COS 116
The Computational Universe
Pseudocode Reference

Pseudocode is a way to describe how to accomplish tasks using basic steps like those a computer might perform. In this week's lab, you saw how a form of pseudocode can be used to program the Scribbler robot. The Scribbler control panel has a point and click interface, but in the rest of the course you will write your own simple pseudocode to express computations. The advantage of pseudocode over plain English (as you saw in case of scribbler) is that it has a precise meaning. The exact syntax is not too important—what counts is expressing a computation clearly and precisely.

You can use the handout below in all your homework and exams (even "closed-book" exams).

- **Variables**

  In pseudocode you will need to use *variables* to store data. You can think of these as little boxes that hold a number, 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. Here's a picture of what *A* might look like in memory:

A =	40.20	62.71	52.54	…	22.05

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

$x \leftarrow A[2]$	Sets *x* to the second value in the array *A* (here, 62.71)
$A[3] \leftarrow 2$	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:

$y \leftarrow A[i]$	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 executes one at a time. These are several kinds of instructions:

- **Arithmetic Instructions**

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

$x \leftarrow 5$	Sets the value stored in variable *x* to the value 5
$y \leftarrow x$	Sets *y* to the value stored in *x*; leaves *x* unchanged
$i \leftarrow j + 1$	Sets *i* to the value *j* + 1; leaves *j* unchanged

There are only a few arithmetic operations allowed, these are addition, subtraction, multiplication, and division. Note that exponentiation, logarithms, and more complex operations are *not* basic.

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

(Note: The Scribbler doesn't support all pseudocode instructions. For instance, it does not understand arithmetic instructions, variables, or arrays.)

- **Simple actions**

Sometimes you can use a single instruction to specify a behavior. E.g.:

```
Move forward for 1s
Read the price from the next jar
Place the box back on the shelf.
```

Use instructions like these for actions that are secondary to your program's purpose.

- **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 Then
{
    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 Then
{
    $x \leftarrow x - 1$
    $count \leftarrow count + 1$
}
Else
{
    $x \leftarrow x \div 2$
}
```

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.

- **Loops**

Loops perform a set of actions some number of times. One kind of loop performs an action over and over again "infinitely" (or at least until the computer or robot is turned off). These look like this:

```
Do forever
{
    List of instructions…
}
```

Another flavor of loop makes the computer follow a series of instructions a fixed number of times. For example:

```
Do for $n$ times
{
    List of instructions…
}
```

A third kind of loop adds a variable that counts the passes through the loop:

```
Do for $i = a$ to $b$
{
    List of instructions…
}
```

On the first pass, the variable *i* is set to *a*. At the start of each successive pass, *i* is increased by one, until on the final pass it has the value *b*. In all, the computer performs the list of instructions b – a + 1 times. You could use such a loop to add the integers from 1 to 100 (inclusive) as follows:

```
sum ← 0
Do for i = 1 to 100
{
    sum ← sum + i
}
```

The last kind of loop makes the computer perform some instructions repeatedly as long as a specified condition remains true. It takes the form:

```
Do 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:

```
Do while current < max

{
        current ← current + 1
        Other instructions…
}
```

What happens if current ≥ max the first time the computer evaluates the Do 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 outside the program. For example:

Get *price*	Sets the 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*

- **Comments**

  Comments are not actually instructions; instead, they provide hints about what the program does to help humans understand it. A comment begins with two slashes // and continues until the end of the line, for example:

  ```
  f ← (9/5) * c + 32  // Convert Celsius to Fahrenheit
  ```

  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.

- **Caveat**

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.

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

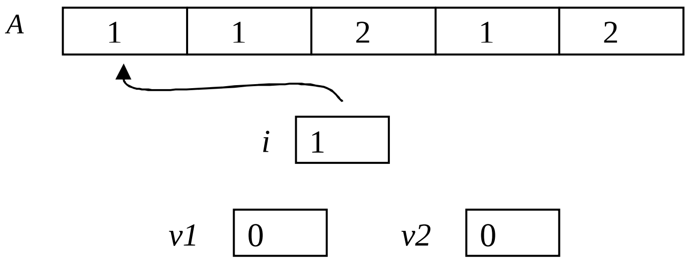
We illustrate this with the following program that counts votes for two candidates. Votes—for candidate 1 or candidate 2—are read one by one. Let's assume that the votes are given to us in an array *A*, and that there are *n* votes total. Let's also assume for simplicity that each vote is stored as either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

```
// Set initial vote counts to zero
v1 ← 0  // v1 holds the tally for candidate 1
v2 ← 0  // v2 holds the tally for candidate 2
Do for i = 1 to n
{
    // See who the next vote is for
    If A[ i ] = 1 then
    {
        // If it's for candidate 1, then increment his tally
        v1 ← v1 + 1
    }
    else
    {
        // Otherwise it's for candidate 2, so increment his tally
        v2 ← v2 + 1
    }
}
Print "Totals:", v1, v2
```

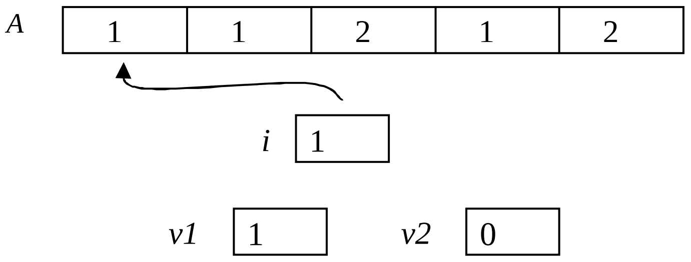
Let’s work through an example with this pseudocode and see exactly what happens when we execute it. Suppose n = 5 and we have the following data in an array A.

A	1	1	2	1	2

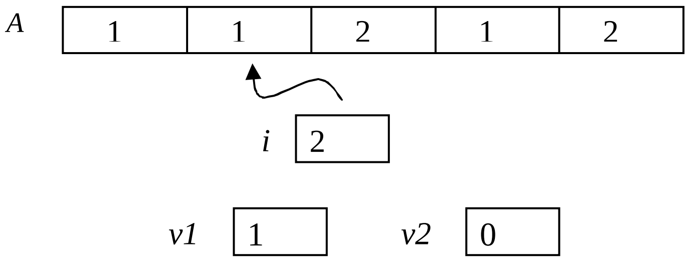
We have our two variables to keep a tally $v1$ and $v2$, and our loop variable i. We will initialize $v1$ and $v2$ to 0, and the first time through the loop we have i = 1, which we will depict as pointing to the first spot in A. Thus at the beginning of the first loop the picture looks like this:



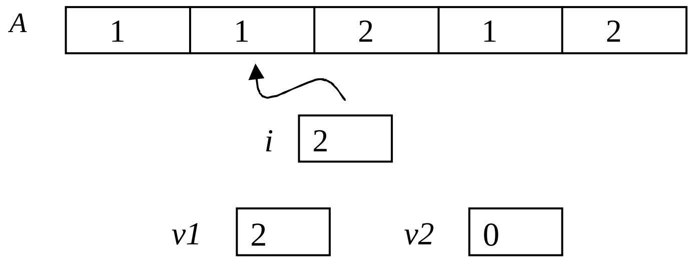
Now we look to see if A[i] = 1. In this case it is indeed, so we increment $v1$.



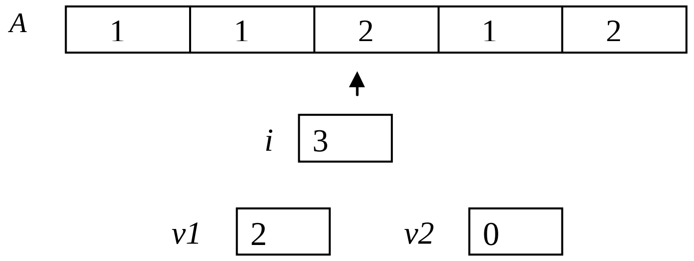
Now we loop around and increment i, which gives us the picture



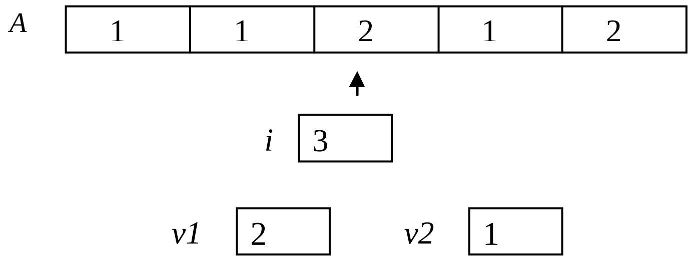
We check whether A[i] = 1, which it is, so we increment $v1$ again.



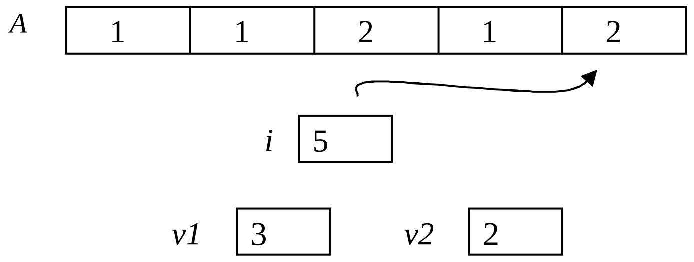
Then we increment *i* again to get



We check if *A*[*i*] = 1, and in this case it is not, so we increment *v2*.



Then we increment *i* and repeat two more times. At the end, our pseudocode will give us something that looks like:



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.

```
// Input n and the list of numbers, which are stored in the array A
```

```
Get $n$, $A[1]$, …, $A[n]$
Do for $i$ = 1 to $n$-1
{
    // Search from position i to position n in the array; find the minimum value,
    // and record its position in best.
    $best \leftarrow i$
    Do for $j$ = $i$+1 to $n$
    {
        If $A[j] < A[best]$ then
        {
            $best \leftarrow j$
        }
    }
    // Swap the minimum value (A[best]) with the i'th value
    $tmp \leftarrow A[best]$
    $A[best] \leftarrow A[i]$
    $A[i] \leftarrow tmp$
}
Print $A[1]$, …, $A[n]$
END
```

Example 3: The Game of life

This example simulates rounds in the Game of Life. The programs starts by getting the number of rounds to simulate (t), the size of the square grid (n) and the initial state of the grid (the array A). A grid value of 1 indicates an occupied square, and 0 indicates an unoccupied square. In every round, the program applies the rules of the game to each square to determine whether the square will be occupied in the next round. At the end of the round, the next round state (array B) replaces the current round state (array A).

```
// Input the number of rounds, the size of the grid, and the initial state of the grid
Get $t$, $n$, $A[1][1]$, $A[1][2]$, … , $A[n][n]$
Do for $step$ = 1 to $t$
{
    // On each round, examine every grid square
    Do for $i$ = 1 to $n$
    {
        Do for $j$ = 1 to $n$
        {
            // Count neighbors by adding all adjacent squares
            $neighbors \leftarrow$  $A[i-1][j-1]$ + $A[i-1][j]$ + $A[i-1][j+1]$ +
                                    $A[I\ ][j-1]$              + $A[i\ ][j+1]$ +
                                    $A[i+1][j-1]$ + $A[i+1][j]$ + $A[i+1][j+1]$;
            // Determine if current square will be occupied in the next round
            If $A[i][j]$ = 1 then
            {
                If neighbors = 2 or neighbors = 3 then
                {    B[i][j] ← 1    // Survival    }
                Else
                {    B[i][j] ← 0    // Death       }
            }
            Else
            {
                If neighbors = 3 then
                {    B[i][j] ← 1    // Birth       }
        }
    }
    // Update the grid with the next round's state
    Do for i = 1 to n
    {
        Do for j = 1 to n
        {
            A[i][j] ← B[i][j]
        }
    }
}
// Output the final state of the game
Print L[1][1], L[1][2], … , L[n][n]
END
```

(Notice that this program isn't correct for the squares at the corners and edges of the grid, which don't have 8 neighbors. How would you fix this?)

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:

1. 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.

2. 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.

3. 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:

1. 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.

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

3. 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.

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

2. 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.

3. 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 "do" 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 side of overestimation rather than underestimation.

Let's analyze the running time in Example 1.

```
// Set initial vote counts to zero
votes_for_candidate_1 ← 0
votes_for_candidate_2 ← 0
Do for i = 1 to n
{
    // See who the next vote is for
    If A[ i ] = 1 then
    {
        // If it's for candidate 1, then increment his tally
        votes_for_candidate_1 ← votes_for_candidate_1 + 1
    }
    else
    {
        // Otherwise it's for candidate 2, so increment his tally
        votes_for_candidate_2 ← votes_for_candidate_2 + 1
    }
}
Print "Totals:", votes_for_candidate_1, votes_for_candidate_2
```

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 $2n + 3$.