Notas de Aula - AED2 – Complexidade de Algoritmos (parte 2)
Prof. Jefferson T. Oliva

Resumo da aula anterior

Análise assintótica
- Big-Oh
- Ômega
- Theta

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. Por isso, é desejável análise do pior caso nos algoritmos

Em muitos casos, o notação big-oh pode ser substituída pela Theta, considerando limites mais apertados. Em outras palavras, quando usamos a notação big-Θ estamos dizendo que temos um limite assintoticamente restrito no tempo de execução.

A função n^3 pode ser O(n^3), Ômega(n^2), Ômega(n^3), ou Theta(n^3). Traduzindo esses termos:

O(n^3): a função não cresce mais rápido que n^3
Ômega(n^2) : a função cresce mais rápido que n^2.
Ômega(n^3) : a função cresce mais rápido que n^3.
Theta(n^3): a função não cresce assintoticamente tão rápido quanto n^3. O limite assintótico é bem definido. Exemplo, o algoritmo de ordenação por inserção é O(n^2)

Taxas de crescimento:
Ômega: pode aumentar
Big-oh: pode diminuir

n^2 = O(n^3)
n^3 = Ω(n^2)

O algoritmo de Coppersmith e Winograd (multiplicação de matrizes quadradas) é de O($n^{2,376}$), mas a cota inferior (conhecida até hoje) é de Ômega(n^2). Portanto, não podemos dizer que o algoritmo é ótimo.

Muitos pesquisadores trabalham para encurtar o intervalo Ômega e Big-oh.

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, na análise de complexidade são considerados irrelevantes.

Análise de Algoritmos

Para proceder a uma análise de algoritmos e determinar as taxas de crescimento, necessitamos de um modelo de computador e das operações que executa.

Assume-se o uso de um computador tradicional, em que as instruções de um algoritmo são executadas sequencialmente (com memória infinita, por simplicidade).

Em outras palavras, devemos definir valores para operações.

Em instruções simples, como soma, atribuição, multiplicação, comparação, subtração, etc, é assumido um custo constante O(1) para cada uma ser executada

Operações complexas, como ordenação de vetores, não são realizadas em uma única unidade de tempo, mas sim em função do tamanho do conjunto de dados.

Operações complexas devem ser realizadas em partes. Por exemplo, do comando mais interno ao mais externo.

Qual a complexidade do algoritmo abaixo?

```
int busca(int x, int v[], int n){
1.      int i;
2.      for (i = 0; i < n; i++)
3.              if (x == v[i])
4.                      return i;
5.      return -1;
}
```

Considera-se somente o algoritmo e suas entradas (de tamanho *n*). Caso o algoritmo chama uma outra função, a mesma deve ser analisada separadamente.

Para o cálculo de complexidade de um algoritmo, consideramos o tempo de execução para: **melhor caso, caso médio, pior caso**

O melhor caso tem pouca utilidade prática, pois oferece um cálculo muito otimista sobre o comportamento do algoritmo (o mínimo de operações necessárias para encontrar a solução). Por exemplo: na busca sequencial, conforme apresentado no slide 7, o melhor caso é quando o elemento a ser procurado encontra-se na primeira posição no vetor.

Busca do elemento 5
[5 150 31 10 17 18 45 98 101 200]

O elemento é encontrado após 1 execução da operação fundamental, ou seja, a melhor situação possível.

O pior caso é a complexidade pessimista, mede o esforço máximo necessário para resolver um problema de tamanho n. É fácil de calcular. Assim, geralmente é utilizado somente a análise do pior caso, pois ela fornece os limites para todas as entradas, incluindo particularmente as entradas ruins. Por essa razão a complexidade de algoritmos geralmente é em termos de big-oh. Ver exemplo de busca sequencial, onde o item procurado pode ser inexistente ou encontrado na última posição do vetor.

Quanto ao custo médio, por mais que possa ser útil, principalmente em sistemas executados rotineiramente, o seu cálculo é mais complexo. A complexidade média pode ser medida por meio da média da complexidade de todas as entradas possíveis.

Idealmente, para um algoritmo qualquer de ordenação de vetores com *n* elementos:
- Qual a configuração do vetor que você imagina que provavelmente resultaria no melhor tempo de execução?
- E qual resultaria no pior tempo?

Exemplo: Soma da subsequência máxima

Dada uma sequência de inteiros (possivelmente negativos) *a_1, a_2, ..., a_n* encontre o valor da máxima soma de quaisquer números de elementos consecutivos.

se todos os inteiros forem negativos, o algoritmo deve retornar 0 como resultado da maior soma

Por exemplo, para a entrada -2, 11, -4, 13, -5 e -2, a resposta é 20 (soma de *a_2, a_3* e *a_4*).

Para esse tipo de problema existem diversas soluções com diversos custos computacionais.

Sobre a análise de complexidade é importante notar (slide 14): Para entradas pequenas, todas as implementações rodam num piscar de olhos; Para entradas grandes, o melhor algoritmo é o 4; Os tempos não incluem o tempo requerido para leitura dos dados de entrada. Para o algoritmo 4, o tempo de leitura/pré-processamento é provavelmente maior do que o tempo para resolver o problema: característica típica de algoritmos eficientes.

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.

Empiricamente seria, por exemplo, rodar o algoritmo e obter o tempo de execução, mas é uma tarefa complexa, pois depende de vários fatores, como o hardware.

Ao analisar um algoritmo teoricamente, além de não precisarmos rodá-lo, podemos estimar a sua complexidade. Essa estimativa é apenas baseada no tamanho dos dados, ou seja, todos os outros fatores (e.g. hardware) são descartados.

Ver slides 20 e 21 (mais um exemplo de cálculo de operações de um algoritmo)

Ter que realizar todos os passos para o cálculo da quantidade de operações de cada algoritmo (principalmente algoritmos grandes) pode se tornar uma tarefa cansativa. Como a complexidade dada em termos de big-oh, as constantes e elementos menores dos cálculos são desconsideradas. Por exemplo, no código apresentado no slide 20, a grandeza de tempo se dá na linha 4 ("para i = 1 até n faça").

Ver regras para o de complexidade nos slides entre 23 e 28.

Exercício no slide 20.

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.

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

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.