SFU

Invariants and Recursion

CMPT 125
Mo Chen
SFU Computing Science
5/2/2020

Lecture 14

Today:

- Induction with strings
- Why Correctness is Important
- Invariants of Recursive Algorithms
- Running Time of Recursive Algorithms

Example: Reversing a String S

reverse(“stressed”) returns “desserts”

head(“stressed”) = “s”

tail(“stressed”) = “tressed”

X ← S; Y ← “”

while X not empty do:

    Y ← head(X) + Y

    X ← tail(X)

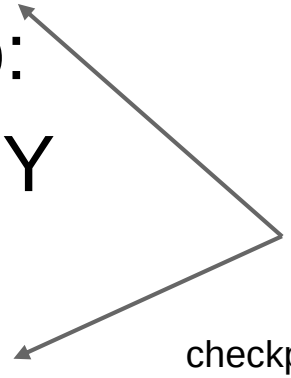
return Y

certainly for any nonempty S:

S = head(S) + tail(S)

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.

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:

$$Y' = \text{head}(X) + Y \text{ and } X' = \text{tail}(X)$$

Now check invariant:

$$\text{reverse}(Y') + X' = \text{reverse}(\text{head}(X) + Y) + \text{tail}(X)$$

$$= \text{reverse}(Y) + \text{head}(X) + \text{tail}(X)$$

$$= \text{reverse}(Y) + X = S.$$

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.

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.

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
- S = reverse(Y) + X = reverse(Y).
- Therefore Y = reverse(S).

But does the loop terminate?

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

Why Correctness is Important

Incorrect programs can be costly.

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.

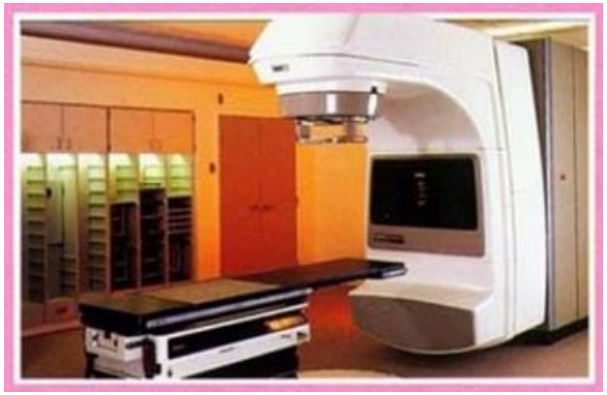
Mariner 1 Probe (1962)

Headline: “The most expensive hyphen in history.”

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.



Therac-25 (1986)

Therac-25 had been “improved” from previous models:

- the hardware safety interlocks from previous models had been “upgraded” by replacing them by software safety checks.

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.

- Program used two decimal digits to represent “age”.

This is similar in nature, but miniscule in scale in comparison, to the “Y2K bug”.

- Millions of programs used two digits for the year, assuming a “standard” 1900-prefix.

Not so Famous “Bug”

A few years ago, I was working with NASA on an automated emergency landing algorithm for planes with unexpected engine failure. In initial simulation testing, we found that the turn radius of the plane was 5 km (expected: a few hundred metres).

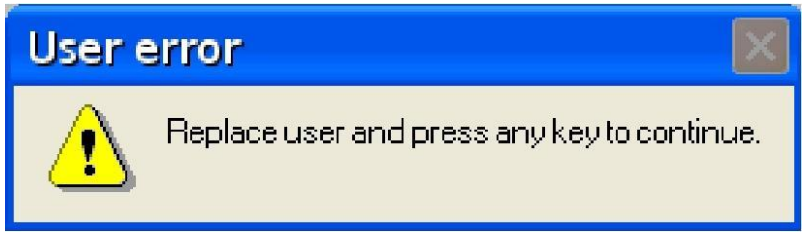
Thanks to well-tested and documented code, we figured out which part of the data was in metric, and which part was in imperial.

Computers Do Not Err

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

- Cannot be misinterpreted by the computer

Modern hardware is essentially bug-free. So, if our bank statement is in error and the banker mumbles 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

Similar programmer acronyms: PEBKAC RTFM

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.
- Q. Is a test set as strong as a proof of a loop invariant?

Proving Correctness

Use mathematical proof techniques to reason about algorithms/programs.

- E.g., assertions and loop invariants.

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 respond `"yes"` or `"no"`?

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

Loop Invariants (Review)

Use mathematical reasoning to capture the behaviour of an algorithm:

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

Last time:

```
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 == j*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 i^{th} square by adding i to itself i times.
- Compute the i^{th} cube by adding the i^{th} square to itself i times.
- Output the i^{th} 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.

```
#include <assert.h>
```

Last time:

```
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++) {
            assert(square == j*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 i^{th} square by adding i to itself i times.
- Compute the i^{th} cube by adding the i^{th} square to itself i times.
- Output the i^{th} 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.

Invariants and Recursion

Rule of Thumb: You may assume the invariant holds for any *smaller* case.

```
//  Post:  Returns  n!
unsigned int fac(unsigned int n) {
    if (n <= 1) {
        return 1;
    }
    return n * fac(n-1);
}
```

Definition of Factorial

$0! = 1$

$1! = 1$

$n! = n \times (n - 1)!$, when $n \geq 2$

Recursive sub call

Assumes that `fac(n-1)` [correctly] returns `(n-1)!`

Another Similar Example

Recursive Definition of b^e

$b^0 = 1$

$b^e = b \times b^{(e-1)}$ when $e > 0$

```
//  Post:  Returns base**exp
int power(int base, unsigned int exp) {
     if (exp == 0) return 1;
     return base * power(base, exp-1);
}
```

Again, you are allowed to assume that the recursive sub call to `power(base, exp-1)`, a smaller case, returns the correct value.

Q. What does the running time depend on?

- It varies with the value of `exp`

Let N be the value of the parameter `exp`

- $T(N) = O(1) + T(N - 1)$ when $N > 0$
- $T(0) = O(1)$

A *recurrence relation*!

Solution: $T(N) = O(N)$

Divide and Conquer Solution

Can you do better?

- Use Divide and Conquer

Key Observation:

- Can you be quick if `exp` is even?
- Can call `power(base, exp/2)` and square the result.

Remember that *any* smaller case is correct.

- Not only an incrementally smaller case.

Divide and Conquer Solution

```
int power(int base, unsigned int exp) {
  if (exp == 0) return 1;
  int x = power(base, exp/2);
  if (exp % 2 == 1) {
    return x * x * base;
  } else {
    return x * x;
  }
}
```

Q. What's the running time?

- Again, let N be the value of `exp`
- $T(N) = O(1) + T(N/2)$ when $N > 0$
- $T(0) = O(1)$

Solution: $T(N) = O(\log N)$

Sorting by Recursion

Use Divide and Conquer to sort recursively.

1. Split the array into two roughly equal pieces.
2. Recursively sort each half.
   - This works because each piece is *smaller*.
3. Join the two pieces together to make one sorted array.

Two famous sorts behave this way: *mergesort* and *quicksort*.