CS 213 M: Introduction

Abhiram Ranade

January 9, 2019

Syllabus

Proving correctness of programs. Analysis of algorithms. Review of recursion, dynamic memory management, class design.
Algorithms for sorting: mergesort, heapsort, quicksort.
Heaps/priority queues, Search trees, Tries.
Graph algorithms: depth first and breadth first search. Shortest paths.

What will not be covered: Dynamic programming, greedy algorithms.

Covered in CS 218 M

Textbooks and reading material

- An introduction to programming through C++. (CS 101 text)
- Algorithms. Dasgupta, Papadimitriou, Vazirani.
- Cormen, Leiserson, Rivest Stein.
- Notes and slides.

Rough Grading scheme and attendance

- 10 % : best $n - x$ out of n Assignments.
- 40 % : Total for 3 quizzes.
- 50 % : End semester exam.

For the most part you will be expected to write C++ code;
Pseudocode not allowed.

You are more fluent in C++ than in English
You will be allowed to bring one book to all exams.

Attendance: 80 % compulsory. May give early DX.

General Instructions:

- Write clearly, in the best handwriting you are capable of.
  May not grade papers if handwriting is bad.
- Write full sentences.
- When writing programs, you are expected to explain and generally use good style.
- When asked to prove that a program is correct, giving an example is usually useless.
- When asked to prove that a program is incorrect, giving an example is usually the best way.
- Please follow instructions. If asked to use a certain method to solve a problem, use that and not any other as far as possible.
  Even if you have a "better" method. I may just want to test if you have understood what I have taught.

Outline for today

How to argue that a program is correct

- Correctness of straight line code
- Correctness of loop based programs
- Correctness of recursive programs

Should we worry about correctness?

People have written incorrect programs which have led to

- Aircraft crashes
- Machinery not working
- Possibility of hacking into code

When you write a program you will have to argue that it is correct

- To your boss..
- To your teacher/TAs..
- To your programmer collegues..
- To yourself!

How do we argue that a program is correct?

Strategy 0: Wave your hands and say "It is obviously correct!"

It may be obvious to you but not to others.

Can we agree about what is obvious and what is not?

Strategy 1: Run many tests.

If no errors discovered, declare program correct.

How do we know we have done enough testing?

Strategy 2: Prove programs correct

Next.

Basics of program correctness proofs

Key Idea: Write down what each statement in the language can do.

These are axioms which do not need proof.

Use these to prove what programs do.

Short cut: It is a bit laborious to write down everything about all statements, so we will assume that we all know it.

Example: Consider the statement `i = k + j;`

Based on what we know about C++, we can say that after execution

- variable `i` will have the sum of variables `k,j`
- No other variable will change in value.

Idea 1: Effect of several statements can be deduced by considering the effect of each.

Easy Example: program which has no loops or branching

"Straight line code"

Correctness of straight line code

1. `int inches, feet, yards;`
2. `cin >> inches; // Let n = what user types`
3. `feet = inches/12;`
4. `yards = feet/3;`
5. `feet = feet % 3;`
6. `inches = inches % 12;`
7. `cout << inches <<',’<<feet<<’,’<<yards<< endl;`

Claim: inches, feet, yard equivalent of n inches is printed.

Proof: After step 2, `inches` $= n$. C++ axiom

After step 3, `feet` $= \lfloor n/12 \rfloor$. C++ axiom

After step 4, `yards` $= \lfloor \lfloor n/12 \rfloor /3 \rfloor$ C++ axiom

After step 5, `feet` $= \lfloor n/12 \rfloor \bmod 3$. C++ axiom

After step 6, `inches` $= n \bmod 12$. C++ axiom

After step 6, inches < 12, feet < 3 $p \bmod q < q$

yards*36 + feet*12 + inches $= n$. Math manipulation

Math manipulation

yards*36 + feet*12 + inches

$= \lfloor \lfloor n/12 \rfloor /3 \rfloor * 36 + (\lfloor n/12 \rfloor \bmod 3) * 12 + n \bmod 12$

$= (\lfloor \lfloor n/12 \rfloor /3 \rfloor * 3 + (\lfloor n/12 \rfloor \bmod 3)) * 12 + n \bmod 12$

$= (\lfloor n/12 \rfloor) + n \bmod 12$

$= n$

Previous Page

$\lfloor p/q \rfloor * q + (p \bmod q) = p$

$\lfloor p/q \rfloor * q + (p \bmod q) = p$

Remarks

- Language designer only guarantees what single statement does.
  Effect of single statement is "obvious"/axiomatic.
- For multiple statements we need to compose the effects of the different statements into a proof.
- Proof = Sequence of claims/assertions about values taken by variables after each statement in terms of values typed in by the user.
- Requirement of a proof:
  - Let P = claim about values before execution of a statement
  - Let Q = claim about values after execution of that statement.
  - Let R = what we know about the statement based on knowledge of C++.
  - P, R should imply Q.

Correctness of looping programs

Proof has two parts:

- If program terminates it will produce the correct answer.
  - Proved by making assertions about values taken by variables before/after each statement in the program.

    Each statement may be executed several times.
    Assertions must hold for every execution.
    Similar to straight line code but trickier.
- The program terminates.

Proof becomes long and complex even for simple programs.

- Often we provide a "Proof sketch" instead of a full proof.
- Proof sketch should contain the main ideas.
- Proof sketch should "explain why the program works".
- Proof sketch should be precise: refer to values of variables at specific points in the program.
- You should be able to extend the proof sketch if needed.

Proof sketch for GCD program

```
int m,n; cin >> m >> n;  // Assume user types in M, N > 0.

while(m % n != 0){
  int r = m % n;
  m = n;
  n = r;
}
cout << n << endl;
```

Lemma 1: On entry to while loop, gcd(M,N) = gcd(m,n). m,n > 0.

Math fact: Suppose m, n > 0.

- If m % n $= 0$ then gcd(m,n) $=$ n.
- If m % n > 0 then gcd(m,n) $=$ gcd(n, m % n).

Lemma 2: n decreases in each iteration.

So no infinite loop, because n > 0.

Lemma 3: Before print statement, gcd(M,N) $=$ gcd(m,n), m % n $= 0$.

So correct value, n, will be printed.

Proof of Lemma 1

Lemma 1: On entry to while loop, gcd(M,N) = gcd(m,n). m,n > 0.

Proof: Induction on number of visits to while loop.
Base case: Clearly true on first visit, because m,n = M, N.

Induction: Assume true on previous visit. Prove for current.
Assume values of m, n on previous visit were m', n'.
So induction hypothesis: gcd(m',n') = gcd(M,N). m',n' > 0.
Loop was entered $\Rightarrow$ m' % n' != 0. So r > 0.
m = n'.
n = m' % n'.
gcd(m,n) = gcd(n', m' % n')
Analysis similar to analysis of straight line code
= gcd(m', n') Math fact
= gcd(M, N) Induction hypothesis

m = n' > 0, n = m' % n' > 0.

So proved by induction.

Proof of Lemma 2 and Lemma 3

Lemma 2: n decreases in each iteration.
Proof:
Value of n at entry to loop > 0. Lemma 1
New value of n is m % n.
But remainder mod n is always less than n.

Lemma 3: Before print statement gcd(m,n) = gcd(M, N). m % n = 0.
Proof:
Control arrives at print statement from entry of while loop.
At entry of while gcd(m,n) = gcd(M,N), which continues to hold.
Loop was not entered, so m % n = 0.

Remark

Loop Invariants: The assertion `gcd(m,n) = gcd(M,N), m,n>0` is expected to hold on entry to each iteration.

Such assertions are called loop invariants.

Potential: The assertion `n decreases in each iteration` is useful often called a potential function argument.

"System stabilizes when potential energy of a system goes to 0."

Proof of recursive programs

Typical strategy: Induction.

Define a notion of "problem size"

Problem size for gcd: n

Problem size for problems on sequences: sequence length

Show that program runs correctly for a (small) problem size.

Assume correct till certain problem size, prove for higher size.

Recursive GCD

```
int gcd(int m, int n){
 if (m % n == 0) return n;
 else return gcd(n, m % n);
}
```

GCD(m,n): the greatest common divisor of m,n.

Induction hypothesis IH(N): gcd(m,n) is correct for all m $>$ 0, n $=$ N.
Base case: N $=$ 1.
gcd(m,1) returns 1 because m % 1 $==$ 0. Correct

Assume IH(1), IH(2), ... IH(N). We prove IH(N+1)
If m % N+1 $=$ 0, then N+1 is returned. correct
So assume m % N+1 $>$ 0.
This returns gcd(N+1, m % N+1). m % N+1 $\leq$ N
So by assumption this correctly returns GCD of N+1, m % N+1.
But this is equal to GCD of m, N+1. GCD fact
Correct in this case too.

Remarks

- Recursive functions are often more compact.
- Proofs of recursive functions are also often more compact.
- We did not need to talk about "value of m in previous iteration", because in the given scope value of m did not change.

"Functional programming style: no reassignment"

Important convention: We can use the name of the variable to denote its current value or to refer to the variable itself; this has to be clear from the context.

To refer to older values we should define notation explicitly, just as we defined m' to mean the value of m in the previous iteration.

Additional remark:

The analysis done to argue correctness is useful also for understanding other properties of the program, e.g. its running time. Also, analysis may reveal possibilities for improvement.

Exercise 1

Prove the correctness of the following program which does multiplication without computing products.

```
int m, n, p = 0; cin >> m >> n;

while(m > 0){
 p = p + n;
 m = m - 1;
}

cout << p << endl;
```

Exercise 2

Complete the following program so that it calculates m^n = m to the power n. Ensure that the given invariant is correct.

```
int m,n,p=1; cin >> m >> n;  // user types M, N.

while(n > 0){   // p * m^n = M^N
  if(n % 2 == 0){
    m = m*m;  n = ___;
  }
  else{
    p = ___;  n = n - 1;
  }
}
cout << p << endl;
```

We often first think of the invariant and then design the code.

How many multiplications does this code do for computing 2^57?
How does this compare to the standard method?

Exercise 3

Show that the value of n reduces by a factor of at least 2 in 2 iterations of the gcd program. If m,n are 32 bit integers, how many iterations will be sufficient to find the gcd, no matter what the precise values of m,n are?

Consider the execution of the gcd program with m, n given the values of the kth and k+1th Fibonacci numbers. How many iterations will this take?

Use induction to argue that the kth Fibonacci number is at most 2^k. Say the first two Fibonacci numbers are 1, 1. Based on this you may observe that running gcd on k bit numbers will take k iterations for some m,n.

Exercise 4

Prove the following function correct. It decides if some two elements of array a of length n sorted in non-decreasing order add up to t.

```
bool sumSearch(int *a, int n, int t){
  int i=0, j=n-1;
  while(i<j){
    if(a[i] + a[j] < t) i++;
    else if(a[i] + a[j] > t) j--;
    else return true;
  }
  return false;
}
```

The invariant will is somewhat complicated. Also note that for something to hold after an if statement, it has to hold at the end of the consequent (then part) and also the alternate (else part).

What is the maximum number of iterations possible in terms of n?