First Name: Zhicheng

Last Name: Yan

Student ID: 1002643142

First Name: Zhihong

Last Name: Wang

Student ID: 1002095207

First Name: Jialiang

Last Name: Yi

Student ID: 1002286929

We declare that this assignment is solely our own work, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters.

This submission has been prepared using LaTeX.

Writer: Zhicheng Yan

Reader: Jialiang Yi, Zhihong Wang

Question 1
In the following procedure, the input is an array $A[1..n]$ of arbitrary integers, $A.size$ is a variable containing the size n of the array A (assume that A contains at least $n \geq 2$ elements), and "return" means return from the procedure call.

```
0. Procedure nothing(A) // A is indexed starting at 1 (not 0)
1.    A[1] := 1
2.    n := A.size
3.    for i = 2 to n do
4.        for j = 1 to i-1 do A[j] := A[j] + 1
5.        if A[i] is not equal to A[i-1] then return
6.    return
```

In Q1 part a and b, we will only discuss about the worst case, which means line 5 will never be executed.
a. State whether $T(n)$ is $O(n^2)$ and justify your answer.

Solution:
$T(n)$ is $O(n^2)$
To prove that, we need to show $T(n) = cn^2$ for constant $c > 0$.
By line 3, the outside loop iterates $n - 1$ times $(i = 2, 3, 4, ..., n)$, which is less than n times.
By line 4, for each iteration of the outside loop, it iterates $i - 1$ times, since $i \leq n$, then it iterates less than n times.
Then, the worst case complexity is always less than n^2.
So, there exists $c > 0$ such that $T(n) \leq cn^2$.
Therefore, $T(n)$ is $O(n^2)$.

b. State whether $T(n)$ is $\Omega(n^2)$ and justify your answer.
Solution:
$T(n)$ is $\Omega(n^2)$
To show that,
There is a loop which is inside another loop
In the 1st iteration of the outside loop($i = 2$), the inside loop iterates 1

time(j=1).
In the 2nd iteration of the outside loop($i = 3$), the inside loop iterates 2 times(j=1,2).
In the 3rd iteration of the outside loop($i = 4$), the inside loop iterates 3 times(j=1,2,3).
In the 4th iteration of the outside loop($i = 5$), the inside loop iterates 4 times(j=1,2,3,4).
$\vdots$
$\vdots$
In the (n-2)th iteration of the outside loop($i = n - 1$), the inside loop iterates $n - 2$ times (j=1,2,3,...n-3,n-2).
In the (n-1)th iteration of the outside loop($i = n$), the inside loop iterates $n - 1$ times (j=1,2,3,...n-2,n-1).
According to that, we can conclude that
$T(n) = \Sigma_{i=2}^{n}(i-1) = 1 + 2 + 3 + ... + (n-1) = \frac{n(n-1)}{2}$.
Since $n \geq 2$
Then $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \frac{n^2}{4} + \frac{n^2}{4} - \frac{n}{2} \geq \frac{n^2}{4} + \frac{2n}{4} - \frac{n}{2} = \frac{n^2}{4} + \frac{n}{2} - \frac{n}{2} = \frac{n^2}{4}$
Then $T(n) \geq \frac{n^2}{4}$
Then $c = \frac{1}{4} > 0$ such that $T(n) \geq cn^2$
Therefore, $T(n)$ is $\Omega(n^2)$

Writer: Jialiang Yi

Reader: Zhihong Wang, Zhicheng Yan

Question 2
We want an efficient algorithm for the following algorithm for the following problem. the algorithm's input consists of:

1. k lists A_1, A_2, ..., A_k, each one consisting of n/k integers sorted in increasing order (for simplicity) assume that n is a multiple of k and all n integers are distinct), and

2. an integer m such that $1 \leq m \leq n$.

The algorithm must output the m smallest elements from A_1, A_2, ..., A_k, *in increasing sorted order.* Note that for the special case where $m = n$, it outputs *all* the elements of A_1, A_2, ..., A_k, in sorted order.

a. Describe a simple algorithm that solves the above problem. Your algorithm *must* use a data structure that we learned in class, and its worst-case time complexity must be $O \log k + k$. *Describe the algorithm, and explain why it works, clearly and precisely in plain English.*
Solution:
The algorithm can be develop base on a min-heap. We construct a min-heap of size k with 1st elements of lists A_1 to A_k, k elements in total using the $build - heap$ method which maintain heap property via $MinHeapify$. We also initiate an output list of size m, say N. According to the properties of min-heap, k elements goes into a complete binary tree with k nodes such that priority of each node is less than or equal to the priority of its children. Customizing the node, we add a new variable 'source' into the node such that we can know which list the node is inserted from.

From the properties of min-heap, we can observed that the root is the smallest element of the heap. For each value of m, perform the operation EXTRACT_MIN from heap library. Since A_1 to A_k are sorted list in increasing order, simply append the extracted node into list N then insert the next node from the source list of the extracted node into the heap (ex: node A, use $A.source$ to get source list of A). If the extracted node is the last element of its source list, in another word, the source list is now empty, the

algorithm will skip the step of inserting new node. Instead, we will start a new iteration by jumping straight to perform another EXTRACT_MIN. This is the steps of one complete iteration.
Repeat the steps m times to attain m smallest elements from A_1 to A_k. At the end, m smallest elements are stored in the list N.

Proof. According to the property of min heap, the operation EXTRACT_MIN always extracts the minimum node from the heap, which is the root in this case. Consequently, the extracted node must be the smallest element from the remaining elements of the lists. We do induction on the number of smallest elements.

Base case: $m = 1$. Since the algorithm inserts all 1st elements of lists A_1 to A_k into the min-heap of size k and all lists are sorted in increasing order, it is easy to see that these k elements are smallest elements from these k lists. By results of EXTRACT_MIN, we get the smallest node of the heap, which is also the smallest element within k lists. Therefore, $m = 1$ holds.

Induction step: suppose it is true for $m = j$ and $m < n$, we want to prove it for $j + 1$. We know that the after j iterations, j_{th} smallest elements are now extracted from the k lists and stored in the output list, say N. That means all remaining elements in the k lists are no smaller than the elements in output list N. Starting the $(j+1)_{th}$ iteration, the operation EXTRACT_MIN extracts the minimum node from the heap, which is the smallest within the remaining elements. Now, we have $j + 1$ smallest elements in the output list N. Therefore, $m = j + 1$ holds, our algorithm is functional.

b. Explain why your algorithm's worst-case time complexity is $O(m \log k + k)$.
Solution:
First step of our algorithm is constructing a min-heap of size k. To build a min-heap, we are using $MinHeapify$ to maintain the heap property. $MinHeapify$ is recursively called at most h times, where h is the height of the subtree starting at i. Each call to $MinHeapify$ takes c steps, for some constant c. Therefore, for all inputs, at most $c * h$ steps are needed. So the worst case time complexity is $O(h)$ where h is the subtree with root at i, which is equivalent to $O(logk)$. At the bottommost level, where the leafs are, there are 2^h nodes, but we do not call heapify on these, so the runtime

is 0. At the next to level there are 2^{h-1} nodes, and each at worst case will move down by 1 level. Clearly, not all heapify operations are $O(logk)$, it depends on the height currently working on.

Actual cost of heapify = $\sum_{h=0}^{\lfloor logk \rfloor} \frac{k}{2^{h+1}} O(h)$
$= O(k \sum_{h=0}^{\lfloor logk \rfloor} \frac{h}{2^{h+1}})$
$= O(k/2 \sum_{h=0}^{\lfloor logk \rfloor} \frac{h}{2^h})$
$= O(k \sum_{h=0}^{\infty} \frac{h}{2^h})$
$= O(2k)$
$= O(k)$

Therefore, the worst case time complexity of building a heap size of k is $O(k)$.

Next is to extract the minimum value of the heap. The EXTRACT_MIN will first extract the root, replace the root with the bottom rightmost node, then sorted the heap to maintain CBT structure and min-heap properties. By definition, in a heap of size k, the height of the heap is $\log k$, thus every EXTRACT_MIN takes at most $c * \log k$ steps, the worst case time complexity is in $O(\log k)$. Then the algorithm will append the extracted node into the output list which runtime is in linear of $O(1)$.

In the case where the source list of the extracted node is non-empty, final step of one single iteration is to insert a new node into the heap. In the process of bubbling up, there are maximum $\log k$ comparison. From this, every insert takes at most $c * \log k$ steps, the worst case time complexity is in $O(\log k)$.

There are maximum one EXTRACT_MIN operation and one insertion operation per iteration. Therefore, finding m smallest elements, the worst case time complexity of extraction is $O(m \log k)$ and the worst case time complexity of insertion is $O(m \log k)$. Including the runtime of constructing the heap, which gives us $O(2m \log k + k)$. Therefore, the worst-case time complexity of the algorithm is $O(m \log k + k)$.

Writer: Zhihong Wang

Reader: Jialiang Yi, Zhicheng Yan

Question 3
This question is about the cost of successively inserting k elements into a binomial heap of size n.

a. Prove that a binomial heap with n elements has exactly $n - \alpha(n)$ edges, where $\alpha(n)$ is the number of 1's in the binary representation of n.

Solution:

Note: Assume a binary representation's index(i) begin with 0 (ie. For 10, 0's index is 0, 1's index is 1), and the size of a binary representation is equal to the max index plus 1. Those index(i) of 1s in a binary representation of size n show that for a binomial heap of size n contains binomial trees of B_i (ie. 1010 means this binomial heap has B_1 and B_3).

Lemma 19.1 (from text book binomialheaps.pdf, p457)

For the binomial tree B_k,
1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly $\binom{k}{j}$ nodes at depth i for $i = 0, 1, ..., k$, and
4. the root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, ..., 0$, child i is the root of a subtree B_i .

Let a binomial heap of size n named H.

For a normal tree of size n, it has n edges in total.

H consists of at most $\lfloor logn \rfloor + 1$ binomial trees. To see why, observe that the binary representation of n has $\lfloor logn \rfloor + 1$ bits, say $\langle b_{\lfloor logn \rfloor}, b_{\lfloor logn \rfloor - 1}, ..., b_0 \rangle$, so that $n = \sum_{i=0}^{\lfloor logn \rfloor} b_i 2^i$. By property 1 of Lemma 19.1, therefore, binomial

tree B_i appears in H if and only if bit $b_i = 1$. Thus, binomial heap H contains at most $\lfloor logn \rfloor + 1$ binomial trees.
This proof is from text book binomialheaps.pdf, p459.

H contains at most $\lfloor logn \rfloor + 1$ binomial trees, which also means $\lfloor lgn \rfloor + 1$ binomial trees will merge $logn$ times, will increase $logn$ number of edges.

So, $logn$ also represents the number of binomial trees, which is the number of 1's in the binary representation of n.

When H has only one binomial tree, the exactly edges of H is $n - 1$ because every nodes in a tree has one edge to connect to its parent, except the root.

So, we can easily get the exactly edges of H is $n-$ number of binomial trees.

Therefore, the exactly edges of a binomial heap with n elements is $n - \alpha(n)$ because $\alpha(n)$ represents the number of binomial trees.

b. Consider the worst-case total cost of successively inserting k new elements into a binomial heap H of size $|H| = n$. In this question, we measure the worst-case cost of inserting a new element into H as the maximum number of pairwise comparisons between the keys of the binomial heap that is required to do this insertion. It is clear that for $k = 1$ (i.e., inserting one element) the worst-case cost is $O(\log n)$. Show that when $k > \log n$, the **average** cost of an insertion, i.e. the worst-case total cost of the k successive insertions divided by k, is bounded above by constant.

Solution:

Lemma: every comparison will create an edge.

Prove Lemma:

Because a comparison is a merge, which will create a edge to connect two sides in this comparison.

Therefore every comparison will create an edge.

Because worst-case cost of inserting a new element into H as the maximum number of pairwise comparisons between the keys of the binomial heap that is required to do this insertion, we know that every insertions in this question will also create an edge just like the Lemma.

Before inserting k new elements, by Q3a, the exactly edges is $n - \alpha(n)$.

After inserting k new elements, by Q3a and Lemma, the exactly edges will be $n + k - \alpha(n + k)$.

So, by Lemma, k successive insertion times

$$\begin{aligned} &= \text{the new edges} \\ &= n + k - \alpha(n+k) - (n - \alpha(n)) \\ &= k - \alpha(n+k) + \alpha(n) \\ &\leq k + \alpha(n) \\ &\leq k + \lceil logn \rceil \\ &\leq k + k \;\#\; k > logn \end{aligned}$$

So, $k \leq 2k$

$1 \leq 2$

Therefore after divided by k, is obvious the average cost of an insertion is bounded above by constant.