What is Pseudocode?

de Weerdt, Mathijs

DOI
10.4233/uuid:8af9f012-24ff-4cb0-95a3-f740ed52d047

Publication date
2019

Document Version
Final published version

Citation (APA)
de Weerdt, M. (2019). *What is Pseudocode?* Delft University of Technology.
https://doi.org/10.4233/uuid:8af9f012-24ff-4cb0-95a3-f740ed52d047

Delft University of Technology

Report - EEMCS, Algorithmics by M.M. de Weerdt

What is Pseudocode?

Nov. 1, 2019

What is Pseudocode?

Mathijs de Weerdt

November 1, 2019

Abstract

Pseudocode is a clear, compact, unambiguous description of an algorithm or computer program aimed to communicate this to people.

1 Introduction

When presenting and discussing algorithms, we often refer to the "pseudocode" of the algorithm. This is often the most precise description of the algorithm that is used to explain it, and therefore it is very important that we can all read and write pseudocode. However, there is no single language definition of pseudocode. Syntax varies across books (and slides) and scientific papers, sometimes even from the same authors. In this note I explain what pseudocode is, what it is used for, and provide concrete guidelines for writing pseudocode. The resources I used can be found at the end of this note and include early books on programming such as by Aho and Hopcroft (1974); Sahni and Horowitz (1978); Baase (2009).

2 What is the goal of pseudocode?

Pseudocode is a clear, compact, unambiguous description of an algorithm or computer program aimed to communicate this to other *people*. This is thus in contrast to code expressed in a programming language, which is primarily aimed at communicating an algorithm (or more generally, computing instructions) to a compiler for processing by a *machine*. Pseudocode is a compromise between the understandability of English and the precision provided by a programming language, and therefore is more informal and high-level than code, but is more structured and precise than English. Software engineering issues such as data abstraction, modularity and error handling are often ignored in order to convey the essence more concisely (Cormen et al., 2001).

There can be several reasons for communicating an algorithm or program to other people, and one or many may apply in each specific situation:

- for yourself, while you are thinking about how to program, before you start to program,

- to efficiently communicate your main idea for a new algorithm to discuss and improve it together with others (such as in a team of developers),
- to explain how to solve a problem such that others can implement this algorithm,
- to explain the main concept behind a large piece of code, such that others can use or adapt it more effectively (as in reference documentation, or developing software in a team),
- to explain functionality to non-programmers,
- to explain how to efficiently solve a problem, so that others can verify its runtime bound, correctness or optimality, or
- to enable proving a runtime bound, correctness, or optimality of the algorithm yourself (and explain this to others).

The focus of pseudocode is therefore on *readability*; and when writing, the main criterion is whether the intended goal of communicating to the intended audience is reached effectively. In principle, pseudocode is independent of a specific programming language, but, depending on the goal and audience, it could be a bit stylized towards a certain language, i.e., one which most readers are familiar with.

3 Guidelines for writing pseudocode

For writing pseudocode some guidelines and general principles are available. These guidelines assume the reader has some understanding of what an algorithm is (i.e., the steps you must take to achieve a specific goal), how a typical flow looks like (a sequence of statements, iterations, and conditional statements), and which English words are used to represent this.

Before presenting the algorithm, make sure the reader understands the aim of the program. For example, describe the problem by giving a concise description of the expected input and intended output.

Then, for the pseudocode, aim to follow these principles:

- Follow guidelines for good code (e.g. follow variable naming conventions, refrain from using break/goto).
- Follow guidelines for proper English (e.g. check spelling and refrain from using slang).
- Use simple non-technical terms where possible: use the vocabulary of the application domain (and not that of a programming language). For example: use "Extract the next word from the line" instead of "Set word to get next token" (Instructional Software Research and Development Group, 2007).
- Make sure that your pseudocode can be understood to mean exactly one thing (no ambiguity).

3.1 Basic constructs, notation and layout

In more detail, the following guidelines can be followed to meet those principles:

- Express each statement or action on its own line.
- Include programming constructs that are common in most (imperative) programming languages whenever applicable (for, while, repeat/until, switch/case, return, if/then/else, etc.).
- Use indentation of the inner parts of loops, if statements, and methods/function bodies (this is preferred over begin/end for conciseness).
- Let keywords and commands stand out clearly (e.g. using bold face).
- Use comments, indicated using a clear symbol and layout (e.g. '//' and position to the right of the code).
- For assignments (e.g., of 42 to x) you may use either $x \leftarrow 42$ or $x := 42$. These are preferred over $x = 42$, because this can then be reserved for the logical statement of whether x is equal to 42.
- Refrain from using language-specific constructs such as using a dot for a method call to an object (as in `this.example()`).
- When methods are using parameters, assume they are passed on by value (i.e., a copy is assumed to be made).

Generally, aim to be *concise*. So include essential details such as the initialisation of variables and return and/or print statements, but leave out unnecessary or obvious details, such as type declarations if this is clear from the context, the implementation of datastructures (unless you are explaining the inner workings of a datastructure), and a sorting routine (unless you are explaining the inner workings of it).

For example, never present a direct copy of your (Java) code (because that is harder to read for humans and contains unnecessary details). Check for balance. If the pseudocode is hard for a person to read or difficult to translate into working code (or worse yet, both!), then something is wrong with the level of detail you have chosen to use. If you need to explain, include this explanation (Godse and Godse, 2008).

A good test is to transfer your pseudocode into code (without thinking too much about it), test whether it works, correct your code until it does, and then go back to your pseudocode to see if you missed something.

3.2 Advanced constructs

When writing for advanced readers, such as (fellow) scientists in mathematics or computer science, also concepts from basic mathematics may be used inside the pseudocode, such as vectors, sets and operations on these (min, max, union, etc.). The extent, however, to which you can do this depends on the context: 1) who is your audience, and 2)

sort the breakpoints, so that $0 = b_1 \leq b_2 \leq \cdots \leq b_n$
$b_n \leftarrow L$
$S \leftarrow \{0\}$ ▷ Selected breakpoints
$x \leftarrow 0$ ▷ Current breakpoint
while $x \neq b_n$ **do**
    let p be the largest integer such that $b_p \leq x + C$
    if $b_p = x$ **then**
        return No solution
    end if
    $x \leftarrow b_p$
    $S \leftarrow S \cup \{p\}$
end while
return S

Figure 1: An algorithm which greedily selects breakpoints such that two subsequently selected points never are further apart than C. This example has been made using the LaTeX package `algpseudocode` (algorithmicx).

what do you aim to convey: if a factor of n is important for the runtime analysis, do not write finding the maximum x of an unsorted set S as $x \leftarrow \max S$ but as a for-loop:

```
1 max ← −∞
2 for x ∈ S do
3     if x > max then
4         max ← x
5 return max
```

If (the level of) your audience is unknown, make sure that at least your peers can implement the code and obtain the solution without understanding the algorithm.

3.3 Technicalities

Besides explaining an algorithm in plain text or markdown or formatting it with a WYSIWYG editor/presenter tool, there are several LaTeX packages and visualization tools available (Hansen et al., 2002) that automatically generate a consistent representation.

4 Examples

This note is concluded with a number of examples, for different audiences and using slightly different visualizations.

- For a second-year undergraduate course on algorithm design: see for example pseudocode in books like by Cormen et al. (2001) or Kleinberg and Tardos (2005),

if $C = \emptyset$ **then**
    return *True*
else
    Select a clause $c \in C$. Let L_1, L_2, and L_3 be the literals of c.
    Assign the variable in L_1 **such that it is true**; let C_1 be the set of clauses from C updated with this assignment.
    Assign the variable in L_1 **such that it is false, in** L_2 **such that it is true**; let C_2 be the set of clauses updated with these assignments.
    Assign the variable in L_1 **and in** L_2 **such that these are false, and in** L_3 **such that it is true**; let C_3 be the set of clauses updated with these assignments.
    if *a clause in* C_1 *is False* **then**
        $c_1 \leftarrow$ False
    else
        $c_1 \leftarrow$ Solve3SAT(C_1)
    if *a clause in* C_2 *is False* **then**
        $c_2 \leftarrow$ False
    else
        $c_2 \leftarrow$ Solve3SAT(C_2)
    if *a clause in* C_3 *is False* **then**
        $c_3 \leftarrow$ False
    else
        $c_3 \leftarrow$ Solve3SAT(C_3)
    return c_1 *or* c_2 *or* c_3

Algorithm 1: Solve3SAT(C)

Input: Weighted directed graph $G = \langle V, E \rangle$; vertex ordering $d : V \to \{1, \ldots, n\}$
Output: Distance matrix D, or INCONSISTENT if G contains a negative cycle

$G \leftarrow \mathsf{DPC}(G, d)$;
return INCONSISTENT **if** DPC did;
$\forall i, j \in V : D[i][j] \leftarrow \infty$;
$\forall i \in V : D[i][i] \leftarrow 0$;
for $k \leftarrow 1$ **to** n **do**
    forall $j < k$ **such that** $\{j, k\} \in E$ **do**
        forall $i \in \{1, \ldots, k-1\}$ **do**
            $D[i][k] \leftarrow \min\{D[i][k], D[i][j] + w_{j \to k}\}$;
            $D[k][i] \leftarrow \min\{D[k][i], w_{k \to j} + D[j][i]\}$;
return D;

Algorithm 2: Snowball (Planken et al., 2012)

or the example in Figure 1 of greedily selecting a subset of breakpoints S from a list of breakpoints such that any two closest points in S never are further from each other than a distance C. This example has been made using the LaTeX package `algorithmics`.

- For a graduate course on exact algorithms for NP-hard problems, an algorithm for solving the 3-satisfiability problem given a formula in 3-conjunctive normal form, i.e., represented by a set of clauses C with each three literals. This (recursive) pseudocode is formatted using the LaTeX package `algorithm2e`, see Algorithm 1.

- For a scientific paper on a fast polynomial algorithm for all-pair-shortest-paths, please see Algorithm 2. This algorithm computes all pair-wise shortest path lengths D given a graph with edges E and edge-lengths form k to j denoted by $w_{k \to j}$.

- Finally, from a scientific paper on a charging scheduling problem where each task i has a deadline, a value v_i, and some total demand m_i, and there is a supply m_t during each time step t, and the objective is to schedule a subset of these tasks that have the highest total value (de Weerdt et al., 2018). The pseudocode is given using a LaTeX numbered list, i.e., `enumerate`:

  1. Sort all charging task triples on deadline (increasing, with arbitrary tie-breaking).
  2. Let $M_1, M_2, \ldots, M_n$ be the *cumulative supply* at the deadlines of tasks $1, 2, \ldots, n$—that is, $M_i = \sum_{t=1}^{d_i} m_t$—and let $M_0 = 0$.
  3. Run a dynamic program based on the following recursion (where m denotes the remaining cumulative supply available for the first i tasks):

     $$OPT(m, i) = \begin{cases} 0 & \text{if } i = 0 \\ OPT\left(\min\left\{m, M_{i-1}\right\}, i-1\right) & \text{if } m < w_i \\ \max\left\{OPT\left(\min\left\{m, M_{i-1}\right\}, i-1\right), \right. & \\ \quad \left. v_i + OPT\left(\min\left\{m - w_i, M_{i-1}\right\}, i-1\right)\right\} & \text{otherwise} \end{cases}$$

     where the first call is $OPT(M_n, n)$.
  4. Recover the set of tasks that get allocated and match this to resources to find a concrete possible allocation.

References

Aho, A. V. and Hopcroft, J. E. (1974). *The design and analysis of computer algorithms.* Pearson Education India.

Baase, S. (2009). *Computer algorithms: introduction to design and analysis.* Pearson Education India.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press.

de Weerdt, M., Albert, M., Conitzer, V., and van der Linden, K. (2018). Complexity of Scheduling Charging in the Smart Grid: Extended Abstract. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018)*, pages 1924–1926. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

Godse, A. P. and Godse, D. A. (2008). *Fundamentals of Computing and Programming*. Technical Publications Pune.

Hansen, S., Narayanan, N. H., and Hegarty, M. (2002). Designing educationally effective algorithm visualizations. *Journal of Visual Languages and Computing*, 13.

Instructional Software Research and Development Group (2007). *Structured system analysis and design*. Tata McGraw-Hill.

Kleinberg, J. and Tardos, É. (2005). *Algorithm Design*. Pearson.

Planken, L. R., de Weerdt, M. M., and van der Krogt, R. P. J. (2012). Computing All-Pairs Shortest Paths by Leveraging Low Treewidth. *Journal of artificial intelligence research*, 43:353–388.

Sahni, S. and Horowitz, E. (1978). *Fundamentals of computer algorithms*. Computer Science Press.