(Still a work-in progress; I want to revisit with intuitive explanations and playing-card examples)

Sorting is a key to CS theory, but easy to forget. I had an itch to review the algorithms in Wikipedia (strange, I know), and here are my notes:

High-level thoughts

- Some algorithms (selection, bubble, heapsort) work by moving elements to their final position, one at a time. You sort an array of size N, put 1 item in place, and continue sorting an array of size N – 1 (heapsort is slightly different).
- Some algorithms (insertion, quicksort, counting, radix) put items into a temporary position, close(r) to their final position. You rescan, moving items closer to the final position with each iteration.
- One technique is to start with a "sorted list" of one element, and merge unsorted items into it, one at a time.
- Complexity and running time
  - Factors: algorithmic complexity, startup costs, additional space requirements, use of recursion (funtion calls are expensive and eat stack space), worst-case behavior, assumptions about input data, caching, and behavior on already-sorted or nearly-sorted data
  - Worst-case behavior is important for real-time systems that need guaranteed performance. For security, you want the guratantee that data from an attacker does not have the ability to overwhelm your machine.
  - Caching — algorithms with sequential comparisons take advantage of spatial locality and prefetching, which is good for caching.
  - Algorithmic time vs. real time — The simple algorithms may be O(N^2), but have low overhead. They can be faster for sorting small data sets (< 10 items). One compromise is to use a different sorting method depending on the input size.
  - "Comparison sorts" make no assumptions on the data and compare all elements against each other (majority of sorts). O(N lg N) time is the ideal "worst-case" scenario (if that makes sense — O(N lg N) is the smallest penalty you can hope for in the worst case). Heapsort has this behavior.
  - O(N) time is possible if we make assumptions about the data and don't need to compare elements against each other (i.e., we know the data falls into a certain range or has some distribution). O(N) clearly is the minimum sorting time possible, since we must examine every element at least once (how can you sort an item you do not even examine?).

Notes

- Assume we are sorting a list or array of N elements
- Once sorted, smaller items are on the left (first item) and larger items are on the right (last item)

Bubble Sort [Best: O(n), Worst:O(N^2)]

Starting on the left, compare adjacent items and keep "bubbling" the larger one to the right (it's in its final place). Bubble sort the remaining N -1 items.

- Though "simple" I found bubble sort nontrivial. In general, sorts where you iterate backwards (decreasing some index) were counter-intuitive for me. With bubble-sort, either you bubble items "forward" (left-to-right) and move the endpoint backwards (decreasing), or

bubble items “backward” (right-to-left) and increase the left endpoint. Either way, some index is decreasing.
- You also need to keep track of the next-to-last endpoint, so you don’t swap with a non-existant item.

Selection Sort [Best/Worst: O(N^2)]

Scan all items and find the smallest. Swap it into position as the first item. Repeat the selection sort on the remaining N-1 items.

- I found this the most intuitive and easiest to implement — you always iterate forward (i from 0 to N-1), and swap with the smallest element (always i).

Insertion Sort [Best: O(N), Worst:O(N^2)]

Start with a sorted list of 1 element on the left, and N-1 unsorted items on the right. Take the first unsorted item (element #2) and insert it into the sorted list, moving elements as necessary. We now have a sorted list of size 2, and N -2 unsorted elements. Repeat for all elements.

- Like bubble sort, I found this counter-intuitive because you step “backwards”
- This is a little like bubble sort for moving items, except when you encounter an item smaller than you, you stop. If the data is reverse-sorted, each item must travel to the head of the list, and this becomes bubble-sort.
- There are various ways to move the item leftwards — you can do a swap on each iteration, or copy each item over its neighbor

Quicksort [Best: O(N lg N), Avg: O(N lg N), Worst:O(N^2)]

There are may versions of Quicksort, which is one of the most popular sorting methods due to its speed (O(N lgN) average, but O(N^2) worst case). Here’s a few:

Using external memory:

- Pick a “pivot” item
- Partition the other items by adding them to a “less than pivot” sublist, or “greater than pivot” sublist
- The pivot goes between the two lists
- Repeat the quicksort on the sublists, until you get to a sublist of size 1 (which is sorted).
- Combine the lists — the entire list will be sorted

Using in-place memory:

- Pick a pivot item and swap it with the last item. We want to partition the data as above, and need to get the pivot out of the way.
- Scan the items from left-to-right, and swap items greater than the pivot with the last item (and decrement the “last” counter). This puts the “heavy” items at the end of the list, a little like bubble sort.
- Even if the item previously at the end is greater than the pivot, it will get swapped again on the next iteration.
- Continue scanning the items until the “last item” counter overlaps the item you are examining – it means everything past the “last item” counter is greater than the pivot.
- Finally, switch the pivot into its proper place. We know the “last item” counter has an item

greater than the pivot, so we swap the pivot there.

- Phew! Now, run quicksort again on the left and right subset lists. We know the pivot is in its final place (all items to left are smaller; all items to right are larger) so we can ignore it.

Using in-place memory w/two pointers:

- Pick a pivot and swap it out of the way
- Going left-to-right, find an oddball item that is greater than the pivot
- Going right-to-left, find an oddball item that is less than the pivot
- Swap the items if found, and keep going until the pointers cross — re-insert the pivot
- Quicksort the left and right partitions
- Note: this algorithm gets confusing when you have to keep track of the pointers and where to swap in the pivot

Notes

- If a bad pivot is chosen, you can imagine that the "less" subset is always empty. That means we are only creating a subset of one item smaller each time, which gives us O(N^2) behavior in the worst case.
- If you choose the first item, it may be the smallest item in a sorted list and give worst-case behavior. You can choose a random item, or median-of-three (front, middle, end).
- Quicksort is *fast* because it uses spatial locality — it walks neighboring elements, comparing them to the pivot value (which can be stored in a register). It makes very effective use of caching.
- The pivot is often swapped to the front, so it is out of the way during the pivotting. Afterwards, it is swapped into place (with a pivot item that is less than or equal to it, so the pivot is preservd).
- The quicksort algorithm is complicated, and you have to pass left and right boundary variables

Heapsort [Best/Avg/Worst: O(N lg N)]

Add all items into a heap. Pop the largest item from the heap and insert it at the end (final position). Repeat for all items.

- Heapsort is just like selection sort, but with a better way to get the largest element. Instead of scanning all the items to find the max, it pulls it from a heap. Heaps have properties that allow heapsort to work in-place, without additional memory.
- Creating the heap is O(N lg N). Popping items is O(1), and fixing the heap after the pop is lgN. There are N pops, so there is another O(N lgN) factor, which is O(N lg N) overall.
- Heapsort has O(N lgN) behavior, even in the worst case, making it good for real-time applications

Counting sort [Best/Avg/Worst: O(N)]

Assuming the data are integers, in a range of 0-k. Create an array of size K to keep track of how many items appear (3 items with value 0, 4 items with value 1, etc). Given this count, you can tell the position of an item — all the 1′s must come after the 0′s, of which there are 3. Therefore, the 1′s start at item #4. Thus, we can scan the items and insert them into their proper position.

- Creating the count array is O(N)
- Inserting items into their proper position is O(N)
- I oversimplified here — there is a summation of the counts, and a greatest-to-least ordering which keeps the sort stable.

Radix sort [Best/Avg/Worst: O(N)]

Get a series of numbers, and sort them one digit at a time (moving all the 1000′s ahead of the 2000′s, etc.). Repeat the sorting on each set of digits.

- Radix sort uses counting sort for efficient O(N) sorting of the digits (k = 0…9)
- Actually, radix sort goes from least significant digit (1′s digit) to most significant, for reasons I'll explain later (see CLRS book)
- Radix & counting sort are fast, but require structured data, external memory and do not have the caching benefits of quicksort.

Actually doing the sorts

For practice, I wrote most of the sorts above in C, based on the psuedocode. Findings

- Even "easy" sorts like bubble sort get complicated with decrements, off-by-one errors, > vs >= as you try to avoid walking off the end of the array with a swap.
- Mocking up the problem on paper is crucial, just like writing the code to swap items in a linked list. Don't have it all in your head.
- I found and fixed bugs in all of my initial sorts. Create a good test harness that makes it *easy* to test.
    - I separated my sorting routines into a DLL (I'm learning how to do Windows programming — it's pretty different from Unix)
    - I created a simple command-line .exe that took a list of numbers, turned them into an array, and called my sorting function, printing the result. This type of testing was encouraged by Kernighan — the tests are easy, do not require compilation (such as hard-coding a "testing" program)
- Because testing was easy, I made every test case I could think of: Pre-sorted forward, backwards, 1 element, 2 elements, even and odd items, etc.
- For debugging, I printed the intermediate array at each stage of the sort.