1 Sorting I

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

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

(a) Insertion sort. Show the sorted and unsorted portions at every step.

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

(b) Selection sort. Show the sorted and unsorted portions at every step.

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

(c) Merge sort. Show how the list is broken up at every step.

[106 351 214 873 615 172 333 564]
[106 351 214 873] [615 172 333 564]
[106 351] [214 873] [615 172] [333 564]
[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]

(d) Use heapsort to sort the following array (hint: draw out the heap):

106, 615, 214, 873, 351.

873 615 214 106 351 (turns the array into a valid heap)
615 351 214 106 873 ('delete' 873, then sink 351)
351 106 214 615 873 ('delete' 615, then sink 106)
214 106 351 615 873 ('delete' 351, then sink 214)
106 214 351 615 873 ('delete' 214)
106 214 351 615 873 ('delete' 106)

(e) Give an example of a situation when 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 < logN). Also if N < roughly 15 or so.

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: Heapsort, merge sort, insertion sort, selection sort.

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

Selection sort

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

Insertion sort

(c)
```
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

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

Insertion 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
Worst case	n^2	n^2	$n \log n$	$n \log n$
Best case	n	n^2	$n \log n$	n

(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, 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 $\Theta(n)+\Theta(n-1)+...+\Theta(1)=\Theta(\frac{n(n+1)}{2})=\Theta(n^2)$.
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 $\Theta(n)$.

4 MergeTwo

Suppose you are given two sorted arrays of ints. Fill in the method mergeTwo to return a new array containing all of the elements of both arrays in sorted order. Duplicates are allowed (if an element appears s times in a and t times in b, then it should appear $s+t$ times in the returned array).

```
public static int[] mergeTwo(int[] a, int[] b) {
    int[] merged = new int[a.length + b.length];
    int i = 0;       // current index in a
    int j = 0;       // current index in b
    int k = 0;       // current index in merged
    while (i < a.length && j < b.length) {
        if (a[i] < b[j]) {
            merged[k] = a[i];
            i++;
            k++;
        } else {
            merged[k] = b[j];
            j++;
            k++;
        }
    }
    while (i < a.length) {
        merged[k] = a[i];
        i++;
```

```
            k++;
        }
        while (j < b.length) {
            merged[k] = b[j];
            j++;
            k++;
        }
        return merged;
    }
```