CS 1110: WORKED EXAMPLES REGARDING LOOPS AND INVARIANTS

http://www.cs.cornell.edu/courses/cs1110/2014fa/materials/loop_invariants.pdf

D. GRIES, L. LEE, S. MARSCHNER, AND W. WHITE

An invariant is, in the end, a way to *document the meaning of your variables.* Inadvertently changing the meaning of variables partway through your code is the cause of many, many bugs that are often very hard to track down, because in your head you are thinking of one particular meaning for a variable and don't see the change that happened. This is why we are stressing the importance of writing code that maintains an invariant. The general principle is to pre-commit to what you want your variables to mean, then write down your promise, then check to make sure your code keeps your own promise. (If you notice that keeping your promise is making it difficult to solve the problem you set out to tackle, that's usually a sign that you need to re-think your approach and what your variables are keeping track of; in that case, start over at the "pre-commit" step.)

In this class, we use a particular notation for expressing invariants that we've found to be particularly convenient over the years, but in your future programming adventures, just remember the general principle: in comments or docstrings, clearly document the meaning of your variables whenever confusion could arise.

One technique for developing an invariant is to start with a clear statement of the precondition ("what do we know before the code runs?" which is usually "not much") and the postcondition or desired goal ("what are we supposed to know after our code runs?", which is usually "everything"). Then, the invariant "interpolates" between the two, covering both as special cases: it usually describes the situation where we "know something" (which may be nothing, or everything), and it often includes new variables needed to describe that "something".

When you do the worked examples in Sections 3 and later, *your answers usually need to* **exactly** *match ours in order to be correct*, unless you're the type of person who likes to increment indices *before* doing swaps, in which case, adjust accordingly.[1] You should certainly make sure you understand how the code below is correct.

Code implementing the worked examples below can be found at

http://www.cs.cornell.edu/courses/cs1110/2014fa/materials/loop_invariants.py

Run the module to see charmingly primitive ASCII "animations" of all the implemented algorithms in action.

1. Warm-Up

1.1. **Counting the values in `b[x..y-1]`.** Here, we practice using the formula for the number of items in a given range. This is important, because in order to determine loop termination conditions, you often need to figure out when there are no more items left to process.

How many items are in the segment `b[x..y-1]`?

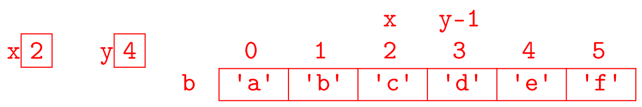
Solution: Remember the mnemonic "follower minus first". This means that `b[i..j]` has $j + 1$ (follower) minus i (first), or $j + 1 - i$ items. Hence, something of the form `b[m..m-1]` or `b[m+1..m]` is empty (has no items in it); something of the form `b[p..p]` has one item in it, namely, `b[p]`.[2]

So `b[x..y-1]` has $y - x$ items in it.

[1]And unless there's typos in this handout...

[2]Don't confuse the ranges denoted by our "dot-dot" notation with the ranges denoted by Python's `range` function. Python's `range(n)` means `0..n-1`.

Example:



`b[x..y-1]` is `b[2..3]`, which contains the two items `'c'` and `'d'`.

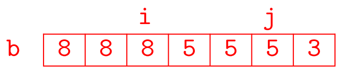
1.2. **Drawing List Diagrams.** Draw a list `b` that satisfies these conditions

`b[0..i] > 5, b[i+1..j] = 5, b[j+1..] < 5.`

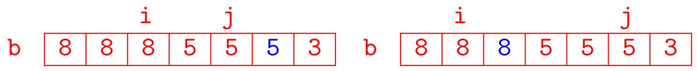
Include the indices `i` and `j` in your drawing.

The notation `b[k..]` means all the items from index `k` all the way to the end of the list, including of the last item. Be careful whether you put an index to the left or the right of the relevant "vertical bar".

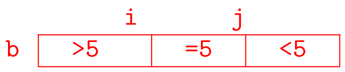
Solution: Here is a specific example.



Incorrect responses (problem indicated in blue):



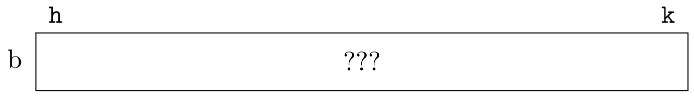
General form of solution:



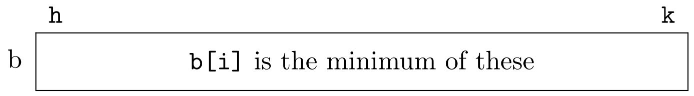
2. Finding the Location of the Minimum of a List

While we can use `min(b)` to find the minimum *value* in list `b`, we often want the *index* of that minimum element in only a single pass through the list (as opposed to calling `b.index(min(b))`). The following are assertions for an algorithm to find a position in `b` of the minimum of `b[h..k]`:

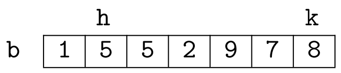
Precondition: `0` $\leq$ `h` $\leq$ `k < len(b)`
Since $h \leq k$, `b[h..k]` must contain at least one item. If we allowed $k = h - 1$, the list could have been empty, but what is the minimum of an empty list? Also note that because $k <$ `len(b)`, k may be the index of the last item in `b`, but it does not "go beyond" the end of `b`.



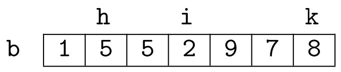
Postcondition: `i` is an index such that `b[i]` is the minimum of `b[h..k]`



For example, if we start with this (so that h is 1 and k is 6):



then we should end with i being 3:

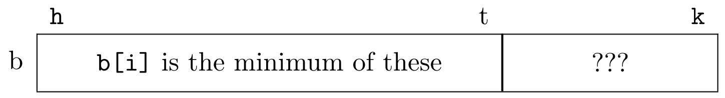


In what follows we present several different loop invariants that generalize the above precondition and postcondition. You are to write a loop (with initialization) for each one.

2.1. **Invariant P1.** General idea: t marks the right end of the portion whose minimum value we know, and i is an index where that minimum value can be found.

Text version: `b[i]` is the minimum of `b[h..t]`.

Pictorial version:



Write your loop, including initialization, for this invariant:

Solution: In this case, we need to begin with knowing "something", in order to initialize i. It is easy to determine where the minimum of the list `b[h..h]` is. It is too philosophical for us to consider the question of where the minimum of the empty list `b[h..h-1]` is.

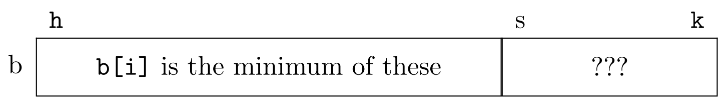
```
# Inv says b[t+1..k] are '???'.  Start:  b[h+1..k] '???'; end:  b[k+1..k] '???'.
i = h; t = h
while (t < k):  # still have '???'  to process
    # peek at next '???'
    if b[t+1] < b[i]:
        i = t+1 # location of smaller value
        t = t+1 # update knowledge, progress towards termination
    else:  # the next unknown is at least as big as the known min
        t = t+1 # update knowledge, progress towards termination
    # Also OK to have only one t = t+1 line, placed here (outside the if/else).
```

And here is an ASCII-art rendering of the algorithm in action. Here, `b` is `[1,5,5,2,9,7,8]` with $h = 1$ and $k = 6$. The "animation", which is produced by adding appropriate printing after the initialization step and at the end of the repetend (loop body) in the code above, runs down the first column and then down the second; the periods allow you to see the exact position of each index drawn above the list in question.

```
......k       ......k
.h.....       .h.....
.t.....       .....t.
.i.....       ...i...
1552978       1552978


......k       ......k
.h.....       .h.....
..t....       ......t
.i.....       ...i...
1552978       1552978


......k
.h.....
...t...
...i...
1552978


......k
.h.....
....t..
...i...
1552978
```

2.2. **Invariant P2.** General idea: s marks the left end of the unknowns, and i is an index where the minimum value of items from index h up to but not including s can be found.

Text version: `b[i]` is the minimum of `b[h..s-1]`.
Pictorial version:

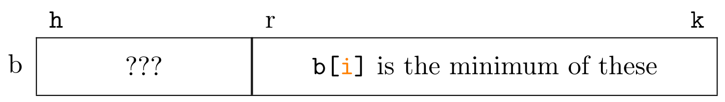


Solution:

```
# Inv says b[s..k] are '???'.  Start:  b[h+1..k] '???'; end:  b[k+1..k] '???'.
i = h; s = h+1
while (s < k+1):  # still have '???'  to process
    # peek at next '???'
    if b[s] < b[i]:
        i = s # location of smaller value
        s = s+1 # update knowledge, progress towards termination
    else:  # the next unknown is at least as big as the known min
        s = s+1 # update knowledge, progress towards termination
    # Also OK to have just one s = s +1 line, placed here (outside the if/else)
```

2.3. **Invariant P3.** General idea: r marks the left end of the portion whose minimum value we know, and i is an index where that minimum value can be found.

Text version: `b[i]` is the minimum of `b[r..k]`.
Pictorial version:



Solution:

```
# Inv says b[h..r-1] are '???'.  Start:  b[h..k-1] '???'; end:  b[h..h-1] '???'.
i = k; r = k
while (r > h):  # still have '???'  to process
    # peek at next '???'
    if b[r-1] < b[i]:
        i = r-1 # location of smaller value
        r = r-1 # update knowledge, progress towards termination
    else:  # the next unknown is at least as big as the known min
        r = r-1 # update knowledge, progress towards termination
    # Also OK to have just one r = r - 1 line, placed here (outside the if/else)
```

3. Partitioning on a Fixed Value

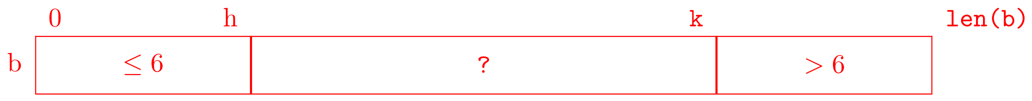
The purpose of the algorithms in this question is to swap the values of list b and to store a value in k so that the postcondition given below is true. List b is not necessarily sorted initially. The precondition and postcondition are as follows:

Precondition: `b[0..]` $= ?$ (i.e. nothing is known about the values in b)
Postcondition: `b[0..k]` ≤ 6 and `b[k+1..]` > 6

Here is the invariant we want you to maintain in your code. Note that it introduces the new variable h; the general idea is that h marks the right end of the items known to be ≤ 6, and k marks the right end of the unknown items.

P3.1: `b[0..h]` ≤ 6 and `b[k+1..]` > 6

Solution:



Write your loop, with initialization, for this invariant.

Solution:

```
# Inv says b[h+1..k] are '?'.  Start:  b[0..len(b)-1] '?'; end:  b[h+1..h] '?'.
h = -1; k = len(b)-1
while k > h:  # still have '???'  to process
    # peek at next '?'
    if b[h+1] <= 6:
        h = h+1 # update knowledge, progress towards termination
    else:  # the next unknown is bigger than 6
        b[h+1], b[k] = b[k], b[h+1] # swap to put big val with right-hand brethren
        k = k-1 # update knowledge, progress towards termination
```

4. The General Partitioning Algorithm

The general partitioning algorithm is an integral component of the supremely elegant (and, in practice, often quite efficient) *quicksort* sorting algorithm. Quicksort works as follows: given a non-empty list segment `b[h..k]` to sort (if the segment is empty, quicksort simply returns immediately), quicksort *partitions* the list so that there is an index i where `b[h..i] <= b[i] <= b[i+1..k]`; then, it applies itself recursively to the segments `b[h..i-1]` and `b[i+1..k]`, which necessarily results in the desired segment `b[h..k]` being itself sorted.

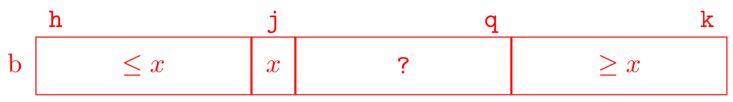
Your task is to implement the "true" partition algorithm, which is a generalization of what was considered in the previous section, according to the invariant given below. Again, list `b` is not necessarily sorted initially.

Precondition: `0` $\leq$ `h` $\leq$ `k` $<$ `len(b)`. Let x be the value of `b[h]`.
(this is just so we can talk about `b[h]` initially; `x` is not a program variable.)
Postcondition: `b[h..j-1]` $\leq x =$ `b[j]` $\leq$ `b[j+1..k]`.

P4.1: `b[h..j-1]` $\leq x =$ `b[j]` $\leq$ `b[q+1..k]`

General idea: j is where the pivot value is currently stored; q marks the right end of the unknowns. Draw the pictorial representation of this invariant.

Solution:



Write your loop, with initialization, for this invariant.

Solution:

```
# Inv says b[j] is pivot, b[j+1..q] '?'.  Start:  b[h+1..k] '?'; end:  b[j+1..j] '?'.
j=h; q=k
while (q > j):  # still have '?'  to process
    # peek at next '?'
    if b[j+1] <= b[j]:
        b[j+1], b[j] = b[j], b[j+1]
        j = j+1 # location of pivot value moves
    else:  # next ?  is bigger than b[j]
        b[j+1], b[q] = b[q], b[j+1] # swap to put big val with right-hand brethren
        q = q-1 # update knowledge, progress towards termination
```