Brave New World

Pseudocode Reference

Pseudocode is a way to describe how to accomplish tasks using basic steps like those a computer might perform. The advantage of pseudocode over plain English is that it has a precise meaning that allows us to express a computation clearly and precisely. The difference between pseudocode and actual code is that pseudocode is meant for paper only and its exact syntax is not important. Pseudocode will not necessarily work if you put it into an actual program. Because it may be cumbersome to use actual working code when you only need to lay out an algorithm on paper for reference purposes, pseudocode comes in handy.

You can use the handout below for your homework.

Variables

In pseudocode, as in real programming, you will need to use *variables* to store data. You can think of these as little boxes that hold information, and you are allowed to look at what's in the box or replace its contents with something else. We call whatever is inside the box the *value* of the variable.

An *array* is a shorthand way of naming a bunch of variables. If *A* is an array of length *n*, you can imagine it as *n* boxes lined up in a row. Then we write *A*[*i*] to refer to the *i*'th box, where i = 1is the first box.[1] Here's a picture of what *A* might look like in memory:

A =	40.20	62.71	52.54	…	22.05

Here, A[1] is 40.20.

Here's how you can create an array of integers in pseudocode:

```
A = int[5]
```

This means that A is "an array of five elements (integers)", but does not specify what those elements are. Or something like:

```
A = {40.20, 62.71, 52.54, 22.05}
```

[1] In most programming languages, an array of length n would be indexed from 0 to n-1, rather than 1 to n.

This means A is an array of size four whose elements are 40.20, 72.71, 52.54, and 22.05.

You can use arrays in pseudocode instructions the same way you use variables:

`x = A[2]`	Sets *x* to the second value in the array *A* (here, 62.71)
`A[3] = 2`	Sets the third value in the array *A* to the value 2 (replacing 52.54)

Sometimes you will use a variable to specify which element of the array you mean:

`y = A[i]`	Sets *y* to the *i*'th array value

Arrays can be *multidimensional*. While a one-dimensional array is like a list, a two-dimensional array is like a grid. If *A* is a two-dimensional array, *A*[*i*][*j*] refers to the value in row *i*, column *j* of the grid.

Instructions

A pseudocode program is written as a series of *instructions* that the computer should execute one at a time (which does not exclude possibility of looping over a set of instructions, which we'll see later). But basically you should assume that your pseudocode will be traced instruction by instruction. These are several kinds of instructions:

Arithmetic Instructions

Arithmetic instructions usually affect values stored in v*ariables*, named pieces of memory. These instructions take the form *variable = arithmetic expression*. For example:

```
x = 5       // Sets the value of variable x to 5
y = x       // Sets y to x's value; leaves x unchanged
i = j + 1   // Sets i to the value j + 1; leaves j unchanged
```

For our purposes here, there are only a few basic arithmetic operations, these are addition, subtraction, multiplication, and division. Note that we typically use * for the multiplication operator (to distinguish from a variable called x) and / for division. Exponentiation, logarithms, and more complex operations are *not* basic.

Two useful operations that are a little non-standard but that you may also use when you write pseudocode are the ceiling and floor operators. ceil(*x*) rounds x up, denoting the smallest integer greater than or equal to *x*. For example, ceil(3.4) = 4, ceil(4.1) = 5, ceil(2) = 2. floor(*x*) rounds x down, or truncates, and is defined analogously as the greatest integer less than or equal to *x*.

Conditionals

Conditional ("branch") instructions perform one set of actions only if some specified condition is true and another set only if the condition is false. They take this form:

```
if (true/false condition) {
   First list of instructions…
} else {
   Second list of instructions…
}
```

(You can omit the else branch if you don't want to take any special action when the condition is false.)

Here is a simple example:

```
if (x is odd) {
   x = x – 1
   count = count + 1
} else {
   x = x /2
}
```

This conditional checks whether *x* is odd. If so, it subtracts one from *x* and adds one to *count*; otherwise, it divides *x* by two.

It's important to understand that the true/false condition is checked only once, at the beginning of the conditional. If *x* is odd, the computer subtracts one from *x*, making it even. However, the computer **does not** go on to divide *x* by two.

You can also have a more complex structure of conditionals, when you would check one condition first, and if it is not true, then check another condition, then, if the second condition is also false, check the third, et cetera. If none of the conditions is true, then only you would fall through to the last (default) else clause:

```
if (x == 1) {
    print "One"
} else if (x == 2) {
    print "Two"
} else {
    print "Not One and not Two"
}
```

Here, like in Processing, == means "is equal to". It is the equality operator as opposed to the assignment operator (=). In other words, y=x means "give y the value x has", while

"y==x" is a statement which is either true or false, depending on whether y and x have the same value or not.
For example, consider the following pseudocode:

```
x = 5    // x is now 5
y = 10   // y is now 10
x == y   // x is not equal to y so this is FALSE;
         // note that this is not a valid instruction!!!!
y = x    // y gets the value of x and therefore is now 5
x == y   // this is now TRUE  (again, this is an expression, not an
instruction)
```

Loops

Loops perform a set of actions some number of times. The one that is used a lot makes the computer follow a series of instructions a fixed number of times. For example, the following loop will execute 100 times, with *n* taking on different values (from 1 to 100) each time:

```
for (n=1 to 100) {
   // List of instructions…
}
```

The next kind of loop makes certain instructions get executed repeatedly as long as a specified condition remains true. It takes the form:

```
while (true/false condition) {
   // List of instructions…
}
```

At the beginning of the loop, the computer checks whether the condition is true. If so, it executes the list of instructions. Then it checks again whether the condition is true; if so, it executes the instructions again. This process is repeated until the computer checks the condition and it is false. Here's an example:

```
while (current < max) {
    current = current + 1
    // Other instructions…
}
```

Finally, this loop performs an action over and over again "infinitely" (or at least until the computer or robot is turned off):

```
while (true) {
   // List of instructions…
}
```

It is easy to see that this is exactly the same as a regular while loop, except the condition here is always true.

What happens if *current ≥ max* the first time the computer evaluates the while instruction? In this case, the operations inside the loop are not executed at all.

Input and Output

Sometimes you want to get values from some external source outside the program. For example: `Get price` will set a variable *price* to a value from outside the program.

Similarly, your program can present its results using instructions like these:

```
print "The lowest price was:" // Display a fixed message

print minimum   // Display the value of the variable minimum
```

Caveat

Let's reiterate: pseudocode looks deceptively like English, and that is its advantage: it should be understandable by your average lay-person. However, **please** be aware of the differences between pseudocode and plain English: pseudocode is meant to be executed *verbatim*, and not "interpreted" with human common sense. For example, a conditional statement is executed only *once*, whereas a loop is repeated.

Comments

Comments are not actually instructions; instead, they provide hints about what the program does to help humans understand it. They are useful in pseudocode as well.

Remember, comments don't change the meaning of your program, since computers skip over them entirely. They do help make the meaning clearer to human readers. Too many comments is usually not good, no comments at all is also usually not good. For very simple programs comments are usually not necessary.

Comments can be represented by two slashes (as in Processing):

```
// This is a comment
x = 5  // we are setting x's value to 5; this comment is
       // unnecessary!
f = (9/5) * c + 32 // Convert Celsius to Fahrenheit; a more useful
                   // comment
```

Understanding pseudocode

Just as a child first learns to understand language, and then to speak, and finally to write, you also should first learn to understand pseudocode written by others and only then attempt to write your own.

In order to understand pseudocode, you don't just read it as you would a poem or a novel. You work through it. You take a blank sheet of paper, designate some space on the paper for the variables, arrays etc. Then you "execute" the pseudocode line by line on this data. After you do this a few times, you begin to understand what it does.

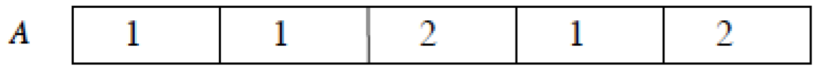
Example 1: Vote counting machine

We illustrate this with the following program that counts votes for two candidates. Votes—for candidate 1 or candidate 2—are read one by one. Let's assume that the votes are given to us in an array *A*, and that there are *n* votes total. Let's also assume for simplicity that each vote is stored as either "1" or "2".

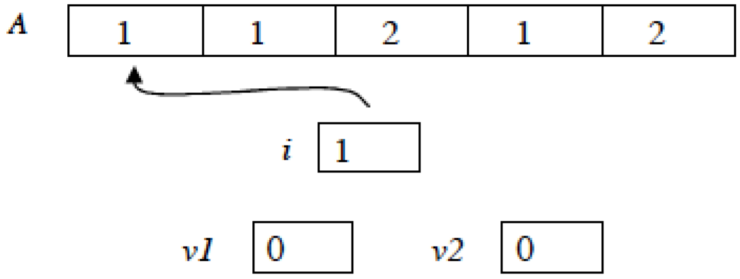
```
1        // Set initial vote counts to zero
2        v1 = 0 // v1 holds the tally for candidate 1
3        v2 = 0 // v2 holds the tally for candidate 2

4        for (i = 1 to n) {
5           // See who the next vote is for
6           if (A[ i ] ==1) {
7              // If it's for candidate 1, then increment his tally
9              v1 = v1 + 1
10          } else {
11             // Otherwise it's for candidate 2, so increment his
12             // tally
13             v2 = v2 + 1
14          }
15       }
16       Print "Totals:", v1, v2
```

On the next page, we work through an example with this pseudocode.

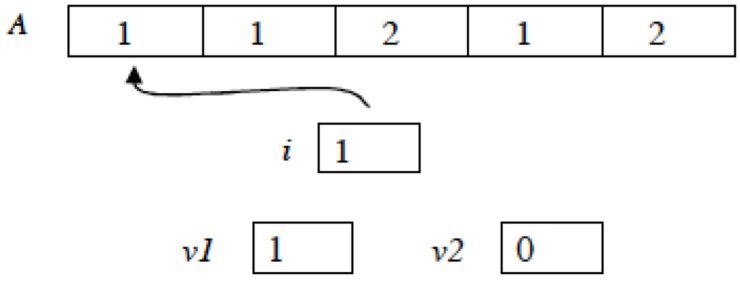
Let's work through an example with this pseudocode and see exactly what happens when we execute it. Suppose $n = 5$ and we have the following data in an array A.



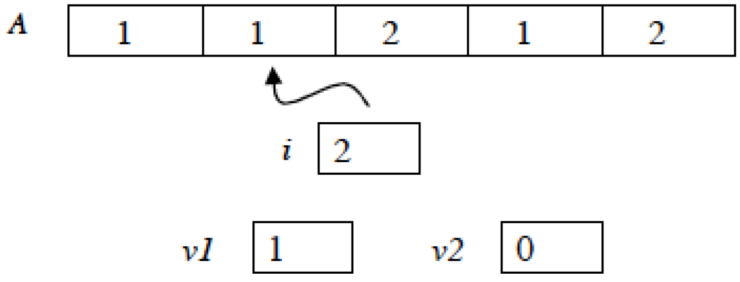
We have our two variables to keep a tally *v1* and *v2*, and our loop variable *i*. We will initialize *v1* and *v2* to 0, and the first time through the loop we have $i = 1$, which we will depict as pointing to the first spot in A. Thus at the beginning of the first loop the picture looks like this:



Now we look to see if $A[\, i \,] = 1$. In this case it is indeed, so we increment *v1*.



Now we loop around and increment *i*, which gives us the picture



We check whether $A[\, i \,] = 1$, which it is, so we increment *v1* again.

Example 2: Sorting a list of numbers

This program sorts a list of *n* numbers using the selection sort method discussed in lecture. Notice how comments make the program easier to understand.

```
1        // Input n and the list of numbers, which are stored in the
2        array A
2        Get n, A[1], …, A[n]
3        for (i = 1 to n-1)
5           // Search from position i to position n in the array; find
2           // the minimum value,
6           // and record its position in best.
7           best = i
8           for (j = i+1 to n) {
10             if (A[i] < A[best]) {
12                best = i
13             }
14          }
15          // Swap the minimum value (A[best]) with the i'th value
16          tmp = A[best]
17          A[best] = A[i]
18          A[i] = tmp
19       }
20       Print A[1], …, A[n]
```

Suggestions for writing your own pseudocode

Unfortunately there is no one way to convert an idea of an algorithm into a pseudocode. (Think about it, this would in essence be an algorithm for writing algorithms!) But to get you pointed in the right direction, here are several general guidelines that will help you in writing your own pseudocode.

Let's think again about Example 1: the vote counting machine. Remember our goal: count all the votes and then print out the number of votes for each candidate. Let's think about how to write the pseudocode for this task.

Points to consider when thinking about the algorithm:

- Imagine giving your program to a 7-year old who can understand English and do elementary arithmetic but doesn't have much common sense or experience. He or she should be able to understand exactly what to do given your pseudocode.

- Your program should work for arbitrarily long input, in this case arbitrarily many votes. Thus, although saying "Just count the votes" might make sense for 10 votes,

if you are given 10,000,000,000 votes then it's not as obvious what "Just count the votes" means.

- Remember that the instructions are executed step by step. Whoever is running your program is not allowed to look at the program "as a whole" to guess what you actually meant it to do.

With these points in mind, let's think about how to count votes. Say we are given the votes in a big pile. One way to count would be:

Idea A*:*

Take the first vote, see who it's for. If it's for candidate 1 then mark a tally for 1, or if it's for 2 then mark a tally for 2. Then keep repeating this for the rest of the votes until you go through the entire pile.

This is the idea (or the algorithm). Now we need to turn it into pseudocode.

Points to consider when you are trying to turn an idea into pseudocode:

- What kind of information is recorded in the process of doing the task? This information will have to be stored in variables when you write the pseudocode.

- Where do you make decisions about selecting one of two actions to do? These will usually become conditional statements in the pseudocode.

- Where do you repeat things? These will become loops in the pseudocode.

OK now let's look at Idea A and try to translate it into pseudocode.

- What are we keeping track of? The tallies of votes for each candidate. Thus, these two tallies will become variables in the pseudocode.

- Where do we make a decision between two actions? When we decide which candidate's tally we should add to. This will become a conditional statement.

- Where do we repeat? When we are done with one vote we move on to the next and repeat the same procedure. Going through the pile of votes is like looping through an array, where each element tells us someone's vote.

Now if you go back and look at the pseudocode for Example 1, you'll see exactly how the idea was transformed into pseudocode. Also, remember that there's more than one

way to write pseudocode for the same algorithm, just like there's more than one way to express the same idea in English.

How fast does your algorithm run?

The central measure of "goodness" of an algorithm (assuming it does its job correctly!) is how fast it runs. We want a machine-independent measure and this necessarily implies we have to sacrifice some precision. In general, the relative speeds of arithmetic operations (+, * etc.) differ among machines, but we will assume all of them take the same amount of time. The three central points to remember when discussing running time are:

1. Even though we call the speed of an algorithm its "running time", we won't actually measure it in seconds or minutes, but in the number of "***elementary operations***" it takes to run. For this class, elementary operations are arithmetic (addition, subtraction, multiplication, division), assigning a value to a variable, and condition checks (either in an "if" statement or in a "while" statement).

2. The running time in general ***depends on the size of the input***. For example, if we are sorting an array of n elements, it is natural (and unavoidable) that it will take longer to sort $n = 10{,}000{,}000$ elements than to sort $n = 10$ elements.

3. We will usually analyze ***worst-case*** running time. That is, how long will this algorithm run given the *worst possible* input of size n? Whenever in doubt, we err on the side of overestimation rather than underestimation.

Let's analyze the running time in Example 1.

It takes 2 steps to initialize the variables. Then we run the loop n times: each time, we check 1 condition (i.e. who the vote is for) and possibly make 1 assignment (i.e. incrementing the tally). Thus each time we go through the loop we execute at most 2 steps. Finally it takes 1 step at the end to print the results. Thus adding everything up the algorithm runs in time $2n + 3$.