Wikipedia

Sorting algorithm

In computer science, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output. More formally, the output of any sorting algorithm 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 (a reordering, yet retaining all of the original elements) of the input.

For optimum efficiency, the input data in fast memory should be stored in a data structure which allows random access rather than one that allows only sequential access.

Sorting algorithms are often referred to as a word followed by the word "sort" and grammatically are used in English as noun phrases, for example in the sentence, "it is inefficient to use insertion sort on large lists" the phrase *insertion sort* refers to the insertion sort sorting algorithm.

Contents

History

Classification
- Stability

Comparison of algorithms
- Comparison sorts
- Non-comparison sorts
- Others

Popular sorting algorithms
- Simple sorts
  - Insertion sort
  - Selection sort
- Efficient sorts
  - Merge sort
  - Heapsort
  - Quicksort
  - Shellsort
- Bubble sort and variants
  - Bubble sort
  - Comb sort
- Distribution sort
  - Counting sort
  - Bucket sort
  - Radix sort

Memory usage patterns and index sorting

Related algorithms

See also

References

Further reading

External links

History

From the beginning of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Among the authors of early sorting algorithms around 1951 was Betty Holberton (born Snyder), who worked on ENIAC and UNIVAC.[1][2] Bubble sort was analyzed as early as 1956.[3] Comparison sorting algorithms have a fundamental requirement of $\Omega(n \log n)$ comparisons (some input sequences will require a multiple of $n \log n$ comparisons, where n is the number of elements in the array to be sorted). Algorithms not based on comparisons, such as counting sort, can have better performance. Asymptotically optimal algorithms have been known since the mid-20th century—useful new algorithms are still being invented, with the now widely used Timsort dating to 2002, and the library sort being first published in 2006.

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time–space tradeoffs, and upper and lower bounds.

Sorting small arrays optimally (in least amount of comparisons and swaps) or fast (i.e. taking into account machine specific details) is still an open research problem, with solutions only known for very small arrays (<20 elements). Similarly optimal (by various definition) sorting on a parallel machine is an open research topic.

Classification

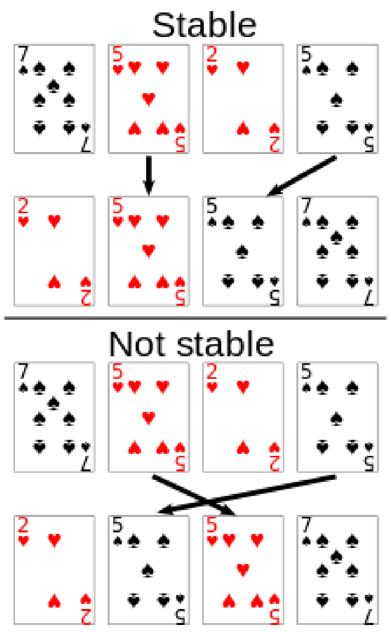
Sorting algorithms are often classified by:

- 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 $O(\log^2 n)$, and bad behavior is $O(n^2)$. (See Big O notation.) 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 $\Omega(n \log n)$ comparisons for most inputs.
- Computational complexity of swaps (for "in-place" algorithms).
- 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".
- Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, *etc.* Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
- Whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
- Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Stability

Stable sort algorithms sort repeated elements in the same order that they appear in the input. When sorting some kinds of data, only part of the data is examined when determining the sort order. For example, in the card sorting example to the right, the cards are being sorted by their rank, and their suit is being ignored. This allows the possibility of multiple different correctly sorted versions of the original list. Stable sorting algorithms choose one of these, according to the following rule: if two items compare as equal, like the two 5 cards, then their relative order will be preserved, so that if one came before the other in the input, it will also come before the other in the output.

Stability is important for the following reason: say that student records consisting of name and class section are sorted dynamically on a web page, first by name, then by class section in a second operation. If a stable sorting algorithm is used in both cases, the sort-by-class-section operation will not change the name order; with an unstable sort, it could be that sorting by section shuffles the name order. Using a stable sort, users can choose to sort by section and then by name, by first sorting using name and then sort again using section, resulting in the name order being preserved. (Some spreadsheet programs obey this behavior: sorting by name, then by section yields an alphabetical list of students by section.)
Another reason: unstable sort may yield different output for the same input from run to run. Such behavior is unsuitable for some applications, for example for client-server applications where the server uses pagination for output and performs a new search-and-sort for every new page requested by the client.



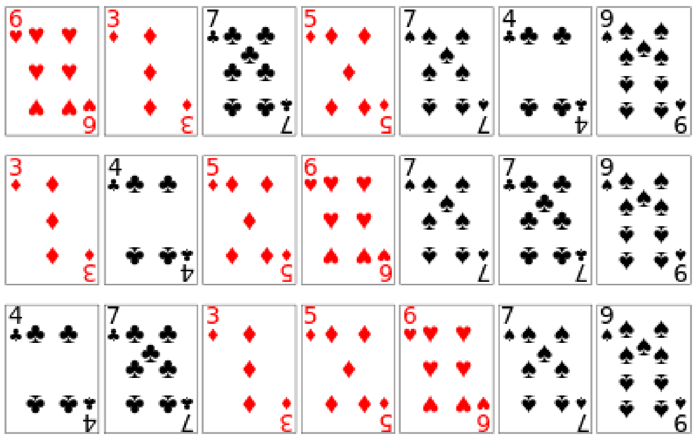
An example of stable sort on playing cards. When the cards are sorted by rank with a stable sort, the two 5s must remain in the same order in the sorted output that they were originally in. When they are sorted with a non-stable sort, the 5s may end up in the opposite order in the sorted output.

More formally, the data being sorted can be represented as a record or tuple of values, and the part of the data that is used for sorting is called the *key*. In the card example, cards are represented as a record (rank, suit), and the key is the rank. A sorting algorithm is stable if whenever there are two records R and S with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list.

When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original input list as a tie-breaker. Remembering this order, however, may require additional time and space.

One application for stable sorting algorithms is sorting a list using a primary and secondary key. For example, suppose we wish to sort a hand of cards such that the suits are in the order clubs (♣), diamonds (♦), hearts (♥), spades (♠), and within each suit, the cards are sorted by rank. This can be done by first sorting the cards by rank (using any sort), and then doing a stable sort by suit:



Within each suit, the stable sort preserves the ordering by rank that was already done. This idea can be extended to any number of keys and is utilised by radix sort. The same effect can be achieved with an unstable sort by using a lexicographic key comparison, which, e.g., compares first by suit, and then compares by rank if the suits are the same.

Comparison of algorithms

In this table, n is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. The run times and the memory requirements listed below should be understood to be inside big O notation, hence the base of the logarithms does not matter; the notation $\log^2 n$ means $(\log n)^2$.

Comparison sorts

Below is a table of comparison sorts. A comparison sort cannot perform better than $O(n \log n)$.[4]

Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have *d* inversions.
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm[9] with a bottom-up merge sort.
Quadsort	n	$n \log n$	$n \log n$	n	Yes	Merging	Uses a 4-input sorting network.[10]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes *n* comparisons when the data is already sorted or reverse sorted.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists.[11]
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes *n* comparisons when the data is already sorted or reverse sorted.
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes.
Library sort	$n \log n$	$n \log n$	n^2	n	No	Insertion	Similar to a gapped insertion sort. It requires randomly permuting the input to warrant with-high-probability time bounds, what makes it not stable.
Patience sorting	n	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[12]	No	Selection	Variation of Heap Sort.
Cocktail shaker sort	n	n^2	n^2	1	Yes	Exchanging	A variant of Bubblesort which deals well with small values at end of list
Comb sort	$n \log n$	n^2	n^2	1	No	Exchanging	Faster than bubble sort on average.
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
UnShuffle Sort[13]	n	kn	kn	n	No	Distribution and Merge	No exchanges are performed. The parameter *k* is proportional to the entropy in the input. *k* = 1 for ordered or reverse ordered input.
Franceschini's method[14]	—	$n \log n$	$n \log n$	1	Yes	?	Performs $O(n)$ data moves.

Odd–even sort	n	n^2	n^2	1	Yes	Exchanging	Can be run on parallel processors easily.

Non-comparison sorts

The following table describes integer sorting algorithms and other sorting algorithms that are not comparison sorts. As such, they are not limited to $\Omega(n \log n)$.[15] Complexities below assume n items to be sorted, with keys of size k, digit size d, and r the range of numbers to be sorted. Many of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that $n \ll 2^k$, where $\ll$ means "much less than". In the unit-cost random access machine model, algorithms with running time of $n \cdot \frac{k}{d}$, such as radix sort, still take time proportional to $\Theta(n \log n)$, because n is limited to be not more than $2^{\frac{k}{d}}$, and a larger number of elements to sort would require a bigger k in order to store them in the memory.[16]

Non-comparison sorts

Name	Best	Average	Worst	Memory	Stable	$n \ll 2^k$	Notes
Pigeonhole sort	—	$n + 2^k$	$n + 2^k$	2^k	Yes	Yes	
Bucket sort (uniform keys)	—	$n + k$	$n^2 \cdot k$	$n \cdot k$	Yes	No	Assumes uniform distribution of elements from the domain in the array.[17]
Bucket sort (integer keys)	—	$n + r$	$n + r$	$n + r$	Yes	Yes	If r is $O(n)$, then average time complexity is $O(n)$.[18]
Counting sort	—	$n + r$	$n + r$	$n + r$	Yes	Yes	If r is $O(n)$, then average time complexity is $O(n)$.[17]
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	Yes	No	$\frac{k}{d}$ recursion levels, 2^d for count array.[17][18]
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	Yes	No	Stable version uses an external array of size n to hold all of the bins.
MSD Radix Sort (in-place)	—	$n \cdot \frac{k}{1}$	$n \cdot \frac{k}{1}$	2^1	No	No	d=1 for in-place, $k/1$ recursion levels, no count array.
Spreadsort	n	$n \cdot \frac{k}{d}$	$n \cdot \left(\frac{k}{s} + d\right)$	$\frac{k}{d} \cdot 2^d$	No	No	Asymptotic are based on the assumption that $n \ll 2^k$, but the algorithm does not require this.
Burstsort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	No	No	Has better constant factor than radix sort for sorting strings. Though relies somewhat on specifics of commonly encountered strings.
Flashsort	n	$n + r$	n^2	n	No	No	Requires uniform distribution of elements from the domain in the array to run in linear time. If distribution is extremely skewed then it can go quadratic if underlying sort is quadratic (it is usually an insertion sort). In-place version is not stable.
Postman sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	—	No	A variation of bucket sort, which works very similar to MSD Radix Sort. Specific to post service needs.

Samplesort can be used to parallelize any of the non-comparison sorts, by efficiently distributing data into several buckets and then passing down sorting to several processors, with no need to merge as buckets are already sorted between each other.

Others

Some algorithms are slow compared to those discussed above, such as the bogosort with unbounded run time and the stooge sort which has $O(n^{2.7})$ run time. These sorts are usually described for educational purposes in order to demonstrate how run time of algorithms is estimated. The following table describes some sorting algorithms that are impractical for real-life use in traditional software contexts due to extremely poor performance or specialized hardware requirements.

Name	Best	Average	Worst	Memory	Stable	Comparison	Other notes
Bead sort	n	S	S	n^2	N/A	No	Works only with positive integers. Requires specialized hardware for it to run in guaranteed $O(n)$ time. There is a possibility for software implementation, but running time will be $O(S)$, where S is sum of all integers to be sorted, in case of small integers it can be considered to be linear.
Simple pancake sort	—	n	n	$\log n$	No	Yes	Count is number of flips.
Spaghetti (Poll) sort	n	n	n	n^2	Yes	Polling	This is a linear-time, analog algorithm for sorting a sequence of items, requiring *O*(*n*) stack space, and the sort is stable. This requires *n* parallel processors. See spaghetti sort#Analysis.
Sorting network	$\log^2 n$	$\log^2 n$	$\log^2 n$	$n \log^2 n$	Varies (stable sorting networks require more comparisons)	Yes	Order of comparisons are set in advance based on a fixed network size. Impractical for more than 32 items.
Bitonic sorter	$\log^2 n$	$\log^2 n$	$\log^2 n$	$n \log^2 n$	No	Yes	An effective variation of Sorting networks.
Bogosort	n	$(n \times n!)$	unbounded (certain), $(n \times n!)$ (expected)	1	No	Yes	Random shuffling. Used for example purposes only, as even the expected best-case runtime is awful.[19]
Stooge sort	$n^{\log 3/\log 1.5}$	$n^{\log 3/\log 1.5}$	$n^{\log 3/\log 1.5}$	n	No	Yes	Slower than most of the sorting algorithms (even naive ones) with a time complexity of $O(n^{\log 3 / \log 1.5}) = O(n^{2.7095...})$.

Theoretical computer scientists have detailed other sorting algorithms that provide better than *O*(*n* log *n*) time complexity assuming additional constraints, including:

- **Thorup's algorithm**, a randomized algorithm for sorting keys from a domain of finite size, taking $O(n \log \log n)$ time and *O*(*n*) space.[20]
- A randomized integer sorting algorithm taking $O\left(n\sqrt{\log \log n}\right)$ expected time and *O*(*n*) space.[21]

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, 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.

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.

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 similar to how we put money in our wallet.[22] 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.

Selection sort

Selection sort is an in-place comparison sort. It has O(n^2) 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.[23] It does no more than *n* swaps, and thus is useful where swapping is very expensive.

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(n^2) 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).

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.[24] 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[25] and Java (as of JDK7[26]). Merge sort itself is the standard routine in Perl,[27] among others, and has been used in Java at least since 2000 in JDK1.3.[28]

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.[29] 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.

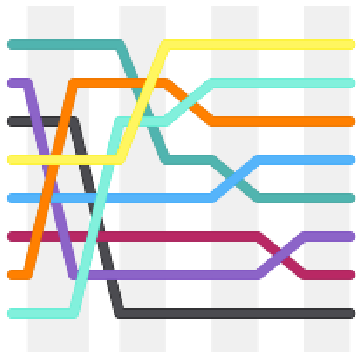
Quicksort

Quicksort is a divide and conquer algorithm which relies on a *partition* operation: to partition an array, an element called a *pivot* is selected.[30][31] 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(n^2); 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(n^2) 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.

Shellsort

Shellsort was invented by Donald Shell in 1959.[32] It improves upon insertion sort by moving out of order elements more than one position at a time. The concept behind Shellsort is that insertion sort performs in $O(kn)$ time, where k is the greatest distance between two out-of-place elements. This means that generally, they perform in $O(n^2)$, 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.


A Shell sort, different from bubble sort in that it moves elements to numerous swapping positions.

The worst-case time complexity of Shellsort is an open problem and depends on the gap sequence used, with known complexities ranging from $O(n^2)$ to $O(n^{4/3})$ and Θ(*n* $\log^2$ *n*). This, combined with the fact that Shellsort is in-place, only needs a relatively small amount of code, and does not require use of the call stack, makes it is useful in situations where memory is at a premium, such as in embedded systems and operating system kernels.

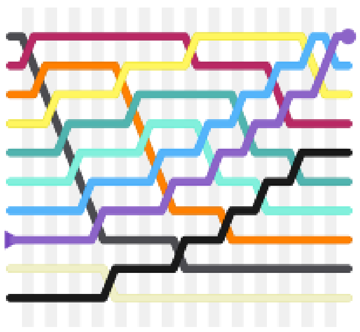
Bubble sort and variants

Bubble sort, and variants such as the shell sort and cocktail sort, are simple, highly inefficient sorting algorithms. They are frequently seen in introductory texts due to ease of analysis, but they are rarely used in practice.

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.[33] This algorithm's average time and worst-case performance is O(n^2), 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 2*n* time.

[34]


A bubble sort, a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order.

Comb sort

Comb sort is a relatively simple sorting algorithm based on bubble sort and originally designed by Włodzimierz Dobosiewicz in 1980.[35] It was later rediscovered and popularized by Stephen Lacey and Richard Box with a *Byte* Magazine article published in April 1991. 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) It accomplishes this by initially swapping elements that are a certain distance from one another in the array, rather than only swapping elements if they are adjacent to one another, and then shrinking the chosen distance until it is operating as a normal bubble sort. Thus, if Shellsort can be thought of as a generalized version of insertion sort that swaps elements spaced a certain distance away from one another, comb sort can be thought of as the same generalization applied to bubble sort.

Distribution sort

Distribution sort refers to any sorting algorithm where data is 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.

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 *i*th bin to count the occurrences of the *i*th 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.

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.

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.

Memory usage patterns and index sorting

When the size of the array to be sorted approaches or exceeds the available primary memory, so that (much slower) disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can dominate the performance characteristics of an algorithm. Thus, the number of passes and the localization of comparisons can be more important than the raw number of comparisons, since comparisons of nearby elements to one another happen at system bus speed (or, with caching, even at CPU speed), which, compared to disk speed, is virtually instantaneous.

For example, the popular recursive quicksort algorithm provides quite reasonable performance with adequate RAM, but due to the recursive way that it copies portions of the array it becomes much less practical when the array does not fit in RAM, because it may cause a number of slow copy or move operations to and from disk. In that scenario, another algorithm may be preferable even if it requires more total comparisons.

One way to work around this problem, which works well when complex records (such as in a relational database) are being sorted by a relatively small key field, is to create an index into the array and then sort the index, rather than the entire array. (A sorted version of the entire array can then be produced with one pass, reading from the index, but often even that is unnecessary, as having the sorted index is adequate.) Because the index is much smaller than the entire array, it may fit easily in memory where the entire array would not, effectively eliminating the disk-swapping problem. This procedure is sometimes called "tag sort".[36]

Another technique for overcoming the memory-size problem is using external sorting, for example one of the ways is to combine two algorithms in a way that takes advantage of the strength of each to improve overall performance. For instance, the array might be subdivided into chunks of a size that will fit in RAM, the contents of each chunk sorted using an efficient algorithm (such as quicksort), and the results merged using a *k*-way merge similar to that used in mergesort. This is faster than performing either mergesort or quicksort over the entire list.[37][38]

Techniques can also be combined. For sorting very large sets of data that vastly exceed system memory, even the index may need to be sorted using an algorithm or combination of algorithms designed to perform reasonably with virtual memory, i.e., to reduce the amount of swapping required.

Related algorithms

Related problems include partial sorting (sorting only the *k* smallest elements of a list, or alternatively computing the *k* smallest elements, but unordered) and selection (computing the *k*th smallest element). These can be solved inefficiently by a total sort, but more efficient algorithms exist, often derived by generalizing a sorting algorithm. The most notable example is quickselect, which is related to quicksort. Conversely, some sorting algorithms can be derived by repeated application of a selection algorithm; quicksort and quickselect can be seen as the same pivoting move, differing only in whether one recurses on both sides (quicksort, divide and conquer) or one side (quickselect, decrease and conquer).

A kind of opposite of a sorting algorithm is a shuffling algorithm. These are fundamentally different because they require a source of random numbers. Shuffling can also be implemented by a sorting algorithm, namely by a random sort: assigning a random number to each element of the list and then sorting based on the random numbers. This is generally not done in practice, however, and there is a well-known simple and efficient algorithm for shuffling: the Fisher–Yates shuffle.

See also

- Collation – Assembly of written information into a standard order
- Schwartzian transform
- Search algorithm – Any algorithm which solves the search problem
- Quantum sort – Sorting algorithms for quantum computers

References

1. "Meet the 'Refrigerator Ladies' Who Programmed the ENIAC" (http://mentalfloss.com/article/53160/meet-refrigerator-ladies-who-programmed-eniac). *Mental Floss*. 2013-10-13. Retrieved 2016-06-16.
2. Lohr, Steve (Dec 17, 2001). "Frances E. Holberton, 84, Early Computer Programmer" (https://www.nytimes.com/2001/12/17/business/frances-e-holberton-84-early-computer-programmer.html). NYTimes. Retrieved 16 December 2014.
3. Demuth, Howard B. (1956). *Electronic Data Sorting* (PhD thesis). Stanford University. ProQuest 301940891 (https://search.proquest.com/docview/301940891).
4. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009), "8", *Introduction To Algorithms* (https://books.google.com/books?id=NLngYyWFl_YC) (3rd ed.), Cambridge, MA: The MIT Press, p. 167, ISBN 978-0-262-03293-3
5. Sedgewick, Robert (1 September 1998). *Algorithms In C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4* (https://books.google.com/books?id=ylAETlep0CwC) (3 ed.). Pearson Education. ISBN 978-81-317-1291-7. Retrieved 27 November 2012.
6. Sedgewick, R. (1978). "Implementing Quicksort programs". *Comm. ACM*. **21** (10): 847–857. doi:10.1145/359619.359631 (https://doi.org/10.1145%2F359619.359631).
7. Ajtai, M.; Komlós, J.; Szemerédi, E. (1983). *An O(n log n) sorting network*. STOC '83. *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. pp. 1–9. doi:10.1145/800061.808726 (https://doi.org/10.1145%2F800061.808726). ISBN 0-89791-099-0.
8. Huang, B. C.; Langston, M. A. (December 1992). "Fast Stable Merging and Sorting in Constant Extra Space" (http://comjnl.oxfordjournals.org/content/35/6/643.full.pdf) (PDF). *Comput. J.* **35** (6): 643–650. CiteSeerX 10.1.1.54.8381 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8381). doi:10.1093/comjnl/35.6.643 (https://doi.org/10.1093%2Fcomjnl%2F35.6.643).
9. Kim, P. S.; Kutzner, A. (2008). *Ratio Based Stable In-Place Merging*. TAMC 2008. *Theory and Applications of Models of Computation*. LNCS. **4978**. pp. 246–257. CiteSeerX 10.1.1.330.2641 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.330.2641). doi:10.1007/978-3-540-79228-4_22 (https://doi.org/10.1007%2F978-3-540-79228-4_22). ISBN 978-3-540-79227-7.
10. https://qiita.com/hon_no_mushi/items/92ff1a220f179b8d40f9
11. "SELECTION SORT (Java, C++) - Algorithms and Data Structures" (http://www.algolist.net/Algorithms/Sorting/Selection_sort). *www.algolist.net*. Retrieved 14 April 2018.
12. http://dbs.uni-leipzig.de/skripte/ADS1/PDF4/kap4.pdf
13. Kagel, Art (November 1985). "Unshuffle, Not Quite a Sort". *Computer Language*. **2** (11).
14. Franceschini, G. (June 2007). "Sorting Stably, in Place, with O(n log n) Comparisons and O(n) Moves". *Theory of Computing Systems*. **40** (4): 327–353. doi:10.1007/s00224-006-1311-1 (https://doi.org/10.1007%2Fs00224-006-1311-1).
15. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "8", *Introduction To Algorithms* (https://books.google.com/books?id=NLngYyWFl_YC) (2nd ed.), Cambridge, MA: The MIT Press, p. 165, ISBN 0-262-03293-7
16. Nilsson, Stefan (2000). "The Fastest Sorting Algorithm?" (http://www.drdobbs.com/architecture-and-design/the-fastest-sorting-algorithm/184404062). *Dr. Dobb's*.
17. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7.
18. Goodrich, Michael T.; Tamassia, Roberto (2002). "4.5 Bucket-Sort and Radix-Sort". *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons. pp. 241–243. ISBN 978-0-471-38365-9.
19. Gruber, H.; Holzer, M.; Ruepp, O., "Sorting the slow way: an analysis of perversely awful randomized sorting algorithms", *4th International Conference on Fun with Algorithms, Castiglioncello, Italy, 2007* (http://www.hermann-gruber.com/pdf/fun07-final.pdf) (PDF), Lecture Notes in Computer Science, **4475**, Springer-Verlag, pp. 183–197, doi:10.1007/978-3-540-72914-3_17 (https://doi.org/10.1007%2F978-3-540-72914-3_17).
20. Thorup, M. (February 2002). "Randomized Sorting in O(n log log n) Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations". *Journal of Algorithms*. **42** (2): 205–230. doi:10.1006/jagm.2002.1211 (https://doi.org/10.1006%2Fjagm.2002.1211).
21. Han, Yijie; Thorup, M. (2002). *Integer sorting in O(n√(log log n)) expected time and linear space*. The 43rd Annual IEEE Symposium on Foundations of Computer Science. pp. 135–144. doi:10.1109/SFCS.2002.1181890 (https://doi.org/10.1109%2FSFCS.2002.1181890). ISBN 0-7695-1822-2.
22. Wirth, Niklaus (1986), *Algorithms & Data Structures*, Upper Saddle River, NJ: Prentice-Hall, pp. 76–77, ISBN 978-0130220059
23. Wirth 1986, pp. 79–80

24. Wirth 1986, pp. 101–102
25. "Tim Peters's original description of timsort" (http://svn.python.org/projects/python/trunk/Objects/listsort.txt). *python.org*. Retrieved 14 April 2018.
26. "OpenJDK's TimSort.java" (http://cr.openjdk.java.net/~martin/webrevs/openjdk7/timsort/raw_files/new/src/share/classes/java/util/TimSort.java). *java.net*. Retrieved 14 April 2018.
27. "sort - perldoc.perl.org" (http://perldoc.perl.org/functions/sort.html). *perldoc.perl.org*. Retrieved 14 April 2018.
28. Merge sort in Java 1.3 (http://java.sun.com/j2se/1.3/docs/api/java/util/Arrays.html#sort(java.lang.Object%5B%5D)), Sun. Archived (https://web.archive.org/web/20090304021927/http://java.sun.com/j2se/1.3/docs/api/java/util/Arrays.html#sort(java.lang.Object%5B%5D)#sort(java.lang.Object%5B%5D)) 2009-03-04 at the Wayback Machine
29. Wirth 1986, pp. 87–89
30. Wirth 1986, p. 93
31. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009), *Introduction to Algorithms* (3rd ed.), Cambridge, MA: The MIT Press, pp. 171–172, ISBN 978-0262033848
32. Shell, D. L. (1959). "A High-Speed Sorting Procedure" (http://penguin.ewu.edu/cscd300/Topic/AdvSorting/p30-shell.pdf) (PDF). *Communications of the ACM*. **2** (7): 30–32. doi:10.1145/368370.368387 (https://doi.org/10.1145%2F368370.368387).
33. Wirth 1986, pp. 81–82
34. "kernel/groups.c" (https://github.com/torvalds/linux/blob/72932611b4b05bbd89fafa369d564ac8e449809b/kernel/groups.c#L105). Retrieved 2012-05-05.
35. Brejová, B. (15 September 2001). "Analyzing variants of Shellsort". *Inf. Process. Lett.* **79** (5): 223–227. doi:10.1016/S0020-0190(00)00223-4 (https://doi.org/10.1016%2FS0020-0190%2800%2900223-4).
36. "tag sort Definition from PC Magazine Encyclopedia" (https://www.pcmag.com/encyclopedia_term/0,2542,t=tag+sort&i=52532,00.asp). *www.pcmag.com*. Retrieved 14 April 2018.
37. Donald Knuth, *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998, ISBN 0-201-89685-0, Section 5.4: External Sorting, pp. 248–379.
38. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures*, H. Freeman & Co., ISBN 0-7167-8042-9.

Further reading

- Knuth, Donald E. (1998), *Sorting and Searching*, The Art of Computer Programming, **3** (2nd ed.), Boston: Addison-Wesley, ISBN 0-201-89685-0
- Sedgewick, Robert (1980), "Efficient Sorting by Computer: An Introduction", *Computational Probability* (https://archive.org/details/computationalpro00actu/page/101), New York: Academic Press, pp. 101–130 (https://archive.org/details/computationalpro00actu/page/101), ISBN 0-12-394680-8

External links

- Sorting Algorithm Animations (https://web.archive.org/web/20150303022622/http://www.sorting-algorithms.com/) at the Wayback Machine (archived 3 March 2015)
- Sequential and parallel sorting algorithms (http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm) – explanations and analyses of many sorting algorithms
- Dictionary of Algorithms, Data Structures, and Problems (https://www.nist.gov/dads/) – dictionary of algorithms, techniques, common functions, and problems
- Slightly Skeptical View on Sorting Algorithms (http://www.softpanorama.org/Algorithms/sorting.shtml) – Discusses several classic algorithms and promotes alternatives to the quicksort algorithm
- 15 Sorting Algorithms in 6 Minutes (Youtube) (https://www.youtube.com/watch?v=kPRA0W1kECg) – visualization and "audibilization" of 15 Sorting Algorithms in 6 Minutes
- A036604 sequence in OEIS database titled "Sorting numbers: minimal number of comparisons needed to sort n elements" (https://oeis.org/A036604) – which performed by Ford–Johnson algorithm
- Sorting Algorithms Used on Famous Paintings (Youtube) (https://www.youtube.com/watch?v=d2d0r1bArUQ) - Visualization of Sorting Algorithms on Many Famous Paintings.