1 Introduction

Algorithms: input output, finiteness, unambiguous, correctness, feasibility, efficiency.

Time efficiency and space efficiency — how we compare and rate algorithms

A priori analysis – theoretical perspective — independent of implementation details – compare order of growth between algorithms – ***measure of complexity.***

A posteriori analysis – empirical evaluation of implementation of algorithm – ***tied to implementation an platform*** – performance compared — *** measure of performance ***

External factors affecting time of execution but not connected to complexity: Size of intpu, speed of computer, quality of compiler.

To sum up: Performance vs complexity Performance: how much memory, time, etc is used when algorithm is run — depends on external factor as well as code Complexity: How resource requirements scale as input gets larger

Complexity affects performance but performancedoes not affect complexity

Because algorithms are platform independent, empirical analyses cannot necessarily be generalised to all combinations of platform, etc.... so an independent measure of complexity is necessary to compare algorithm performance. This measure can be assigned to a number of orders of magnitude.

Complexity families: Constant, Logarithmic (log n), Sublinear, linear, n log(n), Polynomial, exponential

To evaluate complexity, most expensive computation should be identified, and the order of that computation will be the order of the complexity of the algorithm.

— Higher order term will dominate

In addition to size of input, specific characteristics of the input can affect the performance of an algorithm. For instance, given two sorting algorithms, one may outperform the other on an input with very few inversions, while the other may be more efficient given an input which is less sorted initially. Similarly, one algorithm may perform particularly well on smaller inputs while performing poorly on larger ones. Insertion sort, for example, examined in more detail below, performs extremely well on input sizes up to n=20 but performance falls off as n increases giving it by far the worst overall performance of the five algorithms examined here. The differential efficiency depending on input characteristics can be exploited by algorithm designers to make highly efficient adaptive hybrid algorithms which switch strategy to best tackle the particular nayure of the part of the input data on which they are currently working. Timsort, for instance, reduces the number of comparisons made by identifying and merging (using a merge sort) pre-existing runs in its input, and will add to those runs, where they are below a minimum threshold, using insertion sort which is very efficient on small inputs (Wikipedia contributors, 2020). The implementation of introsort used here, described below, uses a recursive partitioning algorithm to divide the sorting task into sublists and switches to heapsort when a maximum recursion depth is reached, as well as to insertion sort when a sublist contains fewer than 20 elements.

It follows from the fact that an algorithm will perform better or worse depending on the nature, as well as on the size of its input, that choice of algorithm does not rest solely on its complexity relative to otehr competing algorithms. It also depends on the nature of the input in terms of its size, degree of sortedness, and the underlying probability distribution of the specific inputs it is likely to encounter.

Worst, Best, and Average cases. These are classes of inputs, specific to a particular algorithm, for which that algorithm exhibits its least efficient, most efficient, and most usually efficient performances, respectively. In practice, best and worse cases are uncommon, but algorithms are often compared and chosen based on their worst case performances as this defines the lower bound on the algorithm's efficiency.

Generally speaking, the performance of an algorithm vis-*á*-vis the number of operations it must perform or the time it takes to complete its task, can vary significantly over all possible input combinations of size *n*. The worst cases are those instances where the algorithm performs the greatest number of operations or takes the longest time, over of all of those input instances. Worst case offers a guarantee that an algorithm will perform no worese - will take no longer - than it does in that case.

Therefore, choosing an agorithm

Complexity can be assessed based on use of whatever resources are scarce and need to be optimised for. Most commonly, algorithms are assessed based on time taken, or number of operations, per input size. Asymptotic Notation In algorithm analysis, Big O notation (e.g. $\mathrm{O(n^2)}$) describes the performance of an algorithm in the worst case scenario. This can be used to classify algorithms by complexity - two algorithms with the same Big O complexity will perfom similarly in their worst cases, and, of two algorithms with different Big O values, the one with the lesser value will perform much more efficiently in its worst case than the other will in its own worst case.

Tightest upper bound should be specified

Omega (Ω) notation represents the complexity of an algorithm in its best case, and Theta (Θ) notation represents its complexity in the average, or most usual case

In describing an algorithm in terms of efficiancy, it is necessary to isolate its description from its specific implementation. It is therefore analysed in terms of *n*, the number of elements in, or the size of, its input, and *f(n)* - the runtime, or the number of operations it takes to complete its task, given *n*.

Formal definition of sorting and the property of being sorted... Sorting: reorganising a list, A,such that if $\mathrm{A_i} < \mathrm{A_j}$ then $\mathrm{i} < \mathrm{j}$ must be true.enwiki:997404113

If there exists any pair of elements in a collection A, at positions i and j, such that $i < j$ but $\mathrm{A_i} > \mathrm{A_j}$ – 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 $\mathrm{A_i} = \mathrm{A_j}$ then there must be no k such that $\mathrm{i} < \mathrm{j} < \mathrm{k}$ and $\mathrm{A_i} \neq \mathrm{A_k}$

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

Insertion sort is a comparison sort algorithm which works in linear time in best case (Ω(n)), i.e. an already sorted input, and quadratic time in the average and worst cases ($\theta(\mathrm{n}^2)$ and $O(n^2)$). The algorithm sorts in-place so space complexity is constant.

The algorithm consists of two loops, one nested inside the other. The outer loop iterates from the second element in the input list to the end. If the index of the current outer loop value is i, the inner loop compares the value at i with the value to its left (i - *1*). If the element at i - *1* is less than the element at i, no action is necessary as the two elements are sorted with respect to one another. However, if the value at i - *1* is greater than the value at i then the two values are swapped and the inner loop continues leftward until either the start of the list is reached or a value smaller than the current iteration's tracked value (the value originally at index i) is found.

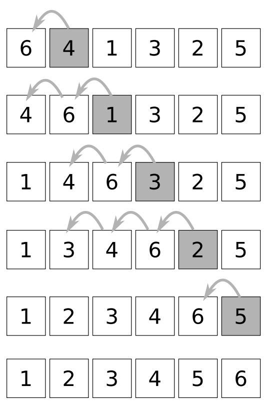

Figure 1: Insertion sort.

Figure 1 demonstrates the algorithm operating on an unsorted list of six digits. Each row represents an iteration of the outer loop and each grey arrow represents an iteration of the inner loop. Note that the elements to the left of the current target value, or *key*, are always sorted with respect to one another, and the elements to the right of the key, have yet to be inserted into this sorted sub-array. Also note that the inner loop only needs to run once for each inversion of which it is a part, and no more. This is why the algorithm has a running time of Ω(n) for an already sorted list – because a single run of the outer loop without any iterations at all of the inner loop will confirm that the input is sorted.

To confirm the worst case running time of $O(n^2)$ we can see that the algorithm will always have to iterate through the full list once, and for each iteration (i) it will have to execute the inner loop (*n-1*) × the number of inversions in which i participates. In a worst case –

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 Ω(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

Re insertion sort slower on value based as opposed to refernce based– python interns first 255 unsigned? — so ref based

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.

$n=$	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	**90**	**95**
Insertion sort	0.007	0.018	0.039	0.043	0.076	0.112	0.127	0.177	0.236	0.311	0.367	0.365	0.467	0.516	0.617	0.697	0.797	0.804	0.958
Quicksort	0.017	0.032	0.051	0.067	0.078	0.095	0.114	0.132	0.143	0.170	0.175	0.198	0.204	0.223	0.254	0.270	0.285	0.291	0.339
Heapsort	0.019	0.037	0.059	0.088	0.109	0.145	0.173	0.213	0.242	0.262	0.301	0.342	0.382	0.406	0.456	0.466	0.526	0.531	0.563
Counting sort	0.027	0.034	0.041	0.046	0.049	0.046	0.053	0.059	0.058	0.057	0.070	0.066	0.075	0.076	0.084	0.083	0.083	0.099	0.095
Introsort	0.011	0.023	0.047	0.049	0.049	0.086	0.091	0.094	0.116	0.124	0.137	0.161	0.167	0.180	0.196	0.227	0.210	0.221	0.292

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

https://www.oreilly.com/library/view/python-cookbook/0596001673/ch17.html – Tim Peters on timing Peters (2002) "The slowest result is often computed on the first try, because your machine's caches take extra time to adjust to the new task."

References

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

Peters, T. (2002). *Introduction to algorithms chapter. Python cookbook*. O Reilly.

Wikipedia contributors. (2020). Timsort — Wikipedia, the free encyclopedia [[Online; accessed 9-May-2021]]. https://en.wikipedia.org/w/index.php?title=Timsort&oldid=997404113