Nathan Aun

2/18/19

Lab 3

Sorting Algorithms

The combination of sorting algorithms that I will be using for this lab will be a mixture of merge sort and bubble sort. Merge sort will be the main way this is sorted, but once the sub-arrays are of size 4 or less, they will be sorted with bubble sort instead. This is going to be done because it allows for us to skip certain steps that would take place during the merge sort that are almost unnecessary and will hopefully allow for a faster execution of the code. Also, if the merge sorted sub-arrays are already in a sorted position (which isn’t incredibly likely but it’s not entirely out of question either since they are so small) then it won’t need to go through them as much and therefore will, in theory, execute faster than merge sort on its own.

Pseudocode

The algorithm will look roughly as follows:

proposedAlgo (array, left, right)

BEGIN

IF left > right

INIT middle = ((left + right) / 2)

IF (right – left > 4)

proposedAlgo(array, left, middle)

proposedAlgo(array, middle+1, right)

END IF

IF (right – left <= 4)

INIT n = right – left

FOR i = 0 < n

FOR INIT j = left < right – i

IF (array[j] > array[j+1])

Swap the two numbers

END IF

END FOR

END FOR

END IF

Merge(array, left, middle, right)

END

Testing the Program

For testing this program, I will be timing the executions in nanoseconds and will record the amount of time it takes for each section of code to run. I chose nanoseconds because it was small enough to see significant differences in the results of array sizes that won’t cause stack overflow errors. Each array is of size 1000 and a copy of the same array will be tested with all 4 algorithms in order to compare them at each step. The five test cases that I will be using are as follows:

Test 1 will have a completely random array. This result will be run 5 times, with a new array generated each time so we can see how an average run of these algorithms will look without any trend in the data. This one is going to be run 5 times in order to give us a sort of baseline that we will be able to compare other results to. Running it multiple times also helps us deal with potential statistical outliers, as having a randomly generated array has the possibility of giving us atypical results.

Test 2 will take a fully sorted array, from 0 to 9999. This test will hopefully show bubble sort as it’s fastest since the array is already sorted and it will also hopefully show that my proposed algorithm can be faster than merge and quick. It will likely also show quick sort in a bad position because the pivot will always be smaller than the rest of the array.

Test 3 will take a fully sorted array, except it is sorted in decreasing order, starting at 9999 and going to 0. Since all these algorithms sort in ascending order, this is possibly the worse-case scenario for them, so it will be interesting to compare their results at their worst. This will for sure be a worst-case scenario for both bubble sort and quick sort, and since my algorithm takes advantage of bubble sort it will likely show it as slower than merge alone.

Test 4 will take an array that is half sorted, and half random. This will potentially show my proposed algorithm’s advantage over merge sort, where it won’t need to break the first half of the data up as far and will hopefully show it as faster. Quick sort will likely struggle with this algorithm quite a bit because the pivot will always be smaller than the rest of the values until the halfway point.

Test 5 will take an array that has sorted parts and will try to see how they get put together. The first and third quarter of the array is sorted in ascending order while the second and fourth are sorted in descending order. This is done to simulate sorting data that follows a trend of some kind, such as temperatures rising and falling depending on the time of day. This isn’t a perfect representation of a situation like that but it’s the closest to an example of the algorithms working with a “real” situation.

Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Test 1 (Avg of 5 runs) | Test 2 | Test 3 | Test 4 | Test 5 |
| Proposed | 4460136.6 | 5414510 | 4170004 | 3363113 | 6116056 |
| Merge Sort | 3625001.2 | 5671643 | 3644788 | 3628553 | 4649154 |
| Quick Sort | 6388291.2 | 48610899 | 93615522 | 30149080 | 17940001 |
| Bubble Sort | 173380470.8 | 488967 | 44132034 | 97508382 | 81019046 |

(all results have been recorded in Nano-seconds)

As shown in the data above, the proposed algorithm was only the fastest algorithm once in test 4, where the array was half sorted. This makes sense as the algorithm was able to essentially “skip” steps for half of the algorithm. While it wasn’t the fastest in each run-time, it consistently ranked 2nd in every other test case so it wasn’t the worst case either. Merge sort was usually the fastest algorithm in my test cases, with it coming in first during three of the five different test cases. Merge sort also ran within a more consistent range when compared to how the proposed algorithm did, with merge sort taking more or less the same amount of time for each test, where the proposed algorithm fluctuated slightly more. Quick sort was in a bit of a weird place, as none of my chosen arrays were ideal for quick sort. Quick sort was last in two of the five test cases, and second to last in three of them. My first test case is the only one I’m surprised that the algorithm didn’t do better in, but it’s possible that the algorithm got unlucky when choosing pivots, but that’s mostly because it was impossible to have a way to choose a “good” pivot. Lastly, bubble sort was, as expected, the worst algorithm in three of the five test cases. It did have one test case where it was significantly faster than the rest and that was in test 2 where the array is already fully sorted. That makes sense, since bubble sort is the only algorithm that, by default, has a check to see if the array is sorted, while the others just do their job as usual. Overall, it

Conclusion

Through testing and observing my data it is clear that my proposed algorithm isn’t the number one best algorithm for sorting, but in none of them are really. Merge sort may have been fastest in the most categories but that doesn’t mean it’ll always be the best algorithm and it’s interesting to see that the “best” algorithm for the job really depends on the data being sorted. Even though my proposed algorithm wasn’t the fastest it was still very close to merge sort in terms of speed, to the point of it almost not even mattering. I’m sure that using larger sets of data will make this difference matter more but still, using either merge or my proposed algorithm seems like a safe bet regardless of the data that’s being sorted.

A potential follow-up project to further test sorting algorithms could be to compare more combinations of sorting algorithms, as well as comparing them in more conditions than I did. I only looked at one very specific combination of algorithms but there are so many possibilities that it’d be really interesting to see the ways that others come together and how well others work compared to the one used in this lab.