FIAP ON

ALGORITMOS:
APRENDA A PROGRAMAR

ANÁLISE *ALGORÍTMICA*

JORGE LUIZ SURIAN

6

LISTA DE FIGURAS

SUMÁRIO

6 ANÁLISE ALGORÍTMICA

Vimos desde o princípio deste curso que vários problemas possuem soluções distintas, mas que mesmo corretas, alguns algoritmos, por causa de suas estratégias, são muito mais eficientes que outros. No caso das seis soluções para descobrirmos o maior entre três números, tivemos soluções que faziam um número muito grande de comparações, dezoito no pior caso. Tivemos também uma solução que fez somente duas comparações.

Então é chegado o momento de nos determos mais na importante questão conceitual do desempenho algorítmico.

Notemos que se um programa ou algoritmo está correto, isto é, se atende à especificação para a qual foi concebido. Assim, para determinado conjunto de entrada de dados válidos, o algoritmo fornecerá uma saída esperada e adequada.

Para garantia da **corretude** de um programa ou algoritmo, podemos efetuar testes e simulações, mas na verdade apenas garantiremos que o algoritmo “funciona” para os testes executados. Nada além disso...

Essa questão pode levar a situações descomunais. Pensemos num vetor com 10 elementos para execução de qualquer processo que seja. Como esses elementos poderão ser combinados em arranjos distintos, o número exato de possibilidades é 10, ou seja, nada mais que 3.628.800 possibilidades!

Ou seja, é absolutamente impraticável testar, caso a caso, situações como essa. A verificação do algoritmo é a ferramenta mais poderosa que existe para garantir a corretude de um algoritmo. Na prática, várias situações prescindem desse tipo de abordagem, todavia se for necessária a comprovação de uma rotina, essa é uma das melhores formas de fazê-lo.

Em geral, pelo menos no processo de desenvolvimento de um algoritmo, fazemos alguns testes, habitualmente com um conjunto finito e bem determinado de itens, capazes de permitir a realização de testes que nos permitam descobrir erros grosseiros.

Já a eficiência de um programa ou algoritmo é avaliada em função da velocidade em que esse algoritmo resolve um problema, aliado ao consumo de memória que o programa acarreta.

Naturalmente, o tempo de execução será, em última medida, função da lógica que norteou a construção do algoritmo, do número de instruções que executa, da velocidade de cada tipo de instrução e da máquina na qual o algoritmo foi implementado.

6.1. Análise da Complexidade de um Algoritmo

A análise da complexidade de um algoritmo visa, em última análise, estimar o tempo de execução de um algoritmo. A complexidade do algoritmo nada mais é que uma ideia do esforço computacional despendido para o computador resolver determinado problema.

Antes de 1970, esse estudo do comportamento algorítmico era feito de forma empírica, por meio de medições. Ou seja, a partir de determinado conjunto de dados, simplesmente se media o tempo que um conjunto de operações de um algoritmo demorava. Assim se determinava o desempenho do algoritmo.

Embora bastante útil, esse método traz em seu bojo vários problemas:

- Depende do computador (características físicas da máquina) onde for executado.
- Depende da linguagem em que foi desenvolvido.
- Depende do compilador em que a linguagem foi desenvolvida.
- Dependo do sistema operacional em que o sistema está rodando.
- Depende da existência de outros programas compartilhando a memória e processador da máquina.

Devemos observar que para podermos afiançar se um programa funciona com o desempenho necessário, o processo empírico é muito frágil para garantir algo.

Todavia, se tomarmos um computador três vezes mais veloz que outro ou um compilador dez vezes mais rápido que outro, determinado algoritmo irá funcionar três ou dez vezes mais rápido, na dependência exclusiva do hardware e/ou software envolvidos.

Parece fácil intuir que separados os aspectos tecnológicos envolvidos, ao final, resta apenas a qualidade do algoritmo a determinar o desempenho do algoritmo em questão.

Em outras palavras, o que se deseja efetivamente é avaliar o desempenho do algoritmo independentemente de sua implementação, em função apenas das instruções executadas.

A complexidade algorítmica é determinada, por sua vez, pelo número de vezes que uma instrução básica é executada. Nesse quesito, podemos facilmente intuir que a complexidade é diretamente influenciada pelos laços de repetição. Obviamente, dependendo dos valores na entrada, de seus tipos e tamanhos, poderemos ter aumento ou redução dos tempos associados à execução de um algoritmo.

Vamos imaginar que precisemos calcular um aumento de todos os itens de uma lista com 10.000 registros. Obviamente, esse cálculo será muito mais demorado que o **mesmo** cálculo aplicado a uma lista de 10 registros.

Por outro lado, pensemos no cálculo da folha de pagamento de 10.000 funcionários ou de 10. Caso os cálculos dos 10.000 seja simplesmente duas ou três operações aritméticas e dos 10 seja um cálculo intrincado, envolvendo pesquisas e percentuais obtidos em tabelas, já não é fácil garantir o que será processado em menor tempo. Aliás, pela descrição apresentada, parece muito mais provável que a segunda lista, não obstante, seja maior, e será processada num tempo maior.

6.2 Análise da complexidade de algoritmos em vetores

Os algoritmos que executam operações sobre vetores tratando-os como listas lineares, ou seja, com acesso sequencial entre eles, a complexidade é expressa em função do tamanho do vetor, como seria natural se esperar.

Vamos chamar de *n* o número de elementos desse vetor. Ora, nessa situação, então, a complexidade associada a qualquer algoritmo que envolver essa lista será função de ***n.***

Por outro lado, como os valores da lista e a configuração dos dados influem no processo, não é possível obter uma única função que descreva todas as possibilidades existentes, relativas a qualquer algoritmo. Mesmo em vetores de tamanhos idênticos, se estivermos buscando um elemento aleatoriamente, é bastante provável que não importa o local onde este esteja, teremos um mesmo tempo para listas de tamanho idêntico.

Já se nossa rotina sempre busca do primeiro para o último, se os números que procurarmos estiverem sempre numa determinada posição, dependendo dessa posição um dos algoritmos deverá levar vantagem sobre o outro. Basta imaginar o esforço que o algoritmo que busca sempre a partir da primeira posição teria, se sempre o valor procurado estiver na primeira posição...

Vamos então estudar algumas situações particulares, essas sim muito mais interessantes do que soluções meramente especulativas ou específicas.

Seguem alguns casos diferenciados:

- **Pior Caso:** caracteriza-se por entradas que resultam em maior crescimento do número de operações executadas pelo algoritmo quanto maior for o valor de *n.*
- **Melhor Caso:** caracteriza-se por entradas que resultam em menor crescimento do número de operações executadas pelo algoritmo quanto maior for o valor de *n.*
- **Caso Médio:** caracteriza-se pela esperança matemática envolvida no comportamento médio do algoritmo, em função da probabilidade matemática associada às entradas de dados possíveis.

Deve-se notar que as ferramentas de análise matemática muitas vezes resolvem o problema da complexidade com sutileza e fineza. Noutras, entretanto, podemos ser levados a situações extremamente complexas e permanentemente pendentes.

De toda forma, se tivermos dois algoritmos equivalentes, apenas sua complexidade poderá demonstrar cabalmente qual é o melhor deles.

6.3 Notação O

A notação O é usada para expressar comparativamente o crescimento assintótico de duas funções. Esse crescimento representa a velocidade com que uma função tende ao infinito.

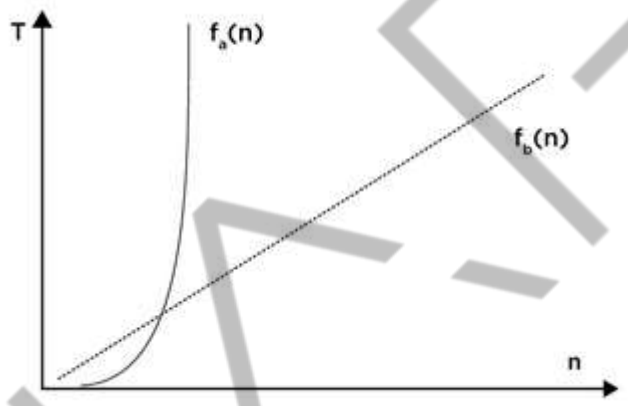

Figura 6.1 – Exemplo de crescimento assintótico
Fonte: FIAP (2015)

Quando estudamos a complexidade algorítmica, é relevante saber como se comporta a função, à medida que aumentamos o tamanho de *n* (ou seja, o tamanho do vetor), do que conhecer valores específicos que a função possui para particulares *n*.

Algumas funções elementares são utilizadas como referência para classes algorítmicas, como: 1, *n*, ln *n*, *n* ln *n*, n^2, 2^n etc.

Quando dizemos que uma função de complexidade f(*n*) é de ordem n2, na verdade estamos dizendo que as duas funções: f(*n*) e *n*2 tendem ao infinito com igual velocidade, ou seja, tem o mesmo comportamento assintótico. Indicamos essa situação da seguinte maneira:

$$f(n) = O(n^2)$$

Em matemática, essa informação é expressa por um limite:

$$\lim_{n \to \infty} \frac{f(n)}{n^2} = c$$

Onde *c* é uma constante.

Para determinar a complexidade típica de um algoritmo, temos alguns resultados notáveis apresentados adiante.

Função	Significado
1	Tempo Constante, ou seja, o número de operações é o mesmo, qualquer que seja o tamanho da entrada.
N	Tempo Linear, ou seja, quando n dobrar de tamanho, o número de operações também dobrará.
n^2	Tempo Quadrático, ou seja, se n dobrar de tamanho, o número de operações associadas passa a ser o quadrado de n.
ln *n*	Tempo Logarítmico, ou seja, se n dobrar de valor, o número de operações varia numa constante logarítmica.
n ln *n*	Tempo n ln n, se n dobrar de valor o número de operações varia numa constante logarítmica em produto com o número de termos.
2n	Tempo Exponencial, ou seja, se n dobrar o número de operações, é elevado à potência de n.

Figura 6.2 – Significado das funções
Fonte: FIAP (2015)

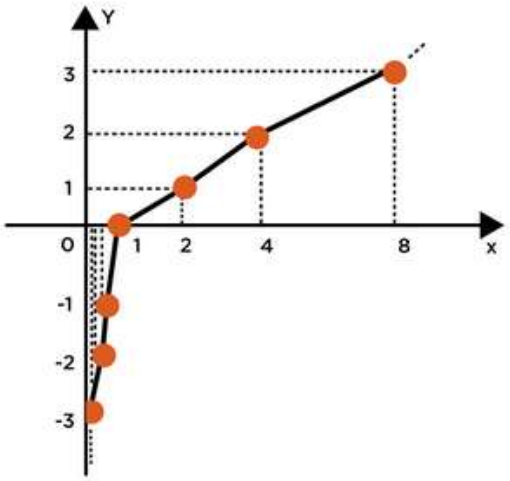

Figura 6.3 – Exemplo de função logarítmica
Fonte: FIAP (2015)

Nos capítulos relativos à pesquisa e de ordenação, serão feitas análises de complexidade para um algoritmo de pesquisa (busca sequencial) e de ordenação (método da inserção), uma vez que os cálculos matemáticos envolvidos ficam mais compreensíveis à luz de algoritmos cujo funcionamento já tenha sido entendido.

REFERÊNCIAS

FEOFILOFF, Paulo. **Algoritmos em Linguagem C.** Rio de Janeiro: Campus, 2009.

FORBELLONE, André L.V.; EBERSPACHER, Henri F. **Construção de Algoritmos e Estruturas de Dados**. São Paulo: Pearson Prentice Hall, 2010.

FURGERI, Sérgio. **Java 2, Ensino Didático***.* São Paulo: Érica, 2002.

GANE, Chris; SARSON, Trish. **Análise Estruturada de Sistemas***.* São Paulo: LTC-Livros Técnicos e Científicos, 1983.

GONDO, Eduardo. **Apostila:** Notas de Aula*.* São Paulo, 2008.

LATORE, Robert. **Aprenda em 24 horas Estrutura de Dados e Algoritmos***.* Rio de Janeiro: Campus, 1999.

MANZANO, José A. N. G.; OLIVEIRA, Jayr F. **Algoritmos:** Lógica para o Desenvolvimento de Programação. 23. ed. São Paulo: Érica, 2010.

PIVA JUNIOR, Dilermando et al. **Algoritmos e Programação de Computadores.** Rio de Janeiro: Campus, 2012.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de Programação e Estrutura de Dados**. São Paulo: Pearson Prentice Hall, 2009.

ROCHA, Antonio Adrego. **Estrutura de Dados e Algoritmos em Java***.* Lisboa: FCA- Editora de Informática, 2011.

RODRIGUES, Rita. **Apostila**: Notas de Aula*.* 2008.

SALVETTI, Dirceu Douglas; BARBOSA, Lisbete Madsen. **Algoritmos***.* São Paulo: Makron Books, 1998.

SCHILDT, Herbert. **Linguagem C - Guia Prático***.* São Paulo: McGraw Hill, 1989.

SCRIPTOL. **Documentation**. [s.d.]. Disponível em: <http://www.scriptol.org/>. Acesso em: 14 jan. 2011.

WOOD, Steve. **Turbo Pascal – Guia do Usuário***.* São Paulo: McGraw Hill, 1987.

ZIVIANI, Nivio**. Projeto de Algoritmos com implementações em Pascal e C.** São Paulo: Pioneira, 1999.