Writing Pseudocode

Most problems in CSE 421 are written is pseudocode. Pseudocode is a form of writing that is easier to read than actual code, but less ambiguous than English prose, to communicate a **high-level** description of an algorithm. Your goal is to be specific and concrete enough that

- A competent programmer (say, one of your classmates) could implement the code and get the right output.
- You can analyze the running time and prove the code correct.

BUT still be high-level enough that

- A reader can quickly and easily understand the main ideas behind your algorithm.
- A reader unfamiliar with ⟨ your favorite language ⟩ can still understand your algorithm and implement it in their favorite language.

This document provides tips to write better pseudocode.

1. Use universal code idioms explicitly

Loops, conditionals, and especially recursion are generally clearer in code-like structures, rather than in English. The only exception is extremely simple loops/conditionals.

Examples

In these examples, imagine that you were asked to design an algorithm which prints every integer i up to n (inclusive) for which $1 + \cdots + i$ is even.

Good

```
for $i$ from 1 to $n$ do
    $\texttt{sum} \leftarrow \texttt{sum} + i$
    if sum is even then
        print $i$
```

Good

1. For every i from 1 to n,
    a. Update sum to $\texttt{sum} + i$.
    b. If sum is even, print i.

Ok

```
for i from 1 to n
    sum = sum + i
    if sum is even
        print i
```

Bad

For every integer from 1 to n, add the integer to the running sum. If the sum to that point is even then print that integer.

Discussion

The English is hard to understand—is the "If..." inside the loop or outside? The monospace is also a bit hard to read, since having different fonts for text, variables, and math expressions is very helpful for legibility, but at least the meaning is clear. Note that we are only judging the writing style here—you may find it algorithmically more elegant to solve the problem by printing n whenever n mod 4 is 0 or 3.

2. Be concise, relative to the main topic

You should be aware of what the main idea of the topic of the class is, and highlight that in your algorithm. There is no need to specify excessive implementation details for snippets and subroutines that are not the main focus of the class.

Examples

Good

```
if $n < 100$ then
    triangleCount $\leftarrow 0$
    for each triple of distinct vertices $(u, v, w)$ do
        if $(u, v, w)$ forms a triangle then
            triangleCount $\leftarrow$ triangleCount $+ 1$
    trangleCount $\leftarrow$ triangleCount / 6                ▷ divide by 3! for overcounting
```

Bad

```
if $n < 100$ then
    triangleCount $\leftarrow 0$
    for each vertex $u$ do
        for each vertex $v$ do
            for each vertex $w$ do
                if $u \neq v$ and $v \neq w$ and $u \neq w$ then
                    if $(u, v) \in E$ and $(u, w) \in E$ and $(v, w) \in E$ then
                        triangleCount++
    trangleCount $\leftarrow$ triangleCount / 6                ▷ divide by 3! for overcounting
```

Discussion

In this course, the grader will assume that you are able to expand easy subroutines like this into code, so by writing more, not only are you making more work for yourself, you are making more work for the grader to read, when it is unnecessary.

However, take care to not oversimplify. Whenever there is a subtle point, for example, anywhere you'd need an in-line comment in actual code to clarify something, it may be worth pointing out what the subtlety is.

3. Use variable names

Look at the bad example from Section 1. "The running sum" and "the sum" refer to the same thing, but we used different words for them. In everyday English this is okay, but in a formal setting, this is potentially confusing. In the last line is "that integer" the iteration-counting variable (i) or the running sum (`sum`)?

In general, anytime:

- You need 3 or more words to refer to a thing (like "the vertex that just came off the queue" or "the edge we are currently examining"), OR
- You refer to the same thing with words multiple times ("the integer" and later "that integer")

You probably want a variable name.

4. Use math notation and terminology

Mathematicians have invented many notations that make life easier. Use them.

Examples

Good

1. Let $\bar{x}$ be the mean of $x_1, \dots, x_n$.
2. ...

Good

```
mean ← $\frac{x_1 + \cdots + x_n}{n}$
```

Bad

```
sum ← $0$
for $i$ from $0$ to $n - 1$ do
    sum $+=$ $x[i]$
mean ← ((double) sum)$/n$
```

Discussion

It takes 4 lines of real code to communicate this idea. It takes 1 line or math or English. Use the thing that's faster to read, write, and understand!

5. Make sure your intention is unambiguous

A sentence like:

> For every element, decide using memoization if it's better to keep this element or omit this element by comparing the maximum cost including this element vs maximum cost excluding this element.

is not specific enough for pseudocode in this class. What are those values that I'm comparing? Once I know them what does it mean to be better (bigger? smaller? Closer to a target value?). This would be a good one-sentence of intuition in a comment, but it's not specific enough for a programmer to implement.

6. In general...

- If you could put your pseudocode into a Python interpreter, and it would pretty much compile, you're probably including too much detail.
- If you ever use three or more words to refer to an object (like "the vertex we're processing now" or "the next vertex to come out of the queue") you probably want to make things more code-like (e.g. a variable name like `curr` or notation like `next` ← `queue.removeMin()`)
- Phrases like "repeat this process until..." or "...and so on" or "make a recursive call" are likely to be ambiguous in English and are much better written in code form.
- Practice good coding style in your pseudocode! Break long code into meaningful subroutines, use meaningful variable names, write comments when necessary.

7. A bigger example

Problem: Given an unweighted graph $G = (V, E)$ and vertex $u \in V$, modify BFS to find all vertices v such that there is a path (not necessarily simple) from u to v using exactly 3 edges.

Good

```
Place $u$ on a queue $q$.
$u.\texttt{dist} \leftarrow 0$
Initialize $v.\texttt{reachIn3} \leftarrow \texttt{false}$ for all $v \in V$.
**while** $q$ is not empty **do**
    $\texttt{curr} \leftarrow q.\text{dequeue}()$
    **for** each $\{\texttt{curr}, v\} \in E$ **do**
        $v.\texttt{dist} \leftarrow \texttt{curr.dist} + 1$
        **if** $v.\texttt{dist} = 3$ **then**
            $v.\texttt{reachIn3} \leftarrow \texttt{true}$
        **else if** $v.\texttt{dist} < 3$ **then**
            Place $v$ on the queue $q$.
**return** $\{v \in V \mid v.\texttt{reachIn3} = \texttt{true}\}$
```

Good

1. Initialize a queue q with u.
2. Set u.`dist` to 0.
3. Initialize v.`reachIn3` to `false` for all $v \in V$.
4. While q is not empty,
    a. Let `curr` be the next vertex in the queue.
    b. For each $\{\texttt{curr}, v\} \in E$,
        (i) Set v.`dist` to `curr.dist` $+ 1$.
        (ii) If $v.\texttt{dist} = 3$, set $v.\texttt{reachIn3} = \texttt{true}$.
        (iii) If $v.\texttt{dist} < 3$, place v on the queue q.
5. Return $\{v \in V \mid v.\texttt{reachIn3} = \texttt{true}\}$.

Ok

```
Place u on a queue q.
u.dist = 0
Initialize v.reachIn3 = false for all v in V.
while q is not empty
    curr = q.dequeue()
    for each {curr, v} in E
        v.dist = curr.dist + 1
        if v.dist = 3
            v.reachIn3 = true
        else if v.dist < 3
            Place v on the queue q.
return {v in V | v.reachIn3 = true}
```

Bad (too vague and unclear)

Modify BFS by adding a boolean `reachIn3` and an integer `dist`. Every time you process an edge, update the destination vertex's `dist` to be one more than the other vertex's `dist`. If it's 3, set `reachIn3` to be true. Put the vertex on the queue. Additionally, if a distance ever becomes 4 or more, the algorithm ends.

Bad (too much detail)

Assume the graph is available to us via the adjacency list representation, where `adj`$[v]$ is a list of all vertices adjacent to v.

```
$q \gets$ new Queue()
$q$.enqueue($u$)
$u$.dist $\gets 0$
for each $v \in V$ do
    $v$.reachIn3 $\gets$ false
while $q$.isEmpty() $=$ false do
    curr $\gets q$.dequeue()
    for $i$ from $1$ to adj[curr].len do
        $v \gets$ adj[curr][$i$]
        $v$.dist $\gets$ curr.dist $+ 1$
        if $v$.dist $= 3$ then
            $v$.reachIn3 $\gets$ true
        else if $v$.dist $< 3$ then
            $q$.enqueue($v$)
output $\gets$ []
for each $v \in V$ do
    if $v$.reachIn3 $=$ true then
        output.append($v$)
return output
```

8. Physically writing pseudocode

8.1. General conventions

Here a few general conventions that we like to use. Following them is not mandatory, but we encourage it—it makes the job of grading much easier!

1. Use indentation instead of braces.
2. Prefer to 1-index array-like structures, and prefer loop bounds ("from 1 to n") to be inclusive. Prefer subscripts $(x_1, \ldots, x_n)$ over square brackets $x[1], \ldots, x[n]$.
3. Use math mode for single-character variables and all expressions ($ in LaTeX, equation editor in Google Docs), and monospace font for variables longer than one character (`\texttt{sum}` in LaTeX, `` `sum` `` in Google Docs with Markdown enabled). These changes are not needed when handwriting.
4. Use $\gets$ for variable assignment, reserving $=$ for comparison (`\gets` in LaTeX, set up autocorrect in Google Docs to convert `<-` into $\gets$). The symbol $==$ is not generally used, nor $+=$ (use $a \gets a + 1$ instead) or $<=$ (use $\leq$ instead)

8.2. In LaTeX

For fancy-looking pseudocode, we use the `algpseudocode` package. Use Google to find the documentation and examples. There are other options like `algorithm2e` as well.

For numbered-list style pseudocode, use the `enumerate` environment. You can nest them to create indentation. For all-monospace style pseudocode, use the `verbatim` environment. Note that you cannot use commands or math mode inside a `verbatim` environment. The main advantage of verbatim is that it doesn't "eat" your tabs/spaces and newlines the way normal LaTeXdoes.

8.3. In Google Docs

We recommend the numbered-list or all-monospace styles in Google Docs, which are both easy to use with built in formatting tools.

8.4. When handwriting

When handwriting pseudocode, it is most common to mimic the fancy `algpseudocode`-style formatting. Instead of bold, it is common to underline keywords like for and if.