Pseudocode for Algorithms

When designing an algorithm, it's natural to start with an English description of what it does and how it will function. Pseudocode is meant to bridge the gap between this description and an actual functioning implementation. Because pseudocode is informal, one might assume that any code-like collection of your thoughts will have roughly the same effect as any other. In practice, you'll find that some styles of pseudocode will be far more useful than others.

Pseudocode helps algorithm designers work through the complex details of an algorithm without regard for the minutiae of a language implementation. It can help identify spotty reasoning or missing logic which are easily missed in colloquial descriptions. For the reader, pseudocode clearly conveys how the algorithm works, often best in tandem with a high level description.

"increment every element in the list"

for $i = 1, \ldots, n$
    $A[i] \leftarrow A[i] + 1$

```
for (list<int>::iterator it = myList.begin(); it != myList.end(); ++it) {
    (*it)++;
}
```

Guidelines

The purpose of pseudocode is to present the algorithmic level of program *functionality*. It should be perfectly clear to a reader what your algorithm is doing even though some details may be left out. With rare exception, pseudocode should be *language independent*. Language-specific details should be left out unless they are directly relevant to the algorithm itself.

$A_L, A_R \leftarrow$ `split`(A)
`separately sort` A_L, A_R
$A = A_L \oplus A_R$

You shouldn't be afraid to introduce syntax and notation or to use English in your pseudocode. However, you should clearly define and explain the meaning of anything nonstandard. You can helper functions which may require writing further pseudocode for clarity.

Conventions

Remember, program code is read by a compiler but pseudocode is read by a person. Things like indentation, which don't matter at all for real code, greatly affect the readability of pseudocode. Similarly, syntactic details such as () versus [] referencing, or internal details such as indexing at 0 or 1, are entirely up to you. You should find a style that works for you but what is most important is that whatever you choose remains clear and self-consistent.

Variable names for programs are often long and descriptive for the purpose of maintainability and scope resolution whereas pseudocode can get by with much more succinct names. Pseudocode often shares many of

it's variable naming conventions with math; i, j are counters while m, n represent sizes. Capital letters A, B often mean arrays or matrices while their lower case counterparts a, b, c are used for constants. In general your variables names should be meaningful, without necessarily being long. Here are some conventions you will see in our pseudocode solutions.

- **Arrays**: Typically indexed via either `()` or []. Occasionally we will use $A[i \ldots j]$ to indicate a sub-array consisting of elements $A[i]$ through $A[j]$.
- **Blocking**: Indentation is used (without {}) to indication logical blocks.
- **Assignments**: Almost stated as `=` though sometimes you might see `:=` (defined as) or $\leftarrow$.
- **Conditionals**: These follow the expected layout.

  `if` *condition*`:`
  `else if` *condition*`:`
  `else:`

- **Iteration**: Definite loops (loops where the number of iterations is known) will usually take one of these two forms. In a lot of cases they will function identically but one distinction is the variation on the right doesn't strictly imply any ordering on the loop while the left does.

  `for` i `=` $1 \ldots n$`:` `for all` $i \in \{1, \ldots, n\}$`:`

  Indefinite loops execute until a condition is met. In the instance on the left, typically this means we expect the condition to the checked after the code block rather than before (do/while vs. while).

  `while` *condition*`:` `repeat until` *condition*`:`

Implementation Details

Pseudocode algorithms are written in abstractions. They deal with queues and graphs by their properties rather than their implementation details (whether a queue is implemented with an array or a linked list; whether the graph is stored as an adjacency list or adjacency matrix; etc.). As a rule of thumb, if it does not affect the function of your algorithm, then it should be left out of the pseudocode.

However, while the functionality of your algorithm may be preserved, these details can greatly alter its actual behavior, particularly in terms of run time. When additional specificity is needed, these kinds of things should be written alongside the pseudocode but separate from it. Your pseudocode should remain the cleanest presentation possible of your design. When properly written, an average programmer should be able to more or less "effortlessly" translate it into a working program.

Pitfalls

Part of the reason to write pseudocode is to avoid mistakes easily made with more abstract descriptions. However, it's still easy to make mistakes or miss issues if the writer is not careful. Consider the following two edge traversals of a graph.

`for all` $v \in V$
  `for all` $(v, u) \in E$
	`for all` $v \in V$
	  `for all` $(u, v) \in E$

The left traversal is asking for all the children of each node v and can be performed by a simple walk over the adjacency list. The code on the right, while looking extremely similar, is asking to iterate over the *parents* of node v instead of the children. Though the two loops are almost identical in pseudocode, it's not immediately apparently the one on the right would work. The point is that when writing pseudocode, we have to be conscious of what our programs are actually doing.

These kinds of mistakes are especially easy to make when information is involved which is "obvious" to an omnipotent designer but perhaps nontrivial for the program. For instance, for a node v in a graph, we may be tempted to use something like "the number of nodes in v's SCC" without showing how the program would get this number.

Conclusion

Pseudocode depends on the context. Expectations for large software projects will be necessarily different than for algorithm analysis. The level of detail will certainly depend on the intended audience but one of the advantages of pseudocode is you do not have to be uniform. Details can be focused on the most important and possibly confusing parts of your algorithm while more pedestrian tasks can be relegated to single lines. Writing pseudocode is an art. It is inherently subjective and what is "good" will vary depending on the reader. It is a skill that comes with practice and through exposure to algorithms over time.