James Zafiri

07/10/2023

CSC6023 – Module 01

**Project 01 Report**

Summary

In this assignment, I implemented the “quicksort” sorting algorithm on 10 random vectors that incremented from sizes 1000-10000. Using C-Profiler, I was able to see the time it took to sort each vector. With these findings, I can analyze the time complexity of this algorithm and see if it is working as expected.

The quicksort algorithm, which is a divide & conquer sorting algorithm like merge sort, is in the Big Oh time complexity class of O(n log n). This is explained in further detail in the asymptotic analysis section. When looking at the C-Profiler data, as mentioned in class, we are not worried about how long it took to create the array, but what we want to know is the time the actual sorting method took. To do this, I looked at the line that actually notes the time of the quicksort function I had in my program. From there, using the C-Profile documentation page, I am focusing on the “cumtime” column: “the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.” Since this function is recursive, I thought it would be more accurate than the “totttime” column.

The algorithm performed how I would have expected it to. Using the example from the PowerPoint in class, algorithms with O(n log n) time complexity (linear-logarithmic) have a linear graph. When looking at the graph I added at the bottom of the report, you can see that it is definitely linear. The only thing I noticed was that the jump from size 6000 to 7000 was smaller compared to every other increment, but the time relative to size of the vector still increased at a constant rate for the most part. The bigger the problem gets, the longer it takes, but it is not like there are very huge jumps in time when dealing with larger data sets. It was a good sign to see this algorithm match up with what we learned in class about the Big-Oh time complexity for O(n log n) functions.

Asymptotic Analysis

Base case: T(n) = O(1), n <= 1

This happens if the length of the array is 1 or if it is empty. The function will then return the array in O(1) time complexity.

Recursive: T(n) = 2T(n/2) + O(n)

If not the base case, the function splits the array into two sub-arrays which are just about half the size. Then using recursion, it will also sort those arrays. Splitting it takes O(n) time since you must go through the entire array.

We will now use the Master method to complete the time complexity analysis.

T(n) = 2T(n/2) + n 🡪 T(n) = 2 \* T(n/2) + n^1 where a = 2, b = 2, d = 1

When you compare a to b^d, this shows that 2 = 2^1 which is correct.

This means that case 2 of the method applies, and this algorithm is in the Big-Oh time complexity of: O(n log n).

C-Profiler Data

For 1000 Elements:

A screen shot of a computer

Description automatically generated

For 2000 Elements:

A screen shot of a computer

Description automatically generated

For 3000 Elements:

A screen shot of a computer

Description automatically generated

For 4000 Elements:

A screen shot of a computer

Description automatically generated

For 5000 Elements:

A screen shot of a computer

Description automatically generated

For 6000 Elements:

A screen shot of a computer program

Description automatically generated

For 7000 Elements:

A screen shot of a computer

Description automatically generated

For 8000 Elements:

A screen shot of a computer

Description automatically generated

For 9000 Elements:

A screen shot of a computer

Description automatically generated

For 10000 Elements:

A screen shot of a computer code

Description automatically generated

Graph – Size vs Time

A graph on a sheet of paper

Description automatically generated