TP4 - Algoritmo de Bubble Sort

Janeiro, 2022

Bruno Miguel Ferreira Fernandes - a95972

Hugo Filipe de Sá Rocha - a96463

Começamos por importar as bibliotecas do PySMT e definir a função prove que verifica a validade de uma fórmula lógica usando um SMT solver.

In [20]:
```
from pysmt.shortcuts import *
from pysmt.typing import *


def prove(f):
    with Solver(name="z3") as s:
        s.add_assertion(Not(f))
        if s.solve():
            print("Failed to prove.")
        else:
            print("Proved.")
```

Definir variáveis

In [21]:
```
seq = [-2,1,2,-1,4,-4,-3,3]
n = tam = len(seq)
j = len(seq)-1
changed = True
```

Algoritmo Bubble Sort

Tentamos fazer a prova deste algoritmo através do invariante e para isso introduzimos a variável j da qual depende completamente o invariante deste algoritmo.

In [22]:
```
print(f"Avaliar pré-cond. e inv.: j = {j}; n = {n}; changed = {changed}; seq = {seq}")
while changed or j != 0:
    changed = False
    for i in range(n-1):
        if seq[i] > seq[i+1]:
            seq[i], seq[i+1] = seq[i+1], seq[i]
            changed = True
    j -= 1
    print(f"j = {j}: {seq}//Todos os elementos a partir da posição {j} estão ordenados")
pass
print(f"Avaliar pós-cond: j = {j}; changed = {changed}; seq = {seq}")
```

```
Avaliar pré-cond. e inv.: j = 7; n = 8; changed = True; seq = [-2, 1, 2, -1, 4, -4, -3,
3]
j = 6: [-2, 1, -1, 2, -4, -3, 3, 4]//Todos os elementos a partir da posição 6 estão orde
nados
j = 5: [-2, -1, 1, -4, -3, 2, 3, 4]//Todos os elementos a partir da posição 5 estão orde
nados
j = 4: [-2, -1, -4, -3, 1, 2, 3, 4]//Todos os elementos a partir da posição 4 estão orde
nados
```

```
j = 3: [-2, -4, -3, -1, 1, 2, 3, 4]//Todos os elementos a partir da posição 3 estão orde
nados
j = 2: [-4, -3, -2, -1, 1, 2, 3, 4]//Todos os elementos a partir da posição 2 estão orde
nados
j = 1: [-4, -3, -2, -1, 1, 2, 3, 4]//Todos os elementos a partir da posição 1 estão orde
nados
j = 0: [-4, -3, -2, -1, 1, 2, 3, 4]//Todos os elementos a partir da posição 0 estão orde
nados
Avaliar pós-cond: j = 0; changed = False; seq = [-4, -3, -2, -1, 1, 2, 3, 4]
```

Definimos a pré-condição e a pós-condição que descrevem a especificação deste algoritmo.

Tentamos provar a correção pelas 3 propriedades do invariante que são: -> Ser válido no inicio do programa -> Ser útil -> Ser preservado ao fim de cada iteração do ciclo

```
pre: n == len(seq) and n >= 1 and changed == True and j == len(seq)-1
pos: forall k . 0 <= k < n-1 -> seq[k] <= seq[k+1] and changed == False and j == 0
inv: forall k . j <= k < n-1 -> seq[k] <= seq[k+1]
```

Tanto a pré como a pós-condição verificam-se no início e final do programa, respetivamente.

```
Podemos também perceber que o invariante é válido no início do programa e a sua
utilidade também
é garantida visto que ~(changed == True or j != 0) and inv. => pós-condição.
                      changed == False and j == 0 and inv. => pós-condição.
```

Quanto à preservação do invariante em todas as iterações, é possível perceber que o algoritmo procura empurrar os maiores elementos para o final da lista pelo que na k-ésima iteração, os últimos k-ésimos elementos estão ordenados. Para isso utilizamos a variável j.

In [23]:
```
seq = Symbol('seq', ArrayType(INT, INT))
changed = Symbol('changed', BOOL)
n = Symbol('n',INT)
j = Symbol('j',INT)
k = Symbol('k', INT)


pre = And(n>=Int(1), Equals(n, Int(tam)), Iff(changed, Bool(True)),
          Equals(j, (n+Int(-1))))


pos = And(ForAll([k],Implies(And(k>=Int(0),k<=(n+Int(-2))),
                     Select(seq,k)<=Select(seq,k+1))),
                     Iff(changed, Bool(False)), Equals(j, Int(0)))

inv = ForAll([k], Implies(And(k>=j ,k<=(n-2)),
                  Select(seq,k) <= Select(seq,k+1)))



init = Implies(pre, inv)

util = Implies(And(Iff(changed, Bool(False)), Equals(j,Int(0)), inv), pos)

#não conseguimos provar a preservação...

prove(init)
prove(util)
```

```
Proved.
Proved.
```

Outra alternativa que não necessitava da variável j nem do invariante seria a técnica SAU descrita abaixo:

In [24]:
```
# Auxiliares
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")

# A classe "Sigle Assignment Unfold"
class SAU(object):
    """Trivial representation of a while cycle and its unfolding."""
    def __init__(self, variables, pre , pos, control, trans, sname="z3"):

        self.variables = variables       # variables
        self.pre = pre                   # pre-condition as a predicate in "variables"
        self.pos = pos                   # pos-condition as a predicate in "variables"
        self.control = control           # cycle control as a predicate in "variables"
        self.trans = trans               # cycle body as a binary transition relation
                                         # in "variables" and "prime variables"

        self.prime_variables = [prime(v) for v in self.variables]
        self.frames = [And([Not(control),pos])]
                 # inializa com uma só frame: a da terminação do ciclo

        self.solver = Solver(name=sname)

    def new_frame(self):
        freshs = [fresh(v) for v in self.variables]
        b = self.control
        S = self.trans.substitute(dict(zip(self.prime_variables,freshs)))
        W = self.frames[-1].substitute(dict(zip(self.variables,freshs)))

        self.frames.append(And([b , ForAll(freshs, Implies(S, W))]))

    def unfold(self,bound=0):
        n = 0
        while True:
            if n > bound:
                print("falha: número de tentativas ultrapassa o limite %d "%bound)
                break

            f = Or(self.frames)
            if self.solver.solve([self.pre,Not(f)]):
                self.new_frame()
                n += 1
            else:
                print("sucesso na tentativa %d "%n)
                break




# O ciclo
i = Symbol("i", INT)
changed = Symbol("changed", BOOL)
seq = Symbol("seq", ArrayType(INT, INT))

variables = [i, changed, seq]

pre  =  And(n>=Int(1), Equals(n, Int(tam)), Iff(changed, Bool(True)))
```

```
pos = And(ForAll([k],Implies(And(k>=Int(0),k<=(n+Int(-2))),
                             Select(seq,k)<=Select(seq,k+1))),
                             Iff(changed, Bool(False))) # pós-condição

cond =  Iff(changed , Bool(True)) # condição de controlo do ciclo

#trans =  # corpo do ciclo como uma relaçao de transição, não conseguimos fazer...

#W = SAU(variables, pre, pos, cond, trans)
```