The Correctness of Algorithms and Programs

CMPT 125
Feb. 1

A Puzzle For You

Problem: Write a program to output the first N cubes, but without using multiplication (only addition/subtraction).

Historically, CPUs are relatively slow at multiplication vs addition/subtraction.

- The differences can be small (3x) or large (20x).

```
N = 10

Output:

0
1
8
27
64
125
216
343
512
729
```

Lecture 13

Today:

- Assertions and Invariants
- Good Invariants and Post-Conditions
- Proving Programs Correct

Puzzle Solution

```
int main () {
    int N = 10;
    for (int i = 0; i < N; i++) {
        //  Compute square = i*i
        int square = 0;
        for (int j = 0; j < i; j++) {
            //  Assertion:  square ==
            i*i
            square += i;
        }

    //  Compute cube = i*i*i
        int cube = 0;
        for (int j = 0; j < i; j++) {
            cube += square;
        }
        printf("%d\n", cube);
    }
}
```

The Algorithm (Pseudocode):

For each i from 0 to N - 1:

- Compute the ith square by adding i to itself i times.
- Compute the ith cube by adding the ith square to itself i times.
- Output the ith cube.

Do you believe that at the end of this loop, the value of `square` will equal `i*i`?

Q. What's a good assertion?

Good assertions, also called *loop invariants*, are usually related to the post-condition.

What makes a *good* loop invariant?

A loop invariant is a statement that is true every loop.

- usually asserted at the beginning of the loop
- usually parametrized by the loop index (`j` in this case)

```
//  Post:  square == i*i
int square = 0;
for (int j = 0; j < i; j++) {
    //  Assertion:  square == j*I
    square += i;
}
```

A good loop invariant should indicate the progress of the algorithm

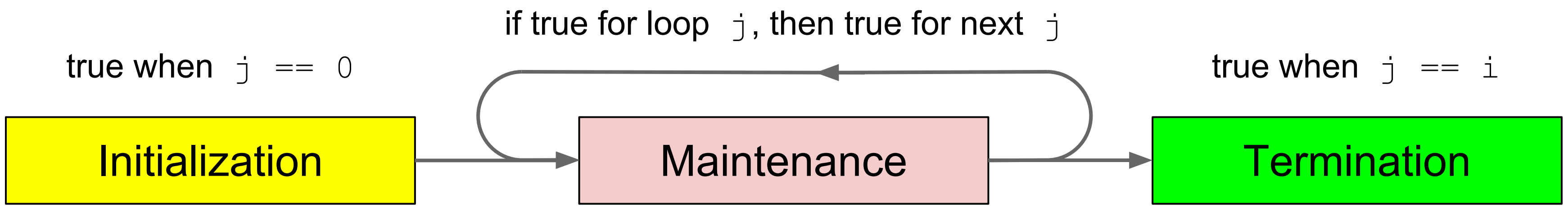
- the invariant should carry all state information, loop to loop.
- the invariant should imply the post-condition (the goal of the algorithm) at the end of the last loop.

Proving Correctness

```
//  Post:  square == i*i
int square = 0;
for (int j = 0; j < i; j++) {
    //  Assertion:  square == j*i
    square += i;
}
```

Use mathematical reasoning to capture the behaviour of an algorithm:

- State *invariants* at various *checkpoints*.
- Show that the invariant holds:
  - at the first checkpoint
  - during execution between checkpoints
- Conclude that the post-condition holds
  - the invariant holds at / after the last checkpoint



Proof

```
// Post: square == i*i
int square = 0;
for (int j = 0; j < i; j++) {
    // Assertion: square == j*i
    square += i;
}
```

Initialization:

- Is the invariant true on the first loop?
  - When `j == 0`, `square` has been initialized to 0. These values satisfy `square == j*i`.

Maintenance:

- If the invariant holds at the beginning of loop `j`, does it also hold for the beginning of loop `j+1`?
  - At the beginning of loop `j`, `square == j*i`.
  - After running the loop, `square == j*i + i == (j+1)*i`, which is the invariant of the next loop.

Termination:

- Since the invariant holds for all `j`, it holds after the last loop.
  - Therefore, when `j == i`, `square == i*i`.

Yay - we proved it! So what?

We honestly won’t care whether or not you can do a proof of correctness 5 years from now in your job. And neither will:

- your boss
- your co-workers
- ♥♥♥ your secret crush ♥♥♥

You learn to do proofs to get better at reasoning about code.

The more practiced you are at thinking about invariants:

- the better your resulting code will be
- the easier it will be to figure out other people’s code

A computer won’t be able to verify your programs for you

- in general, this is an impossible problem.

What does it do?

```
int main () {
    int N = 10;
    int a = 6;
    int b = 1;
    int c = 0;

    for (int i = 0; i < N; i++) {
        printf("%d\n", c);
        c += b;
        b += a;
        a += 6;
    }
}
```

Two ways to get started:

1. Simulate the execution on paper.
2. Key in the program and run it!

```
Output:

0
1
8
27
64
125
216
343
512
729
```

Q. Why is this program significant?

What are the invariants?

```
int main () {
    int N = 10;
    int a = 6;
    int b = 1;
    int c = 0;

    for (int i = 0; i < N; i++) {
        //  Assertion: a == 6*(i+1)
        //  Assertion: b == 3*i*(i+1) + 1
        //  Assertion: c == i*i*i
        printf("%d\n", c);
        c += b;
        b += a;
        a += 6;
    }
}
```

Initialization: When `i` = 0, the assertions are:

- `a` = 6(0+1) = 6
- `b` = 3·0·(0+1) + 1 = 1
- `c` = $(0)^3$ = 0

which are the 3 initial values for `a`, `b`, `c`.

Maintenance: At the start of loop `i`, the assertions are:

- `a` = 6(`i` + 1)
- `b` = 3`i`(`i` + 1) + 1
- `c` = i^3

After `c += b;` the value of `c` changes to

- `c` = $i^3 + 3i(i + 1) + 1$
  $= i^3 + 3i^2 + 3i + 1$
  $= (i + 1)^3$

After `b += a;` the value of `b` changes to

- `b` = $3i(i + 1) + 1 + 6(i + 1)$
  $= (i + 1)(3i + 6) + 1$
  $= 3(i + 1)(i + 2) + 1$

After `a += 6;` the value of `a` changes to

- `a` = $6(i + 1) + 6$
  $= 6(i + 2)$

which are the values for `a`, `b`, `c` on loop `i` + 1.

Since the assertion `c` = i^3 holds on every loop, the algorithm is correct.

What are the invariants?

```
int main () {
    int N = 10;
    int a = 6;
    int b = 1;
    int c = 0;

    for (int i = 0; i < N; i++) {
        //  Assertion: a = 6(i + 1)
        //  Assertion: b = 3i(i + 1) + 1
        //  Assertion: c = i^3
        printf("%d\n", c);
        c += b;
        b += a;
        a += 6;
    }
}
```

Since the assertion $c = i^3$ holds on every loop, the algorithm is correct.

Initialization: When $i = 0$, the assertions are:

- $a = 6(0+1) = 6$
- $b = 3\cdot 0\cdot(0+1) + 1 = 1$
- $c = (0)^3 = 0$

which are the 3 initial values for `a`, `b`, `c`.

Maintenance: At the start of loop `i`, the assertions are:

- $a = 6(i + 1)$
- $b = 3i(i + 1) + 1$
- $c = i^3$

After `c += b;` the value of `c` changes to

- $c = i^3 + 3i(i + 1) + 1$
  $= i^3 + 3i^2 + 3i + 1$
  $= (i + 1)^3$

After `b += a;` the value of `b` changes to

- $b = 3i(i + 1) + 1 + 6(i + 1)$
  $= (i + 1)(3i + 6) + 1$
  $= 3(i + 1)(i + 2) + 1$

After `a += 6;` the value of `a` changes to

- $a = 6(i + 1) + 6$
  $= 6(i + 2)$

which are the values for `a`, `b`, `c` on loop $i + 1$.

Famous Bugs

In the early 1960s, one of the American spaceships in the Mariner series sent to Venus was lost forever at a cost of millions of dollars, due to a mistake in a flight control computer program.

In 1981, one of the television stations covering provincial elections in Quebec, was led by its erroneous computer programs into believe that a small party, originally thought to have no chance at all, was actually leading. This information, and the consequent responses of commentators, were passed on to millions of viewers.

Famous Bugs

In a series of incidents between 1985 and 1987, several patients received massive radiation overdoses from Therac-25 radiation-therapy systems: three of them died from resulting complications. The hardware safety interlocks from previous models had been replaced by software safety checks, but all these incidents involved programming mistakes.

Famous Bugs

Some years ago, a Danish lady received, around her 107th birthday, a computerized letter from the local school authorities with instructions as to the registration procedure for first grade in elementary school. It turned out that only two digits were allotted for the “age” field in the database. This is similar in nature, but miniscule in scale, in comparison to the “Y2K bug”, where millions were spent on correcting programs that used two digits for the year, assuming a “standard” 1900-prefix.

Computers Do Not Err

Algorithms for computer execution are written in a formal unambiguous programming language

- Cannot be misinterpreted by the computer

A hardware error is such a rarity in modern computers that when our bank statement is in error and the banker mumbles something to the effect that the computer made a mistake, we can be sure that it was not the computer that erred. Either:

- incorrect data was input to a program; or
- the program itself contained an error

Compilers find syntax errors

Can also suggest common bug-fixes

What common warnings have you seen so far?

Testing and Debugging

The more you test your program, the more likely you are to find bugs. Test sets can find:

- run-time errors
- logic errors
- infinite loops

But results are only as good as your test sets.

- Some bugs might never be discovered.

Proving Correctness

Use mathematical proof techniques to reason about algorithms/programs.

Can we automate this proof process?

Does there exist some sort of "super-algorithm" that would accept as inputs: a description of a problem P and an algorithm A and say "yes" or "no"?

In general, this is just wishful thinking: no such verifier can be constructed

What about mathematical theorem proving?

4-color theorem

a proof of a theorem by a computer that could only ever be understood by a computer

Philosophically: The proof is only as strong as human fallibility, because there might be bugs in the theorem prover!

Example: Reversing a String S

reverse(“stressed”) returns “desserts”

head(“stressed”) = “s”

tail(“stressed”) = “tressed”

certainly for any nonempty S:

S = head(S) + tail(S)

```
X ← S; Y ← “”
while X not empty do:
    Y ← head(X) + Y
    X ← tail(X)
return Y
```

Adding Invariants & Checkpoints

```
X <- S; Y <- “”
while X not empty do:
    Y <- head(X) + Y
    X <- tail(X)
return Y
```

What’s a good invariant?

checkpoint 1: before the first loop

S = reverse(Y) + X

checkpoint 2: at the end of each loop

Proving the Invariant

```
X <- S; Y <- “”
chkpt 1: // inv:  S = reverse(Y) + X
while X not empty do:
    Y <- head(X) + Y
    X <- tail(X)
    chkpt 2: // inv:  S = reverse(Y) + X
return Y
```

Is it true at checkpoint 1?

Yes, because reverse(Y) + X = reverse(“”) + X = “” + X = X

Is it true at checkpoint 2?

Execute checkpoint to checkpoint: If the invariant is true at the beginning of the loop, then is it true at the end of the loop.

Proving the Invariant

Suppose that S = reverse(Y) + X at the beginning of some loop. Then after running the next loop Y’ = head(X) + Y and X’ = last(X)

So, just need to show that S = reverse(Y’) + X’.

reverse(Y’) + X’ = reverse(head(X) + Y) + last(X) = reverse(Y) + head(X) + last(X) = reverse(Y) + X = S.

One Last Detail

So, the invariant holds at the beginning of the first loops, and at the end of every successive loop. Including the last loop!

- When X = “”, the loop terminates and S = reverse(Y) + X = reverse(Y). Thus Y = reverse(S).

But does the loop terminate?

- Another invariant: that |X| is natural and decreasing.