

Reading: GT 4.5–6, pp. 241–244 [up to 30 minutes]

In this lecture we will survey a multitude of sorting algorithms. Thus it is important to have a good understanding of what is going on in those that you already know: INSERTION-SORT, SELECTION-SORT, HEAP-SORT, MERGE-SORT, and QUICK-SORT.

The new algorithms (based on buckets, and radix-sorting) will be used as another example of talking about the algorithms in a *black-box* fashion. You are advised to understand p. 241–244 for the exam, but for know what is essential for us is the *externally* observable properties of these algorithms (time and space complexity, requirements on input, stability, etc).

Important: Remind yourself from GPP slides what is the minimum number of comparisons a sorting algorithm needs to make.

Note: This note serves for two lectures and is accompanied by another note summarizing this information concisely in a table.

Ingredients: Intro, SELECTION-SORT, INSERTION-SORT, MERGE-SORT, QUICK-SORT, HEAP-SORT, BUCKET-SORT, RADIX-SORT.

This note aims at characterizing several most well known sorting algorithms, without explaining how they work. We aim at users of algorithms, not at designers of new sorting algorithms. The grand goal is to demonstrate that abstract (black-box) reasoning about algorithms can be fruitful and useful. We believe that this is the way most programmers should act when searching for algorithms: focus on properties of algorithms available, not on their internal workings.

We hope to achieve the goal by first explaining the main algorithmic *concepts* relevant for sorting algorithms; second, by giving a guide to sorting algorithms characterizing them by these properties and suitable applications.

General Remarks

Definition 1. A sorting algorithm is called *stable* iff it preserves the relative order of elements with identical input.

Example: MERGE-SORT is stable.

Definition 2. A sorting algorithm is called *internal* if the file to be sorted will fit into memory (random access memory)

Definition 3. A sorting algorithm is called *external* if sorting on tapes or disks (but sometimes also on slow access memory, like flash).

Linked lists sorting is harder than sorting arrays, as linked lists only support sequential access. In this aspect it sorting linked lists is somewhat similar to external sorting.

Definition 4. A sorting algorithm is a *comparison sorting algorithm* if it only inspects keys by comparing them.

Comparison sorting algorithms are convenient for sorting arbitrary objects (for example in Java it suffices that the key object implements the `Comparable` interface).

There exist non-comparison sorting algorithm like *bucket sorting* that exploit the fact that keys are numbers. Such algorithms can be *theoretically* more efficient, up to linear speed.

Pointer sorting (reference sorting) is also often used. Sort pointers to objects instead of objects. This is often more efficient (moving objects is hard). It also allows sorting readonly collections to objects and maintaining views to the same collections in different orders, when required.

Selection-Sort

SELECTION-SORT is insensitive to the preexisting order in data (just a slow on sorted as on random instances). It always run in $O(n^2)$ time.

It is an *in-place* sorting algorithm (uses $O(1)$ extra memory), suitable for sorting linked lists, and stable.

Most importantly SELECTION-SORT minimizes the amount of exchanges (one cannot possibly do better). It is a method of choice for sorting files with huge items and small keys, or whenever comparisons are more than a constant factor faster from exchanges.

Property 5. Selection sort uses about n exchanges.

Property 6. Selection sort uses linear time for files with large items and small keys.

Insertion-Sort

INSERTION-SORT is a quadratic stable in-place sorting algorithm, suitable for sorting linked lists and arrays.

Property 7. INSERTION-SORT uses $O(N^2)$ comparisons and $N^2/4$ exchanges on the average and twice that many at worst.

Insertion-sort is good for presorted files. And we often see presorted files in sorting applications. Note that selection sort is not useful at all for this kind of files.

Consider that each position in a file is close to its final position (this can be achieved by running quick-sort with a constant-size *cut-off*).

Property 8. INSERTION-SORT (...) [uses] a linear number of comparisons and exchanges for files with at most a constant number of inverted ('out-of order') elements in the input.

Now consider that we have a sorted file to which we have appended a constant number of elements, perhaps unsorted.

So INSERTION-SORT can be used as a merge in such circumstances.

Merge-Sort

MERGE-SORT is an easy to implement, optimal stable sorting algorithm suitable for both internal and external sorting.

MERGE-SORT accesses elements in order which makes it perform better on modern machines with caches (in comparison to HEAP-SORT). MERGE-SORT is a method of choice for sorting linked-lists and its main principle easily adapts to external sorting.

MERGE-SORT is recommended for sorting when time guarantees are needed and stability is required, while additional memory use is not a problem.

Runs in $O(n \log n)$ time, uses $O(n)$ extra space.

Property 9. Mergesort is stable, if the underlying MERGE is stable

Property 10. The resource requirements of MERGE-SORT are insensitive to the initial order of its input

MERGE-SORT is input insensitive, so INSERTION-SORT may still be faster on sorted and presorted sequences.

MERGE-SORT is used in perl since 5.8 as the default sorting algorithm. It is also used in Java's `Arrays.sort` to sort object arrays (not arrays of simple values though).

Quick-Sort

QUICK-SORT is a divide-and-conquer based algorithm (like MERGE-SORT): split the problem into two independent subproblems and then solve the subproblems independently. QUICK-SORT is normally used for sorting arrays only.

Property 11. QUICKSORT uses $O(n^2)$ comparisons in the worst case and $O(n^2)$ additional memory.

Nearly all this space is used for storing the recursion call stack. If properly implemented QUICK-SORT can be fixed to use only memory proportional to $\lg N$ in the worst case.

Worst cases for QUICKSORT: sorted sequence, reversely sorted sequence, sequence with keys all identical (for many simple implementations). Many of these can be overcome, but in general we cannot remove the worst-case, we just make it less and less likely.

Property 12. QUICKSORT uses about $O(n \log n)$ comparisons on the average. In such case it also uses memory proportional to $\lg N$

QUICK-SORT is by far the most popular sorting algorithm ever. The standard function `qsort` from ISO C implements QUICKSORT. QUICKSORT is also the sorting algorithm on .NET platform (remember that it is unstable!). Standard Template Library (STL) of C++ uses an algorithm called INTROSORT, which as far as I can remember, is a variation of QUICKSORT. Java's `Array.sort` uses QUICKSORT as default for sorting arrays of simple types (numeric types + characters).

Heap-Sort

HEAP-SORT is an unstable optimal in-place sorting algorithm. It is guaranteed to sort n elements in place in time proportional to $n \log n$. Typically used for

sorting arrays.

Property 13. HEAP-SORT uses $O(n \log n)$ comparisons to sort N elements, and it is in-place ($O(1)$ space).

The guaranteed worst-case performance comes at a price: the inner loop is more complicated than in QUICKSORT and the algorithm performs more comparisons than QUICKSORT for random files. So HEAP-SORT is likely to be slower than QUICKSORT for typical or random files.

The choice between HEAP-SORT and MERGE-SORT essentially reduces to a choice between a sort that is not stable and one that uses extra memory (is not in-place); the choice between HEAP-SORT and QUICKSORT reduces to a choice between average-case speed and worst-case speed.

Bucket-Sort

BUCKET-SORT (AKA COUNT-SORT) sorts only collections of objects with keys taken from a small domain (typically integer numbers from a small interval, or characters). COUNTING-SORT is stable, but not in-place.

BUCKET-SORT is not a comparison sort (it is not comparing the elements with a comparison), but a sort that truly exploits the fact that keys are integer to address an open addressing table of buckets. As such it cannot be used to sort a collection for which only comparisons are available (for example objects implementing the `Comparable` interface in Java).

Property 14. BUCKET-SORT is a linear-time sort provided that the range of distinct key values is within a constant factor of the input size. Otherwise if we sort n elements from range $[0; N]$, BUCKET-SORT runs in $\Theta(n + N)$ time.

BUCKET-SORT uses memory linear in the size of the key domain (N). It can be made in place at the cost of sacrificing stability.

Radix-Sort

RADIX-SORT exploits the fact that comparing objects often does not require comparing the entire keys. For example to find an entry in a telephone book (or in a dictionary) it typically suffices to search by the first several letters (a prefix) of a person's last name.

RADIX-SORT assumes so that its keys are sequences of elements taken from a small domain (for example numbers are sequences of digits, or more typically strings, which are sequences of characters).

RADIX-SORT will be inefficient on long keys that differ mostly on least significant digits, but then if properly implemented, it will not investigate more digits than a comparison sort would. Typically however the comparison operator is efficiently implemented, so your mileage may vary. (several semesters ago one student at ITU implemented a version of RADIX-SORT in Java that was actually faster than the built-in MERGE-SORT for sorting strings in natural language).

One of the worst cases are: all equal.

RADIX-SORT's running time is linear in the number of elements n , size of digit domains N , and in the length of sorted keys d (more precisely it runs in $O((n + N)d)$).