It is possible to prove that $O(n \log n)$ is the best possible time complexity of a general purpose sorting algorithm. The assumption here is that the algorithm determines the correct order by comparing elements to each other.

Let us consider a situation where the algorithm is given a list of length n, where each element in the range $1 \dots n$ occurs exactly once. There are $n!$ possibilities for the initial order of the elements, and in each case the algorithm has to do a different set of moves to accomplish the sorting.

Whenever the algorithm compares two elements to each other, it obtains some information that can be used in sorting. Consider a situation, where $n = 3$ and the algorithm has not made any comparisons yet. Since the algorithm has no information about the order yet, it has to consider all of the following orders as possible:

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

Suppose then that the algorithm compares the first two elements of the list to find out that the first element is bigger than the second element. Now the algorithm can narrow down the set of possible orders to the following:

$$[2, 1, 3], [3, 1, 2], [3, 2, 1]$$

For the algorithm to operate correctly it has to continue comparisons until the set of possible orders is reduced to a single order. Initially, there are $n!$ possible orders, and after each comparison, at least half of the orders might remain possible. Thus the algorithm may need at least $\log_2(n!)$ comparisons before it has enough information to complete the sorting. This gives us the lower bound

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n) \geq (n/2) \log_2(n/2)$$

which means that the algorithm may need at least $O(n \log n)$ comparisons.

Notice that sorting can be done more efficiently if the type of elements is restricted. For example, if the elements are small integers, sorting can be done in

$O(n)$ time. One such algorithm is *counting sort*.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI