AULA 3

CORREÇÃO DE ALGORITMOS E PROGRAMAS

É fato que estamos muito interessados em construir algoritmos e programas eficientes, conforme vimos na aula 2. No entanto, de nada vale um algoritmo eficiente mas incorreto. Por correto, queremos dizer que o algoritmo sempre pára com a resposta correta para todo entrada. Devemos, assim, ser capazes de mostrar que nossos algoritmos são eficientes e corretos.

Há duas estratégias básicas para mostrar que um algoritmo, programa ou função está correto, dependendo de como foi descrito. Se é recursivo, então a indução matemática é usada imediatamente para mostrar sua correção. Por outro lado, se não-recursivo, então contém um ou mais processos iterativos, que são controlados por estruturas de repetição. Processos iterativos podem ser então documentados com invariantes, que nos ajudam a entender os motivos pelos quais o algoritmo, programa ou função programa está correto. Nesta aula veremos como mostrar que uma função, recursiva ou não, está correta.

Esta aula é inspirada em [1, 2].

3.1 Correção de funções recursivas

Mostrar a correção de uma função recursiva é um processo quase que imediato. Usando indução matemátca como ferramenta, a correção de uma função recursiva é dada naturalmente, visto que sua estrutura intrínseca nos fornece muitas informações úteis para uma prova de correção. Na aula 1, mostramos a correção da função recursiva `maximo` usando indução.

Vamos mostrar agora que a função `potR` descrita no exercício 1.1 está correta. Reproduzimos novamente a função `potR` a seguir.

```
/* Recebe um dois números inteiros x e n e devolve x a n-ésima potência */
int potR(int x, int n)
{
   if (n == 0)
      return 1;
   else
      return x * potR(x, n-1);
}
```

Proposição 3.1. *A função* `potR` *recebe dois números inteiros x e n e devolve corretamente o valor de x^n.*

Demonstração.

Vamos mostrar a proposição por indução em n.

Se $n = 0$ então a função devolve $1 = x^0 = x^n$.

Suponha que a função esteja correta para todo valor k, com $0 < k < n$. Ou seja, a função `potR` com parâmetros x e k devolve corretamente o valor x^k para todo k, com $0 < k < n$.

Agora, vamos mostrar que a função está correta para $n > 0$. Como $n > 0$ então a última linha do corpo da função é executada:

```
return x * potR(x, n-1);
```

Então, como $n-1 < n$, por hipótese de indução, a chamada `potR(x, n-1)` nesta linha devolve corretamente o valor x^{n-1}. Logo, a chamada de `potR(x, n)` devolve $x \cdot x^{n-1} = x^n$. □

3.2 Correção de funções não-recursivas e invariantes

Funções não-recursivas, em geral, possuem uma ou mais estruturas de repetição. Essas estruturas, usadas para processar um conjunto de informações de entrada e obter um outro conjunto de informações de saída, também são conhecidas como processos iterativos da função. Como veremos daqui por diante, mostrar a correção de uma função não-recursiva é um trabalho mais árduo do que de uma função recursiva. Isso porque devemos, neste caso, extrair informações úteis desta função que explicam o funcionamento do processo iterativo e que nos permitam usar indução matemática para, por fim, mostrar que o processo está correto, isto é, que a função está correta. Essas informações são denominadas invariantes de um processo iterativo.

3.2.1 Definição

Um **invariante** de um processo iterativo é uma relação entre os valores das variáveis envolvidas neste processo que vale no início de cada iteração do mesmo. Os invariantes explicam o funcionamento do processo iterativo e permitem provar por indução que ele tem o efeito desejado.

Devemos provar três elementos sobre um invariante de um processo iterativo:

Inicialização: é verdadeiro antes da primeira iteração da estrutura de repetição;

Manutenção: se é verdadeiro antes do início de uma iteração da estrutura de repetição, então permanece verdadeiro antes da próxima iteração;

Término: quando a estrutura de repetição termina, o invariante nos dá uma propriedade útil que nos ajuda a mostrar que o algoritmo ou programa está correto.

Quando as duas primeiras propriedades são satisfeitas, o invariante é verdadeiro antes de toda iteração da estrutura de repetição. Como usamos invariantes para mostrar a correção de um algoritmo e/ou programa, a terceira propriedade é a mais importante, é aquela que permite mostrar de fato a sua correção.

Dessa forma, os invariantes explicam o funcionamento dos processos iterativos e permitem provar, por indução, que esses processos têm o efeito desejado.

3.2.2 Exemplos

Nesta seção veremos exemplos do uso dos invariantes para mostrar a correção de programas. O primeiro exemplo, dado no programa 3.1, é bem simples e o programa contém apenas variáveis do tipo inteiro e, obviamente, uma estrutura de repetição. O segundo exemplo, apresentado no programa 3.2, é um programa que usa um vetor no processo iterativo para solução do problema.

Considere então o programa 3.1, que recebe um número inteiro $n > 0$ e uma seqüência de n números inteiros, e mostra a soma desses n números inteiros. O programa 3.1 é simples e não usa um vetor para solucionar esse problema.

Programa 3.1: Soma n inteiros fornecidos pelo(a) usuário(a).

```
#include <stdio.h>

/* Recebe um número inteiro n > 0 e uma seqüência de n números
   inteiros e mostra o resultado da soma desses números */
int main(void)
{
   int n, i, num, soma;

   printf("Informe n: ");
   scanf("%d", &n);

   soma = 0;
   for (i = 1; i <= n; i++) {
      /* variável soma contém o somatório dos
         primeiros i-1 números fornecidos */
      printf("Informe um número: ");
      scanf("%d", &num);
      soma = soma + num;
   }

   printf("Soma dos %d números é %d\n", n, soma);

   return 0;
}
```

É importante destacar o *comentário* descrito nas duas linhas seguintes à estrutura de repetição `for` do programa 3.1: este é o *invariante* desse processo iterativo. E como Feofiloff destaca em [2], o enunciado de um invariante é, provavelmente, o único tipo de comentário que vale a pena inserir no corpo de um algoritmo, programa ou função.

Então, podemos provar a seguinte proposição.

Proposição 3.2. *O programa 3.1 computa corretamente a soma de $n \geqslant 0$ números inteiros fornecidos pelo(a) usuário(a).*

Demonstração.

Por conveniência na demonstração, usaremos o modo matemático para expressar as variáveis do programa. Dessa forma, denotaremos i no lugar de `i`, *soma* no lugar de `soma` e *num* ao invés de `num`. Quando nos referirmos ao i-ésimo número inteiro da seqüência de números inteiros fornecida pelo(a) usuário(a), que é armazenado na variável `num`, usaremos por conveniência a notação num_i.

Provar que o programa 3.1 está correto significa mostrar que para qualquer valor de n e qualquer seqüência de n números, a variável *soma* conterá, ao final do processo iterativo, o valor

$$soma = \sum_{i=1}^{n} num_i \, .$$

Vamos agora mostrar que o invariante vale no início da primeira iteração do processo iterativo. Como a variável *soma* contém o valor 0 (zero) e i contém 1, é verdade que a variável *soma* contém a soma dos $i - 1$ primeiros números fornecidos pelo(a) usuário(a).

Suponha agora que o invariante valha no início da i-ésima iteração, com $1 < i < n$.

Vamos mostrar que o invariante vale no início da última iteração, quando i contém o valor n. Por hipótese de indução, a variável *soma* contém o valor

$$\alpha = \sum_{i=1}^{n-1} num_i \, .$$

Dessa forma, no decorrer da n-ésima iteração, o(a) usuário(a) deve informar um número que será armazenado na variável num_n e, então, a variável *soma* conterá o valor

$$\begin{aligned} soma &= \alpha + num_n \\ &= \left(\sum_{i=1}^{n-1} num_i \right) + num_n \\ &= \sum_{i=1}^{n} num_i \, . \end{aligned}$$

Portanto, isso mostra que o programa 3.1 de fato realiza a soma dos n números inteiros fornecidos pelo(a) usuário(a). □

O próximo exemplo é dado pelo seguinte problema: dado um vetor com n números inteiros fornecidos pelo(a) usuário(a), encontrar um valor máximo armazenado nesse vetor. O programa 3.2 é bem simples e se propõe a solucionar esse problema.

Programa 3.2: Mostra um maior valor em um vetor com n números inteiros.

```
#include <stdio.h>

#define MAX 100

/* Recebe um número inteiro n > 0 e uma seqüência de n
   números inteiros e mostra um maior valor da seqüência */
int main(void)
{
   int n, vet[MAX], i, max;

   scanf("%d", &n);
   for (i = 0; i < n; i++)
      scanf("%d", &vet[i]);

   max = vet[0];
   for (i = 1; i < n; i++) {
      /* max é um maior elemento em vet[0..i-1] */
      if (vet[i] > max)
         max = vet[i];
   }
   printf("%d\n", max);

   return 0;
}
```

Vamos mostrar agora a correção do programa 3.2.

Proposição 3.3. *O programa 3.2 encontra um elemento máximo de um conjunto de n números fornecidos pelo(a) usuário(a).*

Demonstração.

Novamente, por conveniência na demonstração usaremos o modo matemático para expressar as variáveis do programa: trocaremos `i` por i, `vet` por *vet* e `max` por *max*.

Provar que o programa 3.2 está correto significa mostrar que para qualquer valor de n e qualquer seqüência de n números fornecidos pelo(a) usuário(a) e armazenados em um vetor *vet*, a variável *max* conterá, ao final do processo iterativo, o valor do elemento máximo em $vet[0..n-1]$.

Vamos mostrar que o invariante vale no início da primeira iteração do processo iterativo. Como *max* contém o valor armazenado em $vet[0]$ e, em seguida, a variável i é inicializada com o valor 1, então é verdade que a variável *max* contém o elemento máximo em $vet[0..i-1]$.

Suponha agora que o invariante valha no início da i-ésima iteração, com $1 < i < n$.

Vamos mostrar que o invariante vale no início da última iteração, quando i contém o valor $n-1$. Por hipótese de indução, no início desta iteração a variável *max* contém o valor o elemento máximo de $vet[0..n-2]$. Então, no decorrer dessa iteração, o valor $vet[n-1]$ é comparado com *max* e dois casos devem ser avaliados:

(*i*) $vet[n-1] > max$

Isso significa que o valor $vet[n-1]$ é maior que qualquer valor armazenado em $vet[0..n-2]$. Assim, na linha 15 a variável *max* é atualizada com $vet[n-1]$ e portanto a variável *max* conterá, ao final desta última iteração, o elemento máximo da seqüência em $vet[0..n-1]$.

(*ii*) $vet[n-1] \leqslant max$

Isso significa que existe pelo menos um valor em $vet[0..n-2]$ que é maior ou igual a $vet[n-1]$. Por hipótese de indução, esse valor está armazenado em *max*. Assim, ao final desta última iteração, a variável *max* conterá o elemento máximo da seqüência em $vet[0..n-1]$.

Portanto, isso mostra que o programa 3.2 de fato encontra o elemento máximo em uma seqüência de n números inteiros armazenados em um vetor. □

Exercícios

3.1 O programa 3.3 recebe um número inteiro $n > 0$, uma seqüência de n números inteiros, um número inteiro x e verifica se x pertence à seqüência de números.

Programa 3.3: Verifica se x pertence à uma seqüência de n números.

```
#include <stdio.h>

#define MAX 100

int main(void)
{
   int n, C[MAX], i, x;

   scanf("%d", &n);
   for (i = 0; i < n; i++)
      scanf("%d", &C[i]);
   scanf("%d", &x);

   i = 0;
   while (i < n && C[i] != x)
      /* x não pertence à C[0..i] */
      i++;

   if (i < n)
      printf("%d é o %d-ésimo elemento do vetor\n", x, i);
   else
      printf("%d não se encontra no vetor\n", x);

   return 0;
}
```

Mostre que o programa 3.3 está correto.

3.2 Escreva uma função com a seguinte interface:

```
void inverte(int n, int v[MAX])
```

que receba um número inteiro $n > 0$ e uma seqüência de n números inteiros armazenados no vetor, e devolva o vetor a seqüência de números invertida. Mostre que sua função está correta.

3.3 Mostre que sua solução para o exercício 1.6 está correta.

3.4 Mostre que sua solução para o exercício 1.7 está correta.

3.5 Mostre que sua solução para o exercício 1.8 está correta.