Solutions for Homework Assignment #1

Answer to Question 1.

Answer: $T(n)$ **is** $\Theta(n)$. To prove this, we now show that $T(n)$ is both $O(n)$ and $\Omega(n)$:

1. $T(n)$ is $O(n)$.

   Consider the first iteration of the outer loop (lines 3 – 6), when $i = 1$. In this iteration, when j reaches the value $n-1$ in the inner loop (lines 4 – 6), the condition $i + j > n - 1$ in line 6 is met, and the procedure terminates.

   So, for *all* possible input arrays A: (a) there is only one iteration of the outer loop, and (b) the inner loop is executed *at most* $n - 1$ times.

   Since each iteration of the inner loop takes **constant** time (because each one consists of a constant number of comparisons and additions), it is now 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*** $c \cdot n$ time.

2. $T(n)$ is $\Omega(n)$.

   This is not obvious because the procedure may return "early" (e.g., after executing only a constant number of inner loop iterations) because of the loop exit conditions in line 5 and 6. Thus, to show that $T(n)$ is $\Omega(n)$, we must show that there is at least one input array A such that the procedure executes a linear number of inner loop iterations on this input, ***despite the loop exit conditions of lines 5 and 6.*** We do so below.

   Consider the array $A = [-1, 2, -1, 2, -1, \ldots]$ of alternating -1 and 2 of size $n \geq 2$. Note that:

   (a) For all $j \geq 1$, $A[j] = 1 - A[j + 1]$. So with ***this*** input array A, the return condition $A[j] \neq 1 - A[j + 1]$ in line 5 of the code is ***never*** true. Thus, with ***this*** input, the procedure ***never*** returns in line 5.

   (b) When $i = 1$ (in the first iteration of the outer loop of lines 3 – 6), the condition $i + j > n - 1$ in line 6 holds only after at least $n - 2$ complete iterations of the inner loop.

   So, for *some* input array A, namely $A = [-1, 2, -1, 2, -1, \ldots]$, the inner loop is executed *at least* $n - 2$ times.

   Since each iteration of the inner loop takes constant time, it is now clear that there is a constant $c > 0$ such that for all $n \geq 2$: there is ***some*** input A of size n such that executing the procedure NOTHING(A) takes ***at least*** $c \cdot n$ 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 $A[3] \neq -1$, 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 $\Omega(n)$, a correct argument ***must explicitly describe*** some specific input array A of size n for which the execution of NOTHING(A) does take time proportional to n.

Answer to Question 2. (Solution Sketch)

1. The algorithm uses a *binary Max Heap* to store the m smallest keys input so far. It works as follows:

   Insert each of the first m input keys into the Max Heap (using the binary heap INSERT operation). Then:

   - *When a key input occurs:* first insert this key into the Max Heap (using the INSERT operation), and then remove the maximum key of the Max Heap (using the EXTRACTMAX operation).

     A simple induction argument shows that, after m or more input keys occur, the Max Heap contains the m smallest keys that were input so far.
   - *When a print occurs:* print all the m keys that are currently in the Max Heap.

2. It is clear that, at all times, the Max Heap has at most $m + 1$ elements. So each INSERT and EXTRACTMAX operation takes $O(\log m)$ time in the worst-case. Therefore:

   - Processing each input key takes $O(\log m)$ time in the worst-case.
   - Each *print* operation takes $O(m)$ time, because it consists of printing the m keys in the array that represents the Max Heap.