C3. Casos de estudo

Copiar link

Correcção de algoritmos de pesquisa num *array*

É interessante estudar a correcção (e invariantes) de algoritmos de pesquisa, por se tratar de algoritmos relativamente simples, que contêm um único ciclo, mas cujo número de iterações não é constante nem previsível, ao contrário de exemplos vistos anteriormente em que é efectuada uma travessia completa do array (como o somatório ou contagem de ocorrências).

Pesquisa linear

O programa seguinte procura *k* no array *vector* entre os índices *a* e *b,* guardando o índice da primeira ocorrência na variável *r,* que ficará com o valor -1 caso não exista qualquer ocorrência.

O programa inclui 3 pós-condições. A primeira delimita os valores possíveis de *r*, e as duas seguintes caracterizam cada um desses casos.

```
method search(v:array<int>, a:int, b:int, k:int) returns (r:int)
    requires 0 <= a <= b < v.Length
    ensures r == -1 || (a <= r <= b)
    ensures r == -1 ==> forall x: int :: a <= x <= b ==> v[x] != k
    ensures a <= r <= b ==> v[r] == k
{
    var i := a;
    while (i<=b && v[i]!=k)
      invariant ...
    {
      i := i+1;
    }
    if (i>b) { r := -1; }
    else { r := i; }
}
```

Note-se que a comparação é feita na própria condição do ciclo!

Na escrita do invariante teremos de ter em conta que a condição poderá falhar agora, caso tenha sido atingido o índice b, ou ainda caso (v[i] == k), mas **de certeza que k não ocorreu**

nas posições anteriores do array, porque nesse caso a execução não teria atingido o ponto actual.

Sendo assim um candidato a invariante será
$I \equiv (0 \leq i \leq b+1) \wedge (\forall x.\, a \leq x < i \rightarrow v[x] \neq k)$

Os triplos de Hoare correspondentes às 3 propriedades do invariante são os seguintes:

1. **Inicialização:** $\{0 \leq a \leq b < v.Length\}$ `i := a` $\{I\}$
2. **Preservação:** $\{I \wedge i \leq b \wedge v[i] \neq k\}$ `i := i + 1` $\{I\}$
3. **Utilidade:**

$\{I \wedge (i > b \vee v[i] = k)\}$
`if (i > b) {r := -1} else {r := i}`
$\{(r = -1 \vee (a \leq r \leq b) \wedge (r = -1 \rightarrow \forall x.\, a \leq x \leq b \rightarrow v[x] \neq k) \wedge (a \leq r \leq b \rightarrow v[r] = k)\}$

Note-se que I é claramente verdade imediatamente a seguir à avaliação da condição do ciclo, mesmo quando essa condição falha. Por outro lado, é fácil argumentar sobre a utilidade deste invariante para provar a pós-condição. A execução termina num dos seguintes cenários:

- $i = b+1$ (foi atingido o final do *array*). Neste caso, o invariante I implica que k não se encontra nas posições $[a \ldots b]$ de v, o que é coerente com a instrução `r := -1` que será executada em seguida

- $i < b+1$ e $vector[i] = k$ (k encontrado na posição actual). Neste caso, o invariante I implica que k não ocorre nas posições $[a \ldots i-1]$, pelo que i é o índice da primeira ocorrência, o que mais uma vez é coerente com a instrução `r := i` que será executada em seguida

Exercícios

1. Considere a seguinte versão alternativa do algoritmo de procura linear, em que a comparação é agora feita no corpo do ciclo (note que a especificação é a mesma da versão anterior).
   a. Defina um invariante para o ciclo
   b. Escreva os triplos de Hoare correspondentes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
   c. Verifique o programa recorrendo ao Dafny

```
method search(vector:array<int>, a:int, b:int, k:int) returns (r:int)
    requires 0 <= a <= b < vector.Length
    ensures r == -1 || (a <= r <= b)
    ensures r == -1 ==> forall x: int :: a <= x <= b ==> vector[x] != k
    ensures a <= r <= b ==> vector[r] == k
```

```
{
    r := -1;
    var i := a;
    while (i<=b && r == -1)
      invariant ...
    {
       if (vector[i] == k) { r := i; }
       i := i+1;
    }
}
```

Pesquisa num *array* ordenado

Se o *array* se encontrar à partida ordenado, o algoritmo de pesquisa linear poderá ser optimizado por forma a terminar a execução do ciclo antecipadamente, caso seja consultado um elemento *superior* ao procurado (assumindo que a ordenação do array é *crescente*).

Note que neste caso terá de ser incluída uma pré-condição exigindo que o *array* se encontre ordenado, como se segue:

```
method searchSorted(vector:array<int>, a:int, b:int, k:int) returns (r:int)
    requires 0 <= a <= b < vector.Length
    requires forall i1,i2 : int :: a <= i1 <= i2 <= b ==> vector[i1] <= vector[i2]
    ensures r == -1 || (a <= r <= b)
    ensures r == -1 ==> forall x: int :: a <= x <= b ==> vector[x] != k
    ensures a <= r <= b ==> vector[r] == k
{
   ...
}
```

Exercício
Complete o programa optimizado acima, incluindo o invariante de ciclo necessário para provar a sua correcção, e verifique-o com Dafny.

Pesquisa Binária

Podendo assumir-se que o *array* se encontra à partida ordenado, um método de pesquisa muito mais eficiente é a pesquisa binária, que em cada passo diminui para metade o comprimento da secção do array em que é feita a pesquisa, calculando o índice que se encontra a meio dessa secção e comparando-o com o elemento procurado.

Exercícios
Escreva invariantes apropriados e verifique cada uma das seguintes versões do algoritmo.

Dica: o invariante deverá afirmar que, caso *k* ocorra no array, terá de ocorrer na secção actualmente considerada para pesquisa, ou seja entre os índices *l* e *u*.

Versão 1 — comparação feita no corpo do ciclo:

```
method search(vector:array<int>, a:int, b:int, k:int) returns (r:int)
    requires 0 <= a <= b < vector.Length
    requires forall i1,i2 : int ::
             a <= i1 <= i2 <= b ==> vector[i1] <= vector[i2]
    ensures r == -1 || (a <= r <= b)
    ensures r == -1 ==> forall x: int :: a <= x <= b ==> vector[x] != k
    ensures a <= r <= b ==> vector[r] == k
{
    r := -1;
    var m; var l := a; var u := b;
    while (l <= u && r == -1)
       invariant ...
   {
       m := l + (u-l) / 2;
       if (vector[m] < k) { l := m+1; }
       else { if (vector[m] > k) { u := m-1; }
              else { r := m; }}
    }
}
```

Versão 2 — comparação feita na condição do ciclo:

```
method search(vector:array<int>, a:int, b:int, k:int) returns (m:int)
    requires 0 <= a <= b < vector.Length
    requires forall i1,i2 : int ::
```

```
            a <= i1 <= i2 <= b ==> vector[i1] <= vector[i2]
    ensures m == -1 || (a <= m <= b)
    ensures m == -1 ==> forall x: int :: a <= x <= b ==> vector[x] != k
    ensures a <= m <= b ==> vector[m] == k
{
    var l := a; var u := b;
    m := l + (u-l) / 2;
    while (l < u && vector[m]!=k)
       invariant ...
    {
       if (vector[m] < k) { l := m+1; }
       if (vector[m] > k) { u := m-1; }
       m := l + (u-l) / 2;
    }
    if (m < l || vector[m] != k) { m := -1; }
}
```

Determinação de elementos repetidos num *array*

Considere agora o problema de se determinar se um *array* contém ou não algum elemento repetido. O seguinte programa faz isso de forma eficiente, colocando r com valor *true* caso exista um par de posições do *array* contendo o mesmo elemento, e parando a execução quando isso acontece.

```
  r := false;
  i := 0;
  while (i<N-1 && !r)
  {
     j := i+1;
     while (j<N && !r)
     {
        if a[i] == a[j] { r := true; }
        j := j+1;
     }
```

```
      i := i+1;
   }
```

Em Dafny poderíamos escrever o seguinte método:

```
method dups(a: array<int>) returns (r: bool)
   requires a.Length > 0
   ensures r <==> (exists k, l :: 0 <= k < l < a.Length && a[k] == a[l])
{
   r := false;
   var j; var N := a.Length;
   var i := 0;
   while (i<N-1 && !r)
      invariant ... <= i <= ...
      invariant !r ==>
      invariant  r ==>
   {
      j := i+1;
      while (j<N && !r)
         invariant ... <= j <= ...
         invariant !r ==>
         invariant  r ==>
      {
         if a[i] == a[j] { r := true; }
         j := j+1;
      }
      i := i+1;
   }
}
```

Escreva os invariantes necessários para provar a correcção do programa, procurando descrever o que é garantido no início de cada iteração dos ciclos interior e exterior, nos casos em que r seja verdadeiro e falso.