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.

```
Algorithm 1: NOTHING($A$)
1 Function NOTHING($A$)
2     $A[1] := 1$
3     $n := A.size$
4     for $i = 2$ to $n$ do
5         for $j = 1$ to $i - 1$ do
6             $A[j] := A[j] + 1$
7             if $A[i] \neq A[i-1]$ then
8                 return
9         return
```

Let $T(n)$ be the worst-case time complexity of executing procedure NOTHING() on an array A of size $n \geq 2$. Assume that assignments, comparisons and arithmetic operations, like additions, take a constant amount of time each.

1. State whether $T(n)$ is $O(n^2)$ and justify your answer.

   Proof. Let $n > 0$ be arbitrary. The outer loop is executed at most n times. And because $i - 1 \leq n$, the inner loop is also executed at most n times. Also note that it takes constant time for each iteration of inner loop. Let $c, d \in \mathbb{R}$ be constant, we have running time of an arbitrary input x_n in

$$\begin{aligned} t(x_n) &\leq \sum_{i=1}^{n} \sum_{j=1}^{n} c + d \\ &\leq c\frac{n(n+1)}{2} + d \\ &= \mathcal{O}(n^2) \end{aligned}$$

   Since n is arbitrary, we proved that for all size n of A, and all inputs of size n, the running time is in $\mathcal{O}(n^2)$. Therefore, worst-time complexity is bounded from above $T(n) = \mathcal{O}(n^2)$ □

2. State whether $T(n)$ is $\Omega(n^2)$ and justify your answer.

   Proof. Let $n > 0$ be arbitrary. Consider an input of size n such that the if statement at line 7-8 never executes. Let $c, d \in \mathbb{R}$, we have running time of this particular input as x_ϵ

$$
\begin{aligned}
t(x_\epsilon) &= \sum_{i=2}^{n}\sum_{j=1}^{i-1} c + d \\
&= \sum_{i=2}^{n} c(i-1) + d \\
&= c\frac{(n+2)(n-1)}{2} - c(n-1) + d \\
&= \Omega(n^2)
\end{aligned}
$$

Since n is arbitrary. Then for any n, there exists an input x_ϵ such that its running time is $\Omega(n^2)$. Hence, the worst time complexity is bounded from below $T(n) = \Omega(n^2)$ □

Question 2

Design an efficient algorithm for the problem where the input is

1. k lists $A_1, A_2, \cdots, A_k$, each one consisting of $\frac{n}{k}$ integers sorted in increasing order, and
2. an integer m such that $1 \leq m \leq n$

The algorithm must output the m smallest elements from $A_1, \cdots, A_k$ in increasing sorted order.

1. Describe a simple algorithm solving the problem, using a data structure learnt in class, and with a worst-case time complexity of $\mathcal{O}(m \log k + k)$.

   Solution. The algorithm is as follows

   (a) Move the first element in each list $A_1, \cdots, A_k$ to form an unsorted array $A_{unsorted}$, while maintaining a pointer to the next element down the list for each element moved.

   (b) Construct a min-heap H from array $A_{unsorted}$

   (c) Create an empty array of size m called A_{sorted}. extract the root element (with mimimum value stored) of H, i.e. e_i where $1 \leq i \leq k$, and push to the end of A_{sorted}. Move the element that e_i's pointer is pointed to e_{new} in the list of arrays to the root of H while create a new pointer to the next element down the array.

   (d) Maintain heap property by using MAX-HEAPIFY on the root of H

   (e) Repeat previous step (c) m times to extract the m smallest elements

□

2. Explain why your algorithms worst-case time complexity is $O(m \log k + k)$.

   Consider the steps listed above

   (a) The move operation takes constant time for each element and there are k number of them moved. Therefore running time is in $\mathcal{O}(k)$
   (b) Construction of a min heap from an array of size k takes $\mathcal{O}(k)$
   (c) Creation of an empty array, move elements, as well as maintaining pointers takes $\mathcal{O}(1)$.
   (d) $MAX-HEAPIFY$ requires constant time for each level of H, therefore takes time proportional to height of H, $\log k$. Therefore requires $\mathcal{O}(\log k)$ time
   (e) m iteration where each iteration takes $\mathcal{O}(\log k)$ time requires $\mathcal{O}(m \log k)$ time

   Therefore overall the algorithm has worst case complexity of $\mathcal{O}(m \log k + k)$

Question 3

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

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

   Proof. Define predicate

   $$P(H_n): \textbf{ binomial heap of } n\textbf{, } H_n\textbf{, has exactly } n-\alpha(n) \textbf{ edges}$$

   Note that a binomial heap is a collection of binomial heaps with distinct size. Construction or insertion of new elements to the binomial heap involves the UNION(H, H') operation, which keeps the property of binomial heap by

   (a) link binomial trees B_i, B_j from H and H' if B_i and B_j are of same size
   (b) simply incoporate the binomial heap in the collection if all binomial trees from H and H' are of different size

   Here we use structural induction to prove that the UNION(H, H') operation preserves P. Binomial heap consisting of a single node, H_1 has 0 edges. Since binary representation of 1 is 1 with one 1, hence $1 - \alpha(1) = 0$. Therefore $P(H_1)$ holds. Suppose two arbitrary binomial heap H_u, H_v, consists of u and v nodes respectively. Assume $P(H_u)$ and $P(H_v)$ holds, i.e. H_u has $u - \alpha(u)$ edges and H_v has $v - \alpha(v)$ edges. Now we construct a new binomial heap H_n using UNION(H_u, H_v).

(a) If there exists binomial tree $B_i \in H_v$ and $B_i' \in H_u$ such that they are of same size i, B_i and B_i' are linked together to form B_k, which is of size $2i$. In general, for binomial tree B_i of size i, there exists $k \in \mathbb{N}$ such that $i = 2^k$; hence binary representation of i is composed of one leading 1 at position $k+1$ followed by k zeros. Hence $\alpha(i) = 1$ for arbitrary i. Since a binomial heap of size n, H_n, has distinctly sized binomial trees, say $B_{n_1}, B_{n_2}, \cdots, B_{n_q}$, where sizes follow $n_1 \neq n_2 \neq \cdots \neq n_q$ and $n = n_1 + n_2 + \cdots + n_q$, we have

$$\alpha(n) = \alpha(n_1) + \alpha(n_2) + \cdots + \alpha(n_q)$$

This is true because position of 1's are unique. With this property and the fact that B_i, B_i' and B_j are all binomial trees, it follows that $\alpha(i) = \alpha(k) = 1$. Because H_u and H_v are binomial heaps, then for some for some θ, ϕ, we have

$$\begin{aligned}
\alpha(u) + \alpha(v) &= \alpha(u_1) + \alpha(u_2) + \cdots + \alpha(i) + \cdots + \alpha(u_\theta) + \alpha(v_1) + \alpha(v_2) + \cdots + \alpha(i) + \cdots + \alpha(v_\phi) \\
&= \alpha(u_1) + \alpha(u_2) + \cdots + \alpha(u_\theta) + \alpha(v_1) + \alpha(v_2) + \cdots + \alpha(v_\phi) + \alpha(k) + 1 \\
&= \alpha(u+v) + 1 && \text{(two } B_i \text{ merged into a single } B_k\text{)}
\end{aligned}$$

Here we are making an assumption that there is no binomial tree of size k in both heaps. We can assume this without lose of generality. In part because the case where there exists binomial tree of size B_k can be considered as a case where rules for constructing the binomial heaps are applied recursively again. The number of edges of H_n is the sum of edges originally present in H_u and H_v in addition to the edge created by the linking. Then by induction hypothesis, H_n of size $u+v$ has

$$\begin{aligned}
u - \alpha(u) + v - \alpha(v) + 1 &= u + v - \alpha(u+v) - 1 + 1 \\
&= (u+v) - \alpha(u+v)
\end{aligned}$$

edges. Therefore $P(H_n)$ holds.

(b) If all binomial trees from H_u and H_v are of different size, then the position of 1's in binary representation of u and v are mutually exclusive. Hence $\alpha(u) + \alpha(v) = \alpha(u+v)$. The number of edges in H_n is just the sum of that in H_u and H_v. Then by induction hypothesis, H_n whis is a binomial heap of size n has

$$u - \alpha(u) + v - \alpha(v) = u + v - \alpha(u+v)$$

edges. Herefore $P(H_n)$ holds

By structural induction, $P(H_n)$ holds for all binomial heaps H_n. Hence any binomial heap with n element has $n - \alpha(n)$ edges

$\square$

2. 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. We are only interested in the number of pairwise comparisons between keys of the biomial tree during insertion. This procedure is only required during the linking step in situations where there exist binomial trees of same size. Each comparison procedure therefore corresponds to an addition of an edge to the binomial heap. Conversely, the number of edges in a binomial heap is proprotional to the number of comparison procedures. Therefore for a binomial heap of size n, H_n, the worst-case cost is therefore

   $$c(H_n) = n - \alpha(n)$$

   (results from previous question.) Similarly, a binomial heap of size $n + k$, H_{n+k} has worst-case cost of

   $$c(H_{n+k}) = n + k - \alpha(n + k)$$

   The average cost C of inserting k element to H_n is just the worst-cost difference between H_{n+k} and H_n, over k i.e.,

   $$C = \frac{c(H_{n+k}) - c(H_n)}{k} = \frac{k - \alpha(n + k) + \alpha(n)}{k}$$

   Note binary representation of a decimal m has $\lfloor \log(m) \rfloor + 1$ digits. In conjunction with $k > \log n$ we have

   $$C < \frac{k + \lfloor \log(m) \rfloor}{k} < \frac{k + k}{k} = 2$$

   Hence $C = \mathcal{O}(1)$. Therefore we proved that the average cost of an insertion is bounded above by constant. □

Question 4

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).

1. 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).

   Solution. Assume H_n is a ternary heap of size $Heapsize$. H_n is stored in an array of size $Heapsize$, A, with $A[1]$ storing root of H_n and children as well as parent accessed via some arithmetic manipulation of indices. Let i be arbitrary index then,

   (a) $Parent(i) = \lfloor \frac{i+1}{3} \rfloor$
   (b) $Left(i) = 3i - 1$
   (c) $Middle(i) = 3i$
   (d) $Right(i) = 3i + 1$

   □

2. Suppose that the heap contains n elements.

   (a) What elements of array A represent internal nodes of the tree? Justify your answer.

   Solution. Denote e_i to be the element in the array A at index i. Since heap has a tree structure where leaves at the largest depth are filled to the left of the tree. Therefore the last internal node e_k has a child, which is the last element in the heap e_n. They have a parent-child relationship, therefore,

   $$k = \lfloor \frac{n+1}{3} \rfloor$$

   Hence, the internal nodes in array A has index $1, 2, \cdots, \lfloor \frac{n+1}{3} \rfloor$ □

   (b) What is the height of the tree? Justify your answer.

   Solution. Given a heap tree with height h. Let d be the depth of the tree. There are at most 3^d nodes at depth d. Then there are at most $\sum_{d=0}^{h} 3^d = 3^{h+1} - 1$ nodes in the tree. Similarly a tree of height $h-1$ has at most $3^h - 1$ nodes. Then a tree of height h has at least $3^h - 1 + 1 = 3^h$ nodes. The following inequality holds,

   $$\begin{aligned} 3^h \leq & n \leq 3^{h+1} - 1 < 3^{h+1} \\ h \leq & \log_3 n < h + 1 \end{aligned}$$

Since $n \in \mathbb{N}$, the height must satisfy

$$h = \lfloor \log_3 n \rfloor$$

□

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

   (a) INSERT(A, key): Insert key into A.

   (b) EXTRACTMAX(A): Remove a key with highest priority from A.

   (c) 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.

   (d) 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.

   (a)

```
INSERT(A, KEY)
A.Heapsize = A.Heapsize + 1
i = A.Heapsize
A[i] = key
while $i > 1$ and $A[Parent(i)] < A[i]$ do
    exchange $A[Parent(i)]$ with $A[i]$
    $i = Parent(i)$
```

The algorithm first assigns the new key to the end of the tree to maintain tree structure. Then the key is moved to correct for heap order. The while loop runs once for each height of the tree. Therefore the worst case running time has $T(n) = O(\log_3 n)$

(b)

```
MAX-HEAPIFY($A, i$)
Input: children of tree rooted at $A[i]$ are heaps
Output: $A$ is a max heap
$l = Left(i)$
$m = Middle(i)$
$r = Right(i)$
if  $l \leq A.Heapsize$ and $A[l] > A[i]$ then
    $largest = l$
else
    $largest = i$
if  $m \leq A.Heapsize$ and $A[m] > A[largest]$ then
    $largest = m$
if  $r \leq A.Heapsize$ and $A[r] > A[largest]$ then
    $largest = r$
if $largest \neq i$ then
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY($A, largest$)
```

The MAX-HEAPIFY algorithm determines the largest element of $A[i]$ and children of $A[i]$. If the tree does not satisfy heap order. $A[largest]$ is moved to the correct position, i.e. i. And the subtree rooted at *largest* might violate heap ordering of the subtree, hence the function is called recursively.. The running time is proportional to height of tree, i.e. $O(\log_3 n)$.
The EXTRACTMAX removes the root i.e. $A[1]$ and replaces it with $A[A.Heapsize]$ such that the tree structure is maintained. Then call MAX-HEAPIFY(A, 1) to reinforce heap order. The algorithm takes $O(\log_3 n)$ since it performs the a constant time of workj on top of $O(\log_3 n)$ time for MAX-HEAPIFY

(c) The idea is that you change set $A[i] = key$ and use MAX-HEAPIFY(A, i) to ensure that heap order is maintained. Worst case is equivalent to EXTRACTMAX(A) and so have worst case complexity of $O(\log_3 n)$

(d) The idea is set $A[i] = A[A.Heapsize]$ such that tree structure is maintained. Then call MAX-HEAPIFY(A, i) to restore heap order. The running time is $\log_3 n$ in worst case.

Question 5

Let I_n be the set of n integers $\{1, 2, \cdots, n\}$ where n is some power of 2. Note that we can easily use an n-bit vector (i.e., an array of n bits) $B[1 \cdots n]$ 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:

(a) INSERT(j) insert j to S

(b) DELETE(j) ddelete j from S

(c) MEMBER(j) return true if $j \in S$ otherwise false

Describe a data structure that supports the above and also MAXIMUM return the largest int in S. such that

(a) worst case complexity of INSERT(j),DELETE(j), and MAXIMUM is $O(\log n)$ while that for MEMBER(j) is $O(1)$

(b) The data structure uses $O(n)$ bits of storage
Note binary representation of an integer i where $1 \leq i \leq n$ takes $\Theta(\log n)$ bits. Same for pointers as well