Construção e Análise de Algoritmos

aula 12: Corretude de algoritmos III

1 Introdução

Nas aulas 07 e 08, nós utilizamos uma analogia entre máquinas e algoritmos para introduzir a ideia dos argumentos de corretude.

Mas, e máquinas recursivas, existe isso?

Aqui, a nossa analogia parece que pára de funcionar.

Uma máquina pode até dividir a sua tarefa entre componentes mais simples.

Mas nenhum desses componentes é uma cópia da máquina ela mesma.

Com os algoritmos a coisa é diferente.

Um algoritmo pode fazer uma chamada (recursiva) a ele mesmo.

E essa chamada recursiva pode fazer uma nova chamada recursiva ao mesmo algoritmo.

Esse é o comportamento típico dos algoritmos de divisão e conquista.

Como é que nós podemos demonstrar a corretude de um algoritmo desse tipo?

2 Raciocinando sobre a árvore de recursão

Como sempre, a análise de um algoritmo de divisão e conquista é feita com base na sua árvore de recursão.

Abaixo nós temos a árvore de recursão para o caso em que o problema é quebrado na metade, apenas como um exemplo ilustrativo.

`< Figura: árvore de recursão de DC-Gen() >`

A intuição para o argumento de corretude é semelhante àquela que foi utilizada para os algoritmos estruturados:

- Se todos os componentes realizam o seu trabalho corretamente, então o algoritmo funciona corretamente.

Ou seja, para demonstrar a corretude do algoritmo basta mostrar que todas as chamadas recursivas da árvore funcionam corretamente.

E isso não é tão difícil assim, pois todas elas estão fazendo a mesma coisa — todas são chamadas ao procedimento `DC-Gen()`.

Examinando as coisas com um pouco mais de cuidado, nós fazemos a seguinte distinção:

- na base da árvore estão as chamadas que recebem instâncias triviais do problema, que são resolvidas pelo procedimento `Resolve()`
- no interior da árvore estão as chamadas que quebram o seu problema, realizam chamadas recursivas, e combinam as soluções recebidas

A ideia, então, é desmonstrar a corretude desses dois casos.

O primeiro deles em geral é simples, pois o procedimento `Resolve()` sempre trabalha com uma instância trivial do problema.

Para o segundo caso, é útil examinar o trecho de pseudo-código correspondente:

```
Procedimento  DC-Gen ( P )
 {
     ( . . . )

4.   (P1,P2)  <--  Quebra (P)
5.        S1  <--  DC-Gen (P1)
6.        S2  <--  DC-Gen (P2)
7.         S  <--  Combina (S1,S2)
8.   Retorna (S)
```

Aqui, mais uma vez, a ideia é fazer a análise baseada em componentes.

Isto é, se todos os componentes desse código funcionarem corretamente, então o procedimento funcionará corretamente.

E aqui nós observamos que será necessário demonstrar a corretude dos procedimentos `Quebra()` e `Combina()`.

Mas, o que dizer das linhas `5` e `6`?

Isto é, aqui nós precisamos demonstrar a corretude das chamadas a `DC-Gen()`, mas isso é precisamente o que nós estamos tentando fazer ...

Como resolver isso?

A solução consiste mais uma vez em examinar a árvore de recursão do algoritmo.

E observar que, no caso das chamadas na base da árvore a corretude já foi demonstrada!

`< Figura: árvore de recursão com a base demonstrada >`

Isso significa que nós já podemos demonstrar a corretude das chamadas no penúltimo nível da árvore.

Isto é, as chamadas do penúltimo nível fazem as chamadas do último nível, que nós já sabemos que funcionam corretamente.

E agora basta mostrar que o procedimento `DC-Gen` funciona corretamente sabendo que todos os seus componentes (i.e., as chamadas das linhas `4,5,6,7`) funcionam corretamente.

Suponha que isso já foi feito.

Então, o próximo passo é demonstrar a corretude das chamadas no ante-penúltimo nível.

Na verdade, não!

Isto é, não há mais nada a fazer.

Vejamos.

- Suponha que a corretude dos procedimentos `Resolve()`, `Quebra()` e `Combina()` já foi demonstrada
- Isso significa, em particular, que as chamadas a `DC-Gen()` na base da árvore funcionam corretamente.
- A seguir, suponha que nós conseguimos demonstrar que a corretude das chamadas `4-7` implicam a corretude da chamada a `DC-Gen()`
- Então, com base nisso, nós já podemos concluir que as chamadas a `DC-Gen()` no penúltimo nível da árvore funcionam corretamente.
- Mas, se isso é o caso, então as chamadas no ante-penúltimo nível também funcionam corretamente, pois seus componentes `4-7` funcionam corretamente
- E, então, as chamadas no ante-ante-penúltimo nível também funcionam
- A coisa é como um dominó ...

3 Corretude do algoritmo Mergesort

A discussão da seção anterior nos deu uma receita para demonstrar a corretude de qualquer algoritmo de divisão e conquista

→ Demonstrar a corretude dos procedimentos `Resolve()`, `Quebra()` e `Combina()`

→ Argumentar que a corretude das chamadas recursivas (juntamente com a corretude dos procedimentos de quebra e combinação) são suficientes para garantir a corretude do algoritmo

E, a seguir, nós vamos colocar essa receita em prática na análise do algoritmo Mergesort.

```
 Procedimento  Mergesort ( V[i..j] )
  {
1.    Se ( i = j )  Retorna

2.    k  <--  (i+j)/2

3.    Mergesort ( V[i..k] )
4.    Mergesort ( V[k+1..j] )

5.    Intercalação ( V[i..k], V[k+1..j] )
  }
```

Nesse caso, o correspondente ao procedimento `Resolve()` não faz nada, logo ele não pode estar errado.

O correspondente ao procedimento `Quebra()` também é trivial, e não há nada a fazer aqui também.

Finalmente, a corretude do procedimento de intercalação (que corresponde ao procedimento `Combina()`) já foi demonstrada na aula 06.

Portanto, a única coisa a fazer é argumentar que a corretude das chamadas recursivas nas linhas `3` e `4` é suficiente para garantir a corretude do algoritmo.

Nesse caso, a corretude dessas chamadas significa que, quando nós chegarmos na linha `5`, as porções `V[i..k]` e `V[k+1..j]` estarão ordenadas.

E agora é fácil, pois a corretude do procedimento `Intercalação()` garante que, após a linha `5`, a porção `V[i..j]` da lista estará ordenada.

Como é isso o que a chamada `Mergesort (V[i..j])` precisa fazer, a corretude do algoritmo está demonstrada.

Observação: A primeira vista, nós não fizemos muita coisa aqui: após a demonstração da corretude dos procedimentos de quebra, combinação, e resolução das instâncias triviais, bastou um argumento simples para obter a corretude do algoritmo.

Mas, é isso mesmo; esse é o espírito do método de divisão e conquista.

Lembre que, na etapa de construção, uma vez que nós descobrimos como quebrar o problema e combinar as soluções dos subproblemas, o problema estava essencialmente resolvido.

4 Corretude do algoritmo Quicksort

A seguir, nós vamos aplicar a receita do argumento de corretude ao algoritmo Quicksort.

```
 Procedimento  Quicksort ( V[i..j] )
  {
1.    Se ( i = j )  Retorna

2.    k  <--  Partição ( V[i..j] )

3.    Quicksort ( V[i..k-1] )
4.    Quicksort ( V[k+1..j] )
  }
```

Nesse caso, o correspondente ao procedimento `Resolve()` não faz nada, logo ele não pode estar errado.

E o correspondente ao procedimento `Combina()` também não faz nada, logo ele não pode estar errado.

Finalmente, a corretude do procedimento de partição (que corresponde ao procedimento `Combina()`) já foi demonstrada na aula 06.

Portanto, a única coisa a fazer é argumentar que a corretude das chamadas recursivas nas linhas `3` e `4` é suficiente para garantir a corretude do algoritmo.

Nesse caso, a corretude dessas chamadas significa que, após as linhas `3` e `4`, as porções `V[i..k-1]` e `V[k+1..j]` estarão ordenadas.

Ora, mas a corretude do procedimento `Partição()` implica que

- todos os elementos da porção `V[i..k-1]` são menores ou iguais ao elemento `V[k]`
- o elemento `V[k]` é menor do que todos os elementos na porção `V[k+1..j]`

Portanto, após a execução das linhas `3` e `4`, a porção `V[i..j]` da lista estará completamente ordenada.

Como é isso o que a chamada `Quicksort (V[i..j])` precisa fazer, a corretude do algoritmo está demonstrada.