Binary Search

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 8.2

Introduction

- Binary search is a classic example that illustrates general recursion
- We will look at a function for binary search

Things to notice about this case study

- Use of invariants to make sure that code is correct
- Use of halting measure to guarantee termination
  - Justification relies on the invariant (!)
- Use of Java illustrates that our tools work in other languages
- Iterative loop illustrates how our tools work in imperative code.

Learning Objectives

- At the end of this lesson you should be able to:
  - explain what binary search is and when it is appropriate
  - explain how the standard binary search works, and how it fits into the framework of general recursion, invariants, and halting functions
  - write variations on a binary search function

Binary Search

- Given an array **A[0:N]** of non-decreasing integer values and a target **tgt**, find an **i** such that **A[i]** = **tgt**, or else report not found.

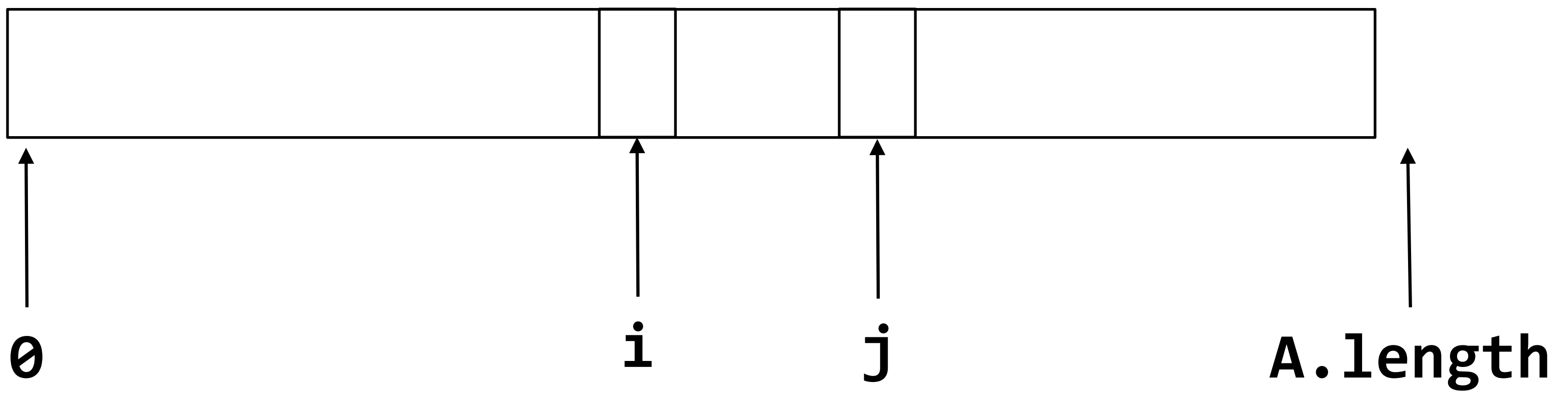
We will use Java arrays

- In Java, we declare an array variable as **int[] A**
- The length of the array is written as **A.length**, and the valid indices into such an array go from 0 to **A.length**-1.
- (An array can be empty, with A.length = 0). For binary search, we want A to be *non-decreasing*, that is:

  (for all i,j)((0 <= i <= j <= A.length) ➔ A[i] <= A[j])
- For the rest of this case study, when we say "A is non-decreasing," this is what we mean.

A picture of a non-decreasing array

(for all i,j)((0 <= i <= j <= A.length) → A[i] <= A[j])



A[i] <= A[j]

Pictures like this turn out to be very useful. Notice that this picture tells us that the indices into the array range from 0 to A.length -1

Our Purpose Statement

GIVEN: a non-decreasing array of ints A and a target 'tgt'

RETURNS: a number k such that
0 <= k < A.length and f(k) = tgt
if there is such a k, otherwise returns -1

Let's do the obvious generalization

- Instead of searching from 0 to A.length-1, we can search an arbitrary range in the array.
- We don’t want to lose any solutions, so we need to make sure that if tgt exists anywhere in the array, it exists in [lo,hi-1].

Purpose Statement for the generalized function

```
GIVEN: two integers lo and hi, a non-decreasing
array of ints A, and a target tgt
WHERE: 0 <= lo <= hi <= A.length
AND    (forall j)(0  <= j < lo ==> A[j] < tgt)
AND    (forall j)(hi <= j < A.length ==> A[j] > tgt)

RETURNS: a number k such that lo <= k < hi and f(k)
= tgt if there is such a k, otherwise -1.
```

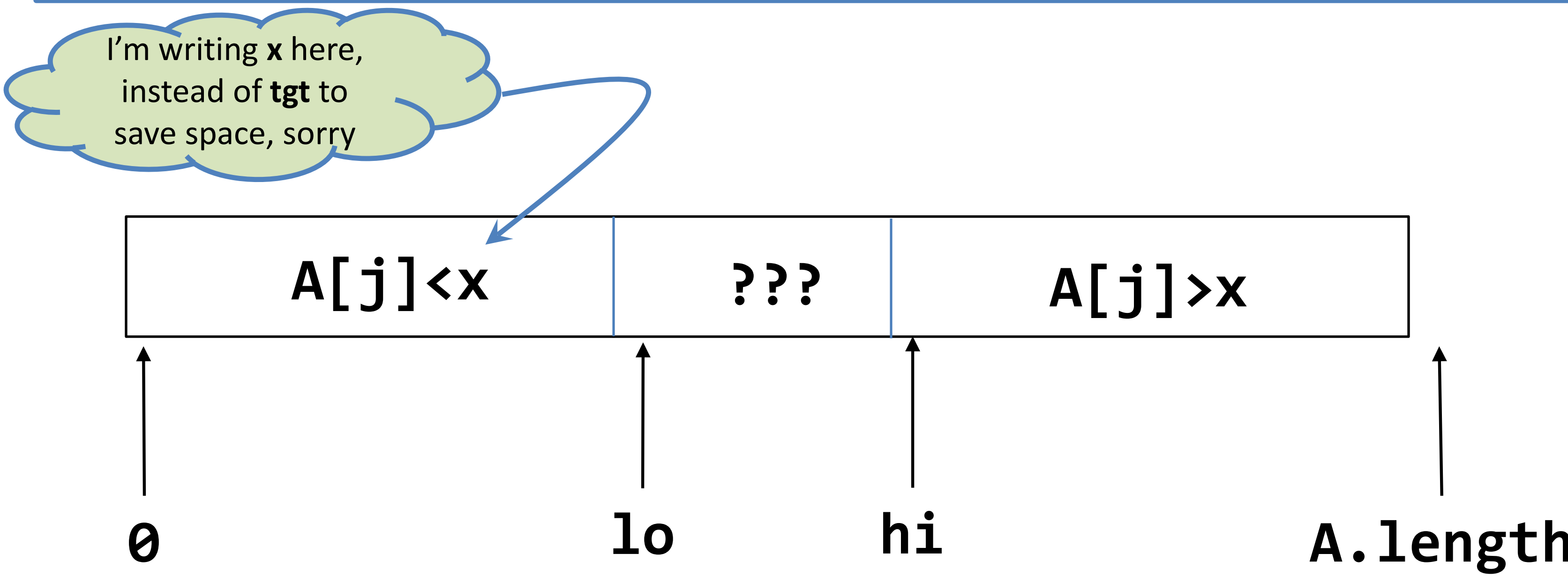
I've highlighted the occurrences of the new arguments

Make sure that there are no occurrences of tgt in the array outside of [lo,h-1]

This invariant divides the array into three regions:

- 0 <= j < lo where A[j] < tgt
- lo <= j < hi where we don't know anything
- hi <= j < A.length where A[j] > tgt

A picture of our invariant



Notice that the arrows point just to the right of the boundary. This tells us which region A[lo] and A[hi] belong to. Similarly, the 0 and the A.length are just to the right of the boundary.

Drawing the arrows just to the right or just to the left of the boundary prevents many off-by-one errors.

Now we can write the main method

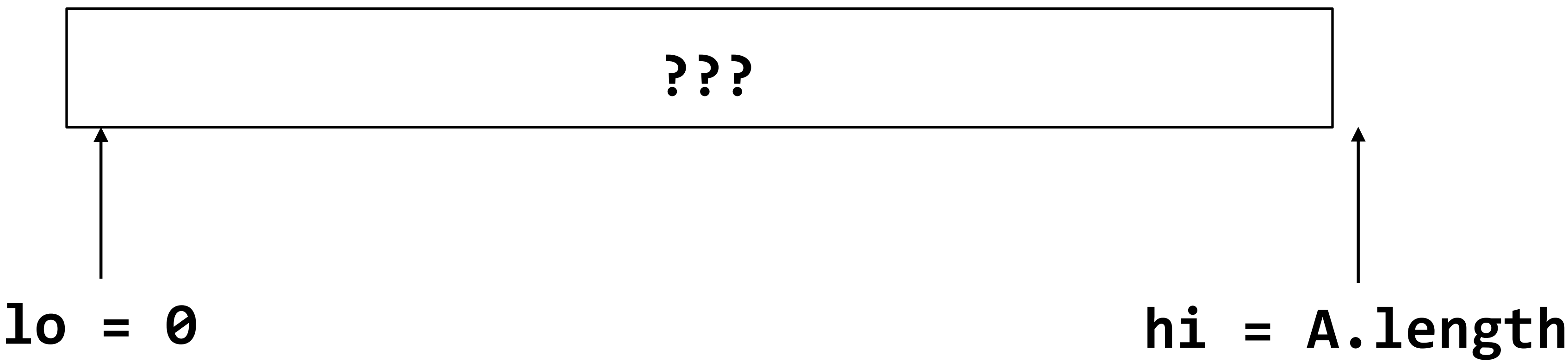
```
static int binsearch_recursive (int[]A, int tgt) {

   // GIVEN: two integers lo and hi, a non-decreasing
   // array of ints A, and a target tgt
   // WHERE: 0 <= lo <= hi <= A.length
   // AND   (forall j)(0  <= j < lo ==> A[j] < tgt)
   // AND   (forall j)(hi <= j < A.length ==> A[j] > tgt)

   // RETURNS: a number k such that lo <= k < hi and f(k)
   // = tgt if there is such a k, otherwise -1.

   return recursive_loop (0, A.length, A, tgt);
}
```

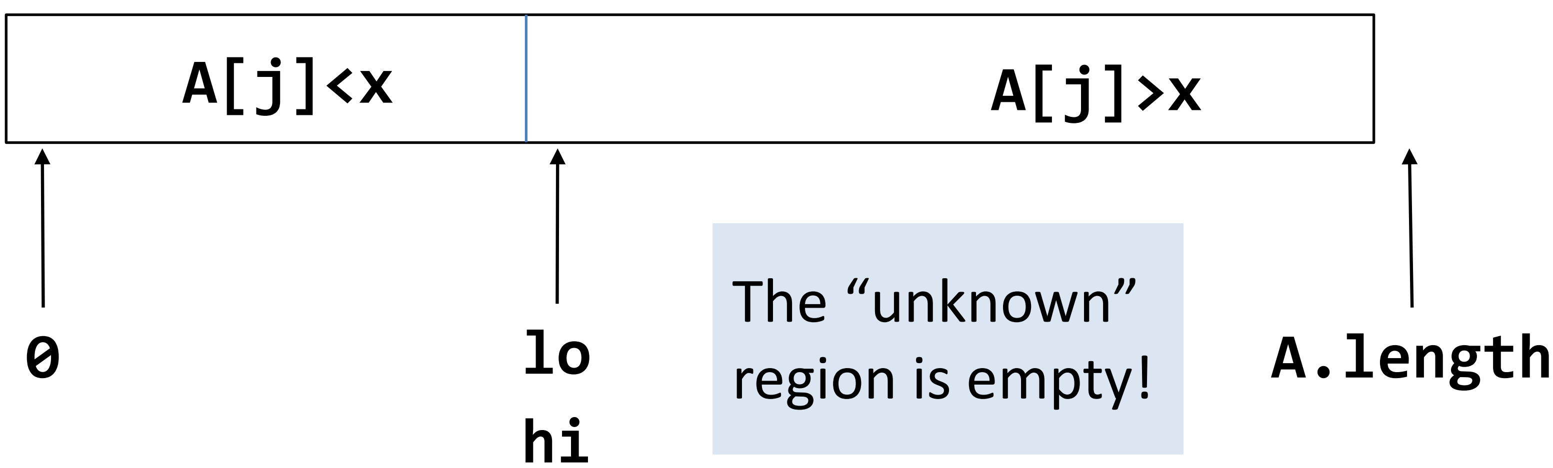
The invariant when **recursive_loop** is called



The unknown region is the entire array; the other regions are empty.

What are the easy cases for recursive_loop?

- if **lo=hi**, the search range **[lo,hi-1]** is empty, so the answer must be **-1**
- Otherwise we will have to work harder.



What if the search range is larger?

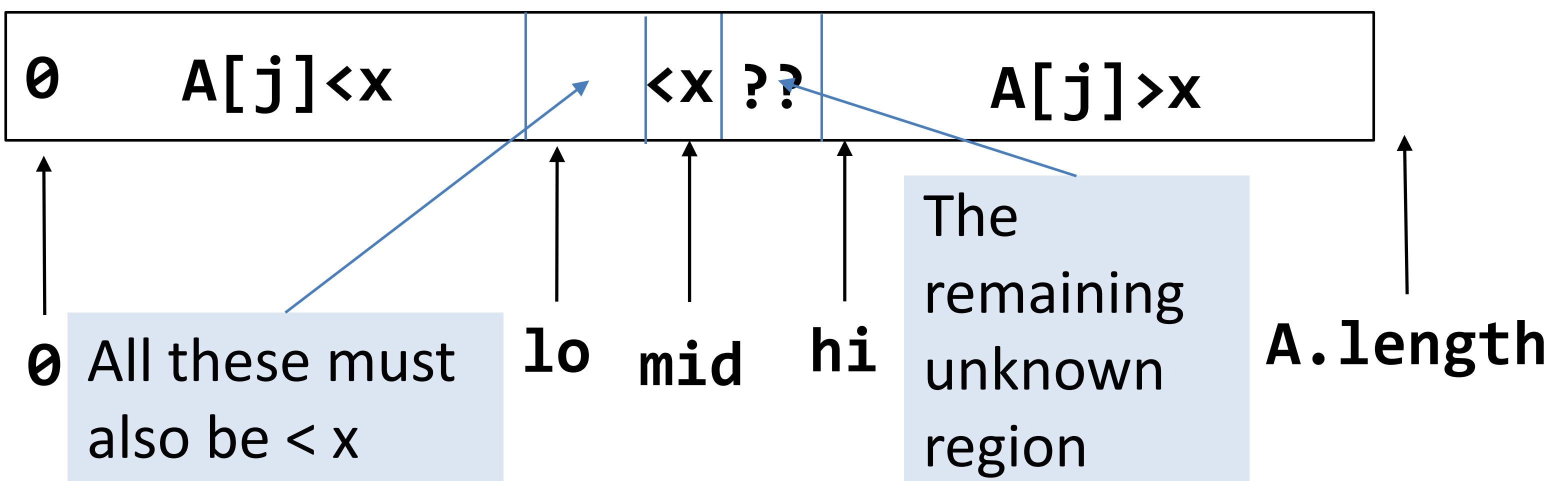
- Insight of binary search: divide it in half.
- At this point we know that lo < hi.
- Choose a midpoint **mid** in [**lo,hi-1**] and compare **A[mid]** to **tgt**.
  - **mid** doesn't have to be close to the center– any value in [**lo,hi-1**] will lead to a correct program
  - but choosing **mid** to be near the center means that the search space is divided in half every time, so you'll only need about $\log_2$(**hi**-**lo**) steps.

What are the cases?

- Case 1: A(mid) = tgt
  - then mid is our desired k.
  - Done!

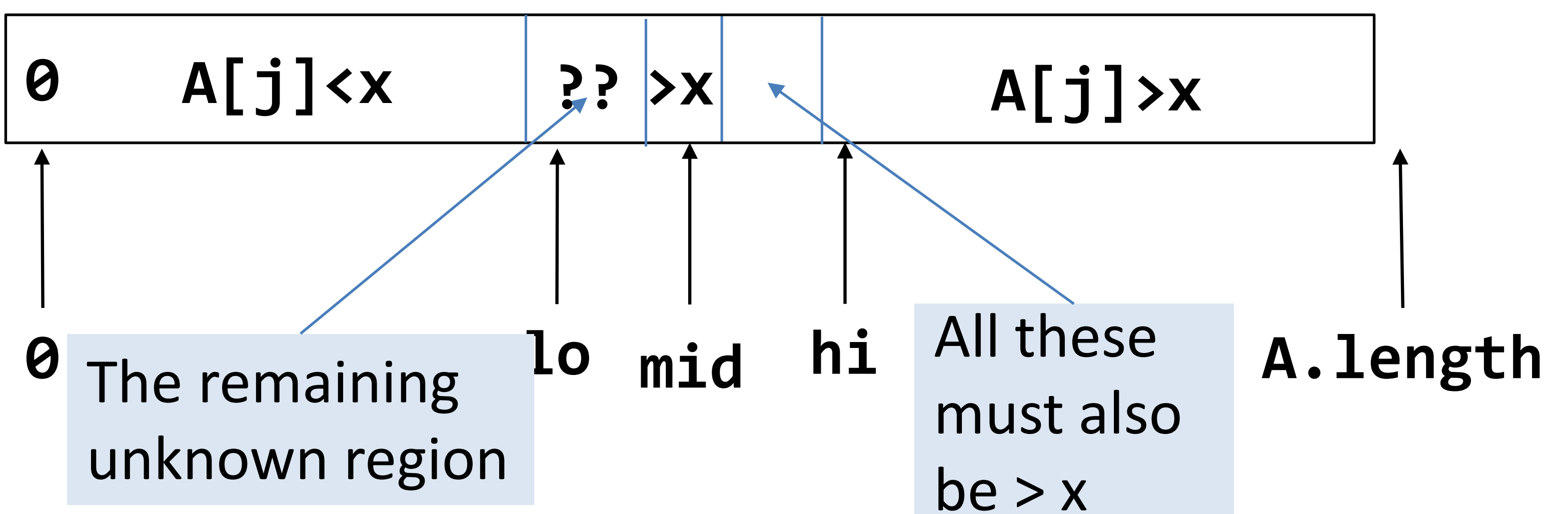
What are the cases?

- Case 2: A(mid) < tgt
  - so we can rule out **mid**, and all values less than mid (because if j < mid, then A[j] ≤ A[mid] < tgt).
  - So the answer k, if it exists, is in [mid+1, hi-1]
  - So set lo to mid+1, leave hi unchanged



What are the cases?

- Case 3: A[mid] > tgt
  - so we can rule out mid and all values greater than mid, because if mid < j, then tgt < A[mid] <= A[j].
  - So the answer k, if it exists, is in [lo,mid-1]
  - So leave lo unchanged, and set **hi** to **mid** .



As code:

```
static int recursive_loop (int lo, int hi, int[] A, int tgt) {
       if (lo == hi) { // the search area is empty
           return -1;
       }
       else { /* do nothing */}
       // choose an element in [lo,hi) .
       int mid = (lo + hi) / 2;
       if (A[mid] == tgt) { // we have found the target
           return mid;
       }
       else if (A[mid] < tgt) {
           // the target can't be to the left of mid, so search right half
           return recursive_loop (mid+1, hi, A, tgt);
       }
       else {
           // otherwise the target can't be to the right of mid, so
           // search left half.
           return recursive_loop (lo, mid, A, tgt);
       }
   }
```

Let’s watch this work

- Imagine A is an array with A[i] = i^2 for i in [0,40).
- Let’s find an element of A that contains 49.

Watch this work

```
(recursive_loop 0 40 A 49)
```

mid = 20

```
= (recursive_loop 0 20 A 49)
```

mid = 10

```
= (recursive_loop 0 10 A 49)
```

mid = 5

```
= (recursive_loop 6 10 A 49)
```

mid = 8

```
= (recursive_loop 6 8 A 49)
```

mid = 7

```
= 7
```

What's the halting measure?

- Proposed halting measure: hi-lo
  - (the size of the search region)
- Justification:
  - Since the invariant says that lo <= hi, we are guaranteed that hi-lo is a non-negative integer
  - Must check to see that hi-lo decreases on every recursive call.
  - At the first recursive call, lo increases (since lo <= mid < mid+1) and hi stays the same.
  - At the second recursive call, lo stays the same but hi decreases (mid will always be less than hi because integer quotient rounds down).

Doing it with a loop

- The calculation we showed above looks like the trace of a loop!

```
  (recursive_loop 0 40 A 49)
= (recursive_loop 0 20 A 49)
= (recursive_loop 0 10 A 49)
= (recursive_loop 6 10 A 49)
= (recursive_loop 6   8 A 49)
= 7
```

- So let’s write a loop that does the same thing.

We want the loop trace to look like this

```
looptop: lo=0 hi=40 tgt=49
```

mid = 20

```
looptop: lo=0 hi=20 tgt=49
```

mid = 10

```
looptop: lo=0 hi=10  tgt=49
```

mid = 5

```
looptop: lo=6 hi=10  tgt=49
```

mid = 8

```
looptop: lo=6 hi=8  tgt=49
```

mid = 7

```
loopexit: return 7
```

In this case, we can rewrite the recursion as a loop

Instead of saying

```
return recursive_loop (..., ..., A, tgt);
```

we say

```
   lo = ...
   hi = ...
```

and go to the top of the loop.

The Method Definition (1)

```
static int binsearch_iterative (int[] A, int tgt) {

       // GIVEN: An array A of integers and an integer target 'tgt'
       // WHERE: A is non-decreasing
       // RETURNS: a number k such that
       //         0 <= k < A.length
       //     and A[k] = tgt
       // if there is such a k, otherwise returns -1

       int lo = 0;
       int hi = A.length;

       // INVARIANT:
       //       0 <= lo <= hi <= A.length
       // AND   (forall j)(0  <= j < lo       ==> A[j] < tgt)
       // AND   (forall j)(hi <= j < A.length ==> A[j] > tgt)

       // Note that lo = 0 and hi = A.length makes the invariant
       // true, since in both cases there is no such j.

       // HALTING MEASURE: hi-lo
       // JUSTIFICATION: Same as above.
```

The Method Definition (2)

```
while (lo < hi) {  // the search area is non-empty
        // choose an element in [lo,hi) .
        int mid = (lo + hi) / 2;
        if (A[mid] == tgt) {
            // we have found the target
            return mid;
        }
        else if (A[mid] < tgt) {
            // the target can't be to the left of mid, so search right half.
            lo = mid+1;
        }
        // otherwise the target can't be to the right of mid, so search left half.
        else
            hi = mid;
    }

    // the search area is empty
    return -1;
}
```

Summary

- You should now be able to:
  - explain what binary search is and when it is appropriate
  - explain how the standard binary search works, and how it fits into the framework of general recursion, invariants, and halting functions
  - give the halting measure and explain the termination argument for binary search
  - write variations on a binary search function

Next Steps

- Study the file 08-2-binary-search.java in the Examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 8.3
- Go on to the next lesson