Ben Haase

CS3130 Project 2

10/26/15

**Part A: Merge Sort**

The running time of merge sort is defined by the equation T(n)=2T(n/2) + f(n), where f(n) is the merge step which runs in constant time with O(n). Using the master theorem with a=2, b=2, and d=1, we get 2=21. This tells us that the T(n)εΘ(nlgn). As n gets sufficiently large this equation will be dominated by n giving a linear appearance where n < T(n) < n2. This is shown by the data when represented graphically.

|  |  |  |  |
| --- | --- | --- | --- |
| **Merge Sort (merge two sorted arrays)** | | | |
| **N** | **M = 25** | **M = 50** | **M = 100** |
| 1000 | 0.002 | 0.003 | 0.002 |
| 10000 | 0.024 | 0.025 | 0.025 |
| 100000 | 0.269 | 0.271 | 0.27 |

Two different steps were taken when combining the two sorted arrays, the step show above involves sorting both arrays separately and merging them together into a new array. Since the merge function εO(n) this does not have a large effect on the time complexity of the algorithm which is dominated by nlgn. The second option for combining the two arrays is to append the smaller array to the larger array and resort them. This performed much more poorly than merging the two sorted arrays.

|  |  |  |  |
| --- | --- | --- | --- |
| **Merge Sort (resorting)** | | | |
| **N** | **M = 25** | **M = 50** | **M = 100** |
| 1000 | 0.004 | 0.004 | 0.005 |
| 10000 | 0.049 | 0.048 | 0.048 |
| 100000 | 0.534 | 0.533 | 0.533 |

It takes about twice the time to resort the array but this is what we would expect theoretically since creating the new array will take O(n) time and resorting it adds another nlgn giving us nlgn + n + nlgn = 2nlgn +n, while sorting the two arrays separately will only take nlgn + mlgm + n where is significantly smaller. Since two is a constant however the time complexities will stay relatively similar.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Merge Sorted Arrays f(10n)/f(n)** | | | **Resorting f(10n)/f(n)** | | |
| 10N/N | **M = 25** | **M = 50** | **M = 100** | **M = 25** | **M = 50** | **M = 100** |
| 10000/1000 | 12 | 8.333333 | 12.5 | 12.25 | 12 | 9.6 |
| 100000/10000 | 11.20833 | 10.84 | 10.8 | 10.89796 | 11.10417 | 11.10417 |

By taking the run time of f(10n)/f(n) we are able to see the actually impact the size of n has on the runtime. For we expect to see these coefficients somewhere around 10 (O(n)) and < 100(O(n2)). Both methods of combining the arrays are relatively similar in time complexity and are about what we would expect them to be. Both of these performed similarly for all values of M, it is clear that the size of M has little to no effect on the actual running time of the program for such small values.

**Part B: Quick Sort**  
 The running time of quick sort may vary depending on how well the partition element is selected. In the best case scenario the partition element is exactly the median element and the array is split evenly. For the best case we get the recurrence Tbest(n)=2T(n/2) + f(n) where f(n) is the partition function with f(n)εO(n). Using the master theorem with a=2, b=2, and d=1, we get 2=21. This tells us that the Tbest(n)εΘ(nlgn). In the worst case scenario all elements will either be < or > the partition element. This gives us the recurrence Tworst(n) = T(n-1)+cn; with Tworst(1)=A. So Tworst(n)=T(n-1)+cn ->

T(n-2)+c(n-1)+cn...... -> T(1)+2c+3c+...cn -> A+c(2+3+...+n) = A+c((2+n)/2)(n+1). So Tworst(n)εΘ(n2) giving us a range nlgn ≤ Tavg(n) ≤ n2. Since the running time is dominated by the partition function quick sort will have a running time of O(n+x) solving E[x]= n-1  n  we obtain Tavg(n)εO(nlgn).

Σ Σ 1/(j-i+1)

i=1  j=i+1

This is similar to what was observed in the implementation of this algorithm for all cases.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Quick Sort (Admiral)** | | | **Quick Sort (personal machine with VS)** | | |
| **N** | **QS** | **QS B** | **QS C** | **QS** | **QS B** | **QS C** |
| 100 | 0.0001 | 0 | 0.0001 | 0.00002 | 0.00001 | 0.00002 |
| 1000 | 0.0005 | 0.0004 | 0.0008 | 0.00011 | 0.00012 | 0.00015 |
| 10000 | 0.0069 | 0.0056 | 0.01 | 0.00144 | 0.00139 | 0.00174 |

In the first version of quick sort implemented, the partition element is not selected and is just taken as it is. This worked well in testing due to the arrays consisting of randomly generated numbers. However if an already sorted array was given to this function at higher values of N it would run out of memory. The second version, quick sort B, relied on taking the median of p, q, and floor[(q+p)/2] for the partitioning element. This was coded as a series of if, else if, else comparisons taking a maximum of 3 comparisons to determine the median value. This version of the algorithm performed very well due to the speed it was able to find the median element with. Since the median function takes place in constant time it did not have a noticeable impact on the time efficiency and was well worth the trade off. It also performed excellently when passed already sorted arrays. The third version, quick sort C, performed the worst of all. It relied on selecting 5 random subscripts of the array from p to q and using the median value to partition. Random number generation is a fairly expensive procedure, as is sorting the elements to find the median value even if it is occurring in constant time. The poor performance may also be attributed to the fact that it was passed an array of random numbers making the random selection of elements less helpful. Although it was outperformed by the first version, it did not suffer from running out of memory making it a superior choice for nearly sorted or sorted arrays. Due to the low running times of these algorithms they had to be run on 100 random arrays instead of 10 in order to register time. Repeatedly sorting an array was not an option due to the first version running out of memory on already sorted arrays. Admiral had less precision for the timer than my personal machine with visual studio and would register the smallest value of n for quick sort B so a secondary data set was provided. Looking at the time efficiency by taking f(10n)/f(n) we can see these functions are very close to linear time.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Quick Sort (Admiral)** | | | **Quick Sort (personal machine with VS)** | | |
| **10n/n** | **QS f(10n)/f(n)** | **QS B f(10n)/f(n)** | **QS C f(10n)/f(n)** | **QS f(10n)/f(n)** | **QS B f(10n)/f(n)** | **QS C f(10n)/f(n)** |
| 1000/100 | 5 | NA | 8 | 5.5 | 12 | 7.5 |
| 10000/1000 | 13.8 | 14 | 12.5 | 13.09091 | 11.58333 | 11.6 |

We can see that even though a large constant was introduced with quick sort C its time complexity remains intact and may even outperform on very large data sets.