Solutions for Homework Assignment #1

Answer to Question 1.

a. $T(n)$ is $O(n^2)$. This is because for *every* $n \geq 2$:

(i) For *every* input array A of size n, the outer **for loop** of Line 3 consists of doing *at most* $(n-1)$ iterations, and *each* such iteration causes *at most* $(n-1)$ inner iterations of the nested **for loop** of Line 4; so a total of at most $(n-1)(n-1) < n^2$ inner loop iterations are executed.

(ii) Each inner loop iteration, and each one of the statements in line 1, 2, 4 and 5, takes constant time (because each consists of a constant number of comparisons and additions).

So it is clear that there is a constant $c > 0$ such that for all $n \geq 2$: for *every* input A of size n, executing the procedure **nothing(A)** takes *at most* $c \cdot n^2$ time.

b. $T(n)$ is $\Omega(n^2)$. This is not obvious because the **for loop** of Line 3 may end "early" because of the loop exit condition in Line 5: if the condition of Line 5 is satisfied then the procedure call immediately ends. Thus, to show that $T(n)$ is $\Omega(n^2)$, we must show that there is at least one input array A such that the procedure takes time proportional to n^2 on this input, *despite the loop exit condition of Line 5.* We do so below.

$T(n)$ is $\Omega(n^2)$ because for *every* $n \geq 2$:

(i) There is an input array A of size n, namely array $A[1..n] = \langle 1, 2, 3, 4, ..., n \rangle$ (i.e., the array A such that $A[i] = i$), where the **for loop** of Line 3 causes the execution of $n-1$ outer iterations, one for each i, $2 \leq i \leq n$. This is because for all i, $2 \leq i \leq n$, just before the loop of Line 4 is executed $A[i-1] = i-1$, and after the loop of Line 4 is executed $A[i-1] = i = A[i]$, and so the procedure does *not* return in Line 5.

Thus, *with this specific input*, *each* iteration of the outer **for loop** of Line 3 with $i - 1 \geq n/2$ will in turn cause the execution of $n/2$ inner iterations of the nested **for loop** of Line 4.

So, for input $A[1..n] = \langle 1, 2, 3, 4, ..., n \rangle$, there are at least $n^2/4$ iterations of the inner **for loop** of Line 4.

(ii) Each inner loop iteration takes constant time.

So it is clear that there is a constant $c > 0$ such that for all $n > 1$: there is *some* input A of size n (namely, $A[1..n] = \langle 1, 2, 3, 4, ..., n \rangle$) such that executing the procedure **nothing(A)** takes *at least* $c \cdot n^2$ time.

Important note: For many arrays A of size n, for example all those where $A[2] \neq 2$, those where $A[2] = 2$ but $A[3] \neq 3$, etc..., the execution of procedure **nothing(A)** takes only constant time! This is because the execution stops "early", in Line 5, on these arrays.

So to prove that the worst-case time complexity of **nothing()** is $\Omega(n^2)$, a correct argument ***must explicitly describe*** some input array A of size n for which the execution of **nothing(A)** does take time proportional to n^2.

Note that since $T(n)$ is both $O(n^2)$ and $\Omega(n^2)$, it is $\Theta(n^2)$.

Answer to Question 2.

a. The basic idea is to maintain a Min Heap that contains k elements, specifically, the smallest integer from each one of the k sorted lists $A_1, A_2, \ldots, A_k$. More precisely:

1. First build a Min Heap that contains the following k elements: $(a_1, 1), (a_2, 2), \ldots, (a_j, j), \ldots, (a_k, k)$ where each a_j is the smallest element in the sorted list A_j, and the a_j's are used as the heap keys. Remove each a_j from A_j.

2. Then repeatedly do the following:

   (a) Do an EXTRACT_MIN to find and remove the element with the smallest key from the Min Heap; say this element is (x, i). Output x.

   (b) Note that the above x came from list A_i. Remove the smallest (remaining) element from the list A_i, say it is y, and insert (y, i) into the Min Heap.

   Note: If A_i is empty, then skip step (b). In this case, the Min Heap size decreases by one because the element (x, i) that was removed from the Min Heap in step (a) is not replaced.

Proof Sketch: Note that at the start of each repeat loop (Part 2 above) the Min Heap contains the minimum remaining element of each list: This is clearly true immediately after Part 1 (because we took the smallest element of each list and inserted it in the heap), and it remains true after each iteration of the loop because when we remove the min element from the heap, say this element came from some list j, we replace it with the smallest remaining element from list j.

Since the Min Heap contains the minimum remaining element of each list, the root of this heap is the global minimum of all the remaining elements. So performing m EXTRACT_MIN returns the m smallest elements in increasing sorted order.

b.

1. The initial Min Heap contains k keys. So it takes at most $O(k)$ time to build it using BUILD_MIN_HEAP (a procedure that is very similar to BUILD_MAX_HEAP of Section 6.3).

2. Each one of the m outputs requires:

   (a) one EXTRACT_MIN, which takes $O(\log k)$ time in the worst-case, and

   (b) at most one INSERT, which also takes $O(\log k)$ time in the worst-case.

   Thus the worst-case time complexity for the m outputs is $O(m \log k)$.

So the worst-case time complexity of the above algorithm is: $O(k) + O(m \log k)$, i.e., $O(k + m \log k)$.

Answer to Question 3.

a. 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. Binomial heap H has n nodes before the insertions. Thus, by Part (a), H has $n - \alpha(n)$ edges before the insertions. After the 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 by the k consecutive insertions is:
$[(n + k) - \alpha(n + k)] - [n - \alpha(n)] = k + \alpha(n) - \alpha(n + k) \leq k + \alpha(n)$ edges.

As we explained in class, the number of pairwise comparisons between the keys of H needed to execute k consecutive insertions is equal to the number of new edges created by these insertions: each key comparison creates a new edge in H and each new edge in H comes from a key comparison. So k consecutive insertions require at most $k + \alpha(n)$ key 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, and the average cost per insertion is at most $(k + \lfloor \log_2 n \rfloor + 1)/k = 1 + \lfloor \log_2 n \rfloor / k + 1/k$. Thus, for $k > \log n$, this average cost is less than 3.