Capítulo 2

O algoritmo de ordenação por inserção

2.1 A correção do algoritmo de ordenação por inserção

Iniciaremos esta seção com o estudo de um algoritmo de ordenação bem simples: o algoritmo de ordenação por inserção. Queremos ordenar $n > 0$ números naturais, que se encontram armazenados no vetor $A = \langle a_1, a_2, \ldots, a_n \rangle$, em ordem crescente de acordo com o seguinte pseudocódigo:

Algorithm 1: InsertionSort(A)

```
1 for j = 2 to A.length 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
```

Existem dois aspectos básicos que precisam ser considerados no estudo dos algoritmos: correção e complexidade. Assim, a primeira pergunta que precisamos responder é: Este algoritmo é correto? Isto é, ele satisfaz as especificações em relação ao problema que propõe resolver? Neste caso, queremos ordenar os elementos de um vetor, e assim esperamos que os elementos do vetor gerado após a execução do algoritmo coincidam com os elementos do vetor original, e além disto estejam ordenados de forma crescente. O princípio da indução é bastante útil para provarmos a correção de algoritmos. Em algoritmos com laços, como o acima, a ideia é usar indução sobre o número de vezes que o laço é executado. A hipótese de indução deve expressar as relações existentes entre as variáveis durante a execução do laço, ou seja, propriedades que sejam válidas durante a execução do laço e que nos permitam estabelecer a correção do algoritmo após a execução do laço. Estas hipóteses de indução são chamadas de *invariantes*

de laço. Por exemplo, dada a dinâmica do algoritmo InsertionSort, considere a seguinte invariante de laço:

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

Assim, se esta propriedade for verdadeira, ao final da execução de InsertionSort para um vetor A contendo n elementos, *i.e.* antes da $n+1$-ésima iteração, teremos que o vetor gerado consiste dos elementos do vetor original A ordenados crescentemente, o que corresponde a dizer que InsertionSort é correto.

Em geral, a prova de uma invariante é dividida em três etapas:

1. **Inicialização**: Consiste em mostrar que a invariante é verdadeira antes da primeira iteração do laço;
2. **Manutenção**: Consiste em mostrar que, se a invariante é verdadeira antes de uma iteração do laço então ela continua verdadeira antes da próxima iteração;
3. **Finalização**: Quando o laço termina, a invariante nos fornece uma propriedade que nos permite concluir que o algoritmo é correto.

Observe que o laço **for** contém um laço interno (*while*) que decide em qual posição do subvetor $A[1..j]$ que o elemento *key* deve ser inserido. Para garantirmos que a inserção foi feita na posição correta, e que portanto após a execução da k-ésima iteração do laço **for**, o subvetor $A[1..k]$ está ordenado precisamos provar a seguinte invariante para o laço *while*:

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

Dividiremos a prova em três etapas como informado acima:

1. **Inicialização**: Antes da primeira iteração do laço *while*, temos que $i = j-1$ e $A[i] > key$ pelo teste do laço, e também $A[j] = key$. Logo, a invariante é verdadeira.
2. **Manutenção**: A invariante continua verdadeira durante a execução do laço porque o elemento $A[i]$ é deslocado uma posição à direita, e sabemos que $A[i] > key$ ou seja, $A[i+1]$ contém um valor maior ou igual a *key*.
3. **Finalização**: Quando o laço termina, temos que o subvetor $A[(i+1)..j]$ contém apenas elementos maiores ou iguais a *key*. Utilizaremos esta informação para estabelecermos a correção de InsertionSort ao mostrarmos a invariante do laço **for**.

Exercício: Prove detalhadamente que InsertionSort é correto. Para isto, prove que a invariante acima é verdadeira antes de cada execução do laço **for**.

2.2 A complexidade do algoritmo de ordenação por inserção

Nesta seção estudaremos a complexidade do algoritmo de ordenação por inserção, ou seja, estudaremos a eficiência deste algoritmo. O que significa dizer que um algoritmo é eficiente? Podemos analisar a eficiência de um algoritmo de duas formas: eficiência temporal e eficiência espacial. No primeiro caso, estamos interessados no tempo de execução do algoritmo, enquanto que no segundo caso, queremos analisar a quantidade de espaço (memória) que é utilizado pelo algoritmo durante sua execução. A forma de determinar a eficiência de um algoritmo deve permitir a comparação de algoritmos distintos que resolvam o mesmo problema de forma que possamos determinar qual dos dois é o mais eficiente. Inicialmente, poderíamos pensar em utilizar o tempo de execução de um programa que implementa um algoritmo, mas esta não é uma boa medida porque depende do computador utilizado e da implementação feita. Precisamos de um método que nos informe sobre a eficiência do algoritmo independentemente do computador em que ele venha a ser implementado, da linguagem de programação e do estilo de programação utilizados. O método deve ser preciso e geral de forma que possa ser utilizado para diversos algoritmos e aplicações. O método que usaremos, e que hoje é amplamente utilizado para analisar algoritmos foi desenvolvido pelo matemático e cientista da computação Donald Knuth [3], atualmente com 83 anos, foi laureado com o ACM Turing Award (considerado o prêmio Nobel da Computação) e é considerado o *pai da análise de algoritmos*.

O modelo computacional que utilizaremos é o de uma máquina de acesso aleatório (*random-access machine* - RAM) com um processador, e devemos lembrar que nossos algoritmos serão implementados como programas neste modelo, onde as instruções são executadas sequencialmente, sem operações concorrentes. As instruções no modelo RAM são as encontradas em computadores reais:

- aritmética: soma, subtração, multiplicação, divisão, resto, piso e teto
- movimento de dados (*load*, *store*, *copy*)
- controle (ramos condicionais e não-condicionais, chamadas a subprocedimentos e retorno)

Assumiremos que cada uma destas instruções é executada em tempo constante. Além disto, os tipos abstratos de dados deste modelo são os números inteiros e números em ponto flutuante.

O algoritmo *InsertionSort* acima recebe o vetor A contendo n elementos, digamos $\langle a_1, a_2, \ldots, a_n \rangle$ e retorna o vetor $\langle a'_1, a'_2, \ldots, a'_n \rangle$ contendo os mesmos elementos do vetor A, mas ordenados crescentemente, ou seja, $a'_1 \leq a'_2 \leq \ldots \leq a'_n$. Certamente, ordenar um vetor com 1000 demanda mais tempo do que ordenar apenas 3 elementos, assim é usual descrever o tempo de execução de um algoritmo em função do tamanho da entrada, neste caso n. O tempo de execução de um algoritmo, para uma entrada particular, é o número de operações primitivas ou o número de "passos"executados. Vamos adotar algumas convenções para que a ideia de tempo de execução seja independente do computador a implementar o algoritmo. Por exemplo, assumiremos que cada linha do pseudocódigo é executada em tempo constante que pode diferir de uma linha para outra. Assim, denotaremos por c_i a constante que corresponde ao tempo de execução da

i-ésima linha do pseudocódigo. Vejamos, então, o custo de execução do algoritmo InsertionSort. O **for** da linha 1 é executado $n-1$ vezes, uma vez para cada $j = 2, \ldots, n$, onde $n = A.length$. Denotaremos por t_j o número de vezes que o teste do **while** da linha 4 é executado, assim .

Linha	Custo	Número de execuções	Custo total
1	c_1	n	$c_1.n$
2	c_2	$n-1$	$c_2.(n-1)$
3	c_3	$n-1$	$c_3.(n-1)$
4	c_4	$\sum_{j=2}^{n} t_j$	$c_4.\sum_{j=2}^{n} t_j$
5	c_5	$\sum_{j=2}^{n}(t_j-1)$	$c_5.\sum_{j=2}^{n}(t_j-1)$
6	c_6	$\sum_{j=2}^{n}(t_j-1)$	$c_6.\sum_{j=2}^{n}(t_j-1)$
8	c_8	$n-1$	$c_8.(n-1)$

Portanto, o custo total, que denotaremos por $T(n)$ é dado por:

$$T(n) = c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.\sum_{j=2}^{n} t_j + (c_5+c_6).\sum_{j=2}^{n}(t_j-1) + c_8.(n-1)$$

Agora note que, mesmo para entradas de mesmo tamanho, o tempo de execução pode mudar. De fato, se o vetor dado como entrada já estiver ordenado então $t_j = 1, \forall 2 \leq j \leq n$, e portanto

$$\begin{aligned} T(n) &= c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.(n-1) + c_8.(n-1) \\ &= (c_1+c_2+c_3+c_4+c_8).n - (c_2+c_3+c_4+c_8) \end{aligned}$$

ou seja, uma função linear de n. Por outro lado, se o vetor de entrada estiver ordenado em ordem decrescente, então teremos que $t_j = j$ (por que?), e portanto

$T(n) = c_1.n + (c_2+c_3+c_8).(n-1) + c_4.(\frac{n.(n+1)}{2} - 1) + (c_5+c_6).(\frac{n.(n-1)}{2})$, ou seja, uma função quadrática de n.

A forma de análise feita para InsertionSort acima apresenta alguns problemas porque as constantes utilizadas podem mudar dependendo do computador, da linguagem de programação ou mesmo do estilo de programação utilizados. Uma maneira de ignorar estas especificidades e fazer uma análise que seja independente destes aspectos seria olhar para o número de operações básicas, e contar o número de vezes que estas são executadas. Em diversos casos, para analisar um algoritmo, podemos isolar uma operação fundamental relacionada ao problema em estudo e apenas contar o número de vezes que estas operações são realizadas. Em diversos algoritmos, apenas uma operação básica é realizada em cada passo do *loop* principal do algoritmo, de forma que esta abordagem é semelhante à feita anteriormente para InsertioSort. Por exemplo, qual seria a operação básica para InsertionSort? O algoritmo realiza basicamente duas operações: atribuição (linhas 2, 3, 5, 6 e 8) e comparação (linha 4). Observando a contagem feita anteriormente, concluímos que a parcela mais relevante para o somatório é a dada pela linha 4, o que nos leva a tomar a operação de comparação de chaves (entradas do vetor) como sendo a operação básica neste caso.

Alguns exemplos de operações básicas para diferentes problemas:

Problema	**Operação básica**
Ordenação de um vetor	Comparação de duas entradas do vetor
Busca em um vetor	Comparação com elementos do vetor
Multiplicação de duas matrizes	Multiplicação de dois números reais

Uma vez que a operação básica é bem escolhida, o número total de operações realizadas é proporcional ao número de operações básicas, e portanto temos uma boa medida para determinar o tempo de execução de um algoritmo, e um bom critério para comparar diversos algoritmos. Como veremos, normalmente estaremos interessados na taxa de crescimento do tempo de execução de um algoritmo em função do crescimento do tamanho da entrada. Assim, como o número total de operações é proporcional ao número de operações básicas, uma contagem apenas das últimas é suficiente para nos dar uma boa ideia da viabilidade de utilizar o algoritmo para entradas grandes, *i.e.* para grandes valores de n. O "tempo de execução de um algoritmo"ou a "quantidade de trabalho realizado por um algoritmo"são expressões para denotar a "complexidade de um algoritmo".

O parâmetro n que denota o tamanho da entrada também precisa ser fornecido a partir de uma medida adequada para que possamos apresentar uma análise concisa. Para o caso de ordenação de um vetor, vimos que o número de elementos do vetor representa uma medida adequada. Vejamos mais exemplos:

Problema	**Tamanho da entrada**
Ordenação de um vetor	Tamanho do vetor
Busca em um vetor	Tamanho do vetor
Multiplicação de duas matrizes	Dimensão das matrizes

Voltando ao exemplo de InsertionSort, no caso em que o vetor de entrada já está ordenado temos o melhor dos casos (Complexidade do melhor caso). Em geral, este caso não é de interesse porque normalmente ocorre em situações muito particulares. Normalmente, queremos saber o que ocorre no pior caso (Complexidade do pior caso), já que nos fornece uma cota superior para o trabalho realizado pelo algoritmo.

Definição 2.2.1 (Complexidade no Pior Caso)**.** *Sejam D_n o conjunto das entradas de tamanho n para o algoritmo em questão, e $I \in D_n$. Seja $t(I)$ o número de operações básicas executadas pelo algoritmo na entrada I. Definimos a função W por*

$$W(n) = \max\{t(I) \mid I \in D_n\}$$

A análise do pior caso é importante porque fornece uma cota superior para o tempo de execução do algoritmo. Isto quer dizer que o algoritmo não tem como se comportar pior do que o estabelecido pela análise do pior caso. Esta é a principal razão para iniciarmos sempre por aqui.

A análise do pior caso para InsertionSort de acordo com as considerações acima será dada pela contagem do número de comparações realizadas na linha 4. Como o vetor de entrada está ordenado decrescentemente, para cada $2 \leq j \leq n$, o **while** da linha 4 realiza $j - 1$ comparações. Logo, no total teremos:

$$W(n) = 1 + 2 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{n.(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$

Observe que esta análise é mais sucinta, e fornece essencialmente a mesma informação obtida anteriormente, mas sem as constantes. Como veremos mais adiante, apenas o termo de maior grau será relevante na análise assintótica.

Exercício 2.2.2. *Faça a análise assintótica da complexidade de tempo do seguinte algoritmo:*

Algorithm 2: BubbleSort(A)

```
1 for i = 1 to A.length − 1 do
2 |   for j = A.length downto i + 1 do
3 |   |   if A[j] < A[j − 1] then
4 |   |   |   exchange A[j] with A[j − 1];
5 |   |   end
6 |   end
7 end
```