OVERVIEW OF PSEUDO-CODE AS IT IS USED IN THIS CLASS

Abstract. This is condensed version of about three weeks' worth of lecture notes. In this document, we review the pseudo-code that we have used consistently in this class, and we use that pseudo-code to develop the logic for a common *interchage sort* that arranges integers in ascending order by *selection*.

1. Why Pseudo-Code?

Pseudo-code provides a way for us to describe algorithms in a language-independent manner. Once we get accustomed to reading and writing pseudo-code, we can unambiguously express simple and complex algorithms—again, in a language-independent way.

As a case in point, pseudo-code is especially useful in expressing the effect of *assignment statements*, which appear in JavaScript as statements that appear as:

```
const PI = 3.14159;
var myTotal;
var myTotal = 100;
myTotal = myTotal + 1;
```

Algorithm 1 A translation of the first collection of assignment statements from above.

$PI \leftarrow 3.14159$ ▷ Note that we have no distinction for constants.
$myTotal$
$myTotal \leftarrow 100$
$myTotal \leftarrow myTotal + 1$

Many "shorthands" are also more easily visualized in pseudo-code, such as those above which are pseudo-coded below:

```
myTotal++;
myTotal *= 10;
myName = "Tom";
myName += "Reinhardt";
```

Algorithm 2 A translation of the shorthands from above

$myTotal \leftarrow myTotal + 1$
$myTotal \leftarrow myTotal * 10$
$myName \leftarrow$ "`Tom`"
$nyName \leftarrow myName +$ "`Reinhardt`"

2. Using Pseudocode to describe algorithms

Recall that an algorithm is a finite set of instructions that solves a particular kind of problem, informally one that has a clear solution. We talked about "four" basic "moves" that underlie any algorithm. In the introductory comments, above, we used pseudocode to clarify the meaning of "assignment statements" as they appear in JavaScript. But, pseudocode is more commonly employed to describe complete algorithms which may (or may not) use assignment statements in their description.

2.1. **The basic four...** Considering the basic four, in no particular order, we begin with two related behaviors: sequence and concurrency. These are related in that one is the "specialization" of the other, meaning that I can express one (which?) in terms of the other, but not vice versa. It turns out, interestingly, that these two moves are more difficult to visualize through pseudocode than one might think.

2.1.1. *Sequence.* This is the easiest in that we merely read down the page. The first line of code is the first "instruction" that is to be executed. The *implicit* understanding here is that the first instruction completes *before* the next instruction begins. And, in most circumstances, this is true—although it is sometimes not the case in so-called parallel or concurrent languages, happily this is not our concern at the present.

Sequence is also implied by *composition.* Taking an example from this week's in-class exercise, I can compose the following sequential statements:

```
var input1 = window.prompt( ‘‘Enter a positive integer:’’ );
input1 = parseInt( input1 );
```

into

```
var input1 = parseInt( window.prompt( ‘‘Enter a positive integer: ‘‘));
```

because (1) both `prompt` and `parseInt` are *functions*, and the value returned by `prompt` can be understood (or used as an input to) the function `parseInt`. Assuming that the String passed from `prompt` to `parseInt` is a valid name of an integer, the effect is the same as executing these statements in the explicit order as in the first example.

We will use composition in almost every programming setting in this course. What's critical for you to understand at this point is that composition, generally, is not commutative—you cannot, for example, change the order in which we composed the functions above.

2.1.2. *Concurrency.* We have no direct examples of concurrent behaviors in this particular code example. In fact, most pseudo-code programming languages do not explicitly represent concurrency within the language, instead, concurrency results from the bigger context in which a particular algorithm is executed. In the case of web programming, concurrency is a result of how users interact with our algorithms through various event handlers, such as "onclick" or "onload," etc.

This has several implications: for one, it suggets that the pseudo-code that we use assumes that the flow of control is sequential, which is common in contemporary computing. But, many computer scientists (as well as computer manufacturers, vendors, and others) do recognize that the future of computing relies upon the successful design and evolution of existing and new algorithms that can take advantage of concurrent computation. The idea behind this thinking is simple: we've exhausted the possibilities in current silicon, and any hope for additional improvements in

computational speed will likely come from the ability of programs to execute many instructions in parallel or, at least, concurrently (which is different that "parallel,"' . . . , how?).

2.1.3. *Conditional execution.* Conditional execution includes constructions that change the order in which instructions are executed based upon some state. The most common construction that we will first use is the `if` statement, which appears in pseudocode appears as:

```
if( <conditional expression> ) {
      <then-block>
} else {
     <else-block>
}
```

Or,

```
if( <conditional expression> ) {
    <then-block>
}
```

In these skeletal statements, the `<then-block>` and the `<else-block>` consists of zero or any number of JavaScript statements. The next Algorithm shows the pseudocode representation of conditional statements that are used to ensure that only a non-negative difference between two positive integers is computed. (Note the use of the "require" and "ensure" markers in the next example; these are "preconditions" and "postconditions" when discussed in class.)

Algorithm 3 Return the absolute difference of two positive integers.

Require: Two positive integers are provided by the user.
Ensure: Their absolute difference is returned.

$v1 \leftarrow$ getInput("Enter a positive integer: '')
$v2 \leftarrow$ getInput("Enter a positive integer: '')
if $v1 \leq v2$ **then** ▷ Compare one against the other
  return $v2 - v1$ ▷ Ensure that the difference is non-negative
else return $v1 - v2$
end if

What the last example does not explicitly show is that

- An `if` statement need not have an `<else-block>` .
- `if` statements can contain more than 2 conditions:

```
if( <condition1> ) {
     do stuff ...
} else if( <condition2> ) {
    do different stuff ..
} else {
    when all else fails ...
}
```

Often, languages provide additional "shorthands" for commonly-requested patterns of conditional execution.[1]

- And, `if` statements may appear within `if` statements:

```
if( <condition1> } {
   if( <nestedCondition1> ) {
      ... } else {
        // note the importance of spacing here!
   } else {
     // otherwise which statement owns this else?
}
```

The examples above were provided in JavaScript, and so we should follow-up with the pseudo-code equivalent, especially for that last usage:

Algorithm 4 Example of multi-part if statement in pseudo-code.

$choice \leftarrow$ getValueFrom('$itemsList$')
if $choice = Win$ **then**
  $score \leftarrow score + 1$
else if $choice = Lose$ **then**
  $score \leftarrow score - 1$
else if $choice = Reset$ **then**
  $score = 0$
else
  $document.getElementById)(\text{“}results''\text{).}innertHTML = score;$
end if

2.1.4. *Iteration.* JavaScript provides several statements that allow us to express repitition, or "iteration" as it is more commonly called. When discussing iteration, we often distinguish *bounded* from *unbounded* iteration, or, said differently, repeating until some state becomes true or false as opposed to repeating a fixed number of times. JavaScript provides several kinds of iterative constructions based upon this distinction.

As an example of "unbounded" iteration, consider the problem of repeating a set of actions until some state is seen: we will call this the "termination condition." JavaScript provides several ways of saying this, depending upon when we wish to check for the termination condition. Examine the following script and its accompanying pseudocode:

```
function sum( range ) {
   var total=0;
   while( range > 0 ) {
       total += range;
       range--;
   }
   return total;
}
```

[1]A little later, we will examine the switch statement, but you should know that its behavior can be completely described as a series of if statements, albeit with some additional complications.

Here's the pseudocode:

function SUM(range)
  $total \leftarrow 0$
  while $range > 0$ **do**
    $total \leftarrow total + range$
    $range \leftarrow range - 1$
  end while
  return *sum*
end function

Try to describe in English what it computes, identifying its preconditions and post conditions. See if you can also identify (describe in unambiguous English) its "terminating condition." Ask:

- how does its terminating condition relate to its precondition(s)?
- when is the terminating condition "tested" in this algorithm?

In addition to the `while-statement`, JavaScript provides the `do-while` statement which differs from the `while` statement in that it places the "test case" *after* the body of instructions. This means that the body of instructions is executed *at least once* and then some test is performed to determine if the body of instructions is to be executed again. This pattern is commonly seen in "nag" code:

```
function nag( message, low, high ) {
   var result;
   do {
     result = parseInt( window.prompt( message ));
   while( result < low || result > high )
   return result;
}
```

Perhaps the logic is clearer in the pseudocode?

function NAG(message, low, high)
  result ← `undef`
  repeat
    message ← WINDOW.PROMPT(*message*)
    result ← PARSEINT(*message*)
  until *result* ≥ *low* and *result* ≤ *high*
  return *result*
end function

In an attempt to document the `nag` function, we can ask some questions, in particular:

(1) What do we know about the "types" of the arguments to the `nag` function?
(2) What do we know about the "type" of the value returned by the `nag` function?

We know that the `message` must be a `String`, and both the `low` and `high` arguments must be numbers that can be compared. Finally, we know that the "type" of the value returned by the `nag` function must be an `int`. Why?

Use this information to construct the preconditions and postconditions for this function.

2.2. **Matching control structures with data-structures.** For some data-structures, in particular, arrays and strings (which we shall discuss next week), the `for` statement is often used because the sizes of arrays and strings are known ahead of time and the `for` statement is a good fit:

```
var anArray = [ 1, 2, 3 ];
for( var index=0; index < anArray.length; index++ ) {
    window.alert( 'For index' + index + 'found  ' + anArray[ index ] + '\n';
}
```

It is instructive to write this in pseudo-code:

Algorithm 5 Iterate over elements in an array. Note, we hide the details of string creation.

array $\leftarrow [1, 2, 3]$
for $i \leftarrow 0$; $i <$ *array.length*; $i \leftarrow i + 1$; **do**
    WINDOW.ALERT(...)
end for

Although we have not yet talked about strings as data-types, we'll see that the class (data-type) `string` provides a *method* called `length` that serves the same purpose as the `length` property provided for arrays, and therefore many of the iterative constructions that we use for one we will also be able to use for the other, albeit with some appropriate changes.

3. Putting it together ...

As you're likely working on Project 4 at this time, you are being asked to write two *interchange sorts*, the `selectionSort` and the `insertionSort`. These are well-known sorts of this class, i.e., the class "interchange sorts." Let's review how these sorts work.

3.1. **Sorting by Selection.** The idea here is intuitive and simple: given a non-empty array of integers, choose the smallest and place it in the first location, swapping what was originally in the first location for the object you just moved.[2] Move to the next location in the array and continue this process. Upon arriving at the end of the array, all of its elements should be in their correct locations, i.e., in ascending order. Note that duplicates will be neighbors. We can visualize this by looking at the unsorted elements of an array:

```
var anArray = [ 3, 1, 2, 4, 0 ];
```

Suppose that we ask, "what is the *index* (location in the array) of the *smallest element*, starting from some index (the start of the array in this case)? Well, it's the `0` which is found at end of the array, which is index= 4 (make sure that you see how this index was determined).[3]

Exchange the contents of `anArray` so that what lived at index 0 now lives at index 4, and vice versa, and this gives:

```
[ 0, 1, 2, 4, 3 ]
```

[2]Although the algorithm is assumed to be given integers, this same algorithm works for any collection of objects that can be "compared." We will revisit these algorithms, for example, when we discuss strings.

[3]If we had more than one 0, we could choose *any* of them, it turns out, because we will eventually have to examine each of them in the process of sorting the entire array.

But, we're not done because we have not accounted for every element in the array. We now examine the next element, which is at index= 1, and we see that it is a `1` and that it is smaller than any element to its right; so we leave it where it is. Ditto for the `2`. When we continue with the `4`, however, we see that it is not the smallest element, instead we find the `3` at index= 4 and we exchange the elements at index= 3 (the number 4) with the element residing at index= 4 (the number 3), and we get:

```
[ 0, 1, 2, 3, 4 ]
```

Our algorithm now sees that it is at index= 4, which is the length of `anArray` and we are done: meaning that `anArray` is now sorted, in ascending order.

3.1.1. *Reducing the idea to pseudo-code.* Let's sketch what we've done in a algorithm using pseudo-code. In order to simplify matters, we assume that we can construct two additional helpers here: one called `findMin()` that takes the starting index and the array in question and returns the index of the smallest element to the right of the index, and `swap()` that takes the current index, the index of the smallest element, and the array, and if these indices are different, exchanges the elements belonging to each index within the array.

Algorithm 6 Sketch of a selection sorting algorithm.

for $i \leftarrow 0$; $i < anArray.length$; $i \leftarrow i + 1$ **do**
  $indexSmallest \leftarrow$ FINDMIN$(i, anArray)$
  SWAP$(indexSmallest, i, anArray)$
end for

Often, we *postpone* providing the definitions of all of the required functions and procedures when sketching an algorithm. In software engineering, this is sometimes called "writing a stub" which is essentially an IOU for the missing method. Doing this simplifies the design process by allowing us to focus on a particular aspect of the problem with the expectation that the missing stubs will be provided as needed. In this case, we have stubbed the definitions of a function, `findMin()` and a procedure, `swap()`. Before we define these we should make sure that we understand the algorithm as written, and that we are convinced that it is *correct*.

3.1.2. *Asking about the correctness of our algorithm.* Showing that the algorithm is correct ensures that we understand it. Computer scientists often provide formal proofs in response to this question, but we can do just as well using a less formal approach: Begin by asking how could we convince ourselves that the algorithm, as described above, is "correct," meaning that it sorts, i.e., arranges in ascending order, a finite array of integers (or any other kind of objects that allow for comparisons)?

Maybe we can begin by asking: can we find a case where the algorithm does not work? Consider the smallest possible array which is the empty array. Let's agree to say that the empty array is sorted—*by definition*. What about an array that contains only one element? It's also sorted because that's the only possbililty. What about an array containing two elements? Well, either it's already sorted meaning that the smallest element appears before the larger element(s), which can be determined by one comparison, or it can be arranged to be sorted within one exchange of elements.

It should be pretty clear now that for arrays of any size n, we can perform a finite number of comparisons and exchanges $(n - 1)$ so that their elements are "sorted," leaving only one element which is either already in its proper place, or can be exchanged in one step with an element already in the array and this results in a totally ordered (sorted) array.

3.2. **Developing the missing procedures ...** Back to the task: first we will discuss/develop the `swap()` *procedure* and then devote some time to the more "interesting" function, `findMin()`.

3.2.1. *Swapping contents around an array.* Arguably, one key notion here is that we are *modifying* the array being given to us; `swap` *is not creating and returning a new array*, thus it is a *procedure*—it is executed for its "effect."

As we discussed in class, this problem is similar to exchanging the contents of two glasses of water: we need a third class, large enough to hold the contents of one of the glasses in question. And all glasses should be large enough to hold the others' contents. Fortunately, the way that the language works and the way in which we have structured our problem ensures that these arrays will contain elements that are compatible. This leaves the task of writing out the sequence of steps required. Let's sketch it out in pseudo-code, leaving the actual JavaScript implementation to you:

Algorithm 7 Swap exchanges the contents of its array in-place, which means that the array parameter is *modfied* as a result of calling the procedure—hence, we note this in the statement of its pseudo-code definition.

procedure SWAP(from, to, array)
  $tmp \leftarrow array[from]$
  $array[from] \leftarrow array[to]$
  $array[to] \leftarrow tmp$
end procedure

We have given you the postconditions for this procedure, but as a habit of mind your should ask "what are the preconditions for the successful use of the `swap` procedure?

3.3. **Finding min/max elements in an array.** First, we should observe that an empty array contains no maximum or minimum elements, thus we have at least one precondition here: the arrays in question should be non-empty. Second, we need to be careful to distinguish the case where we are asked for the "element" as opposed to when we are asked for the "location of the element." Lastly, we need to be a little sensitive to the question of whether or not it matters that elements may be repeated within an array.[4]

For the majority of the sorting algorithms that operate over arrays, we are less interested in the "value" of a particular element, but more often its *location* because we need to *exchange* (or "swap") elements at one location with elements at another location in the process of "ordering" the array. Moreover, we need to be clear from the outset about how we will handle certain "exceptional" conditions. For one, the empty array contains no minimum or maximum elements (by definition—meaning that we have agreed to this ahead of time). Thus, we need to explicitly state as a precondition that our function(s) are only meaningful for non-empty arrays (again, of objects that may meanginfully be considered as having comparable values). I will leave it to the reader to either write some candidate statements at this point, or to at least make a "mental note" of these as this will be relevant in the definition of the pseudo-code.

[4]For the mathematics and philosophy majors: we have a more fundamental question here: how do we ensure that any non-empty collection of objects *must contain* a minimum or maximum element? This question is somewhat finessed by our choice of integers. But, you should ask: is is always the case that a finite collection of objects has a minimum or maximum element?

3.3.1. *Developing the pseudo-code ...* In developing the pseudo-code, we follow the same pattern of reasoning that we used in the previous section where we discussed the "correctness" of the seletion sorting algorithm. Following that line of reasoning: we ask what is the smallest example of a non-empty array of integers, and, based upon our answer to this question, what must be the index of its minimum element?

Algorithm 8 Finds the index of the minimum (smallest) element (integer) in the array.

```
1: function findMin(start, array)
2:     indexMin ← start                          ▷ Consider first element the smallest
3:             ▷ Follow the bounds-checking logic in this for statement. Do you see why it works?
4:     for index ← start + 1; index < array.length; index ← index + 1 do
5:         if array[index] < array[indexMin] then
6:             indexMin ← index
7:         end if
8:     end for
9:     return indexMin
10: end function
```

To make it easier to outline our logic, I've numbered the lines in the algorithm.

Line 1: We must provide two parameters: one `start` is a non-negative integer that tells the function where to start its search. This is necessary in order to ensure that we do not go backwards!

Line 2: Assume that the array contains *exactly* one element, then that element must be the minimum (smallest), and that would mean that it is the element that corresponds to the index `start`. In that case, convince yourself that the `for` statement *does not execute*, and the `indexMin` (which is the `start`) is returned.

Line 4: Pay close attention to how we make sure that the `index` in the `for` statement cannot ever be set to a value equal to the length of the `array` because this would cause an array out of bounds error!

Line 5: Again, check how for `for` statement was set up. Substitute a small integer, such as 2, for the length of the `array` and convince yourself that this `if` statement is "safe."

Line 6: Sets a new "winner" in the event that some element to the right of the original winner (`indexMin`) has been discovered in our comparison.

Line 9: Don't forget to return the index that has been discovered by our search of the array.

3.3.2. *Review of reasoning about algorithms.* The `findMin` algorithm, above, is complex at first, but with time, it should become somewhat easier. It's not the language (usually) that makes understanding algorithms complex; I think that it's the ability to see how the steps of any particular algorithm contribute to the solution. In this regard, many computer science algorithms have a similar "framework" in that they can be approached as responding to the following question(s):

- What is the "smallest" example of the problem being discussed here?
- What are the structural considerations, i.e., how is the problem being represented?
- What does success look like?
- What must *always* be true during every step of this algorithm?

3.4. **Putting these elements together in order to sort an array of integers.** Having identified the parts, we can assemble them into a "whole," which is the algorithm that we call `selectionSort` and is presented below:

Algorithm 9 Sorts an array of integers by selection.

```
procedure selectionSort(array)
    for start ← 0; start < array.length; start ← start + 1 do      ▷ For each element in array
        minIndex ← findMin(start, array)                            ▷ Get the index of the next smallest
        if minIndex ≠ start then                                    ▷ If it's not this one, then swap it around.
            swap(start, indexMin, array)
        end if
    end for
end procedure
```

Again: this is a *procedure* because it does not return any value(s), instead the `selectionSort` procedure *modifies* the `array` in place. Most sorting routines do this because they are asked to sorts millions of items, and usually these items must be in memory in order to do so. We do not want to make copies of millions of data items in order to re-arrange the original array!

Before talking about yet another way of sorting an array of integers, you should review the discussion on the previous page about how one thinks of algorithms. See if you can apply that kind of thinking to this last section: ask, what is the "smallest problem" and how does that kind of thinking help us in designing and understanding algorithms? Can I see that reasoning at work in these last few examples?

4. Another way of sorting elements in an array

For Project 4, you're asked to provide two sorting rouines: one sorts by selection, and we just outlined that algorithm above. The other sorts by "insertion," which we now describe in some detail.

Suppose you are being dealt a hand of five playing cards, one at a time, and that you wanted to arrange these cards in ascending order by their face-values. The first card that you receive must be in order, because any collection of one is ordered. The next card must be compared with the first card and placed either before or after it. You would continue in this fashion until the last card was dealt; always searching for a place for the next card dealt.

Let's make this clearer with some data. Keep in mind that unlike the description above, we will be given (dealt) the entire collection of elements (playing cards in one hand), but we will examine them one at a time, from left to right. Consider

```
var myArray = [2, 6, 1, 4, 2];
```

Start by examining the element at the first position and ask is this element in sort-order? Note, because it is only one element, it must be in order because any collection containing only one element must be in order. Move right by one and starting from the beginning of the array, find an *insertion point*, i.e., a place in the array where the element satisfies the ordering relation.

```
[ 2, 6, ... ]
```

Advance to the next index, and ask about the `1`:

```
[2, 6, 1, ... ]
```

Here we see that `1` is *out of order* so search from the start of the array and look for a place for the `1` such that the sub-array from the starting index (index at 0) to this point is *ordered*. And, we find that the first position is the only place where the `1` is allowed if we're to maintain an ordered array, giving us:

```
[1, 2, 6, ...]
```

We advance to the next element (we left off where the `6` currently resides) and we repeat the process by moving the `4` to a location between the start of the array and the current index, giving us:

```
[1, 2, 4, 6, 2,]
```

Finally, we find a place for the the last element, `2`. Again, we do this by examining the array from start to this point, giving us:

```
[1, 2, 2, 4, 6]
```

Having reached the end of the array, we're done.

4.1. **Reducing these moves to an algorithm.** Our next task is to reconstruct the procedure we outlined above as an algorithm. In class, I always ask: what is the smallest case(s) that we need to consider.

- The empty array which we shall define (agree by definition) as sorted.
- Any array containing only one element must be ordered (in sort-order).
- Arrays containing more than one element can visualized as two *sub-arrays*, one we call "left" the other "right." The left is the array beginning at index=0 and continuing through the last element sorted, and the right contains the elements starting with the element being sorted and continuing to the end of the array. *Insert* the element at the current index (this is the element being sorted) into left sub-array in its proper position, advance the index, and continue.

Algorithm 10 Selection sort: sorts any array of integers in ascending order.

```
1: procedure insertionSort(array)
2:     valueToInsert, atPosition
3:     for i ← 1; i < array.length; i ← i + 1 do
4:         valueToInsert ← array[i]
5:         atPosition ← i
6:         while atPosition > 0 and array[atPosition − 1] > valueToInsert do
7:             array[atPosition] ← array[atPosition − 1]
8:             atPosition ← atPosition − 1
9:         end while
10:        array[atPosition] ← valueToInsert
11:    end for
12: end procedure
```

5. Some fine points about translating Pseudo-code to JavaScript

I encourage students to get comfortable with pseudo-code because it's a better way of learning how to think about programming and algorithms, in general. The problem with teaching algorithms exclusively through the use of any one programming language is that students become dependent

upon the features of a particular language and fail to see the "big picture" absent the context (and artificial supports) provided by the language.

This being said, when interpreting pseudo-code into any language, you need to be mindful of some subtle problems. In the case of JavaScript, we have a few which we discuss below.

5.1. **The use of the equals sign.** First and foremost, the equals sign, =, in JavaScript is problematic. It means "assignment," which is written in pseudo-code as a left-facing arrow, $\leftarrow$. Recall from class, the assignment statement stores the value on its right into the location on its left. Therefore you must replace all left-handed arrows with `=` in JavaScript.

5.2. **Shorthand operators.** Most modern languages provide lots of "shorthand" operators for commonly used programming idioms, such as adding 1 to a variable. The following are all equivalent in JavaScript

```
n = n + 1;
n++;
n += 1;
```

But, this can sometimes cause problems, consider the following translation from pseudo-code to JavaScript:

for $i \leftarrow 0, i < size, i \leftarrow i + 1$ **do**
  $answer \leftarrow answer + i$
end for

Then, we write

```
for( var i=0; i < size; i = i+1 ) {
   answer += i;
}
```

The use of the `i = i+ 1` in the `for` statement as it is written in the JavaScript makes me very nervous. You're better off writing:

```
for( var i = 0; i < size; i++ ) {
   answer += i;
}
```

5.3. **Comparing for equality.** We have several ways of comparing two objects for equality in JavaScript:

```
x == y
x === y
```

Usually, these give the same results. But the use of `===` is preferred unless you have a reason for not caring or ensuring that the *types* of both `x` and `y` are the *same* as well as their "values." Obviously, in pseudo-code, the single = is used for equality because the left arrow symbol is provided for assignment.

5.4. **Procedure versus Function.** JavaScript does NOT distinguish procedures from functions; you must use `function` regardless of the pseudo-code's usage of either term.