Changelog

- Update 1: Question 1 A is fixed (it is $\theta(n^2 \log n)$ now)

Question 1 (1.5 points)

A. $T(n) = 9T(n/3) + n^2$

Recursive Tree Method:

$$T(n) = 9 \cdot \frac{n^2}{9} + n^2 = n^2$$

$$T(n) = 9 \cdot \left(\frac{n^2}{9}\right) + n^2 = \theta(n^2 \log n)$$
Master Theorem Method:

\[ T(n) = 9T\left(\frac{n}{3}\right) + n^2 \]

\[ a = 9, \; b = 3, \text{ and } f(n) = n^2 \]

The function \( n^{\log_b a} = n^{\log_3 9} = n^2 \)

We have \( f(n) = n^2 = \theta(n^{\log_b a \log n}) \)

\[ \theta(n^{\log_b a \log n}) = \theta(n^2 \log n) \]

B. \( T(n) = 9T(n/3) + n \)

Recursive Tree Method:

\[ T(n) = 9T\left(\frac{n}{3}\right) + n \]

\[ a = 9, \; b = 3, \text{ and } f(n) = n \]

The function \( n^{\log_b a} = n^{\log_3 9} = n^2 \)

We have \( f(n) = n < n^2 \) therefore \( \theta(n^{\log_b a \log n}) = \theta(n^2 \log n) \)
C.\( T(n) = 9T(n/3) + 1 \)

Recursive Tree Method:

\[
T(n) = 9T(n/3) + 1
\]

\[
a = 9,\ b = 3,\ \text{and}\ f(n) = 1
\]

The function \( n^{\log_b a} = n^{\log_3 9} = n^2 \)

We have \( f(n) = 1 < n^2 \) therefore \( \theta(n^{\log_b a}) = \theta(n^{\log_3 9}) = \theta(n^2) \)
Question 2 (2 points)

```python
def find_max(A, begin, end):
    if end - begin == 2:
        return max(A[begin], A[end - 1])
    if end - begin == 1:
        return A[begin]
    mid = begin + (end - begin) / 2
    return max(find_max(A, begin, mid + 1), find_max(A, mid, end))
```

Above algorithm has base cases on line 3 and 5, and recursive calls on line 7. There are two recursive calls that are roughly half the size.

\[
T(n) = 2T\left(\frac{n}{2}\right) + 1
\]

\[a = 2, b = 2, \log_b a = 1\]

\[f(n) = \theta(1) = O(n^{\log_b a - \epsilon}) = O(n^{1-\epsilon}), \epsilon = 1 > 0\]

\[T(n) = \theta(n^{\log_b a}) = \theta(n)\]
Question 3 (1.5 points)

```python
def find_kth(A, k):
    random_elem = A[random(0, len(A))]
    less_than = []
    equal_to = []
    greater_than = []
    for elem in A:
        if elem < random_elem:
            less_than.append(elem)
        elif elem > random_elem:
            greater_than.append(elem)
        else:
            equal_to.append(elem)
    if len(less_than) >= k:
        return find_kth(less_than, k)
    elif len(less_than) + len(equal_to) >= k:
        return random_elem
    else:
        return find_kth(greater_than, k - (len(less_than) + len(equal_to)))
```

Lines 2-5 take constant time
Lines 6-12 take n time
Lines 13, 15, 16, and 17 takes constant time
Lines 14, and 18 are recursive calls but since they are conditioned on the size of less_than, there will only be one recursive call for each function call

**Best case**

\[ k = \text{median of the array} \]

The random element is chosen to be the median of the array Therefore the function would be:

\[ T(n) = n + c \implies f(n) = n \]

\[ \Omega(n) \]

\[ Cn < n \]

\[ C = \frac{1}{2}, n_0 = 1 \]

**Worst case**

\( K = 1 \) or \( K = \text{length}(A) \)

The random element chosen is the \( \text{length}(A) - k \) element for all calls to find a random element. Therefore the function would be:

\[ T(n) = T(n - 1) + n + c \]
Number of recursive calls = n-1
Number of nodes per recursive level = 1
Cost per node = n
Cost per level = n

Therefore \( \sum_{i=0}^{n-1} n + c = (n - 1)(n + c) = n^2 + cn - nc = O(n^2) \)

Since \( f(n) \) within \( O(n^2) \) \( \neq \) \( f(n) \) within \( \Omega(1) \) therefore \( \Theta() \) can’t be found
Question 4 (1 point)

```python
# Return boolean if A has at least k elems that are < q

def countLessThan(A, q, k):
    count = 0
    queue = [0]  # Queue for indices to look at (start with root 0)

    while len(queue) > 0 and count < k:
        i = queue[0]  # Get the next element in the queue
        queue = queue[1:]  # Remove it from the queue

        if i < len(A):  # Sanity check to not to go out of bounds
            if A[i] < q:
                count += 1
                queue.append(2*i + 1)  # Add left child to queue
                queue.append(2*i + 2)  # Add right child to queue

    return count >= k
```

We are starting from the root node, assuming it has index 0. We check if it is less than requested \( q \). If that’s the case, we increment our counter, and add current node’s children to the queue. If not, we move on to the next element in the queue, that’s because of the property of min-heaps: we know that parent is always smaller than its children.

We stop checking if we run out of elements in our queue, or if we reach already requested at least \( k \) elements. This addition to while condition actually makes sure that our while loop runs at most \( k + \) some constant times (caused by checking elements that are \( > q \)).