To reason about programs involving loops, we used symbolic execution combined with carefully crafted loop invariants. To reason about programs that employ recursion, we utilize a proof principle called *induction* which allows us to assume our recursive function calls “just work” under a certain set of conditions. Like loop invariants, inductive reasoning can be used for both program verification and design. However, unlike loop invariants which can feel somewhat arbitrary to design, I would argue that inductive reasoning is quite natural to perform when designing recursive programs. I would go so far to say that this is the strength of recursion and why it feels like recursive programs work once you write everything done (although getting there may be a bit difficult)!

In this guided lab, we’ll go through the process of verifying and designing a number of simple recursive functions of increasing complexity. Rather than use a stack-and-heap-based model of execution, we will employ our *substitutive model* coupled with *case analysis* to verify our functions.

### Case Analysis

When verifying a function whose inputs we know, we can simply execute the function by hand. However, we are likely verifying the behavior of functions whose inputs we don’t know in advance. With effectful code, we used symbolic execution to reason about the state of our variables. In the absence of effects, we can simply perform *case analysis* on our inputs and and then evaluate our program.

For example, consider a function which negates a boolean using a conditional rather than the boolean negation operator (!):

```java
public static boolean negate(boolean b) {
    if (b) {
        return false;
    } else {
        return true;
    }
}
```

Note that we cannot simply execute the program because `b` is unknown. However, we know that there are only two possible values of boolean type: `true` and `false`. To proceed we can consider two cases of the function’s execution—when `b` is `true` and `b` is `false”—and then verify each case independently.

**Claim 1.** *negate(b) is correct, i.e., negate(b) is equivalent to !b.*

**Proof.** There are two cases to consider:

- **b == true:** `negate(true)` evaluates to false by going into the if-branch of the conditional.
- **b == false:** `negate(false)` evaluates to true by going into the else-branch of the conditional.

\[\square\]
In this context, the force of the case analysis is to give values to any variables in our program so that evaluation works.

**Exercise (Case Analysis Example)** Consider the following function that computes the exclusive-or (xor) of two boolean values. Recall that the xor function returns true when exactly one of its inputs is true.

```java
public static boolean xor(boolean b1, boolean b2) {
    if (b1) {
        if (b2) {
            return false;
        } else {
            return true;
        }
    } else {
        if (b2) {
            return true;
        } else {
            return false;
        }
    }
}
```

Verify that the function is correct through a case analysis similar to above. Describe what you are performing case analysis over, the possibilities, and prove each case correct to conclude that the overall function is correct.

### Inductive Reasoning

Booleans are a simple type to perform case analysis over because it has a finite set of values. However, most interesting types have an infinite number of values to consider, so we cannot just perform naive case analysis on each value. We must instead determine a way to describe an infinite such set of values using a finite number of cases.

For example, consider the natural numbers which consists of zero and the positive integers, e.g., 1, 5, and 10. Even though there is an infinite amount of such numbers, we can categorize them into two sorts: zero and non-zero natural numbers. For example, consider the factorial function defined over the natural numbers:

```java
// pre: n >= 0
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

1^Note that there is no distinguished type for natural numbers in Java, so we use int instead
Our case analysis consider the case when \( n \) is 0 and when \( n \) is non-zero. Note that this does not give a concrete value to \( n \) in the second case, but it is enough information to drive evaluation through the conditional!

In the case when \( n == 0 \), \( \text{factorial}(0) \) evaluates to 1 by entering the if-branch of the conditional. Thus, our function works when \( n == 0 \). However, in the non-zero case, we know that \( \text{factorial}(n) \) evaluates to the result of \( n \times \text{factorial}(n-1) \), but we need to know how the recursive call \( \text{factorial}(n-1) \) will resolve to make progress!

Inductive reasoning allows us to directly reason about the behavior of the recursive calls of our program. When checking the correctness of our recursive programs, we are allowed to make an inductive assumption that any recursive call in our program is correct provided:

1. The pre-conditions of the function are met.
2. The arguments to the recursive function call are smaller than the inputs.

The first condition ensures that we aren’t violating any assumptions that the function makes. The second condition ensures that our recursive calls eventually terminate, i.e., they do not go into infinite loops.

To be able to assume that \( \text{factorial}(n-1) \) works, i.e., it produces \((n - 1)!\), we must prove these two cases. First, we know that \( n - 1 >= 0 \) because, by virtue of our case analysis, we know \( n \) is non-zero, therefore, subtracting 1 gives us a non-negative value. Second, we see immediately that \( n - 1 \) is smaller than the input to the function, \( n \) by the definition of subtraction.

Because of this, we now know that \( \text{factorial}(n-1) \) produces \((n - 1)!\). With this knowledge, we know that the expression \( n \times \text{factorial}(n-1) \) evaluates to \( n(n-1)! = n! \) as desired.

In summary, to perform inductive reasoning over recursive functions:

1. Describe the inputs to the function in terms of a finite set of cases.
2. Perform case analysis over those cases.
3. In the cases when recursive function calls occur, assume that the recursive function calls “just work”, i.e., make an inductive assumption provided that you can show:
   (a) The recursive function call obeys the pre-conditions of the function.
   (b) The recursive function call is made on smaller arguments than the original inputs to the original function.

Exercise (Simple Verification) Use inductive reasoning to verify that the following program that sums the even natural numbers from 1, . . . , \( n \) is correct:

```java
// pre: n >= 0
public static int sumEvens(int n) {
```
if (n == 0) {
    return 0;
} else {
    if (n % 2 == 0) {
        return n + sumEvens(n-1);
    } else {
        return sumEvens(n-1);
    }
}

Make sure to follow the inductive reasoning above, explicitly proving each of the cases you identify. State any inductive assumptions that you make explicitly along with proofs that the two requirements to make your inductive assumptions are met.

Exercise (Not-as-simple Verification) Use inductive reasoning to verify that the following program that performs binary search is correct:

```java
public static boolean bsearch(int[] arr, int v, int lo, int hi) {
    if (lo >= hi) {
        return false;
    } else {
        int mid = (lo + hi) / 2; // Bad! But why?
        if (v == arr[mid]) {
            return true;
        } else if (v < arr[mid]) {
            return bsearch(arr, v, lo, mid - 1);
        } else {
            return bsearch(arr, v, mid + 1, hi);
        }
    }
}
```

*Hint:* Like our complexity analysis of bsearch, you should consider the primary “input” to the function to be the length of the array under consideration. What are the finite set of cases that you can break then length into?

Exercise (Program Design) Use inductive reasoning to write a recursive function `sumDigits(n)` that takes a natural number `n` and returns the sum of the digits of `n`. For example `sumDigits(3153)` evaluates to 12. Recall that division by 10 and mod by 10 can be used to remove and return the
right-most digit of a number.

When designing this function using inductive reasoning, break up the function into cases and assume that your function works on smaller inputs.