TP4 - Exercício 1

Grupo 1

- Diogo Coelho da Silva A100092
- Pedro Miguel Ramôa Oliveira A97686

1. Import's

In [44]:
```
from z3 import *
```

2.Definição do algoritmo de euclides

In [45]:
```
from z3 import *

def algoritmoEuclides(num1, num2):
    r_prev, r_curr = num1, num2
    s_prev, s_curr = 1, 0
    t_prev, t_curr = 0, 1
    while r_curr != 0:
        q = r_prev // r_curr
        r_prev, r_curr = r_curr, r_prev - q * r_curr
        s_prev, s_curr = s_curr, s_prev - q * s_curr
        t_prev, t_curr = t_curr, t_prev - q * t_curr
    return r_prev

# Example
print(algoritmoEuclides(12, 18))
```

6

In [46]:
```
from z3 import *

def Prova(formula):
    """
    Esta função tenta provar uma fórmula lógica por refutação.
    Basicamente, adiciona a negação da fórmula a um solver SAT (Satisfiab
    e verifica se o resultado é satisfazível.

    Se a negação da fórmula for insatisfazível, significa que a fórmula o
    é uma tautologia (sempre verdadeira), ou seja, provada.

    """

    # Cria uma instância de um solver Z3.
    s = Solver()
    # Adiciona a negação da fórmula ao solver. A ideia é mostrar que a ne
```

```
    s.add(Not(formula))
    # Verifica se o conjunto de restrições no solver (neste caso, apenas
    # que torna a negação verdadeira, e portanto a fórmula original falsa
    if s.check() == sat:
        print("Falha ao provar a propriedade.")
    else:
        print("Propriedade provada com sucesso.")

# Declara um conjunto de variáveis inteiras que serão usadas para represe
x, y, r, rr, s, ss, t, tt, q = Ints('x y r rr s ss t tt q')

# Define uma função recursiva para calcular o máximo divisor comum (MDC)
gcd_func = RecFunction('gcd_func', IntSort(), IntSort(), IntSort())
# Declara duas variáveis inteiras `a` e `b` que serão usadas como argumen
a, b = Ints('a b')
# Neste caso, usa a definição padrão do algoritmo de Euclides para o MDC:
# Se `b` for 0, o MDC é `a`. Caso contrário, o MDC de `a` e `b` é o mesmo
RecAddDefinition(gcd_func, (a, b), If(b == 0, a, gcd_func(b, a % b)))

# Define a pré-condição do algoritmo. Neste caso, exige que os valores in
pre_con = And(x > 0, y > 0)

# Define o estado inicial do algoritmo. Aqui, `r` e `rr` recebem os valor
init_state = And(r == x, rr == y, s == 1, ss == 0, t == 0, tt == 1)

# Define o invariante de loop. Um invariante de loop é uma condição que é
# Aqui, o invariante garante duas coisas:
# 1. O MDC de `x` e `y` é igual ao MDC de `r` e `rr`. Isso significa que
# 2. As igualdades de Bézout são mantidas: `r` pode ser expresso como uma
#    e `rr` também pode ser expresso como uma combinação linear de `x` e
loop_inv = And(gcd_func(x, y) == gcd_func(r, rr),
               r == x*s + y*t,
               rr == x*ss + y*tt)

# Define a condição pré-loop. Esta afirma que se a pré-condição (`pre_con
# então o invariante de loop (`loop_inv`) também será verdadeiro no iníci
pre_loop = And(pre_con, init_state)
# `Implies` cria uma implicação lógica. `pre_final` afirma que `pre_loop`
# Isto é, se a pré-condição e o estado inicial são satisfeitos, então o i
pre_final = Implies(pre_loop, loop_inv)

# Define a condição de saída do loop. O loop termina quando `rr` for igua
# Além disso, o invariante de loop (`loop_inv`) deve continuar válido qua
# `ForAll` quantifica universalmente as variáveis `r`, `rr`, `s`, `ss`, `
exit_cond = ForAll([r, rr, s, ss, t, tt], And(rr == 0, loop_inv))

# Define a pós-condição do algoritmo. Esta descreve o estado desejado após
# Aqui, a pós-condição afirma que:
# 1. O valor final de `r` é igual ao MDC de `x` e `y`.
# 2. A igualdade de Bézout é satisfeita com os valores finais de `r`, `s`
post_con = And(r == gcd_func(x, y),
               r == x*s + y*t)
# `post_final` afirma que se a condição de saída (`exit_cond`) for verdad
```

```
# Isto é, se o loop termina corretamente, então o resultado obtido (o val
post_final = Implies(exit_cond, post_con)

# Prova a especificação do algoritmo. A especificação é que se a pré-cond
# então a pós-condição será satisfeita quando o algoritmo terminar (se te
# A prova é feita verificando se a afirmação que combina `pre_final` e `p
# (após ser implicada por `True`, o que não altera o seu valor de verdade
# Se for insatisfatível, significa que não há nenhuma atribuição de valor
# o que implica que a afirmação original é sempre verdadeira, provando a
Prova(Implies(True, And(pre_final, post_final)))
```

```
Propriedade provada com sucesso.
```

In [ ]: