Escola de Artes, Ciências e Humanidades (USP-Leste)
Algoritmos e Estruturas de Dados I - 2º Semestre de 2015
Prof. Alexandre da Silva Freire (`afreire@ime.usp.br` – sala 322K-I1)

Aula 12 - 25/09 - Invariantes

Prova de corretude de algoritmos iterativos

"O pessimista se queixa do vento, o otimista espera que ele mude e o realista ajusta as velas." (William George Ward)

Ao verificar que um algoritmo funciona corretamente para várias instâncias de um problema, os otimistas tendem a acreditar que tal algoritmo funcionará para qualquer instância desse mesmo problema, o que muitas vezes não acontece na prática. Os pessimistas, ao se frustrarem com os frequentes erros em seus algoritmos, tendem a achar que não se pode confiar em programas de computador e enchem a boca para dizer que "certamente todos eles um dia falharão". Já os realistas, antes de emitirem qualquer julgamento a respeito de um algoritmo, eles o submetem a um exame rigoroso da razão. Os realistas só confiam em um algoritmo se conseguem **provar** que ele funciona para qualquer instância do problema em questão.

Exemplos de algoritmos "quase corretos"

Vejamos alguns exemplos de algoritmos que podem enganar os otimistas, que mais cedo ou mais tarde se frustrarão e, se persistirem no erro de não lançarem mão da razão, correm o risco de se tornarem pessimistas. Nos exemplos abaixo, deseja-se projetar um algoritmo que receba um vetor de inteiros e devolva o valor de um elemento máximo do vetor (ou seja, um elemento que é maior ou igual a todos os outros elementos do vetor). Note que, a documentação de um método deve informar de forma precisa, clara e objetiva **o que** faz o método, assim como na documentação do `elementoMaximo1`. Nos próximos exemplos, omitiremos a documentação, assumindo que será a mesma deste exemplo. Assumiremos que o método sempre será chamado com parâmetros corretos (ou seja, `v!=null` e `v.length>=1`). O ideal seria que o método lançasse alguma exceção (`IllegalArgumentException`, por exemplo) nos casos em que há algum parâmetro incorreto. Como esse não é o foco de nossa disciplina, não faremos isso.

```
/*
 * Recebe: um vetor "v" de inteiros, tal que "v.length >= 1"
 *
 * Devolve: um inteiro "max" satisfazendo (1) e (2)
 * (1) "v[i]=max", para algum "i", tal que  "0 <= i < v.length"
 * (2) "max >= v[j]", para todo "j", tal que "0 <= j < v.length"
 */
int elementoMaximo1(int[] v) {
   int i_max = 0;

   for(int i = 0; i < v.length - 1; i++) {
      if(v[i] > v[i_max]) {
         i_max = i;
      }
   }

   return v[i_max];
}
```

O algoritmo `elementoMaximo1` **não funcionará corretamente se o vetor possuir um único elemento máximo e o mesmo estiver na última posição**. Isso ocorre por causa da condição de parada do `for` que foi mal definida. Vejamos mais uma tentativa de resolver o mesmo problema.

```
int elementoMaximo2(int[] v) {
   int i_max = 1;

   for(int i = 0; i < v.length; i++) {
      if(v[i] > v[i_max]) {
         max = i;
      }
   }

   return v[i_max];
}
```

O algoritmo `elementoMaximo2` funciona para qualquer instância em que o vetor `v` tenha dois ou mais elementos. Ou seja, **o caso trivial em que o vetor possui um único elemento é justamente o caso em que o algoritmo falha** (será lançada uma `NullPointerException` na linha do `if`). Isso ocorre porque a variável `i_max` foi inicializada com o valor 1, porém os vetores em java são indexados a partir do índice 0. Vejamos outro algoritmo.

```
int elementoMaximo3(int[] v) {
   int max = 0;

   for(int i = 0; i < v.length; i++) {
      if(v[i] > max) {
         max = v[i];
      }
   }

   return max;
}
```

O algoritmo `elementoMaximo3` funciona para qualquer vetor de inteiros em que pelo menos um dos elementos é não-negativo. Ou seja, o algoritmo

não funcionará corretamente para nenhuma instância em que todos os elementos são negativos. Isso ocorre porque a variável `max`, que supostamente deveria armazenar o valor de um elemento máximo do vetor, é inicializada com o valor 0. Para corrigir o algoritmo, poderíamos inicializar a variável `max` com o valor `v[0]`. Vejamos se o próximo algoritmo se sai melhor.

```
int elementoMaximo4(int[] v) {
   for(int i = 1; i < v.length; i++) {
      if(v[i-1] > v[i]) {
         int aux = v[i];
         v[i] = v[i-1];
         v[i-1] = aux;
      }
   }

   return v[v.length - 1];
}
```

Embora não seja tão trivial enxergar isso, o algoritmo `elementoMaximo4` devolve o valor de um elemento máximo do vetor para qualquer instância do problema. Ainda assim, este algoritmo não está correto! Note que **o vetor passado como parâmetro é alterado pelo algoritmo, o que não deveria acontecer**. Embora não tenha sido explicitamente documentado que o vetor original não deveria ser alterado, também não foi documentado que o algoritmo poderia rearranjar os elementos do vetor e, nesses casos, o bom senso nos diz que, na dúvida, é melhor assumir que o vetor original não deve ser alterado.

É importante ressaltar que a corretude de um algoritmo está vinculada ao problema que ele se propõe a resolver. Se, no caso do problema de encontrar um elemento máximo, fosse especificado que o vetor de entrada terá pelo menos dois elementos, que sempre há pelo menos dois elementos máximos, que pelo menos um dos elementos será não-negativo e que o vetor pode ser alterado pelo algoritmo, todos os algoritmos apresentados acima estariam corretos.

Provando a corretude de um algoritmo iterativo

Para provar a corretude de um algoritmo iterativo, precisamos identificar qual é a sua "essência", ou seja, qual é a propriedade que se mantém válida ao longo de toda a execução e que nos garante que, após o término da execução, o algoritmo terá feito corretamente aquilo que foi especificado. Chamamos de *invariante* toda propriedade que se mantém válida durante toda a execução de um algoritmo.

Considere o seguinte algoritmo para resolver o problema de encontrar um elemento máximo em um vetor.

```
  int elementoMaximo5(int[] v) {
1.   int k = v[0];
2.
3.   for(int i = 1; i < v.length; i++) {
4.      if(v[i] > k) {
5.         k = v[i];
6.      }
7.   }
8.
9.   return k;
  }
```

Para provar que o algoritmo `elementoMaximo5` está correto, primeiro mostraremos que a seguinte propriedade é um *invariante* do algoritmo.

Lema 1. *Na linha 3 do algoritmo* `elementoMaximo5`*, em qualquer iteração, imediatamente antes de verificar a condição de parada, temos que* $\texttt{k} = \max_{0 \le j \le i-1} \texttt{v[j]}$.

Demonstração. Na primeira iteração (**inicialização**), vale que `k=v[0]` e `i=1`. Logo, o intervalo `v[0..i-1]` contém um único elemento, que é justamente o elemento `k` e, portanto, a propriedade é satisfeita.

Assuma que a propriedade é satisfeita (ou seja, $\texttt{k} = \max_{0 \le j \le i-1} \texttt{v[j]}$) no início de uma iteração qualquer, exceto a última. Provaremos que a propriedade será satisfeita no início da iteração seguinte (**manutenção**). Sejam `i'` e `k'` os valores de `i` e `k` na próxima iteração. Como `i'=i+1` e considerando a condição do `if` da linha 4 e a atribuição feita na linha 5, caso a condição seja satisfeita, temos que

$$\texttt{k'} = \max\{\texttt{k}, \texttt{v[i]}\} = \max\{\max_{0 \le j \le i-1} \texttt{v[j]}, \texttt{v[i]}\} = \max_{0 \le j \le i = i'-1} \texttt{v[j]}.$$

Portanto, a propriedade será satisfeita no início da próxima iteração. □

No Lema 1, provamos que a propriedade enunciada é um invariante do algoritmo, ou seja, ela vale ao longo de toda a execução. Para isso, mostramos que a propriedade é válida na **inicialização** (ou seja, antes da primeira iteração) e se mantém válida ao longo das demais iterações (**manutenção**). Note que nossa prova utilizou o *princípio da indução finita*. Porém, isso ainda não é o bastante para provar a corretude do algoritmo, pois a condição de parada pode estar incorreta, como acontece no algoritmo `elementoMaximo1`. Para finalizar a prova de corretude do algoritmo `elementoMaximo5`, temos que argumentar que a condição de parada está correta.

Proposição 1. *O algoritmo* `elementoMaximo5` *está correto.*

Demonstração. Seja `n=v.length`. Note que na última vez que a linha 3 do algoritmo `elementoMaximo5` é executada, imediatamente antes de verificar a condição de parada, temos que `i=n`. De acordo com o Lema 1, temos que $\texttt{k} = \max_{0 \le j \le n-1} \texttt{v[j]}$ e, portanto, o algoritmo devolve o valor de um elemento máximo de `v`. □

Ordenação por seleção

Considere o seguinte problema (ordenação): dado um vetor v com n inteiros, rearranjar v de forma que $v[0] \leq v[1] \leq \ldots \leq v[n-1]$. Dado que acabamos de projetar um algoritmo para encontrar um elemento máximo de um vetor, uma solução natural para resolver o problema da ordenação seria adotar a seguinte estratégia: para $i = 1, 2, \ldots, n$, selecionar um elemento máximo de $v[0 \ldots n-i]$, digamos $v[k]$, e inverter as posições dos elementos $v[k]$ e $v[n-i]$. Essa estratégia é conhecida como *ordenação por seleção* (*selection sort*) e, tradicionalmente, ao invés de selecionarmos um elemento máximo e o colocarmos na última posição, equivalentemente, selecionamos um elemento mínimo e o colocamos na primeira posição. Para não contrariar a tradição, nosso algoritmo utilizará essa segunda estratégia.

```
/*
 * Recebe: um vetor "v" de inteiros
 *
 * O que faz: rearranja o vetor "v" de forma que
 *  v[0] <= v[1] <= ... <= v[v.length -1]
 */
  void selectionSort(int[] v) {
1.   for(int i = 0; i < v.length; i++) {
2.      int k = indiceElemMin(i, v);
3.      int aux = v[i];
4.      v[i] = v[k];
5.      v[k] = aux;
6.   }
  }

/*
 * Recebe: um vetor "v" de inteiros e um indice "i" dentro do vetor
 *
 * Devolve: o indice de um elemento minimo de "v[i .. v.length -1]"
 */
int indiceElemMin(int i ,int[] v) {
   int k = i;
   for (int q = i; q < v.length; q++) {
      if(v[q] < v[k]) {
         k = q;
      }
   }
   return k;
}
```

Para provar que o algoritmo `selectionSort` está correto, precisamos mostrar o seguintes lemas.

Lema 2. *O algoritmo* `indiceElemMin` *está correto.*

A prova do Lema 2 é similar à prova do Lema 1 + Proposição 1.

Lema 3. *Seja* `n=v.length`. *Na linha 1 do algoritmo* `selectionSort`, *em qualquer iteração, imediatamente antes de verificar a condição de parada, temos que:*

(1) `v[0 .. n-1]` *é uma permutação do vetor original;*

(2) `v[j]` $\leq$ `v[k]` *para quaisquer* `j` *e* `k`*, tais que* $0 \leq$ `j` $\leq$ `i-1` *e* `i` $\leq$ `k` $\leq$ `n-1`*;*

(3) `v[0 .. i-1]` *está em ordem crescente.*

Demonstração. Como o algoritmo só altera os elementos do vetor através de trocas, claramente a propriedade (1) é satisfeita, não havendo necessidade de uma prova mais formal. Assim sendo, provaremos apenas as propriedades (2) e (3).

Na primeira iteração (**inicialização**), como `i=0`, temos que o intervalo `v[0 .. i-1]` é vazio e, portanto, as propriedades (2) e (3) não são violadas.

Assuma, por hipótese de indução, que as propriedades (2) e (3) são satisfeitas no início de uma iteração qualquer, exceto a última. Provaremos que tais propriedades serão satisfeitas no início da iteração seguinte (**manutenção**). Após a execução das linhas 2 a 5, é feita a inversão do elemento `v[i]` com o elemento `v[k]`. Como `v[k]` é um elemento mínimo do intervalo `v[i .. n-1]`, temos que `v[i]` $\leq$ `v[j]`, para `j=i+1,...,n-1`. Além disso, pela hipótese de indução, sabemos que `v[i]` $\geq$ `v[j]`, para `j=0,...,i-1`, e que `v[0 .. i-1]` está em ordem crescente. Logo, após a troca realizada na linhas 2 a 5, as propriedades (2) e (3) estarão satisfeitas no início da próxima iteração. □

Exercícios

Para cada um dos algoritmos abaixo, enuncie o problema que ele resolve e encontre os invariantes necessários para demonstrar que o respectivo algoritmo está correto (não precisa demonstrar os invariantes).

```
void misterio1(int[] v) {
   for (int i = 0; i < v.length; i++) {
      if(v[i] < 0) {
         v[i] = v[i] * (-1);
      }
   }
}


int misterio2(int[] v) {
   int x = 0;
   for (int i = 0; i < v.length; i++) {
      int y = 0;
      for (int j = i; j < v.length; j++) {
         y += v[j];
         x = Math.max(y, x);
      }
   }
   return x;
}
```

```
boolean misterio3(char[][] m, int a, int b) {
   boolean[][] v = new boolean[m.length][m[0].length];
   int[] x = {0,0,-1,1,1,-1,0,0};
   Fila<Integer> f1 = new FilaLL<Integer>();
   Fila<Integer> f2 = new FilaLL<Integer>();
   f1.enfileira(a);
   f2.enfileira(b);

   while (!f1.vazia()) {
      int l = f1.desenfileira();
      int c = f2.desenfileira();

      for (int i = 0; i < 4; i++) {
         int ll = l + x[i];
         int cc = c + x[i + 4];

         if (m[ll][cc] == 'd') {
            return true;
         } else if (!v[ll][cc] && m[ll][cc] == 'l') {
            v[ll][cc] = true;
            f1.enfileira(ll);
            f2.enfileira(cc);
         }
      }
   }

   return false;
}



int misterio4(int[] v) {
   int x = 0;
   int y = 0;

   for (int i = 0; i < v.length; i++) {
      y += v[i];

      if(y < 0) {
         y = 0;
      }

      x = Math.max(y, x);
   }

   return x;
}



void misterio5(int[] v) {
   for (int c = 1; c < v.length; c++) {
      for (int d = 0; d < v.length - c; d++) {
         if (v[d] < v[d+1]) {
            int x = v[d];
            v[d] = v[d+1];
            v[d+1] = x;
         }
      }
   }
}
```