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 Ω(n). Alternatively, we know that the upper bound of this algorithm is O(n), and if the statement that the running time is Ω(n) is true, then the combined notation of running time will be Θ(n), which we know is incorrect, therefore the running time is not Ω(n).

2.

(a)

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

(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, n: int) -> set:
    best = set()
    # add the professor with the lowest difficulty to the set
    best.add(min(A, key=lambda x: x[DIFFICULTY]))
    # then add the professor with the highest humor to the set
    best.add(max(A, key=lambda x: x[HUMOR]))

    # then for each professor (P) in array A, if P is not already in set "best", and
    # there is no professor in set "best" that is both more funny and less difficult than P
    # then add P to set "best"
    for p in A:
        if p not in best:
            # only check for p if not in set
            should_add_to_set = True
            for i in best:
                if i[DIFFICULTY] < p[DIFFICULTY] and i[HUMOR] > p[HUMOR]:
                    # found a professor that is less difficult and more funny
                    # do not add to set
                    should_add_to_set = False
                    break
            if should_add_to_set:
                # there is no professor in best that is more funny and less difficult
                # we can add them to set
                best.add(p)
    return best
```

(b) The worst case happens when there is no professor in input A that is less funny and more difficult than any other professor in A, which means that the inner for-loop will not terminate early in each iteration in the outer loop; and the size of set increase by 1 in each iteration of outer loop. Which means that, in the i-th iteration of outer loop, the inner loop will have i ＋ c iterations, where c = 0 or 1. Removing the constant c, we have the total runtime of the nested loop to be about $\Theta(\sum_{i=1}^{n} i) \rightarrow \Theta(n^2)$

Since the part of code before the loop has runtime complexity of Θ(n), the total runtime of the entire algorithm is $\Theta(n^2)$.

(c) I believe my algorithm has the best possible asymptotic runtime. For each professor in array A, we must compare them with the difficulty and humor of every other professor in the set to decide if they satisfy our selection criteria; and in the worst case, every professor will be added in the set, resulting a $\Theta(n^2)$ runtime.

5.

```
# algorithm in python
def have_common_key(A: MaxHeap, B: MaxHeap, n: int) -> bool:
    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
        try:
            # extract the maximum element of the max heap that has a greater max value
            if max_A > max_B:
                A.extractMax()
            else:
                B.extractMax()
        except Exception:
            # assume that extractMax() will throw an exception if any one of the max heaps
            # becomes empty, in this case there is no common key
            return False
        # decrement n since we removed an element
        n -= 1
    # if both max heap becomes empty (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).