Chapter 1

Reasoning About Algorithms

An *algorithm* takes some type of input and applies a sequence of steps in order to produce some type of output. In general, these inputs and outputs can take many forms. For example, input can come from the keyboard, a mouse's movements or a sensor that is reading the temperature. Outputs might be shapes drawn on a screen, characters sent to a printer, or movements of a robot's arm. For now, we will focus on *offline* algorithms, where all of the input is available when the algorithm begins running, and the algorithm is intended to produce some output and terminate. *Online* algorithms, where there are constant streams of inputs and outputs, are also fascinating objects to study but they are more complicated, so we start with offline algorithms. Many of the techniques we use in analyzing offline algorithms can be generalized to deal with online algorithms too. For example, an online problem can often be analyzed as an infinite sequence of steps, where each step is accomplished by an offline algorithm.

For the offline algorithms we study, the inputs and outputs are finite, but usually unbounded. This means that any individual input given to the algorithm can be represented as a finite string of characters (chosen from a finite alphabet), but we put no limit on the length of the string. Thus, there are usually infinitely many possible inputs, but each possible input is of finite size. Similarly, the outputs produced by the algorithms are finite but unbounded.

A *problem specification* describes

1. which inputs are permitted, and

2. which outputs are acceptable for each possible input.

These two parts of the problem specification are called the *preconditions* and *postconditions* of the problem, respectively. When you design an algorithm to solve the problem, the preconditions are statements that you can assume are

true before the algorithm is run, and the postconditions are statements that ought to be true when (and if) the algorithm terminates.

Example 1 *Addition of two natural numbers.*
Precondition: the input must be a pair of natural numbers x and y.
Postcondition: the output must be the integer $x + y$.
(To be more precise, we might also specify the representation used for the integers. For example, we might say that they must be represented in binary with no leading 0's.)

Example 2 *Finding the maximum element in an array of integers.*
Precondition: the input is an array $A[1..n]$ of integers (where $n \geq 1$).
Postcondition: the output must be some integer $i \in \{1, \ldots, n\}$ such that for all $j \in \{1, \ldots, n\}$, $A[i] \geq A[j]$.
Notice that there may be several different acceptable outputs for a single input if the maximum value occurs several times in the array.

An algorithm is a *correct* solution to a problem if, for every possible input, it produces an output and the output produced is acceptable, according to the specification of the problem.

After spending some time learning about algorithms and designing your own, you have probably realized the following tasks can be difficult.

- designing a correct algorithm for a problem
- convincing yourself that an algorithm is correct
- convincing someone else that your algorithm is correct

As problems become larger and more complex, these tasks can become extremely difficult, so we need some tools to tackle them systematically. You might try accomplishing these tasks by running the algorithm on a number of test inputs and seeing whether the output produced is legal. If you get an incorrect result, you know the algorithm is incorrect. But what if the algorithm produces correct outputs for all of the test cases? This may increase your confidence that the algorithm is correct. However, most interesting problems have infinitely many possible inputs, so this technique is quite useless in showing an algorithm actually *is* correct. (Remember: being correct means that it produces a correct output for *every* possible input.) Thus, testing is a good way to discover that your algorithm is incorrect, but it usually cannot establish that the algorithm is correct.

Suppose we eventually determine (somehow) that an algorithm correctly solves a problem. Then, we may also have to decide whether it is feasible to implement it. For example, if the algorithm will take 100 years to run on your computer before producing an output, it is probably useless to you. Similarly, if the algorithm will require more memory space than you have on your computer, you will not be able to use it. To determine whether an algorithm is feasible to implement, or to compare two (correct) algorithms to see which one is preferable,

we would like to study the *efficiency* of algorithms. In other words, we would like to quantify the resource usage of an algorithm (where the resource might be computation time, memory space, battery usage in a mobile device, or some other quantity we are interested in). We could try running the algorithm on some sample inputs and measuring its resource usage. But again, if there are infinitely many possible inputs, this approach can tell us that the algorithm is infeasible (for example, if we get an "out of memory" error), but is not useful for establishing that the algorithm will have reasonable resource requirements for *all* possible inputs.

We have seen that testing is ultimately useless for providing guarantees that an algorithm is correct or efficient whenever the number of possible inputs is infinite. So what can we do? Fortunately, there is a branch of human knowledge that routinely reasons about infinitely many cases at once: mathematics. Mathematicians spend all of their time proving statements like the following.

"Every even integer can be written as the sum of two odd integers."

"Every right triangle with side lengths $a < b < c$ has $c^2 = a^2 + b^2$."

Mathematicians would never try to prove the latter statement by testing every right triangle one by one, because this would be an endless task. Instead, mathematicians have developed numerous technique for proving such universally quantified statements are true. The kinds of statements we want to prove are also universally quantified:

"Every input, when given to our algorithm, will cause the algorithm to produce a correct output."

"Every input of size n, when given to our algorithm, will be handled using at most $14n^2$ steps."

Thus, we can use the same techniques that mathematicians have developed over the past 2000+ years to reason about algorithm correctness and efficiency.

The rest of this chapter will discuss how to prove, using techniques from mathematics, that an algorithm is correct. We shall come back to the question of quantifying the resource usage of algorithms later.

1.1 Approaching a Problem

Beginners often approach an algorithmic problem by sitting down and starting to type code. This is the slowest way to solve a non-trivial problem, because you can waste a lot of time before discovering that the approach you are using cannot be completed, is incorrect, is way too slow or has some other essential flaw. If any of these problems occur, it is often necessary to throw away the code (or large parts of it) and start over. If you are fortunate enough to get a complete programme that seems to work when you test it, it will probably be extremely ugly and it will likely be difficult to convince yourself or others that it really is correct.

A better approach is to figure out a fairly complete outline of the algorithm (including what data structures to use, a high-level description of the steps needed, etc.) before writing the (beautiful, well-documented) code. The first

step of designing the algorithm is often very difficult and can require a lot of work before you ever write a single line of code. But this approach is usually much faster overall, and leads to more elegant programmes. After the algorithm is designed, one can check that the programme really is correct, perhaps by trying some test cases first to debug it, and then proving that it is correct.

An even better approach is similar to the one described above, but will instead involve designing the algorithm and its proof of correctness *simultaneously*. This approach allows you to clarify the goals of each piece of the algorithm straightaway, and makes it easier to avoid mistakes in the design. Throughout this course, we shall develop tools to help you use this approach of designing an algorithm in conjunction with the proof of correctness.

1.2 An Example

Consider the following problem.
Input: a positive integer n.
Output: "yes" if n can be expressed as the difference of two perfect squares, and no otherwise.
In other words, given n, we want to determine whether there are two integers a and b such that $n = a^2 - b^2$. For example, the answer should be "yes" for the input $n = 12$ because $12 = 4^2 - 2^2$.

Idea 1: Loop through all possible values of a and b and check whether $a^2 - b^2 = n$.

Unfortunately this is not so easy because there are infinitely many values of a and b that could be tried. Unless we can limit the range of a and b, this approach is hopeless.

Idea 2: Try a few random small values of n to get some intuition: $n = 8, 9, 15, 20, 33$.

$$\begin{aligned} 8 &= 3^2 - 1^2 \\ 9 &= 5^2 - 4^2 \\ 15 &= 8^2 - 7^2 \\ 20 &= 6^2 - 4^2 \\ 33 &= 7^2 - 4^2 \end{aligned}$$

Maybe the answer is always "yes" for every n.

No, it turns out that 6 cannot be expressed as the difference of two squares (as we shall soon see).

Idea 3: $a^2 - b^2 = (a + b)(a - b)$. Now, instead of expressing n as the difference of two numbers, we have to express it as the product of two numbers. Maybe this way of viewing the problem will make it easier.

Combining Idea 1 and Idea 3 does yield an algorithm. If a and b are integers, then so are $a+b$ and $a-b$, so we can just check all possible ways of factoring n into

two integers x and y, and see if it is possible to express x and y as $a+b$ and $a-b$ for some integers a and b. For example, for $n = 8$, there are four possible ways to express n as the product of two integers: $8 = 1{\cdot}8 = 2{\cdot}4 = (-1)(-8) = (-2)(-4)$. Some of these factor pairs cannot be expressed as $a + b$ and $a - b$ for integers a and b. For example, if we want $a + b = 1$ and $a - b = 8$, we must have $a = \frac{9}{2}$ and $b = -\frac{7}{2}$, which are not integers. But when we try using the pair of factors $8 = (-2)(-4)$, we find that $a = -3$ and $b = -1$ works; thus, $8 = (-3)^2 - (-1)^2$. This approach gives us the following algorithm.

EXPRESSIBLE(n)
  Precondition: n is a positive integer
  Postcondition: outputs "yes" if $n = a^2 - b^2$ for some integers a, b and "no" otherwise
  for $x = -n..n$ except 0 // try all possible integer factorizations $n = x \cdot \frac{n}{x}$
    // solve the linear equations $a + b = x$ and $a - b = \frac{n}{x}$ for a and b:
    $a = \frac{x+\frac{n}{x}}{2}$
    $b = \frac{x-\frac{n}{x}}{2}$
    if a and b are integers output "yes" and halt
  end for
  output "no"
end EXPRESSIBLE

This algorithm is correct, but not very fast. Next, we shall use one more observation to help us find a much simpler algorithm (and prove that the simpler algorithm is correct).

Idea 4: $a + b$ and $a - b$ always have the same parity[1]. (If a and b have the same parity, then $a + b$ and $a - b$ are both even. If a and b have opposite parities, then $a + b$ and $a - b$ are both odd.)

So if we can express n as $a^2 - b^2 = (a + b)(a - b)$, then we know n is the product of two numbers with the same parity. The converse of this statement is also true.

Proposition 3 *A positive integer n is expressible as $a^2 - b^2$ iff n is the product of two integers with the same parity.*

Proof: We prove each direction separately.

($\Rightarrow$): Assume $n = a^2 - b^2$ for some integers a and b. Then, $n = (a+b)(a-b)$ where $a + b$ and $a - b$ are integers with the same parity.

($\Leftarrow$): Assume $n = xy$ where x and y are two integers with the same parity. We want to find integers a, b such that $x = a + b$ and $y = a - b$. Solving these linear equations for a and b, we get $a = \frac{x+y}{2}$ and $b = \frac{x-y}{2}$. Note that both a and b are integers since x and y have the same parity. ■

Now, how do we determine whether n can be expressed as the product of two integers with the same parity? Since parity seems to be important in this problem, let's consider cases according to the parity of n.

[1]The parity of an integer says whether the number is even or odd

If n is odd, it is fairly easy to see that we can choose $x = n$ and $y = 1$. Then $n = x \cdot y$, where x and y are both odd, so the answer should be "yes".

Now suppose n is even. If $n = x \cdot y$ where x and y have the same parity, then x and y must both be even (since the product of two odd numbers could not be even), so n must be a multiple of 4. Conversely, if n is a multiple of 4, we can choose $x = \frac{n}{2}$ and $y = 2$, which are both even. Thus, for even values of n, the answer should be yes if and only if n is a multiple of 4.

This reasoning gives us a simple algorithm to solve the problem:

```
EXPRESSIBLE(n)
    Precondition: n is a positive integer
    Postcondition: outputs "yes" if n = a^2 - b^2 for some integers a, b and "no" otherwise
    if n is odd then output "yes"
    else if n is divisible by 4 then output "yes"
    else output "no"
    end if
end EXPRESSIBLE
```

The proof that this algorithm is correct is already embedded in the reasoning that we used to build the algorithm.

Proof of Correctness: But let's recap it as a formal proof. It is trivial to see that the algorithm always terminates and outputs an answer. Since the structure of the algorithm is an if statement with three cases, it is natural to structure the proof that the output is correct as a proof by cases.

Case 1 (n is odd): The algorithm outputs "yes" in this case, so we must show that n is expressible as $a^2 - b^2$ for some integers a, b. Let $a = \frac{n+1}{2}$ and $b = \frac{n-1}{2}$. (These are the values we get from the equations above when we take $x = n$ and $y = 1$). Notice that a and b are both integers since n is odd. Then,

$$\begin{aligned} a^2 - b^2 &= \left(\frac{n+1}{2}\right)^2 - \left(\frac{n-1}{2}\right)^2 \\ &= \frac{n^2 + 2n + 1 - n^2 + 2n - 1}{4} \\ &= n \end{aligned}$$

Thus, the algorithm outputs the correct answer in this case.

Case 2 (n is divisible by 4): The algorithm outputs "yes" in tihs case, so we must show that $n = a^2 - b^2$ for some integers a, b. Let $a = \frac{\frac{n}{2}+2}{2}$ and $b = \frac{\frac{n}{2}-2}{2}$. (These are the values we get from the equations above when we take $x = \frac{n}{2}$ and $y = 2$). Notice that a and b are both integers since $\frac{n}{2}$ and 2 are both even. Then,

$$\begin{aligned} a^2 - b^2 &= \left(\frac{\frac{n}{2}+2}{2}\right)^2 - \left(\frac{\frac{n}{2}-2}{2}\right)^2 \\ &= \frac{\frac{n^2}{4} + 2n + 4 - \frac{n^2}{4} + 2n - 4}{4} \\ &= n \end{aligned}$$

Thus, the algorithm outputs the correct answer in this case.

Case 3 (otherwise): We know that n is even but not divisible by 4. The algorithm outputs "no" in this case, so we must show there do *not* exist integers a, b such that $n = a^2 - b^2$. Showing directly that things do not exist is often difficult. As a rule of thumb, it is helpful to use a proof by contradiction to show something does not exist.

To derive a contradiction, assume there are integers a and b such that $n = a^2 - b^2 = (a + b)(a - b)$. Since n is even, either $a + b$ or $a - b$ must be even. So both $a + b$ and $a - b$ must be even (since $a + b$ and $a - b$ always have the same parity). Thus, there exist integers c and d such that $a + b = 2c$ and $a - b = 2d$. Then, $n = (a + b)(a - b) = 2c \cdot 2d = 4cd$ is divisible by 4, contradicting the fact that n is not divisible by 4. Thus, there *cannot* exist integers a, b such that $n = a^2 - b^2$. ■

1.3 Loops

The algorithm that we proved correct in the preceding section had no loops or recursion, so it was pretty straightforward to see what it did. Reasoning about the correctness of programmes that contain loops or recursion is often more difficult. Fortunately, there are some standard techniques that help us, which we shall consider next.

Because a loop can be iterated many times and sometimes the number of iterations is not even known in advance, it is difficult to reason about what is being achieved by all the iterations of the loop. Instead, it is much easier to focus on what a single iteration of the loop does. But we do not want to think only about the first iteration of the loop; we want to think about some arbitrary iteration in the middle of the execution of the programme. In order to figure out what is happening in that iteration, it is helpful to think about what state the memory of the machine is in at the beginning of the loop. (This is like a precondition for a single iteration of the loop.) We formalize our knowledge about the state of the memory at the beginning of the loop by stating a *loop invariant*.

A loop invariant is simply a statement that is true at the beginning of each iteration of the loop. However, deciding what loop invariant to use to prove that a loop accomplishes some task is actually quite tricky. The key idea is that the loop invariant should summarize what has been accomplished by all previous iterations of the loop. The key word is *summarize*: you do not want the loop invariant to be so detailed that you have to think in detail about all previous iterations of the loop; remember that our goal was to think only about a single iteration of the loop. However, it must contain all the crucial information about what has been done so far by the previous iterations of the loop. That is, it must contain all information that we shall need in order to see that the next iteration does what it is supposed to do, and it must allow us to prove that when the loop terminates (if it does), the postconditions of the loop are true.

Whenever you state a loop invariant, you are making a claim that it is true

at the beginning of each iteration of a loop. So, you must prove that your claim is correct. Ordinarily, this is done using mathematical induction. For the base case, you prove that the invariant is true at the beginning of the first iteration of the loop. For the induction step, you assume that the invariant is true at the beginning of some iteration of the loop, and you show that it is true at the beginning of the next iteration.

Coming up with good loop invariants is hard. Once you get more practice, it will become easier. At first, you will often find that when you try to do the induction step, there will be some piece of information missing from your invariant, so you will have to go back and change the invariant, and then try the proof again. You may have to do this a few times before you can make the proof work. Later, with more experience, you will find that you can do a lot of the necessary adjustments to the loop invariant by just thinking through the inductive step in your head.

Suppose you have a typical loop:

assert Pre
loop
    invariant: I
    exit when C
    B
end loop
assert $Post$

Here, B represents a chunk of code that forms the body of the loop, and Pre, I, C and $Post$ represent statements that evaluate to true or false. (Pre is the precondition that must be true before the loop begins, I is the invariant that you choose for the loop, C is the exit condition for the loop and $Post$ is the postcondition that you want to ensure is satisfied when the loop terminates.)

To use your invariant to prove that the loop functions correctly, you should prove the following four things.

1. $Pre \Rightarrow I$: This says that if the preconditions are satisfied, then the invariant will be true at the beginning of the first iteration of the loop. This is the base case of the inductive proof that the invariant is true at the beginning of each loop iteration.

2. If $I \wedge \neg C$ is true and then the body of the loop B is executed, then I will be true again. This is the inductive step of the proof that the invariant is true at the beginning of each loop iteration. (Note that we are allowed to use the fact that the exit condition C of the loop is not satisfied in order to prove the induction step, because if C were true, there would be no iteration of the loop.)

3. $I \wedge C \Rightarrow Post$: This says that *if* the loop terminates (because C is satisfied), then the postconditions are satisfied.

4. The loop must eventually terminate.

Note that the first two items form a proof by induction that I is true at the start of every iteration of the loop.

1.3.1 Example: Binary Search

The search problem in a sorted array is defined as follows.

Precondition: Array $A[1..n]$ contains integers with $n \geq 1$, and $A[i] \leq A[i+1]$ for $1 \leq i < n$ and an integer key k.

Postcondition: If k appears in $A[1..n]$, output an index i such that $A[i] = k$; otherwise output "not found"

We can write the binary search algorithm as a loop. At each iteration, we have narrowed down the set of possible locations for the desired key to some smaller chunk of the array $A[lo..hi]$, where $1 \leq lo \leq hi \leq n$. At each iteration, we try to cut the size of this chunk roughly in half by seeing whether the desired key is in the right or left half of the current chunk. The essential invariant to maintain is that we do not eliminate portions of the array that contains (all copies of) the key k.

BinSearch($A[1..n], k$)
Precondition: $A[i] \in \mathbb{Z}$ for all i, $n \geq 1$, and $A[i] \leq A[i+1]$ for $1 \leq i < n$, $k \in \mathbb{Z}$.
Postcondition: If k appears in $A[1..n]$, output an i such that $A[i] = k$; otherwise output "not found"
$lo = 1$
$hi = n$
loop
invariant: (1) $1 \leq lo \leq hi \leq n$
(2) $A[1..lo-1]$ does not contain k
(3) if $A[hi] \neq k$ then $A[hi+1..n]$ does not contain k.
exit when $lo = hi$
$mid = \lfloor \frac{lo+hi}{2} \rfloor$
if $A[mid] \geq k$ then $hi = mid$
else $lo = mid + 1$
end if
end loop
if $A[lo] = k$ then output lo
else output "not found"
end if
end BinSearch

In the following argument we use lo_ℓ, mid_ℓ and hi_ℓ to represent the value of the variables lo, mid and hi after ℓ iterations of the loop.

Claim 4 *The invariants (1), (2), (3) are true at the start of each iteration of the loop.*

Proof: We prove this by induction.

Base Case Initially, $lo = 1$ and $hi = n$, so $1 \leq lo \leq hi \leq n$ (since $n \geq 1$ according to the precondition). Also, $A[1..lo-1]$ and $A[hi+1..n]$ contain no elements, so they do not contain k.

Inductive Step Let $\ell \geq 1$. Assume the invariants are true after $\ell - 1$ iterations. We prove that the invariants are true after ℓ iterations. Consider the ℓth iteration. Since the code for the body of the loop is divided into two cases by the if statement, it is natural to write the proof of the inductive step in this way too.

Case 1 ($A[mid_\ell] \geq k$): Then we have $lo_\ell = lo_{\ell-1}$ and $hi_\ell = mid_\ell = \left\lfloor \frac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor$. We prove each of the three parts of the invariant.

(1):

$$\begin{aligned} 1 &\leq lo_{\ell-1} && \text{(by part (1) of induction hypothesis)} \\ &= lo_\ell \end{aligned}$$

$$\begin{aligned} lo_\ell &= \tfrac{lo_{\ell-1}+lo_{\ell-1}}{2} \\ &\leq \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} && \text{(by part (1) of induction hypothesis)} \\ \Rightarrow lo_\ell &\leq \left\lfloor \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor && \text{(since } lo_\ell \text{ is an integer)} \\ &= hi_\ell \end{aligned}$$

$$\begin{aligned} hi_\ell &= \left\lfloor \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor \\ &\leq \left\lfloor \tfrac{hi_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor && \text{(by part (1) of induction hypothesis)} \\ &= hi_{\ell-1} \\ &\leq n && \text{(by part (1) of induction hypothesis)} \end{aligned}$$

(2): By part (2) of the induction hypothesis, $A[1..lo_{\ell-1}-1]$ does not contain k. But $lo_\ell = lo_{\ell-1}$, so $A[1..lo_\ell - 1]$ does not contain k.

(3): We have $A[hi_\ell] = A[mid_\ell] \geq k$ and A is sorted. If $A[hi_\ell] = k$, then (3) is trivially satisfied[2]. If $A[hi_\ell] > k$, then all elements of $A[hi_\ell..n]$ are greater than k, so (3) is true after ℓ iterations.

Case 2 ($A[mid_\ell] < k$): Then we have $lo_\ell = mid_\ell + 1 = \left\lfloor \frac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor + 1$ and $hi_\ell = hi_{\ell-1}$. We prove each of the three parts of the invariant.

(1) Note that, because the exit condition is not satisfied after $\ell-1$ iterations, $lo_{\ell-1} \neq hi_{\ell-1}$. By part (1) of the induction hypothesis, $lo_{\ell-1} \leq hi_{\ell-1}$, so $lo_{\ell-1} < hi_{\ell-1}$.

[2]This is the case where we need the phrase "if $A[hi] \neq k$" in invariant (3): if $A[mid_\ell] = k$, we may throw away a portion of the array that contains copies of k, but that's okay because we still have at least one copy of k inside $A[lo..hi]$.

$$\begin{aligned} lo_\ell &= \left\lfloor \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor + 1 \\ &\geq \left\lfloor \tfrac{1+1}{2} \right\rfloor + 1 && \text{(by part (1) of induction hypothesis)} \\ &> 1 \end{aligned}$$

$$\begin{aligned} lo_\ell &= \left\lfloor \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor + 1 \\ &\leq \left\lfloor \tfrac{hi_{\ell-1}-1+hi_{\ell-1}}{2} \right\rfloor + 1 && \text{(since } lo_{\ell-1} < hi_{\ell-1}\text{)} \\ &= hi_{\ell-1} - 1 + 1 \\ &= hi_\ell \end{aligned}$$

$$\begin{aligned} hi_\ell &= hi_{\ell-1} \\ &\leq n && \text{(by part (1) of induction hypothesis)} \end{aligned}$$

(2): We have $A[lo_\ell - 1] = A[mid_\ell] < k$. Since A is sorted, all elements of $A[1..lo_\ell - 1]$ are less than k.

(3): Invariant (3) is identical to part (3) of the induction hypothesis, since $hi_\ell = hi_{\ell-1}$. ■

Claim 5 *If the loop terminates, then the postconditions are satisfied.*

Proof: When the loop terminates, the invariants are true and $lo = hi$.

If $A[hi] = k$, the algorithm outputs hi and this clearly satisfies the postconditions.

If $A[hi] \neq k$, the algorithm outputs "not found", so we have to argue that k does not appear anywhere in the array $A[1..n]$. Since $lo = hi$ we know that $A[1..hi - 1]$ does not contain k, by invariant (2). By invariant (3), $A[hi + 1..n]$ does not contain k. Thus, k is not in $A[1..n]$. ■

Claim 6 *The loop must eventually terminate.*

Proof: To prove this, we define $hi - lo$ to be a measure of how far the algorithm has progressed. This quantity is always a non-negative integer (since values assigned to hi, lo and mid are always integers and we proved that $hi \geq lo$) and we shall show that it decreases at each iteration. Thus it must eventually reach 0 and the loop will then terminate.

To see that the value of $hi - lo$ decreases during the ℓth loop iteration, we consider two cases. Notice that $hi_{\ell-1} - lo_{\ell-1} \geq 0$ by invariant (1) and $hi_{\ell-1} - lo_{\ell-1} \neq 0$; otherwise there would be no ℓth iteration. So, $hi_{\ell-1} - lo_{\ell-1} > 0$.

Case 1 ($A[mid] \geq k$): Then,

$$\begin{aligned} hi_\ell - lo_\ell &= \left\lfloor \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor - lo_{\ell-1} \\ &\leq \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} - lo_{\ell-1} \\ &= \tfrac{hi_{\ell-1}-lo_{\ell-1}}{2} \\ &< hi_{\ell-1} - lo_{\ell-1} && \text{(since } hi_{\ell-1} - lo_{\ell-1} > 0\text{).} \end{aligned}$$

Case 2 ($A[mid] < k$): Then,

$$\begin{aligned} hi_\ell - lo_\ell &= hi_{\ell-1} - (\left\lfloor \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} \right\rfloor + 1) \\ &< hi_{\ell-1} - \tfrac{lo_{\ell-1}+hi_{\ell-1}}{2} + 1 - 1 \quad (\text{since } -\lfloor x \rfloor < -x+1) \\ &= \tfrac{hi_{\ell-1}-lo_{\ell-1}}{2} \\ &< hi_{\ell-1} - lo_{\ell-1} \quad (\text{since } hi_{\ell-1} - lo_{\ell-1} > 0). \end{aligned}$$

■

This completes the detailed proof that the binary search algorithm is correct.

1.3.2 Example with a Counted Loop: Computing Factorials

The factorial is a unary postfix operator ! on non-negative integers defined recursively as follows.

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)!, \text{ for } n > 0 \end{aligned}$$

The following code computes $n!$. In stating the invariant, we have used the notation $result_k$ to indicate the value of the *result* variable after k iterations of the loop.

```
FACTORIAL(n)
    Precondition: n is a non-negative integer
    Postcondition: output value is n!
    result = 1
    for i = 1..n
        invariant: result_k = k!
        result = i · result
    end for
    output result
end FACTORIAL
```

In this case, termination is trivial, since the loop is only performed n times. We establish the loop invariant using a proof by induction.

Base Case ($k = 0$): Initially, $result_0 = 1 = 0!$.

Induction Step: Let $k > 1$. Assume the invariant is true after $k-1$ iterations, i.e., assume $result_{k-1} = (k-1)!$. Then we have

$$\begin{aligned} result_k &= k \cdot result_{k-1} && \text{(according to code of the } k\text{th iteration)} \\ &= k \cdot (k-1)! && \text{(by induction hypothesis)} \\ &= k! && \text{(by definition of } k!) \end{aligned}$$

So, the invariant is true after k iterations.

This completes the inductive proof that the invariant holds after each iteration. Thus, after all n iterations, the value of the result variable will be $result_n = n!$, as required to satisfy the postconditions.