Mathematical Foundations

Loop invariants

CS51 – Spring 2026

Last time we started talking about the idea of proofs being at the heart of what we are doing as computer scientists and set up the foundation of what we will need through propositional Logic. Today we will see a type of proof that allows us to establish that an iterative algorithm, that is an algorithm that has a for/while loop, correctly solves a particular problem.

Proving correctness of iterative algorithms

- Iterative algorithms: algorithms that use loops, such as for and while loops.
- What do we mean by our iterative program being correct?
  - if certain conditions are met and we run the program, then the program will terminate (we say that the program will halt) and we will receive the correct result.
- The condition that needs to be met to run the program is called a **precondition**.
- The result that needs to be met after the program **termination** is called a **postcondition**.
- The precondition and postcondition are known as the **specifications** of the program.

2

Iterative algorithms use loops such as for loops and while loops. How do we go about proving that our iterative program is correct? What do we even mean by our iterative program being correct? We men that if certain conditions are met and we run the program, then the program will terminate (we say that the program will halt) and we will receive the correct result.
The condition that needs to be met to run the program is called a precondition. The result after the program termination is called a postcondition. The pre- and post-condition are known as the specifications of the program.

Proving correctness of iterative algorithms

- To prove the correctness of a program we would need to prove that if the precondition is true, then the program will terminate and the postcondition will be true.
- If we were to think of this in terms of logic, we would need to prove that
  - precondition ⇒ termination ∧ postcondition
- Sometimes, it makes sense to break down the proof in two parts:
  - Proving *termination*: precondition ⇒ termination
  - Proving *partial correctness*: precondition ∧ termination⇒ postcondition

3

To prove the correctness of a program we would need to prove that if the precondition is true, then the program will terminate and the postcondition will be true.
If we were to think of this in terms of logic, we would need to prove that

precondition ⇒ termination ∧ postcondition

Sometimes, it makes sense to break down the proof in two parts:

Proving *termination*: precondition ⇒ termination

Proving *partial correctness*: precondition ∧ termination⇒ postcondition

Practice Time

- Write an iterative Python function called `linear_search` that takes a list `lst` and an element `element` and returns the index of the first encounter of the `element` in `lst`, if it exists, or -1 if it does not.
- What are the pre- and post-conditions?
- Remember, pre-condition is what needs to be true for the program to run. In this case, what would your function need to run?
- Post-condition is the result that needs to be met after the program terminates. In this case, what happens when your function completes its work?

4

Let's see how we go about proving iterative programs through an example.
Write an iterative Python function called `linear_search` that takes a `list` and an `element` and returns the index of the first encounter of the `element` in the list, if it exists, or -1 if it does not.
What are the pre- and post-conditions for your program? Remember, pre-condition is what needs to be true for the program to run. In this case, what would your function need to run?
Post-condition is the result that needs to be met after the program terminates. In this case, what happens when your function completes its work?

Answer

```
def linear_search(lst, element):
  n = len(lst)
  for i in range(n):
    if lst[i]==element:
       return i
  return -1
```

```
def linear_search(lst, element):
   i = 0
   n = len(lst)
   while (i<n):
      if lst[i]==element:
         return i
      i+=1
   return -1
```

- Pre-condition : A list `lst` of n elements, $[e_0, e_1, \ldots, e_{n-1}]$ and an element `element` to search for.
- Post: The index `i` of the element e_i, so that $e_i == lst[i] == element$ and $lst[e_j] != element$, for every $j < i$, or -1 if no element in list is equal to $element$.

5

Your code is probably one of the two, or you might have used enumerate.

Is our output correct?

- To show that the output is correct we would need to know that:
  - If index `i` is returned, then `lst[i]==element` and `i` is the first index with a matching element.
  - If -1 is returned, then the `element` is not in list.
- We will do so by using **loop invariants** and assume that while loops are in the form of:

```
while (guard):
    #body
```

- And for loops are in the form of:

```
for i in range (start, stop, step): #start=0, step=1 by default
    #body
```

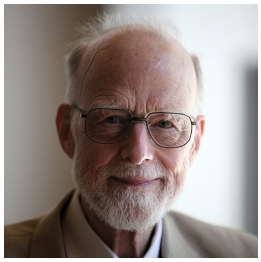
6

Last time, we said that proofs are about being confident that something works all times, not just because we plugged in a few inputs and we got the anticipated result. To show that the output of our program is correct, we would need to that returned index is the index of the first encounter of the element to search or -1 if it doesn't exist. We will prove this by using loop invariants. We will assume that if given a while loop it will be of the while(guard is true): form and if it includes a for loop that it will have a for i in range form.

Loop invariants

- Assuming the precondition holds, a *loop invariant* for a loop L is a logical property P such that:
- i) P is true before L is first executed; and
- (ii) if P is true at the beginning of an iteration of L, then P is true after that iteration of L. that is true at the beginning and end of every iteration of a loop.

Robert Floyd



Tony Hoare

7

Assuming the precondition holds, *a loop invariant* for a loop L is a logical property P such that:
i) P is true before L is first executed; and
(ii) if P is true at the beginning of an iteration of L, then P is true after that iteration of L. that is true at the beginning and end of every iteration of a loop.
This technique was pioneered in the 60s by Robert Floyd and Tony Hoare.

Proving correctness using loop invariants

- To prove correctness, we need to show:
  - *Initialization*: The loop invariant is true before the loop runs for the first iteration.
  - *Maintenance*: We assume that the loop invariant is true before an iteration. Given this assumption we need to show that the loop invariant remains true after the execution of the loop body and before entering the next iteration.
  - *Postcondition*: Show that if the guard fails/is false (so the loop exits), then the desired result of the loop has been achieved: the loop post-condition holds.
  - *Termination*: After a finite number of iterations of the loop, the guard will become false and the loop will terminate.
- All four step would be necessary to show full correctness.

8

To prove correctness of an iterative algorithm using loop invariants, we need to show the following four steps:

Initialization: The loop invariant is true before the loop runs for the first iteration.
Maintenance: If the loop invariant is true before an iteration, it remains true after the execution of the loop body and before entering the next iteration.
Postcondition: Show that if the guard is false (so the loop exits), then the desired result of the loop has been achieved: the loop post-condition holds.

Termination: After a finite number of iterations of the loop, the guard will become false and the loop will terminate.

The first three steps succeeding would prove partial correctness. The fourth step would be necessary to show full correctness.

Loop invariants

- Loop invariants, often make a statement depending on an index i and what you have seen so far, e.g., in `lst[0:i]` (this list slice contains all elements from index 0 to index `i-1`, that is 0 is inclusive but `i` is exclusive).
- Let's go back to the linear search example.
- **Practice Time**: What loop invariant can we can come up with?
  - At the start of each iteration, element does not exist in the sublist lst[0:i]

9

What would be a good loop invariant for linear search?

Loop invariant for linear search - Initialization

- *Initialization*: Before the first iteration, `i==0`, then the slice `lst[0:0]` would be empty. Therefore, `element` would not be in the slice and the loop invariant holds true.

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

10

We will work through each of the steps of loop invariants together.

Loop invariant for linear search - Maintenance

- *Maintenance*: Let's assume the loop invariant is true at the start of the `i`-th iteration of the loop, that is the `element` does not exist in `lst[0:i]`. If `lst[i] == element`, the current iteration is the final one (the `return` statement would be reached); otherwise, if `lst[i] != element`, we have that the loop invariant holds true at the end of this iteration and before the beginning of the next iteration, since `element` is not in `lst[0:i+1]`.

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

11

Loop invariant for linear search - Postcondition

- *Postcondition:* The loop terminates when
  - i) the guard fails, that is `i==n==len(lst)`. In that case, the `element` has not been found, `lst[0:n]` does not contain the `element`, and -1 is correctly returned.
  - ii) the `return` statement is reached because `lst[i]==element` for the first time, and `i` is correctly returned as the index of first occurrence of `element` in the list (since we know that it couldn't have been in the `list[0:i]`).

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

12

Loop invariant for linear search - Termination

- *Termination*: `i` increases by 1 in every iteration and ranges from 0 to maximum `n=len(lst)`. Thus, in a finite number of iterations of the loop, we will either find the `element` and return its index or return -1.

```
def linear_search(lst, element):
    n = len(list)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

13

Practice Time

- Write an iterative function called `iterative_multiplication` that computes and returns the product of two non-negative integers m and n via repeated addition.
- What are the pre- and post-conditions?

Answer

```
def iterative_multiplication(m, n):
    product = 0
    for i in range(n):
        product += m
    return product
```

```
def iterative_multiplication(m, n):
    product = 0
    i=0
    while i<n:
        product += m
        i += 1
    return product
```

- pre-condition: the input variables m and n are non-negative integers.
- post-condition: the returned value is equal to the product of m and n.

15

You either came up with an algorithm using some form of for loop or while loop. The precondition would state that the input variables m and n are non-negative integers. The post-condition is that the output is equal to the product of m and n.

Iterative multiplication - Loop invariant

- **Practice Time**: What is a good loop invariant for this function?,
- After `i` iterations, `product = m x i`.

```
def iterative_multiplication(m, n):
    product = 0
    for i in range(n):
        product += m
    return product
```

```
def iterative_multiplication(m, n):
    product = 0
    i=0
    while i<n:
        product += m
        i += 1
    return product
```

16

Loop invariants, often make a statement depending on an index i and what you have seen so far. For example, for the iterative multiplication, a possible loop invariant would state that after i iterations, product = m*i.

Iterative multiplication - Initialization

- *Initialization*: Before the first iteration `product=0` and `i=0`, thus `product=0=mx0`.
- So, the loop invariant holds before the first iteration.

```
def iterative_multiplication(m, n):
    product = 0
    for i in range(n):
        product += m
    return product
```

```
def iterative_multiplication(m, n):
    product = 0
    i=0
    while i<n:
        product += m
        i += 1
    return product
```

17

For the initialization, we know that before we enter the first iteration of the loop, both the product and I are equal to 0. Thus product is indeed mx0 and the loop invariant holds before the first iteration.

Iterative multiplication - Maintenance

- *Maintenance*: Let's assume the loop invariant is true at the start of the i-th iteration of the loop, that is $product_{old} = m \times i_{old}$.
- In the loop body, we execute $product_{new} = product_{old} + m$ and increment i by 1, that is $i_{new} = i_{old} + 1$
- After this iteration, $product_{new} = (m \times i_{old}) + m = m \times (i_{old}+1) = m \times i_{new}$.
- So, the invariant still holds.

```
def iterative_multiplication(m, n):
    product = 0
    for i in range(n):
        product += m
    return product
```

```
def iterative_multiplication(m, n):
    product = 0
    i=0
    while i<n:
        product += m
        i += 1
    return product
```

18

In the maintenance step, we will assume that the loop invariant is true at the start of the i-iteration of the loop. Once we enter the body, we execute product=product+m and increment i by1. After this iteration, the new product is equal to old product +m. But old product by assumption is equal to mxi which can substitute. (mxi)+m=mx(i+1) which is equal to m x new i. So the invariant still holds at the end of the loop.

Loop invariant for linear search - Postcondition

- *Postcondition:* The loop terminates when the guard fails, that is when `i = n`.
- By the loop invariant `product = m x i = m x n`.
- Which is the exact result we were hoping to compute!

```
def iterative_multiplication(m, n):
    product = 0
    for i in range(n):
        product += m
    return product
```

```
def iterative_multiplication(m, n):
    product = 0
    i=0
    while i<n:
        product += m
        i += 1
    return product
```

19

In the postcondition, we examine what if the guard fails then the desired result is achieved. Here, the guard fails when i=n. By the loop invariant assimption product=mxi, thus product = mxn which is what we were hoping to compute.

Loop invariant for linear search - Termination

- *Termination*: Since `i` starts at 0, increases by 1 at each iteration, and the loop guard is `i < n`, the loop must terminate after exactly n iterations.
- Thus, in a finite number of iterations of the loop, we will calculate the product of m and n.
- Putting it altogether, we have proven the correctness of this program.

```
def iterative_multiplication(m, n):
    product = 0
    for i in range(n):
        product += m
    return product
```

```
def iterative_multiplication(m, n):
    product = 0
    i=0
    while i<n:
        product += m
        i += 1
    return product
```

20

Finally, we need to show that after a finite number of iterations of the loop the guard will fail. Since I starts at 0 and increments by 1 at ech iteration, the loop will terminate exactly after n iterations, that is after a finite number of iterations. Tada! We have successfully proven the full correctness of this program.

Practice time

- You are given the following program that tests whether a positive integer is prime by iterating through all numbers from 2 to $\sqrt{n}$ and checking divisibility.

```
def is_prime(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True
```

- First, test it with a few numbers to convince yourself that it works.
- What are the pre-and post-conditions? What is a good loop invariant?
- Use loop invariants to prove that the function works correctly.

21

Your turn now.

Answer

- Pre-condition : Input n is an integer, $n \geq 2$.
- Post-condition: The function returns True if n is prime and False if it is composite.
- Loop invariant: At the start of each iteration i, no integer k with $2 \leq k < i$ divides n.

```
def is_prime(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True
```

22

Answer

- *Initialization*: At the start of the first loop, `i=2`. There are no integers k with $2 \le k < 2$, so the invariant is trivially true.

```
def is_prime(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True
```

23

Answer

- *Maintenance*: Suppose at the start of some iteration the invariant holds and no integer k with 2 ≤ k < i divides n.
  - Once we enter the loop, if `n % i == 0`, then we return False immediately, which is correct because n would be a composite if it can be divided by i.
  - Otherwise, `n % i != 0`. That means i does not divide n.
  - Then we increment `i` by `1` to `i+1`.
  - Thus at the start of the next iteration, no integer k with 2 ≤ k < (i+1) divides n. The invariant holds.

```
def is_prime(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True
```

24

Answer

- *Post-condition*: The loop exits when `i*i>n`. At this point, every possible divisor between 2 and $\sqrt{n}$ has been checked.
- If any such divisor existed, the loop would have terminated earlier and returned False.
- Since it did not, no divisor exists up to $\sqrt{n}$, so n must be prime. Therefore, the postcondition holds when the loop terminates and returns True.

```
def is_prime(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True
```

25

Answer

- *Termination*: The variable `i` starts at 2 and increases by 1 each iteration. The guard `i * i <= n` eventually becomes False when i > $\sqrt{n}$. Thus, the loop always terminates.

```
def is_prime(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True
```

26

Practice time

- Write an iterative function `count_digits`, that given a (decimal) number it returns how many digits it has. For example,
- `count_digits(7)` → `1`
- `count_digits(123)` → `3`
- `count_digits(10005)` → `5`
- What are the pre-and post-conditions? What is a good loop invariant?
- Use loop invariants to prove that the function works correctly.

27

Answer

- Pre-condition : Input n is an integer, n > 0.
- Post-condition: The function returns the number of digits in the decimal representation of n.
- Loop invariant: after `count` iterations, the last `count` digits have been removed from the original n and n is the part of the original number that remains to be processed.

```
def count_digits(n):
    count = 0
    while n > 0:
        n //= 10
        count += 1
    return count
```

28

Answer

- *Initialization*: At the start of the first loop, `count=0` and n is the full original number. Since 0 digits have been removed from n both count and n are accurate and the invariant holds.

```
def count_digits(n):
    count = 0
    while n > 0:
        n //= 10
        count += 1
    return count
```

29

Answer

- *Maintenance*: Suppose at the start of some iteration the invariant holds and $count_{old}$ holds the number of digits already removed and n_{old} is the remaining portion of the original number, with the last $count_{old}$ digits from $n_{original}$ having been removed.
  - Once we enter the loop, we update $n_{new} = n_{old}$ `//10`, removing the last digit, and increment $count_{new} = count_{old} + 1$ of removed digits by 1.
  - Thus, at the start of the next iteration, one more digit has been removed from n and the number of removed digits count has increased by 1. The invariant holds.

```
def count_digits(n):
    count = 0
    while n > 0:
        n //= 10
        count += 1
    return count
```

30

Answer

- *Post-condition*: The loop exits when n=0. By the invariant, we have `count` being the number of right-most digits that have been removed from the original number and n being the remaining portion of the original number. But since all the digits have been removed once n is 0, `count` will be equal to the total number of digits in the original number n.

```
def count_digits(n):
    count = 0
    while n > 0:
        n //= 10
        count += 1
    return count
```

31

Answer

- *Termination*: At each iteration, the statement `n //= 10` strictly decreases n. Since n is positive integer, this process can only happen a finite number of times (at most equal to the number of digits of n). Eventually, n becomes 0, making the guard n > 0 False, so the loop terminates.

```
def count_digits(n):
    count = 0
    while n > 0:
        n //= 10
        count += 1
    return count
```

32