Exercises #1
Loop Invariants

Theoretical Background

In order to prove the correctness of a loop using invariants, we must first find a suitable loop invariant condition and then show the following three things:

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

We also need to show that the loop terminates:

- **Progress:** Each iteration gets us closer to the end until eventually we finish

---

Proving correction of loop based programs

1. **Prime Numbers**

   Consider the following (non-efficient) function that checks if an integer number n is prime:

   ```
   1: isPrime(n)
   2:     for i = 2 to ⌊√n⌋
   3:         if n is dividable by i then return false
   4:     return true
   ```

   (a) State an useful **invariant** of the loop towards proving the correction of the algorithm:

   (b) **Prove** that the algorithm is correct using your previously defined invariant. Are the four conditions mentioned above followed?

2. **Second largest element**

   Consider you have an array A with n distinct integer numbers and that you want to find the second largest number of the array. Write (pseudo) code for this algorithm using a single pass trough the elements (with a loop) and prove its correction using invariants.

   What is the invariant? Are the four conditions mentioned above followed?

3. **Insertion Sort**

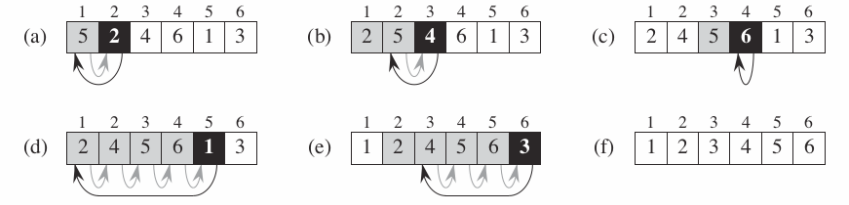
Prove the correction of the following program using loop invariants.

You can treat the inner `while` loop more informally (understanding what it does), but you should state an invariant for the outer `for` loop [if you want to test yourself and obtain a full proof, you also need to (separately) prove that the inner loop is correct].

(code and image taken from "Introduction to Algorithms")

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



4. **Binary Search**

Assuming you have an array A with n numbers sorted increasingly, in order to find if an element x exists in the array you case use binary search. Let m be the middle position: if $x = A[m]$ then we found it; if $x > A[m]$ then x can only be in the $A[1 \ldots m-1]$ half; if $x < A[m]$ then x can only be in the $A[m+1 \ldots n]$ half. This effectively reduces the search space in two.

Write (pseudo)code for an iterative version (not recursive) of this algorithm and prove its correction using invariants.