Software Algorithm Testing – Examples

Software algorithms used for processing large amounts of data need to be efficient, incorporating industry best practices. This is especially important for fast searching, sorting, and merging of data files. The primary goal in looking at software algorithm performance during developmental testing is to ensure that industry best practices have been employed to ensure operational mission threads involving large data sets operate efficiently. Algorithm testing can also be used to compare the performance of proposed COTS solutions to aid in choosing the one that provides the best mix of capability and processing efficiencies. Significant insights can be learned from focused testing in a DT controlled environment, even though the tester may not have direct access to the data structures or software code. These insights should be sufficient to determine whether the developer has used industry best practices when writing data structures and algorithms. Because this type of testing does not focus on trying to manifest and detect software bugs, the process will be explained in more detail in this example. However, TEMP language can be very simple to insert.

Example TEMP language

Example 1 (generic): Algorithm performance testing will be executed during DT for those parts of mission thread execution involving the manipulation of large data sets supporting a major theater war level of scenario, where the response time may be excessive to the point of potential mission impact.

Example 2 (AOC-WS): Algorithm performance testing will be performed during DT for the Target List Merge Process that is used to create the Joint Integrated Prioritized Target List (JIPTL).

Example 3 (AOC-WS): Algorithm performance testing will be performed during DT for the auto-planning process used to determine aircraft routes to deliver weapons to multiple targets.

Types of algorithms that may need performance testing

There are several types of algorithms that may need performance testing to try to ascertain whether the developer used industry best practices. Each of these categories of work needing to be performed can be categorized based on roughly how much longer the processing should take as the data set increases in size.

- Searching one or more large data sets to find data elements matching certain criteria, to include creation and execution of complex ad hoc data queries
- Sorting a large data set into a particular sorted order
- Merging two or more data sets, at least one of which is large, with resultant list possibly in some sorted order

- Optimization algorithms which seek to determine optimal routing of a delivery vehicle to visit multiple locations (for example, a optimizing a bomber route as it flies over or near multiple targets)

Industry best practices

The subject of combinatorial algorithms deals with the problems associated with performing fast computations on discrete data structures. More information on this can be obtained through university-level course work on data structures and combinatorial algorithms. Many types of algorithms can also be found through simple internet searches, and Wikipedia will show the name of the algorithm and best case, average case, worst case, memory usage, and whether the algorithm is stable. http://en.wikipedia.org/wiki/Sorting_algorithm shows information for various sorting algorithms. Unless significant information is known about the data sets, industry best practices should generally use algorithms based on good average performance.

In this example, big O notation is used to classify algorithms by how they respond (e.g., in their processing time or working space requirements) to changes in input size (e.g. the number of data elements in a large data file). Big O notation characterizes functions such as the processing time according to their growth rates, usually providing an upper bound on the growth rate of the function. See http://en.wikipedia.org/wiki/Big_O_notation.

Performance is usually expressed in terms of the size of the data set. For example, if $\boldsymbol{n}$ represents the number of elements in a large data set, then the average performance of an algorithm operating on the elements of the data set would be expressed as being $O(\boldsymbol{n} \ln \boldsymbol{n})$ or $O(\boldsymbol{n}^2)$.

Algorithm performance testing – sorting example

When examining software data structures and algorithms in a black box environment, the goal is merely to determine whether the data structures and algorithms likely used in the software application belongs to a class exhibiting $O(\boldsymbol{n} \ln \boldsymbol{n})$ or $O(\boldsymbol{n}^2)$ average type behavior, for example.

Suppose the software function being tested is an algorithm that sorts a large data file into a particular sorted order. Good average performance for a sorting algorithm is $O(\boldsymbol{n} \ln \boldsymbol{n})$, whereas bad average performance would be $O(\boldsymbol{n}^2)$. Several industry standard sorting algorithms that exhibit “good” performance are Quick sort, Heap sort, and Merge sort. Sorting algorithms that do not exhibit good average performance would include Bubble sort, Insertion sort, and Selection sort, all of which exhibit $O(\boldsymbol{n}^2)$ average performance. Figure 1 illustrates the rate of growth, in time, based on $O(\boldsymbol{n} \ln \boldsymbol{n})$ or $O(\boldsymbol{n}^2)$ type performance, as the size of the data file increases.

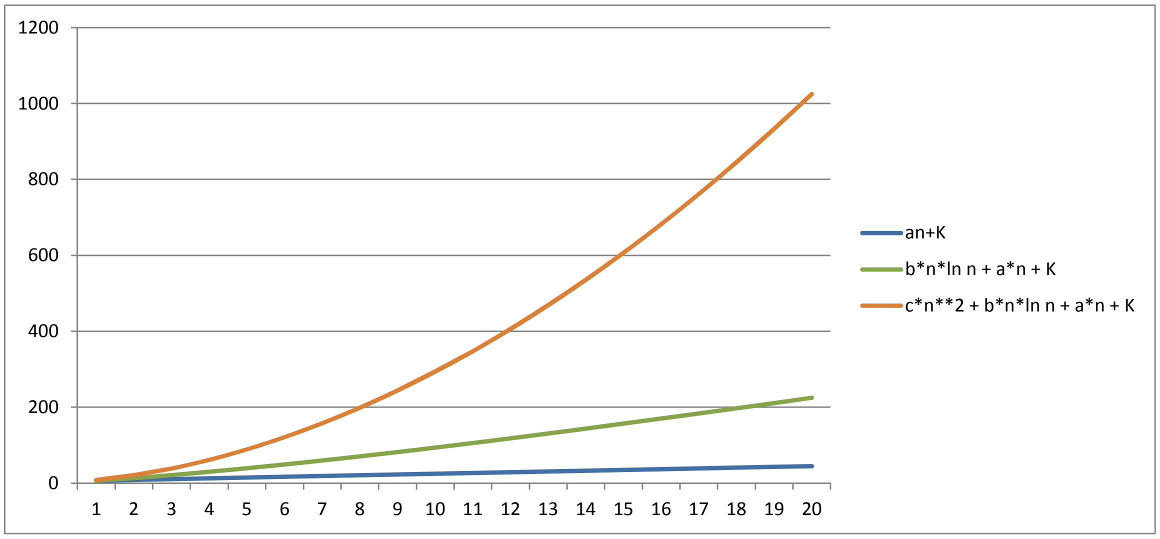

Figure 1. Performance Growth

In Figure 1, the bottom curve, expressed as “a*n + K”, represents linear, or O(***n***) growth in time. The middle curve, expressed as “b*n ln n + a*n + K” represents O(***n*** ln ***n***) growth in time. The upper curve, expressed as “c*n^2 + b*n ln n + a*n + K” represents O($\boldsymbol{n}^2$) growth. This figure only serves to illustrate graphically how much faster an O($\boldsymbol{n}^2$) curve rises, compared to an O(***n*** ln ***n***) curve. Thus, a sorting algorithm that exhibits O($\boldsymbol{n}^2$) growth would become excessively slow as the file size increases.

Suppose during early DT testing, a process that sorts a list seems to take a relatively long time, to the point where the ability to accomplish a mission thread in a timely manner may be questionable. During early testing, the goal is not to determine whether the developer may have used Quick sort or Heap sort. Rather, it is simply to determine if the performance appears to be O(***n*** ln ***n***) or O($\boldsymbol{n}^2$), and whether more extensive testing or structured code walkthroughs may be required. For the example problem of sorting a large data set, four or five test runs sorting significantly different sized data sets could be completed with the response time plotted on the Y axis, against the data set size on the X axis. If the plotted data appears to be O(***n*** ln ***n***), additional testing of this kind may not be warranted. If the data suggests that the response time rises as O($\boldsymbol{n}^2$), then more thorough performance testing would be warranted, as well as a review of coding methods by individuals specifically trained in analysis of algorithms.

To perform more thorough performance testing, it is recommended that it be performed in a carefully controlled DT environment where “noise” contributions, such as resource contention from other sources, can be eliminated. One could start by establishing a data set of size ***n*** in a pseudo-random sorted order. This list could then be sorted, with the response time being recorded. This same list could be placed into a different pseudo-random sorted order, and then sorted again, with the time recorded. This process would be repeated several times, allowing one to compute the mean response time for sorting the lists of size ***n***. Even though the distribution of response times is not likely to be normally distributed, we use mean (instead of median) response time since we are checking for the average performance characteristics of the algorithm used. Then, a similar process is followed for obtaining mean response times for sorting lists of size 2***n***, 3***n***, and 4***n***. The four mean response times for sorting lists of size ***n***, 2***n***, 3***n***, and 4***n*** would be used to solve a system of 4 equations, 4 unknowns for an equation of form

Response Curve = $cX^2 + bX \ln X + aX + K$

For the simple sorting of one large dataset example, the worst case behavior expected would be $O(n^2)$, so only coefficients "c", "b", "a", and "K" would be needed. If the estimated value for "c" is very near zero, then the algorithm is likely to be "good", since it would be exhibiting O(***n*** ln ***n***) performance. Conversely, if the estimated value for "c" is significantly greater than zero, then the sorting algorithm probably exhibits $O(\boldsymbol{n}^2)$ average behavior and is not "good" by industry best practices. "Bad" algorithms should be replaced with "good" algorithms early in development so that when the algorithms are used on large data sets during OT, significant performance degradation is not a problem.

Note we can also plot the worst case response times for sorting lists of size 2***n***, 3***n***, and 4***n***, and this may give additional insight into the worst case performance of the sorting algorithm used.

Algorithm performance testing – file merging example

Suppose the task is to merge two data files containing ***m*** and ***n*** records, respectively, and that duplicates must be removed. Suppose also that each list is sorted in some priority order, but not on a unique key field for each record. The merged list needs to be sorted in this same priority order.

For this type of problem, a "bad" algorithm would perform, on average, in O(***mn***) or possibly in O((***m***)(***m***+***n***)). It might not change the sorted order of either list, and instead would consider each element from the second list and walk through all elements of the first list to ensure no duplicates, and it would be inserted into the first list. This process would continue until all **n** records from the second list had been correctly processed into the first list, eliminating duplicates.

A "good" algorithm would perform in O((***m***+***n***) ln (***m***+***n***)), which is significantly better as ***m*** and ***n*** become large. Each list could be sorted in order by unique key, assuming the use of a good sorting algorithm. This would require, on average, O(***m*** ln ***m***) to sort the first list, and O(***n*** ln ***n***) to sort the second list. Then, it becomes very easy to make one linear pass through each list, inserting and removing duplicates, and this requires O(***m***+***n***) time. Finally, the merged list could be resorted into the desired priority order, requiring on average O((***m***+***n***) ln (***m***+***n***)), again assuming the use of a good sorting algorithm.

Similar to the approach for sorting a large data set, four or five test runs merging significantly different sized data sets could be completed with the response time plotted on the Y axis, against the data set sizes (***m***+***n***) on the X axis. If the plotted data appears to be O((***m***+***n***) ln (***m***+***n)***), additional testing of this kind may not be warranted. If the data suggests that the response time rises as O(***mn***) or worse, then more thorough performance testing, similar to that discussed for the sorting algorithm, would be warranted.

If the problem were merging three large data sets of size ***m***, ***n***, and ***p***, testing would consider O((***m***+***n+p***) ln (***m***+***n+p)***) type average behavior as good, with average performance worse than that being bad.

Algorithm performance in requirements or for product selection

For COTS solutions for which code development is desired to be minimized, algorithm performance can be used as one further attribute for product selection. Performance considerations could include algorithm characteristics such as best case, average case, worst case, memory usage, and stability.