WIKIPEDIA

Comparison sort

A **comparison sort** is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation (often a "less than or equal to" operator or a three-way comparison) that determines which of two elements should occur first in the final sorted list. The only requirement is that the operator obey two of the properties of a total order:

1. if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)
2. for all a and b, either $a \leq b$ or $b \leq a$ (totalness or trichotomy).

It is possible that both $a \leq b$ and $b \leq a$; in this case either may come first in the sorted list. In a stable sort, the input order determines the sorted order in this case.

Sorting a set of unlabelled weights by weight using only a balance scale requires a comparison sort algorithm

A metaphor for thinking about comparison sorts is that someone has a set of unlabelled weights and a balance scale. Their goal is to line up the weights in order by their weight without any information except that obtained by placing two weights on the scale and seeing which one is heavier (or if they weigh the same).

Contents

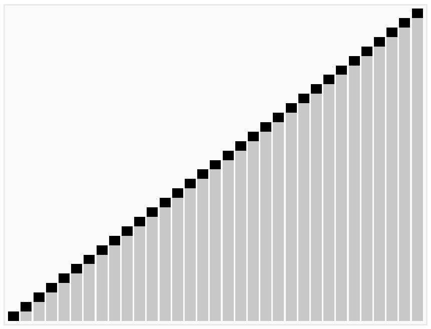
Examples

Some of the most well-known comparison sorts include:

- Quicksort
- Heapsort
- Shellsort
- Merge sort
- Introsort
- Insertion sort
- Selection sort
- Bubble sort

- Odd–even sort
- Cocktail shaker sort
- Cycle sort
- Merge insertion (Ford–Johnson) sort
- Smoothsort
- Timsort

Performance limits and advantages of different sorting techniques


Quicksort in action on a list of numbers. The horizontal lines are pivot values.

There are fundamental limits on the performance of comparison sorts. A comparison sort must have an average-case lower bound of $\Omega(n \log n)$ comparison operations,[1] which is known as linearithmic time. This is a consequence of the limited information available through comparisons alone — or, to put it differently, of the vague algebraic structure of totally ordered sets. In this sense, mergesort, heapsort, and introsort are asymptotically optimal in terms of the number of comparisons they must perform, although this metric neglects other operations. Non-comparison sorts (such as the examples discussed below) can achieve $O(n)$ performance by using operations other than comparisons, allowing them to sidestep this lower bound (assuming elements are constant-sized).

Note that comparison sorts may run faster on some lists; many adaptive sorts such as insertion sort run in $O(n)$ time on an already-sorted or nearly-sorted list. The $\Omega(n \log n)$ lower bound applies only to the case in which the input list can be in any possible order.

Also note that real-world measures of sorting speed may need to take into account the ability of some algorithms to optimally use relatively fast cached computer memory, or the application may benefit from sorting methods where sorted data begins to appear to the user quickly (and then user's speed of reading will be the limiting factor) as opposed to sorting methods where no output is available for display until the whole list is sorted.

Despite these limitations, comparison sorts offer the notable practical advantage that control over the comparison function allows sorting of many different datatypes and fine control over how the list is sorted. For example, reversing the result of the comparison function allows the list to be sorted in reverse; and one can sort a list of tuples in lexicographic order by just creating a comparison function that compares each part in sequence:

```
function tupleCompare((lefta, leftb, leftc), (righta, rightb, rightc))
    if lefta ≠ righta
        return compare(lefta, righta)
    else if leftb ≠ rightb
        return compare(leftb, rightb)
    else
        return compare(leftc, rightc)
```

Balanced ternary notation allows comparisons to be made in one step, whose result will be one of "less than", "greater than" or "equal to".

Comparison sorts generally adapt more easily to complex orders such as the order of floating-point numbers. Additionally, once a comparison function is written, any comparison sort can be used without modification; non-comparison sorts typically require specialized versions for each datatype.

This flexibility, together with the efficiency of the above comparison sorting algorithms on modern computers, has led to widespread preference for comparison sorts in most practical work.

Alternatives

Some sorting problems admit a strictly faster solution than the $\Omega(n \log n)$ bound for comparison sorting; an example is integer sorting, where all keys are integers. When the keys form a small (compared to n) range, counting sort is an example algorithm that runs in linear time. Other integer sorting algorithms, such as radix sort, are not asymptotically faster than comparison sorting, but can be faster in practice.

The problem of sorting pairs of numbers by their sum is not subject to the $\Omega(n^2 \log n)$ bound either (the square resulting from the pairing up); the best known algorithm still takes $O(n^2 \log n)$ time, but only $O(n^2)$ comparisons.

Number of comparisons required to sort a list

The number of comparisons that a comparison sort algorithm requires increases in proportion to $n\log(n)$, where n is the number of elements to sort. This bound is asymptotically tight.

Given a list of distinct numbers (we can assume this because this is a worst-case analysis), there are *n* factorial permutations exactly one of which is the list in sorted order. The sort algorithm must gain enough information from the comparisons to identify the correct permutation. If the algorithm always completes after at most $f(n)$ steps, it cannot distinguish more than $2^{f(n)}$ cases because the keys are distinct and each comparison has only two possible outcomes. Therefore,

$$2^{f(n)} \geq n!, \text{ or equivalently } f(n) \geq \log_2(n!).$$

By looking at the first $n/2$ factors of $n! = n(n-1)\cdots 1$, we obtain

$$\log_2(n!) \geq \log_2\left((n/2)^{n/2}\right) = n/2\log n - n/2 = \Omega(n\log n).$$

This provides the lower-bound part of the claim. A better bound can be given via Stirling's approximation.

An identical upper bound follows from the existence of the algorithms that attain this bound in the worst case.

The above argument provides an *absolute*, rather than only asymptotic lower bound on the number of comparisons, namely $\lceil\log_2(n!)\rceil$ comparisons. This lower bound is fairly good (it can be approached within a linear tolerance by a simple merge sort), but it is known to be inexact. For example, $\lceil\log_2(13!)\rceil = 33$, but the minimal number of comparisons to sort 13 elements has been proved to be 34. Determining the *exact* number of comparisons needed to sort a given number of entries is a computationally hard problem even for small *n*, and no simple formula for the solution is known. For some of the few concrete values that have been computed, see A036604.

Lower bound for the average number of comparisons

A similar bound applies to the average number of comparisons. Assuming that

- all keys are distinct, i.e. every comparison will give either $a>b$ or $a<b$, and
- the input is a random permutation, chosen uniformly from the set of all possible permutations of *n* elements,

it is impossible to determine which order the input is in with fewer than $\log_2(n!)$ comparisons on average.

This can be most easily seen using concepts from information theory. The Shannon entropy of such a random permutation is $\log_2(n!)$ bits. Since a comparison can give only two results, the maximum amount of information it provides is 1 bit. Therefore, after *k* comparisons the remaining entropy of the permutation, given the results of those comparisons, is at least $\log_2(n!) - k$ bits on average. To perform the sort, complete information is needed, so the remaining entropy must be 0. It follows that *k* must be at least $\log_2(n!)$.

Note that this differs from the worst case argument given above, in that it does not allow rounding up to the nearest integer. For example, for $n = 3$, the lower bound for the worst case is 3, the lower bound for the average case as shown above is approximately 2.58, while the highest lower bound for the average case is 8/3, approximately 2.67.

In the case that multiple items may have the same key, there is no obvious statistical interpretation for the term "average case", so an argument like the above cannot be applied without making specific assumptions about the distribution of keys.

n	$\lceil \log_2(n!) \rceil$	**Minimum**
1	0	0
2	1	1
3	3	3
4	5	5
5	7	7
6	10	10
7	13	13
8	16	16
9	19	19
10	22	22
11	26	26
12	29	30[2][3]
13	33	34[4][5][6]
14	37	38[6]
15	41	42[7][8][9]
16	45	45 or 46[10]
17	49	49 or 50
18	53	53 or 54
19	57	58[9]
20	62	62
21	66	66
22	70	71[6]
n	$\lceil \log_2(n!) \rceil$	$n \log_2 n - \frac{n}{\ln 2}$
10	22	19
100	525	521
1 000	8 530	8 524
10 000	118 459	118 451
100 000	1 516 705	1 516 695
1 000 000	18 488 885	18 488 874

Above: A comparison of the lower bound $\lceil \log_2(n!) \rceil$ to the actual minimum number of comparisons (from A036604) required to sort a list of *n* items (for the worst case). Below: Using Stirling's approximation, this

Notes

1. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. pp. 191–193. ISBN 0-262-03384-4.
2. Mark Wells, Applications of a language for computing in combinatorics, Information Processing 65 (Proceedings of the 1965 IFIP Congress), 497–498, 1966.
3. Mark Wells, Elements of Combinatorial Computing, Pergamon Press, Oxford, 1971.
4. Takumi Kasai, Shusaku Sawato, Shigeki Iwata, Thirty four comparisons are required to sort 13 items, LNCS 792, 260-269, 1994.
5. Marcin Peczarski, Sorting 13 elements requires 34 comparisons, LNCS 2461, 785–794, 2002.
6. Marcin Peczarski, New results in minimum-comparison sorting, Algorithmica 40 (2), 133–145, 2004.
7. Marcin Peczarski, Computer assisted research of posets, PhD thesis, University of Warsaw, 2006.
8. Peczarski, Marcin (2007). "The Ford-Johnson algorithm is still unbeaten for less than 47 elements". *Inf. Process. Lett*. **101** (3): 126–128. doi:10.1016/j.ipl.2006.09.001 (https://doi.org/10.1016%2Fj.ipl.2006.09.001).
9. Cheng, Weiyi; Liu, Xiaoguang; Wang, Gang; Liu, Jing (October 2007). "最少比较排序问题中S（15）和S（19）的解决" (http://fcst.ceaj.org/EN/abstract/abstract47.shtml) [The results of S(15) and S(19) to minimum-comparison sorting problem]. *Journal of Frontiers of Computer Science and Technology* (in Chinese). **1** (3): 305–313.
10. Peczarski, Marcin (3 August 2011). "Towards Optimal Sorting of 16 Elements". *Acta Universitatis Sapientiae*. **4** (2): 215–224. arXiv:1108.0866 (https://arxiv.org/abs/1108.0866). Bibcode:2011arXiv1108.0866P (http://adsabs.harvard.edu/abs/2011arXiv1108.0866P).

References

- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1997.

ISBN 0-201-89685-0. Section 5.3.1: Minimum-Comparison Sorting, pp. 180–197.

lower bound is well-approximated by $n \log_2 n - \frac{n}{\ln 2}$.