Due: Thu 25 April, 23:59

Submission is through `give` and should consist of:

1. A single pdf file with typeset answers, maximum size 4Mb. Prose should be typed, not handwritten. Use of LaTeX is strongly encouraged, but not required.
2. A file `partner.txt` with a single line on the format. `z<digits>` . This must be the zID of your group partner. For individual submissions, write your own zID.

Submit your work using the web interface linked on the course website, or by running the following on a CSE machine

```
give cs2111 assn3 assn3.pdf partner.txt
```

This assignment is to be done in pairs. Individual submissions are discouraged, but will be accepted. Only one member of the pair should submit. Work out who will submit ahead of time—duplicate submissions are extremely annoying to sort out.

Late submission is allowed up to 5 days (120 hours) after the deadline. A late penalty of 5% per day will be deducted from your total mark.

Discussion of assignment material with others is permitted, but you may not exchange or view each others' (partial) solutions. The work submitted *must* be your own, in line with the University's plagiarism policy. Note in particular that attempting to pass off AI writing (e.g. ChatGPT output) as your own still counts as plagiarism.

Background This assignment is an open-ended research task where you apply Hoare logic to an algorithm of your choice.

This means you should start by choosing an algorithm! It's your choice, but keep the following in mind:

- The algorithm should contain at least one loop.
- You need to find an algorithm that makes the tasks below feasible in the given timeframe. If the algorithm is longer than 10-20 lines or so, you should probably choose something simpler.
- You need an algorithm that you can meaningfully specify with a Hoare triple, and reason about with Hoare logic.

What does the last bullet point mean exactly? I'll give some examples. You want to avoid:

- Algorithms that are probabilistic, in the sense that they aren't guaranteed to always give a correct answer. An example is the Miller-Rabin primality test.
- Algorithms that give approximate answers only, such as numerical algorithms.

- Concurrent or parallel algorithms.
- Algorithms that have fuzzy specifications, where it's difficult to characterize precisely what it means for the output to be correct. Examples include most algorithms in machine learning and image processing.
- Algorithms that require non-compositional control flow operators, such as `goto`, `break` or `continue`. (But you can probably find an alternative way to formulate the algorithm that doesn't require these).

A good place to start looking for algorithms is the table of contents of your favourite algorithms textbook [CLRS09, Knu68, Knu69, Knu73, KT06], or the Wikipedia list of algorithms.[1]

Once you've found an algorithm, do **ask your tutor, or Johannes, for input on your choice**! We are happy to offer advice on your choice.

[1] `https://en.wikipedia.org/wiki/List_of_algorithms`

Problem 1 (30 marks)

Present your algorithm, giving at least the following information:

1. The name of the algorithm
2. The origin of the algorithm. Is it known Who invented it? Cite the original paper that introduced the algorithm, if possible.[2] Also cite any other sources you've consulted in preparing this assignment.
3. The purpose of the algorithm. What problem does the algorithm solve? What are the requirements that the algorithm is designed to meet?[3]
4. Give pseudocode for the algorithm.
5. A brief explanation of the pseudocode, to help the reader understand the algorithm.

Problem 2 (70 marks)

1. Give a vocabulary containing the necessary features to express your algorithm in $\mathcal{L}$, explaining the intended meaning of each symbol when it's not obvious.

   For algorithms involving numbers, you can use the lectures for inspiration. For algorithms involving things beyond that, such as arrays, trees, or pointers, there will be design decisions for you to make. (10 marks)

2. Express your algorithm in $\mathcal{L}$ using this vocabulary. (You may have already done so in Problem 1).

   Where possible, use the existing primitives in $\mathcal{L}$ (possibly including derived operators) to express the algorithm. This may involve some translation, such as by replacing `for` loops with `while` loops.

   If your algorithm requires operators that are not expressible in $\mathcal{L}$, you may add them to $\mathcal{L}$ provided you can formulate a reasonable Hoare rule for it. If you're not sure how, ask us for advice. For example, if your algorithm requires generating random integers in an interval, something like this would do:

$$\overline{\left\{\forall I \in \mathbb{Z}.\ e \leq I \leq e' \to \varphi[I/x]\right\}\ x := \mathsf{Math.random}(e, e')\ \{\varphi\}}$$

   (10 marks)

3. Based on the requirements identified in Problem 1, write a formal specification for the algorithm, in the form of a Hoare triple. Explain any requirements that your Hoare triple does not capture, and why. (10 marks)

[2] For example, absent a groundbreaking archeological discovery, you may have difficulty finding the primary source for Eratosthenes' sieve.

[3] Requirements here should be construed broadly. This includes functional requirements of the kind one can fit in Hoare triples ("the output should be sorted"), as well as non-functional requirements ("the algorithm should run in $O(n^2)$ time", "the algorithm should run in-place", "the algorithm should be easier to understand than the competition", ...)

4. Annotate your program with appropriate loop invariants for any loops present in your algorithm. (10 marks)

5. Show that your annotations are correct, in the sense that they follow from the precondition, establish the postcondition, and are re-established by every loop iteration.

   This can take the form of an informal argument (for partial marks), or a complete annotation of the program (for full marks). (20 marks)

6. If your algorithm is guaranteed to terminate, find a loop variant (aka measure) or a well-founded order for every loop in your program. If it is not, explain why. (10 marks)

The mark distribution and criteria for the above sub-questions is indicative only, and may be adjusted depending on your choice of algorithm. In particular:

- If you chose a more difficult algorithm, expectations on completeness and formality decrease somewhat. (Note that a shorter algorithm is not necessarily easier to verify than a long one.)
- A failure to solve a problems may still yield marks. For example, a compelling description of the difficulties you encountered when attempting to formulate an invariant.

Assorted advice

Arrays

Many algorithms require arrays. Adding arrays to $\mathcal{L}$ is just a matter of adding array indexing etc. to the vocabulary. To save you some time though, here is a tempting, but unsound, way to formalise array updates:

$$\frac{}{\{\varphi[e/x[i]]\}\ x[i] := e\ \{\varphi\}}\ \textbf{UNSOUND RULE DO NOT USE}$$

The issue here is that φ can make claims about $x[i]$ without mentioning $x[i]$ directly. For example, the following is of course not a valid Hoare triple: $\{x \text{ is sorted}\}\ x[0] := 17\ \{x \text{ is sorted}\}$ — but it is provable with the incorrect rule above.

The solution is to model array updates as updates to the array as a whole. That is, model $x[i] := e$ as $x := x[i \mapsto e]$, where the RHS reads "the array x, but with index i updated to hold the value e. With that change, array updates can be handled by the standard assignment rule.

This illustrates the more general point that the Hoare logic assignment rule is specifically designed for situations where the LHS is a variable, not a compound expression.

Pointers

Pointers are tricky, and this assignment is no exception.

If you're feeling adventurous enough to do a program with non-trivial use of pointers, you'll need a slightly different notion of state than we've used in the course so far. We've mostly been content to let

states be mappings from variable names to values; for pointer programs this will fail to capture the effects of aliasing, where updating the value pointed to by x may change the value pointed to by y indirectly.

My recommendation is to let the state have two components: the first is the standard mapping from variable names to values, but with the minor twist that values can be memory addresses. The second component is the *memory*, which is a mapping from memory addresses to values.

Scope

If your algorithm turns out to be too complicated to verify, one way to reduce the scope is to axiomatise auxiliary functions. By this, I mean specifying auxiliary functions with appropriate Hoare triples, and then using those Hoare triples (without proof) in the correctness proof for the main function.

As an example, if you're verifying merge sort, you might axiomatise an auxiliary `merge` function.

This may feel a bit like cheating, but it's done all the time in real-world verification projects: somewhere, you need to draw the boundary between what's verified and what isn't. Even if some code is left out of scope, the verification is still valuable: it tells you exactly where you need to beware of bugs.

References

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[KT06] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.