How to Improve Your Pseudocode

As you get more comfortable with writing pseudocode for algorithms to solve the homework problems, you should start to think about how to improve your pseudocode. After all, it isn't enough to write a correct algorithm if no one else (in this case, the TAs grading your homeworks) can understand it! If you have a clean solution that is easy to follow, not only does it make it easier for someone else to read it, but it makes it easier for you to hand-simulate your algorithm and/or prove that it is correct. This handout has 4 examples of algorithms, ranging from very confusing to totally awesome, that solve the same problem: eliminating duplicate edges in an array of edges. The purpose of this handout is to provide some tips on how to write good pseudocode.

1 Example 1

When coming up with an algorithm, it can be helpful to think in Java since you are so used to writing code to tackle the problem you are given. However, writing pseudocode that contains a lot of language-specific syntax can often obfuscate what your algorithm actually does. A good goal to keep in mind is to write an algorithm that could be implemented in *any* language!

It is also important to use data structures and their methods at a high level. As long as you are not using some super-special implementation of a data structure, you can assume that you can use any of the data structures about which you have learned in CS16.

Here is a solution that is pretty difficult to read. It does some funky things with appending to an array and might not even work depending on the hash function used!

```
algorithm makeChains
 input: array edges[0...n-1] of n directed edges
 output: array of edges without duplicates

List[] hashTable = new List[n];

h = hashfunction;        //see hashfunction below

for (int i = 0; i < n; i++) { //insert all the edges
    index = h(edges[i].start());
    if (hashTable[index] == null) {
        hashTable[index] = new List<Edge>();
    }
    hashTable[index].append(edges[i]);
}

for (int i = 0; i < n; i++) { //remove edges whose reverse is in the table
    index = h(reverse(edges[i]).start());
    if (hashTable[index].contains(reverse(edges[i]))) {
        hashTable[index].remove(reverse(edges[i]));
        hashTable[h(edges[i]).start()].remove(edges[i]);
    }
}

edge[] results = new edge[hashTable.size()];

for (int i = 0; i < n; i++) { //make an array of results to return
    if (hashTable[i] != null) {
        for (int j = 0; j < hashTable[i].size(); j++) {
            results.append(hashTable[i].getElement(j));
        }
    }
}

return results;


algorithm hashFunction
//omitted some hash function defined here
```

Let's go through how the algorithm works, and then discuss ways to improve it.

First, make an array of lists and use a hash function (defined at the end) to implement a hash table. Next, insert every input edge into the hash table using the hash function to index into the array. If there is no list at the index, make a new list, and append the edge to the list. Then, for every input edge, find the index of the reversed edge, and check if the list at that index actually contains the reversed edge. If it does, remove the reversed edge and the original edge from the hash table. Finally, make an array of results, append all the remaining edges in the hash table to the results array, and return the results.

Even though it is still a bit confusing, the paragraph above does a much better job of explaining the algorithm than the page of code before it! How can we make our algorithm easier to read?

First, we can incorporate some code-to-pseudocode conversions [1] to make our algorithm less language-specific. We can change all the for loops of this form:

```
for (int i = 0; i < n; i++) {
    //do something
}
```

to this:

```
for i from 1 to n-1:
    //do something
```

or alternatively:

```
for i <- 1 to n-1:
    //do something
```

The assignment arrow above is equivalent to using = as an assignment operator. Some people prefer to use ← for assignments, since it makes it clear which variable is being assigned what. Remember those bugs you used to get when you accidentally switched which variable was on which side of the assignment operator? If Java used arrows for assignment, I bet that wouldn't have been a problem! E.g.

```
currMax <- values[i]
```

Also, in pseudocode it is acceptable to use a single = instead of double == in equality testing, so using an arrow for the assignment operator makes it extra clear what you are doing. However, in most cases it won't make a big difference, so this choice is up to you – as long as you're consistent.

Another syntax-specific convention is declaring the types of all variables. In pseudocode, it usually is clear enough from context what the type of a variable is. For example:

```
H <- new hash table
```

[1] You can find more examples on the course website under `Docs>>CS16 Pseudocode Standards`

is clear enough without declaring H as type hash table. Notice how we also said "hash table" instead of "HashTable," since we are using the abstract data structure, rather than a specific implementation of a hash table (e.g. Java's `java.util.Hashtable`). We can also get rid of the implementation details of the hash table and just use one, as long as we specify that the hash table can store multiple values for keys (to eliminate any confusion about how the hash table handles collisions). In our next example, we will show how to create a hash table and use the standard hash table methods without knowing how they are implemented. We can also get rid of semi-colons.

2 Example 2

Let's incorporate the basic improvements from the previous example and rewrite our algorithm.

```
algorithm makeChains
 input: array of n edges
 output: array of edges without duplicates

H <- new hash table     //can contain multiple values for a key

for i <- 0 to n - 1:
    H.insert(edges[i])

for i <- 0 to n - 1:
    r <- reverse(edges[i])
    if (H.contains(r)):
        H.remove(r)
        H.remove(edges[i])

results <- new Array

for i <- 0 to n - 1:
    if (H.contains(edges[i])):
        results.append(edges[i])

return results
```

So much better! With those few, simple changes, the algorithm becomes so much more intelligible. This is the *minimum* quality of a solution that we expect on homeworks.

We already made the improvement of using `for i from 1 to n-1` to index into an array. However, since we don't really care what the index of each element is and we just want the *element*, instead of accessing the elements in the input array by indexing into the array with a for loop, it is much cleaner to use a for-each loop as below:

```
for each element e in the array E:
    //do something
```

For this problem, we specified the input as "an array edges[0 .. n-1] of n directed edges," but for many problems in future homeworks, your input might be a graph $G =< V, E >$, or something similarly general. In that case, your algorithm should be as general as possible to allow for any implementation of the input that is given. Let's change our algorithm to take an input E, a set of edges. Then instead of iterating over the indices in the input array, we can just use a for-each loop as such:

```
for each edge e in E:
    //do something
```

We can also make our pseudocode flow more naturally if we make it sound like an explanation, rather than code. For example, instead of:

```
for each edge e in E:
    H.insert(e)
```

we can write

```
for each edge e in E:
    insert e into H
```

There are times when it is more clear to use the "code-y" syntax of calling a method on an object, so this is a stylistic choice that you can make. Sometimes it is reasonable to assume you can summarize code, and sometimes you should write it out in your pseudocode. If you are doing something really trivial like iterating over a hash table to make a list of edges to return, you can just summarize it, but if you're not sure, you can always post to the Google group and ask.

3 Example 3

With the new changes, our algorithm is getting even better...

```
algorithm makeChains(E)
 input: E edges
 output: edges without duplicates

H <- new hash table

for each edge e in E:
    insert e into H

for each edge e in E:
    if H contains reverse(e):
        remove reverse(e) from H
        remove e from H

return edges in H
```

Up to now, most of our changes have been stylistic changes to how we write the pseudocode in order to make the algorithm easier to read, and at this point I would say that the algorithm is about as clear as it can get. However, we can still improve the steps of the algorithm itself. You might notice that we insert reverse edges just to remove the edge and the reverse later. What if we never inserted these edges at all? We can do this by checking whether the reverse edge is in the hash table *before* we insert an edge. Although this doesn't change the big-O runtime, it could improve the runtime by up to a factor of 2, which in real life is a huge improvement.

The final algorithm is on the next page.

4 Example 4

Here it is!

```
algorithm makeChains(E)
 input: E edges
 output: edges without duplicates

H <- new Hashtable

for each edge e in E:
    if H contains reverse(e):
        remove reverse(e) from H
    otherwise insert e into H

return edges in H
```

Six lines! Although line-count is not a perfect gauge of the quality of an answer, a simpler, correct answer is preferable to a more complicated, correct answer.