CH 04 : ADVANCED TECHNIQUES IN ALGORITHMS

Part 01 : Sorting algorithms

By Dr. LOUNNAS Bilal

CONTENTS

1 Introduction 2
2 Classification 2
3 Popular sorting algorithms 3
3.1 Simple sorts 3
3.2 Efficient sorts 4
3.3 Bubble sort and variants 5
3.4 Distribution sort 7

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order).
2. The output is a permutation (reordering but with all of the original elements) of the input.

2 CLASSIFICATION

Sorting algorithms are often classified by:

1. Computational complexity (worst, average and best behavior) in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n), with parallel sort in O(log2 n), and bad behavior is O(n2). Ideal behavior for a serial sort is O(n), but this is not possible in the average case. Optimal parallel sorting is O(log n). Comparison-based sorting algorithms need at least O(n log n) comparisons for most inputs.
2. Computational complexity of swaps (for "in-place" algorithms).
3. Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in-place". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(n)) additional memory is considered "in-place".
4. Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
5. Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
6. Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
7. General method: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort. Also whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
8. Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

3 POPULAR SORTING ALGORITHMS

While there are a large number of sorting algorithms, in practical implementations a few algorithms predominate. Insertion sort is widely used for small data sets, while for large data sets an asymptotically efficient sort is used, primarily heap sort, merge sort, or quicksort. Efficient implementations generally use a hybrid algorithm, combining an asymptotically efficient algorithm for the overall sort with insertion sort for small lists at the bottom of a recursion. Highly tuned implementations use more sophisticated variants, such as Timsort (merge sort, insertion sort, and additional logic), used in Android, Java, and Python, and introsort (quicksort and heap sort), used (in variant forms) in some C++ sort implementations and in .NET.

For more restricted data, such as numbers in a fixed interval, distribution sorts such as counting sort or radix sort are widely used. Bubble sort and variants are rarely used in practice, but are commonly found in teaching and theoretical discussions.

When physically sorting objects, such as alphabetizing papers (such as tests or books), people intuitively generally use insertion sorts for small sets. For larger sets, people often first bucket, such as by initial letter, and multiple bucketing allows practical sorting of very large sets. Often space is relatively cheap, such as by spreading objects out on the floor or over a large area, but operations are expensive, particularly moving an object a large distance locality of reference is important. Merge sorts are also practical for physical objects, particularly as two hands can be used, one for each list to merge, while other algorithms, such as heap sort or quick sort, are poorly suited for human use. Other algorithms, such as library sort, a variant of insertion sort that leaves spaces, are also practical for physical use.

3.1 Simple sorts

Two of the simplest sorts are insertion sort and selection sort, both of which are efficient on small data, due to low overhead, but not efficient on large data. Insertion sort is generally faster than selection sort in practice, due to fewer comparisons and good performance on almost-sorted data, and thus is preferred in practice, but selection sort uses fewer writes, and thus is used when write performance is a limiting factor.

3.1.1 *Insertion sort*

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shellsort (see below) is a variant of insertion sort that is more efficient for larger lists.

3.1.2 *Selection sort*

Selection sort is an in-place comparison sort. It has O(n2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

3.2 Efficient sorts

Practical general sorting algorithms are almost always based on an algorithm with average time complexity (and generally worst-case complexity) O(n log n), of which the most common are heap sort, merge sort, and quicksort. Each has advantages and drawbacks, with the most significant being that simple implementation of merge sort uses O(n) additional space, and simple implementation of quicksort has O(n2) worst-case complexity. These problems can be solved or ameliorated at the cost of a more complex algorithm.

While these algorithms are asymptotically efficient on random data, for practical efficiency on real-world data various modifications are used. First, the overhead of these algorithms becomes significant on smaller data, so often a hybrid algorithm is used, commonly switching to insertion sort once the data is small enough. Second, the algorithms often perform poorly on already sorted data or almost sorted data these are common in real-world data, and can be sorted in O(n) time by appropriate algorithms. Finally, they may also be unstable, and stability is often a desirable property in a sort. Thus more sophisticated algorithms are often employed, such as Timsort (based on merge sort) or introsort (based on quicksort, falling back to heap sort).

3.2.1 *Merge sort*

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(n log n). It is also easily applied to lists, not only arrays, as it only requires sequential access, not random access. However, it has additional O(n) space complexity, and involves a large number of copies in simple implementations.

Merge sort has seen a relatively recent surge in popularity for practical implementations, due to its use in the sophisticated algorithm Timsort, which is used for the standard sort routine in the programming languages Python and Java (as of JDK7). Merge sort itself is the standard routine in Perl, among others, and has been used in Java at least since 2000 in JDK1.3.

3.2.2 *Heapsort*

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heapsort to run in O(n log n) time, and this is also the worst case complexity.

3.2.3 *Quicksort*

Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted. This yields average time complexity of O(n log n), with low overhead, and thus this is a popular algorithm. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries.

The important caveat about quicksort is that its worst-case performance is O(n2); while this is rare, in naive implementations (choosing the first or last element as pivot) this occurs for sorted data, which is a common case. The most complex issue in quicksort is thus choosing a good pivot element, as consistently poor choices of pivots can result in drastically slower O(n2) performance, but good choice of pivots yields O(n log n) performance, which is asymptotically optimal. For example, if at each step the median is chosen as the pivot then the algorithm works in O(n log n). Finding the median, such as by the median of medians selection algorithm is however an O(n) operation on unsorted lists and therefore exacts significant overhead with sorting. In practice choosing a random pivot almost certainly yields O(n log n) performance.

3.3 Bubble sort and variants

Bubble sort, and variants such as the cocktail sort, are simple but highly inefficient sorts. They are thus frequently seen in introductory texts, and are of some theoretical interest due to ease of analysis, but they are rarely used in practice, and primarily of recreational interest.

Some variants, such as the Shell sort, have open questions about their behavior.

3.3.1 *Bubble sort*

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average time and worst-case performance is O(n2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any number of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2n time.

3.3.2 *Shell sort*

Shellsort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. The concept behind Shellsort is that both of these algorithms perform in O(kn) time, where k is the greatest distance between two out-of-place elements. This means that generally, they perform in O(n2), but for data that is mostly sorted, with only a few elements out of place, they perform faster. So, by first sorting elements far away, and progressively shrinking the gap between the elements to sort, the final sort computes much faster. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

The worst-case time complexity of Shell sort largely depends on the gap sequence used, and can range from O(n2) to O(n log2 n). Also, unlike efficient sorting algorithms, Shellsort does not require use of the call stack, making it useful in embedded systems where memory is at a premium.

3.3.3 *Comb sort*

Comb sort is a relatively simple sorting algorithm originally designed by Wlodzimierz Dobosiewicz in 1980. It was later rediscovered and popularized by Stephen Lacey and Richard Box with a Byte Magazine article published in April 1991. Comb sort improves on bubble sort. The basic idea is to eliminate turtles, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. (Rabbits, large values around the beginning of the list, do not pose a problem in bubble sort)

3.4 Distribution sort

Distribution sort refers to any sorting algorithm where data are distributed from their input to multiple intermediate structures which are then gathered and placed on the output. For example, both bucket sort and flashsort are distribution based sorting algorithms. Distribution sorting algorithms can be used on a single processor, or they can be a distributed algorithm, where individual subsets are separately sorted on different processors, then combined. This allows external sorting of data too large to fit into a single computer's memory.

3.4.1 *Counting sort*

Counting sort is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in $O(|S| + n)$ time and $O(|S|)$ memory where n is the length of the input. It works by creating an integer array of size $|S|$ and using the ith bin to count the occurrences of the ith member of S in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm often cannot be used because S needs to be reasonably small for the algorithm to be efficient, but it is extremely fast and demonstrates great asymptotic behavior as n increases. It also can be modified to provide stable behavior.

3.4.2 *Bucket sort*

Bucket sort is a divide and conquer sorting algorithm that generalizes counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

A bucket sort works best when the elements of the data set are evenly distributed across all buckets.

3.4.3 *Radix sort*

Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in $O(n * k)$ time. Radix sort can process digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit, and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the counting sort algorithm to be used internally by the radix sort. A hybrid sorting approach, such as using insertion sort for small bins improves performance of radix sort significantly.