Solutions for Homework Assignment #1

Answer to Question 1. (12 marks)

a. (4 marks) Let $T(n)$ be the worst-case running time of the given HeapSort algorithm on an array of n elements. To prove an upper bound on the growth rate of $T(n)$, we argue that $T(n)$ cannot be more than $c \cdot n \log n$ for some positive constant c.

Since A contains n elements, the BuildMaxHeap call on line 2 takes at most $O(n)$ time (we proved this result in class). Since the heap H never contains more than n elements (it has exactly n following the BuildHeap, and the algorithm never inserts any additional elements into the heap), each ExtractMax operation on line 4 takes at most $O(\log n)$ time (again, we saw this result in class). The for loop executes at most n times (actually, exactly n times), so the total cost of the loop over all iterations is at most $O(n \log n)$. The algorithm does only a constant amount of work on the remaining lines, thus the running time for HeapSort is at most $O(n + n \log n) = O(n \log n)$.

b. (8 marks) To prove a lower bound on the growth rate of $T(n)$, we must demonstrate a "bad" input that takes at least $c \cdot n \log n$ time for some positive constant c. If we can demonstrate one input that takes at least this much time, then the worst-case must certainly be no less than this. Remember that we must show such a "bad" input for every n (for big enough n, i.e., for n bigger than some bound).

The key for this question is to pick a "bad" order of input, and make some argument on how much work (how many comparisons) must be performed during the algorithm. Let us consider the input $I = [n, n-1, \ldots, 2, 1]$. This order corresponds to a heap, so the heap built by BuildBinaryHeap won't change the order.

We observe the following fact: in a max-heap containing n elements, the largest $m = \lceil \frac{n}{2} \rceil$ elements in the heap form a subtree within the heap (in other words, the biggest elements form a connected portion of the heap; this doesn't have to be the upper triangle of the hear, or a subtree of a particular node, only that the smaller elements can't be between two of these m elements). Also, at least $\lfloor \frac{m}{2} \rfloor$ of these elements must be non-leaves of that subtree, since a binary tree with k leaves has at least $k-1$ non-leaves. Therefore, at least $\lfloor \frac{m}{2} \rfloor$ of the largest m elements appear in the first $\lfloor \frac{n}{2} \rfloor$ positions of the heap (they don't have to appear consecutively).

The key point is that those elements must each be promoted to the root position before reaching their final destinations. When we do the first $\lfloor \frac{n}{2} \rfloor$ ExtractMax operations, the root has to be the largest remaining element, so the largest $\lfloor n2 \rfloor$ elements must be propagated *up* to the root position. This includes the $\lfloor \frac{m}{2} \rfloor$ non-leaf elements. (It is important that they are non-leaves, since during the first $\lfloor \frac{n}{2} \rfloor$ extracts leaves can jump to the root position and propagate down, but non-leaves must propagate up.) The movement of these $\lfloor \frac{m}{2} \rfloor$ elements propagating up to the root contributes at least $\sum_{k=1}^{\lfloor \frac{m}{2} \rfloor} \log k = \frac{1}{2} m \log m + O(m) \in \Omega(m \log m) = \Omega(n \log n)$.

[Credit was given for knowing how to approach this problem correctly, even if you were not able to get all the details... as you can see, some of the details are a bit challenging to come up with and do correctly.]

Answer to Question 2. (15 marks)
The basic idea is to iteratively find the minimum element of all lists, return it as the next minimum element and replace it with the next element in its list (and, of course, to use a heap for these operations!).

MergeLists($l_1, l_2, \ldots, l_k$):
//Precondition: the lists $l_1, l_2, \ldots, l_k$ are sorted in nondecreasing order

1: for each list l_j, create a node x_j containing the value of the first element of list l_j, its list number j and its index (in this case, 1). (Each node has three fields: *value*, *listNum* and *listPos*.)
2: create an array B of k nodes, with node x_j stored in position j of B
3: $H \leftarrow$ BuildMinHeap(B) {where Min-Heapify arranges nodes based on the *value* field}

4: create an array A of n elements for storing the sorted list
5: **for** $i = 1$ to n **do**
6: $x \leftarrow H$.ExtractMin() {ExtractMin uses the *value* field to select the minimum}
7: $A[i] \leftarrow x.value$
8: insert into H a node with the element at position $x.listPos + 1$ of list $l_{x.listNum}$ (along with its list and index number), if such an element exists
9: **end for**
10: return A

Correctness: All the lists l_i are provided sorted in nondecreasing order (for simplicity, we'll assume no duplicate elements for now). Therefore, we know that initially the smallest element is in the first position of some list. By inserting the elements at the front of each list into a heap and extracting the minimum, we find the first element. Assume that the first element is in list j. Now, we have to find the second smallest element. This element is either already in the heap (i.e., its the smallest element in some list that is not j) or it is the second element in list l_j. Thus, to find the second smallest element, we insert the second element from list j into the heap, and extract the minimum. Inductively, we can show that after extracting the i^{th} largest element from the heap, the $i + 1$ largest element will either still be in the heap or it will appear immediately following the i^{th} largest element in a list. By recording the list number and index number of each element in the heap, we can easily find the element that appears after the i^{th} largest element in a list. As a result, we can find and insert this new element into the heap (if it exists) and repeat the process. After n iterations of the for loop, n elements will be extracted from the heap and all elements will have been returned in order from smallest to largest. (If there are duplicate elements, it doesn't matter in which order we choose the identical elements, so the same argument holds.)

Worst-case running time: We consider each stage of the algorithm:

1. Creating all the nodes x_j for each list l_j takes $O(k)$ time.
2. Building an array with the k nodes also takes $O(k)$ time.
3. The worst-case running time of BuildMinHeap is in $\Theta(k)$.
4. Each of the ExtractMin and Insert operations have worst-case time complexity in $O(\log k)$, since there are at most k elements in the heap H when these operations are performed.
5. The for loop is executed n times; thus, the ExtractMin and Insert operations operations are performed at most n times.

Overall, the algorithm MergeLists has a worst-case running time in $O(k + n \log k)$, which simplifies to $O(n \log k)$.

Answer to Question 3. (18 marks)

a. (6 marks) A binomial heap H with n vertices consists of $\alpha(n)$ trees. Let T_i, $1 \leq i \leq \alpha(n)$, denote the trees of H. A tree T_i with n_i vertices has $n_i - 1$ edges. So the total number of edges in H is $\sum_{i=1}^{i=\alpha(n)}(n_i - 1) = (\sum_{i=1}^{i=\alpha(n)} n_i) - \alpha(n) = n - \alpha(n)$

b. (12 marks) Binomial heap H has n nodes before the insertions. By Part (a), it has $n - \alpha(n)$ edges before the insertions. After k consecutive insertions, H has $n+k$ nodes, hence it now has $(n+k)-\alpha(n+k)$ edges. So the number of new edges created during the k consecutive insertions is:
$[(n + k) - \alpha(n + k)] - [n - \alpha(n)] = k + \alpha(n) - \alpha(n + k) \leq k + \alpha(n)$ edges.

The number of pairwise comparisons between the elements of H needed to execute k consecutive insertions is equal to the number of new edges created during these insertions (each new edge is the result of a pairwise comparison, and each pairwise comparison creates a new edge in H). So k consecutive insertions require at most $k+\alpha(n)$ comparisons. By definition $\alpha(n)$ is the number of 1's in the binary representation of n, therefore, $\alpha(n) \leq \lfloor \log_2 n \rfloor + 1$. So k consecutive insertions require at most $k+\lfloor \log_2 n \rfloor + 1$ comparisons. Note that if $k > \log_2 n$, k is the dominant factor in $k + \lfloor \log_2 n \rfloor + 1$. So, when $k > \log_2 n$, k consecutive insertions require just $O(k)$ pairwise comparisons (a constant number of comparisons per insertion on the average).