Arrays and Loops - two examples from lab

1. The first example is removing an element from an array.

/\*\*

\* Remove k from list[0..n-1].

\* Precondition: list[0..n-1] is sorted, and k is an element of list[0..n-1].

\* Postcondition: list[0..n-2] is sorted.

\* @param list the sorted array.

\* @param n the number of items in list.

\* @param k the number to remove from list.

\*/

public static void remove(int[] list, int n, int k) {

Our first attempt:

Have a boolean variable indicate whether we have located k.

Create a single loop from 0 to n.

If we find k, set the indicator variable to true.

If the indicator variable is true and this is not the last index, shift the next element down to this element.

This loop idea will work fine. However, it is more complicated than needed and will run slightly slower than needed.

The problem is that the loop does two tasks. We need if statements to check for which task we are doing, and we need a special

if statement to avoid a possible error at the end of the loop.

We can make the code much simpler by breaking the loop into two. Each loop does one task. The first loop finds k.

The second loop shifts everything after k down one index.

/\*\*

\* Remove k from list[0..n-1].

\* Precondition: list[0..n-1] is sorted, and k is an element of list[0..n-1].

\* Postcondition: list[0..n-2] is sorted.

\* @param list the sorted array.

\* @param n the number of items in list.

\* @param k the number to remove from list.

\*/

public static void remove(int[] list, int n, int k) {

int i;

// step 1: find index of k

// iteration subgoal: At end of each iteration, we know that k is not in the first i elements

for (i = 0; i < n && k != list[i]; i = i + 1)

;

// <- at this point, k = list[i] (or i == n if k is not in the list)

// step 2: remove k and slide the rest of the entries down

// iteration subgoal: list[0,..,i-1] is sorted, list[i] is no longer used

while (i < n - 1) {

list[i] = list[i + 1];

i = i + 1;

}

}

2. The second example is adding an element to the array.

/\*\*

\* Add k to list[0..n-1].

\* Precondition: list[0..n-1] is sorted and list[n] is unused.

\* Postcondition: list[0..n] is sorted, and list contains k.

\* @param list the sorted array.

\* @param n the number of items in list.

\* @param k the number to add to list.

\*/

public static void insert(int[] list, int n, int k) {

An obvious attempt (though no one in class claimed to have tried this)

Have a loop that runs through until we find an index with list[index] <= k <= list[index + 1]

Now shift everything down by using a saved value that starts at k. Repeat: save the value at index + 1, place the previously saved value into index + 1, increment the index.

This sliding is complex and requires extra memory. A better solution is to write a loop that runs from the back to the front, sliding as it

goes. (Note that we do not need to find the location for k first. Instead, we just slide and stop when we get to the spot for k.)

It is a little tricky getting the stopping condition right. Logic helps!

// Start from back, slide each element to the right, and stop when we get to the spot where k goes

for (i = n - 1; i >= 0 && k < list[i]; i = i - 1) {

list[i + 1] = list[i];

}

// <- at this point, we have k >= list[i] OR i == -1

list[i + 1] = k;

}

We can also think about a subgoal for the loop iteration, and that helps see that our code is correct.

The subgoal needs to say something about i and k. Basically, we need that everything to the right of i is larger than k.

Here is a more formal subgoal:

iteration subgoal: list[i+2..n] is sorted, k < list[i+2], and list[0..i] is sorted with list[i] <= list[i+2], and list[i+1] is unused

(Note that the subgoal starts at i+2. Why? Because we moved the element at i to i+1, and so index i is "empty", and then we decreased i as the last step of the loop iteration.

That means that at the end of each iteration, index i+1 will be "empty".)

What happens when we stop? The loop subgoal is still true, but the loop condition is now false, and so we have:

list[0..i] is sorted, list[i] <= k < list[i+2], list[i+2..n] is sorted, and list[i+1] is unused. (Note this even works if i == -1.)

This logic shows that there is only one thing to do to complete the task: put k into list[i+1].

Final note: In class we wrote the loop condition as k <= list[i]. After class, a student pointed out that it would be better to test if k < list[i]. That way, if we have the list

contain all entries with value k, the loop will immediately stop and place k at the end of the array instead of running to the front to place k at the start.

Morals of the Lecture:

1) Break a problem up into smaller tasks. Use a separate loop for each task.

2) Place all stopping criteria in the condition of the loop.

3) Sometimes, working in "reverse" leads to simpler code. Always think about the problem in both directions.

In each case, we ended up with simpler loops that were easier to code, and our programs will run faster because there are fewer steps inside each loop.