1 Introduction

Formal definition of sorting and the property of being sorted... Sorting: reorganising a list, A,such that if $A_i < A_j$ then i < j must be true.

If there exists any pair of elements in a collection A, at positions i and j, such that $i < j$ but $A_i > A_j$ – with respect to whatever comparator function is relevant – that pair of elements is known as an inversion. The degree of disorder or "unsortedness" of a collection of elemetns is measured by the number of inversions present.

To be sorted: Each item in the collection is less than or equal to its successor.

Equal-valued elements must be contiguous; i.e. if $A_i = A_j$ then there must be no k such that i < j < k and $A_i \neq A_k$

The contents of a collection, A, must be the same before and after searching; i.e. the sorted collection A must be a permuattion fo the pre-sorted collection A.

$<$, $=$, $\neq$, and $>$ can be interpreted in terms of mathematical equality or any othey arbtrary but well defined system of ordering. It should be possible to define a *comparator function* which can take two elements, say a and b, and return a value based on whether $a < b$, $a > b$, or $a = b$.

Sorting algorithms, in general, operate independently of the precise definitions of *less than*, *greater than*, and *equal to* differing instead in how they go about making comparisons between pairs of elements with the goal of a completely sorted collection.

The particular problem's defintion of equivalence is encoded in the comparator function and the precise nature of the comparator function is irrelevant to the sorting algorithm which merely requires a black box through which it can pass two values and which returns a codification of the equivalence of those values.

More concretely; the following pseudocode demonstrates a comparator function which for comparing numerical values.

Algorithm 1 A function for comparing numerical values

```
1: procedure COMPARATOR(a, b)
2:     if a < b then return -1
3:     if a = b then return 0
4:     if a > b then return 1
```

1.1 Analysing algorithm complexity

Some algorithms can have good time complexity but are not practical for certain kinds of input data, e.g. insertion sort performs poorly on lage datasets but extemely well on smaller ones.

1.2 Sorting algorithm properties

1.2.1 Stability

Stability is the property whereby two equally valued elements in the input collection maintain their positions with respect to one another in the output (sorted) collection. Stability is unlikely to be an important consideration when sorting a collection of simple objects such as numbers but, when there is satellite data, it can become important [Cormen et al., 2001, p. 170].

1.2.2 Time efficiency

1.2.3 Memory efficiency

In-place sorting: Only a fixed amount of memory over the size of n (size of input) required, regardless of size of n. Non-in-place algorithms generally need an amount of memory that increases monotonically with n.

1.2.4 Suitability for a particular input

e.g. Size of input, degree of sorting, domain of input (e.g. integers from 1 to 1000), memory requirements, storage location (e.g. external?)

1.3 Comparison Sorts

Only uses comaprison operators to order elements. A comparison based sorting algorithm orders a collection by comparing pairs of elements until the collection is sorted.

No comparison sorting algorithm can perform better than $n \log n$ in the average or worst cases. Some non-comparison based sorting algorithms can, under certain circumstances, with better worst-case times than $n \log n$.

2 Sorting Algorithms

2.1 Insertion sort

2.2 Heapsort

2.3 Quicksort

2.4 Bucket sort

2.5 Timsort

3 Implementation & Benchmarking

References

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2nd edition.