

Loop Invariants

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

Outline

Topics and Learning Objectives

- Practice writing loop invariants

Exercise

- Loop Invariant

Extra Materials

- **Chapter 2** of Introduction to Algorithms, Third Edition
- <https://www.win.tue.nl/~kbuchin/teaching/JBP030/notebooks/loop-invariants.html>

Loop Invariant Proofs

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

```
FUNCTION SumArray(array)
```

```
    sum = 0
```

```
    i = 0
```

```
WHILE i < array.length
```

```
    sum = sum + array[i]
```

```
    i = i + 1
```

Example

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

Loop Invariants

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

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

- True before entering first loop
- True after executing any iteration
- True when loop terminates

Induction

- Prove for the base case
- State/assume inductive hypothesis
- Prove inductive step

Relation to Induction Proofs

Loop Invariant

- True before entering first loop
- True after executing any iteration
- True when loop terminates

Induction

- Prove for the base case
- State/assume inductive hypothesis
- Prove inductive step

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

1. *At the start of the iteration with* `<reference the looping variable>`, ...
 - For example: “`index j`”
2. *The* `<reference to partial solution>` ...
 - For example: “`subarray array[0 ..= j-1]` consists of the elements originally in `array[0 ..= j-1]`...”
3. `<something about why the partial solution is correct>`.
 - For example: “In sorted order.”

Example

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

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**

Exercise

Example

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

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**

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]`.

Example

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

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

1. Initialization
2. Maintenance
3. Termination

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]`.

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]`:

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

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

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]`.

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]`:

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

which is the sum of all values in the array.

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, $v.d = \text{delta}(s, v)$ 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)

Loop Invariant:

At the start of each iteration of the while loop, $v.d = \text{delta}(s, v)$ for each vertex v in S.

Initialization:

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

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, $v.d = \text{delta}(s, v)$ for each vertex v in S.

Maintenance:

<long proof by contradiction on page 661 of the textbook>

Dijkstra's Algorithm

DIJKSTRA (G, w, s)

$S = \text{null}$

$Q = G.V$

while Q is not null

$u = \text{EXTRACT-MIN}(Q)$

$S = S \text{ union } \{u\}$

for each vertex v adjacent to u

$\text{RELAX}(u, v, w)$

Loop Invariant:

At the start of each iteration of the while loop, $v.d = \text{delta}(s, v)$ for each vertex v in S .

Termination:

At termination, $Q = \text{null}$ which, along with our earlier invariant that $Q = V - S$, implies that $S = V$. Thus, $u.d = \text{delta}(s, u)$ for all vertices in $G.V$.