Computer Science CSC263H
St. George Campus

January 10, 2019
University of Toronto

Homework Assignment #1

Due: January 17, 2019, by 5:30 pm

- You must submit your assignment as a PDF file, named **a1.pdf**, of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at:

  `https://markus.teach.cs.toronto.edu/csc263-2019-01`

  To work with one or two partners, you and your partner(s) must form a group on MarkUs.

- The **a1.pdf** PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTeX typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can use other typesetting systems if you prefer, but handwritten documents are not accepted.

- If this assignment is submitted by a group of two or three students, the **a1.pdf** PDF file that you submit should contain for each assignment question:

  1. The name(s) of the student(s) who *wrote* the solution to this question, and
  2. The name(s) of the student(s) who *read* this solution to verify its clarity and correctness.

- By virtue of submitting this assignment you (and your partners, if you have any) acknowledge that you are aware of the homework collaboration policy that is stated in the csc263 course web page: `http://www.cs.toronto.edu/~sam/teaching/263/#HomeworkCollaboration`.

- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.

- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.

- Your **a1.pdf** submission should be no more than 2.5 pages long in a 10pt font.

$\underset{①}{\forall\, n\,,\ \text{at most}}\ \underset{②}{f(n) \in O(n)} \Rightarrow \underset{③}{\theta(n)}$

$\underset{①}{\exists\, n\,.\ \text{at most}}\ \underset{②}{2n \in \Omega(n)}$

fake

Question 1. (1 marks)

In the following procedure, the input is an array $A[1..n]$ of n arbitrary integers, and "**return**" means the procedure stops ***immediately*** (breaking out of the loop of the procedure). Note that the indices of array A start at 1.

```
NOTHING(A)
1  A[1] = −1
2  n = A.size
3  for i = 1 to n                    // i = 1, 2, ..., n (including n)
4      for j = 1 to n − 1            // j = 1, 2, ..., n − 1 (including n − 1)
5          if A[j] ≠ 1 − A[j + 1] then return
6          if i + j > n − 1 then return
7  return
```

Assume that each assignment, comparison and arithmetic operation takes ***constant time***.
Let $T(n)$ be the worst-case time complexity of calling NOTHING(A) on an array A of size $n \geq 2$.

Give a function $f(n)$ such that $T(n)$ is $\Theta(f(n))$.

Justify your answer by explaining why it is $O(f(n))$, and why it is $\Omega(f(n))$. Any answer without a sound and clear justification may receive no credit.

Question 2. (1 marks)

Design an efficient algorithm for the following problem. The algorithm is given an integer $m \geq 1$, and then a (possibly infinite) sequence of distinct integer keys are input to the algorithm, ***one at a time***. A *print* operation can occur at any point between keys in the input sequence. When a *print* occurs, the algorithm must print (in any order) the m smallest keys among all the keys that were input before the *print*.

For example, suppose $m = 3$, and the keys and *print* operations occur in the following order:

$$20, 15, 31, 6, 13, 24, \textit{print}, 10, 17, 9, 16, 5, 11, \textit{print}, 14, \ldots$$

Then the first *print* should print 15, 6, 13 (in any order), and the second *print* should print 6, 9, 5 (in any order).

Assume that at least m keys are input before the first *print* occurs and that m does not change during an execution of the algorithm.

Describe a ***simple*** algorithm that solves the above problem with the following worst-case time complexity:

- $O(\log m)$ to process each input key.
- $O(m)$ to perform each *print* operation.

Your algorithm must use a data structure that we learned in class.

- State which data structure you are using and describe the items that it contains.
- Explain your algorithm for each operation ***clearly*** and ***concisely***, in English.
- Explain why your algorithm achieves the required worst-case time complexity described above.
- Prove that your algorithm is correct (Hint: use induction. What is your induction hypothesis?)

```
1  H = max_heap()          # H is a Max Heap
2
3  algorithm ( x , m )
4      if x is not print operation :
5          process_k ( x , m , H )
6      else :
7          print (H)
8
9  print ( H ) :
10         result = ""
11         for key in H :
12             result += str(key) + ","        ] — O(m)
13         return result
14
15 process_key ( key , m , H ) :
16         if H.heapsize = m and max(H) > key :
17             extract_max(H)
18             insert ( H , key )
19         elif H.heapsize < m :
20             insert (H , key)
```

- State which data structure you are using and describe the items that it contains.
- Explain your algorithm for each operation ***clearly*** and ***concisely***, in English.
- Explain why your algorithm achieves the required worst-case time complexity described above.
- Prove that your algorithm is correct (Hint: use induction. What is your induction hypothesis?)

Note:

① x is an integer or a print operation. (if x is the print operation, we print the m smallest keys from the sequence of distinct integer keys; if not print operation, then we do process_key() to process the key).

② m is an integer that >= 1, m is the number of smallest keys among all keys that input before print that will be printed.

③ H is a Max Heap, which has H.heapsize <= m.

And we Assume that at least m keys are input before the first *print* occurs and that m does not change during an execution of the algorithm.

1) The idea is to maintain a Max Heap that contains m keys, specifically, the m smallest keys from the sequence of distinct integer keys.

2) First built a Max Heap, H. (on line 1). The algorithm function takes 2 input x and m (we talked the inputs in details above).

if statement: On line 4, when x is not the print operation, we do process_key() for processing the key to the max Heap, H. In function process_key() it take 3 inputs (an integer key that we will process it, a integer m (same as the m in algorithm()), and a Max Heap, H). On line 16. when the H.heapsize = m and the largest element in H is larger than the key, we first extract_Max(H) to remove the largest element in H (line 17), so the H.heapsize decrease by 1. On line 18. we insert the key into H, so H.heapsize increase by 1. After iterating the line 18, H.heapsize = m. And H contains m smallest keys among all keys that input before print. On line 19-20, if H.heapsize < m, the we simply insert the key into H.

else statement: Since at least m keys are input before the first print occurs, so the else statement on line 6 will not be iterated until m iterations of calling algorithm(). When x is the print operation, it will do print() to print the m smallest keys among all keys that input before print. for print() function on line 9-13, it print all the element in H.

3) first, We will describe the worsecase runtime of function, print() on line 9-13: line 10 and line 13 take constant time. The for loop on line 11-12 take m iterations, since it loops every element of H, and there are m elements in H. So total runtime is m + 1 iteration for the worse case. so $m+1 \in O(m)$, 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, excuting the precedure print(H) takes at most cm times.

so the runtime of performing each print operation is $\in O(m)$

Second, we will describe the worsecase runtime of function process_key() on line 15-20. line 16 and line 19 each take constant time. From the lecture we know that extract_max(H) and insert(H, key) both take $O(\log m)$ in Worsecase Runtime where m = H.heapsize. so the if statement 16-18 take at most $\log m + \log m + 1 = 2\log m + 1$ iteration. And the else statement at most take $\log m + 1$ iterations. So the function process_key() at most take $2\log m + 1$ iteration. So $2\log m + 1 \in O(\log m)$, since it is clear that there is a constant $C >= 3$, such that for all $m >= 10$: for every input H of size m, excuting the precedure process_key() takes at most $c\log m$ iteration.

So the runtime of processy each input key is $\in O(\log m)$

4) Since we will iterate algorithm() again and again.
So, Assume the number of iteration of algorithm() is K where $k \in \mathbb{Z}^+$.
$\forall k \in \mathbb{Z}^+$, define P(k) :
x ∈ distinct integer or print operation, $m \in \mathbb{Z} \wedge m >= 1$
after k iteration of calling algorith(x, m), when a print occurs, the algorithm must print (in any order) the m smallest keys among all keys that were input before print. And H store m smallest key after k=m.

Assumption *: from the question, we assume that at least m keys are input before the first print occurs and that m does not change during an execution of the algorithm().

let $k \in \mathbb{Z}^+$, $m \in \mathbb{Z} \wedge m >= 1$,
let X be an integer or print operation.

Base cases: $k <= m$: since $k <= m$, so by assumption *, x will not be print operation when $k <= m$. In our procedure, all the input x (on $k <= m$ iterations of algorithm()) will be store in a Max Heap, H. Since $k <= m$, there is no print operation as input of algorithm(). Since $k <= m$, so number of input $<= m$. So those keys were stored in H are the m smallest keys among all keys that were input before $k > m$ iteration. So the P(k) when $k <= m$ is vacuously true.

$k = m + 1$: before $k = m+1$ iteration, there are m keys stored in the Max Heap H by our procedure. Also, by assumption *, on $k > m$ iterations (also $k >= m+1$ iterations), the input x could be either a distinct integer key or a print operation.

for x is a distinct integer key: by our procedure on line 4-5, there are m keys stored in H, and these m keys in H are the m smallest keys among all keys were input before $k = m+1$. so H.heapsize = m, so it will run line 16-18. If $x <= Max(H)$, the x_{m+1} is one of the new m smallest key. So we first remove the Max(H), the H.heapsize decrease by 1 and equal to m-1, then we insert

the input X_{m+1} to H, H.heapsize increase by 1 and equal to m. Right now, the m keys stored in H are still the m smallest key, since we just replaced the old largest key with a new smaller key.

for x is print operation :
before $k = m+1$ iteration, there are m keys stored in H. Those keys are the m smallest keys among all keys that were input before $k=m+1$ 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.

Inductive step: let $k \in \mathbb{Z}^+$ and Assume $k > m+1$,
IH: Assume $P(k)$ holds. WTS $P(k+1)$ holds.
On $k+1$ iteration, the input x could be either an distinct integer or a print operation.
case input x is a distinct integer:
it runs line 4-5, by our IH, the Max Heap H are stored the m smallest keys, and H.heapsize = m. By procedure, if input $x >= Max(H)$, we extract the largest key in H and replace it by inserting a smaller key. So the H.heapsize still equal to m, and H still store the m smallest keys.
case input x is print operation:
it run line 6-7. by IH and our procedure, H stores the m smallest element, so when the print operation occurces, it print every keys in H. So the m smallest key will be printed as wanted.

```
def insert ( ArrayList A , int value , int m ) {
    A.add ( value ) ;
    int i = 0 ;
    while ( i <= 1 ) {
        if ( A[i] >= A[⌊i/2⌋] ) {
            A[i] , A[⌊i/2⌋] = A[⌊i/2⌋] , A[i] ; }
        }
}
def getSmallestM ( ArrayList A , int m ) {
    """ pre_condition: len(A) >= m """
    int i = 0 ;
    List<> leastM = new ArrayList<>() ;
    for ( int i = 0 , i < m , i++ ) {
        leastM.add ( A[i] ) ;
    }
    for ( int ind = 0 , ind < A.length() , ind++ ) {
        if A[0] > leastM[0]
```

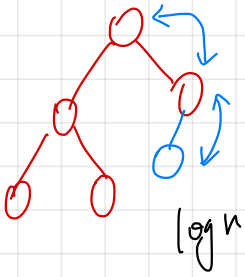
A B C D

[A , C , D , B]

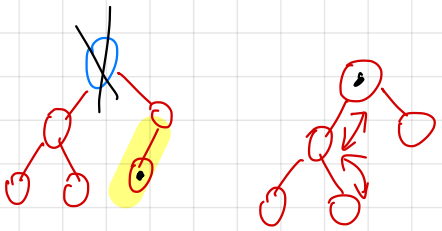
[[_ _ _] , [_ _ _ _] , [_ _ _ _] , . . . , [_ _ _]]

insert $\in O(\log m)$

smallest $\in O(m)$



$\log n$



$\log n$ (Total : $\log n + (\log n)$

$2 \log n \in O(\log n)$

```
def magic ()
    insert ( value ) → O(log m)
    extra_max ( ) ; → O(log n)
```

[***The questions below will not be corrected/graded. They are given here as interesting problems that use material that you learned in class.***]

Question 3. (0 marks) In class we studied *binary* heaps, i.e., heaps that store the elements in complete *binary* trees. This question is about *ternary* heaps, i.e., heaps that store the elements in complete *ternary* trees (where each node has at most *three* children, every level is full except for the bottom level, and all the nodes at the bottom level are as far to the left as possible). Here we focus on MAX heaps, where the priority of each node in the ternary tree is greater or equal to the priority of its children (if any).

a. Explain how to implement a ternary heap as an array A with an associated $Heapsize$ variable (assume that the first index of the array A is 1). Specifically, explain how to map each element of the tree into the array, and how to go from a node to its parent and to each of its children (if any).

b. Suppose that the heap contains n elements.
(1) What elements of array A represent internal nodes of the tree? Justify your answer.
(2) What is the height of the tree? Justify your answer.

c. Consider the following operations on a ternary heap represented as an array A.

- INSERT(A, key): Insert key into A.
- EXTRACT_MAX(A): Remove a key with highest priority from A.
- UPDATE(A, i, key), where $1 \leq i \leq A.Heapsize$: Change the priority of $A[i]$ to key and restore the heap ordering property.
- REMOVE(A, i), where $1 \leq i \leq A.Heapsize$: Delete $A[i]$ from the heap.

For each one of these four operations, describe an efficient algorithm to implement the operation, and give the worst-case time complexity of your algorithm for a heap of size n. Describe your algorithm using high-level pseudo-code similar to that used in your textbook, with clear explanations in English. Express the worst-case time complexity of your algorithm in terms of Θ and justify your answer.

Question 4. (0 marks) Let A be an array containing n integers. Section 6.3 of our textbook (CLRS) describes a procedure, called BUILD-MAX-HEAP(A), that transforms array A into a max-heap in $O(n)$ time. That procedure works "bottom-up", using MAX-HEAPIFY repeatedly.
Another way of transforming A into a max-heap is to insert the elements of A into the heap one at a time. Specifically, the algorithm is as follows:

```
BUILD-BY-INSERTS(A)
    A.heapsize := 1
    for i := 2..n do
        MAX-HEAP-INSERT(A, A[i])
```

a. Give an example of an input array A for which the two procedures BUILD-MAX-HEAP and BUILD-BY-INSERTS produce different outputs. Keep your example as small as possible.

b. Let $T(n)$ be the worst-case time complexity of BUILD-BY-INSERTS for an input array A of size n. Prove that $T(n)$ is $\Theta(n \log n)$. (Recall that the worst-case time complexity of BUILD-MAX-HEAP is $O(n)$, and therefore BUILD-MAX-HEAP is more efficient than BUILD-BY-INSERTS.)

Question 5. (0 marks)

Let I_n be the set of n integers $\{1, 2, \ldots, n\}$ where n is some power of 2.

Note that we can easily use an n-bit vector (i.e., an array of n bits) $B[1..n]$ to maintain a subset S of I_n and perform the following three operations (where j is any integer in I_n) in constant time each:

INSERT(j): insert integer j into S.

DELETE(j): delete integer j from S.

MEMBER(j): return **true** if $j \in S$, otherwise return **false**.

Describe a data structure that supports all the above operations ***and*** also the following operation

MAXIMUM: return the greatest integer in S

such that:

- The worst-case time complexity of operations INSERT(j), DELETE(j), and MAXIMUM is $O(\log n)$ each. The worst-case time complexity of MEMBER(j) is $O(1)$.
- The data structure uses only $O(n)$ bits of storage.

  Note that the binary representation of an integer i where $1 \leq i \leq n$ takes $\Theta(\log n)$ bits. Assume that any pointer also takes $\Theta(\log n)$ bits.

A solution that does not meet ***all*** the above requirements may not get any credit.

HINT: Complete binary trees can be implemented without using pointers.

a. Describe your data structure by drawing it for $n = 8$ and $S = \{1, 2, 6\}$, and by explaining this drawing briefly and clearly. Your drawing must be very clear.

b. Explain how the operations INSERT(j), DELETE(j), and MAXIMUM are executed, and why they take $O(\log n)$ time in the worst-case. Be brief and precise.

c. Explain how the operation MEMBER(j) is executed, and why it takes $O(1)$ time in the worst-case. Be brief and precise.