Problem 2

Part (a)

A suitable loop invariant is:

$$t = \begin{cases} \frac{i}{2} & \text{if } i \equiv 0 \pmod 2 \\ -\frac{i+1}{2} & \text{if } i \equiv 1 \pmod 2 \end{cases}$$

Part (b)

Before the loop starts, $i = 0$ and $t = 0$. Since $0 \mod 2 = 0$, the invariant requires $t = \frac{0}{2} = 0$, which matches $t = 0$. So, it holds initially.

Part (c)

Assume the invariant holds for $i = k$:

$$t = \begin{cases} \frac{k}{2} & \text{if } k \equiv 0 \pmod 2 \\ -\frac{k+1}{2} & \text{if } k \equiv 1 \pmod 2 \end{cases}$$

After the loop body, $i = k + 1$.

- If $k + 1$ is even, k is odd, so $t = -\frac{k+1}{2}$. Then $t := t + (k + 1) = -\frac{k+1}{2} + (k + 1) = \frac{k+1}{2}$, which fits the invariant for $k + 1$ even.

- If $k + 1$ is odd, k is even, so $t = \frac{k}{2}$. Then $t := t - (k + 1) = \frac{k}{2} - (k + 1) = -\frac{k+2}{2}$, and since $k + 1$ is odd, the invariant requires $t = -\frac{(k+1)+1}{2} = -\frac{k+2}{2}$, which holds.

Thus, the invariant is preserved.

Part (d)

When the loop exits, $i = n$, and the invariant gives:

$$t = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod 2 \\ -\frac{n+1}{2} & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

This matches the postcondition: $n \mod 2 = 0 \implies t = \frac{n}{2}$ and $n \mod 2 = 1 \implies t = -1 \times \frac{n+1}{2}$.

Part (e)

A decrementing function is $n - i$. At start, $i = 0$, so $n - i = n \geq 0$. Each iteration increases i by 1, reducing $n - i$ by 1. When $i = n$, $n - i = 0$, and the loop stops.

Problem 3: Loopy Square Root

(a) Testing the Initial Code

I began by writing a temporary `Main()` method to call `loopysqrt()` with several inputs. For example, running the algorithm produced the following results:

- $n = 4$**:**
  - **Iteration details:**
    - Initially: `root = 0`, `a = 4`
    - 1st iteration: `root` becomes 1, then
      $$a := 4 - (2 \times 1 - 1) = 4 - 1 = 3.$$
    - 2nd iteration: `root` becomes 2, then
      $$a := 3 - (2 \times 2 - 1) = 3 - 3 = 0.$$
  - **Final output:** `root = 2`.
  - **Check:** $2^2 = 4 \geq 4$ and $(2-1)^2 = 1 < 4$.
- $n = 25$**:** The algorithm produced `root = 5` (since $5^2 = 25$ and $4^2 = 16 < 25$).
- $n = 30$**:** The iterations eventually yield `root = 6` because:
  - $6^2 = 36 \geq 30$,
  - $(6-1)^2 = 25 < 30$.

These tests convinced me that for these positive numbers the algorithm *appears* to compute the square root correctly.

(b) Revealing the Bug

I then tested the code with $n = 0$ using the temporary `Main()` method:

- $n = 0$**:**
  - Initial state: `root = 0`, `a = 0`.
  - Since the loop condition is `a > 0`, the loop never executes.
  - Final output: `root = 0`.
- **Postcondition check:** The specification requires:
  $$\texttt{root}^2 \geq n \quad \text{and} \quad (\texttt{root} - 1)^2 < n.$$
  For $n = 0$:
  - $0^2 = 0 \geq 0$ holds, but
  - $(0-1)^2 = 1$ is not less than 0.

Thus, the bug is that when $n = 0$ the postcondition $(\texttt{root} - 1)^2 < n$ is violated. In other words, the algorithm does not work correctly for $n = 0$.

(c) Making the Code Correct by Changing the Precondition

Since the intended postcondition

$$\texttt{ensures}\ \ \texttt{root}^2 \geq n \quad \text{and} \quad (\texttt{root} - 1)^2 < n$$

cannot hold for $n = 0$ (because $(-1)^2 = 1 \not< 0$), the simplest correction is to restrict the input to strictly positive numbers. Thus, we change the precondition from

```
requires n >= 0
```

to

```
requires n > 0
```

After this change, re-running the tests with $n = 4$, 25, and 30 still gives the correct outputs as before.

(d) Verification Status

At this point the Dafny code does **not** verify because the loop body is missing the annotations that tell Dafny how the state evolves. In particular, without a proper *loop invariant* and a *decreases* clause (to show termination), Dafny cannot prove the postcondition even though our informal testing shows that the algorithm works correctly for our tests.

Failure to verify does not mean the code is buggy,it only means that the verifier needs extra guidance (invariants and a termination measure).

(e) Adding the Invariant and Decreases Annotations

I determined the following annotations to help Dafny verify the code:

Invariant

The invariant expresses the relationship between `a` and `root`. Initially, we have:

$$a = n - 0^2 = n.$$

In each iteration, we update:

$$\texttt{root} := \texttt{root} + 1, \quad a := a - (2 \times \texttt{root} - 1).$$

If we let the old value of `root` be r, then after the update:

$$a = n - r^2 - (2(r+1) - 1) = n - (r+1)^2.$$

Thus, the invariant is:

```
invariant a == n - root * root
```

Decreases Clause

Since in every iteration, `a` decreases by a positive odd number (for $\texttt{root} \geq 1$, $2 \times \texttt{root} - 1 \geq 1$), a good choice is:

```
decreases a
```

By adding these annotations, Dafny can now verify that:

1. The invariant is maintained.
2. The loop terminates (since `a` is a natural number that decreases each iteration).
3. The postcondition follows upon termination.

The algorithm's body is unchanged; the annotations only provide extra information to the verifier.

(g) Computational Induction Proof of Total Correctness

Base Case

- **Initialization:** Before entering the loop, we set:

$$\texttt{root} := 0 \quad \text{and} \quad a := n.$$

- **Invariant Check:** The invariant states:

$$a = n - \texttt{root}^2.$$

  For $\texttt{root} = 0$, we have $a = n - 0 = n$. Hence, the invariant holds.

Inductive Step

Assume that before some iteration the invariant holds:

$$a = n - \texttt{root}^2.$$

Let the current value of `root` be r, so that $a = n - r^2$. During the loop body:

1. **Increment:** $\texttt{root} := r + 1$.
2. **Update:**

$$a := a - (2 \times (r + 1) - 1).$$

   Substituting the invariant:

$$a = (n - r^2) - (2r + 2 - 1) = n - r^2 - (2r + 1) = n - (r^2 + 2r + 1) = n - (r + 1)^2.$$

Thus, the invariant

$$a = n - \texttt{root}^2$$

is maintained for the new value of `root` (namely, $r + 1$).

Termination

The *decreases* clause ensures that in every iteration, `a` decreases by at least 1 (since $2 \times \texttt{root} - 1 \geq 1$ for $\texttt{root} \geq 1$). Given that `a` is a natural number (with the modified precondition $n > 0$), it cannot decrease indefinitely. Therefore, the loop terminates when $a \leq 0$.

Postcondition

At termination, we have $a \leq 0$ and by the invariant:

$$a = n - \texttt{root}^2 \leq 0 \quad \Longrightarrow \quad \texttt{root}^2 \geq n.$$

Furthermore, let $r' = \texttt{root} - 1$ be the value before the final iteration. At that point, the invariant was:

$$a' = n - (r')^2 > 0,$$

which implies:

$$(r')^2 < n.$$

Thus, when the loop terminates, the output `root` satisfies:

$$\texttt{root}^2 \geq n \quad \text{and} \quad (\texttt{root} - 1)^2 < n,$$

which is exactly the postcondition.

Conclusion

By computational induction, we have shown that:

1. The invariant $a = n - \texttt{root}^2$ holds initially and is maintained during each iteration.
2. The *decreases* clause guarantees termination.
3. The postcondition holds upon termination.

1 Problem 4: Progressive Dot Product

(a) Loop Invariant

A good loop invariant is as follows:

Invariant: At the start of each iteration with index i (where initially $i = 1$ and later $1 \leq i \leq$ a.Length), the following holds:

- c is a non-null array of length a.Length.
- For every index j with $0 \leq j < i$,

$$c[j] = \sum_{k=0}^{j} (a[k] \times b[k])$$

In words, each entry of `c` up to index $i - 1$ contains the cumulative (or progressive) dot product up to that index.

(b) Base Case

Before the loop begins, we set:

$$i := 1 \quad \text{and} \quad c[0] := a[0] \times b[0]$$

Since the only index j satisfying $0 \leq j < 1$ is $j = 0$, the invariant requires that:

$$c[0] = \sum_{k=0}^{0} (a[k] \times b[k]) = a[0] \times b[0]$$

which is exactly what we have. Also, the array c is correctly allocated with the same length as a. Thus, the invariant holds before entering the loop.

(c) Inductive Step

Assume that at the start of an iteration with index i (where $1 \leq i <$ a.Length) the invariant holds; that is, for every j with $0 \leq j < i$:

$$c[j] = \sum_{k=0}^{j} (a[k] \times b[k]).$$

During the i-th iteration, the code computes:

$$c[i] := c[i-1] + a[i] \times b[i]$$

By the induction hypothesis, $c[i-1] = \sum_{k=0}^{i-1}(a[k] \times b[k])$. Therefore,

$$c[i] = \left(\sum_{k=0}^{i-1}(a[k] \times b[k])\right) + a[i] \times b[i] = \sum_{k=0}^{i}(a[k] \times b[k]).$$

Then the index is incremented ($i := i + 1$). Thus, the invariant is maintained.

(d) Postcondition

The loop continues while $i <$ a.Length. When the loop exits, we have $i =$ a.Length. The invariant tells us that for every j with $0 \leq j < i$ (i.e. $j <$ a.Length):

$$c[j] = \sum_{k=0}^{j}(a[k] \times b[k]).$$

In particular, it immediately gives:

- $c[0] = a[0] \times b[0]$
- For every k with $0 < k <$ a.Length, we have

$$c[k] = \left(\sum_{j=0}^{k-1}(a[j] \times b[j])\right) + a[k] \times b[k] = c[k-1] + a[k] \times b[k].$$

Thus, the postcondition is satisfied.

(e) Decrementing Function

A suitable variant (decrementing function) is:

$$f(i) = \text{a.Length} - i.$$

- **Non-negative before the loop:** Initially, $i = 1$ so $f(1) = \text{a.Length} - 1 \geq 0$ (since a.Length > 0).
- **Decreases on each iteration:** In each iteration, i increases by 1, so $f(i)$ decreases by 1.
- **Loop termination:** When $f(i) = 0$ we have $i =$ a.Length, which is precisely the loop's exit condition.