**ANALYSIS OF QUICKSORT ALGORITHM**

In modern computer science, improvements in computers (processor speeds in particular) are only followed by an increase in the rigor of problems scientists want to solve with computers. Thus, although building better and faster computers is important, finding ways to minimize the time it takes to execute an algorithm is equal crucial. In this project, we will be analyzing the quicksort algorithm and finding ways to ensure the algorithm spends the least time possible for all input types.

(See java files for algorithm)

**PROJECT SETUP:**

Since the quicksort algorithm is fundamentally a partition element that is called recursively many times, improving the runtime of the partition algorithm will significantly improve the runtime of the entire algorithm and since the partition algorithm is fundamentally a comparison algorithm that compares a given set of elements with another element, reducing the number of comparisons at each point is a reliable way to reduce the runtime of the algorithm. Thus, we are going to use the number of comparisons (counts) made throughout the run of a quicksort call on a given array as a proxy for the runtime of the algorithm.

In this analysis, we will be comparing the runtime of quicksort for a randomized partition algorithm with the runtime of quicksort for a deterministic partition algorithm. A randomized partition algorithm is one in which the pivot used for partition is randomly selected.

To help us understand how these algorithms fare on all array types, for each algorithm, we will analyze its runtime on an array with random inputs, an array with inputs that are partly sorted, and on an array with mostly sorted input all of sizes ranging from 0 to 10,000. Generating an array with random inputs (inputs that have no particular order to them) is what is referred to as randomization over the input.

Because each array of elements is different and the time a quicksort-call on that array takes depends on the elements in the array, we are going to repeat our experiments with the same constraints several times – 100 in our case – and compute the average so as to get an analysis that works fairly well regardless of the instance of the array we are running quicksort on.

So, in this experiment, we have two main algorithms: The randomized quicksort and the deterministic quicksort algorithm. For each of these algorithms, we are going to analyze the runtime on arrays of different sizes that contain randomized inputs, partly sorted inputs, and mostly sorted inputs.

DETERMINISTIC QUICKSORT ANALYSIS

1. Average Analysis:

Chart, line chart

Description automatically generated

In the graph above, we plotted the average (expected) runtime of deterministic quicksort for all array types of sizes between 0 and 10,000 and compared that to the bound 2nln(n). We realize that as n increase, the runtime for everything but arrays with randomized inputs exceeds the bound with the mostly sorted arrays having the highest runtimes, followed by the partially sorted arrays. The arrays with random inputs, however, do not exceed the bound.

This analysis is prone to errors as the runtime for the randomized inputs may exceed the bound as n gets large but we are unable to confirm that as our computers lack the capacity to test for larger values of n.

1. Variance Analysis:

Chart, histogram

Description automatically generated

Variance For RandomInput: 19024.250166666665

Standardized Variance RandomInput: 1.3818429436418233E-4

Chart, histogram

Description automatically generated

Variance For PartiallySortedInput: 1601033.6035185757

Standardized Variance PartiallySortedInput: 0.001883046079176806

Chart, histogram

Description automatically generated

Variance For MostlySortedInput: 775171.3875696979

Standardized Variance MostlySortedInput: 1.958794919419153E-5

Chart, histogram

Description automatically generated

Variance For RandomInput: 153263.17961236363

Standardized Variance RandomInput: 1.1671890693229522E-4

Chart, histogram

Description automatically generated

Variance For PartiallySortedInput: 3.8823394296559565E7

Standardized Variance PartiallySortedInput: 0.0016911208376719491

Chart, histogram

Description automatically generated

Variance For MostlySortedInput: 1.5212307888690652E7

Standardized Variance MostlySortedInput: 9.857505693631922E-6

Chart, histogram

Description automatically generated

Variance For RandomInput: 1774294.794088931

Standardized Variance RandomInput: 9.536762711280286E-5

Chart, histogram

Description automatically generated

Variance For PartiallySortedInput: 8.565100967696832E8

Standardized Variance PartiallySortedInput: 5.449678040723683E-4

Chart, histogram

Description automatically generated

Variance for MostlySortedInput: 2.826250841589367E8

Standardized Variance MostlySortedInput: 2.2629545495460694E-6

From testing out several array sizes and choosing different n values for variance computations, and looking at the various graphs indicated above, the general pattern is that arrays with randomized inputs have the least variance followed by arrays with mostly sorted inputs then arrays with partially sorted inputs.

Upon calculating the standardized variance however, we realize that arrays with mostly sorted inputs have the least standardized variance followed by arrays with randomized inputs then arrays with partially sorted inputs.

Because the mean runtime of arrays with mostly sorted inputs is significantly higher than that of arrays with randomized inputs, we get the impression that the variance between inputs of arrays with mostly sorted inputs is higher than the variance of arrays with randomized inputs, but the standardized variance proves this false.

From these, we can then conclude that for arrays with randomized inputs, the runtimes of each array instance are all close to the mean (not as close to the mean as arrays with mostly sorted inputs are though) which is very small. For arrays with mostly sorted inputs however, you will get values being the closest to the mean but because the mean is relatively higher than that of arrays with randomized inputs, you will find that your runtime for most arrays of that type will be consistently higher than the runtimes for arrays with randomized inputs. Lastly, for arrays with partially sorted inputs, we get the highest fluctuations in runtime with a high mean value which does not really help us.

In conclusion, if you run the deterministic algorithm on an array with randomized inputs, there is good chance you will get a very lower value than you would if you ran it over an array with partially or mostly sorted inputs.

RANDOMIZED QUICKSORT ANALYSIS

1. Average:

Chart, line chart

Description automatically generated

In the graph above (facetted for visibility purposes), we plotted the runtime of our random quicksort algorithm against array size (between 0 and 10,000) for all array types (random, partially sorted, and mostly sorted) and compared that to the bound 2nln(n) for the same range of n values.

From the graph above, we notice first of all that the averages for all n-values of all array types are lower in this random algorithm than they are in their deterministic counterparts.

Secondly, we also notice that these runtimes are less than the bound and the difference gets pronounced as n increases.

Aside these differences however, the runtimes between array types in the randomized algorithms are mostly similar and follow the same trend. Thus, the randomized algorithm reduces the runtime significantly and also makes sure the runtime isn’t so dependent on the input at hand.

1. VARIANCE

Chart, histogram

Description automatically generated

Variance For RandomInput: 2028.0276117575675

Standardized Variance RandomInput: 2.848879141640547E-5

Chart, histogram

Description automatically generated

Variance For PartiallySortedInput: 3208.055360191925

Standardized Variance PartiallySortedInput: 4.5121652358982545E-5

Chart, histogram

Description automatically generated

Variance For MostlySortedInput: 2491.869225646468

Standardized Variance MostlySortedInput: 3.5127773368653166E-5

Chart, histogram

Description automatically generated

Variance For RandomInput: 16271.91086039396

Standardized Variance RandomInput: 2.6613714632070087E-5

Chart, histogram

Description automatically generated

Variance For PartiallySortedInput: 16306.131392363628

Standardized Variance PartiallySortedInput: 2.6632220723084844E-5

Chart, histogram

Description automatically generated

Variance For MostlySortedInput: 18826.49496408082

Standardized Variance MostlySortedInput: 3.076803507537659E-5

Chart, histogram

Description automatically generated

Variance For RandomInput: 132506.61425950547

Standardized Variance RandomInput: 1.735852356824559E-5

Chart, histogram

Description automatically generated

Variance For PartiallySortedInput: 179552.67310703048

Standardized Variance PartiallySortedInput: 2.3566113338483538E-5

Chart, histogram

Description automatically generated

Variance For MostlySortedInput: 123085.8578372827

Standardized Variance MostlySortedInput: 1.6120371027453093E-5

Just like the average, the variances are similar between all array types and so is the case too for the standard variance which are all lower than our values for the deterministic algorithm. However, the variance relationship between input types for a given array type isn’t cut out or clear and varies depending on which n-value you choose.

IMPROVEMENTS TO ALGORITHM

1. DETERMINISTIC:

From the analysis we made about the average of our deterministic algorithm, we realize that arrays with randomized inputs had significantly lower averages and fairly lower variances. So, we thought if we were able to randomize the values inside an array before running our algorithm, we would be able to save sometime. Thus, we ran the Fisher Yates algorithm on each input before calling deterministic quicksort and our analysis turned out right. This is illustrated in the graph below

[Graph for quicksort deterministic runtimes after shuffles]

Chart, histogram

Description automatically generated

Shown above is the graph of the Deterministic Algorithm for the randomly sorted inputs of array size 800 and shown below is the graph for the improved deterministic algorithm after implementing the Fisher Yates shuffle. The graph does not show any visible improvement because the inputs are already randomized and using the Fisher Yates algorithm to further randomize them does not result in any observable improvement. Chart, histogram

Description automatically generated

However, there is an observable change in the runtimes for the partially sorted inputs.

Chart, histogram

Description automatically generated

The histogram for the Deterministic algorithm shown above has a minimum runtime of 26000 and a maximum runtime of 32000 whereas after the Fisher Yates shuffle the runtime drops to a minimum of 11600 and a maximum of 12300. The Fisher Yates shuffle randomizes the elements in the array so that at each partition, the pivot selected is not so high a value as it was originally when it was partially sorted. This makes less comparisons occur.

Chart, histogram

Description automatically generated

The is an even greater improvement for the mostly sorted inputs. The histogram of the deterministic algorithm for mostly sorted inputs has a minimum runtime of 197000 and a maximum runtime of 20100.

Chart, histogram

Description automatically generated

The minimum runtime after the implementation of the Fisher Yates shuffle is 11500 and the maximum runtime is 12100. The drastic improvement in runtime is because the Fisher Yates shuffle randomizes the input so that the partition element is not the largest element in the array and is closer to the median.

Chart, histogram

Description automatically generated

LIMITATIONS:

We could not come up with an improvement for the random algorithm although theoretically it makes sense to us that if we found a way to make the partition element select the median in the specified range as the pivot each time, we could reduce significantly the number of counts. It is hard to do this because a fundamental part of the random algorithm is choosing pivot randomly and there is no guarantee as to where this will fall.

CONCLUSION:

To conclude everything, it is clear that quicksort has a much less rutime when the pivot element is randomly selected and/or when the element in the array is shuffled to prevent a deterministic algorithm from constantly selecting values that are too high which would mean there will be more comparisons, wasting too much time. This speaks to the importance of randomization as a tool for improving the efficiency of algorithms.