SortDoublingRatios

Natan Zamanzadeh
Intro To Algorithms —— Professor Leff

WHITE PAGES

Using a MacBook Pro 2018 i5 quad core.

—

Sedgwick's Doubling Ratio test's parameters on how to predict the next running time of your algorithms: observe that your algorithm approaches a value of 2^b, and multiply the previous run time by your observed 2^b value.

—TEST CODE WRITEUP—

In my programmed tester, each array of length N was tested 10 times before being doubled.

In order to test the array multiple times and have usable data, the arrays needed to be the same. So each test, the array would be copied over and tested again. This would normally increase the run-time, but the Stopwatch was not started until right before the Sort occurred.

The first 5 tests were considered a 'warm up' period, and the last 5 were counted towards an average run-time, while also saving the final run-time as a separate value.

After concluding all 10 tests and storing their run times values, the times would be printed, and array would then be doubled and the tests would start over with that doubled array. This process would include a prediction using Sedgwick's model for predicting future run-times.

Each run would consist of 5 tries and 10 tests per double, 50 tests in total.

—END OF TEST-CODE WRITEUP —

- **BUBBLE SORT**

Using Bubblesort, which has an average and worst-case run-time complexity of n^2, the run time of an algorithm that has had its values doubled should be increased by a factor of 4 from the previous run.

I was expecting to achieve such values after each run, but on early numbers (<16,000) I would receive ratio values exceed 3.0-6.0.

As I increased the starting doubling size to a size of 8,000 - 16,000, a more stable, but still not as predicted, ratio became to come about.

These ratios would approach numbers between 4.2-4.6, although sometimes showing values of below 4.0 and sometimes greater than 5.0.

I attribute these unexpected changes in the ratios to an exceeding amount constant values, as an 'extreme-worst-case' scenario; This is in regards to the previous sort, where I assume there were many more operations that took place in comparison to the previous run, causing the ratio to become larger. (OR vice versa for when reaching ratio values in the 3.0 range).

—

**I do want to state, that although my ratios seemed to exceed 4.0, had I not converted my values to a double when doing fractions, I would have achieved an exact ratio of 4 almost every time.

—

While the values did not approach my predicted ratios of exactly 4.0, I was shocked that sometimes the ratios would approach values of 5.0+. What was most shocking was when the values reaches 256,000 on bubble sort, the ratio from 128k-256k was 7.0 with a 66 minute run time. I did not try tests that large afterwards.

—

- **JDK SORT**

Using JDKSort, which has a runtime complexity of nlog(n), which is slowed by a factor of *2/log(n) + 2,* I was expecting to see ratios of above 2 and much below 4.

Although I did receive a variety of different ratios, in the 2.0 range. Mostly from 2.1-2.7, as the numbers increased the began to close in on the ratio value of 2.4-2.5. I guess using doubles paid off here. This is when the input values would start at 250k+ up to 1,046k. These tests took minutes compared to the Bubble sort struggling and breaking my computer over these same numbers.

I assume for the same issues as bubble sort that there would be drastic differences in my tests, depending on the data itself being drastically different between tests as the data would be randomized and one test could be easier to solve that its doubled and randomized counterpart.

—-

—In comparison —

I did not expect to feel such a difference in run time as I did while running these codes. What was exceedingly shocking for me was how much faster JDK was able to run in comparison to Bubble sort. While I knew there was a difference in the time complexity, I did not realize how much faster one step up (nlogn -> n^2) would be. It took my computer a whopping 66 minutes to do the 10 sort-tests for 250,000 values in an array. The exact test took JDK sort under a couple minutes. When starting the values to be doubled at 256,000 for JDK sort, doubling 5 times and each run including 10 tests, took only MINUTES!

This would have either broken my computer, or taken days to complete, as the ratio went from 4.4 at 64,000 values to 128,000 values, to a whopping 7.0 runtime ratio at 256,000.

I am guessing this ratio would only increase if I had continued to double the N inputed values.