CIS 212

Introduction to Computer Science III

David Atkins
Spring 2006

Course Info

www.cs.uoregon.edu/classes/06S/cis212
Software Development

- Real world software is developed by large teams, consists of huge amounts of code
- Development takes too long, costs too much, not high quality
- Maintenance (fixing bugs, adding features) is a significant cost

Software Development Life Cycle

- Requirements specification
- Design
- Risk Analysis
- Verification
- Coding
- Testing
- Maintenance
The Life Cycle of Software

Figure 1.1
The life cycle of software as a water wheel that can rotate from one phase to any other phase

Software Engineering Practices

- Object Oriented Design
  - Encapsulation
  - Data Hiding
  - Abstract Data Types
  - Code and design reuse
- Design Patterns
Algorithmic Problem Solving

- We want to design a computer program to solve a problem. The solution will typically consist of an algorithm that performs operations on data.
- An algorithm is:
  - a well ordered collection of unambiguous, computable operations
  - that produces a result when executed and halts in a finite amount of time

Examples of problems

- Multiply two numbers: 23 x 357
- Find the largest number in a list: 13, -10, 24, 0, 3, -71, 80, 12, 15
- Solve a scrambled Rubik's cube.
- What data and steps are involved?
Algorithm Design

- How to describe an algorithm?
  - Primitive operations, e.g.,
    - assignment, calling method, arithmetic, comparison, indexing array, etc.
  - Repetitions (loops)
- Top down design
  - First describe in non-primitive terms
  - Then refine until primitives

Algorithm Example

- Find the area (in square feet) of a house.
- High level approach
  - Add the areas of the rooms in the house
- How to find the area of a room?
  - Multiply room dimensions – a primitive op
Algorithm Pseudo Code

current room = first room;
result = current room width \times length;
while (there is another room in house)
  current room = next room;
  result = result +
    (current room width \times length);
return result;

Algorithm Analysis

- Is the algorithm correct?
  - Under what conditions?
  - Why?
- If the house has more rooms, does the time required by the algorithm grow?
  - How?
  - What if the rooms are bigger?
Algorithm Design

- Many problems, many possible solutions for each problem
- Some common patterns for algorithms
  - Decomposition
  - Dynamic Programming
  - Search algorithms
    - Backtracking, Branch and bound, ...

Decomposition

- Divide and conquer strategy
  - Partition the problem into smaller parts which can be more easily solved
  - Combine results to solve whole problem
- By Iteration
  - Loops – often used on arrays, indexed lists
- By Recursion
  - Recursive function – often used on linked lists, trees
Iterative Decomposition

- General form of iteration
  - Initialize overall solution
  - while (there are more elements)
    - select next partition
    - solve the problem for that partition
    - combine result with overall solution
  - Return overall solution

Iteration Example

- Problem: Given a non-empty array of integer values, find the minimum value in the array.
- Partition by elements in the array
- Smaller problem is to find minimum of single element
- How to initialize overall solution?
- How to combine with overall solution?
Minimum Example

function array minimum (given array of n integers, \( n \geq 1 \))

\[
\text{Initialization – minimum of one element array}
\]

\[
\text{Initialization – minimum of one element array}
\]

\[
\text{Return overall solution}
\]

Another approach

function array minimum (given array of n integers, \( n \geq 1 \))

\[
\text{Initialization – minimum of one element array}
\]

\[
\text{Return overall solution}
\]
Sum Example

function array sum (given array of n integers, \( n \geq 0 \))

\[
\text{asum} = 0 \quad \text{Initialization – sum of empty array is zero}
\]

for \( i \) from 1 through \( n \) by 1

\[
\text{asum} = \text{asum} + i^{\text{th}} \text{ integer} \quad \text{Loop over all elements}
\]

\[
\text{asum} = \text{asum} + i^{\text{th}} \text{ integer} \quad \text{Sum of a single element is itself}
\]

return asum \quad \text{Combine local sum with overall sum}

Return overall solution

Iterative Decomposition

- Design Methodology
  - Determine solution to initial trivial problem
  - Determine how to iterate through all partitions and halt
  - Determine how to solve local subproblems (what operations to use)
  - Determine how to combine local solution with overall solution
Algorithm Correctness

- How do we know if our iteration algorithm is correct? Can we prove it?
- Pre and post conditions
  - Start with preconditions, demonstrate that algorithm steps lead to postconditions
- Invariants
  - Conditions that are always true
  - Loop invariants – true before and after each loop execution

Proving Correctness

- As comments in the program:
  - Identify pre and post conditions
  - Identify loop invariants
    - Prove true after initialization
    - Prove invariant maintained by loop body

- Formal verification is difficult to do in general, but can be very effective analysis for small chunks of algorithms
Pre and Post Conditions

- Can be expressed as the transformation performed by the algorithm
  - Input given (e.g., method arguments given)
  - Output returned (e.g., method return value)
  - Side effects (e.g., values changed, output)
  - Description of nature of transform
- Similar to Java API method doc

Loop Invariants

- A loop invariant is a property that is true
  - Before the loop begins (but after any initialization steps)
  - Before every iteration of the loop
  - After every iteration of the loop
  - After the loop completes
Correctness with Loop Invariants

- Prove iterative algorithm is correct by
  - Show invariant true before loop starts
  - Show loop execution preserves invariant
  - Show invariant captures intent of algorithm
  - Show the loop terminates
- All of these together prove the correctness of the iterative algorithm
  - Can also use to prove loop is wrong

Example: sum of array

```java
int sum(int[] array) {
    int result = 0;
    int i = 0;
    while (i < array.length) {
        result += array[i];
        ++i;
    }
    return result;
}
```
sum of array annotated

// Given an array of integers (which may be empty)
// Return the sum of the integers in the array
// With no side effects on the array
int sum(int [] array) {
    int result = 0; // Sum of empty array
    int i = 0;
    // Invariant: result is sum of elts 0 to i-1
    while (i < array.length) {
        // Invariant: result is sum of elts 0 to i-1
        result += array[i];
        // result is sum of elts 0 to i-1 plus elt i
        ++i;
        // Invariant: result is sum of elts 0 to i-1
    }
    // Invariant: result is sum of elts 0 to i-1
    // and i is array.length
    return result;
}

Another Example

Problem: Find the largest three values in an array of integers (assume at least three integers in array)

Decompose

- Base case: array of just three
- Iteration (induction): adjust by replacing (if necessary) a max value with additional value
maxThree

// Given an array A of at least three integers
// Return an array of the three largest values of A
// With no side effects on the array A
int[] maxThree(int[] A) {
    int[] result = new int[3];
    int i = 3;
    // Invariant: result has max 3 of A[0..i-1]
    while (i < A.length) {
        // Invariant: result has max 3 of A[0..i-1]
        replaceMin(result, A[i]); // replace minimum by A[i]
        ++i;
        // Invariant: result has max 3 of A[0..i-1]
    }
    // Invariant: result has max 3 of A[0..i-1], i is A.length
    return result;
}

replaceMin

// Given a non-empty array M of integers, another integer n
// Return nothing
// With the smallest value in M replaced with n if n is larger
void replaceMin(int[] M, int n) {
    int m = 0, i = 1;
    // Invariant: m is index of smallest of M[0..i-1]
    while (i < M.length) {
        // Invariant: m is index of smallest of M[0..i-1]
        if (M[i] < M[m]) m = i;
        ++i;
        // Invariant: m is index of smallest of M[0..i-1]
    }
    // Invariant: m is index of smallest of elements of M
    if (M[m] < n) // If smallest less than n
        M[m] = n; // Replace smallest by n
    
}
Loop Invariants

- Proving correctness by loop invariants is really proof by induction
  - Assertion of truth of base case
  - Proof of step (if true for case k-1, then true for case k)
- Conclude assertion is true at end of loop
  - Verify that it is desired result