Algorithms for Data Science
Cheatsheet for Exercises 1

Loop invariants: To prove that an algorithm is correct, one can often use a *loop invariant*. Formulating an invariant also helps to find mistakes!
An invariant should be

- true, and
- helpful.

We look now at these two points in detail.

A loop invariant is defined with respect to a loop in the program. It is any logical statement on how the variables are related to each other. The statement must be true for every iteration of the loop (therefore the name "invariant"—something that does not change). Normally, the invariant concerns the moment of the iteration, when the condition in the loop head is checked.

As an example, consider the following algorithm.

```
i = 0
j = 0
k = 0
n = 100
while i ≠ n do
    i = i + 1
    j = j + 2
    k = k + 3
```

A possible invariant is "We have $2i = j$."

To prove that an invariant is always true, it suffices

1. to show that it is true in the beginning of the first loop iteration, and
2. to show that, if it is true at the beginning of a loop iteration, it is also true at the end of that iteration (which implies that it is true at the beginning of the next iteration).

In our example, we have $2i = 0 = j$ in the beginning of the first while loop iteration. Thus, the first step holds. Regarding the second step of our proof, suppose that the invariant is true for some value $i = i'$ and $j = j'$, that is, $2i' = j'$ by our assumption. During the loop iteration, we increase i by 1 and j by 2. Thus, at the end of that iteration, we have $i = i' + 1$ and $j = j' + 2$. Hence, if we check the invariant statement, we see that $2i = 2(i' + 1) = 2i' + 2 = j' + 2$ (recall that $2i' = j'$). This means that the invariant holds also at the end of that statement. Thus, it is true in the beginning of every iteration.

So, are invariants always helpful? The answer is: no, sometimes they might not be helpful! For instance, if we wanted to prove that $k = 3n$ at the end of our algorithm above, our invariant will be of no help as it makes no statements about k. However, if we wanted to prove that $j = 2n$, our invariant is perfect: since our invariant is always true, it is also true at the moment when we leave the while loop. On the other hand, we know that the while loop is only left if the loop condition $i \neq n$ is violated, that is, when $i = n$. Thus, at the end, we have $j = 2i = 2n$.
Note that for the same algorithm, there are infinitely many correct invariant statements. But we are interested only in one that helps us to show that our algorithm is correct. Also note that an invariant alone is not enough to prove the correctness of an algorithm. We also need to show that all the loops in the algorithm stop at some point, that is, that the algorithm terminates. In our example, we could argue that the difference between n and i drops by exactly 1 in each iteration, and therefore will be 0 after a finite number of iterations.

Induction: For completeness, let us note that invariants are strongly related to the concept of "proof by induction". In fact, proofs by invariants are just a special case. In a proof by induction, we want to show that a logical statement depending on some variable i is true for all (infinitely many) values of i (from some given range). The statement can of course depend on more than one variable.
As an example, consider the statement: "We have $2^0 + \cdots + 2^i = 2^{i+1} - 1$." As the range for i, let us choose $i \geq 0$. To prove the correctness,

1. we show that the statement is true for the smallest i (in our example: $i = 0$ yields $2^0 = 1$ and $2^{0+1} - 1 = 2 - 1 = 1$), and

2. by using the assumption that it is true for some i, we show that the statement is also true for $i+1$. In our example: $2^0 + \cdots + 2^{i+1} = (2^0 + \cdots + 2^i) + 2^{i+1} = (2^{i+1} - 1) + 2^{i+1} = 2 \cdot 2^{i+1} - 1 = 2^{i+2} - 1$.

This is enough. These two steps automatically imply that our statement is true for all infinite many i. Why? The second step can be thought of as writing an algorithm whose input is i and whose output is a proof for the correctness of the statement for $i+1$ where the proof assumes that the statement is true for i. By taking the proof for the smallest value for i, say $i = 0$, (step 1) and calling our algorithm with this value (step 2), we obtain in total a proof for the value $i + 1 = 1$. By repeatedly calling our algorithm for $i = 1$, $i = 2$, ..., we get in total a proof that our statement is true for any possible value for i.

$$\text{proof}(i = 0) \rightarrow \text{proof}(i = 1) \rightarrow \text{proof}(i = 2) \rightarrow \ldots$$

In our course:

- `x++` in Pseudocode (and in C++) means $x = x + 1$ (increasing the variable x by 1).
- `x--` means $x = x - 1$.
- `for i=1 to n` in Pseudocode means the same as
  - `for (int i=1; i <= n; i++)` in C++, and
  - `for i in range(1,n+1)` in Python.

  Importantly, the last iteration is for $i = n$.
- `a[2..n]` in Pseudocode defines an array a with available index positions from 2 to n (thus, this array contains $n-2+1 = n-1$ elements; hence, the *size* of a is $n-1$). An array is the same as a Python list or a C++ vector. Note that in Pseudocode, we can specify the range of available positions of an array, whereas in Python and C++ the first position is always 0 and only the last position is specified.
- To create a new array in Pseudocode, you can write any of this:
  1. `a[2..n] = new array`,
  2. `a[2..n] new array`, or simply just
  3. `a[2..n]`

  If the array elements are assumed to have an initial value c just write "filled with c". For example, `a[2..n] new array filled with 0s`. If you want to specify the type of the elements, for example that they should be integers, then write `a[2..n] new array of integers`. (If not specified, we assume an array to be of type integers.)
- `b = a[i..j]` creates a new array b that is a copy of the subarray $a[i..j]$. For example, if the array a is defined as `a[1..n]`, then `b = a[1..n/2]` is a new array that contains the first $n/2$ elements of a (as copies).
- We assume that variables created within a function call are **automatically deleted** at the end of that function call.
- `//` and `/*` (C++ style) and `#` (Python style) begin a comment. Thus, when writing `i = 2 // + 3`, we set the value of i to 2 and not to 5 as "+3" is just a comment.
- If b is a Boolean value (true/false), then `if b` is short for `if b==`**`True`** and `if !b` is short for `if b==`**`False`**.