Notas de Aula - AEDI – Complexidade de Algoritmos
Prof. Jefferson T. Oliva

Por que estudar a complexidade de algoritmos?

Antes disso, uma outra pergunta: como o “desempenho” de um algoritmo é mensurado?

Na disciplina AED1 vocês verão algoritmos diferentes que resolvem os mesmos problemas, sendo os mesmos iterativos e recursivos. Exemplo disso são os algoritmos de pesquisa e de ordenação.

Se eles foram desenvolvidos para um mesmo problema, como compará-los? Aí que entra a motivação para o estudo de complexidade de algoritmos, que tem o objetivo de analisar o comportamento de cada algoritmo.

Antes de dar continuidade, vocês sabem diferenciar um programa de um algoritmo? (ver a tabela do slide 4)

Enquanto um programa considera a linguagem de programação, o sistema operacional e hardware, o algoritmo é independente de todas essas características. Afinal independente de todas essas características o algoritmo irá desempenhar a sua tarefa. O que pode ocorrer é a variação do tempo de execução em cada hardware, sistema operacional. Nesta aula, vocês verão que há uma forma de analisar os algoritmos de forma independente do “ambiente”. Além disso, um programa pode conter vários algoritmos. Por exemplo, editores de texto, que basicamente têm um algoritmo para salvar, abrir, etc.

Afinal, como mensurar a eficiência de um algoritmo?

Análise de algoritmos

Para a análise de algoritmos, é importante mensurar os recursos necessários para a execução: tempo e espaço

Um algoritmo que soluciona um determinado problema, mas que demore muito tempo (exemplo: um ano), não deve ser usado.

No slide 7 é apresentado um exemplo de comparação de dois algoritmos fictícios: TripleX e Simplex. É muito tentador escolher o TripleX para o exemplo apresentado, mas diversos fatores devem ser considerados, como a linguagem de programação, o hardware, o sistema operacional e as habilidades do programador, os quais variam muito. Assim, considerando essas premissas, é muito difícil avaliar o desempenho de um programa. Também, será que o desempenho do TripleX ainda é superior ao SimpleX para um tamanho de conjunto de dados maior?

Por essa razão, a comunidade de computação vem pesquisando formas para comparar algoritmos de forma independente de hardware, sistema operacional, linguagem de programação e habilidade do programador. Por isso, é desejável a avaliação de algoritmos e não programas.

Dessa forma, surgiu uma área da computação denominada análise/complexidade de algoritmos, que tem o propósito de determinar os recursos necessários para executar um dado algoritmo e compará-los com outros. Também, nessa área é verificado se a solução (algoritmo) de um determinado problema é ótima, mas essa parte não é tratada nessa disciplina. Provavelmente vocês verão isso em projeto de algoritmos.

Sabe-se que processar 100.000 números leva mais tempo do que 10.000 números e cadastrar 20 itens em um sistema de vendas leva mais tempo do que cadastrar 10. Então, uma ideia interessante seria medir a eficiência de um algoritmo de acordo com a quantidade de dados processados (tamanho do problema).

Geralmente, é assumido que n é o tamanho do problema, ou seja, o tamanho do conjunto de dados. E se conjunto de dados for bidimensional (matriz)? O tamanho do problema é dado pela dimensão do conjunto (nro de linhas x nro de colunas).

A partir de conjunto de dados é calculado o número de operações realizadas. Desse modo, o melhor algoritmo é aquele que requer menos operações sobre a entrada.

Que operações? Atribuição, comparação, soma, subtração, etc.

Toda operação leva o mesmo tempo? Não, em razões das diferentes configurações de hardware e do sistema operacional. Por isso, assumimos que cada operação tem o custo de uma unidade, cuja explicação será mais adiante.

Voltando ao exemplo TripleX vs. SimpleX no slide 17, onde é apresentada a equação referente à quantidade de operações necessárias para cada um. Exercício: calcular a quantidade operações para os tamanhos de entrada 1, 10, 100, 1.000, 10.000.

Nesse exemplo, vimos que o SimpleX é mais rápido que o TripleX para conjuntos de dados a partir de n = 1.000. Assim, podemos dizer a função Triplex cresce mais rápido que o seu concorrente, ou seja, realiza mais operações.

Cálculo do Tempo de Execução

Existem basicamente 2 formas de estimar o tempo de execução de programas e decidir quais são os melhores: Empiricamente e Teoricamente.

Na análise empírica, o tempo de execução é a principal forma de avaliação de algoritmos, mas isso depende de vários fatores, que foram citados no início da aula.

Para proceder a uma análise de algoritmos e determinar as taxas de crescimento, necessitamos de um modelo de computador e das operações que executa. Para isso, assume-se o uso de um computador tradicional, em que as instruções de um algoritmo são executadas sequencialmente. Para simplificar, é considerado que o computador possui memória infinita.

A partir de agora serão apresentadas algumas regras para o cálculo de quantidade de instruções.

Repertório de instruções simples: soma, multiplicação, comparação, atribuição, etc.
- Por simplicidade e viabilidade da análise, assume-se que cada instrução demora exatamente uma unidade de tempo para ser executada. Obviamente, em situações reais, isso pode não ser verdade: a leitura de um dado em disco pode demorar mais do que uma soma.
- Operações complexas, como inversão de matrizes e ordenação de valores, não são realizadas em uma única unidade de tempo. Essas operações devem ser analisadas em partes.

Repetições: tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada mais a última atribuição e comparação (quando a variável i, por exemplo, atinge o critério de parada). O tempo de execução de uma repetição é pelo menos o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada

O exemplo abaixo executa 3n + 2 instruções, ou seja, 1 inicialização de i + n * (1 comparação (i < n) + 1 atribuição na variável k + 1 atribuição de i) + 1 última comparação (atinge o critério de parada, isto é, i passa ser igual ou maior que n) .

```
para i = 0 até n faça
        k = 0;
```

Se... então... senão: para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste (então) mais o tempo do senão. No cálculo do tempo de execução, é considerado o comando que leva mais tempo, se a intenção for uma projeção pessimista (mais utilizada)

- O exemplo abaixo pode executar até 4n + 3 instruções

```
se i < j
        então i = i + 1
        senão para k 1 até n faça
                i = i * k;
```

Chamadas de sub-rotinas: uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou.

Repetições aninhadas: a análise é feita de dentro para fora.
- Tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições.

O exemplo de fragmento de código que executa $4n^2 + 4n + 2$ instruções.

```
para i = 0 até n faça
        para j = 0 até n faça
                k = k + 1;
```

Ver exemplo nos slides 22 e 23

Notação O (Big Oh)

A comparação de algoritmos é realizada por meio análise assintótica, que é uma forma de descrever o comportamento dos limites de funções que representam a quantidade de operações realizadas de acordo com o tamanho do conjunto de dados.

Devemos nos preocupar com a eficiência de algoritmos quando o tamanho de n for grande. Em outras palavras, para conjuntos muito pequenos, pode não fazer muito sentido essa análise.

Sobre as funções de crescimento, o algoritmo que tiver menor taxa de crescimento é o que rodará mais rápido quando o problema for grande. As funções são dadas em termos não negativos.

Ao dizer que f(n) = O(g(n)), garante-se que f(n) cresce numa taxa não maior do que g(n), ou seja, g(n) é seu limite superior.

O termo assintótico mais utilizado é o big-oh, pois em casos reais, o problema cresce com o decorrer do crescimento do volume de dados.

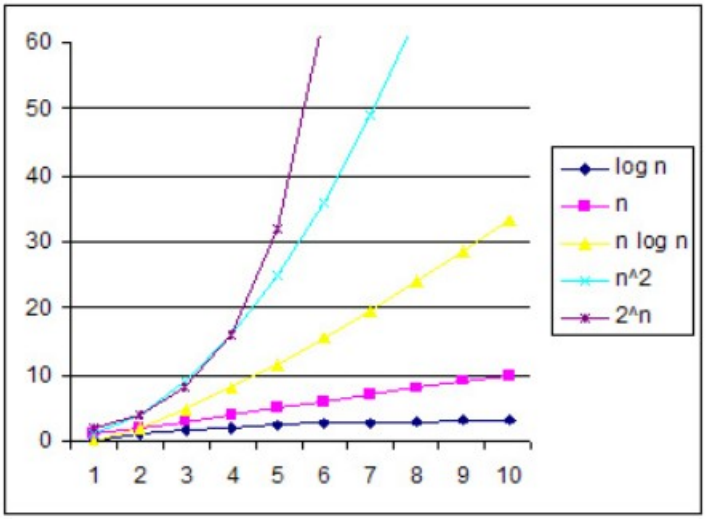
Ver slides de 29 ao 34

Taxas de Crescimento

Taxa de crescimento mais comuns: constante (operações simples independentes de n), logarítmica (divisão e conquista), quadrática (dados processados em pares, como ordenação), cúbica (multiplicação de matrizes), exponencial (força bruta), fatorial (força bruta).

Apesar de às vezes ser importante considerarmos constantes ou termos de menor ordem para a análise de algoritmos. Portanto, para medir a taxa de crescimento, os termos menores são irrelevantes.

Principais Taxas de crescimento:



Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Marin, L. O. Complexidade de Algoritmos - parte 2. AE23CP-3CP. Algoritmos e Estrutura de Dados II. Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2017.

Marin, L. O. Complexidade de Algoritmos - parte 1. AE23CP-3CP. Algoritmos e Estrutura de Dados II. Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2018.

Rosa, J. L. G. Análise de Algoritmos. SCE-181 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2008.

Rosa, J. L. G. Análise de Algoritmos - parte 1. SCC-201 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2016.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.