1.

No, because this algorithm may terminate in the middle of the vector and results in a running time smaller than n, which means that n is not the correct lower bound of the running time and we cannot say it is $\Omega(n)$. Alternatively, we know that the upper bound of this algorithm is $O(n)$, and if the statement that the running time is $\Omega(n)$ is true, then the combined notation of running time will be $\Theta(n)$, which we know is incorrect, therefore the running time is not $\Omega(n)$.

2.

(a)

$$O\left(\frac{3+\lceil\sqrt{n}\rceil}{2}\right) \rightarrow O(\sqrt{n})$$

(b) Yes. Since the definition of polynomial is n^x where x > 0; and $\sqrt{n} = n^{\frac{1}{2}}$, we can say that the running time of this algorithm is polynomial.

3.

Since we are not asked for a formal proof and are only required to demonstrate the claim is incorrect. We can use a counterexample to show that the claim is incorrect.

Counterexample: 5 is an odd positive integer; and $5^2 + 2 * 5 = 35$ which is not evenly divisible by 3. Therefore, we have shown here that the claim is incorrect.

4. (a)

```
# algorithm in python
def find_best_prof(A: list, n: int) -> list:
    best = list()
    # sort the array of professors by increasing difficulty
    # (alternatively, we can also sort by decreasing humor and get the same
    # result)
    by_difficulty = sorted(A, key=lambda x: x[DIFFICULTY])

    # this variable stores the highest humor score of professors added to
    # the set. Initiate it as -1 since the range is [0, 100] so that the
    # first (easiest) professor will be included in set
    highest_humor = -1

    # Since the array is sorted by increasing difficulty, the difficulty
    # of professor p will be greater than every other professor in the set.
    # therefore, p must have a higher humor score than every professor
    # currently in the set to satisfy our selection criteria and be added
    # to set. Otherwise, p is less funny and more difficult and cannot be
    # added to set.
    for p in by_difficulty:
        if p[HUMOR] > highest_humor:
            # update the highest humor score
            highest_humor = p[HUMOR]
            best.append(p)
    return best
```

(b) The worst case happens when more difficult professor in A also has higher humor. Which means in every iteration of the for-loop, we need to execute the if branch. However, since everything in the if branch takes amortized constant time to run, which means the for-loop takes $\Theta(n)$ time even in worst case.

Assuming the sorting algorithm takes $\Theta(n \log n)$ time in the worst case, the total runtime of this algorithm is $\Theta(n \log n)$.

(c) I believe my algorithm has the best possible asymptotic runtime. Since we have to compare the scores of professors, we have to use comparison-based sorting or similar algorithm to achieve our goal. Therefore, $O(n \log n)$ should be the best possible asymptotic runtime.

5.

```
# algorithm in python
def have_common_key(A: MaxHeap, B: MaxHeap, n: int) -> bool:
    try:
        while n > 0:
            # get the max element of each max heap and compare them
            max_A = A.max()
            max_B = B.max()
            if max_A == max_B:
                # common key found
                return True
            # extract max from whichever max heap that has larger max value
            if max_A > max_B:
                A.extractMax()
            else:
                B.extractMax()
            # decrement n since we removed an element
            n -= 1
    except Exception:
        # assume that max() and extractMax() will throw an exception if the max
        # heap becomes empty, in this case there is no common key
        return False
    # if both max heap becomes empty (while-loop terminates, n == 0)
    # also indicates no common key
    return False
```

The while-loop has a runtime of O(n) since it decrement n by 1 each time and may terminate early. Inside the loop, we know that extractMax() function takes O(log n) time, the max() function takes constant time, and every other operation also takes constant time, which makes each iteration of the loop O(log n) time. Multiply them together and we have the total runtime to be O(n log n).