Projeto e Análise de Algoritmos

Indução (parte 2)

Flávio L. C. de Moura*

21 de outubro de 2024

1 A correção de algoritmos

Considere a seguinte função recursiva:

$$f(n) = \begin{cases} 0, & \text{se } n = 0 \\ 2n + f(n-1) - 1, & \text{se } n > 0 \end{cases} \tag{1}$$

e observe que $f(0) = 0$, $f(1) = 1$, $f(2) = 4$, $f(3) = 9$ e $f(4) = 16$. A partir do que observamos para estes valores iniciais, podemos conjecturar que $f(n) = n^2, \forall n \geq 0$. Utilizaremos indução como estratégia de prova para esta conjectura, já que indução constitui o caminho natural para provar propriedades de funções recursivas.

- (base da indução): Se $n = 0$ então $f(0) = 0 = 0^2$.
- (passo indutivo): Se $n > 0$ então $f(n) \overset{def.}{=} 2n + f(n-1) - 1 \overset{h.i.}{=} 2n + (n-1)^2 - 1 = n^2$.

Portanto, podemos concluir que $f(n) = n^2, \forall n \geq 0$. Ou seja, $f(n) = n^2, \forall n \geq 0$ não é mais uma conjectura, mas um teorema.

No paradigma funcional de programação, algoritmos são essencialmente funções. Neste sentido, acabamos de provar que o algoritmo (1) computa **sempre** o quadrado do seu argumento.

Considere agora uma versão recursiva do algoritmo de ordenação por inserção utilizando a estrutura de listas. A etapa principal deste algoritmo consiste na inserção de um elemento x em uma lista ordenada l. A função *insert* x l definida a seguir, insere o elemento x na lista l:

$$\mathit{insert}\ x\ l = \begin{cases} x :: nil, & \text{se } l = nil \\ x :: l, & \text{se } x \leq h \text{ e } l = h :: tl \\ h :: (\mathit{insert}\ x\ tl), & \text{se } x > h \text{ e } l = h :: tl \end{cases} \tag{2}$$

A inserção é feita de forma que se l está ordenada então a lista resultante também está ordenada.

Exercício 1.1. *Simule a execução de insert* $2\ (1 :: 3 :: 5 :: nil)$.

Exercício 1.2. *Mostre que se l é uma lista ordenada e x é um inteiro, então ($insert\ x\ l$) é uma lista ordenada.*

O algoritmo de ordenação por inserção recursivo é dado pela seguinte função recursiva:

*flaviomoura@unb.br

$$insertion\_sort\ l := \begin{cases} l, & \text{se } l = nil \\ insert\ x\ (insertion\_sort\ tl), & \text{se } l = h :: tl \end{cases} \quad (3)$$

Exercício 1.3. *Simule a execução de insertion_sort* $(10 :: 3 :: 5 :: 1 :: nil)$.

Agora, queremos concluir que o algoritmo *insertion_sort* é correto, mas o que isto significa?

Exercício 1.4. *Escreva um enunciado, análogo ao do Exercício 1.2, que caracterize a correção do algoritmo insertion_sort. Em seguida prove que o algoritmo insertion_sort satisfaz a propriedade que você enunciou, e conclua que insertion_sort é correto.*

Considere a versão iterativa do algoritmo de ordenação por inserção utilizando vetores [1, 2]. Neste caso, queremos ordenar $n > 0$ números naturais em ordem crescente, e para isto vamos supor que estes números estão armazenados no vetor $A[0..n-1]$. Ao final do processo queremos obter uma permutação de $A[0..n-1]$, digamos $A'[0..n-1]$ tal que $A'[i-1] \leq A'[i]$, para todo $1 \leq i < n$:

Algoritmo 1: InsertionSort($A[1..n]$)

```
1 for i = 2 to n do
2     key ← A[i];
3     j ← i − 1;
4     while j > 0 and A[j] > key do
5         A[j + 1] ← A[j];
6         j ← j − 1;
7     end
8     A[j + 1] ← key;
9 end
```

Queremos mostrar que o algoritmo InsertionSort é correto. Isto corresponde a mostrar que o resultado da execução do laço **for** (linhas 1-9) retorna o vetor $A[1..n]$ ordenado. Podemos utilizar indução no número de vezes que o laço é executado. A propriedade a ser provada deve expressar as relações existentes entre as variáveis do programa, e como esta propriedade deve ser satisfeita ao longo de toda a execução do laço, ela é normalmente chamada de **invariante de laço**. No caso de InsertionSort, a cada execução do laço **for** devemos inserir um novo elemento em um subvetor ordenado. Mais precisamente, temos a seguinte invariante:

Antes de cada iteração do laço **for** (linhas 1-9) indexado por i, o subvetor $A[1..(i-1)]$ está ordenado e contém os mesmos elementos do vetor original $A[1..(i-1)]$.

A prova de uma invariante é dividida em três passos:

1. (**Inicialização**) A invariante é verdadeira antes da primeira iteração do laço;

2. (**Manutenção**) Se a invariante é verdadeira antes de uma iteração do laço, então ela continua verdadeira antes da próxima iteração;

3. (**Terminação**) Ao término da execução do laço, a invariante implica na correção do algoritmo.

O passo de inicialização para a invariante acima é trivial porque antes da primeira iteração o vetor $A[1..n]$ ainda não foi modificado, e temos $i = 2$. Portanto o subvetor $A[1..(i-1)] = A[1]$ está ordenado, e contém o mesmo elemento do vetor original.

O passo de manutenção é provado da seguinte forma: Suponha que a invariante seja verdadeira antes de uma iteração arbitrária, digamos quando $i = k$, onde $2 \leq k \leq n$. Precisamos provar que a invariante continua verdadeira antes da próxima iteração, ou seja, quando $i = k+1$. Quando $i = k$, estamos assumindo que o subvetor $A[1..(k-1)]$ está ordenado e contém os mesmos elementos do vetor original $A[1..(k-1)]$. Na linha 2, o elemento $A[k]$ é armazenado na variável auxiliar key, e o laço **while** (linhas

4-7) move os elementos $A[k-1]$, $A[k-2]$, ... que são estritamente maiores do que $A[k]$ uma posição para a direita. Quando um elemento menor ou igual a $A[k]$ é encontrado ou chegamos na primeira posição do vetor, ou seja, quando as condições da linha 4 não são satisfeitas, então encontramos a posição correta para inserir o elemento $A[k]$ (que está armazenado na variável *key*). A inserção é feita na linha 8. Portanto, o subvetor $A[1..k]$ está ordenado e contém os mesmos elementos do vetor original $A[1..k]$, e a invariante está preservada.

O passo de terminação nos permite concluir a correção do algoritmo. Observe que ao final do laço **for**, temos que $i = n+1$, e neste caso a invariante nos garante que o "*subvetor* $A[1..n]$ *está ordenado e contém os mesmos elementos do vetor original* $A[1..n]$". Em outras palavras, isto significa que InsertionSort é correto.

Observe que na prova do passo de manutenção acima, afirmamos que "*o laço **while** (linhas 4-7) move os elementos $A[k-1]$, $A[k-2]$, ... que são estritamente maiores do que $A[k]$ uma posição para a direita. Quando um elemento menor ou igual a $A[k]$ é encontrado ou chegamos na primeira posição do vetor, ou seja, quando as condições da linha 4 não são satisfeitas, então encontramos a posição correta para inserir o elemento $A[k]$*". Assim, podemos dizer que o laço **while** é responsável por encontrar a posição correta para inserir o elemento $A[k]$. Podemos provar que a posição encontrada pelo laço **while** é, de fato, correta por meio de outra invariante de laço:

> Antes de cada iteração do laço **while** (linhas 4-7), o subvetor $A[(j+1)..i]$ possui elementos maiores ou iguais a *key*.

Exercício 1.5. *Prove a invariante de laço para o **while**, e conclua que o laço **while** retorna a posição correta para inserir key.*

Exercício 1.6. *Considere o seguinte pseudocódigo:*

Algoritmo 2: dec_to_bin(n)

```
t ← n;
k ← 0;
while  t > 0 do
    k ← k + 1;
    b[k] ← t mod 2;
    t ← t div 2;
end
return b;
```

O algoritmo dec_to_bin(n) recebe o número decimal n como argumento e retorna o vetor b contendo a representação binária de n. A variável k indica uma posição do vetor, t mod 2 retorna o resto da divisão de t por 2 e t div 2 retorna o quociente da divisão de t por 2. Mostre que o algoritmo dec_to_bin(n) é correto. Para isto, prove a seguinte invariante:

> *Se a representação binária do inteiro m é dada pelo vetor $b[1..k]$, então $n = t.2^k + m$.*

Exercício 1.7. *Prove que o algoritmo BubbleSort é correto.*

Algoritmo 3: BubbleSort($A[0..n-1]$)

```
for i = 0 to n − 2 do
    for j = 0 to n − 2 − i do
        if A[j + 1] < A[j] then
            swap A[j] and A[j + 1];
        end
    end
end
```

Exercício 1.8. *Prove que o algoritmo BubbleSort2 é correto.*

Algoritmo 4: BubbleSort2($A[0..n-1]$)

```
for i = 0 to n − 2 do
    for j = n − 1 downto i + 1 do
        if A[j] < A[j − 1] then
            swap A[j] and A[j − 1];
        end
    end
end
```

Exercício 1.9. *Prove que o algoritmo SelectionSort é correto.*

Algoritmo 5: SelectionSort($A[0..n-1]$)

```
for i = 0 to n − 2 do
    min ← i;
    for j = i + 1 to n − 1 do
        if A[j] < A[min] then
            min ← j;
        end
    end
    swap A[i] and A[min];
end
```

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[2] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.