Problem 1

a) For $k = 1$ this trivially holds since we compute the max of the S_i's max.

Assume it is true for all values $< k$. Then we know that up to and including value $j = k - 1$ we produce the top $k - 1$ results.

Now the next result is either already contained in A's heap at time of deletion of the top $k - 1$ in which case it would be produced next or it is in a heap S_r.

But such heap can only hold this value if its largest element was in the top $k - 1$, and hence at time of deletion of the element in top $k - 1$ it got transferred into A's heap.

b) $\Theta(n(\log n + \log m))$

The loop runs n times. Each iteration, it does a Heap.DeleteMax() on the current heap, which requires $\Theta(\log m)$ time in the worst case, followed by a Heap.Insert() on server A, which requires $\Theta(\log n)$ time (in the worst case).

Therefore, $\Theta(n(\log n + \log m))$.

c) $\Theta(k(\log n + \log m))$

The loop runs k times. Each iteration, it does a Heap.DeleteMax() on A. This is $\Theta(\log n)$ time. Next, it performs a Heap.DeleteMax() on the server which had originally had item res, which will cost $\Theta(\log m)$ time. Then it will do Heap.Insert() on A which will require $\Theta(\log n)$ time.

Therefore, we get $\Theta(k(\log n + \log m))$ in the worst-case.

d) The proof is similar to part (a) except that A's sorted list hold only largest k elements seen so far and discards the rest. So we must make sure that we never require a value that has been discarded.

The proof proceeds by induction. For $k = 1$ this trivially holds since we compute the max of the S_i's max.

Assume it is true for all values $< k$. Then we know that up to and including value $j = k - 1$ we produced the top $k - 1$ results. Now the next result is either already contained in A's sorted list at time of deletion of the top $k - 1$ in which case it would be produced next or it is in a sorted list S_r.

Observe that it cannot be in the discarded pile from A's list as only elements which have lost k comparisons are ever discarded.

Again as in part (a) the sorted list can only hold this value if its largest element was in the top $k - 1$, and hence at time of deletion of the element in top $k - 1$ it got transferred into A's sorted list where it still resides as it cannot itself lose k comparisons.

e) $\Theta(nk)$

The loop runs n times. Each time, it calls K_InsertionSort(k, res). Inside K_InsertionSort(k, res), all the checking is constant time. The only part of the code where it is not constant time is when we call A.SortedCandidateList.InsertionSort(item). This is when we can input "item" into A's list. Because we know that this list is not any bigger than size k (and we know that the list is already sorted), inserting an item would just cost $\Theta(k)$ time (in the worst case).

Altogether, the run time is $\Theta(nk)$.

f) $\Theta(k^2)$

The loop will run k times. Within the loop, SortedCandidateList.DeleteMax() and S_i.SortedList.DeleteMax() takes $\Theta(1)$ time (since it is a list).
A.SortedCandidateList.K_InsertionSort(k, res) takes $\Theta(k)$ time (as explained in part e).

Dropping all constants, the runtime is $\Theta(k^2)$

g) We accepted the answers $k \in O(\log n)$, $k < \log n$, $k < \log n + \log(10000)$ or again $k \in O(\log n + \log m)$ and $k < \log n + \log m$

Recall that the worst case costs T_{Heap} and T_{Marc} of respectively the heap based solution and Marc's solution satisfies

$$T_{\text{Heap}}(n, k, m) \in \Theta((n+k)(\log n + \log m)), \quad T_{\text{Marc}}(n, k, m) \in \Theta((n+k)k).$$

For this question, we will make the simplifying assumption that actually

$$T_{\text{Heap}}(n, k, m) = (n+k)(\log n + \log m), \quad T_{\text{Marc}}(n, k, m) = (n+k)k.$$

With this assumption, it is easier to understand how the solutions compare. Indeed Marc's solution is faster when $k < \log n + \log m = \log n + \log(10000) \simeq \log n + 13$.

Note : We have not proved that Marc's solution is faster when $k < \log n + \log m$. We just expect that it will be the case because we made the "reasonable" assumption that the runtime of every instance would behave like the worst-case, and that the constants inside the big-Oh are 1.

h) $k = 1000$ is bigger than $\log(n) + \log(m) \simeq 37$. Therefore, the heap solution should be much faster.