CS 61B Discussion 10 Solution Fall 2015

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 (assume the pivot is always the first item in the sublist being sorted and the array is sorted in place). At every step circle everything that will be a pivot on the next step and box all previous pivots.

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

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

(c) LSD radix 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.

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

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

(c) 12, 32, 14, 11, 17, 38, 23, 34
12, 14, 11, 17, 23, 32, 38, 34
MSD radix sort

(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

(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

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, along with when that runtime would occur.

	Insertion sort	Selection sort	Merge sort	Heapsort	Radix sort
Worst case	n^2	n^2	$n \log n$	$n \log n$	nk
Best case	n	n^2	$n \log n$	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 $\Theta(1)+\Theta(2)+\ldots+\Theta(n-1) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$ 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, then we only have to do $\Theta(n)$ swaps over all the items, giving us a best case runtime of $\Theta(n)$.

(b) Selection sort.
Worst case: $\Theta(n^2)$ - Finding the first smallest item takes $\Theta(n)$ time since we have to pass through all of the items. Finding the second smallest item takes $\Theta(n-1)$ time since we have to pass through all of the unsorted items. We repeat this until we only have one item left.

Our runtime is thus $\Theta(n)+\Theta(n-1)+\ldots+\Theta(1)=\Theta(\frac{n(n+1)}{2})=\Theta(n^2)$.
Best case: $\Theta(n^2)$ - We have to pass through all of the unsorted elements regardless of their ordering to search for the smallest one, so our worst case runtime is the same as our best case runtime.

(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 $\Theta(n\log n)$.
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: $\Theta(n\log n)$ - If all of the items are distinct, then creating a valid heap from the array takes $\Theta(n)$ time since we have to sink each item. Then we keep removing the minimum 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 minimum valued item takes $\Theta(n)$ time since we don't have to bubble the new root down. This gives us a runtime of $\Theta(n)$.

(d) 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 nn integers, this gives us a runtime of nk.
Best case: $\Theta(nk/r+n)$ - MSD radix sort can short-circuit if each sub-list is of size 1 after running. For example, with the list [2122, 511, 925, 31]. One iteration will count the number of , yielding a runtime of $r+n$ where r is the radix and n is the number of elements. For LSD, you still have to look through all n items k times, so you get $\Theta(nk)$.

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 $k<\log N$).

(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 beter when the items you're trying to sort don't make sense from a lexographic perspective. Radix sort can also be very inefficient for large k.