CS 61B Discussion 10 Solution Fall 2019

1 Sorting I

Show the steps taken by each sort on the following unordered list:

`106, 351, 214, 873, 615, 172, 333, 564`

(a) Quicksort. After each partition during the algorithm, write the ordering of the list, circle the pivot that was used for that partition, and box the sub-array being partitioned. Assume that the pivot is always the first item in the sublist being sorted and that the array is sorted in place.

In the solution, we will **bold** the next pivot(s) to be used and (circle) the current unsorted elements of the list.

106 351 214 873 615 172 333 564
106 **351** 214 873 615 172 333 564
106 **214** 172 333 351 **873** 615 564
106 172 214 333 351 **615** 564 873
106 172 214 333 351 564 615 873

Note that depending on how the partition method is implemented, you may have different orderings of elements on either side of your pivot after a partition. The only property that must hold is that all elements less than your pivot go to the left, and all elements greater go to the right.

(b) Merge sort. Show the intermediate merging steps.

106 | 351 | 214 | 873 | 615 | 172 | 333 | 564
106 351 | 214 873 | 172 615 | 333 564
106 214 351 873 | 172 333 564 615
106 172 214 333 351 564 615 873

(c) LSD radix sort. Show the ordering of the list after each round of counting sort.

106 351 214 873 615 172 333 564
351 | 172 | 873 333 | 214 564 | 615 | 106
106 | 214 615 | 333 | 351 | 564 | 172 873
106 172 | 214 | 333 351 | 564 | 615 | 873

2 Sorting II

Match the sorting algorithms to the sequences, each of which represents several intermediate steps in the sorting of an array of integers. Assume that for quicksort, the pivot is always the first item in the sublist being sorted.
Algorithms: Quicksort, merge sort, heapsort, MSD radix sort, insertion sort.

(a)
```
12, 7, 8, 4,  10, 2,  5,  34, 14
7,  8, 4, 10, 2,  5,  12, 34, 14
4,  2, 5, 7,  8,  10, 12, 14, 34
```

Quicksort, using the first element as a pivot.

(b)
```
23, 45, 12, 4, 65, 34, 20, 43
4, 12, 23, 45, 65, 34, 20, 43
```

Insertion sort. The first four elements are sorted.

Another solution is merge sort (with a recursive implementation), where the left half has already been fully merge-sorted.

(c)
```
12, 32, 14, 11, 17, 38, 23, 34
12, 14, 11, 17, 23, 32, 38, 34
```

MSD radix sort. We have sorted by the first digit so far.

(d)
```
45, 23, 5,  65, 34, 3,  76, 25
23, 45, 5,  65, 3,  34, 25, 76
5,  23, 45, 65, 3,  25, 34, 76
```

Merge sort. We have finished 2 levels of merging stages.

(e)
```
23, 44, 12, 11, 54, 33, 1, 41
54, 44, 33, 41, 23, 12, 1, 11
44, 41, 33, 11, 23, 12, 1, 54
```

Heap sort. The second line creates a max-heap.

3 Runtimes

Fill in the best and worst case runtimes of the following sorting algorithms with respect to n, the length of the list being sorted. For radix sort, assume we are sorting integers and k is the average number of digits in the strings being sorted. If the runtimes are different for MSD and LSD, specify for both algorithms.

	Insertion sort	Quicksort	Merge sort	Heapsort	Radix sort
Worst case	n^2	n^2	$n \log n$	$n \log n$	nk
Best case	n	$n \log n$	$n \log n$	n	nk

(a) Insertion sort.
Worst case: $\Theta(n^2)$ - If we use a linked list, it takes $\Theta(1)$ time to sort the first item, a worst case of $\Theta(2)$ to sort the second item if we have to compare with every sorted item, and so on until it takes a worst case of $\Theta(n-1)$ to sort the last item. This gives us $\Theta(1)+\Theta(2)+...+\Theta(n-1) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$ worst case runtime. If we use an array, we can find the right position in a worst case of $\Theta(\log n)$ time using binary search, but we then have to shift over the larger items to make room for the new item. Since there are n items, we once again get a worst case runtime of $\Theta(n^2)$.
Best case: $\Theta(n)$ - If the list is almost sorted and has only $\Theta(n)$ inversions, then we only have to do $\Theta(n)$ swaps over all the items, giving us a best case runtime of $\Theta(n)$.

(b) Quicksort.
Worst case: $\Theta(n^2)$ - If we end up always choosing the smallest item or largest item for our next pivot, each partition of a sub-array of size m will leave us with only one non-empty partition of size $m-1$, since every other item will be larger than or smaller than our pivot. With a total of n elements, we end up recursing n times, each time doing a linear time partition, for a total of $\Theta(n^2)$ time.
Best case: $\Theta(n \log n)$ - If we can always partition our sub-arrays roughly in half, our recursive call tree is identical to that of merge sort.

(c) Merge sort.
Worst case: $\Theta(n \log n)$ - At each level of our tree, we split the list into two halves, so we have $\log n$ levels. We have to do comparisons for all of the elements at each level, so our runtime is $\Theta(n \log n)$. Another approach is to sum up the amount of work done at each level of the call tree. Each level has 2^i nodes, and each node does $n/(2^i)$ amount of work from the linear-time merging operation. So that total amount of work is $\sum_{i=0}^{\log n} 2^i(\frac{n}{2^i}) = n \log n$.
Best case: $\Theta(n \log n)$ - We still have to do all of the comparisons between items regardless of their ordering, so our worst case runtime is also our best case runtime.

(d) Heapsort.
Worst case: $\Theta(n \log n)$ - For this procedure we use a max-heap. If all of the items are distinct, then creating a valid heap from the array takes $\Theta(n)$ time using the Heapify procedure discussed in lab. Then we keep removing the maximum valued item (the root), but this takes $\Theta(\log n)$ for each item since we have to replace the root with the last item and bubble it down. Since there are n items, this takes $\Theta(n \log n)$ time. $\Theta(n)+\Theta(n \log n) = \Theta(n \log n)$.
Best case: $\Theta(n)$ - If all of the items are the same, removing the maximum valued item takes $\Theta(1)$ time since we don't have to bubble the new root down. This gives us a runtime of $\Theta(n)$.

(e) Radix Sort.
Worst case: $\Theta(nk)$ - There are n items, and each have approximately k digits. For each of these digits, we have to look through all n numbers and sort them by that digit. Since there are k digits and n integers, this gives us a runtime of $\Theta(nk)$.
Best case: LSD: $\Theta(nk)$ - For LSD, you still have to look through all n items k times regardless of the input, so you get $\Theta(nk)$.
MSD: $\Theta(nk)$ - MSD radix sort can short-circuit once each item is in its own bucket. If $n \leq r$,

where r is the size of the radix, then it's possible to get each item in its own bucket after only 1 pass of counting sort. This yields a runtime of $\Theta(n)$. However, this is not strictly a correct asymptotic runtime. As n grows towards infinity and becomes greater than r, it's not possible to put every item in its own bucket after looking at only the most significant digit. We need to look at at least $\log_r n$ digits to possibly get all of our items in their own buckets and end MSD early. If $\log_r n$ is greater than k, the length of our items, we can't end MSD early and will simply go through the entire k passes of counting sort, leading to a runtime of $\Theta(nk)$ regardless of input.

4 Comparing Algorithms

(a) Give an example of a situation where using insertion sort is more efficient than using merge sort.
Insertion sort performs better than merge sort for lists that are already almost in sorted order (i.e. if the list has only a few elements out of place or if all elements are within k positions of their proper place and $k < \log N$, as this implies that there are less than $N \log N$ inversions).

(b) When might you decide to use radix sort over a comparison sort, and vice versa?
Radix sort gives us nk and comparison sorts can be no faster than $n \log n$. When what we're trying to sort is bounded by a small k (such as short binary sequences), it might make more sense to run radix sort. Comparison sorts are more general-purpose, and are better when the items you're trying to sort don't make sense from a lexicographic perspective or can't be split up into individual "digits" on which you can run counting sort. However, if comparisons take a long time, radix sort might be a better option. Consider sorting many strings of very long length that are very similar. Using a comparison sort will take at least $n \log n$ comparisons, but each comparison may require us to iterate through entire strings, giving us a runtime of $nL \log n$, where L is the average length of our strings. Meanwhile, radix sort will give us a better runtime of nL. On the other hand, if our long strings are very dissimilar, our comparisons will take constant time because we can quickly determine that 2 strings are unequal. In this case, a comparison sort's runtime can be $n \log n$, which will likely be smaller than a radix sort's nL runtime.