Binary Search, More on Reasoning About Loops, and Generics and Arrays

Here is a standard loop for searching for an item in an array:

/\* returns the index of x in array or -1 if x is not in array \*/

public int linearSearch(int[] array, int x) {

for (int i = 0; i < array.length; i++) {

if (x == array[i])

return i;

} // at this point i = array.length so we checked all entries of array

return -1;

}

Note that returning the index of the element gives us the most information out of the method. Returning x would be useless, and returning true/false would be only a little better.

What is the subgoal of the loop. It is what must be true after each iteration for the loop to continue: "x is not in the first i elements of array"

What if we want this to work on arrays of Object? The only change needed (besides the parameter types) is to test equality on the contents of the objects instead of their memory addresses.

/\* returns the index of x in array or -1 if x is not in array \*/

public int linearSearch(Object[] array, Object x) {

for (int i = 0; i < array.length; i++) {

if (x.equals(array[i]))

return i;

} // at this point i = array.length so we checked all entries of array

return -1;

}

Is this the best search method we can write?

In the worst case, how many array locations does it inspect?

All of them.

So in the worst case, the time the method takes will be proportional to the length of the array.

Can we write a faster loop?

No! What is the proof? No matter how we decide to run through the array, until we look at every location, we can not be sure if x is not in the array.

So, unless we know more information about the array, our linearSearch method is optimal.

Searching in a sorted array

Suppse the array is sorted in non-decreasing order. Now can we write a faster method?

Proposed Algorithm (Binary Search)

1. Examine the middle element.

2. If the middle element is smaller than x, repeat on the upper half of the array.

3. If the middle element is larger than x, repeat on the bottom half of the array.

This algorithm looks more complicated to code, but is it fundamentally faster?

We start with a list of n elements, and each time, we cut the number of elements we are considering in half.

n -> n/2 -> n/4 -> n/8 -> ... -> 1

So, the number of iterations (and the number of elements of the array we check) is k where n / (2^(k-1)) = 1.

k = log(base 2) n + 1

How much an improvement is this over linear search?

Array size # elements linear search checks # elements binary search checks

8 8 4

1000 1000 11

1,000,000 1,000,000 21

1,000,000,000 1,000,000,000 31

1,000,000,000,000,000 1,000,000,000,000,000 51

So, it is very worth it to write this method. Here is the first attempt we made in class.

We are using a while loop because we do not know, at first, how many times we will need to iterate.

Here is our first incorrect attempt:

/\*\* Return the index of x in a or -1 if x is not in array.

\* Precondition: a is sorted in non-decreasing order

\*/

public static int binarySearch(int[] array, int x) {

int front = 0; // stores the smallest index of array in the region still being considered

int end = a.length - 1; // stores the largest index of array in the region still being considered

while (front != end) {

int middle = (front + end) / 2;

if (x == a[middle])

return middle;

else if (x < a[middle])

end = middle;

else /\* only other case is x > a[middle] \*/

front = middle;

}

return -1;

}

Let us check the reasoning of our loop.

1) What is the goal:

At the end of the loop, x should not be in array. (Because if we reached the end of the loop, we did not return x.)

2) What is the subgoal for each loop iteration:

After each iteration, if x is in array, it is in array[front...end] (i.e. it is between front and end,inclusive).

Now let us verify the subgoal. Recall the three properties we need:

a) The subgoal is true immediately before the first iteration of the loop starts.

b) When the loop condition becomes false, the subgoal and the conditiond being false logically implies the goal.

c) If the subgoal is true before a loop iteration executes, it will still be true at the end of a loop iteration.

Property a is easy to verify. Before we start the loop, front = 0 and end = array.length - 1. Of course, if x is in array it will be in array[0..length-1].

Proberty b is problematic:

b) When the loop condition becomes false, the subgoal and the condition being false implies the goal.

The loop condition becomes false when front == end.

The subgoal is now:

"If x is in array, it is in array[end...end]"

That does NOT logically imply the loop goal! x could still be in array if x is at array[end].

Using the logic reasoning, we see that our loop is incorrect. Maybe thorough testing would have found this problem, but the error will only occur in specific situations (do you see why?).

Let us fix the method by changing the loop condition so that when it is false we do get the loop goal:

while (front <= end) {

Now, when the loop stops, end < front and so the loop subgoal becomes "if x is in array, it is in array[end..(end-1)]", and an array where the first index is larger than the last index is mathematically empty.

Are we guaranteed that the loop will terminate, or could it run forever?

To guarantee that it runs forever, we have to show that the gap between front and end decreases with each iteration. Thus, either front increase or end decreases.

Is it possible to have a situation where front and end to do not change? YES!! Consider front = 5 and end = 6 and the element we are looking for is at index 6. Note that in this situation, middle = 5, array[middle] is smaller

than the element, and so front = middle = 5. front never changed!

What is our fix? To note that we know that the element is not at the array[middle] and so we will not include it. Since we know that front <= middle <= end, setting front = middle + 1 and end = middle - 1 guarantees that

either front is increased or end is decreased on each iteration.

/\*\* Return the index of x in a or -1 if x is not in array.

\* Precondition: a is sorted in non-decreasing order

\*/

public static int binarySearch(int[] array, int x) {

int front = 0; // stores the smallest index of array in the region still being considered

int end = a.length - 1; // stores the largest index of array in the region still being considered

while (front <= end) {

int middle = (front + end) / 2;

if (x == a[middle])

return middle;

else if (x < a[middle])

end = middle - 1;

else /\* only other case is x > a[middle] \*/

front = middle + 1;

}

return -1;

}

The moral: using the formal logic reasoning helps us design correct loops before we code them into the program.

What about binary search on an array of Object?

First, recall that generics and arrays do not mix well. The -only- thing you can do with generics or parameterized types and arrays is to declare an array variable:

T[] array

or

Comparable<Integer>[] array

You can not create an array that is a generic type (new T[]) or contains a generic type (new Comparable<T>[]) nor can you create an array of a parameterized type, even if the

parameter type is specified (new Comparable<Integer>[]).

Java forbids any use of generic/parameterized types that can not be verified by the compiler using the "current type" of the object.

One thing we know is that the Object type must be Comparable so we have access to the compareTo method. Comparable takes a generic type so we will need to declare a generic.

Finally, remember the issue with HourlyEmployee in the OrderedLinkedList. HourlyEmployee implements Comparable<Employee>. We need to make sure that we do not restrict

the Comparable generic so that it allows Employee but prevents HourlyEmployee.

/\*\* Return the index of x in a or -1 if x is not in array.

\* Precondition: a is sorted in non-decreasing order

\*/

public static <T extends Comparable<? super T>> int binarySearch(T[] array, T x) {

int front = 0; // stores the smallest index of array in the region still being considered

int end = a.length - 1; // stores the largest index of array in the region still being considered

while (front <= end) {

int middle = (front + end) / 2;

if (x.compareTo(a[middle]) == 0)

return middle;

else if (x.compareTo(a[middle]) < 0)

end = middle - 1;

else /\* only other case is x > a[middle] \*/

front = middle + 1;

}

return -1;

}

Finally, what if we did binary search on linked list?

There is no reason to!!

Binary search is fast because we can access the middle element quickly. How do we get to the middle element on a linked list?

So, on a linked list, binary search will be no faster than the simpler linear search.