Loop Invariants

https://cs.pomona.edu/classes/cs140/

Outline

Topics and Learning Objectives

- Some asymptotic complexity review
- Practice writing loop invariants

Exercise

- Loop Invariant

Extra Resources

- **Chapter 2** of Introduction to Algorithms, Third Edition
- Loop Invariant Proofs (Web Archive)

Loop Invariant Proofs

- A procedural way to prove the correctness of some code with a loop
- Very similar to inductive proofs for recursive algorithms

Example

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

How do we prove that this code sums all values in the given array?

Some useful syntax:

- array[start ..= end] is the subarray
  - **Including** array[start], array[end], and everything in between
  - <u>Inclusive</u> lower and upper bounds

- array[start ..< end] is the subarray
  - **Including** array[start], **excluding** array[end], and **including** everything in between
  - <u>Inclusive</u> lower bound, <u>exclusive</u> upper bound

Loop Invariants

A loop invariant is a predicate (a statement that is either true or false) with the following properties/conditions:

1. It is true upon entering the loop the first time. Initialization

2. If it is true upon starting an iteration of the loop, it remains true upon starting the next iteration. Maintenance

3. The loop terminates, and the loop invariant plus the reason that the loop terminates gives you the property that you want. Termination

Relation to Induction Proofs

Loop Invariant

- Initialization: true before entering first iteration
- Maintenance: true after executing any iteration
- Termination: true after the final iteration

Induction

- Base case: true when acting on the smallest input
- Inductive hypothesis: assume true for smaller inputs
- Inductive step: true after executing on current input

Relation to Induction Proofs

Loop Invariant

- Initialization: true before entering first iteration
- Maintenance: true after executing any iteration
- Termination: true after the final iteration

Induction

- Base case: true when acting on the smallest input
- Inductive hypothesis: assume true for smaller inputs
- Inductive step: true after executing on current input

How to perform a proof by loop invariant

1. State the loop invariant
    1. A statement that can be easily proven true or false
    2. The statement must reference the purpose of the loop
    3. The statement must reference variables that change each iteration

Initialization

2. Show that the loop invariant is true before the loop starts

Maintenance

3. Show that the loop invariant holds when executing any iteration

4. Show that the loop invariant holds once the loop ends Termination

Loop Invariant

At the start of the iteration with <reference the looping variable>,

the <reference to partial solution>

<something about why the partial solution is correct>.

Using Insertion Sort as example.

At the start of the iteration with index j,

the subarray `array[0 ..= j-1]` consists of the elements originally in `array[0 ..= j-1]`

rearranged into nondecreasing order.

Example

1. State the loop invariant
    1. A statement that can be easily proven true or false
    2. The statement must reference the purpose of the loop
    3. The statement must reference variables that change each iteration

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

Exercise

Example

1. State the loop invariant
    1. A statement that can be easily proven true or false
    2. The statement must reference the purpose of the loop
    3. The statement must reference variables that change each iteration

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

What would be a bad loop invariant for proving this procedure?

What would be a good loop invariant for proving this procedure?

Example

1. State the loop invariant
   1. A statement that can be easily proven true or false
   2. The statement must reference the purpose of the loop
   3. The statement must reference variables that change each iteration

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

At the start of the iteration with index `i`, the variable `sum` is the sum of all values in the subarray `array[0 ..< i]`.

At the start of the iteration with index `i`, the variable `sum` is the sum of all values in the subarray `array[0 ..< i]`.

Example

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

1. **Initialization**
2. Maintenance
3. Termination

At the start of the iteration with index `i`, the variable `sum` is the sum of all values in the subarray `array[0 ..< i]`.

Example

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

Initialization:
Upon entering the first iteration, `i = 0`. There are no numbers in the subarray `array[0 ..< i]`. The sum of no terms is the identity for addition (0).

Example

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

At the start of the iteration with index i, the variable sum is the sum of all values in the subarray array[0 ..< i].

Maintenance:
Upon entering an iteration with index i, assume that sum is equal to the sum of all values in the subarray array[0 ..< i]:

$$sum = \sum_{k=0}^{i-1} array[k]$$

The current iteration adds array[i] to sum and then increments i, so that the loop invariant holds upon entering the next iteration.

At the start of the iteration with index `i`, the variable `sum` is the sum of all values in the subarray `array[0 ..< i]`.

Example

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

Termination:
The loop terminates with `i = n`. According to the loop invariant, `sum` is equal to the sum of all values in the subarray `array[0 ..< i]`:

$$sum = \sum_{k=0}^{i-1} array[k] = \sum_{k=0}^{n-1} array[k]$$

which is the sum of all values in the array.

At the start of the iteration with index `i`, the variable `sum` is the sum of all values in the subarray `array[0 ..< i]`.

Example

```
FUNCTION SumArray(array)
  sum = 0
  i = 0
  WHILE i < array.length
    sum = sum + array[i]
    i = i + 1
```

1. Initialization
2. Maintenance
3. Termination

A more complex example: Dijkstra’s Algorithm

```
DIJKSTRA (G, w, s)
  S = null
  Q = G.V
  while Q is not null
    u = EXTRACT-MIN(Q)
    S = S union {u}
    for each vertex v adjacent to u
      RELAX(u, v, w)
```

Loop Invariant:
At the start of each iteration of the while loop, v.d = delta(s, v) for each vertex v in S.

Loop Invariant:
At the start of each iteration of the while loop, v.d = delta(s, v) for each vertex v in S.

Dijkstra’s Algorithm

Initialization:
Initially, S = null and so the invariant is trivially true

```
DIJKSTRA (G, w, s)
   S = null
   Q = G.V
   while Q is not null
      u = EXTRACT-MIN(Q)
      S = S union {u}
      for each vertex v adjacent to u
         RELAX(u, v, w)
```

Dijkstra’s Algorithm

Loop Invariant:
At the start of each iteration of the while loop, v.d = delta(s, v) for each vertex v in S.

```
DIJKSTRA (G, w, s)
    S = null
    Q = G.V
    while Q is not null
        u = EXTRACT-MIN(Q)
        S = S union {u}
        for each vertex v adjacent to u
            RELAX(u, v, w)
```

Maintenance:
<long proof by contradiction on page 661 of Cormen>

Loop Invariant:
At the start of each iteration of the while loop, v.d = delta(s, v) for each vertex v in S.

Dijkstra's Algorithm

```
DIJKSTRA (G, w, s)
   S = null
   Q = G.V
   while Q is not null
      u = EXTRACT-MIN(Q)
      S = S union {u}
      for each vertex v adjacent to u
         RELAX(u, v, w)
```

Termination:
At termination, Q = null which, along with our earlier invariant that Q = V – S, implies that S = V. Thus, u.d = delta(s, u) for all vertices in G.V.

Practice (Running Time and Loop Invariants)

```
FUNCTION NaiveExponentiation(x, n)
    IF n == 0
        RETURN 1
    result = 1
    FOR i IN [0 ..< n]
        result = result * x
    RETURN result
```

```
FUNCTION FastExponentiation(x, n)
    result = 1
    a = x
    WHILE n != 0
        r = n % 2
        IF r == 1, result = result * a
        n = n // 2
        a = a * a
    RETURN result
```