1. For $x, y \in \mathbb{Z}^+$, the greatest common divisor of x and y, written as $\gcd(x, y)$ is largest positive integer that divides both x and y. For example, $\gcd(300, 180) = 60$. We usually use the following algorithm to find $\gcd(x, y)$:

$$\gcd(x,y) = \begin{cases} x \text{ or } y, & \text{if } x = y \\ \gcd(x-y, y), & \text{if } x > y \\ \gcd(x, y-x), & \text{if } x < y \end{cases}$$

For example, $\gcd(300, 180) = \gcd(120, 180) = \gcd(120, 60) = \gcd(60, 60) = 60$.
Create a program for the above algorithm, then give its full proof outline under total correctness.

- Let's start with the statement itself.

$$\textbf{while } x \neq y \textbf{ do if } x > y \textbf{ then } x := x - y \textbf{ else } y := y - x \textbf{ fi od}$$

- Then let's add precondition and post condition:

$\{x > 0 \wedge y > 0 \wedge x = x_0 \wedge y = y_0\}$
$\textbf{while } x \neq y \textbf{ do if } x > y \textbf{ then } x := x - y \textbf{ else } y := y - x \textbf{ fi od}$
$\{x = y = \gcd(x_0, y_0)\}$

- What can be a loop invariant? First, $x > 0$ and $y > 0$ are always true after each iteration. Also, we want show that $\gcd(x, y) = \gcd(x_0, y_0)$ after each iteration. Thus, we can get the following minimal proof out line:

$\{x > 0 \wedge y > 0 \wedge x = x_0 \wedge y = y_0\}$
$\{\textbf{inv } x > 0 \wedge y > 0 \wedge \gcd(x, y) = \gcd(x_0, y_0)\}$
$\textbf{while } x \neq y \textbf{ do}$
    $\textbf{if } x > y \textbf{ then } x := x - y \textbf{ else } y := y - x \textbf{ fi}$
$\textbf{od}$
$\{x = y = \gcd(x_0, y_0)\}$

- Then we expand this proof outline to a full proof outline under partial correctness.

$\{x > 0 \wedge y > 0 \wedge x = x_0 \wedge y = y_0\}$
$\{\textbf{inv } p \equiv x > 0 \wedge y > 0 \wedge \gcd(x, y) = \gcd(x_0, y_0)\}$
$\textbf{while } x \neq y \textbf{ do}$
    $\{p \wedge x \neq y\}$
    $\textbf{if } x > y \textbf{ then}$
        $\{p \wedge x \neq y \wedge x > y\}\ \{p[x - y / x]\}\ x := x - y\ \{p\}$
    $\textbf{else}$
        $\{p \wedge x \neq y \wedge x \leq y\}\ \{p[y - x / y]\}\ y := y - x\ \{p\}$
    $\textbf{fi}$
    $\{p\}$
$\textbf{od}$
$\{p \wedge x = y\}$
$\{x = y = \gcd(x_0, y_0)\}$

In the above proof outline, red is for loop rule and blue is for conditional rule 1 and backward assignment.

- To get the above proof outline, we need to show the following logic implications:
  - $x > 0 \wedge y > 0 \wedge x = x_0 \wedge y = y_0 \Rightarrow p$

- $p \wedge x = y \Rightarrow\ x = y = \text{gcd}(x_0, y_0)$
- $p \wedge x \neq y \wedge x > y \Rightarrow\ p[x - y\ /\ x]$
- $p \wedge x \neq y \wedge x \leq y \Rightarrow\ p[y - x\ /\ y]$

Here we omitted these arguments since we are focused on creating an outline in this example. But they are all correct, and we can easily show them.

- We can see that all the conditions in the proof outline are safe, and the statement itself cannot create error, so we don't need to add any domain predicates.

- Now we need to find a bound expression. Both x and y in the loop condition can be decrease after each iteration, so we can try $x + y$ as the bound: $x + y > 0$ and no matter we go to true or false branch in each iteration, its value is always decreased. Then we have the following full proof outline under total correctness.

$\{x > 0 \wedge y > 0 \wedge x = x_0 \wedge y = y_0\}$
$\{\textbf{inv}\ p \equiv x > 0 \wedge y > 0 \wedge \text{gcd}(x, y) = \text{gcd}(x_0, y_0)\}$
$\{\textbf{bd}\ x + y\}$
$\textbf{while}\ x \neq y\ \textbf{do}$
$\quad \{p \wedge x \neq y \wedge x + y = t_0\}$
$\quad \textbf{if}\ x > y\ \textbf{then}$
$\quad\quad \{p \wedge x \neq y \wedge x > y\ \wedge x + y = t_0\}$
$\quad\quad \{p[x - y\ /\ x]\ \wedge (x - y) + y < t_0\}\ x \coloneqq x - y\ \{p \wedge x + y < t_0\}$
$\quad \textbf{else}$
$\quad\quad \{p \wedge x \neq y \wedge x \leq y \wedge x + y = t_0\}$
$\quad\quad \{p[y - x\ /\ y]\ \wedge x + (y - x) < t_0\}\ y \coloneqq y - x\ \{p \wedge x + y < t_0\}$
$\quad \textbf{fi}$
$\quad \{p \wedge x + y < t_0\}$
$\textbf{od}$
$\{p \wedge x = y\}$
$\{x = y = \text{gcd}(x_0, y_0)\}$

In the above proof outline, green is for bound expression.

Loop Rule for Total Correctness*

- In assignments and exams, we will only have questions with either possible runtime error or loop while proving total correctness. The combination of two can be too complicated to create.

- To show that a loop $W \equiv \{\textbf{inv}\ p\}\{\textbf{bd}\ t\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}$ is totally correct, we need:
  a) p is safe.
  b) p is a loop invariant: $\{p \wedge B\}\ S\ \{p\}$ so $\{\textbf{inv}\ p\}\ W\ \{p \wedge \neg B\}$.
  c) p can imply t is at least 0 and safe.
  d) Loop bound t is decreased after each iteration: $\{p \wedge B \wedge t = t_0\}\ S\ \{p \wedge t < t_0\}$.
  e) No runtime error while evaluating B or executing S: $p \Rightarrow D(B)$ and $p \wedge B \Rightarrow D(S)$

  Thus, we have the following rule:

  1. $p \Leftrightarrow \downarrow p$
  2. $p \Rightarrow D(B)$
  3. $p \wedge B \Rightarrow D(S)$
  4. $p \Rightarrow \downarrow (t \geq 0)$
  5. $\{p \wedge B \wedge t = t_0\}\ S\ \{p \wedge t < t_0\}$

6. $\{\mathbf{inv}\ p\}\{\mathbf{bd}\ t\}$ **while** B **do** S **od** $\{p \wedge \neg B\}$ loop 1,2,3,4,5

Note that, each antecedent triple and the consequent triple in the rule are provable under total correctness.

Finding loop invariants and creating loops

- It is not always easy to create a loop that works correctly, and finding a good loop invariant is usually the most important part while creating a loop.
- Like loop bound expressions, there is no algorithm to find a good loop invariant; but there are some heuristics, they don't work in all cases.

- While introducing loop invariants, we discussed some basic needs of a loop invariant. Let loop $W \equiv \{\mathbf{inv}\ p\}$ **while** B **do** S **od** and if we need to show $\{p_0\}\ W\ \{q\}$, we need:
  - $p_0 \Rightarrow p$
  - $\{p \wedge B\}\ S\ \{p\}$ is provable.
  - $p \wedge \neg B \Rightarrow q$

  When we create a loop in a program, we don't have the loop body S and we don't have the loop condition B yet, all we have are conditions p_0 and q.

- To get a loop invariant, we ***usually start with q and try to weaken it***. Here are the reasons why:
  1) q is the postcondition here, it is usually an expectation from the user.
  2) We observe that $p \wedge \neg B \Rightarrow q$, but p itself should not be stronger than q (aka, we should have $q \Rightarrow p$): or else, there is no need to have W and the postcondition is already satisfied. Since $p \wedge \neg B \Rightarrow q$ and $q \Rightarrow p$, so p should be similarly strong as q.
  3) While weakening the postcondition q, we usually can get some insight on B at the same time.

- We have seen several ways to weaken a predicate:
  1) Replace a constant by a variable: for example, we can replace $2 + x = z$ by $y + x = z$ where y is a variable whose domain includes 2.
  2) Generalize an operation: for example, we can change $x = 0$ to $x \geq 0$.
  3) Add a disjunct: for example, we can weaken p to $p \vee r$.
  4) Remove a conjunct: for example, we can weaken $p \wedge r$ to p.

2. Create a program that sums the first n positive integers up and has postcondition $s = sum(0, n)$.

   1) First, let's try to find some possible loop invariants. Here, replacing a constant by a variable seems to be the best idea. There are two constants 0 and n in the expression so there are two ways to replace.
      a. If we replace n by a variable k, we get $s = sum(0, k)$. Since we need the sum of the first n integers, and k will equal to n after the loop, so we can initialize $k = 0$ and increase it in each iteration until $k = n$. And we will get a loop looks like this:

         $\{\mathbf{inv}\ s = sum(0,k) \wedge 0 \leq k \leq n\}\{\mathbf{bd}\ n - k\}$
         while $k \neq n$ **do**
            … make k larger and something else …
         od
         $\{s = sum(0,k) \wedge 0 \leq k \leq n \wedge k = n\}$ # $p \wedge \neg B$
         $\{s = sum(0,n)\}$

b. If we replace 0 by a variable k, we get $s = sum(k, n)$. Since we need the sum of the first n integers, and k will be equal to 0 after the loop, so we can initialize $k = n$ and decrease it in each iteration until $k = 0$. And we will get a loop looks like this:

$\{\mathbf{inv}\ s = sum(k, n) \wedge 0 \le k \le n\}\{\mathbf{bd}\ k\}$
while $k \neq 0$ **do**
    … make k smaller and something else …
od
$\{s = sum(k, n) \wedge 0 \le k \le n \wedge k = 0\}$ $\quad\quad$ $\#\ p \wedge \neg B$
$\{s = sum(0, n)\}$

- When we replace a constant c with a variable k, we need to consider the range of values of k can be. We usually end the program with $k = c$, so we need another variable d so that k has the range $[c, d]$ (or $[d, c]$, depend on whether k is increased or decreased in each iteration).