Loop Invariant Proofs

The following is a lightweight tutorial for loop invariant proofs. It assumes that you have heard about these proofs, but don't yet know what to do with them and how to do them.

1. Introduction
2. General Strategies
3. Steps of the Proof
4. Example: Sum of Numbers
5. Example: Maximum of Numbers

1. Introduction

Loop invariant proofs might seem scary at first, in particular if you are not used to writing mathematical proofs. But they shouldn't be: when you plan to write a loop invariant proof, you already have an algorithm and you have an intuitive notion of why the algorithm is correct. Loop invariant proofs provide you with a very structured way of translating your intuition into something solid.

Let us start with a very simple example. Consider the following computational problem: given an array A (of size n) of numbers, output the sum of the numbers in A. Note that strictly speaking, we will always use a python list instead of an array.
Important: Since python lists are indexed from $0,\ldots,n-10,\ldots,n-1$, I will assume that we are using indices from 0,...,n-1. Note that in textbooks you more often will have indices $1,\ldots,n1,\ldots,n$. Thus, if you compare the following examples with textbook solutions, you should keep in mind that there might be an index shift by 1. Further note that we use $A[0:j]$A[0:j] (or simply $A[:j]$A[:j]) to refer to the subarray of AA from $A[0]$A[0] to $A[j-1]$A[j−1] (not including A[j]). In pseudo-code you will often see $A[0..j-1]$A[0..j−1] instead, which in contrast to the notation we use includes the last index (in this example $j-1$j−1). I provide an additional notebook with the same content but starting with index 1.

Back to the computational problem. A possible algorithm would be:

In [1]:

```
def sum(A):
    answer = 0
    for i in range(len(A)): # in pseudo-code for i=0,...,len(A)-1
        answer += A[i]
    return answer
```

Let's first test whether it actually works:

In [2]:

```
sum([1, 5, 10])
```

Out[2]:

```
16
```

In [3]:

```
sum([])
```

Out[3]:

```
0
```

In [4]:

```
sum(range(10)) == 9*10/2 # arithmetic series
```

Out[4]:

```
True
```

Before thinking about other steps in the loop invariant proof, we need a loop invariant. The algorithm seems obviously correct. But why?

Since we are running a loop, we are gaining information step by step. And if we loop over an array, the information that we have gained before we start the iteration with index jj is information about the subarray A[0:j], that is, the subarray from A[0] to A[j-1]. Thus the loop invariant will be something like:

Loop Invariant: At the start of the iteration with index jj, …… subarray $A[0:j]$A[0:j] …….

2. General Strategies for finding loop invariants

We still need to formulate a loop invariant for the algorithm given above. The list below is not meant to be complete, and also one strategy might work better for you then another. Here are some approaches:

1.) Think about a specific iteration: Imagine the algorithm has been running for a while, and the next iteration would be iteration (with index) jj. Alternatively you could imagine that the algorithm is interrupted just before interation jj. *What information did it gain so far?* If jj is not specific enough for you, you could think about a specific jj, e.g., 10. It might also work to think about the first few iterations, i.e., before iteration 0, before iteration 1, ... In the algorithm above we see in this case that only the value of *answer* changes: first it is 0, then A[0], then A[0]+A[1], and so on. This might help enough to observe that the information that we have gained before iteration jj is that *answer* = A[0]+A[1]+...+A[j-1].

2.) Think about: What you want to know at the end? Answering this question in some way shouldn't be difficult, because we wrote the algorithm to solve a specific computational problem. For the algorithm above, the answer simply is: "The variable *answer* should contain the sum of all numbers in A". But how could this help in finding a loop invariant? As we stated the incomplete loop invariant above, it read:

Loop Invariant: At the start of the iteration with index jj, …… subarray $A[0:j]$A[0:j] …….

Now lets assume array AA has size nn, and we are stepping out of a loop of the kind **for i=0,...,n-1**. After the iteration $n-1$n−1 you may assume that i=(n-1)+1=n. Thus, the 'subarray $A[0:n]$A[0:n]' is actually AA! Therefore at this point the loop invariant rephrased reads: After the loop, ... AA .... Now lets just try to replace in the sentence "The variable *answer* should contain the sum of all numbers in A" the array A by subarray[0:j], and plug that into the loop invariant. We get:

Loop Invariant: At the start of the iteration jj of the loop, the variable *answer* should contain the sum of all numbers in subarray $A[0:j]$A[0:j].

Perfect! Now will this always work? Unfortunately not. One problem might be that the information that we have gained after the loop is not exacly what we wanted to compute, but we are using it to get our final result. But this approach can work, and sometimes it is just useful, to get started thinking about the right loop invariant.

Think about the algorithmic technique used. This maybe sounds complicated, but is actually quite simple. In our algorithm above, we have a variable answer, which we manipulate while the loop is running. So what we are actually doing is *incrementally building a solution*. And while the above is only one example, we actually use loops very often to incrementally build a solution. In these cases, the loop invariant is often a statement of the form: *The solution computed so far, is the correct solution for the things that I have seen so far*. Merging this into the incomplete loop invariant from above, this would state.

Loop Invariant: At the start of the iteration jj of the loop, the variable *answer* should contain the correct solution for the subarray $A[0:j]$A[0:j].

Now in this specific example, "the correct solution" is the sum of the numbers, so indeed we again end up with the loop invariant:
Loop Invariant: At the start of the iteration jj of the loop, the variable *answer* should contain the sum of the numbers from the subarray $A[0:j]$A[0:j].

3. The steps of the proof

Lets assume we have successfully formulated a loop invariant. Fantastic, the hardest part is done!

Now we need to handle the three steps of the proof: *Initialization*, *Maintenance*, *Termination*. Before we look at how to handle them, lets remind ourselves, why they together constitute a proof.
About Termination: We have formulated a loop invariant, that -if true after stepping out of the loop- gives us a statement with which we can prove correctness. This is exactly what *Termination* is about: Conclude from the loop invariant after the last iteration that the algorithm does what it should do. There is a slight subtlely: While the name *Termination* seems to imply that we need to prove that the algorithm terminates, *Termination* in most cases is more about showing that the loop invariant after completing the loop gives us a statement from which we then can conclude correctness. To make things more confusing: for the **for**-loops that we are considering, it is always obvious that they terminate, but it can happen that you have for instance a **while**-loop for which this is not so obvious. If the loop indeed doesn't terminate, then the proof won't work. So somewhere you will need to prove that the loop terminates. However, often this already happens in the analysis of the running time. In short: plug nn into the loop invariant, and argue why this means that your algorithm works correctly. For our running example this means:
Termination: When the **for**-loop terminates $i=(n-1)+1=n$i=(n−1)+1=n. Now the loop invariant gives: The variable *answer* contains the sum of all numbers in subarray A[0:n]=A. This is exactly the value that the algorithm should output, and which it then outputs. Therefore the algorithm is correct.
About initialization: Now we know that for 'Termination', we want that the loop invariant is true at the end? The easiest way to prove this: proving that it was true all the time. Naturally, this is done in two steps: showing that it is true at the beginning and then showing that it remains true while the loop is running.
Initialization is the first part of this, that is, showing that the loop invariant is true when we first enter the loop. At this point $i=0$i=0 and we will have a statement about $A[0:0]$A[0:0]?!
Now, $A[0:0]$A[0:0] is the subarray starting at index 0 and ending before index 0. There is no such index, so $A[0:0]$A[0:0] is an *empty array*. If this is confusing, one easy way to understand what $A[0:0]$A[0:0] means, is to look at what happens when we go from $A[0:j+1]$A[0:j+1] to $A[1:j]$A[1:j]: we drop $A[j]$A[j]. In our setting this means $A[0:0]$A[0:0] is $A[0:1]=A[0]$A[0:1]=A[0] but with $A[0]$A[0] removed. This means that nothing is left, and therefore $A[0:0]$A[0:0] is is indeed simply an empty array. So here is what happens if we plug this into our running example:
Initialization: Before the first iteration of the loop, the loop invariant states: 'At the start of iteration 0 of the loop, the variable *answer* should contain the sum of the numbers from the subarray $A[0:0]$A[0:0], which is an empty array. The sum of the numbers in an empty array is 0, and this is what *answer* has been set to.
About maintenance: Finally! At this point it should be clear that we want to show that *if* the loop invariant is true at the beginning of iteration jj of the loop that it is *then* true at the beginning of the next iteration. This means that if the loop invariant makes a statement about the subarray A[1:j] then after iteration jj we should get the same statement but for subarray A[1:j+1]. Generally the maintenance proof might look somethink like
Maintenance: Assume that the loop invariant holds at the start of iteration jj. Then it must be that [...write here what the loop invariant states]. In iteration jj, [...write here what the loop does; it should result in providing proof of the following sentence]. Thus at the start of iteration $j+1$j+1, [...write here

the loop invariant but with the variable increased by one, e.g., if the loop invariant makes a statement about subarray A[1:j], here you would have the same statement but for subarray A[1:j+1]] which is what we needed to prove.
Lets do this for our example:

Maintenance: Assume that the loop invariant holds at the start of iteration jj. Then it must be that *answer* contains the sum of numbers in subarray $A[0:j]$A[0:j]. In the body of the loop we add $A[j]$A[j] to *answer*. Thus at the start of iteration $j+1$j+1, *answer* will contain the sum of numbers in $A[0:j+1]$A[0:j+1], which is what we needed to prove.

That's it. So in summary, if the question is to prove correctness of the algorithm above, a complete solution could be the following (which is just a copy of the relevant parts above).

4. Example: Sum of Numbers

We prove correctness by a loop invariant proof using the following invariant:

Loop Invariant: At the start of iteration jj of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[0:j].

Initialization: At the start of the first loop the loop invariant states: 'At the start of the first iteration of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[0:0], which is an empty array. The sum of the numbers in an empty array is 0, and this is what *answer* has been set to.

Maintenance: Assume that the loop invariant holds at the start of iteration jj. Then it must be that *answer* contains the sum of numbers in subarray $A[0:j]$A[0:j]. In the body of the loop we add $A[j]$A[j] to *answer*. Thus at the start of iteration $j+1$j+1, *answer* will contain the sum of numbers in $A[0:j+1]$A[0:j+1], which is what we needed to prove.

Termination: When the **for**-loop terminates $i=(n-1)+1=n$i=(n−1)+1=n. Now the loop invariant gives: The variable *answer* contains the sum of all numbers in subarray A[0:n]=A. This is exactly the value that the algorithm should output, and which it then outputs. Therefore the algorithm is correct.

5. Example: Maximum of Numbers

This example demonstrates what effect an **if**-statement in the body of the loop has on the proof. Lets assume we want to find the largest number in a non-empty array A.

In [19]:

```
def max(A):
    answer = A[0]
    for j in range(1,len(A)): # in pseudo-code: for i=1,...,len(A)-1
        if (A[j]>answer): answer = A[j]
    return answer
```

Let's first test whether it actually works:

In [20]:

```
max([10])
```

Out[20]:

```
10
```

In [21]:

```
max([10,20])
```

Out[21]:

```
20
```

In [23]:

```
max([30,10,5])
```

Out[23]:

```
30
```

Now lets first do the proof. A lot of this is just copy-and-paste from the previous example.
Loop Invariant: At the start of the iteration with index j of the loop, the variable *answer* should contain the maximum of the numbers from the subarray A[0:j].
Initialization: At the start of the first loop, we have j=1. Therefore the loop invariant states: 'At the start of the iteration with index j of the loop, the variable *answer* should contain the maximum of the numbers from the subarray A[0:1], which is A[0]. This is what *answer* has been set to.
Maintenance: Assume that the loop invariant holds at the start of iteration jj. Then it must be that *answer* contains the maximum of numbers in subarray $A[0:j]$A[0:j]. There are two cases: (1) $A[j]$A[j] > answer. From the loop invariant we get that $A[j]$A[j] is larger than the maximum of the numbers in $A[0:j]$A[0:j]. Thus, $A[j]$A[j] is the maximum of $A[0:j+1]$A[0:j+1]. In this case, the algorithm sets answer to $A[j]$A[j], thus in this case the loop invariant holds again at the beginning of the next loop. (2) $A[j]$A[j] $\leq$≤ answer. That is, the maximum in $A[0:j]$A[0:j] is at least as large as $A[j]$A[j], thus the maximum of $A[0:j+1]$A[0:j+1] is the same as the maximum of $A[0:j]$A[0:j]. The algorithm also doesn't change *answer*, thus in this case the loop invariant holds again at the beginning of the next loop.
Termination: When the **for**-loop terminates $j=(n-1)+1=n$j=(n−1)+1=n. Now the loop invariant gives: The variable *answer* contains the maximum of all numbers in subarray $A[0:n]=A$A[0:n]=A. This is exactly the value that the algorithm should output, and which it then outputs. Therefore the algorithm is correct.
What we see here is that an **if**-statement in the algorithm results in a *case distinction* in the proof. In the lecture we saw the example of better_linear_search. There we also had an **if**-statement, and therefore a case distinction. The better_linear_search proof is a bit more involved, because there was a **return**-statement in the body of the loop.

I hope these examples help you get started. The best way to learn how to do loop invariant proofs is to do them. So I recommend to simply do the proofs for the examples from the practice exercises and from the homework assignment, and let me know in case you get stuck.