CS125 : Introduction to Computer Science

Lecture Notes #26
Searching

Lecture 26 : Searching

Today, we will start by thinking back to lecture 1, where we were discussing searching for a penny under a box, or searching a list of names for a particular name. We had said that, if all we knew was that we had some collection of values to inspect, and there was no information we could take advantage of to rule out any of the values, then we would need to inspect each and every one of them. This is the problem we are going to discuss now. Specifically, since the only way we know how to store a collection of information is in an array, we are going to search an array for a particular value. We'll start with an array of integers – and thus we are searching the array for a particular integer – but later on we'll talk about what we might have to worry about if the type were different.

So, let's imagine we are searching an array for the value 39. One way to choose the subproblem is to have our recursive call simply be to search everything but the first cell for the value `39`. That is, we can check the first cell, and if it is `39`, we are done, and if it is not `39`, then we'll recursively search the rest of the array. For example:

```
            ---------------------------------------
  A[i]      93  11  71  6   39  67  41  88  23  54
  -----------------------------------------------
    i       0   1   2   3   4   5   6   7   8   9
lo == 0, hi == 9
```

In the example above, we could check `A[lo]` (which is `A[0]` here). Since `A[lo]` is not `39`, we have not found our value yet. So, we could recursively search the rest of the array. That is, given the index range `lo...hi` we did not find our value in `A[lo]`, so we will recursively check the index range `lo+1...hi`. That is, next we check the index range `1...9` (we'll indicate this in the next diagram with the shorter horizontal line above our array values).

```
                -----------------------------------
  A[i]      93  11  71  6   39  67  41  88  23  54
  -----------------------------------------------
    i       0   1   2   3   4   5   6   7   8   9
lo == 1, hi == 9
```

So now again we check `A[lo]`, but this time since `lo` is `1`, we are checking `A[1]`. And `A[1]` is not our value 39, so we again recursively check the index range `lo+1...hi`. So our next recursive step is over the index range `2...9`.

```
                    -------------------------------
  A[i]      93  11  71  6   39  67  41  88  23  54
  -----------------------------------------------
    i       0   1   2   3   4   5   6   7   8   9
lo == 2, hi == 9
```

We check `A[lo]` (which this time is `A[2]`) and that is not equal to our value `39`, so we again recursively check the index range `lo+1...hi`. So our next recursive step is over the range `3...9`.

```
                          --------------------------
  A[i]    93  11  71  6   39  67  41  88  23  54
  ----------------------------------------------
    i      0   1   2   3   4   5   6   7   8   9
lo == 3, hi == 9
```

We check `A[lo]` (which this time is `A[3]`) and that is not equal to our value `39`, so we again recursively check the index range `lo+1...hi`. So our next recursive step is over the range `4...9`.

```
                              ----------------------
  A[i]    93  11  71  6   39  67  41  88  23  54
  ----------------------------------------------
    i      0   1   2   3   4   5   6   7   8   9
lo == 4, hi == 9
```

And now, when we check `A[lo]`, it *is* equal to our value `39`. We have found a cell containing `39`. There is no need for further work.

That is the basic idea behind our algorithm. We'll check the first cell of our range, and if that cell's value is not equal to `39`, then we'll recursively check the rest of the range (everything in the range except the first cell).

This raises a number of questions, and the answers to these questions will enable us to take our rough sketch of what we want to do, and produce a complete algorithm from it, by filling in the rest of the details.

- We have outlined how the recursive case should work, but what is the base case?
- What should be returned by our algorithm?
- How can we generalize our algorithm to search for integers other than `39`? More interestingly, what if we are not searching for integers at all? What if our array contains a different type, and we are searching for a value of *that* type?

We will consider these questions in order.

First, when dealing with the base case, we have to ask ourselves, for what size subarray do we know *for sure* whether the value exists or not? Well, if the subarray were of size 1, we would still need to check that cell, but if the subarray were of size 0, then we are certain our value is not there, because nothing is there. So, we could have `lo > hi` be the condition that would trigger the base case – the condition that indicates our subarray defined by `lo` and `hi` is of size `0`.

Next, we need to consider what to return. We are asking if `39` is in the array or not, and so the answer is either "yes" or "no". When you are limited to such answers, a `boolean` works very nicely – you return `true` if the value is in the array and `false` if it is not in the array.

However, there might be a better choice. If all we care about is knowing whether the value is in the array or not, a `boolean` is fine, but we might care about *retrieving* the value as well, or more specifically, retrieving the cell it is in. For example, perhaps we want to know if `39` is in the array, and, if it is, we want to later replace it with `40`. In that case, we want to know not just that `39` is in the array, but also, *at what index it is located.*

So instead of returning `true` or `false`, what we could do is return a pair of values: a `boolean` *and* an integer. This sort of return type is often useful, because there are often situations where we run a method on a collection of data but we only have a meaningful return value in certain cases – such as our searching example above, where we can only return "the index where our value is located" if the value actually exists in the array. In those cases, if we have no meaningful data to send back, we have to send back some meaningless data of some kind (because *something* needs to be returned), so returning the `boolean` along with the data means you can use the `boolean` to check if you should bother to read your data or not.

Since we can only return one value, any attempt to return a "pair" must make use of an object of some kind. For our search method, the following could be the necessary class:

```
public class ReturnPair
{
   public boolean statusFlag; // true if found, false if not
   public static int index;  // meaningful only if the boolean is true
}

// returned result is therefore one of the following:
//       (true, index_where_value_is_located)
//       (false, meaningless_integer_that_we_will_not_bother_reading)
```

and then our search method would have a return type of `ReturnPair` and the method would have to allocate a `new ReturnPair()` from inside the method, so that there was some object to return. This is a way to get around the only-one-return-value restriction and, in doing so, return a *collection* of information from the method – just make your "one value" that you returned, an address of a dynamically-allocated object, rather than a value of primitive type.

However, though conceptually, returning this pair of information is what we want to do, we actually can achieve this with only one integer and nothing else. We can take advantage of the fact that arrays have a limited index range – namely, `0` through `length-1`, where `length` is whatever the length of the array is. So, if you were to return a negative number, or some number greater than or equal to the length of the array, then that return value could be checked by the client of your algorithm. If the returned value is within the index range of the array, it would interpreted as a "true" – i.e. the value is in the array – and the returned integer is then assumed to be the index of the cell containing `39`. On the other hand, if the returned value is *NOT* in the index range of the array, then that is interpreted as a "false" – i.e. the value `39` is not in the array – and the client knows not to use the returned index to access the array. We get the benefits of returning a `boolean` (knowing whether the value is there or not) along with the benefits of knowing where the value is, if it is there. And we can do this while still returning just one integer, rather than having to create a `ReturnPair` object and return that. Since `-1` is *always* outside the index range of any array, it's a nice return value to use for this purpose, and that's what we'll use.

Finally, we are not likely to always need to search for `39`. More likely, the client will have some particular value in mind to search for. And if it's something the client specifies, then probably it should be a parameter. We often refer to such a value – i.e. the value to be searched for – as a *key* or *search key*. We could then compare the value in an array cell, to the parameter, rather than to `39` specifically.

Putting all of that together, leads to the algorithm expressed with the following Java code:

```
// LinearSearch
//    - parameters : arr - the array to search
//                 : key - the value to search for
//                 : lo - the low index of this subarray
//                 : hi - the high index of this subarray
//    - return value : array index or -1
//    - if key is in A, returns index where it
//        is located. Otherwise, returns -1.
public static int LinearSearch(int[] arr, int key, int lo, int hi)
{
   if (lo > hi)
      return -1;
   else if (arr[lo] == key)
      return lo;
   else
      return LinearSearch(arr, key, lo+1, hi);
}
```

Note that we could also write a *wrapper method* to call the above method, a wrapper method that would only need from the client the things the client actually cared about – namely, the array to be searched and the key to be searched for:

```
// LinearSearch
//    - parameters : arr - the array to search
//                 : key - the value to search for
//    - return value : array index or -1
//    - if key is in A, returns index where it
//        is located. Otherwise, returns -1.
public static int LinearSearch(int[] arr, int key)
{
   return LinearSearch(arr, key, 0, arr.length-1);
}
```

In the recursive method, we have two base cases. If we actually find our value, we return a real index, and if we reduce the subarray to size `0`, we return a `-1` to serve as a "not found" indicator. `-1` is not a real index, so if our search algorithm returns it, we know it is because the key was not found in the array. But if we have not found our key and yet there is more of an array to the right of our current low cell, then we should search that array – hence the point of the recursive call.

As algorithms get more complex – though admittedly, this algorithm is not terribly complex – we sometimes like to have only one `return` statement in our algorithm, so that there are not multiple exit points to keep track of. If we wanted to make such a modification above, it's a simple matter to just store the returned value in a variable and return it at the end:

```
public static int LinearSearch(int[] arr, int key, int lo, int hi)
{
   int returnVal;
   if (lo > hi)
      returnVal = -1;
   else if (arr[lo] == key)
      returnVal = lo;
   else
      returnVal = LinearSearch(arr, key, lo+1, hi);

   return returnVal;
}
```

The code would be similar, but not identical, if we were dealing with different types. For example, if we were searching for a `String`, we can't use `==` to compare `String` objects for equality, but must instead use the `equals` instance method:

```
public static int LinearSearch(String[] arr, String key, int lo, int hi)
{
   if (lo > hi)
      return -1;
   else if (arr[lo].equals(key))
      return lo;
   else
      return LinearSearch(arr, key, lo+1, hi);
}
```

We might also search on *part* of an object. For example, what if our `Clock` class from the object-oriented-programming lectures, had an instance method `public int getHour()` that returned the hour? In that case, perhaps we then wish to search a collection of `Clock` objects for one that is set to a particular hour. In that case, the `key` would be of type `int`, even though the array itself was of type `Clock` instead of `int`:

```
public static int LinearSearch(Clock[] arr, int key, int lo, int hi)
{
   if (lo > hi)
      return -1;
   else if (arr[lo].getHour() == key)
      return lo;
   else
      return LinearSearch(arr, key, lo+1, hi);
}
```

And if we were searching for a `double`, well, recall that computer arithmetic can introduce rounding errors. So, if we tried something like this:

```
public static int LinearSearch(double[] arr, double key, int lo, int hi)
{
   if (lo > hi)
      return -1;
   else if (arr[lo] == key)
      return lo;
   else
      return LinearSearch(arr, key, lo+1, hi);
}
```

we might not get correct results because we might have done the computations to fill the array one way, and gotten a value such as `24.839523176`, and yet, when we computed our `key` before passing it into this method, we might have tried doing the same computation, but put the steps in a different order, and thus had a different sort of rounding error and ended up with `24.839523172` instead. On MP1, we said it was okay to have those sort of differences in the final decimal place like that. But if we tried doing an exact comparison, as with the code above, well, those two numbers are, technically, *different*, and thus our comparison-for-equality (`==`) will evaluate to `false`. So, if we want to compare floating-point values for equality, we generally want to see if they are equal to within some acceptable error range, rather than seeing if they are precisely equal:

```
public static int LinearSearch(double[] arr, double key, int lo, int hi)
{
   if (lo > hi)
      return -1;
   else if ((arr[lo] > key - 0.000000005) && (arr[lo] < key + 0.000000005))
      return lo;
   else
      return LinearSearch(arr, key, lo+1, hi);
}
```

Now, we will claim we have a match, as long as `A[lo]` is within `0.000000005` of `key`.

All these examples follow the same basic algorithm – we just have different code to implement the “equality” comparison, depending on exactly what sort of comparison we are trying to do.

Now, this algorithm – however we implement it – doesn't take into account the possibility that the search key occurs multiple times in the array. If there *were* duplicate instances of the search key – for example, if we are searching for `39` and it occurs five times in the array – what can we do about that? Well, in many cases, handling duplicates requires just a small modification of your non-duplicate algorithm. However, there are also often a few ways that duplicates can be handled, and so before we can modify the non-duplicate algorithm to deal with duplicate-handling, we need to decide exactly how we want to handle duplicates in the first place. A few of the possibilities are listed below:

- return the index of the first occurence (i.e. occurence with lowest index) of the key in the array (return -1 if it never appears)
- return the index of the last occurence (i.e. occurence with highest index) of the key in the array (return -1 if it never appears)
- return the number of times the key appears in the array
- return a new array holding as its values all indices where the key appears in the original array

The first possibility is what our first `LinearSearch` algorithm does already – that algorithm was designed to start returning an index as soon as it found our search key, so if there were many other instances of the search key in the array, we would never see them since we wouldn't even inspect those cells. Likewise, the second possibility could be implemented simply by having our current `LinearSearch` algorithm search from `hi` to `lo` rather than from `lo` to `hi` – if we are trying to find the last (i.e. rightmost) instance of our search key, well, that's just the first one we will see if we start our search from the right instead.

The third possibility, though – counting the number of occurences – will require that we *always* look over *every* cell of the array. If there were some cell we did not inspect, well, our total might be wrong, depending on what our assumption was. If we assumed the value in that cell was indeed our search key, and it wasn't, our total would be too big by `1`. On the other hand, if we assumed it wasn't equal to our search key, and it was, then our total would be too small by `1`. So we need to look at every cell.

So instead of just inspecting the cell, as in the regular `LinearSearch`, we will want to inspect the cell, and also add one to a total if the value in the cell is equal to our search key. The base case is then in charge of initializing the total, since in the base case we have *no* instances of the key in this subarray – it follows from that, that the base case should return `0` (the total number of occurences of our key in the subarray of size `0`).

```
public static int CountingLinearSearch(int[] arr, int key, int lo, int hi)
{
   if (lo > hi)
      return 0;
   else if (arr[lo] == key)
      return 1 + CountingLinearSearch(arr, key, lo + 1, hi);
   else
      return CountingLinearSearch(arr, key, lo+1, hi);
}
```

Note that in both the case where `arr[lo]` is equal to our key, and the case where it isn't, we still need to make our recursive call. This is because we need to count how many times the `key` occurs in the other cells, regardless of whether the key is in arr[lo] or not. So, we could rearrange that code a bit so at least the same method call doesn't appear twice in the code:

```
public static int CountingLinearSearch(int[] arr, int key, int lo, int hi)
{
   if (lo > hi)
      return 0;
   else
   {
      int total = CountingLinearSearch(arr, key, lo + 1, hi);
      if (arr[lo] == key)
         return 1 + total;
      else
         return total;
   }
}
```

Either form is acceptable. And, as with the earlier, non-duplicate version, we could also have arranged this so that we have only one return statement, if we wanted:

```
public static int CountingLinearSearch(int[] arr, int key, int lo, int hi)
{
   int returnValue = 0;
   if (lo <= hi)
   {
      returnValue = CountingLinearSearch(arr, key, lo + 1, hi);
      if (arr[lo] == key)
         returnValue = returnValue + 1;
   }
   return returnValue;
}
```

which looks much less like the original LinearSearch but is still correct.

If the way we want to handle duplicates is to return an array containing the indices where the duplicates are located, then we've got a few issues to worry about there as well. The big conceptual issue is what to return if we have an unsuccessful search, and here we have two options. Our return type must be "integer array reference" (i.e. int[]), if we are returning an array of integers – all the array indices will be integers, so if we want to return an array of indices, we are returning an array of integers. And if our return type is an integer array reference, we can always return null if there is no array to return. The upside to that is, we don't need to allocate an array object if there is an unsuccessful search. The downside is, we have no object returned so we need a conditional to treat the unsuccessful search as a special case:

```
// Example client code calling a hypothetical version of linear
//  search that returns an array of all indices where search key
//  is located. Our code below would be calling some wrappper
//  method around the recursive version, not the recursive version itself.
int[] result = LinearSearchForDuplicates(arr, key);
if (result == null)
   System.out.println("There were 0 occurences of the key.);
else
   System.out.println("There were " + result.length + " occurences of the key.");
```

Sometimes, having to deal with the lack of an array object as a separate case can become troublesome. So, we also have the option of returning an array of size `0`, instead of returning `null`:

```
// in our hypothetical LinearSearchForDuplicates method, replace
//              return null;
// with
//              return new int[0];
```

Now, the client code doesn't need to make a special check to see if the returned reference is `null` – we know whatever is returned, will always be a reference to an array, even in the case of an unsuccessful search. And hence, we can do this:

```
// Example client code calling a hypothetical version of linear
//  search that returns an array of all indices where search key
//  is located. Our code below would be calling some wrappper
//  method around the recursive version, not the recursive version itself.
int[] result = LinearSearchForDuplicates(arr, key);
System.out.println("There were " + result.length + " occurences of the key.");
```

and not bother to continually check to see if our returned array even exists.

The other issue that would need dealing with would be the increasing of the size of the array to be returned, as we found more and more occurences of our search key. You could either put a loop inside your recursive algorithm, to copy values to a new array, or else you could write a helper method (whether recursive or not) to accomplish the same task, and then call that method from your algorithm. Either way, if we intend to return an array of exactly the right size, we'll need to be changing the array size as we find additional occurences of the search key. We will leave that as an exercise for you to try.

We can also consider another search problem: finding the minimum value in an array. More specifically, we want to return the index where the minimum value is located.

In this case, it is difficult to even define the problem if the subarray we are searching is of size `0`. You could *search* an empty collection (you would not find your value), and you could *print* an empty collection (you would print nothing), but how can you "return the minimum value of an empty collection"? There's nothing to return!

So, you have two options:

- You could simply say that finding the minimum of a collection only makes sense if you actually have values in the collection – and as a result, design your method to assume that the parameter subarray has size at least `1`. Subarrays of size `0` won't be allowed.

- Alternatively, since we are returning the *index* of the minimum, rather than the minimum itself, we could have a special case that returns `-1` in cases where the subarray is of size `0`.

We will do things the first way – we'll simply define "find the minimum" to be an operation that makes no sense on subarrays of size zero.

Now, finding the minimum value in an array can be solved recursively. If the array is of size 1, then certainly that one cell is the minimum, so return its index. Otherwise, find the minimum of the *rest of the cells* (i.e. all the cells but the first one) recursively. Then, compare that result to the first cell's value. If the first cell's value is lower than the minimum of the remaining cells, then by definition it is lower than *all* the remaining cells, and thus is the overal minimum. On the other hand, if the minimum of the remaining cells is also less than the first cell, then the minimum of the remaining cells is the minimum of all the cells.

```
public static int findMinimum(int[] arr, int lo, int hi)
{
   if (lo == hi)
      return lo;
   else
   {
      int locOfMinOfRest = findMinimum(arr, lo + 1, hi);
      if (arr[lo] <= arr[locOfMinOfRest])
         return lo;
      else
         return locOfMinOfRest;
   }
}
```