Bryan Arnold

CSE 2100

Homework 3

12/5/16

[R-12.2]

The downward arrows indicate the separation of a node in the tree, by separation I mean taking half the elements in the previous recursive call or node, to be recursively called again until there is only one element in a node. So, basically the separation of half the node elements and then leads to the next recursive call on those halves of the total elements. The upward arrows represent the return of the merge sort recursive calls, so when the nodes with elements with an arrow pointing upwards have been sorted, and the recombination of the elements into another node.

[R-12.4]

This array-based implementation of the merge-sort algorithm is stable. For a sort to be stable, if two elements in the array share the same key, the one that precedes the other will precede that element before and after sorting. When the two halves are inserted back into one array, the preceding element that shares a key will always be put into the first half due to the line of code at line 5. That comparator statement ensures that if two elements have the same key, and that I < S.length, the first preceding element will be put in first, making it precede the same element sharing the key next. Even if there are 3 keys all the same in a row, the algorithm handles them one at time, each preceding one before the next.

[R-12.5]

This linked-list-based implementation is not stable. For a sort to be stable, if two elements in the array share the same key, the one that precedes the other will precede that element before and after sorting. When the two queues are inserted into the bigger queue, it is checked if the first elements in the queue are greater than each other or not. If the elements are equal, code line 5, the if statement is false and the default will be to put the first element of the second queue into the bigger queue first. In the initial separation into halves of the queues, the preceding element with the same key would have been put in the first queue, and should have been put in the bigger queue first. This is not the case for this algorithm, so this algorithm is not stable.

[R-12.9]

Since the pivot to be selected will always be the element in the middle of the sorted sequence, this new version will split the sequence of elements into two almost equally sorted pieces. This division of the sequence for sorting would be logn. But since the pivot is in the middle, every element of the sequence causes another run, so this would-be n. Combining these two running times at the same time creates a run time of O(nlogn).

[R-12.13]

Changing this would be a flaw because eventually the right and left sides of the sequence will meet at the chosen pivot. If the if statement had no <= to it, when the two sides of left and right reach the pivot, it won't be known that they are at the that the element that left and right markers are now on is in fact in place. This would then be sorted by accident later per the algorithm, making the sorting almost never end, or most likely never end. An example sequence for this would be the sequence {76, 20, 59, 42, 10, 29, 100, 56} with left being 85 and right being 96 and the pivot being 56. The sort would run as normal until left and right hit 42, which they would skip over and assign 42 as the new pivot. This pivot will now sort the rest of the algorithm incorrectly and will be sorted itself later when left and right match up to another pivot.

[R-12.14]

This change would be a flaw because when the left and right markers cross each other, it won't be registered by the if statement and won't shrink the range of values to be searched and swapped. The shrinking of the range ensures that new values are compared to each other with the pivot and not old ones, while the new statement can attempt to resort old values that have been sorted per the pivot. An example sequence would be the same sequence as before, {76, 20, 59, 42, 10, 29, 100, 56} with left being 85 and right being 96 and the pivot being 56. When they cross, the swapping of the left and right won't occur and the range won't shrink. So the range will still be the same, making older values subject to being sorted again, increasing run time.

[R-12.18]

The bucket sort algorithm is not in place. For an algorithm to be in place, it must only use a constant amount of memory in addition to that of objects being stored. There is a pointer B that must grow to the same amount of memory size that S has, but this pointer is not in place with S. So, more memory is needed as extra space to compensate for O(S) more memory to grow. Since the elements also must end up in a different place to be sorted, this is a new memory holder, B, which is extra memory space beyond the object memory being stored and the constant memory of S. The buckets are constant, but the elements will be moved to a new place. So, this algorithm is not in place.

[R-12.19]

To do a radix-sort of this situation, it is relatively simple. First, you would need to iterate through the index i of the triplets. Start at the last index $(d - 1)$ and go down to the first index, sorting as you go along. To extend this for d-tuples, for each iteration, the triplets need to be sorted by their entry. To do this, use a bucket sort of the triplets of S by their i-th entry using N buckets. This would create a radix-sort method for lexicographically sorting a sequence of S triplets.

[R-12.20]

For the merge-sort implementations, the run time of them would be O(nlogn). Merge sort is not dependent on what the element values are, this does not affect the merge sort implementation. The runtime of the implementation would be O(nlogn) time normally, but in sequences of n

elements, this merge sort algorithm becomes more linear, so it's real runtime is O(n), where n is the number of elements. Quick sort on the other hand, is dependent on the values of the elements. Since the element's values are only 0 and 1, this means that the pivot chosen will always either be the biggest or smallest value in the sequence. So, in order to properly sort this, the algorithm will have to check each value in the sort twice, making the run time O(n^2) time.

[R-12.21]

To sort the sequence stably with the bucket sort algorithm, the run time is mainly based upon how far apart the element values are in the sequence. If the elements' values are super spread out or super close together, the run time will be quick. The bucket sort is at its slowest when there are multiple values that are the same, since the bucket must allocate and take in all those elements. This would increase the size of the bucket to be large and would cause a separate sorting algorithm to take longer. So, in this case, if all the elements are 0 and 1, this would create two very large buckets. This would make the run time become O(n^2), since all the elements must be put into buckets and then sorted.

[R-12.25]

First, given a sequence, S = (1, 2, …, n). Now, take two empty sequence sets of L and R, as well as letting p = a1. (This theorem was found online) For i = 2, ..., n, if ai ≤ p appends ai to L otherwise append ai to R. Now, instead of appending to L as stated previously, let L remain empty and R contain n -1 elements. This step requires n−1 comparisons. Through mathematical induction, the recursive call to a quick select algorithm with set R and n -1, Quickselect(r, n−1), needs (n−1)(n−2)/2 comparisons. This means that the whole algorithm needs n − 1 + (n − 1) (n − 2)/2 = n (n − 1)/2 comparisons. Thus, it can be concluded that the run time is Ω(n^2), since the number of elements to be compared is n^2.