TP4 - Grupo 14

André Lucena Ribas Ferreira - A94956

Paulo André Alegre Pinto - A97391

Enunciado do Problema 2

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

In [1]:
```
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
print(seq)
```

```
[-4, -3, -2, -1, 1, 2, 3, 4]
```

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

Resolução

1. Pré-condição e pós-condição

As condições definidas para a resolução do problema são as demonstradas a seguir. No entanto, devido à implementação do algoritmo SAU demonstrado na teoria, não se coloca a redundância da terminação da execução do ciclo na pós-condição durante a implementação.

```
 Pré-Condição: assume n >= 0 and changed == True
     Pós-Condução: assert forall j . 0 <= j < n-1 -> seq[j] <= seq[j+1] and changed == False
```

In [2]:
```
from pysmt.shortcuts import *
from pysmt.typing import *
```

De seguida, definem-se as variáveis usadas no problema.

In [3]:
```
N = 8
n = Int(N)
i = Symbol('i', INT)
seq = Symbol('seq', ArrayType(INT, INT))
changed = Symbol('changed', BOOL)
f = Symbol('f', FunctionType(INT,[INT]))
```

A função `prove(g)` recebe um predicado e testa se é possível prová-lo para todo o traço de execução. Esta função é utilizada como auxiliar para testar a funcionalidade das implementações.

In [4]:
```
def prove(g):
    with Solver(name="z3") as s:
        s.add_assertion(Not(g))
        if s.solve():
            print("Failed to prove.")
            for inc in range(N):
                print(s.get_value(Select(seq, Int(inc))))
            for inc in range(N):
                x = f(Int(inc))
                print("f", s.get_value(x))
                #print("f", s.get_value())
        else:
            print("Proved.")
```

As seguintes funções são funções auxiliares à manipulação das variáveis da execução por parte do algoritmo SAU.

In [5]:
```
def prime(v):
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())
def fresh(v):
    return FreshSymbol(typename=v.symbol_type(),template=v.symbol_name()+"_%d")
```

2. Definição da transição

Transição com passos intermédios

A seguinte função `trans_seq`, auxiliada pela função `aux`, serve para realizar a transição entre *seq* e *seq´*. Para tal, criam-se *Arrays* intermédios onde se executam as atribuições, par a par de valores.

In [6]:
```
def aux(c,p,I):
    i = Int(I)
    l = list()
    for n in range(N):
        if n != I and n != I+1:
            l.append(Equals(Select(p,Int(n)), Select(c,Int(n))))

    l.append(Equals(Select(p, i), Ite(Select(c,i) < Select(c, i+Int(1)), Select(c,i), Select(c, i+Int(1)))))
    l.append(Equals(Select(p, i+Int(1)), Ite(Select(c,i) > Select(c, i+Int(1)), Select(c,i), Select(c, i+Int(1)

    return And(l)


def trans_seq(seq,seq_p,changed,changed_p):
    seqlist = []
    seqlist.append(seq)
    for i in range(N-2):
        seqlist.append(Symbol('seq' + str(i), ArrayType(INT,INT)))
    seqlist.append(seq_p)
    troca = Iff(Iff(changed_p, Bool(False)), Equals(seq,seq_p))

    return And(And([aux(seqlist[i],seqlist[i+1],i) for i in range(N-1)]),troca)
```

In [7]:
```
def test_solver(conditions):
    with Solver(name="z3") as s:
        if s.solve(conditions):
            for inc in range(N):
                print(s.get_value(Select(seq, Int(inc))))
            print()
            for inc in range(N):
                print(s.get_value(Select(prime(seq), Int(inc))))
        else:
            print("Unsat")
```

In [8]:
```
#Função Store não faz o pretendido
store1 = And(Equals(Select(seq, Int(0)), Int(-2) ),
             Equals(Select(seq, Int(1)), Int(1)  ),
             Equals(Select(seq, Int(2)), Int(2)  ),
             Equals(Select(seq, Int(3)), Int(-1) ),
             Equals(Select(seq, Int(4)), Int(4)  ),
             Equals(Select(seq, Int(5)), Int(-4) ),
             Equals(Select(seq, Int(6)), Int(-3) ),
             Equals(Select(seq, Int(7)), Int(3)  ))

pre  =  And(n>=Int(0), Iff(changed, Bool(True)), store1)
test_solver([pre, trans_seq(seq,prime(seq),changed,prime(changed))])
```

```
-2
1
2
-1
4
-4
-3
3

-2
1
-1
2
-4
-3
3
4
```

Transição através de função recursiva

A função `for_cycle` serve como função de transição tendo em consideração a atribuição a cada uma das entradas do array. Para cada entrada $0 \leq i < n-1$ `seq[i]` atribui-se o menor dos valores entre `seq[i+1]` e o máximo valor de todos os `seq[j]`, para $0 \leq j < i$.

Para tal, definiu-se uma função recursiva `f` que calcula o máximo dos valores até então:

$$\begin{gathered} \mathsf{f}(0) = seq[0] \\ \forall i.\ 0 < i < n \to \mathsf{f}(i) = max(seq[i], f(i-1)) \end{gathered}$$

In [9]:
```
ax1 = Equals(f(Int(0)), Select(seq, Int(0)))
ax2 = ForAll([i], Implies(And(i>Int(0), i<n),
                          Equals(f(i),
                                 Ite(Select(seq,i) >= f(i-Int(1)),
                                     Select(seq,i), f(i-Int(1))))))
axioms = And(ax1,ax2)
```

No seguinte exemplo nota-se no funcionamento do cálculo do máximo.

In [10]:
```
prove(Implies(And(axioms, store1), Equals(f(Int(1)), Int(1))))
prove(Implies(And(axioms, store1), Equals(f(Int(5)), Int(4))))
prove(Implies(And(axioms, store1), Equals(f(Int(5)), Int(1))))
```

```
Proved.
Proved.
Failed to prove.
-2
1
2
-1
4
-4
-3
3
f -2
f 1
f 2
f 2
f 4
f 4
f 4
f 4
```

Por fim, a seguinte função calcula o predicado que transforma `seq` em `seq'`. Também tem em consideração a mudança do `changed` durante a sua execução, alterando-o se e só se o array se manteve igual.

In [11]:
```
def for_cycle(c, p, changed_p):
    indutivo = ForAll([i], Implies(And(i<n-Int(1), i>=Int(0)),
                                   Equals(Select(p,i),
                                          Ite(Select(c, i+Int(1)) <= f(i),
                                              Select(c, i+Int(1)), f(i)))))

    final = Equals(Select(p, n - Int(1)), f(n - Int(1)))

    troca = Iff(Iff(changed, Bool(False)), Equals(c,p))

    return And(axioms, indutivo, troca, final)
```

In [12]:
```
pre  =  And(n>=Int(0), Iff(changed, Bool(True)), axioms, store1)
t = for_cycle(seq,prime(seq),prime(changed))
test_solver([t, pre])
```

```
-2
1
2
-1
4
-4
-3
3

-2
1
-1
2
-4
-3
3
4
```

No entanto, a solução não é forte o suficiente para restringir a execução única e simplesmente aos traços conseguidos a partir da execução da tentativa de descrição do ciclo.

In [13]:
```
test_solver([Not(t), pre])
```

```
-2
1
2
-1
4
-4
-3
3

9
9
9
9
9
9
9
9
```

3. SAU para correção do algoritmo

A seguinte classe é retirada da teoria para auxiliar na perspetiva de correção do algoritmo. No entanto, nenhuma das soluções acima descritas consegue resolver o problema segundo a sua implementação específica. Em comentário encontram-se testes de leitura das execuções conseguidas.

Define-se, de seguida, as variáveis da execução, tal como casos de estudo específicos.

In [15]:
```
variables = [changed, seq]

store2 = And([Equals(Select(seq, Int(i)), Int(-2)) for i in range(N-1)]
             + [Equals(Select(seq, Int(7)), Int(-3))])
store3 = And([Equals(Select(seq, Int(i)), Int(-2)) for i in range(N)])
```

Também se definem os predicados necessários, nas variáveis correspondentes, para a codificação das pré- e pós-condições, tal como da relação de transição e da condição de ciclo.

A solução que utiliza a função recursiva não é capaz de chegar a uma conclusão positiva sobre a ordenação do array, em nenhum dos casos testados.

In [16]:
```
pre   =  And(n>=Int(0), Iff(changed, Bool(True)), axioms, store1)
pos   =  And(ForAll([i], Implies(And(i>=0, i<n-1), Select(seq, i) <= Select(seq, i+1))))
cond  =  Iff(changed, Bool(True))                # condição de controlo do ciclo
trans =  for_cycle(seq, prime(seq), prime(changed))

W = SAU(variables, pre, pos, cond, trans)
W.unfold(8)
```

```
falha: número de tentativas ultrapassa o limite 8
```

In [17]:
```
pre  =  And(n>=Int(0), Iff(changed, Bool(True)), axioms, store2)
W = SAU(variables, pre, pos, cond, trans)
W.unfold(8)
```

```
falha: número de tentativas ultrapassa o limite 8
```

In [18]:
```
pre  =  And(n>=Int(0), Iff(changed, Bool(True)), axioms, store3)
W = SAU(variables, pre, pos, cond, trans)
W.unfold(8)
```

```
falha: número de tentativas ultrapassa o limite 8
```

In [19]:
```
pre  =  And(n>=Int(0), Iff(changed, Bool(True)), axioms)
W = SAU(variables, pre, pos, cond, trans)
W.unfold(8)
```

```
falha: número de tentativas ultrapassa o limite 8
```

A solução que utiliza arrays intermédias tem o problema contrário: considera sucessos em tentativas menores do que aquelas que é suposto.

In [20]:
```
trans = trans_seq(seq,prime(seq),changed,prime(changed))
pre  =  And(n>=Int(0), Iff(changed, Bool(True)), store1)
W = SAU(variables, pre, pos, cond, trans)
W.unfold(8)
```

```
sucesso na tentativa 2
```

In [21]:
```
pre  =  And(n>=Int(0), Iff(changed, Bool(True)), store2)
W = SAU(variables, pre, pos, cond, trans)
W.unfold(9)
```

```
sucesso na tentativa 2
```

Propostas de Resolução

Cremos que o problema limitante foi a não compreensão total do algoritmo SAU, o único que permitia a resolução deste exercício sem uso de invariante. Nenhuma das funções de transição definidas conseguiu utilizar corretamente a sua implementação para gerar traços corretos de execução do ciclo.

No caso da implementação com função recursiva, a questão de como manter os axiomas funcionais e sempre verdadeiros, tal como limitações encontradas ao longo dos testes quando usadas funções com mais que um argumento, nomeadamente problemas com os tamanhos dos arrays (como definir o tamanho do array como variáveil do Solver?) e apenas conseguir provar os casos com execução verdadeira da propriedade testada, o que é o caso menos recorrente na implementação do SAU, impediu a progressão para a resolução plena do exercício.

In [22]:
```
f2 = Symbol('f2', FunctionType(INT,[INT, INT]))
size = Symbol('size', INT)
ax1 = ForAll([size], Equals(f2(Int(0), size), Select(seq, Int(0))))
ax2 = ForAll([i, size], Implies(And(i>Int(0), i<size),
                                Equals(f2(i, size),
                                       Ite(Select(seq,i) >= f2(i-Int(1), size),
                                           Select(seq,i), f2(i-Int(1), size)))))
axioms = And(ax1,ax2)
```

In [23]:
```
size = Int(8)
prove(Implies(And(axioms, store1), Equals(f2(Int(1), size), Int(1))))
prove(Implies(And(axioms, store1), Equals(f2(Int(5), size), Int(4))))
prove(Implies(And(axioms, store1), Equals(f2(Int(5), size), Int(1))))
```

```
Proved.
Proved.
---------------------------------------------------------------------------
SolverReturnedUnknownResultError          Traceback (most recent call last)
Input In [23], in <cell line: 4>()
      2 prove(Implies(And(axioms, store1), Equals(f2(Int(1), size), Int(1))))
      3 prove(Implies(And(axioms, store1), Equals(f2(Int(5), size), Int(4))))
----> 4 prove(Implies(And(axioms, store1), Equals(f2(Int(5), size), Int(1))))

Input In [4], in prove(g)
      2 with Solver(name="z3") as s:
      3     s.add_assertion(Not(g))
----> 4     if s.solve():
      5         print("Failed to prove.")
      6         for inc in range(N):

File D:\Programs\Anaconda\lib\site-packages\pysmt\solvers\solver.py:359, in IncrementalTrackingSolver.solve(sel
f, assumptions)
    357 def solve(self, assumptions=None):
    358     try:
--> 359         res = self._solve(assumptions=assumptions)
    360         self._last_result = res
    361         return res

File D:\Programs\Anaconda\lib\site-packages\pysmt\decorators.py:64, in clear_pending_pop.<locals>.clear_pending
_pop_wrap(self, *args, **kwargs)
     62     self.pending_pop = False
     63     self.pop()
---> 64 return f(self, *args, **kwargs)

File D:\Programs\Anaconda\lib\site-packages\pysmt\solvers\z3.py:218, in Z3Solver._solve(self, assumptions)
    216 assert sres in ['unknown', 'sat', 'unsat']
    217 if sres == 'unknown':
--> 218     raise SolverReturnedUnknownResultError
    219 return (sres == 'sat')

SolverReturnedUnknownResultError:
```

Também se testou devolver uma implicação na função `for_cycle`, para melhor modelar o comportamento desejado, mas descobriu-se que a função de SAU já tem em conta estas implicações para uma execução correta.

In [24]:
```
def for_cycle_implies(c, p, changed_p):
    indutivo = ForAll([i], Implies(And(i<n-Int(1), i>=Int(0)),
                                   Equals(Select(p,i),
                                          Ite(Select(c, i+Int(1)) <= f(i),
                                              Select(c, i+Int(1)), f(i)))))

    final = Equals(Select(p, n - Int(1)), f(n - Int(1)))

    troca = Iff(Iff(changed, Bool(False)), Equals(c,p))

    return Implies(axioms, And(indutivo, troca, final))
```

In [25]:
```
trans =  for_cycle_implies(seq, prime(seq), prime(changed))

W = SAU(variables, pre, pos, cond, trans)
W.unfold(8)
```

```
falha: número de tentativas ultrapassa o limite 8
```

Por fim, a implementação da variável `changed` para modelação completamente correta do ciclo deve ter sido o maior erro cometido na implementação, já que dependendo do tratamento do resto das condições, o comportamento da variável dentro da execução do SAU varia imenso.

In [ ]: