

## Question 1.

Zijin Zhang wrote the solution to this question. Quan Xu read this solution to verify its clarity and correctness and correct some part of solution.

Consider the upper bound of the worse case running time of the function Nothing for an input array A of size  $n$  ( $n \geq 2$ ).

The line 1 and line 2 of the function are assignment statements which take constant time 1.

In worse case, the if condition at line 5 will never be satisfied, so the return statement will not run. However, the if condition at line 6 will be satisfied for the first time when  $i = 1$  and  $j = (n - 1)$  that will trigger the return statement and the function will be terminated. In this case, the outer loop at line 3 will at most iterate 1 time and the inner loop at line 4 will at most iterate  $(n - 1)$  times. The two statement at line 5 and line 6 take constant time 1. So the running time of line 3 - line 6 is at most  $1 \times (n - 1) \times 1 = n - 1$  and the line 7 will never be reached.

Therefore, the total running time  $T(n)$  is at most  $(n - 1) + 1 = n$  which makes  $T(n) \in \mathcal{O}(n)$ .

*Proof.* WTS:  $\exists c_0, n_0 \in \mathbb{R}^+$  s.t.  $\forall n \in \mathbb{N}, n \geq n_0 \implies T(n) \leq c_0 n$

Take  $n_0 = 2, c_0 = 1$

Then we have  $T(n) = n = 1 \times n = c_0 n \leq c_0 n$

Therefore,  $T(n) \in \mathcal{O}(n)$  is proved.  $\square$

Let A be an array of length  $n$  ( $n \geq 2$ ) that contains  $[0, 1, 0, 1, \dots]$ , consider the lower bound of the worse case running time of the function Nothing for input A.

The line 1 and line 2 of the function are assignment statements which take constant time 1.

Since array A is  $[0, 1, 0, 1, \dots]$ , the if condition at line 5 will never be satisfied, so the return statement will not run. However, the if condition at line 6 will be satisfied for the first time when  $i = 1$  and  $j = (n - 1)$  that will trigger the return statement and the function will be terminated. In this case, the outer loop at line 3 will at least iterate 1 time and the inner loop at line 4 will at least iterate  $(n - 1)$  times. The two statement at line 5 and line 6 take constant time 1. So the running time of line 3 - line 6 is at least  $1 \times (n - 1) \times 1 = n - 1$  and the line 7 will never be reached.

Therefore, the total running time  $T(n)$  is at least  $(n - 1) + 1 = n$  which makes  $T(n) \in \Omega(n)$ .

*Proof.* WTS:  $\exists c_1, n_1 \in \mathbb{R}^+$  s.t.  $\forall n \in \mathbb{N}, n \geq n_1 \implies T(n) \geq c_1 n$

Take  $n_1 = 2, c_1 = 1$

Then we have  $T(n) = n = 1 \times n = c_1 n \geq c_1 n$

Therefore,  $T(n) \in \Omega(n)$  is proved.  $\square$

Since we have proved that  $T(n) \in \Omega(n)$  and  $T(n) \in \Omega(n)$ , we can conclude that  $T(n) \in \Theta(n)$ .

## Question 2.

Quan Xu wrote the solution to this question. Zijin Zhang read this solution to verify its clarity, correctness and correct some part of solution.

---

```
1 H = Max_Heap() # initialize a max heap, H
2 algorithm(x, m):
3     if x is not print operation:
4         process_key(x, m, H)
5     else: # when x is a print operation
6         print_(H)
7
8 print_(H): # this could be called when a print operation occurs.
9     for key in H:
10        print(key) #print every key in H.
11
12 processKey(key, m, H):
13     if H.size = m and max(H) > key:
14         extract_max(H)
15         insert(H, key)
```

---

```

16     else if H.size < m:
17         insert(H, key)

```

---

Note:

- 1)  $x$  is a distinct integer or a *print* operation (if  $x$  is *print* operation, we print the  $m$  smallest keys among all keys were input before *print* operation; if  $x$  is an integer key, we process the key).
- 2)  $m$  is an integer and  $m \geq 1$ ,  $m$  is the number of smallest key that will be printed.
- 3)  $H$  is a Max Heap which has  $H.size \leq m$ .
- 4) We assume that at least  $m$  keys are input before the first *print* operation occurs and  $m$  does not change during an execution of the algorithm.

a) The idea is to maintain a Max Heap that contains  $m$  keys, specifically, the  $m$  smallest keys among all keys were input before *print* operation.

b) First build a Max Heap,  $H$ , on line 1. The function *algorithm*( $x, m$ ) take 2 inputs  $x$  and  $m$  (we described above).

In the *if statement* on line 4, when  $x$  is not *print* operation, we do *processKey()* to process the key. In function *processKey*( $x, m, H$ ), it takes 3 inputs.  $x$  is the input integer key,  $m$  is the number of smallest key that will be printed,  $H$  is the Max heap. On line 13, when  $H.size = m$  and the largest key in  $H > x$ , the key, we firstly *extract\_max*( $H$ ) to remove the largest key in  $H$ , and  $H.size$  decrease by 1. And then *insert*( $H, key$ ) to insert the new smaller key to  $H$ , and  $H.size$  increase by 1. So  $H.size$  remain unchanged and equal to  $m$  after line 15.  $H$  contains  $m$  smallest keys among all keys were input before *print* operation. On line 16-17, when  $H.size < m$ , we simply insert  $x$ , the key, to  $H$ .

In the *else statement* on line 5, since Note 4), the *else statement* will not run until  $k=m$  iterations of calling *algorithm()*. And when  $x$  is *print* operation, it will do *print\_(H)* to print all the keys that were stored in  $H$ .

c) Firstly, we consider the worst-case of function *processKey()* on line 12-17. On line 13, *max*( $H$ ) takes  $\mathcal{O}(1)$  times in worst-case. So line 13 and line 17 take constant time. From the lecture we know that, *extract\_max*( $H$ ) takes  $\mathcal{O}(\log(H.size))$  times and *insert*( $H, key$ ) takes  $\mathcal{O}(\log(H.size))$  times in worst-case. Considering, if statement on line 13-15,  $H.size = m$ . So in this case, *extract\_max*( $H$ ) takes  $\mathcal{O}(\log(m))$  times and *insert*( $H, key$ ) takes  $\mathcal{O}(\log(m))$  times, since  $H.size = m$ . So it takes  $\log(m) + \log(m) + 1 = 2\log(m) + 1$  times. Considering the *else statement* on line 16-18,  $H.size < m$ , so it at most takes  $\log(m - 1) + 1$  times. So *processKey()* at most takes  $2\log(m) + 1$ . So  $2\log(m) + 1 \in \mathcal{O}(\log(m))$ , since it is clear that there is a constant  $c \geq 3$ , such that for all  $m \geq 10$ : for every input size  $m$ , executing the procedure *processKey()* takes at most  $c \log m$  iterations.

So the runtime of processing each input key  $\in \mathcal{O}(\log m)$ .

Secondly, we will describe the worst-case runtime of function *print\_(H)* on line 8-10. Line 10 take constant time to print each key. The loop on line 9 will iterate  $H.size$  iteration, since it loop all the keys in  $H$ . And  $H.size = m$  (by procedure of *processKey()* and Note 4)), thus the for loop iterate  $m$  times. So the total runtime is  $m + 1$ . So  $m + 1 \in \mathcal{O}(m)$ , since it is clear that there is a constant  $c > 1$  such that for all  $m \geq 1$  : for every input  $H$  of size  $m$ , executing the procedure *print\_(H)* takes at most  $cm$  times.

So the runtime of performing each print operation  $\in \mathcal{O}(m)$ .

d) Since we will iterate *algorithm()* again and again. Let the number of iteration of *algorithm()* be  $k$  where  $k \in \mathbb{Z}$  and  $k \geq 0$ .

$\forall k \in \mathbb{Z}, k \geq 0$ , define  $P(k)$ : after  $k$  iteration of *algorithm*( $x, m$ ) where  $x$  is integer key or *print* operation,  $m \in \mathbb{Z}$  and  $m \geq 1$ . If  $x$  is a *print* operation, the algorithm must print (in any order) the  $m$  smallest keys among all the keys that were input before the print. If  $x$  is an integer key, we process the key. And  $H$  stores  $m$  smallest keys among all the keys that were input before the print.

Let  $k \in \mathbb{Z}, m \in \mathbb{Z}$  and  $m \geq 1, k \geq 0$ . Let  $x$  be an integer key or a *print* operation and  $H$  be a Max Heap  
Base case:

$k \leq m$ :

Since  $k \leq m$ , according to Note 4),  $x$  will not be *print* operation when  $k \leq m$ . In our procedure, all the input  $x$  (when  $k \leq m$ ), will be stored in the Max Heap,  $H$ . Since  $k \leq m$ , so the number of inputs that were input is  $\leq m$ , So these keys were stored in  $H$  are the  $m$  smallest key among all keys that were input before  $k > m$  iterations. So  $P(k)$  when  $k \leq m$  is true.

$k = m + 1$ :

Before the  $k = m + 1$  iteration, there are  $m$  keys stored in  $H$  by procedure. Also by Note 4), the input  $x$  on  $m+1$  iteration could be either an integer key or a print operation.

Considering,  $x$  is an integer key: by procedure on line 3-4, we do  $processKey(x, m, H)$ . Since  $k = m + 1$ , by the code of  $processKey(x, m, H)$  we inserted  $m$  keys in  $H$  before the  $m+1$  iteration. And the  $m$  keys in  $H$  are the  $m$  smallest keys among all  $m$  keys that were input before  $m+1$  iteration. So it would run the if statement on line 13. If the new input  $x$ , the integer key, is smaller than the  $\max(H)$ , then we extract the old largest key in  $H$  and replace it by the new smaller input integer key. So the  $H$  will still contain the least  $m$  keys.

Considering,  $x$  is print operation: Before  $m+1$  iteration, there are  $m$  keys store in  $H$ . And the  $m$  keys in  $H$  are the  $m$  smallest keys among all  $m$  keys that were input before  $m+1$  iteration. So when  $x$  is print operation, we print all the keys in  $H$ , and we will get all the  $m$  smallest keys as wanted. So  $P(m+1)$  when is true.

Inductive step:

Let  $k \in \mathbb{Z}, m \in \mathbb{Z}$  and  $k \geq 0, m \geq 1$ . Let  $x$  be an integer key or a print operation and  $H$  be a Max Heap.

I.H.: Assume  $k > m + 1$ , WTS:  $P(k+1)$  holds.

On  $k+1$  iteration, the input  $x$  could be either an integer key or a print operation.

Case input  $x$  is an integer key: It runs line 3-4. By I.H., the  $H$  are store the  $m$  smallest keys, and  $H.size = m$ . By procedure, if input  $x < \max(H)$ , then we extract the old largest key, the  $\max(H)$ , in  $H$  and replace it by the new smaller input integer key, the input  $x$ . So the  $H$  will still contain the  $m$  smallest keys and  $H.size$  remains  $m$ .

Case input  $x$  is a print operation: it runs line 5-6, by I.H. and procedure,  $H$  stores the  $m$  smallest keys. So when the print operation occurs, it prints every key in  $H$ , by procedure. So the  $m$  smallest key will be printed as wanted. So  $P(k+1)$  holds.

Therefore,  $\forall k \in \mathbb{Z}, k \geq 0, P(k)$  holds. So our algorithm is correct.