Sydnee Elliott

CS3310

Project 1

1. Generally, Quick Sort is faster overall, and in this given scenario. However, there are a variety of instances where Merge Sort would be the better option and where it would produce faster results than Quick Sort. Merge Sort has a time complexity of O(n log n) and Quick Sort's time complexity varies, with its average case time complexity being O(n log n). Quick Sort does have a worse worst-case time complexity with it being O(n^2), but since our data in this exercise is randomized, the worst case is avoided and Quick Sort outperforms Merge Sort slightly. If our data were not random and if the lists were partially sorted, Merge Sort would be faster and would outperform the Quick Sort. When combining the two lists back together Merge Sort can just insert the sorted portion of a list rather than comparing every element individually like Quick Sort does. In other situations, such as where the list is largely sorted, Merge Sort would be faster. But in the scenario given, the lists are randomized and no repeat numbers are allowed. Thus, Quick Sort is a “quicker” and more efficient sorting method for the task at hand. Additionally, as the size of our data continually gets longer, Merge Sort would have the upper hand as it uses less comparisons.
2. Merge Sort generally has less basic operations. This is because of the difference in how the two sorting methods put the split lists back together. Merge sort can take advantage of a list that is already in order, joining two lists together, whereas quick sort will still have to compare every element in the first and second list while combining them, regardless of if they are already sorted or not. This allows for Merge Sort to use less comparisons total and leading to less basic operations overall.
3. I believe that my estimated times were pretty close. This is because for both Quick Sort and Merge Sort, the scenario given followed their average case time complexity, which is what I used to calculate the estimated times. If the scenario was not randomized and the lists were partially sorted or allowed for repeat numbers in the lists, my estimates would have been further off, as this would lead to Quick Sort being closer to its worst-case time complexity, which would have made my estimates not follow the time deltas as tightly.
4. Using induction to prove the best basic ops for Merge Sort.

Recurrence Relation: b(n) = 2\*b(n/2) + (n/2)

Closed Form Solution: b(n) = (n\*log2(n))/2

For n>=1 and n is a power of 2

These are the constraints because n must be a positive number and n needs to be a power of 2 since Merge Sort is a divide and conquer algorithm. The lists are continually split in two until down to a list size of 1.

Base Case: n=2

b(n) = b(2)

= 2\*b(2/2) + ½

= 1

And

b(n) = (nlog2(n))/2

= (2\*log2(2))/2

= (2\*1)/2

= 1

Candidate solution satisfies recurrence relation for the base case

Inductive step:

We assume that our base case is true and that our candidate solution (b(k) = k\*log2(k))/2 for the recurrence relation for n = 2k, where k is greater than or equal to 2.

Using our recurrence relation:

b(n) = 2\*b(n/2) + (n/2)

=2\*b(2k/2) + (2k/2)

= 2\*b(k) + k

= 2((k\*log2(k))/2 + k (Using our assumption)

= k\*log2(k) + k

Factor out the k

= k(1\*log2(k) + 1)

Logarithmic identity (log(ab) = log(a) + log(b)

= (k+1)\*(log2(k))

= ((2k/2) + 1)\*(log2(2k/2))

Since n = 2k,

= (n/2 + 1) \* (log2(n/2))

=(nlog2(n) – n + 2log2(n))/2

= (nlog2(n))/2 + (2log2(n) – 1)/2

The first part of this is our candidate solution (nlog2(n))/2 ) plus some constant (2log2(n) – 1).

Thus we see that our candidate solution satisfies the recurrence relation for k >=2 where k is a power of 2.