What is pseudocode?

Pseudocode is halfway between a programming language and English.

Pseudocode is not a formal language with strict rules but is a mixture of English and programming language and it allows you to quickly write down your algorithm with pen-and-paper before you type anything into your editor.

You can even develop your own pseudocode and use it to prepare your algorithms and programs.

Check out: **https://en.wikipedia.org/wiki/Pseudocode**

The pages below are from an earlier programming book on the **Modula 2** language (a language related to **Pascal**), which deals with algorithm development and this idea of pseudo code. (Starting from **section 2.2 Algorithm Development** at the bottom of the first page)

Please read this and take notes before you attempt Day 2 lab 1.

Obviously, we could generalize still more—allow several employees, other kinds of charges, different kinds of cable, and so on—but the line must be drawn somewhere or we could go on generalizing forever. In this elementary introduction to the problem-solving process, we wish to keep our examples quite simple.

PROBLEM 2: Pollution Indices. The level of air pollution in the city of Dogpatch is measured by a pollution index. Readings are made at 12:00 P.M. at three locations: the Abner Coal Plant, downtown at the corner of Daisy Avenue and 5th Street, and at a randomly selected location in a residential area. The average of these three readings is the pollution index, and values of 50 or greater for this index indicate a hazardous condition, whereas values of less than 50 indicate a safe condition. Since this calculation must be done daily, the Dogpatch Environmental Statistician would like a program that calculates the pollution index and then determines the appropriate condition, safe or hazardous.

The relevant given information consists of the three pollution readings and the cutoff value used to distinguish between safe and hazardous conditions. A solution to the problem consists of the pollution index and a message indicating the condition. Generalizing so that any cutoff value, not just 50, can be used, we could specify the problem as follows:

Input	Output
Three pollution readings	Pollution index = the average of the pollution readings
Cutoff value to distinguish between safe and hazardous conditions	Condition—safe or hazardous

PROBLEM 3: Summation. When the famous mathematician Carl Friedrich Gauss was a young student, his teacher instructed him to add the first 100 positive integers, 1, 2, 3, . . . , 100. (Perhaps this was a form of punishment comparable to "writing lines" today.) What is the value of this sum,

$$1 + 2 + 3 + \cdots + 100 = ?$$

Here the problem analysis is straightforward. Generalizing to find the sum

$$1 + 2 + 3 + \cdots + \mathit{LastNumber}$$

for any positive integer *LastNumber*, we can specify the problem by

Input	Output
LastNumber	Value of $1 + 2 + \cdots + \mathit{LastNumber}$

2.2 Algorithm Development

Once a problem has been specified, a procedure to produce the required output from the given input must be designed. Since the computer is a machine pos-

sessing no inherent problem-solving capabilities, this procedure must be formulated as a detailed sequence of simple steps. Such a procedure is called an ***algorithm***.

The steps that comprise an algorithm must be organized in a logical, clear manner so that the program that implements this algorithm is similarly well structured. ***Structured algorithms*** and ***programs*** are designed using three basic methods of control:

1. ***Sequential:*** Steps are performed in a strictly sequential manner, each step being executed exactly once.
2. ***Selection:*** One of several alternative actions is selected and executed.
3. ***Repetition:*** One or more steps is performed repeatedly.

These three structures appear to be very simple, but in fact they are sufficiently powerful that any algorithm can be constructed using them.

Programs to implement algorithms must be written in a language that can be understood by the computer. It is natural, therefore, to describe algorithms in a language that resembles the language used to write computer programs, that is, in a "pseudoprogramming language," or as it is more commonly called, ***pseudocode***.

Unlike the definitions of high-level programming languages such as Modula-2, there is no set of rules that defines precisely what is and what is not pseudocode. It varies from one programmer to another. Pseudocode is a mixture of natural language and symbols, terms, and other features commonly used in one or more high-level languages. Typically one finds the following features in the various pseudocodes that appear in textbooks.

1. The usual computer symbols are used for arithmetic operations: + for addition, − for subtraction, * for multiplication, and / for division.
2. Symbolic names (identifiers) are used to represent the quantities being processed by the algorithm.
3. Some provision is made for including comments. This is usually done by enclosing each comment between a pair of special symbols such as (* and *).
4. Certain key words that are common in high-level languages may be used: for example, *read* or *enter* to indicate an input operation; *display*, *print*, or *write* for output operations.
5. Indentation is used to set off certain key blocks of instructions.

The structure of an algorithm can also be displayed in a ***structure diagram*** that shows the various tasks that must be performed and their relation to one another. These diagrams are especially useful in describing algorithms for more complex problems and will be described in more detail in Section 2.5. In this section we restrict our attention to the three simple examples introduced in the preceding section. Using these examples, we illustrate the three basic control structures—sequential, selection, and repetition—and how to present algorithms in pseudocode.

PROBLEM 1: Revenue Calculation—Sequential Structure. As we noted in the preceding section, the input for this problem consists of the basic service charge, unit cable cost, number of installations, and yards of cable used; the output to be produced is the amount of revenue generated.

The first step in an algorithm for solving this problem is to obtain the values for the input items—basic service charge, unit cable cost, number of installations, and yards of cable. The next step is to determine the number of feet of cable used by multiplying the number of yards of cable by 3. The revenue generated can then be obtained by multiplying the number of locations by the basic service charge, multiplying the unit cable cost by the number of feet of cable, and adding these two products. Finally, the output value—revenue generated—must be displayed.

This algorithm can be expressed in pseudocode as follows:

ALGORITHM FOR REVENUE CALCULATION

(* This algorithm calculates *Revenue* generated by the installation of a certain number of yards of cable *(YardsOfCable)* at a given number *(Locations)* of locations. For each installation there is a fixed basic service charge *(ServiceCharge)* and an additional charge of *UnitCost* dollars for each foot of cable. *)

1. Enter *ServiceCharge*, *UnitCost*, *Locations*, and *YardsOfCable*.
2. Calculate *FeetOfCable* = 3 * *YardsOfCable*.
3. Calculate
   Revenue = *Locations* * *ServiceCharge* + *UnitCost* * *FeetOfCable*.
4. Display *Revenue*.

This algorithm uses only sequential control: the steps are executed in order, from beginning to end, with each step being performed exactly once. For other problems, however, the solution may require that some of the steps be performed in some situations and bypassed in others. This is illustrated by our second example.

PROBLEM 2: Pollution Index—Selection Structure. Recall that, for this problem, the input consists of three pollution readings and a cutoff value that distinguishes between safe and hazardous conditions. The output to be produced consists of the pollution index, which is the average of the three readings, and a message indicating the appropriate condition.

Once again, the first step in an algorithm to solve this problem is to obtain values for the input items—the three pollution readings and the cutoff value. The next step is to calculate the pollution index by averaging the three readings. Now, one of two possible actions must be selected; either a message indicating a safe condition must be displayed or a message indicating a hazardous condition must be displayed. The appropriate action is selected by comparing the pollution index with the cutoff value. In the pseudocode description of this algorithm that follows, this selection is indicated by

```
If Index < Cutoff then
   Display 'Safe condition'
Else
   Display 'Hazardous condition'
```

ALGORITHM FOR THE POLLUTION INDEX PROBLEM

(* This algorithm reads three pollution levels, *Level1*, *Level2*, and *Level3*, and a *Cutoff* value. It then calculates the pollution *Index*. If the value of *Index* is less than *Cutoff*, a message indicating a safe condition is displayed; otherwise, a message indicating a hazardous condition is displayed. *)

1. Enter *Level1*, *Level2*, *Level3*, and *Cutoff*.
2. Calculate

$$Index = \frac{Level1 + Level2 + Level3}{3}$$

3. If *Index* < *Cutoff* then
   Display 'Safe condition'
   Else
   Display 'Hazardous condition'

In addition to sequential processing and selection illustrated in the preceding two examples, the solution of other problems may require that a step or a collection of steps be repeated. This is illustrated in our third example.

PROBLEM 3: Summation—Repetition Structure. For this problem, the input consists simply of some positive integer *LastNumber*, and the output is the value of the sum $1 + 2 + \cdots + LastNumber$. To solve this problem in a "brute force" manner (and not using the clever technique discovered by Gauss), we might begin as follows:

```
    0
  + 1
  ---
    1
  + 2
  ---
    3
  + 3
  ---
    6
  + 4
  ---
   10
  + 5
  ---
   15
    .
    .
    .
```

(Although we might not actually write down the first two lines, but rather only "think" them, they are included here for completeness.) We see that the procedure involves two quantities:

1. A counter that is incremented by 1 at each step.
2. The sum of the integers from 1 up to that counter.

```
     0 — sum
+    1 — counter
     1 — sum
+    2 — counter
     3 — sum
+    3 — counter
     6 — sum
+    4 — counter
    10 — sum
     .
     .
     .
```

The procedure begins with 1 as the value of the counter and with 0 as the initial value of the sum. At each stage, the value of the counter is added to the sum, producing a new sum, and the value of the counter is increased by 1. These steps are repeated until eventually we reach

```
                .
                .
                .
+ LastNumber  — counter
??????????    — sum
LastNumber + 1 — counter   Stop!
```

When the value of the counter exceeds *LastNumber*, the value of the sum is the desired answer, and the computation stops.

In the pseudocode description of this algorithm that follows, the repetition is indicated by

While *Counter* ≤ *LastNumber* do the following:
a. Add *Counter* to *Sum*.
b. Increment *Counter* by 1.

This instruction specifies that statements a and b are to be repeated as long as the condition *Counter* ≤ *LastNumber* remains true. Thus, when the value of *Counter* exceeds that of *LastNumber*, this repetition is terminated and Statement 5 is executed.

ALGORITHM FOR THE SUMMATION PROBLEM

(* This algorithm calculates the value of the sum

$$1 + 2 + \cdots + LastNumber$$

for some positive integer *LastNumber*. It uses the variable *Counter* as a counter and the variable *Sum* for this sum. *)

1. Enter *LastNumber*.
2. Set *Counter* to 1.
3. Set *Sum* to 0.

4. While *Counter* ≤ *LastNumber* do the following:
   a. Add *Counter* to *Sum*.
   b. Increment *Counter* by 1.
5. Display *Sum*.

The three control structures in these examples, ***sequential***, ***selection***, and ***repetition***, are used throughout this text in designing algorithms to solve problems. The implementation of each of them in a Modula-2 program is considered in detail in later chapters.

2.3 Program Coding

The third step in using the computer to solve a problem is to express the algorithm in a programming language. In the second step, the algorithm may be described in a natural language or pseudocode, but the program that implements that algorithm must be written in the vocabulary of a programming language and must conform to the *syntax*, or grammatical rules, of that language. The major portion of this text is concerned with the vocabulary and syntax of the programming language Modula-2. In this section, we introduce some elementary features of this language and give an example of a simple Modula-2 program. These features will be discussed in detail in subsequent chapters.

In the three examples in the preceding section, we used names to identify various quantities. These names are called ***variables***. In the first example, the variables *ServiceCharge*, *UnitCost*, *Installations*, and *YardsOfCable* represented the basic service charge, the cost per foot of cable, the number of installations, and the yards of cable used, respectively. The output in this example was the revenue generated and was represented by the variable *Revenue*. In the second example, the variables *Level1*, *Level2*, *Level3*, *Cutoff*, and *Index* were used, and in the third example, the variables were *LastNumber*, *Counter*, and *Sum*.

In Modula-2, variable names must begin with a letter, which may be followed by any number of letters and digits. This allows us to choose names that suggest what the variable represents, for example, *ServiceCharge*, *UnitCost*, *Installations*, *YardsOfCable*, *Revenue*, *Level1*, *LastNumber*, and *Sum*. *Meaningful variable names should always be used because they make a program easier to read and understand.*

In the examples we have been considering, only nonnegative integer values are used. Modula-2 distinguishes between several types of numeric data, and the types of values that each variable may have must be specified. This is done by placing variable declarations of the form

```
VAR
   list1 : type1;
   list2 : type2;
      :
      :
```