Correctness of algorithms

Two issues:

- Given an algorithm, prove that it is correct (always achieves the intended result, e.g. a sorted array).
- Design an algorithm with intended properties from scratch (even more difficult)

Additional reading

- Frank M. Carrano, Janet J. Prichard. *Data abstraction and problem solving with Java.* Addison Wesley Longman, 2001. Chapter 1, Problem solving and software engineering (on verification).
- Duane A. Bailey. *Data structures in Java for the principled programmer*. McGraw-Hill 1999. Chapter 2, Comments, conditions and assertions (pre- and postconditions).
- Roland Backhouse. *Program Construction : Calculating Implementations from Specifications.* John Wiley & Sons 2003.

Disproving correctness

- Just one counterexample is enough
- Testing may fail to discover a bug

Proving correctness

- Formulate precisely the property which has to hold
- If necessary, formulate relevant properties for smaller parts of an algorithm : program assertions

Assertions

- Assertion: claim about values of program variables before or after a statement or a group of statements is executed

  Typical assertions:
- Precondition (usually of a method): what we expect to hold before the method is executed.
- Postcondition: what holds after the method is executed.

Hoare triples

- {P} S {Q}: P precondition, S statements, Q postcondition.
- Meaning: provided P holds before S is executed, then after S is executed, Q holds.
- for example:

{x < 10} **`x = x + 20`** {x < 30}

{x < y} **`while (x < y) x++`** {x=y}

- For a small programming language, can provide axioms for every construct in the language and derive postconditions using axioms

Assignment axiom

- If the only programming construct was assignment, here is an axiom to verify all programs:

{Q(e substituted for x)} **`x = e`** {Q}

- For example, if want to prove

{x < 10} **`x = x + 20`** {x < 30} then the assignment axiom gives

{x+20 < 30} **`x = x + 20`** {x < 30} and from extra knowledge about maths etc we derive that {x+20 < 30} is equivalent to {x < 10}.

Example

- In the programs we usually write there are lots of constructs and they also use other people's code.
- Less formal approach (but good practice): write pre- and postconditions for significant chunks of code/methods.
- Example: code in Bailey's book.

Example

```
public static void sort(int[] arr, int len)
  // pre: len is the length of the array arr
  // post: arr is sorted in ascending order
```

(sorted means:

for all indices i such that 0 <= i < len-1, arr[i] <= arr[i+1],

or arr[0] <= arr[1] <=…<= arr[len-1])

Correctness

To prove that an algorithm is correct:

- Determine preconditions and postconditions for the whole algorithm.
- Cascade statement assertions together, so that postconditions for one provide preconditions for the next.
- Prove correctness of individual statements.
- Hence show that executing algorithm with stated preconditions terminates and leads to stated post-conditions.

Example (sort+reverse=sort in reverse order)

```
public static void sort(int[] arr, int len)
  // pre: len is the length of the array arr
  // post: arr is sorted in ascending order
public static void reverse(int[] arr)
  // post: the order of elements in arr is
  // reversed (e.g. [9 5 10] -> [10 5 9])
// A1: the length of arr is len
sort(arr,len);
// A2: arr is sorted in ascending order:
```

arr[0] <= arr[1] <=…<= arr[len-1]

```
reverse(arr);
// A3: the order of arr is reversed, hence
```

arr[0] >= arr[1] >=…>= arr[len-1]

Loop Invariants

- Assertions for loops are less trivial, because loops may be executed many times over, with slightly different preconditions at the beginning of each iteration. Focus on those preconditions that remain constant between iterations.
- Known as l*oop invariants:* true before and after each iteration through a loop.

Example

```
pos_greatest = 0;
for (int j = 0; j < =i; j++) {
   if( arr[j] > arr[pos_greatest]) {
      pos_greatest = j;
   }
}
```

Invariant: `pos_greatest` is the index of the largest array element between 0 and j inclusive.

(More formally, for all k such that `0 <=k<=j`, `arr[k] <= arr[pos_greatest]. )`

Correctness of loops

To prove correctness of a while loop (or: that assertion A holds after the loop terminates):

- Prove that the loop eventually terminates (by finding the *bound function* for the loop)
- Find a suitable invariant (there are infinitely many invariants for each loop, most of them useless)
- Prove that A is true after last iteration (usually by substituting the state in which the loop terminates in the invariant)

Partition algorithm

```
red = l;  // set at the left border of the
          // range
blue = r; // set at the right border where the
          // pivot sits
while(red < blue) {
   if (arr[red] < pivot) red++;
   else {
        blue--;
        swap(arr, red, blue);
    }
}
swap(arr, blue, r); // put the pivot on the
                    // border
return blue;
```

Postcondition for the partition

```
public int partition(int[] arr, int l, int r)
// post: returns and integer k such that
//       for all indices i such that l<=i<k,
//       arr[i]<arr[k] and
//       for all indices i such that k<=i<r
//       arr[i]>=arr[k]
```

Example

Correctness proof for the partition algorithm:

- loop invariant: for all indices i,
  (if l <= i < red then arr[i] < pivot) and
  (if blue <= i < r then arr[i] >= pivot)
- bound function = `blue - red`
- decreases by 1 at every step; the loop terminates when it is equal to 0 (`blue = red`).
- When `blue = red = k` the invariant becomes: for all indices i, (if l <= i < k then arr[i] < pivot) and (if k <= i < r then arr[i] >= pivot)
- When the pivot is swapped, pivot = arr[k].

Informal coursework

- Informal coursework is on the web.
- Will be discussed in tutorials next week.