Note that many problems in this class are mathematical in nature. As such, there are likely many correct answers – yours do not need to match mine to get full credit, even on questions where we grade on correctness.

1. Recall the linear search algorithm:

```
int linearSearch(const std::vector<int> & numbers, int target)
{
    int i;
    int n = numbers.size();
    for(i=0; i < n; i++)
    {
        if( numbers[i] == target )
        {
            return i;
        }
    }
    throw ElementNotFoundException("Element not found by linear search.");
}
```

   Would it be correct to describe this algorithm as having a running time of $\Omega(n)$, where n is the size of the input vector? In 1-3 sentences, justify your answer.

   Yes it does; by default, we are discussing the worst-case running time (unless we state otherwise) and such, for linear search, is lower-bounded by a linear function of the input size. Do not make the mistake of thinking of Ω notation as the "best case" running time of an algorithm!

2. Recall the following algorithm, which determines if a given input positive integer is prime:

```
bool isPrime?(positive integer n)
  if n = 1 then
    return false
  else if n is 2 then
    return true
  else if n is even then
    return false
  for i from 3 to ⌈√n⌉ by twos do
    if i divides n leaving no remainder then
      return false
  return true
```

   (a) Express the running time of this algorithm in $\mathcal{O}$ notation in terms of n, the value of the input number provided. Assume that each call to the divisibility test inside the for loop takes constant time.

   This is $\mathcal{O}(\sqrt{n})$.

(b) Is this a polynomial time algorithm, according to the definition of that term given in lecture? In 1-3 sentences, justify your answer.

No. Recall that polynomial time refers to a polynomial function *of the input size*, not just "can be written as a polynomial."

The size of the input is $\Theta(\log n)$, as that is how many bits are needed to represent an arbitrary positive integer with value n. Note that we are not assuming any particular data type for n: it could be a C++ unsigned, but it could also be hundreds of bits long. Think about how slow this function would be for such a number and you'll see why we shouldn't consider this to be an efficient algorithm.

3. We say a positive integer n is "resolute" if 3 evenly divides (that is, leaves no remainder when used as a divisor) $n^2 + 2n$. I put forth a claim that all odd positive integers are resolute. Demonstrate that this claim is incorrect.

A complete answer is a counter-example **and** a demonstration that this counter-example applies. Writing just a number **is not** a complete answer (and, in fact, will get no credit).

Here is an example complete answer:

$n = 5$ is a counter-example. When $n = 5$, then $n^2 + 2n = 35$, which is not a multiple of 3.

4. Suppose we have an array A of n professors; each teacher has two characteristics: difficulty (a real number in the range $[0, 10]$, where a higher number indicates a more difficult teacher) and humor (a real number in the range $[0, 100]$, where a higher number indicates a funnier teacher). You may assume that each value is distinct (no two professors are exactly as difficult or exactly as funny), but not that the precision of real numbers is limited (as floats and doubles are in C++ and Java). Our goal is to determine a set of professors that might satisfy an answer to the question "who is the easiest teacher that is the funniest?" We wish to avoid professors with high difficulty and low humor. We would like to find the largest subset of the input data such that no professor in the chosen subset is **both** less funny **and** more difficult than another professor in the original input set. Note that if professor A is easier than professor B, it *does not follow* that professor A is also funnier than professor B.

For example, if our dataset is:

Professor	Difficulty	Humor
Venabili	4	65
Jones Jr	8	15
Jones Sr	8.5	2
Grant	2	35
Moriarty	10	0
Plum	9	85
Walsh	8.2	90

Then we want to return Venabili, Grant, and Walsh.

Each name on that list is a professor from some work of fiction. How many did you recognize?

(a) Devise an algorithm which takes as input A and n, and outputs the resulting set. Your algorithm does not need to be particularly efficient, but you may not give an algorithm that enumerates every subset.

Do not have your answer depend on limitations of any programming language. For example, while difficulty or humor are numeric types, do not use that, say, a **double** in C++ is "not really" a real number.

There are many ways to do this; here's one:

Sort professors along any dimension
for $i = 1 \to n - 1$ **do**
  for $j = i + 1 \to n$ **do**
    if A_i is easier and funnier than A_j **then**
      Remove A_j
      Go to the next value of j
return All professors not removed

(b) Analyze the worst-case runtime of your algorithm using Θ-notation.

This takes time $\Theta(n^2)$

(c) Do you believe your algorithm has obtained the best possible asymptotic runtime? Explain your reasoning. Note that finding the best possible asymptotic runtime **is not** required to get full credit on this question.

Probably not; this doesn't take into account transitivity, for example, which might allow us to not have to check each pair explicitly.

(It turns out this problem can, in fact, be solved in better than $\Theta(n^2)$ time. There are several good reasons not related to transitivity that would satisfy the answer to this part.)

5. Suppose you have two max-heaps, A and B, with a total of n elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time $\mathcal{O}(n \log n)$. For this problem, do not use the fact that heaps are arrays. Rather, use the API of a heap; that is, you may call the following functions:

- `max()`, which returns the maximum element in the heap at the moment.
- `extractMax()`, which removes the maximum element in the heap.
- `insert(e)`, which inserts the parameter into the heap.

Each of these functions takes the time that they did when you learned them in a course like ICS 46. The heaps are implemented as binary heaps via an array (or equivalent).

Give a brief explanation for why your algorithm has the required running time.

Compare the max element of A and B. If they're equal, we're done. Otherwise, remove the larger one (as it can't be in the other heap, as it's larger than that heap's largest element). If either heap is empty, stop and return false. Otherwise, repeat.

This will iterate at most n times, and each iteration takes $\mathcal{O}(\log n)$ time, for a total of $\mathcal{O}(n \log n)$.