Exercício 2 (Correção Bubble Sort) - Trabalho Prático 4

Grupo 4:
Carlos Costa-A94543
Ruben Silva-A94633

Problema:

O programa Python seguinte implementa o algoritmo de bubble sort para ordenação in situ de um array de inteiros seq.

```
seq = [-2,1,2,-1,4,-4,-3,3]
changed = True
while changed:
    changed = False
    for i in range(len(seq) - 1):
        if seq[i] > seq[i+1]:
            seq[i], seq[i+1] = seq[i+1], seq[i]
            changed = True
pass
```

1. Defina a pré-condição e a pós-condição que descrevem a especificação deste algoritmo.
2. Sugira um invariante.
3. O ciclo for pode ser descrito por uma transição $\texttt{seq} \leftarrow exp(\texttt{seq})$. Construa uma relação de transição $\texttt{trans}(\texttt{seq}, \texttt{seq}')$ que modele esta atribuição.
4. Usando a técnica que lhe parecer mais conveniente verifique a correção do algoritmo.

Análise do Problema

Este é um problema sobre um a correção de um algoritmo, neste caso especificio, um BubbleSort.
Para o resolução teorica do problema foi usada as seguintes variáveis:

- $seq \rightarrow$ Array
- $n \in \mathbb{N} \rightarrow$ Indice atual do array

Condições do trans para o trans(seq,seq')

1. **same_list**

$$\forall_n : seq_n{}' = seq_n$$

2. **trans01**

$$seq_n > seq_{n+1} \wedge seq_n{}' = seq_{n+1} \wedge seq_{n+1}{}' = seq_n \wedge samelist$$

3. **trans02**

$$seq_n <= seq_{n+1} \wedge seq_n{}' = seq_n \wedge seq_{n+1}{}' = seq_{n+1} \wedge samelist$$

Pré e Pós Condições

1. Pré-Condição:

Esta pré condição garante-nos que a lista tem o minimo de elementos para ser possível fazer uma comparação

$$n >= 2$$

2. Pós-Condição:
Esta pré condição garante-nos que a cada membro da lista é menor que o seu sucessor

$$\forall_{i<n} : seq[i] \leq seq[i+1]$$

Codificação do Algoritmo

Importar o solver

1. Importar o z3-solver

In [1]:
```
from z3 import *
```

Função "declare"
Esta função é responsável pela declaração de todas as variáveis que serão utilizadas no solver.

1. Parâmetros:
   A. i -> um inteiro que será responsável por dar o nr às variaveis
   B. $size$ -> Tamanho do array
2. Função:
   A. Inicialmente criamos um dicionário para colocar todas as variáveis necessárias.
   B. Criamos n variáveis: (todos os membros do array)
3. Return do novo dicionário com as variáveis

In [2]:
```
def declare(i, size):
    state = {}
    for n in range(size):
        state["seq"+str(n)] = Int("seq"+str(n)+"frame"+str(i))
    return state
```

Função "init"
Esta função é responsável pela inicialização do primeiro node do traço.

1. Parâmetros:
   A. $state$ -> Primeiro node do traço
   B. $sequencer$ -> Array para usar no FOTS
2. Return de um "And" com a seguinte condição lógica: todos os membros do traço inicial com o mesmo valor do $sequence$

In [3]:
```
def init(state, sequence):
    init_list = []
    for n in range(len(sequence)):
        init_list.append(state["seq"+str(n)] == sequence[n])
    return And(init_list)
```

Função "trans"
Esta função é responsável pela criação das conexões lógicas necessárias para o FOTS.

1. Parâmetros:
   A. $curr$ -> Membro atual do traço
   B. $prox$ -> Membro seguinte ao atual do traço
   C. n -> Index atual do array
   D. $size$ -> Tamanho do array
2. Função:
   A. Criamos as condições lógicas:
      a. trans01: $(seq_n > seq_{n+1} \wedge seq_n{}' = seq_{n+1} \wedge seq_{n+1}{}' = seq_n \wedge samelist)$
      b. trans02: $(seq_n <= seq_{n+1} \wedge seq_n{}' = seq_n \wedge seq_{n+1}{}' = seq_{n+1} \wedge samelist)$
3. Return de um "Or" com a seguinte condição lógica: $(trans01 \vee trans02)$

In [4]:
```
def trans(curr, prox, n, size):
    same_list = []
    if (n+1!=size):
        for i in range (size):
            if i != n and i != n+1:
                same_list.append(prox["seq"+str(i)] == curr["seq"+str(i)])


        trans01 = And(curr["seq"+str(n)] > curr["seq"+str(n+1)],
                      prox["seq"+str(n)] == curr["seq"+str(n+1)],
                      prox["seq"+str(n+1)] == curr["seq"+str(n)],
                      And(same_list))

        trans02 = And(curr["seq"+str(n)] <= curr["seq"+str(n+1)],
                      prox["seq"+str(n)] == curr["seq"+str(n)],
                      prox["seq"+str(n+1)] == curr["seq"+str(n+1)],
                      And(same_list))



        return Or(trans01,trans02)

    for i in range (size):
        same_list.append(prox["seq"+str(i)] == curr["seq"+str(i)])
    return And(same_list)
```

Função "traço"
Está é a função principal e é a que irá juntar as funções todas e gerar o traço pretendido e com ele tabelar o output

1. Parâmetros:
   A. $declare$ -> Função declare
   B. $init$ -> Função init
   C. $trans$ -> Função trans
   D. $sequence$ -> Array para usar no FOTS
2. Função:
   A. Iniciamos o Solver
   B. Calculamos o tamanho do traço
   C. Criar o traço com as variáveis
   D. Inicializar as variáveis essenciais no primeiro node do traço
   E. Criar a conexão lógica entre os nodes do traço todos e adicionar ao solver
   F. Correr o Solver e tabelar o resultado

In [5]:
```
def traco(declare, init, trans, sequence):
    s = Solver()
    size = len(sequence)
    k = size*size

    #Gerar o traço
    trace = [declare(i, size) for i in range(k)]

    #Gerar o primeiro estado
    s.add(init(trace[0], sequence))

    #Gerar as condições lógicas do traço
    for i in range(k-1):
        s.add(trans(trace[i], trace[i+1], i%size, size))


    old_sequence =[]
    if s.check() == sat:
        m = s.model()
        for i in range(k):
            sequence_var= []
            for v in trace[i]:
                sequence_var.append(m[trace[i][v]])
            if i%size==0:
                if old_sequence == sequence_var :
                    return
                print("Frame: "+ str(i))
                print(sequence_var)
                old_sequence=sequence_var
```

Correção do Algoritmo

Pré//Pós-condição

In [6]:
```
def pre(sequence):
    return And(len(sequence)>=2)

def pos(curr,sequence):
    return And([curr["seq"+str(i)]<=curr["seq"+str(i+1)] for i in range(len(sequence)-2)])
```

Strongest post-condition

Na abordagem SPC a denotação de um fluxo com um comando de atribuição introduz um quantificador existencial, o que não é adequado à verificação com SMT solvers: $\{C\,; \mathbf{z} := \mathbf{e}\,| \;=\; \exists a.(\mathbf{z} = \mathbf{e}[a/\mathbf{z}]) \wedge |C|[a/\mathbf{z}]$

Para lidar com este problema pode-se converter o programa original ao formato "*single assignment*" (SA). Num programa SA cada variável só pode ser usada depois de ser atribuida e só pode ser atribuida uma única vez.

Um programa (onde variáveis são atribuídas mais do que uma vez) pode ser reescrito num programa SA criando "clones" distintos das variáveis de forma a que seja possível fazer uma atribuição única a cada instância.

Neste caso, a denotação $[\![C]\!]$ associa a cada fluxo $\mathbb{C}$ um predicado que caracteriza a sua correcção em termos lógicos (a sua VC) segundo a técnica SPC, sendo calculada pelas seguintes regras.

$$
\begin{array}{l}
[\![\textsf{skip}]\!] = T\!rue \\
[\![\textsf{assume}\ \phi]\!] = \phi \\
[\![\textsf{assert}\ \phi]\!] = \phi \\
[\![\mathbf{z} := \mathbf{e}]\!] = (\mathbf{z} = \mathbf{e}) \\
[\![(C_1||C_2)]\!] = [\![C_1]\!] \vee [\![C_2]\!]
\end{array}
$$

$$
\begin{array}{l}
[\![C\,; \textsf{skip}\ ]\!] = [\![C]\!] \\
[\![C\,; \textsf{assume}\ \phi]\!] = [\![C]\!] \wedge \phi \\
[\![C\,; \textsf{assert}\ \phi]\!] = [\![C]\!] \rightarrow \phi \\
[\![C\,; \mathbf{z} := \mathbf{e}]\!] = [\![C]\!] \wedge (\mathbf{z} = \mathbf{e}) \\
[\![C\,; (C_1||C_2)]\!] = [\![(C\,; C_1)||(C\,; C_2)]\!]
\end{array}
$$

In [7]:
```
def strongestPostCondition(declare, init, trans, sequence):
    s = Solver()
    size = len(sequence)
    k = size*size

    #Pre-Condição
    pre_Condition = pre(sequence)

    #Gerar o traço
    trace = [declare(i, size) for i in range(k)]

    #Variavel para fazer as condições logicas do programa
    prog = []

    #Gerar o primeiro estado
    prog.append(And(init(trace[0], sequence)))

    #Gerar as condições lógicas do traço
    for i in range(k-1):
        prog.append(And(trans(trace[i], trace[i+1], i%size, size)))

    #Pos-Condição
    pos_Condition = pos(trace[len(prog)-1], sequence)


    #Formula final do strongest post condition
    #Transformar a lista numa sequencia lógica
    prog = And(prog)

    #Funfção final do SPC
    spc = Not(Implies(And(pre_Condition,prog), pos_Condition))

    #Adicionar o spc ao solver
    s.add(spc)

    #Correr
    if s.check() == sat:
        print("Falhou")
    print("Correção do Algoritmo provado com sucesso!!!\n")
```

Executar o código

In [8]:
```
sequence = [-2,1,2,-1,4,-4,-3,3]
strongestPostCondition(declare, init, trans, sequence)
traco(declare, init, trans, sequence)
```

```
Correção do Algoritmo provado com sucesso

Frame: 0
[-2, 1, 2, -1, 4, -4, -3, 3]
Frame: 8
[-2, 1, -1, 2, -4, -3, 3, 4]
Frame: 16
[-2, -1, 1, -4, -3, 2, 3, 4]
Frame: 24
[-2, -1, -4, -3, 1, 2, 3, 4]
Frame: 32
[-2, -4, -3, -1, 1, 2, 3, 4]
Frame: 40
[-4, -3, -2, -1, 1, 2, 3, 4]
```