CSE103 Introduction to Algorithms

Practice Exam

You have 2 hours.
You may consult any document available on the CSE103 page on Moodle. For this, you may use an Internet-connected device with the biggest screen possible (preferably, a laptop computer).
Nothing else is allowed. If you are caught looking up any document not on Moodle, using email or any messaging application of any kind, you will immediately get an F on this exam.
Each question is independent of the others and may be answered in any order.
The maximum score for the exam is 50 points. The number of points awarded by fully answering each question is not fixed yet, it will be written next to the question/subquestion itself for the exam.

Exercise 1: Hoare triples

Which of the following Hoare triples is valid? Justify your answer.

1. $\{x = y\}$ `x = x - 1` $\{x < y\}$
2. $\{$`True`$\}$ `x = 0` $\{x = 42\}$
3. $\{$`True`$\}$
```
if y == 42:
    x = 0
else:
    while True:
        pass
```
$\{y = 42\}$
4. $\{$`True`$\}$
```
if y == 42:
    x = 1
else:
    x = 2 * z * z
```
$\{x \text{ is even } \Rightarrow y \neq 42\}$
5. $\{$`False`$\}$ `<whatever>` $\{$`False`$\}$
6. $\{y = 0\}$
```
while x < 42:
    x = x + 1
```
$\{x = 42\}$

Solution.

1. Valid.
2. Invalid. We definitely know that $x = 0$ after the assignment.
3. Valid: the precondition `True` gives us zero information, but this does not prevent us from saying that either $y = 42$ or $y \neq 42$; in the first case, we follow the `if` branch and do not change y, so it will still be equal to 42 when we terminate; in the second case, we follow the `else` branch and enter an infinite loop, so applying the reasoning that "if an infinite loop terminates, then anything can happen", we say that y is equal to anything we want, so why not 42. So, whatever branch we follow, if the code terminates we will have $y = 42$.

4. Valid. Let us state explicitly the meaning of the triple: "If the code terminates, then if `x` is even, then `y` $\neq 42$". This may be reformulated as "If the code terminates and `x` is even, then `y` $\neq 42$" (remember that $P \Rightarrow (Q \Rightarrow R)$ is logically equivalent to $(P \wedge Q) \Rightarrow R$). This is definitely true: this code always terminates after executing one of the two branches; if we have the additional information that, after terminating, `x` is even, then it must be the case that `y` $\neq 42$, for otherwise we would have entered the `if` branch and `x` would *not* be even (it would be equal to 1).

5. Valid: a triple of the form $\{$`False`$\}$ `<whatever>` $\{Q\}$ is *always* valid, no matter what `<whatever>` and Q are. Indeed, the meaning of this triple starts with "If `False` is true, then…", but `False` can never be true, so we may put anything in the postcondition (including `False`), for any code whatsoever.

6. Invalid. The precondition tells us that `y` $= 0$ but says nothing of `x`. In particular, in may be that `x` > 42, in which case we do not enter the loop and we terminate with `x` $\neq 42$.

Exercise 2: Preconditions and postconditions

Complete the following Hoare triples so that they are valid. Try to make the pre/post-conditions as precise as possible. In any case, you are not allowed to use `False` (or something logically equivalent) as a precondition, unless it is the only possible choice. Similarly, you are not allowed to use `True` as a postcondition unless it is the only possible choice.

1. $\{???\}$ `x = x - 1` $\{x < 0\}$

2. $\{???\}$
```
if x == 3:
    y = 4
else:
    y = 2
```
$\{$`y` is a perfect square$\}$

3. $\{???\}$
```
while i < 10:
    x = x + 1
    i = i + 1
```
$\{x = i\}$

4. $\{x = 3\}$ `x = x - y` $\{???\}$

5. $\{x = 1\}$
```
if x % 2 == 0:
    y = 0
else:
    y = 47
```
$\{???\}$

6. $\{x = 6\}$
```
if y < 0:
    y = -y
else:
    y = y + 1
x = x + y
```
$\{???\}$

Solution. Before giving the solution, let us explain a bit the request that pre/postconditions be "as precise as possible".

When you are asked to complete a triple of the form $\{???\}$ `<something>` $\{Q\}$, the question you need to ask yourself is: what is *the minimum amount of information* I need to know before executing `<something>` such that, when `<something>` terminates, I can be sure that Q is true? The closer you get to the minimum, the more points you obtain.

Similarly, when you are asked to complete a triple of the form $\{P\}$ `<something>` $\{???\}$, the question you need to ask yourself is: if I know that P holds before `<something>` is executed, and if `<something>` terminates, what is the *maximum amount of information* I may infer? The closer you get to the maximum, the more points you obtain.

In this perspective, `True` is the minimum amount of information possible: it tells us nothing at all, because it is always true! On the other hand, `False` is the maximum amount of information possible, so much information that it includes P and $\neg P$ for any statement P. Now you understand why `False` is discouraged as a precondition: we are trying to minimize information, and `False` is usually so far from the minimum that it is likely to give you zero points (not always though: there are some cases in which `False` is the only possible precondition for making a triple valid). As observed in Exercise 1.5, using `False` as a precondition is like a "wildcard": it always works, so the exercise would become trivial if we admitted it without warning.

Dually, `True` is discouraged as a postcondition: a triple of the form $\{P\}$ `<whatever>` $\{$`True`$\}$ is *always* valid, that is, `True` is the "postcondition wildcard". Since, in this case, we are trying to maximize the information, usually `True` will give you zero points (again, not always: in some cases, we really have no information at the end and `True` is the only thing we may infer).

1. $\texttt{x} \leq 0$. Other non-trivial preconditions that also work, but which are *not* minimal:
   - $\texttt{x} = 0$: it contains more information than $\texttt{x} \leq 0$, because it excludes infinitely many values of `x` which are still compatible with the postcondition.
   - $\texttt{x} < 0$: it excludes the possibility that $\texttt{x} = 0$, which is compatible with the postcondition.

2. $\texttt{x} = 3$. The value of `y` is always affected by the code, and it is a perfect square only when the `if` branch is executed, which only happens if $\texttt{x} = 3$. We may add other information to the precondition (for example, $\texttt{x} = 3 \wedge$ `y` is a perfect square), but this is useless and makes the precondition not minimal.

3. $\texttt{x} = \texttt{i}$. Here, we see that the values of `x` and `i` are changed in the same way by the execution of the body of the loop, so the only way of making them equal at the end, is to have them equal at the beginning. Adding extra information about the value of `i` or `x` (like $\texttt{x} = \texttt{i} \wedge \texttt{i} = 0$ or something like that) is ok but useless (not minimal).

4. $\texttt{x} = 3 - \texttt{y}$. This is the maximum information we may obtain here. Indeed, there is no information about `y` in the precondition, so what we know about `x` does not allow us to predict its value after the execution of the assignment: `x` may become any value whatsoever (remember that variables are of type `int`, that is, they may be negative). However, we do know that, when we add `y` to it, we will obtain 3 (in fact, we could also write $\texttt{x} + \texttt{y} = 3$, which is of course equivalent).

5. $\texttt{x} = 1 \wedge \texttt{y} = 47$. This is because, from the precondition, we know that the `else` branch will be executed. The postcondition $\texttt{y} = 47$ is correct too, but not maximal: the value of `x` is not altered by the execution, so we still know what it is and we may add that information. On the other hand, writing something like $\texttt{y} = 0 \vee \texttt{y} = 47$, albeit certainly correct (one of the two branches must be executed!), would give you zero points here because, in this particular case, we *know* which branch will be executed.

6. $\texttt{y} > 0 \wedge \texttt{x} = \texttt{y} + 6$. We know nothing about the initial value of `y`, but we do know that, if $\texttt{y} < 0$, then $\texttt{y} > 0$ at the end (because we change the sign); and, if $\texttt{y} \geq 0$, then still $\texttt{y} > 0$ at the end (because we add 1 to a non-negative number). Therefore, no matter what the initial value of `y` is, we always end up with `y` containing a strictly positive quantity, which is added to `x`, whose

initial value was 6. Writing $y > 0 \wedge x > 0$, $x = y + 6$ or even just $x > 0$ are all correct but not maximal (and progressively worse).

Exercise 3: Proving correctness of a program

Given a non-negative integer `n`, the following code obviously writes in `s` the number `2*n`. Prove that this is the case by completing the assertions in the code so that they obey the rules of Floyd-Hoare logic. (NB: a line of the form `#! ???` may be replaced by more than one assertion, using the consequence rule).

```
#! n ≥ 0
s = 0
#! ???
i = 0
#! s = 2i ∧ i ≤ n
while i < n:
    #! ???
    s = s + 2
    #! ???
    i = i + 1
    #! ???
#! ???
#! s = 2n
```

Solution. Here is the program decorated with assertions, each following from the previous via the rules of Floyd-Hoare logic (two contiguous assertions follow one from the other via a consequence rule, that is, the top assertion logically implies the bottom one):

```
#! n ≥ 0
#! 0 = 2·0 ∧ 0 ≤ n
s = 0
#! s = 2·0 ∧ 0 ≤ n
i = 0
#! s = 2i ∧ i ≤ n
while i < n:
    #! s = 2i ∧ i ≤ n ∧ i < n
    #! s + 2 = 2(i + 1) ∧ i + 1 ≤ n
    s = s + 2
    #! s = 2(i + 1) ∧ i + 1 ≤ n
    i = i + 1
    #! s = 2i ∧ i ≤ n
#! s = 2i ∧ i ≤ n ∧ i ≥ n
#! s = 2n
```

Exercise 4: Three functions on graphs

Let us introduce the following notions:

- A *clique* of a directed graph G is a set of nodes C of G such that, for all $i, j \in C$, there is an edge $i \to j$ in G.
- An *independent set* of a directed graph G is a set of nodes I such that, for all $i, j \in I$, there is no edge $i \to j$ in G.
- A *Hamiltonian cycle* of a graph G is a cycle that goes through each node of G exactly once.

Consider the following Python definitions:

```
def funOne(G, l):
    # We suppose that G is a square matrix with entries in {0,1}
    # and that l is a list whose values are in {0,1,...,len(G)-1}
    # Remember that, in Python, True == 1 and False == 0
    ok = True
    i = 0
    while ok and i < len(l):
        j = 0
        while ok and j < len(l):
            ok = G[l[i]][l[j]]
            j += 1
        i += 1
    return ok

def aux(l):
    r = [0] * len(l)
    ok = True
    i = 0
    while ok and i < len(l):
        if l[i] < len(l):
            r[l[i]] += 1
            i += 1
        else:
            ok = False
    i = 0
    while ok and i < len(r):
        ok = r[i] == 1
        i += 1
    return ok

def funTwo(G, l):
    # We suppose that G is a square matrix with entries in {0,1}
    # Remember that, in Python, True == 1 and False == 0
    ok = len(l) == len(G) and aux(l)
    i = 0
    while ok and i < len(l) - 1:
        ok = G[l[i]][l[i+1]]
        i += 1
    return ok and G[l[len(l)-1]][l[0]]

def funThree(G, l):
    # We suppose that G is a square matrix with entries in {0,1}
    # and that l is a list whose values are in {0,1,...,len(G)-1}
    # Remember that, in Python, True == 1 and False == 0
    H = [0] * len(G)
    for i in range(len(G)):
        H[i] = [0] * len(G)
        for j in range(len(G)):
            H[i][j] = not G[i][j]
    return funOne(H, l)
```

1. What do the above functions do?

2. What are the asymptotic (big O) complexities of `funOne` and `funTwo` as functions of the length of the input list `l`?

3. Describe an algorithm for checking whether a directed graph G with n nodes has a clique of size $\log n$ (no need to code it in Python). What is its asymptotic (big O) complexity as a function of n?

4. Describe an algorithm for checking whether a directed graph G has a Hamiltonian cycle (no need to code it in Python). What is its asymptotic (big O) complexity as a function of the number of nodes of G?

Solution.

1. The function `funOne(G,l)` checks whether the nodes listed in `l` are a clique of the graph (represented by) `G`, and returns `True` if it is the case. It does so by simply checking, for each p, q in the list, whether there is an edge $p \to q$.

   The function `funTwo(G,l)` returns `True` if the list `l` describes a Hamiltonian cycle in `G`. It does so by first checking that `l` is a permutation of `[0,1,2,...,n-1]` (where `n` is the number of nodes of `G`) and then it checks that, for each node in `l`, there is an edge from it to the next node in the list, as well as an edge from the last node to the first (because we are looking for a cycle).

   The function `funThree(G,l)` returns `True` if the nodes listed in `l` are an independent set of `G`. It does so by using the *dual graph* of `G`, that is, the graph $\overline{\texttt{G}}$ having the same nodes as `G` and such that there is an edge `i` $\to$ `j` iff there is no edge `i` $\to$ `j` in `G`. It is an easy observation that an independent set of `G` is exactly the same as a clique of $\overline{\texttt{G}}$, so `funThree` computes $\overline{\texttt{G}}$ and then calls `funOne`.

2. If k is the length of `l`, the complexity of `funOne` is $O(k^2)$, and the complexity of `funTwo` is $O(k)$.

3. The obvious algorithm is: call `funOne(G,l)` on all possible lists of nodes `l` of length $\log n$. (We may restrict on repetition-free lists, but that does not change the asymptotic complexity). There are $n^{\log n}$ such lists, so the total complexity is $O(n^{\log n} \log^2 n)$.

   Notice that this is superpolynomial. It is widely believed that there exists no polynomial-time algorithm for solving this problem.

4. The obvious algorithm is: call `funTwo(G,l)` on all possible permutations of $[0, 1, \ldots, n-1]$, where n is the number of nodes of `G`, which has overall complexity $O(n \cdot n!) = O(n2^{n \log n})$.

   Notice that this algorithm is exponential. As for the clique problem, it is widely believed that there is no polynomial-time algorithm telling whether a graph has a Hamiltonian cycle.