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.

Where an implementation requires the use of nested loops, in most cases this will indicate $O(n^2)$ complexity.

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).

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

Good for small lists and ones which are almost sorted because the inner loop only needs to iterate until it finds the insertion point. Running time in the case of an already sorted list is $\Omega(n)$ because the inner loop will not have to run at all.

Often used in hybrid sorting algotrithms for its efficiency with small inputs.

2.2 Heapsort

A heap structure is a nearly complete binary tree (Cormen et al., 2001, p. 128). The indices of the parent, left child, and right chaild can be calculated as parent = i/2, left child = 2i, and right child = 2i+1, where i is the index of the element. These can quickly be calculated by shifting bits (Cormen et al., 2001, p. 128)

Uses max heap – at most a node is the value of its parent

2.3 Quicksort

Can perform badly if pivot consistently chosen which puts all or almost all elements in one or other of the sub arrays. For instance if 1st or last ais chosen and array is already nearly sorted. Median mgiht be better because best case is if the two sublists are roughly equal at each iteration.

Usual options: first, last, random, median

2.4 Counting sort

2.5 Introsort

Added insertion sort when partition size <= 20

3 Implementation & Benchmarking

Benchmarking: empirical method for comparing algorithm performance a postiori. Can be used to vaildate a priori / theoretical hypotheses

@max Use the min() rather than the average of the timings. That is a recommendation from me, from Tim Peters, and from Guido van Rossum. The fastest time represents the best an algorithm can perform when the caches are loaded and the system isn't busy with other tasks. All the timings are noisy – the fastest time is the least noisy. It is easy to show that the fastest timings are the most reproducible and therefore the most useful when timing two different implementations.

	5	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**	**55**	**6**
Insertion sort	0.007	0.018	0.039	0.043	0.076	0.112	0.127	0.177	0.236	0.311	0.367	0.36
Quicksort	0.017	0.032	0.051	0.067	0.078	0.095	0.114	0.132	0.143	0.170	0.175	0.19
Heapsort	0.019	0.037	0.059	0.088	0.109	0.145	0.173	0.213	0.242	0.262	0.301	0.34
Counting sort	0.027	0.034	0.041	0.046	0.049	0.046	0.053	0.059	0.058	0.057	0.070	0.06
Introsort	0.011	0.023	0.047	0.049	0.049	0.086	0.091	0.094	0.116	0.124	0.137	0.16

https://www.oreilly.com/library/view/python-cookbook/0596001673/ch17.html – Tim Peters on timing

n	**100**	**250**	**500**	**750**	**1000**	**1250**	**2500**	**3750**	**5000**	**6250**	**7500**	**8750**	**10000**
insertion sort	1.082	5.297	21.701	52.072	90.874	143.243	593.199	1318.3	2348.3	3640.56	5350.09	7300.5	9504.14
quicksort	0.28	0.874	1.937	3.609	4.716	6.322	17.591	32.529	53.89	76.678	104.856	133.81	175.071
heapsort	0.565	1.647	3.55	5.921	7.931	10.315	22.512	35.915	49.244	63.411	76.997	92.16	105.975
counting sort	0.091	0.174	0.335	0.477	0.663	0.805	1.642	2.311	2.971	3.712	4.384	5.001	5.772
introsort	0.295	0.797	1.476	2.577	3.029	3.487	10.094	24.093	37.677	46.581	56.788	69.493	80.566

Table 1: Times in milliseconds to sort arrays of size n for each of the algorithms

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (Second). MIT Press.