Introduction to Pseudocode

Mikkel Abrahamsen

September 1, 2023

1 Introduction

The book [1] presupposes the reader to have experience with basic pseudocode or programming in standard programming languages. This note explains what pseudocode is and how to read and write it and will hopefully make the book more accessible to students without any programming experience. The note is intended for the students at the 1st year BSc course DMA (*Diskret Matematik og Algoritmer*, or *Discrete Mathematics and Algorithms*) at the Department of Computer Science, University of Copenhagen. The meaning of pseudocode is explained using examples and the exposition is intentionally informal.

2 What is pseudocode?

Pseudocode is a way of presenting algorithms, and its purpose is to describe algorithms in the most efficient and clear way to a human reader. It shares similarity with code written in any structured programming language. When implementing an algorithm in a programming language, we often have to write code to deal with various technicalities that are not conceptually part of the algorithm, but just needed in order to make the code compile and run on the machine. When writing and reading pseudocode, we only focus on the algorithm itself and can forget about such technicalities.

Another difference between pseudocode and programming is that there is no fixed syntax in pseudocode. This means that there is no precise definition of what is correct pseudocode and what is not. However, it should be clear from the context what everything means. Even though there is no fixed syntax, it is good to stick with a standard way of writing pseudocode, such as following the style of the pseudocode in the book [1]. In the following, we will explain basic pseudocode using a number of examples.

3 Assignment of variables

A pseudocode describes the instructions carried out by an algorithm step by step. The algorithm performs the instructions from top to bottom, one line after the other, just as when we're reading an ordinary text. (However, as we will see later, an algorithm can sometimes "jump" from one line to a line far above or below.) Let us now consider the following pseudocode describing a very simple (and not very interesting) algorithm.

```
Example1(n)
1:  a = 5
2:  b = a + n
3:  return b
```

The line with the text "Example1(n)" tells us that the name of the algorithm is Example1. After the name, we have the parenthesis (n), which means that the algorithm takes as input a value called n. This is similar to the notation $f(x)$ encountered in high-school mathematics, where a function f takes a parameter x as input.

We then have three lines of code, enumerated by the numbers 1, 2 and 3, and the algorithm performs these lines in order. The first line "$a = 5$" means that we assign the value 5 to the variable a, so that a now represents the value 5. The variable a did not exist before, but we have now created it and assigned it a value, so it can be used as the algorithm continues to carry out the succeeding instructions. Following the convention from [1], we use a single equality sign = to assign a value to a variable.

The second line "$b = a + n$" means that the value $a + n$ is assigned to the variable b. Here, a has the value 5, as defined in line 1, and n is the variable given as the input to the algorithm. For instance, if the value 7 had been given as the input, then we would assign the value 12 to b.

The third line "return b" means that the output of the algorithm is the variable b (which was defined in line 2). We use the word "return" to specify the result of an algorithm. Once we perform a return statement, the computation of the algorithm is finished and no further instructions are carried out by that particular algorithm. (As we will see in Section 5, it might be the case that our algorithm was called by another algorithm which now receives the returned value and can continue running.)

As we have now seen, the algorithm Example1 takes a number n as input and returns the number $n + 5$.

4 Reassignment of variables

Let us consider the following pseudocode.

```
Example2()
1:  x = 7
2:  x = x + 1
3:  y = x + 2
4:  return y
```

The line "Example2()" means that the name of the algorithm is Example2 and that it does not take any input, since there is nothing written in the parenthesis.

In the first line, we assign the value 7 to the variable x. The instruction of the second line "$x = x + 1$" may at first seem contradictory. It should be understood as follows: We assign the value $x+1$ to the variable x. When we reach this line, x has the value 7, so $x + 1$ is 8. We hence assign the value 8 to x, thus updating the value of x. This use of the single equality sign "=" is different from the usual meaning in mathematics, where = is used to express that what is written to the left of = equals what is written to the right.

In the third line, we assign the value $x + 2$ to the variable y. Since x has the value 8 at this point, we assign the value 10 to y.

In the final line, we return the variable y, which has the value 10. Hence, the result does not depend on any given input, which agrees with the fact that the algorithm does not take any input.

5 Calling other algorithms

A powerful feature in pseudocode is that we can make an algorithm that calls other algorithms. To show what this means, let us consider the following code.

```
Example3(n)
1:  a = Example1(2n)
2:  b = Example2()
3:  return a + b
```

In line 1 and 2, we define variables a and b as the return values of the algorithms Example1 and Example2, described above. Let us see what happens if we call Example3(4). In that case, $2n$ is 8. Hence, we will call Example1(8) at line 1. Our execution of Example3 will "pause" and we wait until we receive the result from Example1, and then we define a to be the returned value. We concluded above that the output of Example1

is 5 added to the input, so Example1 returns 13, and so a is assigned the value 13. We then proceed to line 2, where we call Example2(). Example2 always returns 10, so we assign b the value 10. In line 3, Example3 will then return 23.

One thing that can be confusing here is that we use n both in Example1 and Example3, and the value in Example1 will be twice that in Example3, because we call Example1 with the input $2n$. The algorithms Example1 and Example3 have their own local versions of a variable called n, and they usually don't have the same value. To avoid this confusion, we could also just have chosen a variable with another symbol in one of the algorithms. In general, we want to avoid as much confusion as possible, but here we used n in both algorithms for pedagogical reasons.

The algorithm can be simplified as the following algorithm Example3a, avoiding the introduction of variables a and b:

```
Example3a(n)
1:  return Example1(2n)+Example2()
```

6 If-else

We will now consider an example where the behavior of the algorithm depends on a condition.

```
Example4(n)
1:  a = 5
2:  if n ≥ 4
3:      return a
4:  else
5:      return n
```

In line 2, we test if the variable n, given as the input, is at least 4. In case the condition is true (i.e., we have $n \geq 4$), we perform the code written below which has increased indentation. In this case, the indented code consists of only the line 3, specifying the output of the algorithm. If the condition is false, we instead perform the code written below the associated word "else" at line 4. This code likewise has increased indentation and here consists of only the line 5, specifying the output of the algorithm in this case.

If we call Example4(3), the condition at line 2 is false, so we get to line 5 and output n, which has the value 3. On the other hand, if we call Example4(7), the condition is true, so we get to line 3 and output a, which has the value 5.

7 More complex conditions

We can use the words "and" and "or" to make more complex conditions. Consider the following algorithm.

```
Example5(n)
1:  x = 7
2:  x = x + 1
3:  if n == 3 or n is even
4:      x = x + 1
5:      n = n + 1
6:  x = x + n
7:  return x
```

In line 3, we test if n has the value 3 or n is even. If one or both of these conditions is true, the whole condition is true. Note that we use double equalities "==" to test if two values are identical. A single equality is only used for assigning a value to a variable. If the condition in line 3 is true, we perform the

indented lines 4 and 5, and then continue at line 6. Here, we have no else-statement, so if the condition is false, we just go from line 3 to line 6, jumping over lines 4 and 5.

Another word often used in conditions is "not," which can be used to negate conditions. For instance, we can negate the condition from before by writing "if not ($n == 3$ or n is even)," which in turn is the same as "if $n \neq 3$ and n is odd." This condition is true if and only if the original was false.

8 For loops

A powerful tool in algorithms is to make a set of instructions that are repeated multiple times until a certain event happens. Such a block of instructions that can be repeated is known as a *loop*. Doing the instructions of a loop once is called an *iteration* of the loop.

One of the main variants of loops is the *for loop*. Here, we have a variable, often called i, j or n, that is repeatedly incremented until it reaches a threshold value. We have one iteration of the loop for each of these values, i.e., a certain block of instructions is performed every time the variable has been incremented. Consider the following example.

```
Loop1(n)
1:  for i from 1 to n
2:      print i
3:  print 15
```

The for loop starting at line 1 means that the variable i should run through all values from 1 to n. For each value, the instructions below with increased indentation should be carried out. In this case, line 2 is the only such instruction, so the loop just consists of the lines 1–2.

The instruction "print" in lines 2 and 3 is used when we want the algorithm to output something while it keeps running. If we use "return," the algorithm terminates as soon as it reaches the first return statement, but when using "print," it proceeds to run. We often use "print" for pedagogical purposes, when we want to look into what happens as an algorithm runs.

If we run Loop1(5), the variable n will run through the values $1, 2, 3, 4, 5$, each of which will be printed at line 2, and then the value 15 will be printed at line 3.

9 While loops

Another important variant of loops is a *while loop*, which is more general than the for loop. In a while loop, we first specify a condition that, if true, makes the loop run one iteration more. The condition is evaluated at the beginning of each iteration, and the first time it is false, we "jump out of the loop" and proceed with the instructions below the loop. As an example, consider the following algorithm.

```
Loop2(n)
1:  i = 1
2:  while i ≤ n
3:      print i
4:      i = i + 1
5:  print 15
```

Here, the indented lines 3 and 4 will be performed each time the condition $i \leq n$ is true. Note that the instruction at line 4 just increments the variable i by 1. This algorithm is equivalent to Loop1 from the previous section; the two algorithms do exactly the same. Every for loop can be rewritten as a while loop in a similar way, but while loops are more general, as we can specify more complex conditions defining whether the loop should continue or stop.

10 Level of detail

Because there is no fixed syntax in pseudocode, it is sometimes "too easy" to specify algorithms, in the sense that we end up with something that is not sufficiently detailed. For instance, consider the following algorithm.

```
PrimeSum(n)
1:  s = 0
2:  for i from 1 to n
3:      if i is a prime number
4:          s = s + i
5:  return(s)
```

This algorithm stores the sum of the prime numbers in the range from 1 to n in the variable s (don't worry if you don't know what a prime number is; it is enough to know that some numbers are prime numbers and some are not). This conceptually describes the behavior of the algorithm. However, it is not described how to perform line 3, where we test whether i is a prime number. Hence, the pseudocode is not sufficiently detailed to describe an actual algorithm. This issue can be solved by adding an accompanying text such as "Testing in line 3 whether i is a prime number can be done as described in Chapter X of the book Y."

The appropriate degree of detail also depends on the intended audience. In an introductory algorithms course, we need a high degree of detail, but in a technical report intended for specialists or a research paper, some details can be left out if they are considered common knowledge.

11 Infinite loops

When designing a loop, we have to be careful that the loop actually stops at some point, as it is very easy to create so-called *infinite loops*. Consider the following example.

```
L(n)
1:  i = 5
2:  while i ≠ n
3:      print i
4:      i = i + 1
```

This algorithm sets i to 5 and then keeps adding 1 to i as long as $i \neq n$. If we run L(10), the loop will stop once i reaches the value 10. Hence, the algorithm will print the values $5, 6, 7, 8, 9$ (note that the value 10 is not printed).

However, if we run L(3), it will always hold that $i \neq n$ because i starts with the value 5 and is only incremented. Hence, the algorithm will proceed to print infinitely many integers $5, 6, 7, \ldots$, so we have an infinite loop.

12 Nested vs. sequential loops: The importance of indentation

We have to be careful with the indentation, since it is crucial in defining the behavior of algorithms. Consider the following two algorithms.

```
TwoLoops1(n)
1:  for i from 1 to n
2:      print i
3:  for j from 1 to n
4:      print j
```

```
TwoLoops2(n)
1:  for i from 1 to n
2:      print i
3:      for j from 1 to n
4:          print j
```

Note that the text defining the two algorithms is exactly the same, but the indentation differs. In the algorithm TwoLoops1, we first perform the loop at lines 1–2, and when it is done, we do the loop at lines 3–4. Here the loops are *sequential*, i.e., we first perform one loop and then the other. In algorithm TwoLoops2, the loop at lines 3–4 is indented, so it is part of a bigger loop consisting of all the line 1–4. Hence, the loop at lines 3–4 is performed for every iteration of the big loop. Here, the loops are *nested*, i.e., one loop is part of the other loop. The loop at lines 3–4 is called the *inner* loop, and the complete loop at lines 1–4 is called the *outer* loop.

If we call TwoLoops1 with input 3, the algorithm will print the numbers $1, 2, 3, 1, 2, 3$, whereas TwoLoops2 will print $1, 1, 2, 3, 2, 1, 2, 3, 3, 1, 2, 3$.

In general, TwoLoops1 will print the numbers $1, \ldots, n$ first from line 2 and then from line 4, so $2n$ numbers will be printed in total.

In algorithm TwoLoops2, the numbers $1, \ldots, n$ will be printed from line 2, but after each of these has been printed, all the numbers $1, \ldots, n$ will also be printed from line 4. Note that we will print n numbers from line 2 and $n \cdot n = n^2$ numbers from line 4, so in total, we will print $n^2 + n$ numbers.

The difference between the two algorithms is huge. A very important concept when studying algorithms is that of *running time*, which is a way to quantify how much time it takes to run an algorithm. Inspired by real-world digital computers, we imagine that each instruction takes a little bit of time to carry out, and we then define the running time as the total time, i.e., the sum of the times it takes to perform all instructions. If we imagine that it takes one millisecond to perform one print instruction, then it takes $2n$ milliseconds to perform the print instructions of TwoLoops1, while it takes $n^2 + n$ milliseconds for TwoLoops2. If we use the input $n = 5$, we have $2n = 10$ and $n^2 + n = 30$, so TwoLoops2 uses three times as much time as TwoLoops1, which is something, but perhaps not remarkable. But if we now use $n = 10000$, we have $2n = 20000$ while $n^2 + n = 100010000$, so TwoLoops2 uses $100010000/20000 = 5000.5$ times as much time on print instructions as TwoLoops1. A significant difference!

13 Arrays

An array can be thought of as a (finite) sequence of elements, where each element can be specified by its index/number in the array. We do this using square brackets: If A is an array, we write $A[i]$ to denote element number i.

Consider an array A of length n, i.e., A contains n elements. When $A[1]$ is the first element, we say that A is 1-indexed, and then the elements are $A[1], A[2], \ldots, A[n]$. Sometimes it is more convenient that the first element is indexed by 0, so that the elements are $A[0], A[1], \ldots, A[n-1]$, in which case we say that A is 0-indexed. Each element in an array is also called an *entry* in the array. The elements stored in an array are often integers, but can also be other elementary data types such as floating point numbers (decimal numbers) or objects of more complex types.

We can assign values to the individual entries of an array just as with ordinary variables. For instance, writing $A[i] = x$ assigns the value x to entry $A[i]$.

We will now see two small examples of algorithms taking an array as input. The following algorithm computes the sum of the entries of a 1-indexed array A of length n.

```
Sum(A, n)
1:  s = 0
2:  for i from 1 to n
3:      s = s + A[i]
4:  return s
```

The following algorithm creates an array B which is A in reversed order. Again, n denotes the length of A. Here, the arrays are 0-indexed.

```
Reverse(A, n)
1:  let B be a new array of length n
2:  for i from 0 to n − 1
3:      B[i] = A[n − 1 − i]
4:  return B
```

Some texts assume that the length of an array A is stored as a value $A.length$. Using that notation, we would not need to provide the length as the input n, and could replace all occurrences of n by $A.length$ in the examples above. The present edition of the book [1] does not use the notation $A.length$ (previous editions did), and we try to follow the book's notation. When giving an array as the argument to an algorithm, we therefore need to specify the length of the array as an extra input value (n in the algorithms above).

14 Modification of an array

An array A has a fixed length which is specified when A is defined. This means that A always stores the same number of elements. The only way A can be modified is to update the value of an entry, i.e., writing $A[i] = x$. Note that we can only update one value at a time. Consider for instance a 1-indexed array $A = [1, 2, 4, 5, 7, 9]$ of length 6. We cannot "erase" the element 2 and thus get $[1, 4, 5, 7, 9]$, because we would get an array of length 5 as a result. Using a single instruction, it is also impossible to remove an entry by "shifting" the whole block of the subsequent entries, thus getting the array $B = [1, 4, 5, 7, 9, 9]$. In order to modify A into B, we need to perform each of the instructions $A[2] = A[3]$, $A[3] = A[4]$, $A[4] = A[5]$, $A[5] = A[6]$.

Likewise, we cannot in one operation "insert" a number in an array, making the subsequent numbers move one place to the right. We need to move each of them individually one spot to the right, starting at the right end. For instance, we cannot get from A to the array $C = [1, 2, 3, 4, 5, 7]$ using a single instruction. In order to get to C, we need to perform the operations $A[6] = A[5]$, $A[5] = A[4]$, $A[4] = A[3]$, and finally $A[3] = 3$.

This has to do with the way a real-world computer works. An array is a consecutive block of cells in the computer's memory, and each cell can store a single element. Slightly simplified, we only have an instruction available to change the content of a single cell at a time. This point does perhaps not seem so important when dealing with small arrays of length 6 as in the above examples, but once we have to modify an array of length, say, 1,000,000, it is important to know that shifting all elements one spot to the right requires 1,000,000 individual instructions, each of which takes some amount of time.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022. URL: http://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.