CSC148H Lecture 7

Dan Zingaro
OISE/UT

October 27, 2008

Two Python Functions

TPS: Is the first function correct? Is the second function correct?

```
def search (a, target):
  found = False
  i = 0
  while (i < len(a)):
    if a[i] == target:
      found = True
    i = i + 1
  return found

def search (a, target):
  found = False
  i = 0
  while (i < len(a)):
    i = i + 1
    if a[i] == target:
      found = True
  return found
```

Program Correctness

- There is no notion of correctness without a specification of what a piece of code should do
- If we wanted to write a program that sometimes bombed out with an exception, the second program on the previous slide is correct according to this requirement
- We can give a specification using a pair of assertions
  - Precondition: conditions under which our code will operate correctly
  - Postcondition: what our program promises to achieve

Linear Search with Pre- and Post-conditions

```
def search (a, target):
  #Precondition: a is a list
  #Postcondition: returns true if and only if
    #target is in a
  found = False
  i = 0
  while (i < len(a)):
    if a[i] == target:
      found = True
    i = i + 1
  return found
```

Now is it Correct?

- We have to argue that, under the stated precondition, we always reach the required postcondition
- Since `found` is set to `False` prior to loop execution, we return the correct value when the list is empty
- How about with a list of one element?
  - If the list has one element and it equals `target`, we will set `found` to `true` and return the correct result
  - If the list has one element and it does not equal `target`, we will keep `found` with value `false` and return the correct result
- We can't possibly do this sort of reasoning for lists with many more elements!

Loop Invariants

- What we can do instead is make a claim that holds on each iteration of the loop
- Such a claim is called a loop invariant
- After the loop terminates, the loop invariant will still hold
- If the loop invariant is strong enough to allow us to conclude our postcondition, we have an informal argument that our program is correct

Loop Invariants...

- Here are some loop invariants for our linear search
  - $2 + 3 = 5$
  - `a` does not change throughout execution of `search`
  - `target` does not change throughout execution of `search`
- These invariants do hold throughout the execution of the loop, but don't help us conclude our postcondition

Linear Search Loop Invariant

"`found` is `True` if and only if `target` exists among elements `a[0..i-1]`"

- The notation $a[0..i-1]$ refers to elements $a[0], a[1], \ldots, a[i-1]$
- In general, the notation $a[i..j]$ refers to elements $a[i], a[i+1], \ldots, a[j]$
- We must show that this loop invariant is
  - true prior to loop execution, and
  - maintained by an arbitrary loop iteration

Linear Search Loop Invariant...

- We can split the invariant into its "only if" and "if" parts
- Only if: "If `found` is `True`, then `target` exists among elements `a[0..i-1]`"
- If: "If `target` exists among elements `a[0..i-1]`, then `found` is `True`"
- Both parts hold prior to loop execution
  - Only if: holds because premise is false
  - If: holds because empty segment can never contain an element

Linear Search Loop Invariant...

- Under the premise that the invariant holds, both parts are preserved by the loop
- Only if: "If `found` is `True`, then `target` exists among elements `a[0..i-1]`"
  - Yes: if we set `found` to `true` in the loop, we must have found `target`
- If: "If `target` exists among elements `a[0..i-1]`, then `found` is `True`"
  - Yes: we do not skip past an occurrence of `target` without setting `found` to `true`

Using the Invariant

- Invariant: "`found` is `True` if and only if `target` exists among elements `a[0..i-1]`"
- Once the loop terminates, we know that the negation of the loop guard holds. (The loop guard was `i < len(a)`.)
- We want to combine this with our loop invariant to conclude that `i == len(a)`, from which we can conclude the program's postcondition
- TPS: But, can we really conclude that `i == len(a)` from the negation of the guard and the invariant? If not, what do we have to do?

Invariants are not Enough

```
def search (a, target):
  #Precondition: a is a list
  #Postcondition: returns true if and only if
    #target is in a
  found = False
  i = 0
  while (i < len(a)):
    pass
  return found
```

- Our invariant holds for this function as well; but, for anything but the empty list, it does not terminate!
- Invariants are not concerned with termination
- They tell us that, if a function does terminate, we know something about the program's state
- By contrast, a variant is a numeric value that is concerned with termination
- A variant is a quantity that is
  - Always ≥ 0
  - Decreased by each iteration of the loop

Fully Annotated Linear Search

```
def search (a, target):
  #Precondition: a is a list
  #Postcondition: returns true if and only if
    #target is in a
  found = False
  i = 0
  while (i < len(a)):
    #Invariant: found is true iff target exists
    #in a[0..i-1];
    #0 <= i <= len(a)
    #Variant: len(a) - i
    if a[i] == target:
      found = True
    i = i + 1
  return found
```

Developing with Invariants

- Invariants can be used to write programs that are correct by construction
- First, we characterize the postcondition we want to achieve
- Then, we weaken the postcondition to arrive at an invariant that
  - is easy to establish
  - allows us to conclude the postcondition upon loop termination
- Claiming correctness then amounts to showing that the invariant is preserved on each iteration and that a variant exists

Longest Plateau

- Goal: write a function that returns the length of the longest plateau in a sorted list
- A plateau is a segment whose elements are all equal
- e.g. in the list $[1, 1, 1, 2, 3, 3]$, the longest plateau is of length 3
- The postcondition is `p` is the length of the longest plateau in `a`, where `a` is the input list
- Loop invariant: `p` is the length of the longest plateau in `a[0..i-1]`
- We can initialize `i = 0` and `p = 0` prior to loop execution to establish the invariant

Longest Plateau...

- Loop invariant: `p` is the length of the longest plateau in `a[0..i-1]`
- Before we increment `i` in the loop, how can we update `p` to maintain the invariant?
- We must account for all new plateaus ending at `a[i]` (all other plateaus are already covered by the invariant)
- Approach 1
  - In an inner loop, scan leftward, counting list elements until we find one that is unequal
- Approach 2
  - Observation: `p` can increase by at most 1 (why?)
  - So we can simply check `a[i] == a[i-p]`; if true, we increment `p`

Fully Annotated Length of Longest Plateau

```
def lengthPlateau (a):
  #Precondition: a is a sorted list
  #Postcondition: returns length of longest plateau in a
  i = 0
  p = 0
  while i < len(a):
    #Invariant: p is length of longest plateau in a[0..i-1]
    #Variant: len (a) - i
    if a[i] == a[i-p]:
      p += 1
    i += 1
  return p
```

Majority Winner

- Goal: given a list of votes, determine the majority winner, if any
- A majority winner is defined as a candidate receiving more than half of the votes
- Approach 1
  - Scan through the list, incrementing each candidate's total when they receive a vote
  - Problem: amount of memory is proportional to the number of candidates

Majority Winner...

- Our approach will be to characterize the progression of the loop by an invariant
- The votes will be stored in list `v`
- We will use `cand` to store any majority winner, and `i` to step through the votes
- Here's an invariant: `cand` is the only possible majority winner among the votes in `v[0..i-1]`
- It is really difficult to extend this invariant to take `v[i]` into account, though
- We don't know "how much of a majority" `cand` has; the next vote may well change who the majority winner is

Majority Winner...

- Let's additionally maintain variable `k`, representing the number of unmatched votes for `cand`
- If we line up the voters for `cand` on one side and all other voters on the other, an unmatched vote is a vote that is not opposed by another voter
- New invariant: among the votes in `v[0..i-1]`, there are `k` unmatched votes for `cand`; the remaining `i-k` votes can be paired so that paired voters disagree
- After processing all votes, `cand` is the only possible majority winner (all other candidates have opposing votes for each of their votes, so can have no more than half of the votes)

Majority Winner...

- Invariant: among the votes in `v[0..i-1]`, there are `k` unmatched votes for `cand`; the remaining `i-k` votes can be paired so that paired voters disagree
- How do we update the variables based on the vote in `v[i]`?
- If `k == 0`
  - There is no majority winner in the first `i` votes
  - The only possible majority winner is the candidate in `v[i]`
  - This new majority winner will have `k = 1`, since this is the only unmatched vote
- If `k` is not 0
  - If `v[i]` is another vote for `cand`, just increment `k`
  - Otherwise, `v[i]` is not a vote for `cand`. This vote thus opposes one vote for `cand`, so we decrement `k`

Majority Winner...

- When we have inspected all votes, we have found the only possible majority winner
- We still have to check if this candidate actually holds a majority, though
- We do this by making another scan through the votes to determine if our possible majority winner owns more than half of the votes
- If it is guaranteed that our list of votes contains a majority, we don't have to do this step
- Let's implement this algorithm ...

Time Analysis

- How can we measure the size of the problem in
  - Linear search?
  - Longest plateau?
  - Majority vote?
- What are their orders of growth?