1. (a) Written by: Xingyu Wang; Read by: Anna Lieu
Let $n \in \mathbb{N}$ such that $n \geq 2$. Let A be an array of n arbitrary numbers. Lines 1-2 take constant time. Denote this as c_1. Each iteration of the inner loop at line 4 takes constant time. Denote this as c_2. There will be $i-1$ line 4 loop iterations, so line 4 takes $c_2(i-1)$ steps. From line 3, we know i can be at most n. So line 4 takes at most $c_2(n-1) = c_2n - c_2$ steps, and this occurs when the loop does not break early. Line 5 takes constant time. Denote this as c_3. So, for one iteration of the outer-loop at lines 3-5, it takes $c_2n - c_2 + c_3$ steps. The loop iterates at most $n-1$ times, and this occurs when the loop does not break early.

Let $c_4 = -c_2 + c_3$.

Altogether, line 3-5 takes at most these steps:

$$(n-1)(c_2n - c_2 + c_3) = (n-1)(c_2n + c_4) = c_2n^2 + c_4n - c_2n - c_4 = c_2n^2 + c_5n - c_4$$

x where $c_5 = c_4 - c_2$.

Denote $c_2n^2 + c_5n - c_4$ as $T_{strange}(n)$. There would $\exists n_0 \geq 2, c \geq c_2$ such that $c_2n^2 + c_5n - c_4 \leq cn^2$, for all $n \geq n_0$, because n^2 grows faster than n and any constants. Eventually function cn^2 will grow faster than $T_{strange}(n)$, after the intersection (n_0, cn_0^2). So $T_{strange}(n) \in \mathcal{O}(n^2)$.

1. (b) Written by: Xingyu Wang; Read by: Anna Lieu
Let $n \in \mathbb{N}$. Consider such input for procedure *strange*: an array of n numbers A_0 such that for each item in the array, the value is the index of that item - 1 (i.e., A_0 of size n = [0, -1, -2, ..., n - 1]) Evidently, each item is smaller by one compared with the last item. In other words,

$$\forall i \in \mathbb{N}, i \geq 2 \Longrightarrow A_0[i] + 1 = A_0[i-1].$$

Lines 1-2 take constant time c_1, where $c_1 \in \mathbb{R}$.

For an arbitrary iteration of the loop on line 3 with variable i's value being i_t, line 4 subtracts all values before $A_0[i_t]$ by 2. Then the new $A_0[i_t - 1]$ = the old $A_0[i_t - 1] - 2 = A_0[i_t] + 1 - 2 = A_0[i_t] - 1$.

Then the if statement on line 5 will never be satisfied. So the program will never return early while in the for loop on line 3, and i will range from 2 to n.

For the inner loop, it takes $c_2(i-1)$ steps, with $c_2 \in \mathbb{R}$.

Let the number of steps on line 5 be $c_3 \in \mathbb{R}$.

As i increases from 2 to n, the sum of each outer-loop iteration will then take:

$$\begin{aligned} t(A_0) &= \Sigma_{i=2}^{n}(c_2(i-1) + c_3) = \Sigma_{i=2}^{n}c_2(i-1) + c_3(n-1) \\ &= c_2\Sigma_{i=2}^{n}(i-1) + c_3(n-1) = c_2\frac{(1+n-1)(n-1)}{2} + c_3n - c_3 \\ &= \frac{c_2}{2}(n^2 - n) + c_3n - c_3 = \frac{c_2}{2}n^2 + (c_3 - \frac{c_2}{2})n - c_3 \\ &\in \mathcal{O}(n^2) \end{aligned}$$

2. (a) Written by: Anna Lieu ; Read by: Xingyu Wang
We have an arbitrary ternary heap, represented by associated array A, with indices that run from 1 to Heapsize n, ie, A[1..n]
Let i = index in the array. Root node of the tree is the first element of the array, at index i = 1. Then we go down a level, and add each element into the array, starting from leftmost node, all the way to rightmost node. We keep going down another level, adding node elements, from left to right, until there are no more levels.
We can find the children (if any) of an element in the array by following these formulas (for calculating index of left, middle, right child of node at index i):
Left(i) = 3i - 1; Middle(i) = 3i; Right(i) = 3i + 1
To find index of parent (if any) of a node at i: Parent(i) = $\lfloor i/3 \rceil$, where $\lfloor \rceil$ is symbol for nearest int

2. (b) (1) Written by: Crystal Yip ; Read by: Anna Lieu
The formula to access the parent of the node at index i is $\lfloor \frac{i-1}{3} \rfloor + 1$ (*). From (*), we know that for a heap with heap size n, the parent of the last leaf, node n, is at index $\lfloor \frac{i-1}{3} \rfloor + 1$ (**). From (**), we know that nodes from index $\lfloor \frac{n-1}{3} \rfloor + 1 + 1$ to n must be leaves since the node at index $\lfloor \frac{n-1}{3} \rfloor + 1$ is the last internal node (***). Combining (*), (**), and (***), if A is an array representing a max-heap starting from index 1, we can conclude that A[1] to A[$\lfloor \frac{n-1}{3} \rfloor + 1$] are internal nodes

2. (b) (2) Written by: Anna Lieu ; Read by: Xingyu Wang
Let n_{max} be the max number of nodes of complete ternary tree T of height h, where last level is full:

n_{max} of Height 0 $\implies 3^0 = 1; n_{max}$ of Height 1 $\implies 3^0 + 3^1 = 4; n_{max}$ of Height 2 $\implies 3^0 + 3^1 + 3^2 = 13$

n_{max} of Height h $\implies \sum_{i=0}^{h} 3^i = \frac{1-3^{h+1}}{1-3} = \frac{3^{h+1}-1}{2}$

Let n be the number of nodes of ternary tree T with height h. The last level of T is not necessarily full, so $n \leq n_{max}$

$$\begin{aligned} n \leq n_{max} &= \frac{3^{h+1}-1}{2} \\ 2n+1 &\leq 3^{h+1} \\ log(2n+1) &\leq (h+1)log(3) \\ log_3(2n+1) &\leq h+1 \\ h &\geq log_3(2n+1) - 1 \end{aligned}$$

For a ternary tree that is full at the last level, ie, with n_{max} nodes:

$$\begin{aligned} n_{max} = \frac{3^{h+1}-1}{2} \implies h &= log_3(2n_{max}+1) - 1 && \text{(Which we derived, where h is an integer)} \\ &\geq log_3(2n+1) - 1 && \text{(Since } n \leq n_{max}\text{)} \end{aligned}$$

Since h is an integer, and the log formula results in an int only if $n = n_{max}$, then we can take the ceiling of the formula to get the height of T that is not necessarily full at the last level. So,

$$h = \lceil log_3(2n+1) - 1 \rceil$$

2.(c) Written by: Crystal Yip ; Read by: Anna Lieu
(1)

```
0. insert(key, A) // key is the int being inserted, and A starts at index 1
1.        A.append(key)
2.        c_index = A.size,     p_index = ⌊(c_index−1)/3⌋ + 1 // set the index for inserted key and its parent
3.        while c_index > 1 and A[c_index] > A[p_index]
4.              swap A[c_index] and A[p_index]
5.              c_index = p_index,     p_index = ⌊(c_index−1)/3⌋ + 1
```

Upper Bound
Claim: while loop executes at most $\lceil log_3(2n+1) \rceil$ times.
Why: The inserted key starts at depth d = h where h is the height of the tree. Each time the while loop executes, the depth of the inserted key reduces by 1 (i.e. the key moves up one level). When the index of the inserted key reaches 1, the while loop does not execute (due to the condition on line 5). Therefore, the while loop executes at most h times and from 2b) (2), we know h = $\lceil log_3(2n+1) \rceil$
So T(n) = $\lceil log_3(2n+1) \rceil$ + c (constant runtime from lines 1,2), and **we have O(logn)**

Lower Bound
Consider the case where the key inserted is larger than all of the existing keys in the heap. I.e. the key needs to be moved to the top of the heap. The key inserted is at depth d = height of the tree on entering the loop.

After 1 iteration of the loop, the key is at depth d-1
. . .
After d-1 iterations of the loop, the inserted key is at the second level of the heap (d = 1) and only one swap remains
Therefore, the total number of iterations = h - 1 + 1 = h = height of the heap = $\lceil log_3(2n+1) \rceil$
Therefore, T(n) = logn and **we have θ(logn)**

(2)
0. **extract_max(A)** // A is indexed starting at 1
1. if A.size $\neq$ 0
2. set the first key index of the heap to the key of the last element in the heap array, then pop the last element
3. bubble_down(A, 1)

0. **bubble_down(A)** // A is indexed starting at 1
1. p_index = 1
2. lt_index = (3*p_index) - 1, md_index = 3*p_index, rt_index = (3*p_index) + 1
3. while lt_index $\leq$ A.size and A[p_index] is less than A[lt_index] or A[md_index] or [rt_index]
4. swap A[p_index] with A[lt_index] or A[md_index] or A[rt_index]
5. p_index = lt_index or md_index or rt_index depending on the swap

Upper Bound
We have a constant runtime from lines 1-5 in extract_max
Claim: Consider the bubble_down code. While loop executes at most h times, where h is the height of the heap
Why: Key being bubbled down starts at depth = 0. Assuming the latter part of the condition on line 5 holds, the loop executes as long as the key has a child node (i.e. lt_index $\leq$A.size). On every iteration of the loop, the depth of the key increases by 1. When the key does not have a child node, it is a leaf and the loop does not execute. Since we know that there cannot exist a leaf such that its depth is greater than the height, we can conclude that the loop can only execute $\lceil log_3(2n+1) \rceil$ times (height of the tree).
Therefore T(n) = $\lceil log_3(2n+1) \rceil + c$, [c is the constant running time for the extract_max code], and **we have O(logn)**

Lower Bound
Consider the case where the last node of the heap is smaller than all of the other elements of a max heap that is FILLED at every level (complete heap). I.e. the last node that is moved to the root needs to bubble down to the bottom level of the heap. The node being bubbled down starts at depth d = 0 on entering the loop.
Suppose h is the height of the heap, i.e. h = $\lceil log_3(2n+1) \rceil$
After 1 iteration of the loop, the node is at depth 1 [line 7]
. . .
After h-1 iterations of the loop, the node reaches the last level of the heap.
Therefore, the total number of iterations = $\lceil log_3(2n+1) \rceil - 1$
Combining with the lines 1-5 in extract_max, T(n) = $\lceil log_3(2n+1) \rceil - 1 + c$
So, **we have θ(logn)**

(3)
0. **update(A, i, key)**
1. A[i] = key
2. max_heapify(A, i)

0. **max_heapify(A, i)** // i must be $\geq$ 1
1. l = left(i), r = right(i), m = middle(i) // the middle child of the key at index i
2. if A[i] is less than any one of its children
3. largest = the max of the left child, right child, middle child, or the key itself
4. if largest $\neq$ i
5. i exchange A[i] switch with A[largest]
6. max_heapify(A,largest)

Upper Bound
We have a constant runtime from line 1 in update
Claim: Consider the max_heapify code. max_heapify is recursively called at most h times, where h is the height of the heap
Why: Suppose A is the array for the heap and i is the index of a key being updated. The precondition restricts i so that it must be at least 1. Assuming the key with the updated value is always smaller than one of its children, max_heapify will be recursively called and a switch between the nodes is made. (#)When the index of the key reaches the leaf level of the heap, *largest* is the key itself [max_heapify: line 3]. The if statement on line 4 will not be executed and no recursive call for max_heap will be made and the current call will terminate. We know that the depth of a leaf cannot be greater than the height of the heap. Thus, we can conclude that max_heapify is recursively called at most h times, where h is the height of the heap and h = $\lceil log_3(2n+1) \rceil$ times (height of the tree).
Therefore T(n) = $\lceil log_3(2n+1) \rceil + c$, [c is the constant running time for the extract_max code], and **we have O(logn)**

Lower Bound
Consider a heap that is FILLED at every level (complete heap). Suppose i = 1 (i.e. first element of the heap is being updated) and A[i] be smaller than all of the nodes. I.e. the newly updated key must move down to the bottom of the heap. The key bubbling down starts at depth = 0.
Suppose h is the height of the heap, i.e. h = $\lceil log_3(2n+1) \rceil$
After 1 iteration of the loop, the node is at depth 1 [line 4]
. . .
After h-1 iterations of the loop, the node reaches the last level of the heap.
Therefore, the total number of iterations = ($\lceil log_3(2n+1) \rceil$) - 1
Combining with the lines 1-5 in extract_max, T(n) = $\lceil log_3(2n+1) \rceil$ - 1 + c
So, **we have θ(logn)**

(4)
0. **remove(A, i)** // i must be $\geq$1
1. ret = A[i]
2. last = A.pop
3. if A.size > 0
4. A[0] = last
5. max_heapify(A, i) // using max_heapify from part (3)

Upper Bound
We have constant runtime from lines 1-4 in remove
Claim: Consider the max_heapify code. max_heapify is recursively called at most h times, where h is the height of the heap
Why: Suppose A is the array for the heap and i is the index of the key being removed in the heap. The precondition restricts i so that it must be at least 1. After A[i] is removed, it is replaced with the last key of the heap. Assuming that key is always smaller than one of its children, max_heapify will be recursively called and a switch between the nodes is made.**refer to (#) in part (3) for the same explanation
Therefore T(n) = $\lceil log_3(2n+1) \rceil + c$, [c is the constant running time for the extract_max code], and **we have O(logn)**

Lower Bound
Consider a heap that is FILLED at every level (complete heap) the case where the first element of the heap is being removed and the last key of the heap be the smallest element of the heap. I.e. the newly updated key must move down to the bottom of the heap. The key bubbling down starts at depth = 0.
After 1 iteration of the loop, the node is at depth 1 [line 4]
. . .
After h-1 iterations of the loop, where h is the height of the heap, the node reaches the last level of the heap.
Therefore, the total number of iterations = $\lceil log_3(2n+1) \rceil$ - 1
Combining with the lines 1-5 in extract_max, T(n) = $\lceil log_3(2n+1) \rceil$ - 1 + c
So, **we have θ(logn)**