# CSE103 Introduction to Algorithms

# TD5: Program Correctness

May 20th, 2021

## Exercise 1: Hoare triples, in English

Recall that a Hoare triple $\{P\}$ `<program>` $\{Q\}$ means "if $P$ is true, then after executing `<program>`, $Q$ will be true". Observe that the triple does *not* state that the execution of `<program>` will terminate, it is just making a claim under the hypothesis that it does.

For example, the meaning of the triple

$$\{\texttt{x} = n\} \texttt{<program>} \{\texttt{y} \geq 2 \wedge \forall z.(z \text{ divides } \texttt{y}) \Rightarrow (z = 1 \vee z = \texttt{y})\}$$

may be expressed in English by the sentence "if the variable `x` contains $n$, then after executing `<program>`, the variable `y` will contain a prime number".

Express in English the meaning of the following Hoare triples:

1. $\{\texttt{x} = n\}$ `<program>` $\{\texttt{x} = n + 3\}$
2. $\{\texttt{x} \leq \texttt{y}\}$ `<program>` $\{\texttt{y} \leq \texttt{x}\}$
3. $\{\texttt{True}\}$ `<program>` $\{\texttt{x} = 5\}$
4. $\{\texttt{True}\}$ `<program>` $\{\texttt{False}\}$
5. $\{\texttt{n} \geq 0\}$ `<program>` $\{\texttt{r}^2 \leq \texttt{n} \wedge (\texttt{r}+1)^2 > \texttt{n}\}$

**Solution.** Here are possible answers:

1. If `x` contains the value $n$, then after executing `<program>`, `x` will contain the value $n + 3$.
2. If the value of `x` is bounded above by the value of `y`, then after executing `<program>` the opposite will hold, *i.e.*, the value of `y` will be bounded above by the value of `x`.
3. After executing `<program>`, `x` will contain the value 5. (A naive formulation would start with "if `True` is true, then…", which is of course always the case, so we just omit it).
4. `<program>` never terminates. (A naive formulation would be "if `True` is true, then after executing `<program>`, `False` will be true". But `True` is always true, so we may just say "after executing `<program>`, `False` will be true". But `False` can never be true, so the only possibility is that there is no "after", that is, the execution of `<program>` never terminates).
5. If `n` is non-negative, then after executing `<program>`, `r` will contain the integer square root of `n`, that is, we will have $\texttt{r} = \lfloor \sqrt{\texttt{n}} \rfloor$.

## Exercise 2: valid and invalid Hoare triples

Which of the following Hoare triples is *valid*, that is, the claimed relation between precondition, program and postcondition always holds?

1. $\{\texttt{True}\}$ `x = 4` $\{\texttt{x} = 4\}$
2. $\{\texttt{x} = 3\}$ `x = x + 1` $\{\texttt{x} = 4\}$

3. $\{\texttt{True}\}$ `x = 3`<br>`y = 1` $\{\texttt{x} = 3\}$
4. $\{\texttt{x} = 0 \wedge \texttt{x} = 1\}$ `x = 5` $\{\texttt{x} = 42\}$
5. $\{\texttt{x} = 42\}$ `pass` $\{\texttt{x} = 41\}$
6. $\{\texttt{x} = 42\}$ `pass` $\{\texttt{x} \leq 100\}$
7. $\{\texttt{True}\}$ `while True:`<br>`  pass` $\{\texttt{False}\}$
8. $\{\texttt{x} = 0\}$ `while x = 0:`<br>`  x = x + 1` $\{\texttt{x} = 1\}$
9. $\{\texttt{x} = 1\}$ `while x != 0:`<br>`  x = x + 1` $\{\texttt{x} = 100\}$
10. $\{\texttt{x} = 1\}$ `while x != 0:`<br>`  x = x + 1` $\{\texttt{x} = 0\}$

**Solution.**

1. Valid.
2. Valid.
3. Valid: the extra assignment doesn't change the fact that $\texttt{x} = 3$ is true.
4. Valid: the precondition is equivalent to `False` (no variable may ever contain two different values at the same time). So the triple is saying "if `False` is true, then…", which is valid no matter what comes after the "then" because, in mathematical logic, anything follows from a false assumption.
5. Invalid: `pass` changes nothing, so the value of `x` after its execution will still be 42, not 41.
6. Valid: since `pass` changes nothing, the value of `x` after its execution will still be 42, which is certainly bounded above by 100.
7. Valid: in Exercise 1.4, we saw that a Hoare triple of the form $\{\texttt{True}\}$ `<program>` $\{\texttt{False}\}$ is stating that `<program>` never terminates, and this is an example of a loop that never terminates.
8. Valid: the loop is executed just once, and the value of `x` goes from 0 to 1.
9. Valid: with the given precondition, the loop never terminates, so any postcondition is correct.
10. Valid: for the same reason as above, but here we actually have an additional reason: if the loop *did* terminate, then it would do so precisely because $\texttt{x} = 0$!

## Exercise 3: the assignment rule

Replace the question marks below with an assertion making the statement a valid instance of the assignment rule of Floyd-Hoare logic. If there is more than one possibility, list them all; if there is none, justify why.

1. 
```
#! ???
x = 2 * x
#! x ≤ 10
```

2. `#!` $???$
   `x = 3`
   `#!` $0 \leq x \wedge x \leq 5$

3. `#!` $y > 0 \wedge y = 7$
   `x = y`
   `#!` $???$

4. `#!` $x + y > 0 \wedge y = 7$
   `x = x + y`
   `#!` $???$

5. `#!` $1 > 0 \wedge x = 7$
   `x = 1`
   `#!` $???$

**Solution.**

1. $2x \leq 10$
2. $0 \leq 3 \wedge 3 \leq 5$
3. There are four possibilities:
   (a) $y > 0 \wedge y = 7$ (no replacement);
   (b) $x > 0 \wedge y = 7$ (we replace the first instance of `y`);
   (c) $y > 0 \wedge x = 7$ (we replace the second instance of `y`);
   (d) $x > 0 \wedge x = 7$ (we replace both instances of `y`).
4. $x > 0 \wedge y = 7$
5. This cannot be made into a valid instance of the assignment rule because the assertion before the assignment contains `x`, which is the variable being assigned to, and the assigned expression is constant (does not contain `x`).

## Exercise 4: a wrong assignment rule

Let `<expr>` be an arbitrary arithmetic expression of mini-Python. Although it seems intuitively correct, the triple

$$\{\texttt{True}\}\ \texttt{x = <expr>}\ \{x = \texttt{<expr>}\}$$

is *not* valid in general. Can you find an example showing why this is the case? (*Hint: if* `<expr>` *is constant, the triple is valid, but if it's not constant...*).

**Solution.** As long as `<expr>` does not contain the variable `x`, the triple is valid:

`#!` `True`
`#!` `<expr>` $=$ `<expr>`
`x = <expr>`
`#!` $x =$ `<expr>` (this is a valid application of the assignment rule because `<expr>` does not contain `x`)

However, take for instance `<expr>` to be `x + 1`. Then the triple becomes

$$\{\texttt{True}\}\ \texttt{x = x + 1}\ \{\texttt{x} = \texttt{x} + 1\}.$$

If this were valid, then, by an application of the consequence rule, so would be

$$\{\texttt{True}\}\ \texttt{x = x + 1}\ \{\texttt{False}\},$$

because the postcondition $\texttt{x} = \texttt{x} + 1$ is contradictory (no variable may be equal to itself plus 1). But the latter triple is invalid: by Exercise 1.4, it is stating that the program `x = x + 1` does not terminate, which is obviously false.

## Exercise 5: the while rule

Knowing that the triples

$$\{\texttt{x} \leq 100 \wedge \texttt{x} < 100\}\ \texttt{x = x + 1}\ \{\texttt{x} \leq 100\} \qquad \text{and} \qquad \{\texttt{x} > 100 \wedge \texttt{x} > 100\}\ \texttt{x = x + 1}\ \{\texttt{x} > 100\}$$

are both valid (if you do not see why, pause one moment and convince yourself that they are!), replace the question marks below with an assertion making the statement a correct instance of the while rule of Floyd-Hoare logic:

1.
```
#! x ≤ 100
while x < 100:
    #! x ≤ 100 ∧ x < 100
    x = x + 1
    #! x ≤ 100
#! ???
```

2.
```
#! x > 100
while x > 100:
    #! x > 100 ∧ x > 100
    x = x + 1
    #! x > 100
#! ???
```

Let `loop1` and `loop2` denote the first and second loop, respectively. Can you prove that the Hoare triples

$$\{\texttt{x} \leq 100\}\ \texttt{loop1}\ \{\texttt{x} = 100\} \qquad \text{and} \qquad \{\texttt{x} > 100\}\ \texttt{loop2}\ \{\texttt{False}\}$$

are valid?

**Solution.**

1. $\texttt{x} \leq 100 \wedge \texttt{x} \geq 100$;
2. $\texttt{x} > 100 \wedge \texttt{x} \leq 100$.

The validity of both triples is obtained by a applying a consequence rule after the while rule:

- in the first case, $\texttt{x} \leq 100 \wedge \texttt{x} \geq 100$ obviously implies $\texttt{x} = 100$;
- in the second case, $\texttt{x} > 100 \wedge \texttt{x} \leq 100$ is contradictory (no variable may contain a value that is *both* smaller *and* strictly greater than a given value), so it implies `False`.

## Exercise 6: proving Exercise 2

For each triple of Exercise 2 which you claimed valid, give a proof of validity using the rules of Floyd-Hoare logic. That is, for each such triple

$$\{P\}\ \texttt{<program>}\ \{Q\},$$

start with the precondition `#!` $P$ and show how it propagates through the instruction(s) of `<program>` via the rules of Floyd-Hoare logic (specifically, the assignment, while and consequence rules) until you obtain the postcondition `#!` $Q$.

**Solution.**

1.
```
#! True
#! 4 = 4
x = 4
#! x = 4
```

2.
```
#! x = 3
#! x + 1 = 4
x = x + 1
#! x = 4
```

3.
```
#! True
#! 3 = 3
x = 3
#! x = 3
y = 1
#! x = 3
```

4.
```
#! x = 0 ∧ x = 1
#! False
x = 5
#! False
#! x = 42
```

6.
```
#! x = 42
pass
#! x = 42
#! x ≤ 100
```

7.
```
#! True
while True:
    #! True ∧ True
    pass
    #! True ∧ True
    #! True
#! True ∧ ¬True
#! False
```

8.
```
#! x = 0
#! x = 0 ∨ x = 1
while x = 0:
    #! (x = 0 ∨ x = 1) ∧ x = 0
    #! x = 0
    #! x + 1 = 1
    x = x + 1
    #! x = 1
    #! x = 0 ∨ x = 1
#! (x = 0 ∨ x = 1) ∧ x ≠ 0
#! x = 1
```

9.
```
#! x = 1
#! x > 0
while x != 0:
    #! x > 0 ∧ x ≠ 0
    #! x > 0
    #! x + 1 > 0
    x = x + 1
    #! x > 0
#! x > 0 ∧ x = 0
#! False (because no variable may be at the same time null and strictly positive)
#! x = 100 (because False implies anything)
```

10.
```
#! x = 1
#! x > 0
while x != 0:
    #! x > 0 ∧ x ≠ 0
    #! x > 0
    #! x + 1 > 0
    x = x + 1
    #! x > 0
#! x > 0 ∧ x = 0
#! x = 0
```

In the last two items we see the difference pointed out in Exercise 2.9 and 2.10: in the last case, we could use that `x` $> 0 \wedge$ `x` $= 0$ implies `False` and that `False` implies `x` $= 0$ (because it implies anything), but we may avoid passing through `False`, because `x` $> 0 \wedge$ `x` $= 0$ *already* implies `x` $= 0$, regardless of being a contradictory statement.

## Exercise 7: `if` without `else`

The conditional rule of Floyd-Hoare logic that we saw in class is written as follows:

$$\frac{\{P \wedge \texttt{<expr>}\}\ \texttt{<code1>}\ \{Q\} \qquad \{P \wedge \neg\texttt{<expr>}\}\ \texttt{<code2>}\ \{Q\}}{\{P\}\ \texttt{if <expr>: <code1> else: <code2>}\ \{Q\}}$$

This rule states that if the triples $\{P \wedge$ `<expr>`$\}$ `<code1>` $\{Q\}$ and $\{P \wedge \neg$`<expr>`$\}$ `<code2>` $\{Q\}$ are both valid, then the triple $\{P\}$ `if <expr>: <code1> else: <code2>` $\{Q\}$ is also valid. Informally, this is expected because it is saying that if $P$ leads to $Q$ when the value of `<expr>` makes us go through the `if` branch, and $P$ leads to $Q$ when the value of `<expr>` makes us go through the `else` branch, then no matter which branch we go through, $P$ will lead to $Q$.

As highlighted in class, the rule is formulated only for `if` statements having an `else` branch. Using the fact that a conditional without `else` like

```
if <expr>:
    <code>
```

may be encoded as

```
if <expr>:
    <code>
else:
    pass
```

formulate a logical rule for `if` statements without `else` and argue that it is correct.

*Hint: your rule must have the form*

$$\frac{\{???\}\ \texttt{<code>}\ \{???\} \qquad ???}{\{P\}\ \texttt{if <expr>: <code>}\ \{Q\}}$$

*where you need to fill in the question marks with logical statements.*

**Solution.** The rule is

$$\frac{\{P \wedge \texttt{<expr>}\}\ \texttt{<code>}\ \{Q\} \qquad P \wedge \neg\texttt{<expr>} \Rightarrow Q}{\{P\}\ \texttt{if <expr>: <code>}\ \{Q\}}$$

The fact that it is correct may either be argued informally, or be shown formally by proving that the above rule is *derivable* from the other rules of Floyd-Hoare logic presented in class, modulo the encoding of `if` without `else` using `pass`:

$$\frac{\{P \wedge \texttt{<expr>}\}\ \texttt{<code>}\ \{Q\} \qquad \dfrac{\dfrac{}{\{P \wedge \neg\texttt{<expr>}\}\ \texttt{pass}\ \{P \wedge \neg\texttt{<expr>}\}} \quad P \wedge \neg\texttt{<expr>} \Rightarrow Q}{\{P \wedge \neg\texttt{<expr>}\}\ \texttt{pass}\ \{Q\}}}{\{P\}\ \texttt{if <expr>: <code> else: pass}\ \{Q\}}$$

More specifically, we used a do-nothing rule, a consequence rule and a conditional rule.

## Exercise 8: an old acquaintance

The very first exercise of TD1 asked you to find, by exhaustive search, the integer square root of a non-negative integer. In mini-Python, it looked something like this:

```
def sqrt(n):
    r = 0
    while r * r < n:
        r = r + 1
    if r * r != n:
        r = r - 1
    else:
        pass
    return r
```

Prove that the above algorithm is correct by showing that the Hoare triple

$$\{0 \leq \mathtt{n}\}\ \texttt{sqrt(n)}\ \{\mathtt{r}^2 \leq \mathtt{n} \wedge (\mathtt{r}+1)^2 > \mathtt{n}\}$$

is valid. That is, start with the assertion `#!` $0 \leq \mathtt{n}$ just before the instruction `r = 0` and show how it may be propagated down, using the rules of Floyd-Hoare logic, until you arrive at the assertion `#!` $\mathtt{r}^2 \leq \mathtt{n} \wedge (\mathtt{r}+1)^2 > \mathtt{n}$ just before the instruction `return r`.

*Hint: a possible loop invariant is* $\mathrm{pred}(\mathtt{r})^2 \leq \mathtt{n}$*, where* pred *is* truncated subtraction*, defined by* $\mathrm{pred}(x) = x - 1$ *for all* $x > 0$*, and* $\mathrm{pred}(x) = 0$ *for all* $x \leq 0$*.*

**Solution.** Here is the program annotated with the assertions:

```
def sqrt(n):
    #! 0 ≤ n
    #! 0 ≤ n ∧ pred(0)² ≤ n  (because pred(0) = 0)
    r = 0
    #! 0 ≤ n ∧ pred(r)² ≤ n
    while r * r < n:
```

```
        #! 0 ≤ n ∧ pred(r)² ≤ n ∧ r² < n
        #! 0 ≤ n ∧ r² ≤ n
        #! 0 ≤ n ∧ pred(r+1)² ≤ n  (because pred(r+1) = r)
        r = r + 1
        #! 0 ≤ n ∧ pred(r)² ≤ n
    #! 0 ≤ n ∧ pred(r)² ≤ n ∧ r² ≥ n
    if r * r != n:
        #! 0 ≤ n ∧ pred(r)² ≤ n ∧ r² ≥ n ∧ r² ≠ n
        #! 0 ≤ n ∧ pred(r)² ≤ n ∧ r² > n
        #! 0 ≤ n ∧ pred(r)² ≤ n ∧ r² > n ∧ r > 0  (because r² > n ≥ 0)
        #! 0 ≤ n ∧ (r-1)² ≤ n ∧ (r-1+1)² > n  (because pred(r) = r-1 when r > 0)
        r = r - 1
        #! 0 ≤ n ∧ r² ≤ n ∧ (r+1)² > n
        #! r² ≤ n ∧ (r+1)² > n
    else:
        #! 0 ≤ n ∧ pred(r)² ≤ n ∧ r² ≥ n ∧ r² = n
        #! r² ≤ n ∧ (r+1)² > n  (because r² = n implies this)
        pass
        #! r² ≤ n ∧ (r+1)² > n
    #! r² ≤ n ∧ (r+1)² > n
    return r
```

## Exercise 9: Nico was correct

Here is an implementation of Lomuto's partitioning scheme in mini-Python (with one-line swap instructions):

```
def partition(l,b,e):
    pivot = l[e]
    p = b
    i = b
    while i < e:
        if l[i] < pivot:
            l[i],l[p] = l[p],l[i]
            p = p + 1
        else:
            pass
        i = i + 1
    l[e],l[p] = l[p],l[e]
    return p
```

Prove the correctness of Lomuto's partitioning scheme by showing that the Hoare triple

$$\{\mathtt{b} \leq \mathtt{e}\}\ \texttt{partition(l,b,e)}\ \{(\forall j.\, \mathtt{b} \leq j < \mathtt{p} \Rightarrow \mathtt{l}[j] \leq \mathtt{l}[\mathtt{p}]) \wedge (\forall j.\, \mathtt{p} < j \leq \mathtt{e} \Rightarrow \mathtt{l}[j] \geq \mathtt{l}[\mathtt{p}])\}$$

is valid, in the sense that, if the precondition holds before executing the first instruction of the `partition` function, then the postcondition will hold just before the `return p` statement.

For the proof, you may ignore "index out of bounds" errors, that is, you may assume that whenever an expression of the form $\mathtt{l}[v]$ is evaluated, the value of $v$ always falls within the length of `l` (which is in fact always the case as long as `b` and `e` are valid indices of `l`, but we will not bother proving it).

For the one-line swap instruction, you may use the rule

```
#! P[a,b]
a,b = b,a
#! P[b,a]
```

which allows to exchange the role of the swapped expressions in an arbitrary statement $P$ containing them (for example, if $P[\mathtt{a},\mathtt{b}]$ is $\mathtt{a} \leq \mathtt{b}$, then $P[\mathtt{b},\mathtt{a}]$ is $\mathtt{b} \leq \mathtt{a}$).

*Hint: the loop inviariant is the one discussed in class, namely*

$$(\forall j.\, \mathtt{b} \leq j < \mathtt{p} \Rightarrow \mathtt{l}[j] < \mathtt{pivot}) \wedge (\forall j.\, \mathtt{p} \leq j < \mathtt{i} \Rightarrow \mathtt{l}[j] \geq \mathtt{pivot}) \wedge \mathtt{l}[\mathtt{e}] = \mathtt{pivot} \wedge \mathtt{i} \leq \mathtt{e}.$$

**Solution.** Here is the program annotated with the assertions. Since some assertions are very long, we write them on several lines using the notation

```
#! P
#!∧ Q
#!∧ R
```

which should be read as a unique assertion

```
#! P ∧ Q ∧ R
```

```
def partition(l,b,e):
    #! b ≤ e

    #! ∀j. b ≤ j < b ⇒ l[j] < pivot
    #!∧ ∀j. b ≤ j < b ⇒ l[j] ≥ pivot
    #!∧ l[e] = l[e] ∧ b ≤ e

    pivot = l[e]

    #! ∀j. b ≤ j < b ⇒ l[j] < pivot
    #!∧ ∀j. b ≤ j < b ⇒ l[j] ≥ pivot
    #!∧ l[e] = pivot ∧ b ≤ e

    p = b

    #! ∀j. b ≤ j < p ⇒ l[j] < pivot
    #!∧ ∀j. p ≤ j < b ⇒ l[j] ≥ pivot
    #!∧ l[e] = pivot ∧ b ≤ e

    i = b

    #! ∀j. b ≤ j < p ⇒ l[j] < pivot
    #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
    #!∧ l[e] = pivot ∧ i ≤ e

    while i < e:

        #! ∀j. b ≤ j < p ⇒ l[j] < pivot
        #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
        #!∧ l[e] = pivot ∧ i ≤ e ∧ i < e

        if l[i] < pivot:

            #! ∀j. b ≤ j < p ⇒ l[j] < pivot
            #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i ≤ e ∧ i < e
```

```
            #!∧ l[i] < pivot

            #! ∀j. b ≤ j < p ⇒ l[j] < pivot
            #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i + 1 ≤ e
            #!∧ l[i] < pivot ∧ l[p] ≥ pivot

            l[i],l[p] = l[p],l[i]

            #! ∀j. b ≤ j < p ⇒ l[j] < pivot
            #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i + 1 ≤ e
            #!∧ l[p] < pivot ∧ l[i] ≥ pivot

            #! ∀j. b ≤ j < p + 1 ⇒ l[j] < pivot
            #!∧ ∀j. p + 1 ≤ j < i + 1 ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i + 1 ≤ e

            p = p + 1

            #! ∀j. b ≤ j < p ⇒ l[j] < pivot
            #!∧ ∀j. p ≤ j < i + 1 ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i + 1 ≤ e

        else:

            #! ∀j. b ≤ j < p ⇒ l[j] < pivot
            #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i + 1 ≤ e
            #!∧ l[i] ≥ pivot

            #! ∀j. b ≤ j < p ⇒ l[j] < pivot
            #!∧ ∀j. p ≤ j < i + 1 ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i + 1 ≤ e

            pass

            #! ∀j. b ≤ j < p ⇒ l[j] < pivot
            #!∧ ∀j. p ≤ j < i + 1 ⇒ l[j] ≥ pivot
            #!∧ l[e] = pivot ∧ i + 1 ≤ e

        #! ∀j. b ≤ j < p ⇒ l[j] < pivot
        #!∧ ∀j. p ≤ j < i + 1 ⇒ l[j] ≥ pivot
        #!∧ l[e] = pivot ∧ i + 1 < e

        i = i + 1

        #! ∀j. b ≤ j < p ⇒ l[j] < pivot
        #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
        #!∧ l[e] = pivot ∧ i ≤ e

    #! ∀j. b ≤ j < p ⇒ l[j] < pivot
    #!∧ ∀j. p ≤ j < i ⇒ l[j] ≥ pivot
```

```
    #!∧ l[e] = pivot ∧ i ≤ e ∧ i ≥ e

    #! ∀j. b ≤ j < p ⇒ l[j] < pivot
    #!∧ ∀j. p ≤ j < e ⇒ l[j] ≥ pivot
    #!∧ l[e] = pivot ∧ l[p] ≥ pivot

    l[e],l[p] = l[p],l[e]

    #! ∀j. b ≤ j < p ⇒ l[j] < pivot
    #!∧ ∀j. p ≤ j < e ⇒ l[j] ≥ pivot
    #!∧ l[p] = pivot ∧ l[e] ≥ pivot

    #! ∀j. b ≤ j < p ⇒ l[j] ≤ l[p]
    #!∧ ∀j. p < j ≤ e ⇒ l[j] ≥ l[p]

    return p
```