APPENDIX F

Loop Invariants

F.1 WHY YOU SHOULD CARE

When you write a function to search or sort a list, how do you know that it will work on all valid inputs? You certainly can't test it on every possible input. In computer science, we prove correctness of our algorithms. A common technique for proving correctness of algorithms with loops is the loop invariant proof. Read on to learn more!

F.2 SUMMING A LIST

Consider the following algorithm (Algorithm F.1) for summing up a list of numbers:

Algorithm F.1: sum(L)

```
input  : A list of numbers $L$ of size $|L|$
output: The sum of the elements of $L$
1: $total = 0$
2: for $i = 0, \ldots, |L| - 1$ do
3:     $total = total + L[i]$
4: return $total$
```

This is a very simple algorithm, but it makes a great first example to see how we can prove that it always works. To prove that the sum algorithm is correct, we introduce a loop invariant.

Definition F.1

A loop invariant is a statement that is true prior to the start of a loop and remains true at the start of each iteration of the loop. Combined with the termination condition of the loop, it tells us that the algorithm did in fact solve the problem correctly.

A loop invariant for the sum problem is given below. Note that for the rest of this chapter we will use the notation $L[i : j]$ to indicate the sub-list consisting of

$L[i], L[i+1], \ldots, L[j-1]$ (i.e., starting from the ith element up to but not including the jth element), similar to the notation used by Python for list slices.

Example F.1

Loop invariant: At the start of iteration i, *total* is equal to the sum of the list $L[0:i]$.

We use loop invariants such as this one to essentially write a proof by induction to show correctness of the algorithm. We start with the loop initialization (similar to the base step) and then show that the invariant is maintained (similar to the inductive step) after each iteration. Unlike a proof by induction that shows the result for all positive n, we terminate our proof based on the termination condition of the loop.

Example F.2

At the **initialization** step, we consider what happens when $i = 0$. At this stage we don't have any elements in our list since we don't include the $i = 0$ element. The sum of an empty list is always zero and we correctly have *total* equal to zero at initialization on Line 1.

Example F.3

At the **maintenance** step, we have to show that each iteration makes progress toward our goal. Similar to induction, we will do this by proving that if the loop invariant holds at the start of the jth iteration, it will also hold after the $(j+1)$st iteration.
We assume that the loop invariant holds for some iteration $j \geq 0$. That is, at the start of iteration j, *total* holds the sum of $L[0:j]$. During iteration j, we only add $L[j]$ into *total* on Line 3. Thus, at the start of iteration $(j+1)$, we have that *total* is equal to the sum of $L[0:j+1]$, as desired.

Example F.4

In the **termination** step, we have to figure out when the loop will terminate and then apply that termination value to the loop invariant to show that the loop accomplished what we set out to do.
The loop for the sum algorithm will sum indexes up through $|L|-1$ and will thus terminate when $i = |L|$. Substituting this into the loop invariant, we get that at the termination of the loop of the sum function, we have that *total* is equal to the sum of the list $L[0:|L|]$, which is precisely the entire list. Thus, at the end of the loop, *total* holds the sum of the entire list L.

Every loop invariant proof needs all of the above four parts: statement of the loop invariant, initialization, maintenance, and termination. We'll see some more examples next.

F.3 EXPONENTIATION

Consider the standard algorithm for exponentiation (Algorithm F.2) below. It very straightforwardly just uses an accumulator to compute the exponent. Think about what would be the loop invariant in this case.

Algorithm F.2: $\exp(a, n)$

```
input  : A real number $a$ and a positive integer $n$
output: The value $a^n$
1: $result = 1$
2: for $i = 1, \ldots, n$ do
3:     $result = result * a$
4: return $result$
```

The invariant involves the *result* variable that starts at 1 and should end at a^n. At each iteration, the value of *result* is multiplied by a. Guided by this, we propose the following invariant:

Example F.5

Loop invariant: At the start of iteration i, *result* is equal to a^{i-1}.

The rest of the proof looks as follows.

Example F.6

At the **initialization** step, before the start of the loop, the value of *result* is initialized to be 1 on Line 1. We can verify that before the start of the first iteration (when $i = 1$), *result* is indeed equal to $a^{i-1} = a^{1-1} = a^0 = 1$.

Example F.7

For the **maintenance** step, we assume that the invariant holds for some iteration $j \geq 1$. That is, at the start of iteration j we have that $result = a^{j-1}$. During iteration j, we multiply *result* by a on Line 3, so its new value will be $a^j = a^{(j+1)-1}$ at the start of iteration $(j + 1)$, as desired.

Example F.8

In the **termination** step, we see that the loop terminates when $i = n + 1$. Substituting this into the invariant, we get that at the termination of the loop the value of *result* is $a^{(n+1)-1} = a^n$, as desired. Thus, the function does correctly exponentiate a^n.

The correctness of the previous algorithm might have looked obvious to you, so let us see how we can write a proof for a more interesting exponentiation algorithm

(Algorithm F.3). It will also show us an example that has a `while` loop rather than a `for` loop.

Algorithm F.3: fastexp(a, n)

```
input  : A real number $a$ and a positive integer $n$
output: The value $a^n$
 1: $result = 1$
 2: $b = a$
 3: $e = n$
 4: while $e > 0$ do
 5:    if $e$ is odd then
 6:       $result = result * b$
 7:       $e = e - 1$
 8:    else
 9:       $b = b * b$
10:       $e = e/2$
11: return $result$
```

This algorithm works similarly to the previous one, accumulating the product into an accumulator variable, except that if the exponent is even it performs the following speedup: it squares the base and halves the exponent. This considerably decreases the number of multiplications needed since we often halve the exponent, earning the algorithm the name **fast exponentiation**.

Example F.9

The following is a trace of the variables at the end of each iteration when computing the value fastexp$(2, 31)$:

iteration	0	1	2	3	4	5	6	7	8	9
result	1	2	2	2^3	2^3	2^7	2^7	2^{15}	2^{15}	2^{31}
b	2	2	2^2	2^2	2^4	2^4	2^8	2^8	2^{16}	2^{16}
e	31	30	15	14	7	6	3	2	1	0

At the start of the while loop (iteration 0), the values of *result*, b, and e are the initialized values 1, 2, 31, respectively. Whenever we encounter an odd exponent e (at the end of iterations 0, 2, 4, 6, and 8), we simply multiply the base b into *result* and decrement the exponent by one. However, when we encounter an even exponent (at the end of iterations 1, 3, 5, and 7) we shorten the rest of the computation by halving the exponent and squaring the base since $b^e = (b^2)^{e/2}$.

To prove correctness of the algorithm, we need to find a pattern in the values of *result*, b, and e. Notice that at every iteration the remaining product that needs to be multiplied into the *result* variable is b^e. After some thought, you might get the following invariant:

Example F.10

Loop invariant: At the start of any iteration, $result \cdot b^e = a^n$.

For example, in the table in the previous example, you should verify that $result \cdot b^e$ is always equal to 2^{31}. Notice that this invariant doesn't have an iteration number as we had in the previous `for` loop examples.

Example F.11

At the **initialization** step, before the start of the loop (Lines 1-3), the value for $result$, b, and e are initialized to 1, a, and n respectively. It is easy to see that $result \cdot b^e = 1 \cdot a^n = a^n$.

Example F.12

For the **maintenance** step, we assume that the invariant holds true for a given iteration and show that it is still true in the next iteration. That is, we assume that at the start of an iteration we have $result \cdot b^e = a^n$. We now have two cases: when e is odd and when e is even.
If e is odd (Lines 6-7), we multiply $result$ by b and decrement e by 1. This causes $result$ to go up by a factor of b and b^e to go down by a factor of b. These cancel each other out and we end up with $result \cdot b^e$ being unchanged (still a^n) as desired.
If e is even (Lines 9-10), we square b and halve e. Since $b^e = (b^2)^{e/2}$, this means that b^e and thus $result \cdot b^e$ remains unchanged (still a^n) as needed.
Thus, the invariant holds true at the start of the next iteration.

Example F.13

In the **termination** step, we see that the `while` loop terminates when $e = 0$ (Line 4). Substituting this into the invariant, we get that $result \cdot b^0 = a^n$ or that $result = a^n$, as we wanted to show.

F.4 INSERTION SORT

The following Insertion Sort algorithm (Algorithm F.4) sorts a list of comparable types (i.e., types that have the > operation defined). It works much in the same way that most people sort playing cards in their hands: as we encounter a card (starting from the left), we insert that card in the correct position for the cards encountered thus far. More precisely, during each iteration i, the algorithm inserts the ith element in the correct position for the sublist $L[0 : i + 1]$. At each successive iteration a larger sublist of the list is sorted. Eventually, when all the items in the list have been encountered, the entire list will be sorted.

Algorithm F.4: InsertionSort(L)

input : A list L of size $|L|$ with comparable elements
output: The list L sorted

```
1: for $i = 1, \ldots, |L| - 1$ do
2:     $value = L[i]$
3:     $j = i - 1$
4:     while $j \geq 0$ and $L[j] > value$ do
5:         $L[j+1] = L[j]$
6:         $j = j - 1$
7:     $L[j+1] = value$
8: return $L$
```

Example F.14

Consider how Insertion Sort works on a list $[3, 1, 4, 1, 5]$ after each iteration i:

- $(i = 1)$ $[\mathbf{1}, \mathbf{3}, 4, 1, 5]$ The 1 at $L[1]$ gets inserted before the 3.
- $(i = 2)$ $[\mathbf{1}, \mathbf{3}, \mathbf{4}, 1, 5]$ The 4 at $L[2]$ gets inserted at its current position.
- $(i = 3)$ $[\mathbf{1}, \mathbf{1}, \mathbf{3}, \mathbf{4}$,5$]$ The 1 at $L[3]$ gets inserted at $L[1]$.
- $(i = 4)$ $[\mathbf{1}, \mathbf{1}, \mathbf{3}, \mathbf{4}, \mathbf{5}]$ The 5 at $L[4]$ gets inserted at its current position.

Note that as we go further into the list, we get a sorted prefix list (i.e., the elements in bold) until the entire list is sorted.

We will formally prove the correctness of Insertion Sort next. Notice that each iteration of Insertion Sort has a larger prefix of the list sorted (indicated in bold in the above example). We will use this idea to develop the following loop invariant for it:

Example F.15

A **loop invariant** for insertion sort is:

> At the start of iteration i, the sub-list $L[0 : i]$ consists of the original elements of $L[0 : i]$ in sorted order.

Example F.16

For **initialization**, we have that when $i = 1$ the list $L[0 : 1]$ consists of a single element which is the original element $L[0]$ and a list of size one is always sorted.

Example F.17

For **maintenance**, we assume that the loop invariant is true for some $k \geq 1$. That is, at the start of iteration k, the sub-list $L[0:k]$ consists of the original elements of $L[0:k]$ in sorted order. We will show that the loop invariant is true at the start of iteration $k+1$. During iteration k, the algorithm copies the kth item into *value* and the `while` loop moves all the items before position k that are larger than *value* up by one position until it finds the correct position for *value* and the algorithm then inserts *value* into this position (Lines 4-7). Since this operation only moves the position of $L[k]$ within the sub-list $L[0:k+1]$, the list $L[0:k+1]$ consists of the original elements of $L[0:k+1]$. Moreover, since $L[0:k]$ was previously in sorted order and the above insertion operation places $L[k]$ in the correct position, the sub-list $L[0:k+1]$ is now in sorted order. Thus, the loop invariant is maintained.

Example F.18

For **termination**, the `for` loop terminates when $i = |L|$. Substituting this into the loop invariant, we get that at termination the sub-list $L[0:|L|]$, which is the entire list L, consists of the original elements of L in sorted order. Thus, at termination the list L is correctly sorted.

Note that the above proof could be made even more rigorous by adding in a separate loop invariant proof for the `while` loop. This is left as an excercise for the reader.

F.5 CHAPTER SUMMARY AND KEY CONCEPTS

- **Loop invariants** are statements that are true at the start of each iteration of a loop. They allow us to prove correctness of the algorithm.
- Any loop invariant proof has four parts: (1) **stating** the invariant, (2) an **initialization** step in which the invariant is shown to be true at the start of the loop, (3) a **maintenance** step in which it is shown that if the invariant was true before the previous iteration then it must be true before the next iteration, and (4) a **termination** step in which the invariant is used to show that the loop correctly computed what it was suppsed to.

EXERCISES

Write out algorithms (with **loops**) for each of the following and prove them correct using loop invariant proofs.

F.1 Finding the maximum of a list

F.2 Finding the minimum of a list

F.3 Summing up the first n positive integers

F.4 Summing up the first n powers of 2

F.5 Computing the factorial of n $(1 \times 2 \times 3 \ldots \times n)$

F.6 Computing the nth Fibonacci number (iterative)

F.7 Linear search of a list

F.8 Reversing a list by creating a new list

F.9 Reversing a list in place

F.10 Selection sort (Hint: First prove a loop invariant for the inner loop, then use this to prove a loop invariant for the outer loop.)

F.11 Bubble sort (Hint: First prove a loop invariant for the inner loop, then use this to prove a loop invariant for the outer loop.)

F.12 Binary search (iterative)