3 Advanced Proof Techniques

This chapter will discuss some more advanced proof techniques, building on what we discussed in the previous section. We'll start in Section 3.1 by reviewing induction, a powerful technique you've probably seen before. In Section 3.2, we introduce correctness proofs for algorithms, the main kind of proof we'll be using in this course, and in Section 3.3, we'll introduce loop invariants, a way to prove the correctness of certain programs using induction. Finally in Section 3.4, we'll discuss proof by contradiction, a powerful technique for proving that particular objects (or algorithms) can't exist, which we'll rely on later in the course, particularly in Chapter 7.

3.1 Induction

One of the most powerful proof techniques is induction. Induction is a technique for proving a "for all" statement quantified over non-negative integers. It is based on the important axiom that every non-empty subset of $\mathbb{Z}_{\geq 0}$ has a minimum element.

An induction proof proves a statement of the form $\forall\, n \in \mathbb{Z}_{\geq 0}$, $P(n)$ by proving

$$\forall\, n \in \mathbb{Z}_{\geq 0}\, (\forall\, k \in \mathbb{Z}_{\geq 0} \text{ with } k < n,\, P(k)) \Rightarrow P(n)$$

Here we have replaced the original predicate $P(n)$ with the new predicate "If $P(k)$ for all $0 \leq k < n$, then $P(n)$". Since we have given ourselves an additional assumption to work with, this new predicate is often easier to prove. This form of induction is often known as "strong" induction, in contrast to "weak" induction where we only assume $P(n-1)$ instead of assuming $P(k)$ for all $0 \leq k < n$. There is no particular benefit to using weak induction, however, so I encourage you to always use the form presented above.

It may not be clear however why proving the inductive version of the statement implies the original statement. To see why induction is valid, it is helpful to think about it as giving us a template to create a proof of $P(n)$ for whatever value of n we want. For $n = 0$, the assumption "$P(k)$ holds for all $0 \leq k < n$" is vacuously true (there are no possible choices of k which are ≥ 0 but also < 0), so our assumption has given us nothing and our induction proof must directly prove $P(0)$ (we often call this a base case). For $n = 1$, our induction proof shows that $P(0) \Rightarrow P(1)$. Since we already have a direct proof that $P(0)$ is true, we can combine these to get a proof that $P(1)$ is true (we know that $P(0)$ is true, and we know "if $P(0)$ is true then $P(1)$ is true", so we conclude $P(1)$ is true). Similarly, our induction proof shows that $P(0) \wedge P(1) \Rightarrow P(2)$. Since we have now shown $P(0)$ and $P(1)$, we can combine these proofs together with the proof that $P(0) \wedge P(1) \Rightarrow P(2)$ to get a proof that $P(2)$ is true. Continuing the pattern, we can use our induction template to construct a direct proof of $P(n)$ for any given n by first constructing direct proofs for $P(0), \ldots, P(n-1)$.

Don't despair even if the correctness of induction feels suspect, however. It's possible to construct correct induction proofs even if the fact that induction works still feels like magic.

Let's see an example of the kind of claim that we can prove with induction.

Theorem 11. *$\forall\, n \in \mathbb{Z}_{\geq 0}$, $n^3 - n$ is divisible by 3*

Proof. We prove this by induction on n.

- Base Case: If $n = 0$, then $n^3 - n = 0^3 - 0 = 0$, which is divisible by 3.
- Induction Hypothesis: Suppose that $\forall\, k \in \mathbb{Z}_{\geq 0}$ with $k < n$, $k^3 - k$ is divisible by 3.
- Induction Step: We will show that $n^3 - n$ is divisible by 3. Since $n > 0$, $n - 1 \in \mathbb{Z}_{\geq 0}$, so by our induction hypothesis we have that $(n-1)^3 - (n-1)$ is divisible by 3. We note that

$$\begin{aligned}(n-1)^3 - (n-1) &= n^3 - 3n^2 + 3n - 1 - (n-1) \\ &= n^3 - 3n^2 + 2n \\ &= n^3 - n - 3(n^2 - n).\end{aligned}$$

  Therefore $n^3 - n - 3(n^2 - n)$ is divisible by 3. Since n is an integer, $3(n^2 - n)$ is divisible by 3 as well, so we conclude that $n^3 - n$ is divisible by 3.

□

This proof illustrates the main pieces of an induction proof.

- First we explicitly state what variable we are inducting on. Sometimes this will be obvious, but other times the choice of induction variable will be trickier (we may need to induct on the sum or difference of two values for example).
- Next we have the base case, where we show the claim for 0 (or whatever the smallest value the claim applies to is) directly
- Then we state our induction hypothesis, which is just assuming that the claim holds for all values of k less than n (We don't literally have to call our variables n and k, but it's important that we call the variable we're assuming the claim holds for something different from the variable we are inducting on. If we're inducting on n and assume that the claim holds for n, our induction hypothesis is already assuming what we're trying to prove in the induction step!)
  - Sometimes it may be convenient to instead assume the claim holds for values up to and including n, then prove it for $n+1$. This is also fine, as long as you use the same form in the induction hypothesis and induction step).
- Finally, we have the induction step, which is usually the meat of the proof. This is where we prove that our claim holds for n using the induction hypothesis. (If your induction step does not use your induction hypothesis either you did something wrong or you didn't need to be using induction in the first place)

At its core, induction is a technique for proving universal statements about non-negative integers. We can apply induction to many other kinds of problems however, by converting them into universal statements about non-negative integers. The key is usually choosing an appropriate non-negative integer variable to induct on. Here is an example of using induction to prove a theorem about strings:

Theorem 12. *Let Σ be an alphabet and let $x, y \in \Sigma^*$ be non-empty strings such that $xy = yx$. Then there exists a string $z \in \Sigma^*$ and integers $a, b \in \mathbb{Z}^+$ such that $x = z^a$ and $y = z^b$.*

Proof. We prove this by induction on $|xy|$.

- Base case: $|xy| = 2$ (this is the smallest possible length for xy since x and y are nonempty). In this case, x and y must both have length 1, so since $xy = yx$, we have that $x = y$. Therefore we can choose $z = x$ and $a = b = 1$, and we have that $z^a = x$ and $z^b = y$.
- Induction Hypothesis: Assume that for all $u, v \in \Sigma^*$ with $|uv| < |xy|$, $uv = vu \Rightarrow \exists\, z \in \Sigma^*, \exists\, a, b \in \mathbb{Z}^+,\ u = z^a \wedge v = z^b$.
- Induction Step: We consider two cases:

  Case 1: Suppose $|x| = |y|$. Since $xy = yx$, we know that x is the same as the first $|x|$ many characters of y, and since $|y| = |x|$, this is all of y, so $x = y$. Therefore we can choose $z = x$ and $a, b = 1$.

  Case 2: Otherwise, $|x| \neq |y|$. Assume without loss of generality that $|x| > |y|$. Since $xy = yx$, the first $|y|$ characters of x are a copy of y, so we can write x as yw, for some non-empty string $w \in \Sigma^*$. Substituting this into our equality $xy = yx$, we get that $ywy = yyw$. Peeling off y from the front of both strings, this implies that $wy = yw$. Furthermore, since $|y| > 0$, $|wy| < |xy|$. Therefore, by our induction hypothesis, there exists a string $z \in \Sigma^*$ and integers $a, b \in \mathbb{Z}^+$ such that $w = z^a$ and $y = z^b$. For the same z, we have that $x = yw = z^b z^a = z^{a+b}$, so x and y can both be created by concatenating copies of z, which is what we wanted to show.

□

Note that this induction proof would actually be complete without the base case. The case where $|xy| = 2$ is already covered in the inductive step by our special case for when $|x| = |y|$. There's no real harm in covering this case twice, though, so it's fine to include a base case even when it's technically not needed.

This proof also used the useful phrase "without loss of generality". This phrase (sometimes abbreviated WLOG) is essentially a shortcut for doing a proof by cases where the proofs for the different cases are exactly the same except with the names of the variables swapped. In this case, rather than giving separate proofs for the case where $|x| > |y|$ and the case where $|y| > |x|$, we arbitrarily choose to assume $|x| > |y|$, and use the phrase "without loss of generality" to let the reader know that they could obtain the proof for the other case by simply swapping x and y in the rest of the proof.

Back in Section 1.4 we saw a simple proof that $\sum_{i=1}^{n} i = n(n+1)/2$. This is probably the most useful summation formula to know for analyzing algorithms, but the second most useful is the one we'll prove now, using induction.

Theorem 13. *For all $n \in \mathbb{Z}_{\geq 0}$ and all $x \in \mathbb{R} \setminus \{1\}$,*

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Proof. We prove this by induction on n.

- Base Case: If $n = 0$, then since $x \neq 1$ (and thus $x - 1 \neq 0$) we have

$$\sum_{i=0}^{0} x^i = x^0 = 1 = \frac{x-1}{x-1} = \frac{x^{0+1} - 1}{x - 1}$$

- Induction Hypothesis: Assume that for all $k \in \mathbb{Z}_{\geq 0}$ with $k < n$, $\sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1}$
- Induction Step: We have that

$$\begin{aligned}
\sum_{i=0}^{n} x^i &= x^n + \sum_{i=0}^{n-1} x^i && \text{(Since } n > 0\text{)} \\
&= x^n + \frac{x^n - 1}{x - 1} && \text{(by our IH)} \\
&= \frac{x^{n+1} - x^n}{x - 1} + \frac{x^n - 1}{x - 1} \\
&= \frac{x^{n+1} - 1}{x - 1}
\end{aligned}$$

□

Note that we split the summation from 0 to n into the last term plus a summation from 0 to $n - 1$ so that we could apply the induction hypothesis. This is a standard technique that we can use to apply induction to many summations. Finding the right formula for a summation can be tricky, but if we have a guess at the right answer (e.g. based on finding a pattern in the value of the summation for small values of n) this technique will usually let us prove that the guess is correct.

3.2 Proving the Correctness of Algorithms

The most common proofs you will need to write in this course are proofs of correctness for algorithms. A proof of correctness for an algorithm can be complicated and may need to be broken down into several pieces, but at the highest level it's just a proof of a "for all" statement, similar to the ones we've seen in sections 2.3, 2.5 and 3.1. In particular, proving an algorithm means proving that for all

inputs the algorithm produces the correct output. For example, if SORT is an algorithm designed to sort an array of integers (in order of increasing value), then to prove the correctness of SORT we need to prove that:

"For every array A of integers, after calling SORT(A), A is sorted."

Sometimes correctness proofs are simple enough that we don't need any special techniques. Consider the following algorithm which finds the square root of any number whose square root is an integer:

```
IntegerSquareRoot(n):
    i ← 0
    While (i + 1)^2 ≤ n:
        i ← i + 1
    Return i
```

Theorem 14. *For every non-negative integer n such that $\sqrt{n}$ is an integer,* INTEGERSQUAREROOT(n) *returns $\sqrt{n}$.*

Proof. Let $n \in \mathbb{Z}_{\geq 0}$ such that $n = k^2$ for some $k \in \mathbb{Z}_{\geq 0}$. For every $i < k$, $(i+1)^2 \leq k^2 = n$, while if $i = k$ then $(i+1)^2 > k^2$. Thus, the while loop will continue until $i = k$. Thus, when the algorithm returns i it is returning $k = \sqrt{n}$. □

Note that if we want to prove an algorithm correct it is important to be clear about what inputs the algorithm is expected to handle. INTEGERSQUAREROOT would not be an algorithm that correctly returns $\sqrt{n}$ if the input could be any integer (e.g. the algorithm fails to compute $\sqrt{2}$, because $\sqrt{2}$ is not an integer). While data validation is important in real-world programming, the correct way to handle invalid inputs tends to be application specific. In this course, if you are asked to design an algorithm that takes a specific kind of input (e.g. a square number), your algorithm may do whatever it likes when the input is not of the type given in the problem, and you do not need to consider these kinds of inputs when proving the correctness of your algorithm.

For recursive algorithms, induction is often a natural approach to proving correctness (we'll look at this more when we talk about divide and conquer algorithms in Chapter 8). For example, consider the following algorithm:

```
Fact(n):
    If n ≤ 1:
        Return 1
    Return n * Fact(n − 1)
```

Theorem 15. *For every positive integer n,* FACT(n) *returns $n!$ (recall from Section 1.5 that $n!$ is n factorial)*

Proof. We prove this by induction on n:

- Base Case: If $n = 1$ then FACT immediately returns 1, which is 1!
- Induction Hypothesis: Suppose that for every $1 \leq k < n$, FACT(k) returns $k!$.
- Induction Step: Since $n > 1$, FACT(n) returns $n \cdot$ FACT($n-1$). By our induction hypothesis, FACT($n-1$) returns $(n-1)!$. Thus, FACT(n) returns $n(n-1)! = n!$.

□

3.3 Loop Invariants

The algorithms we saw in the previous section were so simple that they arguably don't even need a full proof of correctness (While it's always nice to rigorously prove things, in an exam where functions like INTEGERSQUAREROOT or FACT might be a subroutine in a larger program it would be sufficient to say that their correctness follows from the definition of square root or factorial). Now let's look at an algorithm whose correctness proof is a bit less trivial.

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

Theorem 16. *For every array A of integers, after calling* InsertionSort*(A), A is sorted in increasing order.*

Before proving Theorem 16, let's think a bit about what the right approach is. We need to prove that InsertionSort(A) sorts A. Induction is the right idea here, but we have to be careful about what we induct on and what claim we prove inductively. Unlike in our correctness proof for Fact we can't induct on the input directly because the input to InsertionSort is an array, not an integer. We could try inducting on the length of A, but since InsertionSort doesn't recursively call itself, it's not clear how to use an induction hypothesis about the behavior of InsertionSort on arrays of length $< n$ to say something about the behavior of InsertionSort on arrays of length n.

Instead, the correct approach is to induct on the number of iterations of the outer for loop. Our induction hypothesis can't be that the array is sorted (clearly it need not be sorted after just one iteration of the loop), but that each iteration makes increasing progress towards sorting the array, in such a way that by the end of the n'th iteration, the array is sorted. A claim of the form: "After i iterations of this loop, $P(i)$ is true" (where $P(i)$ is some predicate depending on i) is called a *loop invariant*. Defining a suitable loop invariant and then proving it (usually via induction) is a common technique for proving the correctness of algorithms.

Let's prove the correctness of insertion sort using a loop invariant:

Proof. (of Theorem 16) Let A be an array of integers.

We will prove the following loop invariant for every $k \in \{1, \ldots, n\}$: After k iterations of the outer for loop, $A[1:k]$ is sorted. Note that this invariant implies that after n iterations of the for loop (i.e. at the end of the algorithm), $A[1:n]$ is sorted, meaning the entire array has been sorted. Thus, proving this invariant is sufficient to prove the theorem.

We will prove the invariant by induction on k.

- Base Case: Suppose $k = 1$. The first iteration of the outer loop does nothing (because the inner loop runs 0 times). However, $A[1, \ldots, k] = A[1, \ldots, 1]$ consists of just a single element, so it is automatically sorted.

- Induction Hypothesis: Suppose that for every $1 \leq \ell < k$, we have that $A[1, \ldots, \ell]$ is sorted after ℓ iterations of the outer for loop.

- Induction Step: By our induction hypothesis, we know that at the start of the k'th iteration (i.e. after the $(k-1)$'th iteration) $A[1:k-1]$ is sorted. Let x be the value in $A[k]$ at the beginning of the k'th loop iteration. In the k'th iteration of the outer loop, the inner loop repeatedly swaps x with elements of the sorted subarray $A[1, \ldots, k-1]$, starting from the right, until either it reaches the beginning of the array, or it finds an element which is less than or equal to x.

  If we swap x all the way to the beginning of the array, then we know that x is smaller than every other element of $A[1:k]$ and thus belongs in $A[1]$, where it has ended up. Meanwhile all of the elements $A[1:k-1]$ have shifted right one position, but they remain in the same relative order. By our induction hypothesis they were already sorted, so therefore $A[1:k]$ is now sorted.

  On the other hand if we stop swapping x and break out of the loop at some $j > 1$, then we found some j such that $A[j] \leq x$, and our new array begins $A[1:j], x, A[j+1, k-1]$. Since we broke out of the inner for loop for this value of j, but not the previous one, we have that $A[j] \leq x < A[j+1]$. Furthermore, by our IH, $A[1:j]$ and $A[j+1:k-1]$ were already sorted,

so in the rearranged array we have $A[m] \leq A[m+1]$ for all m from 1 to $k-1$, and $A[1:k]$ is sorted.

□

3.4 Proof by Contradiction

The last type of proof it's important to be familiar with is proof by contradiction. In a proof by contradiction we start by assuming the opposite of what we're trying to prove and then show that this implies some statement which we know is false. Just like with induction, it's important to let the reader know when you're using this proof technique, so a proof by contradiction usually begins with "Assume for the sake of contradiction that [opposite of claim you're trying to prove]". "Assume for sake of contradiction" can be abbreviated as AFSOC.

A proof by contradiction is complete when you have shown that the thing you have assumed implies a false statement. Often this will be a contradiction to what you assumed at the start of the proof (e.g. you assumed a number was odd then proved it was even), but it can also be a regular logical impossibility, like deriving that $1 = 2$.

Proof by contradiction is commonly used when we want to show that a particular object (for example, an algorithm) cannot exist. One of the earliest examples of this kind of proof comes from the Greek mathematician Euclid, over 2000 years ago, who proved that there are infinitely many primes (Euclid's proof was not explicitly a proof by contradiction, but the core idea of it is the same as the proof I present here).

Theorem 17. *There is no largest prime number*

Proof. AFSOC that there is a largest prime number, call it p. Let n be the number of prime numbers less than or equal to p, and call these primes $p_1, \ldots, p_n$, with $p_n = p$. Let $m := \prod_{i=1}^{n} p_i$. By definition, m is divisible by every prime less than or equal to p. Therefore, $m+1$ is not divisible by any of these primes, meaning that either $m+1$ is prime, or it is divisible by some other prime bigger than p. In either case, this contradicts our assumption that p is the largest prime (because $m+1 > m \geq p$). Therefore, there is no largest prime number. □

Here's another classic example of proof by contradiction. Note that the claim "$\sqrt{2}$ is irrational" is actually a claim of the form "an object does not exist" in disguise. If we unpack the definition of a rational number, saying that "$\sqrt{2}$ is not a rational number" means saying that there do not exist two integers whose ratio is $\sqrt{2}$. Therefore we can prove this claim by contradiction if we assume that there are two such integers, and then show that this implies nonsense.

Theorem 18. $\sqrt{2}$ *is irrational.*

Proof. AFSOC that $\sqrt{2} \in \mathbb{Q}$. Then $\exists\, p, q \in \mathbb{Z}$ such that $\sqrt{2} = p/q$ and p and q are relatively prime (i.e. the fraction p/q is in simplest terms). We have that $2 = (p/q)^2 = p^2/q^2$, so $2q^2 = p^2$. Therefore p^2 is even, which means that p must also be even (since squaring an odd number gives an odd number). Thus, $p = 2r$ for some integer r. Since we already concluded that $2q^2 = p^2$, this means we now have $2q^2 = 4r^2$, and thus $q^2 = 2r^2$, so q^2 is also even, and therefore q is even as well. But this contradicts our assumption that p and q were relatively prime! (they share a factor of 2, so p/q can be simplified). We have reached a contradiction, so our initial assumption that $\sqrt{2} \in \mathbb{Q}$ must be false, and $\sqrt{2}$ is irrational. □

Now that we've discussed a bunch of valid proof techniques, I want to leave you with a word of warning about an invalid "proof" technique that students sometimes use. While showing that $\neg P \Rightarrow$ False is a valid way to prove that P is true, showing that $P \Rightarrow$ True does *not* mean that P is true (remember that any implication with a false premise is true). For example, if I start from $1 = 2$, I can multiply both sides by 0 to obtain the true statement $0 = 0$, but clearly this does not imply that $1 = 2$.

3.5 Acknowledgments

Some of the topics and problems in this chapter are inspired by ones in Building Blocks for Theoretical Computer Science by Margaret Fleck, the textbook that I used to teach Discrete Structures at UIUC.

3.6 Comprehension Questions

1. Suppose that we want to prove the claim "For every positive integer n, $n^2 + n$ is even." by induction on n. Which of the following would be an appropriate induction hypothesis to use?

   (a) Assume that for every positive integer k, $k^2 + k$ is even.

   (b) Assume that for every integer $k < n$, $k^2 + k$ is even.

   (c) Assume that for every positive integer $a < n$, $a^2 + a$ is even.

   (d) Assume that $n^2 + n$ is even.

   (e) Assume that for every integer n with $0 \leq n < k$, $n^2 + n$ is even.

2. Given that $2^{20} = 1048576$, what is the value of $\sum_{i=1}^{19} 2^i$?

3. Suppose that we design an algorithm ALG to solve the following problem:

   Problem: Given a positive integer n, determine whether or not n is a power of 2 (i.e. determine whether or not $\log_2 n$ is an integer).

   Which of the following statements would we need to prove in order to show that ALG correctly solves the problem? (select all that apply)

   (a) For every positive integer n which is a power of 2, ALG(n) returns true.

   (b) For every positive integer n which is not a power of 2, ALG(n) returns false.

   (c) For every non-positive integer n, ALG(n) returns false.

   (d) For every non-positive integer n, ALG(n) reports that the input is invalid.

4. Which of the following loop invariants would be most appropriate to use in proving the correctness of the sorting algorithm below?

```
ShiftySort(A[1, ..., n]):
  For i from 1 to n − 1:
      For j from n − i to n − 1:
          If A[j] > A[j + 1]:
              Swap A[j] with A[j + 1]
          Else:
              Break
```

   (a) After k iterations of the outer for loop, $A[1, \ldots, k]$ is sorted.

   (b) After k iterations of the outer for loop, $A[1, \ldots, k + 1]$ is sorted.

   (c) After k iterations of the outer for loop, $A[n - k + 1, \ldots, n]$ is sorted.

   (d) After k iterations of the outer for loop, $A[n - k, \ldots, n]$ is sorted.

   (e) After k iterations of the outer for loop, $A[n - k - 1, \ldots, n]$ is sorted.

5. Find the main flaw in the following proof (it's not enough to give a counterexample to the claim, you should explain why the induction proof fails).

   Claim: All cats are the same color.

   Proof: We will show that for any positive integer n, for every set of n cats, all those cats are the same color. Since the size of the set of all cats on earth is some positive integer n,

this suffices to prove the claim. The proof is by induction on n.

Base Case: $n = 1$. Certainly, in a group of one cat, all cats have the same color (the color of that single cat). In order to have two cats of different colors we need at least two cats.

Induction Hypothesis: For every $k < n$, for every group of k cats, all cats in the group are the same color.

Induction Step: Let S be an arbitrary set of n cats. Let $c \in S$ be an arbitrary cat in the set. We want to show that every other cat in S has the same color as c. Let d be another cat in S. By the induction hypothesis, all of the cats in the set $S - \{c\}$ have the same color. Also by the induction hypothesis, all of the cats in the set $S - \{d\}$ have the same color. Therefore, both c and d are the same color as all of the cats in $S - \{c, d\}$, and therefore c and d are the same color as each other. Since c and d were chosen arbitrarily, this proves that all the cats in S are the same color.

3.7 Exercises

1. Use induction to show that

$$\forall\, n \in \mathbb{Z}^+ \quad \sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

2. Use induction to show that

$$\forall\, n \in \mathbb{Z}^+, \quad \sum_{i=1}^{n} \lfloor \log_7 i \rfloor = \lfloor \log_7 n \rfloor (n+1) - \frac{7^{\lfloor \log_7 n \rfloor + 1} - 7}{6}$$

3. Use induction to show that

$$\forall\, n \in \mathbb{Z}_{\geq 0}, \quad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

4. Prove that $\sqrt{3}$ is irrational.

5. Prove that $\log_2 3$ is irrational.

6. Consider the following algorithm:

```
RPM(x, y)
    r ← 0
    a ← x
    b ← y
    While b > 0:
        If b mod 2 = 1 :
            r ← r + a
        a ← 2a
        b ← ⌊b/2⌋
    Return r
```

Prove that for every $x, y \in \mathbb{Z}_{\geq 0}$, $\mathrm{RPM}(x, y) = xy$, i.e. prove that RPM multiplies its inputs.

7. We define a function $f(n) : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ as follows:

$$\begin{aligned} f(0) &= 2 \\ f(1) &= 7 \\ f(k) &= f(k-1) + 2f(k-2) \qquad\qquad \forall\, k \geq 2 \end{aligned}$$

   Prove that $\forall\, n \in \mathbb{Z}_{\geq 0},\ f(n) = 3 \cdot 2^n + (-1)^{n+1}$

8. Prove that the sorting algorithm below is correct (i.e. that given an array of integers, after running the algorithm the array will be sorted in ascending order). Hint: Start by showing that the algorithm always terminates.

```
ExchangeSort(A[1, ..., n])
    Repeat forever:
        i ← −1
        For j from 1 to n − 1:
            If A[j] > A[j + 1]:
                i ← j
        If i = −1:
            //A is now sorted
            Return
        Swap A[i] with A[i + 1]
```