to be the same, which means nothing gets sorted)
- We say that a sorted sequence is a permutation of the original sequence, since it is a rearrangement of the original items without loss of information
- If a sort treats multiple items as the same (by their ordering property), and those items remain in the same order after the sort (relative to one another), then the sort is stable
- Comparison sorts are sorts that compare elements to each other to determine their place in the sorted sequence.
- Distribution sorts are sorts that look at the elements by themselves, without regard to the other elements, and place them based on their value

3 Insertion Sort

- We start with one unsorted list and one (initially empty) sorted list and take the elements in the unsorted list and insert them one at a time into the sorted list at the correct location.
- If we use a linked list, each insert into the sorted list takes O(1) time, but we have to find the correct location in the sorted list to insert into, which takes O(n)
- If we have an array, each insert into the sorted portion of the array takes O(n) time because we have to shift elements over, but finding the correct place to insert at only takes O(lg n)
- If we sort n items, the insertion sort as a whole will take $O(n^2)$
• If the data is almost sorted, with elements some (constant) distance $d$ or less from the correct position in the to-be-sorted sequence, then it insertion sort is fast – $O(d^4n)$, which is $O(n)$

• We say that the number of inversions in the sequence is the number of pairs of values that are out of order

4 Shell’s Sort

• If items are far from the desired location, we must move them a great distance – instead, we can move them greater distances, thereby reducing the number of inversions faster

• Sort subsets of the sequence by choosing increments of $2^k$ and decreasing $k$ until increments are 1 (which is insertion sort)

• Essentially each subset sort is a run of insertion sort only on those elements

• Because inversions are decreased rapidly, this sort runs in $O(n^{1.25})$ in the average case, $O(n^{1.5})$ in the worst case.

5 Selection Sort

• Even simpler sort – start with an unsorted list and a (initially empty sorted list)

• Find the minimum element in the unsorted list, remove it, and place it at the end of the sorted list

• Finding the minimum element takes $O(n)$ time, and we repeat this for $n$ elements, so total time is $O(n^2)$

• Similarly, for an array, we find the minimum element and place it at the correct location in the sorted portion of the array (and since there is usually an element already there, we swap that element with the one we replaced it with)

6 Heap sort

• A simple way of using heaps to do sorting is to start with an empty min-heap, insert all the elements from the sequence to be sorted into the heap, extract them one at a time, placing them at the end of a list

• Even though it requires extra space, and has to build a heap in the process, it is still $O(n \lg n)$ – we perform $n$ insert operations on the heap, each of which is $O(\lg n)$ and $n$ extract operations on the heap, each of which is $O(\lg n)$, so the entire algorithm is $O(\lg n)$

• A better way, still using heaps – if the input is in an array, do bottom-up heap construction, which is $O(n)$, and then perform the $n$ extract operations. The running time is still $O(n \lg n)$, but the algorithm is in reality faster

7 Announcements

• Review questions posted

• Reading: Data Structures (Into Java) – Ch. 8 - 8.6; Goodrich and Tamassia – 7.2.3
8 Insertion sort (implementation)

// Modified from DSIJ
static void insertionSort(int[] A) {
    int N = A.length;
    for (int i = 1; i < N; i++) {
        int x = A[i];
        int j;
        for (j = 1; j > 0 && x < A[j-1]; j--)
        A[j] = x;
    }
}

9 Selection sort (implementation)

// Modified from DSIJ
static void selectionSort(int[] A) {
    int N = A.length;
    for (int i = 0; i < N-1; i++) {
        int m = i;
        for (int j = i+1; j < N; j++) {
            if (A[j] < A[m])
                m = j;
        }
        swap(A, i, m);
    }
}

10 Heap sort (implementation)

// Modified from DSIJ
static void heapSort(int[] A) {
    if (A.length <= 1)
        return;
    MinHeap H = new MinHeap(A); // use bottom-up build-heap
    for (int i = 0; i < A.length; i++)
        A[i] = H.extractMin();
}