SE 2C03 - Assignment 2 solution suggestions

2.1.6 Insertion sort performs a single comparison per element (total $\sim N$) and no exchanges, while selection sort needs to go over $n-1, n-2, \ldots, 2, 1$ comparisons (total $\sim N^2/2$) and no exchanges.

2.1.7 Insertion sort needs $O(N^2)$ comparisons and $O(N^2)$ exchanges, while selection sort needs $O(N^2)$ comparisons and $O(N)$ exchanges, so it is faster.

2.1.8 We use the property "The number of exchanges used by insertion sort is equal to the number of inversions in the array, and the number of compares is at least equal to the number of inversions and at most equal to the number of inversions plus the array size." Suppose that the three possible key values are $a < b < c$. Then about a third of the elements have value a, a third b, and a third c. Since the ordering is random, about half of the about $O(N^2)$ (a, b) pairs will be inversions, so the running time will be quadratic (of course you get about that many inversions from $(a, b), (b, c)$ as well, but we already have the answer to our question...)

2.1.14 The deck will always have two parts: a sorted bottom part (initially empty) and an unsorted top part. Always keep the smaller of the top two cards (and move the other to the bottom) until the second card from the top is the beginning of the sorted part. Then the top card is in its correct place! Move the sorted part from the top to the bottom of the deck (repeated exchanges) and repeat. The whole process is essentially a simulation of selection sort using the deck operations we are given.

2.1.20 Shellsort performs a (fixed) sequence of insertion sorts; the best case for insertion sort is a sorted array; therefore the best case for shellsort is a sorted array.

2.2.19 A recursive algorithm (in the spirit of Merge sort): Split the array into two halves; recursively count the inversions i_1, i_2 within each of the two halves; sort the two halves; merge the two (sorted) halves counting the inversions i_m between the two halves (if the next item to be picked comes from the first half, then it creates an inversion with all items of the second half that have been already 'merged'; if the next item to be picked comes from the second half, then it creates an inversion with all items of the first half that have been not been 'merged' yet); return $i_1 + i_2 + i_m$.

2.2.21 Sort the three lists using merge sort. Then merge them just like in merge sort, so that the first time you encounter the same name in all three arrays, you return it (or return NULL when one of the arrays 'runs out' of elements).

2.2.22 Repeat the proof of Proposition F to prove that it runs in $O(NlgN)$.

2.3.13 The best depth is achieved when the recursion tree is balanced, i.e., if each partition halves the array; then, its depth is lgN. In the average case you expect the two parts in each partition to be about the same size, hence the recursion tree is again $\sim clgN$ for some constant $c \geq 1$. In the worst case, every partition has an empty side, thus forcing to recurse on subproblems reduced in size by just a single element; hence the depth of the recursion is linear.

2.3.15 Maintain two arrays, one for nuts (A) and one for bolts (B). Now simulate quicksort, by finding the bolt (say $B[j]$) that fits nut $A[1]$; then partition B using $A[1]$ and A using $B[j]$ (note that the partitions of the two arrays are identical size-wise, since $A[1], B[j]$ fit at the same place in the ordering of nuts and bolts respectively; then recurse on the two partition sides of A, B.

2.3.20 The stack is used to keep track of the 'partition points' along the path of execution, so its size corresponds to the recursion depth. Since we store at least half (i.e., the larger of the subarrays) of the remaining elements at the stack every time, this number is at least halved (it may become even smaller than half) with every partitioning, and therefore we get a depth of at most lgN (since we start with all N elements in the array initially).

2.4.23 Proposition Q changes to $(t+1)\log_t N$ the time for a *delete maximum* and following 2.4.20, we have at most $(t+1)N$ comparisons for the heap construction (we compare (almost) every element with the maximum of its t children, which we needed t comparisons to find). Therefore, there are at most $2(t+1)N\log_t N = 2(t+1)\frac{\lg N}{lgt}$ comparisons, and the coefficient of $NlgN$ is $(t+1)/lgt$ which is minimized for $t=4$.

2.4.30 Follow the hint, to maintain a min-heap with the $N/2$ largest elements and a max-heap with the $N/2$ smallest elements.

2.4.32 Then you'd need $N\log\log N$ time to implement heapsort, which is impossible because of Proposition I in p.280.

2.5.20 Many different answers. One is to use two heaps (a min one for starting times of jobs, and a max one for finish times) for your calculations (to keep track of the earliest starting time and the latest finish time of overlapping job execution intervals).