**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. 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 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 with 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 quicksort for all array types for our deterministic algorithm and compared it 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 larger but we are unable to confirm that as our computers lack the capacity to test for larger values of n.

1. Variance Analysis:

[GRAPHS FROM LOVEMORE]

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.

From this analysis, we realize that if we shuffle every array before running our deterministic algorithm, we get a way better runtime than we would otherwise. [See graph below for illustration]

RANDOMIZED QUICKSORT ANALYSIS

1. Average:

Chart, line chart

Description automatically generated