CS1110 26 March 2009

Developing array algorithms: initial considerations

Arrays are often the most efficient way to store and work with data, so we need to know how to solve problems involving arrays.

- We've added more consulting hours, so they are now: Mon-Thu 4pm-10pm, Sat 2pm-6pm, ACCEL green room
- Reading for next time: same as today, 8.5.

- A6 out, due Sat. April 11. You get to design your own solutions to an interesting problem ☺! But this means you have to design your own solutions to a significant problem ☹. We suggest reading through A6 ASAP, and starting early.
- Graded prelim 2s available up front.

Recall that an **invariant** is an assertion about variables/state that is true before and after execution.

Working with invariants takes practice, but is well worth it:

"democracy is the worst form of Government except all those other forms that have been tried" (Churchill)

We have to "start simple" today to get the basics down, but keep in mind: In "real" life (and/or CS1110), thinking about (array) invariants helps with:

- writing the book-keeping parts of your program (thus preventing bugs); but also
- coming up with main solution ideas to hard problems; and
- understanding what (your or others') code is doing

The previous methodology, with some addenda:

First, specify the algorithm by giving its precondition and postcondition, *using a pictorial notation*.

Then, **find an invariant** *by drawing another picture* that "generalizes" the precondition and postcondition, since the invariant is true at the beginning and at the end.

Then, answer the 4 loopy questions (which you might as well memorize):

1. How does loop start (how to make the invariant true)?
2. When does it stop (when is the postcondition true)?
3. How does repetend make progress toward termination?
4. How does repetend keep the invariant true?

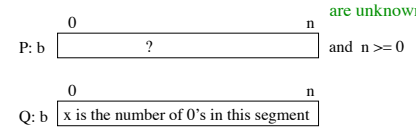
Invariant as picture: Combining pre- and post-condition

Here's a simple problem we can already solve, just to introduce notation.

Count the number of zeroes in an array. Given array b satisfying precondition P, store a value in x to truthify postcondition Q:

Formal, albeit pictorial, notation:

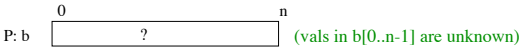
(values in b[0..n] are unknown)



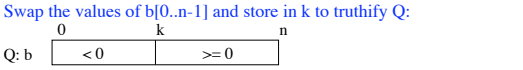
The invariant as picture: Combining pre- and post-condition

Put negative values before non-negative ones via swaps.
(Application: separating different types of items; faster than sorting.)

Given precondition P:



Swap the values of b[0..n-1] and store in k to truthify Q:

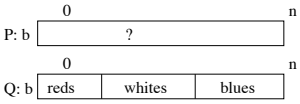


(values in b[0..k-1] are < 0,
values in b[k..n-1] are >= 0)

5

The invariant as picture: Combining pre- and post-condition

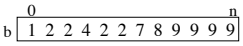
Dutch national flag. Swap values of 0..n-1 to put the reds first, then the whites, then the blues. That is, given precondition P, swap values in b[0.n-1] to truthify postcondition Q:



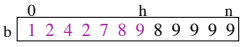
6

Remove adjacent duplicates

change:



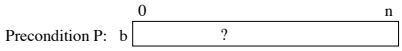
into

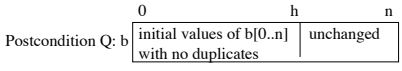


don't care what is in b[h+1..n]

Truthify:

b[0..h] = initial values in b[0..n] but with adj dups removed





7