Projeto e Análise de Algoritmos (2025-1)

Flávio L. C. de Moura*

3 de abril de 2025

Ordenação por inserção

Nesta seção estudaremos o algoritmo *insertion sort* (ordenação por inserção), que tem como etapa principal inserir um elemento em um vetor ordenado. Assim, 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 $1 \leq i < n$. O algoritmo de ordenação por inserção (*insertion sort*) é dado pelo pseudocódigo a seguir:

Algorithm 1: InsertionSort($A[0..n-1]$)

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

A primeira pergunta que precisamos responder é: este algoritmo é correto? Isto é, ele satisfaz as especificações do problema que propõe resolver?

A correção do algoritmo de ordenação por inserção

Em algoritmos iterativos utilizamos as invariantes de laço para estabelecer a correção do algoritmo. Dada a dinâmica do algoritmo InsertionSort, considere a seguinte invariante de laço:

> Antes da j-ésima iteração do laço **for** (linhas 1-9), o subvetor $A[0..j-1]$ está ordenado e contém

*flaviomoura@unb.br

os mesmos elementos do vetor original $A[0..j-1]$.

Assim, se esta propriedade for válida ao final da execução do laço **for**, *i.e.* antes da $n+1$-ésima iteração, teremos que o vetor gerado consiste dos elementos do vetor original $A[0..n-1]$ ordenado. Isto corresponde a dizer que InsertionSort é correto.

Como então provar esta invariante para InsertionSort? A prova é por indução no número de iterações do laço **for**:

- **Inicialização** (Base da indução):
  Antes da primeira iteração do laço **for**, temos que $j = 1$ (condição necessária para iniciar o laço), e portanto a invariante é trivial porque o subvetor unitário $A[0]$ está ordenado por definição.

- **Manutenção** (Passo indutivo):
  Considere a k-ésima iteração, isto é, $j = k$ $(1 < k < n)$. Temos como hipótese que "Antes da k-ésima iteração do laço **for** o subvetor $A[0..k-1]$ é uma permutação que está ordenada do subvetor original $A[0..k-1]$."Assim, durante a k-ésima iteração, o laço **while** vai deslocar cada elemento maior do que $A[k]$, i.e. key, uma posição para a direita até encontrar a posição correta onde o elemento $A[k]$ deve ser inserido, de forma que neste momento o subvetor $A[0..k]$ está ordenado e possui os mesmos elementos do subvetor $A[0..k]$ original. A incrementarmos o valor de k para a próxima iteração, a invariante é reestabelecida. Informalmente estamos dizendo que o laço **while** encontra a posição correta para inserir $A[j]$ (que está armazenado na variável key). Provaremos este fato com a ajuda de uma invariante para o laço **while**:

  Antes de cada iteração do laço **while**, o subvetor $A[i+1..j]$ possui elementos que são maiores ou iguais a key.

  A prova é também por indução no número de iterações do laço **while**:

  1. **Inicialização**: Antes da primeira iteração do **while** temos que $i+1 = j = k$, e como $key = A[j]$ a invariante está satisfeita.

  2. **Manutenção**: Por hipótese de indução temos que o subvetor $A[i+1..j]$ possui elementos que são maiores ou iguais a key. Durante uma iteração do laço, o elemento $A[i]$ é copiado na posição $i+1$ do vetor A, e portanto a invariante continua valendo.

  3. **Finalização**: Ao final da execução do laço, temos que i é, de fato, a posição correta para inserir o elemento $A[k]$ já que todos os elementos do subvetor $A[i+1..j]$ são maiores ou iguais a key. É importante observar que a inserção do elemento $A[k]$ na posição i não elimina nenhum elemento do vetor original porque o elemento que está na posição i foi

copiado para a posição $i+1$, se o laço **while** foi executado pelo menos uma vez, ou ele é o próprio elemento armazenado em key, quando o laço não é executado.

- **Finalização**: Ao final da execução do laço **for**, temos $j = n$, e portanto a invariante corresponde a dizer que o vetor $A[0..n-1]$ obtido ao final da execução do algoritmo está ordenado, e é uma permutação do vetor original $A[0..n-1]$. Assim, concluímos a prova da correção do algoritmo InsertionSort.

Exercício 0.1. *Prove que o algoritmo BubbleSort a seguir é correto.*

Algorithm 2: BubbleSort($A[0..n-1]$)

```
1 for i = 0 to n − 2 do
2 |   for j = 0 to n − 2 − i do
3 |   |   if A[j + 1] < A[j] then
4 |   |   |   swap A[j] and A[j + 1];
5 |   |   end
6 |   end
7 end
```

Exercício 0.2. *Prove que o algoritmo BubbleSort2 [1] a seguir é correto.*

Algorithm 3: BubbleSort2($A[0..n-1]$)

```
1 for i = 0 to n − 2 do
2 |   for j = n − 1 downto i + 1 do
3 |   |   if A[j] < A[j − 1] then
4 |   |   |   swap A[j] and A[j − 1];
5 |   |   end
6 |   end
7 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.

Ordenação por inserção recursivo

Nesta seção estudaremos o algoritmo de ordenação por inserção recursivo. A estrutura de dados utilizada é a de listas, e para simplificar trabalharemos com números naturais, mas as ideias são as mesmas para ordenarmos qualquer estrutura que possua uma ordem total. Vimos no capítulo anterior que as listas de naturais possuem dois construtores: *nil* para representar a lista vazia, e o operador :: que nos permite construir uma nova lista a partir de um número natural e de uma

lista já construída. Assim, a lista unitária contendo apenas o natural 5 é representada por $5 :: nil$, enquanto que a lista $1 :: (5 :: nil)$, ou simplesmente $1 :: 5 :: nil$, representa a lista que tem 1 como primeiro elemento, e a lista $5 :: nil$ como cauda.

A operação que dá nome ao algoritmo é a operação de inserção porque a cada passo queremos inserir um novo elemento em uma lista já ordenada. Suponha, por exemplo, que queiramos inserir o número 3 na lista $1 :: 5 :: nil$, isto é, o nosso objetivo final é obter a lista ordenada $1 :: 3 :: 5 :: nil$. Para isto, precisamos inicialmente comparar o 3 com o primeiro elemento da lista, e o resultado desta comparação nos diz que o 3 deve ser inserido depois do 1, ou seja, em algum lugar da cauda $5 :: nil$. Em seguida, comparamos o 3 com o primeiro elemento da cauda, ou seja, com 5, e como $3 < 5$, sabemos que ele deve ser inserido antes do 5. Esta ideia está implementada na função *ins* definida a seguir:

Definição 0.3. *Sejam x um número natural, e l uma lista de números naturais. A função $(ins\ x\ l)$, que insere o natural x na lista l, é definida recursivamente como a seguir:*

$$ins\ x\ l = \begin{cases} x :: nil, & se\ l = nil \\ x :: l, & se\ l = h :: tl\ e\ x \leq h \\ h :: (ins\ x\ tl), & se\ l = h :: tl\ e\ x > h \end{cases}$$

O algoritmo de ordenação por inserção então consiste em recursivamente, dada uma lista não vazia $h :: tl$, inserir o primeiro elemento h na versão ordenada da cauda tl. Ou seja, o algoritmo de ordenação por inserção que será implementado pela função *is* (*insertion sort*) que recebe uma lista l como argumento. Se l for a lista vazia não há nada a fazer, e caso contrário, recursivamente ordenamos a cauda da lista para então inserir o novo elemento:

Definição 0.4. *Seja l uma lista de números naturais. A função is é definida recursivamente como a seguir:*

$$is\ l = \begin{cases} nil, & se\ l = nil \\ ins\ h\ (is\ tl), & se\ l = h :: tl \end{cases}$$

Vejamos como este algoritmo funciona na prática. Suponha que queiramos ordenar a lista $3 :: 2 :: 1 :: nil$. Ao fornecermos esta lista como argumento para a função *is*, temos:

$$\begin{array}{ll}
is\ (3 :: 2 :: 1 :: nil) = & (\text{def. } is) \\
ins\ 3\ (is\ (2 :: 1 :: nil)) = & (\text{def. } is) \\
ins\ 3\ (ins\ 2\ (is\ (1 :: nil))) = & (\text{def. } is) \\
ins\ 3\ (ins\ 2\ (ins\ 1\ (is\ nil))) = & (\text{def. } is) \\
ins\ 3\ (ins\ 2\ (ins\ 1\ nil)) = & (\text{def. } ins) \\
ins\ 3\ (ins\ 2\ (1 :: nil)) = & (\text{def. } ins) \\
ins\ 3\ (1 :: (ins\ 2\ nil)) = & (\text{def. } ins) \\
ins\ 3\ (1 :: 2 :: nil) = & (\text{def. } ins) \\
1 :: (ins\ 3\ (2 :: nil)) = & (\text{def. } ins) \\
1 :: 2 :: (ins\ 3\ nil) = & (\text{def. } ins) \\
1 :: 2 :: 3 :: nil &
\end{array}$$

Veja que o algoritmo ordenou corretamente a lista $3 :: 2 :: 1 :: nil$, mas será que ele ordena corretamente qualquer lista de números naturais? Para responder esta pergunta, vamos analisar se o algoritmo é correto ou não.

A correção do algoritmo de ordenação por inserção

Nesta seção vamos provar que o algoritmo de ordenação por inserção apresentado na seção anterior é correto. Para isto precisaremos definir algumas noções que serão utilizadas também em outros algoritmos de ordenação. A primeira noção que precisamos definir formalmente é a de ordenação. Ou seja, o que significa dizer que uma lista está ordenada? A definição a seguir apresenta o predicado *sorted* que caracteriza a noção de lista ordenada:

Definição 0.5. *Sejam x e y números naturais, e l uma lista de números naturais. O predicado sorted, que caracteriza o fato de uma lista estar ordenada, é definido por meio das seguintes regras de inferência:*

$$\frac{}{sorted\ nil}\ (sorted\_nil) \qquad \frac{}{sorted\ x :: nil}\ (sorted\_one)$$

$$\frac{x \leq y \qquad sorted\ y :: l}{sorted\ x :: y :: l}\ (sorted\_all)$$

A regra $(sorted\_nil)$ é um axioma que estabelece que a lista vazia está ordenada. A regra $(sorted\_one)$ também é um axioma que estabelece que listas unitárias estão ordenadas. A regra $(sorted\_all)$ possui duas condições para que uma lista da forma $x :: y :: l$ esteja ordenada: $x \leq y$ e a lista $y :: l$ tem que estar ordenada. Em outras palavras, a regra $(sorted\_all)$ diz que uma lista com pelo menos dois elementos está ordenada, se o primeiro elemento é menor ou igual ao segundo elemento, e a cauda da lista (ou seja, a lista do segundo elemento em diante) está ordenada. Note que as variáveis x, y e a lista l estão implicitamente quantificadas universalmente na Definição 0.5. Segundo esta definição, a lista $(1 :: 2 :: 3 :: nil)$ está ordenada. De fato, a prova deste fato é dada pela seguinte árvore de derivação:

$$\dfrac{1 \leq 2 \qquad \dfrac{2 \leq 3 \qquad \dfrac{}{sorted\ (3 :: nil)}\ (sorted\_one)}{sorted\ (2 :: 3 :: nil)}\ (sorted\_all)}{sorted\ (1 :: 2 :: 3 :: nil)}\ (sorted\_all)$$

Com esta definição em mãos, podemos provar uma propriedade da função *ins* que ficou implícita:

Lema 0.6. *Sejam x um número natural, e l uma lista de números naturais. Se l está ordenada então $(ins\ x\ l)$ também está ordenada.*

Demonstração. A prova é por indução na estrutura da lista l. Se l for a lista vazia então $(ins\ x\ l)$ é a lista unitária $x :: nil$ que está ordenada por definição (regra $sorted\_one$). Se l é da forma $h :: tl$ então temos dois casos a considerar:

- $x \leq h$: Neste caso, $ins\ x\ (h :: tl)$ retorna a lista $x :: h :: tl$ que está ordenada já que $x \leq h$ e, por hipótese, a lista $h :: tl$ está ordenada:

$$\frac{x \leq h \qquad \dfrac{}{sorted\ h :: tl}\ (hip.)}{sorted\ x :: h :: tl}\ (sorted\_all)$$

- $x > h$: Neste caso, x será inserido na cauda tl, e por hipótese de indução temos que a lista $(ins\ x\ tl)$ está ordenada. Como a lista $h :: tl$ está ordenada, então h é menor ou igual a todo elemento de tl. Logo h é menor ou igual que todo elemento da lista $(ins\ x\ tl)$, e portanto a lista $h :: (ins\ x\ tl)$ está ordenada.

□

Nosso próximo passo é provar que o algoritmo de ordenação por inserção efetivamente ordena qualquer lista de naturais dada como entrada:

Lema 0.7. *O algoritmo de ordenação por inserção da Definição 0.4 ao receber uma lista l de números naturais como argumento retorna uma lista ordenada. Em outras palavras, a lista $(is\ l)$ está ordenada, para qualquer lista l.*

Demonstração. A prova é por indução na estrutura da lista l. Se l é a lista vazia (base de indução) então, por definição temos que $is\ nil = nil$, e não há o que fazer porque a lista vazia está ordenada. No passo indutivo suponha que l tem a forma $h :: tl$. Temos $is\ (h :: tl) = insh(is\ tl)$, e por hipótese de indução temos que a lista $(is\ tl)$. Então, pelo Lema 0.6 concluímos que a lista $insh(is\ tl)$ está ordenada, e portanto $is\ (h :: tl)$ está ordenada.

□

A segunda parte da prova da correção de um algoritmo de ordenação consiste em mostrar que o algoritmo retorna uma lista que é uma permutação da lista de entrada. Assim, um algoritmo de ordenação será correto se, para qualquer lista l dada como entrada, a saída for uma permutação de l que esteja ordenada. Ou seja, a resposta do algoritmo tem que ser uma lista que contém exatamente os mesmos elementos da lista de entrada e que adicionalmente esteja ordenada.

Como então definir a noção de permutação? Temos pelo menos duas opções. A primeira é simplesmente contar o número de ocorrências de cada elemento e ver que qualquer elemento tem que ocorrer o mesmo número de vezes nas duas listas para que uma seja uma permutação da outra. De maneira mais precisa, podemos definir o número de ocorrências de x na lista l, notação $num\_oc\ x\ l$ da seguinte forma:

Definição 0.8. *Seja x um número natural, e l uma lista de números naturais. Definimos recursivamente o número de ocorrências de x em l por:*

$$num\_oc\ x\ l = \begin{cases} 0, & se\ l = nil \\ 1 + num\_oc\ x\ tl, & se\ l = x :: tl \\ num\_oc\ x\ tl, & caso\ contrário. \end{cases}$$

O predicado *perm*, que define quando duas lista, digamos l e l' são permutações uma da outra.

Definição 0.9. *Sejam l e l' listas de números naturais. Definimos o predicado perm em função de num_oc por perm l $l' := \forall x, num\_oc\ x\ l = num\_oc\ x\ l'$.*

De acordo com esta definição, a lista l' é uma permutação da lista l se o número de ocorrências de x em l é igual ao número de ocorrências de x em l'. Nosso objetivo agora é mostrar que o algoritmo de ordenação por inserção gera uma lista que é uma permutação da lista de entrada, ou seja, queremos provar o seguinte teorema:

Teorema 0.10. *Seja l uma lista de números naturais. O algoritmo de ordenação por inserção gera como saída uma lista que é permutação da lista de entrada, ou seja, o sequente $\vdash perm\ l\ (is\ l)$ é válido.*

Demonstração. A prova é por indução na estrutura da lista l. Quando l é a lista vazia (base da indução), temos que $num\_oc\ x\ (is\ nil) = num\_oc\ x\ nil$ para todo x, ou seja, nil é uma permutação de $(is\ nil)$. Suponha que l tenha a forma $h :: tl$ (passo indutivo). Precisamos provar que $(h :: tl)$ é uma permutação da lista $(is\ (h :: tl))$, que pela definição de is é igual a $(ins\ h\ (is\ tl))$. Por hipótese de indução temos que tl é uma permutação da lista $(is\ tl)$. Considerando que a função $(ins\ h\ (is\ tl))$ apenas adiciona o elemento h à lista $(is\ tl)$, concluímos que a lista $(h :: tl)$ é uma permutação da lista $ins\ h\ (is\ tl)$, que por sua vez é igual a $(is\ (h :: tl))$, como queríamos demonstrar.

□