Impact of Algorithmic Complexity on Performance:

- Algorithmic complexity is a measure of how much computation a program perform when using a particular algorithm. It is a measure of its efficiency and estimate of operation count. It is not a measure of the complexity of the code necessary to implement a particular algorithm. An algorithm with low algorithmic complexity is likely to be more difficult to implement than an algorithm with higher algorithmic complexity. The most important fact is that the algorithmic complexity is not a model of the execution time but a model of the way execution time changes as the size of the input changes.
- Algorithmic complexity represents the expected performance of a section of code as the number of elements being processed increases. In the limit, the code with the greatest algorithmic complexity will dominate the runtime of the application.
- Assume that your application has two regions of code, one that is O(N) and another that is $O(N^2)$. If you run a test workload of 100 elements, you may find that the O(N) code takes longer to execute, because there may be more instructions associated with the computation on each element. However, if you were to run a workload of 10,000 elements, then the more complex routine would start to show up as important, assuming itdid not completely dominate the runtime of the application.
- Different parts of the application will scale differently as the workload size changes, and regions that appear to take no time can suddenly become dominant.
- Another important point to realize is that a change of algorithm is one of the few things that can make an order of magnitude difference to performance. If 80% of the application's runtime was spent sorting a 1,000-element array, then switching from a bubble sort to a quicksort could make a 300×difference to the performance of that function, making the time spent sorting 300× smaller than it previously was. The 80% of the runtime spent sorting would largely disappear, and the application would end up running about five times faster.
- Table 1 shows the completion time of a task with different algorithmic complexities as the number of elements grows. It is assumed that the time to complete a single unit of work is 100ns. As the table illustrates, it takes remarkably few elements for an $O(N^2)$ algorithm to start consuming significant amounts of time.

Elements	O(1)	O(N)	O(N $\log_2$ N)	O(N^2)
1	100ns	100ns	100ns	100ns
10	100ns	1,000ns	3,322ns	10,000ns
100	100ns	10,000ns	66,439ns	1,000,000ns
1,000	100ns	100,000ns	996,578ns	100,000,000ns
10,000	100ns	1,000,000ns	13,287,712ns	10,000,000,000ns

Table 1: Execution Duration at Different Algorithm Complexities

- The same information can be presented as a chart of runtimes versus the number of elements. Figure below makes the same point rather more dramatically. It quickly becomes apparent that the runtime for an O(N2) algorithm will be far greater than one that is linear or logarithmic with respect to the number of elements.

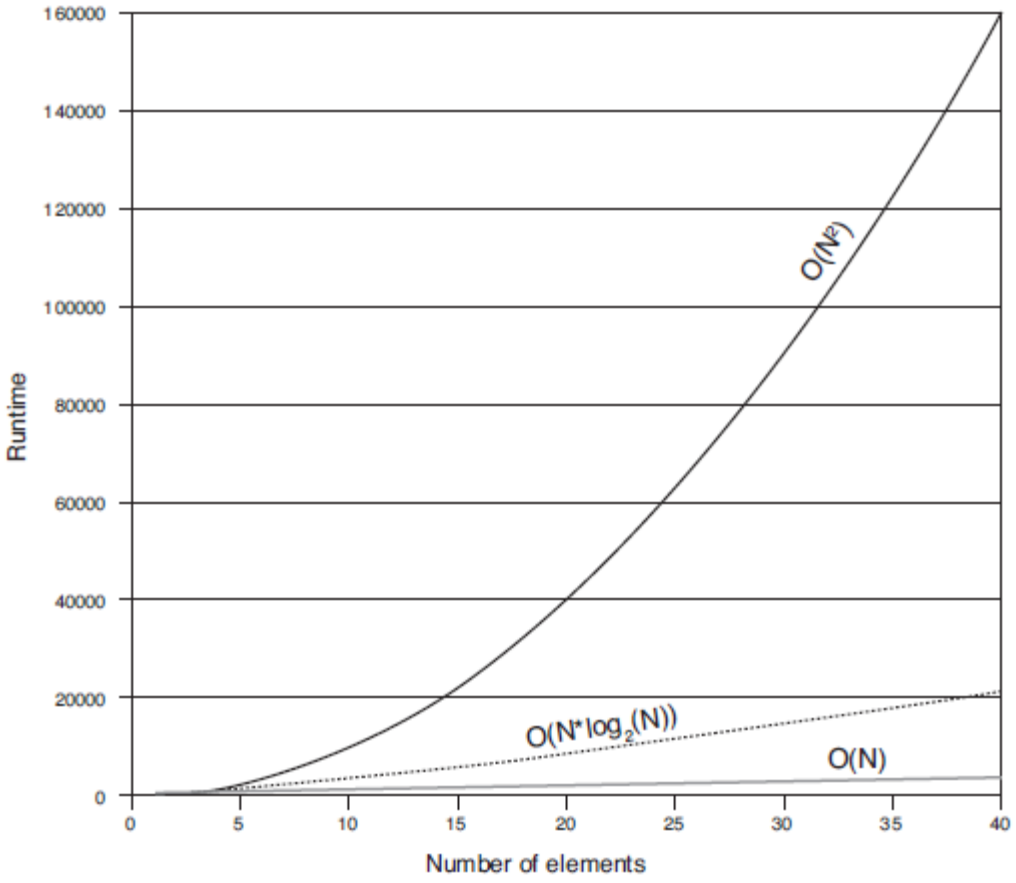

Fig: Different orders of algorithmic complexity

Using Algorithmic Complexity with Care:

- Although algorithmic complexity is a very good guide to where time could be spent, several issues need to be considered. It may be tempting to select the most efficient algorithms for every aspect of the code. Compare the lines of code necessary to implement the quicksort with those required for the bubble sort as well as how easy it is to read those lines of code and understand how the algorithm works. Algorithms with lower algorithmic complexity areusually more difficult to implement and more difficult to understand. Both of these factors will lead to more developer time needed for the implementation, and the code may potentially need a more experienced developer to maintain it. The point is that using more complex algorithms can have an impact on developer time and cost. A simpler algorithm might be easier to implement and result in lower development costs. It may also be possible that the code does not need a more complex algorithm for typical workloads.
- A second point to consider is that algorithmic complexity is concerned with the operation count. It does not consider the cost of those instructions. It may weigh the cost of an add operation the same as a multiply yet be possible to have algorithms that perform the same task with very different numbers of add and multiply operations. At another level, the algorithms don’t consider implementation details such as caches. One algorithm might be very cache friendly, whereas another could incur many cache misses.
- Therefore, it is necessary to both look at the algorithmic complexity and evaluate the actual implementation of the algorithm to determine whether the implementation is likely to achieve good performance.