Construção e Análise de Algoritmos

aula 07: Corretude de algoritmos I

1 Introdução

Há algumas semanas, alguém me perguntou após a aula como era possível se certificar que um algoritmo está correto.

A resposta é que isso requer uma análise do algoritmo.

Mas essa análise é diferente daquela que fazemos para determinar o seu tempo de execução.

Uma analogia vai nos ajudar a entender as coisas mais fácil.

Um algoritmo é como uma máquina.

E como é que você faz para saber se uma máquina funciona direito, isto é, se ela foi bem construída?

Bom, você pode colocar a máquina para funcionar em diferentes situações.

Mas, mesmo que tudo vá bem, você nunca terá a certeza de que ela não vai falhar no próximo caso.

Para ter certeza mesmo, a única alternativa é examinar a estrutura da máquina: as suas peças e como elas estão encaixadas umas com as outras.

Você deve concentrar a sua atenção nas peças principais.

Mas, mesmo que elas estejam em ordem e bem encaixadas, isso ainda não basta.

Uma máquina funciona em movimento.

E com o movimento as coisas podem sair do lugar.

A ideia então é que, depois de ter verificado que tudo está em ordem, você gira a manivela uma vez.

Se tudo continua bem, você gira a manivela outra vez.

Fazendo isso algumas vezes, você pode se dar conta de que o funcionamento da máquina é um processo repetitivo.

Isto é, as coisas estão de novo e de novo acontecendo do mesmo jeito.

Isso significa que, se as coisas estão em ordem em um certo momento, então elas também estarão em ordem alguns passos depois (i.e., se o girar da manivela não tira as coisas do lugar).

Pronto, agora que você descobriu isso, e viu que tudo está em ordem no início, você já tem boas razões para acreditar que a máquina funciona direito.

Bom, é a mesma coisa com os algoritmos.

Se você se certificar que as peças principais estão todas no lugar certo no início.

E que elas continuam no lugar certo quando o algoritmo dá suas voltas em torno dos laços.

Então você tem boas razões para acreditar que o seu algoritmo está correto.

Mas, quais são as peças principais de um algoritmo?

Bom, o nosso palpite é que são: as variáveis auxiliares.

A seguir nós vamos ver alguns exemplos concretos.

2 Corretude do procedimento de intercalação

Abaixo nós temos o pseudo-código do procedimento de intercalação

(Você lembra como ele funciona?)

```
    Procedimento  Intercalação  ( V[1..n], W[1..m] )
     {
      // intercala os elementos das listas ordenadas V e W
      // e retorna o resultado na lista auxiliar A

  1.   i  <--  1     // índice para a lista V
  2.   j  <--  1     // índice para a lista W
  3.   k  <--  1     // índice para a lista A

  4.   Enquanto ( i <= n  e  j <= m )
  5     {
  6.       Se ( V[i] <= W[j] )
            {
  7.          A[k]  <--  V[i];   i++;   k++
            }
  8.       Senão
            {
  9.          A[k]  <--  W[j];   j++;   k++
        }   }

 10.   Enquanto ( i <= n )
        {
 11.       A[k]  <--  V[i];   i++;   k++
        }

 12.   Enquanto ( j <= m )
        {
 13.       A[k]  <--  W[j];   j++;   k++
        }

 14.   Retorna (A)
     }
```

Segundo o nosso palpite, as peças principais do procedimento são as variáveis `i`, `j`, `k`.

Certo.

Mas qual é a função dessas peças?

Bom,

- a variável `i` indica o <u>menor</u> elemento de `V` que ainda <u>não</u> foi intercalado (se houver) — senão ela indica que os elementos de `V` já foram todos intercalados
- a variável `j` indica o <u>menor</u> elemento de `W` que ainda <u>não</u> foi intercalado (se houver) — senão ela indica que os elementos de `W` já foram todos intercalados
- a variável `k` indica a primeira posição vazia de `A`

Legal.

A seguir, nós observamos que o algoritmo possui:

- um bloco onde as variáveis `i,j,k` são inicializadas
- e 3 ciclos:
  - o laço das linhas `4-9`
  - o laço das linhas `10-11`
  - o laço das linhas `12-13`

O papel da análise é mostrar que as peças principais permanecem nos lugares corretos durante toda a execução do algoritmo, de modo que tudo funciona bem.

Vejamos.

É bem fácil ver que tudo está em ordem no início:

```
i  <--  1
j  <--  1
k  <--  1
```

Quer dizer, examinando a função de cada variável, nós vemos que esses são os valores que elas devem ter antes da intercalação começar.

A seguir, nós vamos examinar o que acontece quando nós giramos a manivela a primeira vez (i.e., quando o algoritmo dá uma volta no primeiro laço).

- Existem basicamente dois casos, que correspondem aos dois resultados do teste

  ```
  Se ( V[i] <= W[j] )
  ```

  Na primeira volta, isso é a comparação entre `V[1]` e `W[1]`.

- Suponha que V[1] é o menor dos dois.

  Nesse caso, `V[1]` é copiado para a primeira posição de `A`

  ```
  A[k]  <--  V[i]
  ```

  pois $\mathtt{i} = \mathtt{k} = 1$.

  Em seguida, as variáveis `i` e `k` são atualizadas

  ```
  i++;   k++
  ```

  Nesse momento, nós temos $\mathtt{i} = 2$, $\mathtt{j} = 1$, $\mathtt{k} = 2$, e é fácil ver que isso está de acordo com as funções das 3 variáveis:

  - V[1] já foi intercalado, e portanto V[2] é o menor elemento não intercalado de V
  - W[1] ainda é o menor elemento não intercalado de W
  - A[2] é a primeira posição vazia de A

- O caso em que W[1] < V[1] é análogo — (Você quer tentar fazer?)

Legal.

É fácil fazer a mesma coisa para a segunda volta do laço.

E para a terceira também.

Mas, como é que nós podemos ter certeza de que mais cedo ou mais tarde as coisas não vão sair do lugar?

Para obter essa garantia, nós reproduzimos o argumento em um nível de generalidade maior.

Vejamos.

Suponha que, no início de uma volta qualquer do laço `4-9`, as variáveis $\mathtt{i}, \mathtt{j}, \mathtt{k}$ estão nos lugares corretos. Isto é,

- `i` indica o menor elemento de `V` que ainda não foi intercalado
- `j` indica o menor elemento de `W` que ainda não foi intercalado
- `k` indica a primeira posição vazia de `A`

- Durante a volta, existem basicamente dois casos, que correspondem aos resultados do teste

  ```
  Se ( V[i] <= W[j] )
  ```

- Suponha que V[i] é o menor dos dois.

  Nesse caso, `V[i]` é copiado para a primeira posição de `A`

  ```
  A[k]  <--  V[i]
  ```

Nesse momento,

- $\mathtt{V}[\mathtt{i}+1]$ é o menor elemento não intercalado de V
- W[j] ainda é o menor elemento não intercalado de W
- $\mathtt{A}[\mathtt{k}+1]$ é a primeira posição vazia de A

Portanto, quando as variáveis i e k são atualizadas

```
i++;   k++
```

as 3 variáveis i, j, k estão nos lugares corretos.

- O caso em que $\mathtt{W}[\mathtt{j}] < \mathtt{V}[\mathtt{i}]$ é análogo.

Muito bom.

Esse argumento mais geral nos garante que as voltas do laço 4-9 não tiram as variáveis i,j,k da sua posição correta.

Dessa maneira, o laço continua dando voltas e copiando os elementos para a lista auxiliar A, até que a sua condição de parada seja alcançada

```
i > n     ou     j > m
```

o que sinaliza que uma das listas já foi esgotada.

Aqui novamente nós temos 2 casos.

No caso em que $\mathtt{i} > \mathtt{n}$, o laço que vem em seguida não dá nenhuma volta

```
Enquanto ( i <= n )
```

E o último laço copia os elementos restantes de W para a lista auxiliar A

```
Enquanto ( j <= m )
 {
    A[k]  <--  W[j];   j++;   k++
 }
```

(Esse laço pode ser analisado como fizemos com o laço 4-9, caso se queira.)

No segundo caso, a situação se inverte.

Em resumo,

→ A análise de corretude concentra a atenção sobre as variáveis consideradas mais importantes do procedimento

(É possível fazer escolhas diferentes aqui, e colocar mais ou menos detalhe na análise.)

→ A seguir, nós vamos examinando cada trecho do código para se as variáveis preservam as suas propriedades durante a execução.

→ Uma atenção especial é dedicada aos laços, que requerem argumentos genéricos.

→ Finalmente, uma vez que a análise foi feita, em geral não é difícil ver que o algoritmo produz o resultado que se deseja.

A seguir nós vamos ver mais um exemplo.

3 Corretude do procedimento de partição

Abaixo nós temos o pseudo-código do procedimento de partição

(Você lembra como ele funciona?)

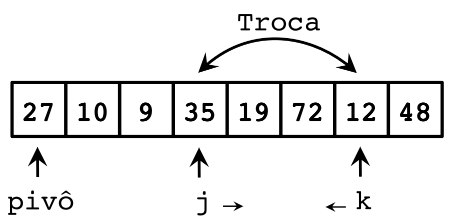
```
Procedimento  Partição  ( V[1..n] )
 {   // particiona a lista V utilizando o primeiro elemento como pivô

   pivô  <--  V[1]
   j  <--  2;    k  <--  n

   Enquanto ( j <= k )
    {
       Se ( V[j] > pivô  e  V[k] <= pivô )
          Troca (j,k)

       Se ( V[j] <= pivô )  j++
       Se ( V[k] >  pivô )  k--
    }
   Troca (1,k)
   Retorna (k)
 }
```

A ideia dessa implementação é encontrar elementos menores que o pivô no final da lista e elementos maiores que o pivô no início da lista, e trocá-los de posição.



Isso é feito utilizando duas variáveis auxiliares:

- a variável `j` indica a posição no <u>início</u> da lista que está sendo examinada
  — antes dela, todos os elementos são menores ou iguais ao pivô

- a variável `k` indica a posição no <u>final</u> da lista que está sendo examinada
  — depois dela, todos os elementos são maiores que o pivô

Além disso, quando `j` $>$ `k` o procedimento de partição está encerrado.

A variável auxiliar `pivô` não muda de valor durante toda a execução, por isso nós não precisamos nos preocupar com ela.

Como se pode ver, além do bloco inicial que inicializa as variáveis, o algoritmo é composto de um único laço.

O papel da análise é mostrar que as variáveis `j` e `k` preservam as suas propriedades ao longo da execução do algoritmo.

É bem fácil ver que no início as propriedades são satisfeitas

```
j  <--  2
k  <--  n
```

A seguir, nós examinamos o que acontece quando damos voltas no laço.

Suponha que no início de uma volta qualquer do laço, as propriedades de `j` e `k` estão valendo. Isto é,

- antes da posição `j` todos os elementos são menores ou iguais ao o pivô

- depois da posição `k` todos os elementos são maiores que o pivô

Durante a volta, existem basicamente dois casos, que correspondem aos resultados do teste

```
Se ( V[j] > pivô  e  V[k] <= pivô )
```

- Suponha primeiro que isso é o caso.

  Então os elementos indicados por `j` e `k` serão trocados de posição

  ```
  Troca (j,k)
  ```

  Após a troca, nós temos V[j] $\leq$ `pivô` e V[k] $>$ `pivô` .

  E isso significa que, em seguida, `j` será incrementado e `k` será decrementado

  ```
  . . .      j++
  . . .      k--
  ```

  Mas, isso quer dizer que, ao final dessa volta do laço, as propriedades de `j` e `k` continuam valendo.

- Agora suponha que a condição do teste não é válida.

  Então ao menos uma das condições abaixo é verdadeira (possivelmente ambas):

  ```
  V[j]  <=  pivô            V[k]  >  pivô
  ```

  No primeiro caso, a variável `j` é incrementada

  ```
  . . .      j++
  ```

  E no segundo caso, a variável `k` é decrementada

  ```
  . . .      k--
  ```

  Em qualquer caso é fácil ver que, ao final dessa volta do laço, as propriedades de `j` e `k` continuam valendo.

Certo.

O argumento acima mostra que as variáveis `j` e `k` preservam as suas propriedades ao longo da execução.

Mas, a análise também mostra uma outra coisa importante:

- A cada volta do laço, ao menos uma das variáveis `j`, `k` muda de valor (possivelmente as duas)

Essa última observação é importante pois ela mostra que as voltas do laço sempre realizam algum progresso, e eventualmente a condição de parada será satisfeita:

```
j > k
```

(Isto é, o laço não roda para sempre.)

Uma outra observação importante é que, ao final do laço, nós sempre temos:

```
j  =  k + 1
```

(porque?)

Sabendo que isso é verdade, não é difícil verificar que, após o laço

- `j` indica o primeiro elemento da lista que é maior que o pivô
- `k` indica o último elemento da lista que é menor ou igual ao pivô

E agora é fácil ver que a última instrução apenas posiciona o pivô (i.e., o elemento `V[1]`) entre as duas partições

```
Troca (1,k)
```