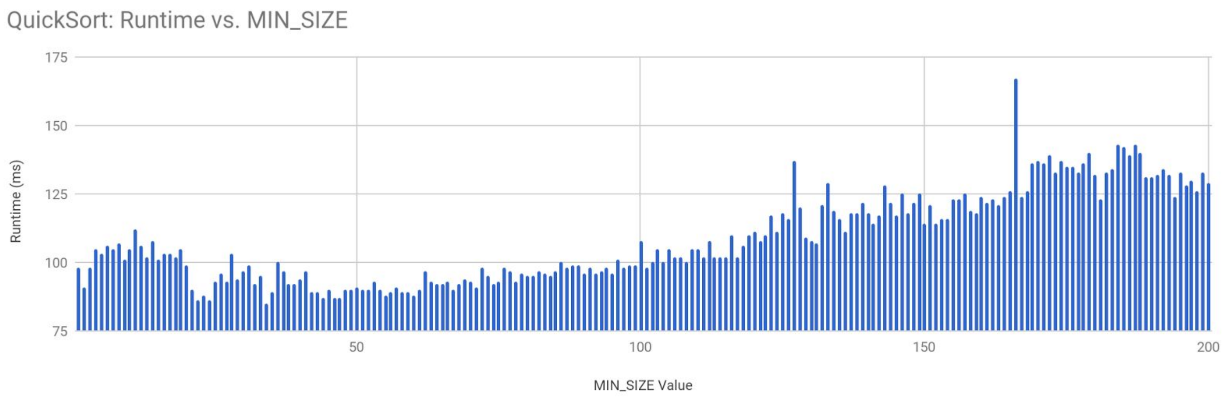
Cole Schiffer CS 201.01

Grant Lee

HW 09: Investigation of QuickSort & ShellSort

Investigation 1.1 - QuickSort: Relationship between runtime and MIN_SIZE

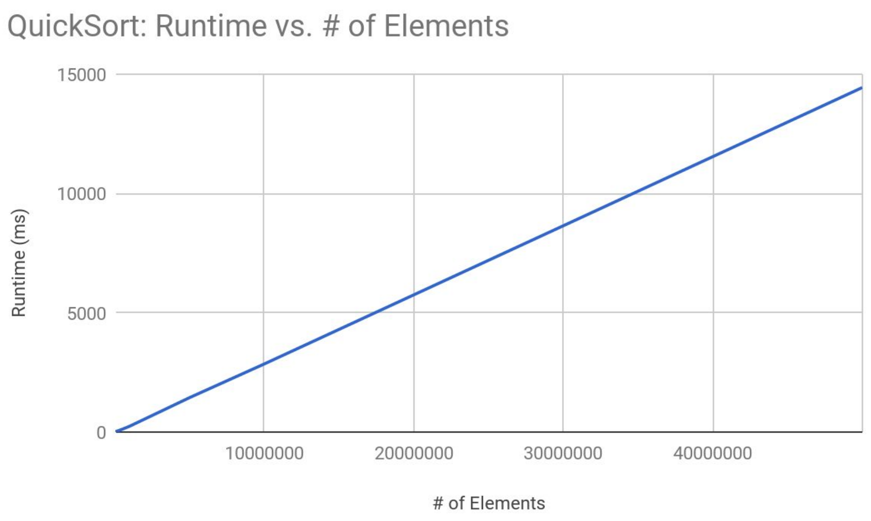


Our test consisted of testing 200 different MIN_SIZE values (1-200) to sort an array of one-million elements. The test consisted of an array that contained all the desired MIN_SIZE values to be tested for the QuickSort method. The mentioned array would be sifted through for the values to be tested for a corresponding runtime. The calculated runtime did not include the runtime of the sifting through of the array of test values. Additionally, to ensure the integrity of the test, each array was shuffled before each sort.

Upon observation of the inconsistencies of the data, the operations were completed multiple times. Upon completion of multiple trials, the runtimes were averaged out and presented in the graph above. In the graph that compares the runtime and the MIN_SIZE, it can be observed that initially there is an increase in runtime, then a decrease in runtime till the MIN_SIZE value of about 24. After the MIN_SIZE of 24, the trend shows an ever increasing runtime. Even though we tried to reduce error, there are still some outliers which can be attributed to noisy data and inconsistencies in the computer's compiler.

The decision to manipulate the MIN_SIZE variable was based on the information that the MIN_SIZE variable affected the size required for the sorting method to switch over to insertion sort instead of quicksort. With this knowledge in mind and the observed results, we can conclude the following: as MIN_SIZE increases, it will continue to decrease sorting performance. This is because the average case of insertion sort is about O(n^2). The larger the MIN_SIZE value, the larger amount of the sorting of the array will be sorted using insertion sort instead of Quicksort which has an average case of O(nlog(n)). Additionally, the initial increase in runtime at small values of MIN_SIZE is observed, because Quicksort is used for a larger part of the sorting of the array which would again be O(nlog(n)); however, in this case, most of the array will already be somewhat sorted which would actually make insertion sort faster since it would become practically O(n).

Investigation: 1.2 - QuickSort: Relationship between runtime Taken and Number of Elements in Array

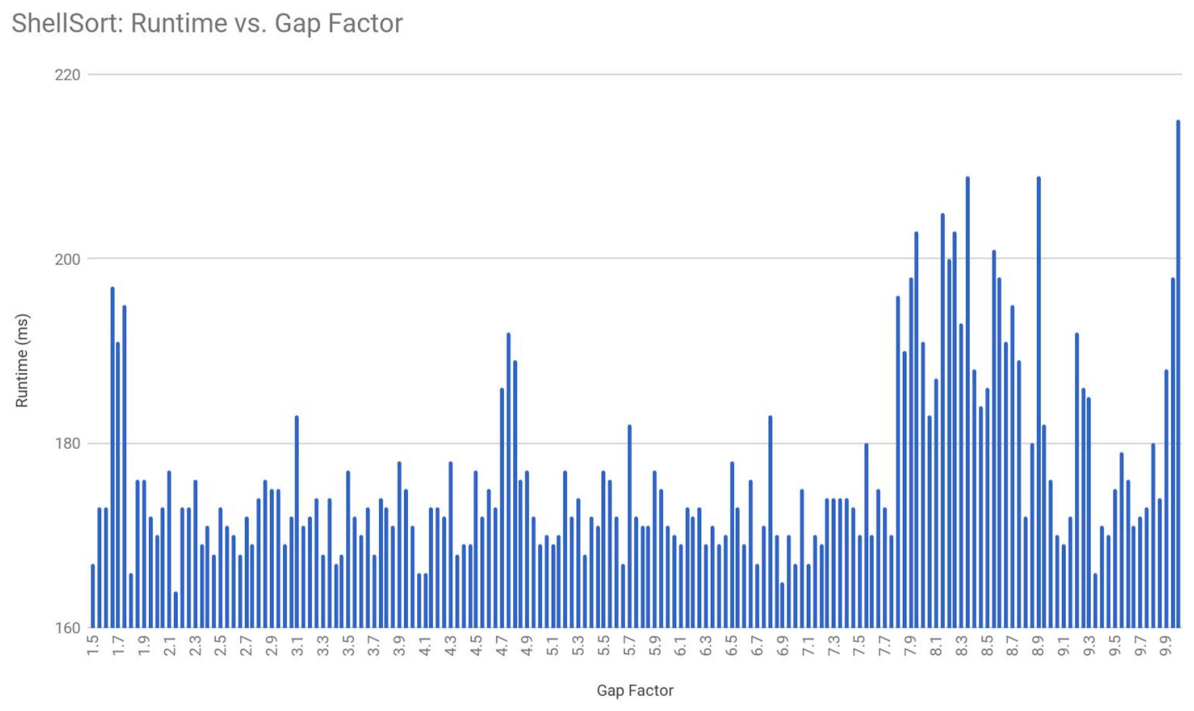


In order to observe the relationship between the size of the array being sorted and the time taken to sort the array, another experiment was conducted. The array sizes of 10,000; 50,000; 100,000; 500,000; 1,000,000; 5,000,000; 10,000,000; 50,000,000 were tested. After observing the unstableness of the data as mentioned before, multiple trials were taken and averaged out.

The data resulting from the experiments are presented in the graph above. A linear trend is observed. Because quicksort has an efficiency of O(nlog(n)), it can be inferred that upon increasing the number of items needing to be sorted, the time taken to sort will increase at a nearly linear rate. Additionally, with this data we concluded that testing with an array of one million elements would be ideal for efficiency and usability for other tests due to noticeable runtime.

Cole Schiffer CS 201.01
Grant Lee

Investigation 2.0 - Shellsort: Runtime and Gap Distance



The experiment conducted monitors the impact of the gap factor on the runtime taken to complete the sorting of an array of one-million elements. Since the ideal gap factor is known to be 2.25, small increments of .05 from 1.5 to 10.0 seemed to be a reasonable range to observe the changes in runtime. The test stored the range of values in an array which would then be sifted through to be tested for corresponding runtimes. The calculated runtime did not include the runtime of the sifting through of the array of test values. Additionally, to ensure the integrity of the test, each array was shuffled before each sort. Each gap factor was tested three times then averaged all with unique arrays of data.

Because the gap factor influences the rate at which the gap size changes through the process of the ShellSort, it makes sense to observe whether a change in the gap factor will have a significant impact on the runtime. Based on the logic behind the gap factor, the larger the gap factor, the quicker the gap size will decrease. With a quicker decrease in the gap size, more time will be spent partitioning the array; however, there may be more elements in the incorrect position when performing the insertion sort when the gap becomes two or less causing for a longer runtime. On the other hand, if the gap factor is smaller, the sorting method will spend more time partitioning the array. The resulting pre-sorted array will probably be very close providing a much quicker run time for the insertion sort. Based on these two possible scenario, it would probably be ideal for the best mixture of the two. Shellsort contrasts with Quicksort in that the Big-O notation is completely dependent on the gap factor. Observing the results from the test in the graph above, it can be concluded that there is not a clear trend to determine which scenario is better.