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 **strange(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 $A[1..n] = \langle 0, -1, -2, -3, -4, ..., -n+1 \rangle$, i.e., the array A such that for all i, $1 \leq i \leq n$, $A[i] = -i + 1$, such that the procedure does *not* return in Line 5.

This is because for all i, $2 \leq i \leq n$: (a) just before the loop of Line 4 is executed $A[i-1] = -(i-1)+1$, (b) just after the loop of Line 4 is executed $A[i-1] = -i$, and (c) since $A[i] = -i+1$, in Line 5, we have that $A[i] = A[i-1] + 1$ 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 $i \geq n/2$ will in turn cause the execution of at least $n/2$ inner iterations of the nested **for loop** of Line 4.

So, for input $A[1..n] = \langle 0, -1, -2, -3, -4, ..., -n+1 \rangle$, 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, $A[1..n] = \langle 0, -1, -2, -3, -4, ..., -n+1 \rangle$) such that executing the procedure **strange(A)** takes *at least* $c \cdot n^2$ time.

Important note: For many arrays A of size n, for example all those where $A[2] \neq -1$, those where $A[2] = -1$ but $A[3] \neq -2$, etc..., the execution of procedure **strange(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 **strange()** is $\Omega(n^2)$, a correct argument ***must explicitly describe*** some input array A of size n for which the execution of **strange(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. A ternary (max) heap H with n elements can be represented by an array A with an associated variable $A.Heapsize = n$, such that the elements of H are in $A[1..n]$. The root of H is stored in $A[1]$, and it contains an element with largest key. The children of $A[i]$ (from left to right in H) are $A[3i-1]$ (if $3i-1 \leq n$), $A[3i]$ (if $3i \leq n$) and $A[3i+1]$ (if $3i+1 \leq n$). For $i > 1$, the parent of $A[i]$ is $A[\lfloor \frac{i+1}{3} \rfloor]$.

b.

1. Consider a ternary heap A with n elements. Element $A[i]$ is an internal node of the heap if and only if (iff) it has at least one child. So $A[i]$ is internal iff $A[3i-1]$ is an element of the heap, i.e., iff $3i-1 \leq n$. Thus $A[i]$ is an internal node iff $i \leq \lfloor \frac{n+1}{3} \rfloor$.

2. A ternary heap A with n elements has height $\lfloor \log_3(2n-1) \rfloor$. To see this, note that a complete ternary tree of height h has:

   - *at most* $3^0 + 3^1 + \ldots + 3^h = \frac{3^{h+1}-1}{2}$ nodes, and
   - *at least* $3^0 + 3^1 + \ldots + 3^{h-1} + 1 = \frac{3^h+1}{2}$ nodes.

   So in a complete ternary tree, the height h and the number of nodes n are related as follows: $\frac{3^h+1}{2} \leq n \leq \frac{3^{h+1}-1}{2}$. Thus, $3^h \leq 2n-1$ and $3^{h+1} \geq 2n+1$. Hence, $\log_3(2n+1) - 1 \leq h \leq \log_3(2n-1)$. Therefore $h = \lfloor \log_3(2n-1) \rfloor = \lceil \log_3(2n+1) - 1 \rceil$.

c.

- Insert(A, key): Insert *key* into A.

  Algorithm sketch: (This is identical to the Insert procedure for binary heaps that we saw in class.) First increment $A.Heapsize$ by one. Then put the (element x with) *key* in $A[A.Heapsize]$ (for simplicity, in this description we identify the element x with its key). Finally, "percolate x up"until it settles to the right place, i.e., until the parent of x is greater or equal to x. To do so, keep comparing x with its parent, and swap the two if x is greater.

- Extract_Max(A): Remove a key with highest priority from A.

  Algorithm sketch: (This is similar to the Extract_Max procedure for binary heaps that we saw in class.) First return $A[1]$, then store $A[A.Heapsize]$ in $A[1]$ (replacing the old content of $A[1]$) and decrement $A.Heapsize$ by one. Let x be the element now in $A[1]$. To restore the max-heap property, "drip x down" until it settles to the right place, i.e., until x is greater or equal to each of its children. To do so, keep comparing x with its children, and if one of them is greater, then swap x with the greatest of its children.

- Update(A, i, key), where $1 \leq i \leq A.Heapsize$: Change the priority of element $A[i]$ to *key* and restore the heap ordering property.

  Algorithm sketch: Let x be the element in $A[i]$.

  – If Update(A, i, key): increases the (key of) x, then "percolate x up" until it settles to the right place, i.e., until the parent of x is greater or equal to x. To do so, keep comparing x with its parent, and swap the two if x is greater. This procedure is similar to Insert above.

  – If Update(A, i, key) decreases the (key of) x, then "drip x down" until it settles to the right place, i.e., until x is greater or equal to each of its children. To do so, keep comparing x with its children, and if one of them is greater, then swap x with the greatest of its children. This procedure is similar to Extract_Max above.

- REMOVE(A, i), where $1 \leq i \leq A.Heapsize$: Delete the element $A[i]$ from the heap.

  Algorithm sketch: Let x be the element in $A[i]$. One way to delete x is to first use the UPDATE(A, i, key) procedure to change the key of x to "infinity" (a key greater than any other key in A). This will make x percolate up all the way to the root of the max-heap A. Then execute EXTRACT-MAX(A) to remove x.

Let h be the height of the max-heap A (recall that $h = \lfloor \log_3(2n-1) \rfloor$, where $n = A.Heapsize$). The worst-case time complexity the above algorithms is both $O(h)$ and $\Omega(h)$, because: (1) they never take more than time proportional to h, and (2) they each have at least one execution that does take time proportional to h (e.g., for UPDATE(A, i, key), such an execution occurs when $i = n$, and the new *key* is greater than any other key in A: this execution makes the leaf $x = A[n]$ percolate up all the way to the root of the heap). So the worst-case time complexity of the above algorithms is $\Theta(h) = \Theta(\lfloor \log_3(2n-1) \rfloor) = \Theta(\log_3 n)$.