CS125 : Introduction to Computer Science

Lecture Notes #25
Subarrays : Recursion on Data Collecetions

Lecture 25 : Subarrays – Recursion on Data Collections

When dealing with recursion on arrays, the subsequent recursive call usually needs to run on a portion of the original array. For example, if our goal is to print the entire array, the goal of the recursive call would be to print *part* of the array. If our goal were to find the minimum value in the array, the goal of the recursive call would be to find the minimum value of *part* of the array. As with our earlier recursion examples, the recursive call in these examples is solving a subproblem – performing the same work, on a smaller amount of data. In the case of an array, this often means that we perform the same work on a piece of the original array, rather than the entire original array.

How can we obtain this "piece of the original array"? One way would be to copy parts of the original array to a new array. For example, if we had an array indexed from `0` through `9`, and the recursive call is designed to deal with everything but the first cell, we could copy cells `1` through `9` of the original array to a new array. Since we are copying nine cells there, the new array must be of size `9` – meaning it will be indexed from `0` through `8`. So we are copying cells `1` through `9` of the original array, to cells `0` through `8` of the new array. And then in the next step, we could copy cells `1` through `8` of that new array, to cells `0` through `7` of an even newer array. And then, in the next step, we could copy cells `1` through `7` of that array into cells `0` through `6` of a still-newer array. And so on.

If we do this, each step has one fewer cell than the step before it. However, this results in a lot of time spent copying and a lot of memory needed for the new array we create at each step. So, rather than recopy all the relevant values to a new array, we can simply use a pair of indices `lo` and `hi` to indicate what portion of the original array we are dealing with, and by changing `lo` and `hi`, we can change the portion of the original array that a particular method call cares about.

Going back to the previous example, if our original array were indexed from `0` through `9`, then we could have `lo` hold the value `0`, and `hi` hold the value `9`. Our code could then be designed to deal with the cells with indices `lo` through `hi`, inclusive. Since `lo` holds the value `0` and `hi` holds the value `9`, then code would then be dealing with the cells indexed from `0` through `9`, inclusive, exactly the way we want. When it comes time to deal with a smaller part of that array, we could increase `lo` by one. Now, `lo` holds the value `1` and `hi` still holds the value `9`, and thus our code deals with the cells with indices `1` through `9` inclusive. Next, we could increase `lo` again, to `2`, and now our code will deal with the cells with indices `2` through `9`, inclusive. If we increase `lo` again, our code will be deal with the cells with indices `3` through `9` inclusive. Each step deals with one fewer cell than the step before, just as in our earlier example. But this time, to go from one step to the next, we only need to increase `lo` by `1`; we don't need to create a new array or copy values from one array to the next.

Portions of the original array – especially when `lo` and `hi` are used to indicate the low and high indices of that portion of the original array – are known as *subarrays*, and by changing the values of `lo` and `hi` we can change what *subarray* we are dealing with.

In order to be able to change `lo` and `hi` for each call, we will need to have them as parameters to which we can pass arguments:

```
ex:  void DoSomething(int[] arr, int lo, int hi)
     {
        if (recursive case condition is true)
        {
           // ...other code here, and then
           // somewhere in here, we make a recursive call
           DoSomething(arr, lo + 3, hi / 2);
        }
     }
```

If the original call to the pseudocode above had as arguments the integers `0` and `9`, then the next call has arguments `3` (`lo + 3` is `3` if `lo` is `0`) and `4` (`hi` divided by `2` is `4` if `hi` is `9`). That next call then operates on the part of the array from cell `3` through cell `4`; it is quicker than copying the array and easier to deal with as well.

As an example, let us consider printing an array. If we want to print the array from cell `lo` through `hi`, some of the possible subproblems are:

- printing the array from `lo+1` through `hi`; (then you only need to print `arr[lo]` yourself; the rest is handled by the recursive call)
- printing the array from `lo` through `hi-1`; (then you only need to print `arr[hi]` yourself; the rest is handled by the recursive call)
- printing the array from `lo` through the middle of the subarray, i.e. from `lo` through `(lo + hi)/2`; (then you need to find some way to print the cells with indices `((lo + hi)/2) + 1` through `hi` yourself, since the recursive call didn't print any of those cells)

Let's build an algorithm out around the first subproblem, printing everything from `lo+1` through `hi`. (Technically, each of the three subproblems above *could* serve as the basis for an algorithm; the one we get from this subproblem will be the most straightforward, though.)

If we want to print the cells in order by index, starting with the lowest-indexed cell, then we need to print `arr[lo]` on our own, *before* recursively printing everything with indices `lo + 1` through `hi`. Thus our algorithm (minus the base case, which we'll add in a minute), looks like this:

```
To print the array cells indexed lo through hi of an array "arr":
   1) Print out arr[lo]
   2) Recursively, print the array cells indexed lo + 1 through hi of "arr"
```

If initially `lo` was `0` and `hi` was `9`, then the first call is trying to print the values at indices `0` through `9`, and does so by printing out the value at `arr[0]`, and then makes a recursive call to print out the values at indices `1` through `9`. And that call works by printing out `arr[1]` (the new value of `lo` in this call is `1`) and then making a recursive call to print out the values at indices `2` through `9`. And that call works by printing out `arr[2]` (the new value of `lo` in this call is `2`) and then making a recursive call to print out the values at indices `3` through `9`. And so on.

Once `lo` is greater than `hi`, there are no cells in that range (i.e. an empty subarray) and nothing left to print. So that can be our base case:

```
To print the array cells indexed lo through hi of an array "arr":
   if (lo <= hi)  // we have at least one cell in our subarray
   {
      1) Print out arr[lo]
      2) Recursively, print the array cells indexed lo + 1 through hi of "arr"
   }
   else  // we have no cells in our subarray
      ;   // there is nothing to do
```

and that gives us the following code:

```
// prints all values of arr whose indices are in range lo...hi inclusive,
//  in order from lowest to highest index
public static void print(int[] arr, int lo, int hi)
{
   if (lo <= hi)
   {
      System.out.println(arr[lo]);
      print(arr, lo+1, hi);
   }
   // else you do nothing
}
```

That's all!

If you had chosen to use `lo...hi - 1` as the subarray instead, then you need to make the recursive call *first* (thus printing all values with indices `lo` through `hi - 1`, inclusive) and then print the value at index `hi` after the recursive call has completed:

```
// prints all values of arr whose indices are in range lo...hi inclusive,
//  in order from lowest to highest index
public static void print(int[] arr, int lo, int hi)
{
   if (lo <= hi)
   {
      print(arr, lo, hi - 1);
      System.out.println(arr[hi]);
   }
}
```

But either one of those versions of the code will print the values with indices `lo` through `hi` inclusive, from the lowest-indexed value to the highest-indexed value.

Now, perhaps the clients of this method don't want to bother having to pass in indices as arguments. Perhaps a client would simply like to say, "print the array" and have it be assumed that the request means, "print the entire array, from index `0` through the highest index of the array". That is, perhaps the client would like to make the following method call:

```
print(arr);
```

rather than the method call:

```
print(arr, 0, arr.length - 1);
```

Unfortunately, right now, the second method call would be required. Since our method needs the `lo` and `hi` parameters in order for the recursion to work, it requires that the client send in some initial values as arguments to those parameters. The flexibility this allows, *is* nice – it means the client could also do this if they wanted to only print part of the array:

```
// one example of an alternate method call that could be made
print(arr, 3, arr.length - 5);
```

but even though the client *could* choose to pass in any pair of bounding indices as the second and third arguments, `0` and `arr.length - 1` are likely to be the most common arguments. That is, the most common situation is that the client wants to print the *entire* array; situations where the client only wants to print *part* of the array, would probably be less common.

So, one possibility is to write another method:

```
public static void print(int[] arr)
{
   // some code goes here
}
```

to which the client could simply send the array reference as an argument, and not the indices as well. The "glue" between this method and the recursive one we wrote earlier, is that the above method can call the recursive one, with the most common arguments!

```
public static void print(int[] arr)
{
   print(arr, 0, arr.length - 1);
}
```

Such a method is known as a *wrapper method*. A wrapper method does exactly as the name implies – it "wraps" around an already existing method that does something similar – by making a call to that already existing method – and thus provides a different interface to that method. In the case of our `print` methods, clients can either call the recursive `print(...)` directly, with three arguments, or they can call non-recursive `print(...)` with one argument, and that method can call the recursive `print(...)` with three argumebts. Clients have a choice of how they want to activate the printing of their array, but in both cases, it's the three-parameter `print(...)` that does most of the work.

Wrapper methods are usually relatively simple; the call to the longer, more interesting method is generally the most important statement of the wrapper method, and only a little bit of work gets done before or after that statement. In the wrapper method above, the work that gets done is to

select initial arguments to the recursive method's `lo` and `hi` parameters, and to calculate what the second of those two index arguments should be (i.e. to calculate `arr.length - 1`). That "set-up work" – the choosing of the initial values of `lo` and `hi` – only needs to happen once, so it should not be in the call to the recursive method itself. Instead, we do that work in a wrapper method, before the call to the recursive method, and then once the call to the recursive method is made and returned from, this wrapper method has nothing else to do.

In other cases, the wrapper method might do a little bit more work before the call to the longer, more interesting method, and it might even do some work after that method call returns as well. But generally, there is only a little bit of work, at most, to do before or after that method call. Wrapper methods don't tend to be very complex; they basically exist to help set up a situation where the more interesting method can get called.

(Note that naming both methods the same – such as `print` in our examples above – is fine. This is an example of *method overloading*, which we discussed back when we introduced constructors. As long as the methods have different parameter lists, they are distinguishable by the compiler even if they have the same name.)

As another example, let's consider the detection of palindromes. A palindrome is a word or number that reads the same forward and backwards, such as RACECAR or 18481. We can consider a sequence to potentially be a palindrome as well. For example, these two sequences:

```
18, 49, 21, 49, 18
```

```
18, 49, 21, 21, 49, 18
```

are both sequences that read the same forward and backward, and thus could be considered palindromes. More generally, if `p` is some palindrome sequence, then `x, p, x`, where `x` is some single value, will be a palindrome sequence as well. We see that in the first sequence above, for example – `49, 21, 49` is a palindrome, and if we put `18` at the beginning and end of that sequence, we still have a palindrome. This suggests that if you are given a sequence such as `18, 49, 21, 49, 18`, one useful question might be to ask if the same value is listed first and last. If the first and last values are the same, then perhaps a good second question to ask is, if everything but the first and last values constitutes a palindrome – in this case, if `49, 21, 49` constitutes a palindrome. If the first and last values are the same, and if everything but the first and last values constitutes a palindrome, then our sequence matches the `x, p, x` pattern and would itself be a palindrome.

However, the following three sequences are NOT palindromes:

```
18, 49, 21, 49, 19
```

```
18, 49, 21, 22, 49, 18
```

```
18, 49, 21, 22, 49, 19
```

In the first sequence, the first and last values are not the same, so you have a different first value when you read the sequence left-to-right versus right-to-left. Therefore the sequence is not a palindrome. In the second sequence, the first and last values are the same, and in fact, the second and second-to-last values are the same as well, but the middle sequence, `21, 22`, is not a palindrome. In the third sequence, we have both problems – the first and last values are not the same, and the sequence between them (`49, 21, 22, 49`) is not a palindrome either. So, just as we were discussing above, it would seem helpful to ask if the first and last values of a sequence are the same, and to ask if the sequence between them is a palindrome. If the answer to either of those questions is "no", then we do not have a palindrome.

Now, since our overall intention is to ask if the entire *sequence* is a palindrome, and as part of that task, we want to ask if some subsequence is a palindrome, well, there's our recursion! So far, we have this:

```
// we aren't quite done yet...
public static boolean isPalindrome(int[] arr, int lo, int hi)
{
   if (arr[lo] != arr[hi])
      return false;  // if outer values are not same, not palindrome
   else
      return isPalindrome(arr, lo+1, hi-1); // check inner section
}
```

If the first and last values are not equal, we *know* we do not have a palindrome. Otherwise, our answer relies on whether the middle section is a palindrome or not.

Of course, we eventually reach the point where the middle section is so small that it *must* be a palindrome and we don't really need to do any processing to figure that out. For example, if the sequence is of size `0` or `1`, then of course the sequence must be a palindrome If you have just one value in a sequence, then it's the same sequence forwards and backwards. And, if you have an empty sequence, then it is empty whether you read it forwards or backwards. This gives us the following code:

```
public static boolean isPalindrome(int[] arr, int lo, int hi)
{
   if (lo >= hi)
      return true;  // size 0 and 1 collections read the same both directions
   else if (arr[lo] != arr[hi])
      return false;  // if outer values are not same, not palindrome
   else
      return isPalindrome(arr, lo+1, hi-1); // check inner section
}
```

We could then write a wrapper method for this as follows:

```
public static boolean isPalindrome(int[] arr)
{
   return isPalindrome(arr, 0, arr.length - 1);
}
```

In this case, even if the client wants to call the wrapper method, they still want to have a `true` or `false` returned to them, so the wrapper method needs to return the value that the call to the recursive method will generate. If instead, the client expected the wrapper method to print a message, then the wrapper method could have been written like this instead:

```
public static boolean isPalindrome(int[] arr)
{
   boolean result = isPalindrome(arr, 0, arr.length - 1);
   if (result == true)   // could also have just said "if (result)"
      System.out.println("array holds a palindrome sequence");
   else
      System.out.println("array does not hold a palindrome sequence");
}
```

So, a wrapper method doesn't even need to do exactly the same sort of thing as the method it calls. In our `print(...)` example, both the wrapper method, and the method it called, printed out values that were stored in the array. In the above case, however, the recursive method returns a value based on whether the subarray is a palindrome, and the wrapper method instead prints a message based on whether the subarray is a palindrome.

As a final example, let's consider adding the diagonal elements of a two-dimensional array that has the same number of rows and columns. In this case, we want four indices – two to indicate the low and high row indices, and two to indicate the low and high column indices.

```
public static int addDiagonal(int[][] arr, int loR, int hiR, int loC, int hiC)
{
   if ((loR > hiR) && (loC > hiC))
      return 0;
   else
      return arr[loR][loC] + addDiagonal(arr, loR+1, hiR, loC+1, hiC);
}
```

You can get away with fewer parameters, if you like. Consider that an exercise for the reader.

In the above case, we might use a wrapper method to see if the matrix we are trying to add the diagonal of is actually a square matrix. That error checking would not need to be done with each recursive call – it would only need to be done once, before the recursion began. So, since it is just some set-up work, we can put it in a wrapper method and then call the recursive method from there.

For example, if we knew our matrix would contain only positive integers – and therefore we knew that the sum could never be `-1` along the diagonal – we might write a method like the following:

```
   public static int addDiagonal(int[] arr)
   {
      if (arr.length != arr[0].length)
         return -1;
      else
         return addDiagonal(arr, 0, arr.length-1, 0, arr[0].length-1);
   }
```

Or in other words, "if this is not a square matrix, return -1; otherwise return the actual sum of the diagonal". If we return -1, since we know that cannot be a legal sum, we know it must be an error signal.

If the array could hold any integers, that is a bad approach, since if we get -1 back as a return value, we'd have no way of knowing whether it was meant to signify an error, or if instead -1 was the actual sum of a diagonal on a square array. So in that case, we might write a class as follows:

```
public class Result {
   public boolean isSquare;
   public int sumIfExists;  // shouldn't read this if isSquare == false
}
```

and then our wrapper method could do this:

```
public static Result addDiagonal(int[] arr)
{
   Result pair = new Result();
   if (arr.length != arr[0].length) {
      pair.isSquare = false;
      pair.sumIfExists = -1;  // who cares? Client shouldn't read this anyway
   }
   else
   {
      pair.isSquare = true;
      pair.sumIfExists = addDiagonal(arr, 0, arr.length-1, 0, arr[0].length-1);
   }
   return pair;
}
```

In the error case, we would return a reference to a `Result` object where the `isSquare` variable is `false`. The client, upon reading that `false` value in that variable, would know it was a signal that the sum doesn't really exist, and would not bother reading the `sum` variable. However, if there was a `true` in the `isSquare` variable, then the client would know that it was indeed a square matrix, and would read the `sum` variable to get the diagonal sum.

The point is that the setup code (the error check) and the cleanup code (taking the return value of the recursive call – if we did make the recursive call – and writing it into the `Result` object) each only need to be done once, so we will put them in a wrapper method, not the actual recursive code where those lines would be repeated in every call.