# CSC263 Assignment 1

Angela Zhu, Xinyi Ji, Zhuozi Zou

January 17, 2019

## 1 Problem 1

(Written by Zhuozi Zou, read by Angela Zhu & Xinyi Ji)
**Define** $f(n) = n$. I will show that $T(n)$ is $\Theta(n)$. In the following explanation, I denote the loop on line 3-6 as the outer loop and the loop on line 4-6 as the inner loop.
Show $T(n)$ is $\mathcal{O}(n)$: Let $A$ be an arbitrary list with size$> 1$. NOTHING($A$) can terminate in two cases: case 1: return on line 5; case 2: not return on line 5.
**case 1:** when $i = 1$, the inner loop can iterate at most $n-1$ times ($j$ goes from 1 to $n-1$). Before or exactly in the $(n-1)$th iteration, $a[j] \neq 1 - a[j+1]$ satisfied so the *return* part on line 5 execute. Since *return* means the procedure stops immediately, in total the outer loop iterates at most 1 time and the inner loop iterates at most $n-1$ times.
**case 2:** in this case, the *if* statement on line 5 can never be satisfied. When $i = 1$, the *if* statement on line 6 will be satisfied when $i + j > n - 1$, this happens as soon as $j = n - 1$ and the *return* part on line 6 executes. Since *return* means the procedure stops immediately, the outer loop iterates 1 time and the inner loop iterates $n - 1$ times ($j$ goes from 1 to $n - 1$).
Thus case 2 costs one more operation to terminate than the worst case of case 1 (one more operation on line 6). Also, in both case 1 and 2, line 7 will never be reached since NOTHING($A$) terminates on either line 5 or line 6. In the worst case (which is case 2): if we count each line as one step, then lines 1-2 takes 2 steps, lines 3-6 takes $2(n - 1)$ steps, and line 7 takes 0 steps. In total there are at most $2n$ steps. Since each assignment, comparison and arithmetic operation takes constant time, we can say that $T(n)$ is $\mathcal{O}(n)$.
Show $T(n)$ is $\Omega(n)$: For each $n \in \mathbb{N}\backslash\{0, 1\}$, consider the input list of length $n$ of the form $A = [0, 1, 0, 1, \ldots]$. Then the *if* statement on line 5 can never be satisfied. When i = 1, the if statement on line 6 will be satisfied when $i + j > n - 1$, this happens as soon as $j = n - 1$ and the *return* part on line 6 execute. Since *return* means the procedure stops immediately, the outer loop iterates only 1 time and the inner loop iterates $n - 1$ times ($j$ goes from 1 to $n - 1$). Also, line 7 will never be reached. If we count each line as one step, then lines 1-2 takes 2 steps, lines 3-6 takes $2(n-1)$ steps, and line 7 takes 0 steps. In total there are at least $2n$ steps. Since each assignment, comparison and arithmetic operation takes constant time, we can say that $T(n)$ is $\Omega(n)$.
Therefore, $T(n)$ is $\Theta(n)$.

## 2 Problem 2

(Parts 1, 2, 3 written by Angela Zhu, part 4 written by Xinyi Ji, entire question read by everyone)

1. The data structure we are using is a max-heap. Memory-wise, all elements are stored in an array representation of this max-heap. In this algorithm, assuming that at least $m$ keys are input before the first print occurs, the max-heap should simply contain the given keys for the first $m$ keys of input. After $m$ inputs, the max-heap contains only the $m$ smallest keys of the current input sequence immediately before taking in the next input key. The root is the current largest key of the $m$ smallest keys of the entire input sequence.

2. For the first $m$ rounds of input, the algorithm receives a key, and performs the $Insert(A, key)$ operation to store the key into the max-heap, following the max-heap property. After $m$ rounds of input, the max-heap should contain $m$ keys. From now on:

   - If a new key is given, compare the new key to the root of the max-heap.
     If new key $\geq$ root, this indicates that the current $m$ keys in the max-heap remains to be $m$ smallest keys of the input sequence. Nothing else needs to be done.
     If new key $<$ root, then the new key is one of the $m$ smallest keys of the input sequence, while the old root is no longer one. The $Extract\_Max(A)$ operation is performed to discard the old root. Instead of replacing the root with the most outer leaf and move it down the heap like we learned in class, the new key directly replaces the root.
   - If the print operation occurs, the following steps are performed:
     Starting from the first element of the array representation of the max-heap, which is the root of the heap, each key in the array is printed. So the order of the printed values would be the node at depth 0 (root), followed by the nodes at depth 1 arranged from left to right, and so on. The operation terminates when all $m$ keys in the are printed.

3. The worst-case time complexity of the described algorithm satisfies:

   - $\mathcal{O}(\log m)$ to process each input key.
     For every new input key after the first $m$ keys, after comparing it with the root which takes 1 step, the algorithm either discards the new key or takes away the old root and inserts the new key. Since discarding the key takes just a constant time, it is not considered in the worst-case analysis. If the old root is replaced, the following steps are performed:
     $Extract\_Max(A)$ removes the root of the max-heap, and in this specific algorithm, the new key is used to replace the root, instead of the deepest leaf of the heap (i.e. last element of the array). We know from class that this operation needs at most $\log m$ steps, in the case that the target key needs to travel all the way from the root to the deepest leaf.
     When there are less than $m$ keys in the max-heap, the algorithm directly inserts the new key into the heap, so the following steps are performed:
     $Insert(A, key)$ is used to insert the new key into the max-heap. Let $n$ be the number of keys currently in the max-heap. As learned from class, we know that this operation takes at most $\log n$ steps to perform. However, $n < m$ and therefore $\log n \leq \log m$. The worst case runtime happens after the first $m$ input keys.
     Thus, the total steps to process an input key is no more than $\log m + 1$ steps, which satisfies $\mathcal{O}(\log m)$.

   - If the print operation occurs, the following steps are performed:
     Starting from the first element of the array representation of the max-heap, which is the root of the heap, each key in the array is printed. So the order of the printed values would be the node at depth 0 (root), followed by the nodes at depth 1 arranged from left to right, followed by the nodes at depth 2, and so on. The operation terminates when all $m$ keys in the max-heap are printed.

4. *Proof.* I will first prove that after $m$ keys of input, the set $A$ will always contain the $m$ smallest keys among all the keys that were input before by induction. **Define** $P(n)$: set $A$ contains the $m$ smallest keys among all the $n$ keys that were input before. I will prove that $\forall n \in \mathbb{N}, n \geq m \Rightarrow P(n)$.
   **Base Case:** $n = m$.
   This means that $m$ keys have been input. According to our algorithm, those $m$ keys are all filled into $A$, which these $m$ keys are also the $m$ smallest keys of the current input sequence since there is no other key.

**Inductive Step:**
Let $n \in \mathbb{N}$, assume $n \geq m$, assume $P(n)$, I will prove that $P(n+1)$ follows. The proof is as the following. Before the last key was input there were $n$ keys been input before, according to $P(n)$ we've assumed, set $A$ now contains the $m$ smallest keys among all the $n$ keys that were input before. (This is the induction hypothesis.) When the last key was input, there are two cases.
First case, the last key does not belong to the $m$ smallest element, so $A$ should not be changed. According to our algorithm, since the root of $A$ is the biggest of the $m$ smallest elements, if the last key is bigger than the root (that is the last key does not belong to the $m$ smallest elements), nothing will change in $A$. Hence A still contains the $m$ smallest keys among all the $n+1$ keys that were input before.
Second case, the last key belongs to the $m$ smallest elements, so the biggest element in $A$ will be abandoned and the last key will be an element of $A$. According to our algorithm, since the root of $A$ is the biggest of the $m$ smallest elements, if the last key is smaller than the maximum (that is the last key belongs to the $m$ smallest element), the root will be extracted, and the last key will be inserted into $A$ by first replacing the old root. Hence $A$ now contains the $m$ smallest keys among all the $n+1$ keys that were input.
Since after $m$ keys of input, the set $A$ will always contain the $m$ smallest keys among all the keys that were input, hence whenever a print occur, according to the algorithm we will print all the element in $A$, that is we will print the $m$ smallest keys among all the keys that were input before. Hence our algorithm is correct.

■