LOOP INVARIANTS

CS340

Definition

- A loop invariant is a loop property that holds before and after each iteration of a loop.
- The loop invariant tells us the strategy an algorithm uses to solve a problem.
- It assumes that with successive iterations, the data move closer to the correct solution (which we call *truth*).

Where is a loop invariant true?

Here is the general pattern of the use of Loop Invariants in your code:

```
...
// the Loop Invariant must be true here
while ( TEST CONDITION ) {
   // top of the loop
   ...
   // bottom of the loop
   // the Loop Invariant must be true here
}
// Termination + Loop Invariant = Goal = Truth
...
```

An algorithm to add the numbers 1..n

WHAT IS THE LOOP INVARIANT?

This algorithm works by walking the array and adding numbers as it goes along.

At the kth iteration of the loop, the variable “answer” contains the sum of the first k numbers.

```
def sum(A):
   answer = 0
    for i=0,...,len(A)-1
         answer += A[i]
    return answer
```

If you want, think of A[0]..A[i] as a subarray, kind of like we did with insertion sort.

An algorithm to add the numbers 1..n

Loop Invariant Proof:

INITIALIZATION:

Before the loop begins, “answer” contains 0. Also, no numbers have been added. On the 0th iteration of the loop, ”answer” contains 0, so the loop invariant is trivially true.

```
def sum(A):
   answer = 0
    for i=0,...,len(A)-1
        answer += A[i]
    return answer
```

An algorithm to add the numbers 1..n

Loop Invariant Proof:

MAINTENANCE:

Each iteration of the loop adds A[i] to answer.

Therefore on the first iteration of the loop, the first 1 numbers are added into answer. On the second iteration the 2nd 2 numbers are added into answer, etc.

Therefore, the loop invariant is maintained.

```
def sum(A):
   answer = 0
    for i=0,...,len(A)-1
        answer += A[i]
    return answer
```

An algorithm to add the numbers 1..n

Loop Invariant Proof:

TERMINATION:

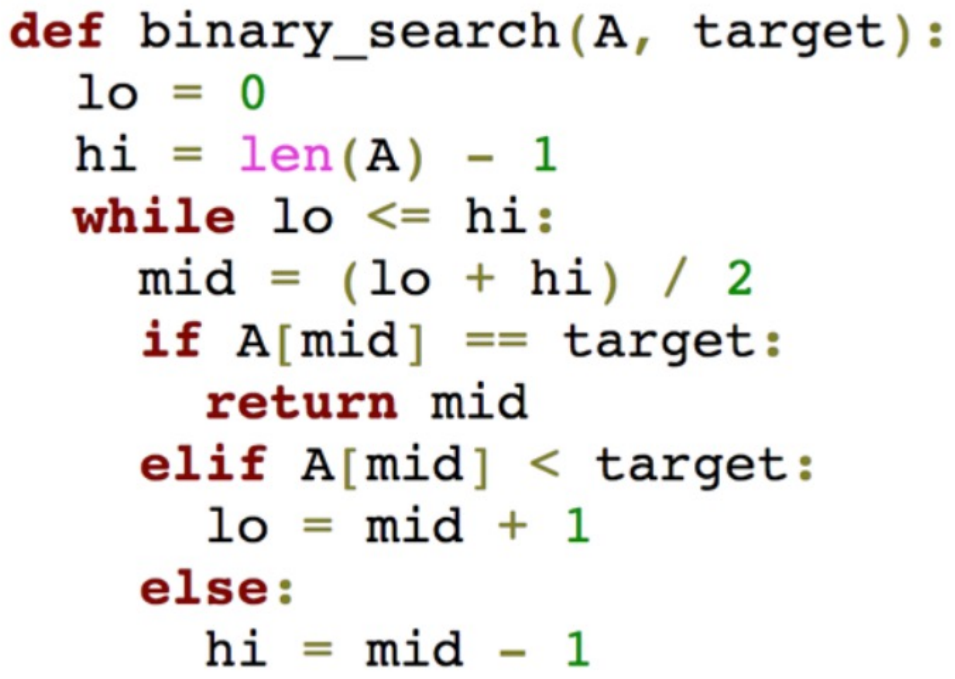
The loop finishes when i = len(A), which is beyond the last cell of the array. Therefore answer contains the sum of the first Len(A) numbers, which is all the numbers. Therefore, the algorithm correctly adds all the numbers.

```
def sum(A):
   answer = 0
    for i=0,...,len(A)-1
        answer += A[i]
    return answer
```

This example is from this web page: https://www.win.tue.nl/~kbuchin/teaching/JBP030/notebooks/loop-invariants.html

Binary Search

7	8	32	42	88	103	105	297



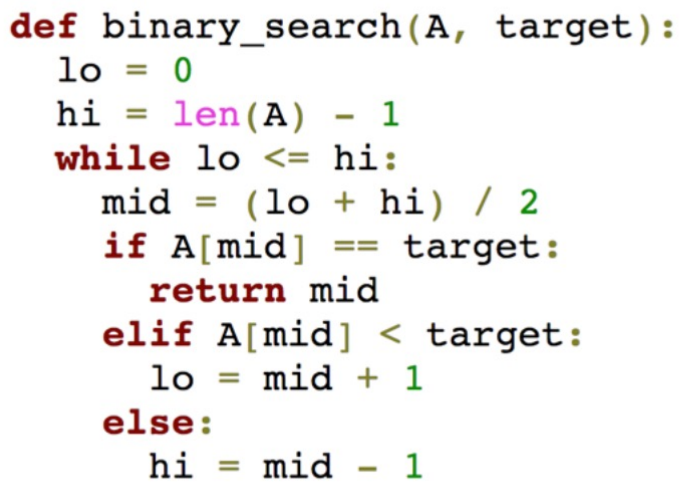
```
def binary_search(A, target):
  lo = 0
  hi = len(A) - 1
  while lo <= hi:
    mid = (lo + hi) / 2
    if A[mid] == target:
      return mid
    elif A[mid] < target:
      lo = mid + 1
    else:
      hi = mid - 1
```

Let’s say we’re looking for 103

Binary Search Loop Invariant

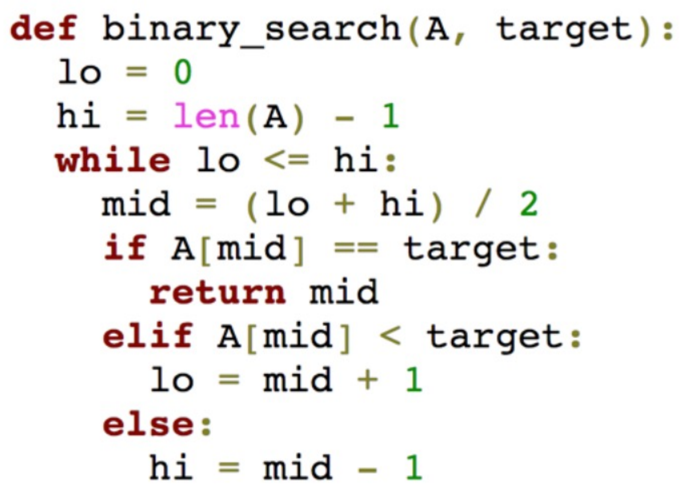
- At each iteration of the loop, the target is bounded by lo and hi.
- A[lo] ≤ target ≤ A[hi]

- INITIALIZATION: lo=0, hi=A.length-1
- All numbers are bounded by lo and hi, therefore target must be bounded by lo and hi.



```
def binary_search(A, target):
  lo = 0
  hi = len(A) - 1
  while lo <= hi:
    mid = (lo + hi) / 2
    if A[mid] == target:
      return mid
    elif A[mid] < target:
      lo = mid + 1
    else:
      hi = mid - 1
```

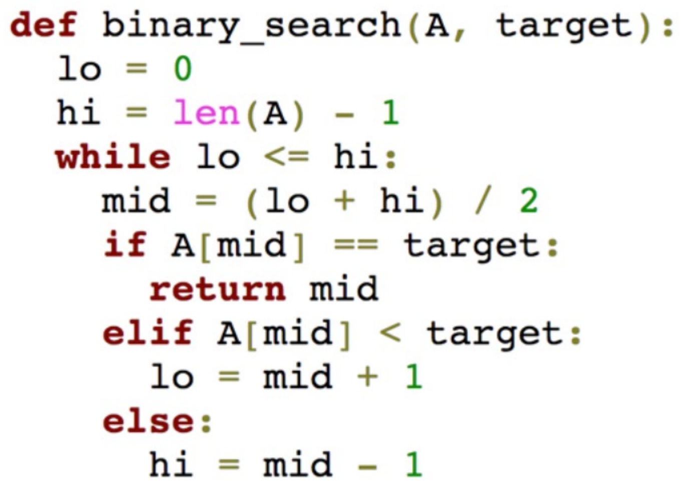
Binary Search



```
def binary_search(A, target):
  lo = 0
  hi = len(A) - 1
  while lo <= hi:
    mid = (lo + hi) / 2
    if A[mid] == target:
      return mid
    elif A[mid] < target:
      lo = mid + 1
    else:
      hi = mid - 1
```

- At each iteration of the loop, the target is bounded by lo and hi.
- A[lo] ≤ target ≤ A[hi]
- MAINTENANCE
  Does the loop maintain the property?
- Case 1: If A[mid] > target, then the target must be between lo and mid • We update hi = mid – 1
- Case 2: If A[mid] < target, then the target must be between mid and hi • we update lo = mid + 1
- YES the loop maintains the property!

Binary Search



```
def binary_search(A, target):
  lo = 0
  hi = len(A) - 1
  while lo <= hi:
    mid = (lo + hi) / 2
    if A[mid] == target:
      return mid
    elif A[mid] < target:
      lo = mid + 1
    else:
      hi = mid - 1
```

- At each iteration of the loop, the target is bounded by lo and hi.
- A[lo] ≤ target ≤ A[hi]

- TERMINATION
- Notice for each iteration, lo always increases and hi always decreases. These values will converge at a single location where lo = hi.
- Because target is bounded by lo and hi, it must be that A[lo] = target = A[hi]

A Software Engineering Tip

- Loop invariants capture key facts that explain why code works.
- This means that if you write code in which the loop invariant is not obvious, you should add a comment that gives the loop invariant.
- This helps other programmers understand the code, and helps keep them (or you!) from accidentally breaking the invariant with future changes.