Notas de Aula - AED2 – Complexidade de Algoritmos (parte 1)
Prof. Jefferson T. Oliva

1 - Por que estudar a complexidade de algoritmos?

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

Na disciplina AED1 vocês viram diferentes algoritmos que resolvem os mesmos problemas, sendo os mesmos iterativos ou 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, o algoritmo irá desempenhar a mesma sequência de comandos. 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.

Uma pergunta: o que é um algoritmo?

Essa palavra vem do nome de um matemático, cujas obras foram traduzidas no ocidente no século XII, tendo uma delas recebido o nome Algorithmi de numero indorum, sobre os algoritmos usando o sistema de numeração decimal (indiano).

Algoritmo é um conjunto de passos para resolver um problema. Um algoritmo processa o que recebe como entrada e retorna uma saída. Um exemplo é a receita de bolo, no qual ingredientes (entrada) devem ser processados (sequência de tarefas). Após levar a massa no forno, a saída é um bolo pronto.

Computacionalmente: um algoritmo recebe valores de entrada, realiza uma sequência de tarefas e produz valores como saída. E se uma função não receber valores como entrada, ele continua sendo algoritmo? Depende: se realiza um conjunto de passos finitos sim.

Um exemplo de problema é a ordenação de vetores, que deve ser crescente (mais comum) ou decrescente. Na ordenação crescente, o item V[i] deve ser menor ou igual ao item V[i + 1]. O problema é como rearranjar esses itens. Como vocês viram em AED1, existem diversos algoritmos de ordenação. Vocês viram diversos algoritmos para isso, tais como bublesort, insert sort, select sort, heap sort, shell sort, quicksort e merge sort.

Quais são as entradas desses algoritmos?

Como é o processamento (passos)?

Por fim, qual é a saída?

Um problema simples: determinar se um número é par:
- Entrada: um número inteiro
- Passos: obter o resto da divisão da entrada por 2 e verificá-lo se o mesmo é igual a 0.
- Saída: 1 (par), 0 (caso contrário)

Como visto anteriormente, algoritmo tem a finalidade de resolver um problema. “É desejável que um algoritmo termine e seja correto”. Ou seja, um algoritmo deve ter um conjunto de passos finitos (em caso de loop ou de chamada recursiva, deve haver um critério de parada) e fazer a operação de forma correta.

Um algoritmo incorreto pode entrar em loop infinito ou produzir saídas erradas.

Além dessas restrições, um algoritmo deve ser eficiente, ou seja, também deve evitar o desperdício de recursos computacionais. Em problemas de ordenação, por exemplo, não existe um algoritmo “universal” que se destaca entre os outros, mas para determinados casos, um pode ser mais eficiente que o outro. Por exemplo, a ordenação por inserção é mais vantajoso para vetores parcialmente ordenados.

Exitem diversas aplicações em que são utilizados algoritmos eficientes: games, filmes, etc.

Afinal, como mensurar a eficiência de um algoritmo?

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 12 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 área 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 (número de linhas x número 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 16, 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.**

Análise Assintótica

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.

A análise assintótica também pode ser empregada para a quantidade de memória usada por um algoritmo. Como atualmente os computadores atuais possuem bastante recursos de memória, raramente é feita uma análise em termos de espaço.

Nos slides 21, 22 e 23 contêm alguns conceitos matemáticos básicos para serem relembrados, pois podem ser essenciais para a análise de complexidade de algoritmos. Sobre as séries do slide 23, vocês verão, em alguma disciplina, como provar se tais equações estão corretas.

As funções são dadas em termos não negativos.

Notações Big-Oh, Omega, Teta.

Notação Big-Oh

Essa notação significa: “no pior das hipóteses, o algoritmo atinge tal função”.

Essa notação descreve o limite superior que uma função possa atingir. Por isso, essa notação é a mais utilizada na análise do algoritmo, pois procuramos fazer a análise de um algoritmo de forma “mais pessimista/realista”.

Conforme apresentado no slide, ao dizer que f(n) = O(g(n)), garante-se que f(n) cresce em uma taxa não maior do que g(n), ou seja, g(n) é seu limite superior.

Ver exemplos no slide 27.

Algoritmo de ordenação quicksort: ele pode ser um algoritmo bastante eficiente para a ordenação de certos conjuntos de dados, mas quando os dados estão ordenados e dependendo da forma que o pivô é escolhido, ele realiza a maior quantidade de operações, que é na ordem de n^2, ou seja, O(n^2).

Notação ômega (big-ômega)

Essa notação significa: “isso é o melhor que posso fazer”

Essa notação descreve o limite inferior que uma função possa atingir. Essa notação é pouco utilizada, pois ela pode ser “muito otimista”.

Ao dizer que f (n) = Ω(g(n)), tem-se que g(n) é o limite inferior de f(n).

Ver exemplo no slide 29.

O melhor que o algoritmo quicksort pode fazer é n log n operações, isto é, Ω(n log n)

Notação theta (big-theta)

Essa notação significa: “não importa quais as circunstâncias, não faço melhor e nem pior”.

Essa notação descreve um limite assintótico estrito.

Ver exemplo no slide 31.

Para o algoritmo de ordenação heapsort, o custo mínimo e máximo é Theta(n log n).

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).

Ver slides 34--39

Referências

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S. Algoritmos: teoria e prática. Elsevier, 2012.

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

Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 2010.

Ziviani, M. Projetos de Algoritmos: com implementações em Pascal e C. Thomson, 2004.