Análise de Algoritmos – Semana 1

Prof. Dr. Juliano Henrique Foleis

1 Aula 1

1.1 Algoritmos e Problemas

Definição 1. **Algoritmo** – Qualquer procedimento computacional bem definido que toma um valor ou conjunto de valores como entrada e produz um valor ou conjunto de valores como saída.

Definição 2. **Problema Computacional** – Especifica em termos gerais a relação desejada entre entrada e saída.

Assim, um algoritmo é uma ferramenta que `resolve` um problema computacional.

Exemplo – Problema de Ordenação

Entrada: Uma sequência de n números $S = \langle a_1, a_2, ..., a_n \rangle$

Saída: uma permutação de S tal que ${a_1}' \leq {a_2}' \leq ... \leq {a_n}'$

Assim, dada a entrada $\langle 7, 8, 4, 0, 2, 10 \rangle$, um algoritmo de ordenação `correto` devolve a sequência $\langle 0, 2, 4, 7, 8, 10 \rangle$ como saída.

Em geral, a `instância de um problema` consiste em uma entrada que satisfaz quaisquer restrições impostas no enunciado do problema. Portanto, sequencias de diferentes tamanhos e números representam as diferentes instâncias que podem ser apresentadas a um problema de ordenação descrito acima.

Um algoritmo é dito `correto` se, para toda entrada, o algoritmo pára com a saída correta. Portanto, um algoritmo correto `resolve` dado problema computacional. No entanto, existem algoritmos incorretos que são úteis. Que tipos de algoritmos são estes? São aqueles os quais é possível calcular o erro.

1.1.1 Especificação de algoritmos

Algoritmos podem ser especificados em linguagens de programação, especificações formais e linguagens algoritmicas. Neste curso será utilizada uma linguagem algorítmica conhecida como pseudocódigo. O pseudocódigo é útil pois não tem que lidar com problemas de implementação como alocação de memória, entre outros. Além disto, alguns autores diferenciam algoritmos e programas, para eles

Definição 3. **Programa** – Uma implementação de um algoritmo em uma linguagem de programação (C/C++, etc).

1.1.2 Tipos de Problemas Computacionais

Problemas Computacionais podem ser categorizados em quatro tipos, a saber:

Definição 4. **Problema de decisão** – Problema onde a saída para todas as instâncias é sim ou não.

Exemplo: "Dado um inteiro positivo n, n é primo?". A resposta pode ser sim ou não.

Definição 5. **Problema de busca** – Problema onde a solução é apresentada de forma arbitrária.

Exemplo: "Dado um inteiro positivo n, quais são os fatores primos não triviais de n?". A resposta pode ser todos os fatores primos não triviais (1 e n) de n. Se $n = 6$, então a saída é $\{2, 3\}$. Se $n = 21$, então a saída é $\{3, 7\}$.

Definição 6. **Problema de contagem** – Problema onde a saída é o número de soluções para dado problema de busca.

Exemplo: "Dado um inteiro positivo n, quantos fatores primos não triviais n possui?". Para $n = 6$, saída 2. Para $n = 27$, saída 1.

Definição 7. Problema de otimização – Problema cuja solução deve ser a melhor entre todas as possíveis soluções para determinado problema.

Exemplo: "Dado um conjunto de cidades e o comprimento das estradas entre todas as cidades, encontre o menor caminho possível entre duas cidades dadas, A e B".

1.1.3 Complexidade de Problemas

Um problema é computacionalmente difícil quando não é conhecida solução algorítimica exata eficiente. Um algoritmo eficiente é um algoritmo que resolve o problema em tempo hábil, utilizando recursos computacionais como processamento e memória de forma ecifiente. Um subconjunto desses problemas, chamado de problemas NP-Completos, são problemas computacionais extremamente interessantes e devem ser estudados por cinco fatores importantes, a saber:

1. Vários problemas conhecidos e importantes na prática são NP-Completos. Exemplos: SAT, Caixeiro Viajante, Cobertura de conjuntos, problema de satisfação de restriçoes, etc.

2. Embora ninguém tenha encontrado solução eficiente para estes problemas, não existe prova que estas soluções não existem!

3. Estes problemas possuem a propriedade notável que, se existir algoritmo eficiente para algum deles, então existem algoritmos eficientes para todos eles.

4. Problemas NP-Completos podem ser transformados uns nos outros. Logo, algoritmos que servem para um problema servem para todos eles.

5. Para vários destes problemas, existem problemas semelhantes, mas não idênticos, os quais existem algoritmos conhecidos eficientes.

Assim, é importante que um cientista da computação saiba identificar problemas difíceis pois é provável que perca muito tempo em uma busca infrutífera por um algoritmo exato. Por outro lado, ao identificar um problema difícil, o cientista pode dedicar seu tempo para desenvolver um algoritmo eficiente que ofereça boas soluções, embora não a melhor possível. Neste contexto, algoritmos heurísticos ou de aproximação podem ser utilizados para obter boas soluções de forma eficientes.

CUIDADO! Embora problemas de otimização sejam mais facilmente rotulados como problemas difíceis, existem problemas difíceis de decisão, busca e contagem!

A área da computação que estuda complexidade de problemas é a Teoria da Computação.

1.1.4 Complexidade de Algoritmos

Algoritmos diferentes podem ser desenvolvidos para resolver determinado problema.

A principal questão que abordamos nesta disciplina é: como determinar a eficiência de algoritmos, e como determinar qual é o melhor deles para determinado problema? Além disto, abordamos a questão: qual é o fator determinante para a eficiencia da solução de um problema computacional: *hardware*, *software* ou o algoritmo?

Mostraremos mais adiante que o algoritmo de ordenação por inserção (*Insertion Sort*) leva, em seu pior caso, tempo aproximadamente igual a $c_1.n^2$ para ordenar n itens, onde c_1 é uma constante que não depende de n e que representa o custo médio da execução de cada instrução. O algoritmo de ordenação por intercalação (*Mergesort*), por sua vez, leva tempo aproximado de $C_2.n.\lg(n)$, onde $\lg(n) = \log_2(n)$ e c_2 outra constante não dependente de n.

Se representarmos a eficiência do algoritmo de ordenação por inserção por $C_1.n.n$ e o algoritmo de ordenação por intercalação por $C_2.n.lg(n)$, nota-se que enquanto o fator de tempo para a ordenação por inserção é n, o fator para o algoritmo de ordenação por intercalação é apenas $lg(n)$. Ou seja, para $n = 1000, \lg(n) = 10$ e para $n = 1000000, \lg(n) = 20$. Mesmo supondo que $c_2 > c_1$, o algoritmo por inserção é mais rápido que o algoritmo por intercalação apenas para um n relativamente pequeno. Conforme n aumenta, o fator $\lg(n)$ compensará n com sobras a diferença em fatores constantes. É possível provar que, independente do quanto c_1 seja menor que c_2, sempre haverá um n inicial tal que o algoritmo por intercalação sempre será mais eficiente que o algoritmo por inserção.

Exemplo – Complexidade de Algoritmos

Consideremos um computador A mais eficiente que executa a ordenação por inserção e um computador B, que executa a ordenação por intercalação. Cada um deve ordenar um vetor com 10 milhões (10^7) de números inteiros. Suponha que o computador A executa 10 bilhões de instruções por segundo (10^{10}), enquanto o computador B executa apenas 10 milhões de instruções por segundo (10^7). Para tornar a diferença ainda maior, considere que o programador do computador A codifique a ordenação por intercalação diretamente em linguagem de máquina, resultando em um programa que exija $2n^2$ instruções. Considere ainda que, o programador do computador B codifique a ordenação por intercalação utilizando uma linguagem de alto nível com um compilador ineficiente, sendo que o código resultante exija $50n.\lg(n)$ instruções. Para realizar as ordenações, o computador A leva

$$\frac{2.(10^7)^2 \text{ instruções}}{10^{10} \text{ instruções/segundo}} = \text{ 20.000 segundos (mais de 5,5 horas)}$$

enquanto o computador B leva

$$\frac{50.(10^7).\lg(10^7) \text{ instruções}}{10^7 \text{ instruções/segundo}} = 1.163 \text{ segundos (menos de 20 minutos)}$$

É importante diferenciar complexidade de algoritmos e complexidade de problemas. A determinação de complexidade de problemas serve para determinar se é possível encontrar algoritmos eficientes para determinado problema, enquanto a complexidade de algoritmos serve para determinar qual algoritmo é mais eficiente para resolver determinado problema. O objetivo principal desta disciplina é prover ferramentas e técnicas matemáticas para analisar e determinar a eficiência de algoritmos.

Bibliografia

[CRLS] CORMEN, T. H. et al. Algoritmos: Teoria e Prática. Elsevier, 2012. 3a Ed. Capítulo 1 (O Papel dos Algoritmos na Computação)

2 Aula 2

2.1 Análise de Algoritmos

Analisar um algoritmo significa prever os recursos que o algoritmo necessita para executar. Embora seja importante prever custos de memória, o custo do tempo de computação é frequentemente a principal preocupação durante o projeto de um algoritmo. Por meio de análise de algoritmos é possível escolher o mais eficiente para resolver determinado problema, mesmo antes de implementá-los.

Neste curso será considerado um modelo de computação genérico baseada em máquinas com memória de acesso aleatório (RAM) com um único processador. As instruções consideradas são: instruções aritméticas, movimentação de dados e de controle. Consideramos também que cada instrução leva tempo constante para executar.

Análise da ordenação por inserção

O tempo necessário para a execução da ordenação (*InsertionSort*) por inserção depende da entrada: ordenar mil elementos demora menos que ordenar cinco mil elementos. Além disto, o *InsertionSort* pode demorar quantidades de tempo diferente para ordenar duas sequencias de mesmo tamanho, dependendo do quanto elas já estejam ordenadas. Em geral, o tempo gasto por um algoritmo depende do tamanho de sua entrada. Assim, é necessário definir

Definição 8. **Tamanho da Entrada** – Uma medida da quantidade de dados da entrada.

Normalmente a noção do tamanho da entrada depende do problema que está sendo estudado. No caso de vários problemas, como a ordenação, a medida mais natural é o número de itens na entrada, ou seja, a quantidade de elementos a serem ordenados. Para outros problemas, como a multiplicação de inteiros, dependem do número total de bits neessários. Em problemas representados por grafos, o tamanho da entrada é dada pela cardinalidade dos conjuntos de vértices e arestas. Em geral, quando o tamanho da entrada é descrito por uma única variável, denominamos essa variável de n.

Definição 9. **Tempo de Execução** – Para determinada entrada, o tempo de execução é o número de operações primitivas (passos) executados.

É importante definir os passos de maneira mais independente de máquina possível. Inicialmente consideremos cada passo como a execução de uma linha de pseudocódigo executando em tempo constante. Uma linha pode demorar uma quantidade de tempo diferente de outra linha, mas consideremos que cada execução da i-ésima linha leva tempo c_i , o qual c_i é constante. Estas considerações estão de acordo com a máquina RAM discutida anteriormente, onde cada instrução é executada em tempo constante. O Algoritmo 1 apresenta o InsertionSort, juntamente com o custo de execução de cada linha, representado pelo custo de tempo de execução de cada linha e a quantidade de vezes que cada linha é executada. Para cada $j = 2, 3, 4, ..., n$, onde $n = A.comprimento$ (quantidade de elementos no vetor), seja t_j o número de vezes que o teste do laço **while** na linha 5 é executado para aquele valor de j. Quando um laço **for** ou **while** termina de maneira usual, isto é, devido ao teste no cabeçalho do laço, o teste é executado uma vez a mais do que o corpo do laço.

Algorithm 1 Ordenação por Inserção e o custo de cada linha

```
1:  procedure InsertionSort(A)
2:      for j = 2 toA.comprimento do                          ▷ c2.n
3:          chave = A[j]                                      ▷ c3.(n − 1)
4:          i = j - 1                                         ▷ c4.(n − 1)
5:          while i > 0 e A[i] > chave do                     ▷ c5. Σ_{j=2}^{n} t_j
6:              A[i+1] = A[i]                                 ▷ c6. Σ_{j=2}^{n} (t_j − 1)
7:              i = i - 1                                     ▷ c7. Σ_{j=2}^{n} (t_j − 1)
8:          end while
9:          A[i+1] = chave                                    ▷ c9.(n − 1)
10:     end for
11: end procedure
```

Assim, o tempo de execução do algoritmo é a soma dos tempos de execução para cada linha executada; uma linha demanda c_i passos para ser executada e é executada n vezes contribuirá com $ci.n$ para o tempo de execução total. Para calcular $T(n)$, o tempo de execução do *InsertionSort* com um vetor de entrada com n posições, somamos o custo de cada linha, obtendo

$$T(n) = c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\sum_{j=2}^{n} t_j + c_6.\sum_{j=2}^{n}(t_j - 1) + c_7.\sum_{j=2}^{n}(t_j - 1) + c_9.(n-1)$$

Mesmo para diferentes instâncias de mesmo tamanho, o tempo de execução pode variar de acordo com a permutação dos elementos da entrada. Por exemplo, no *InsertionSort* o melhor caso ocorre quando o vetor já está ordenado. Assim, para todo $j = 2, 3, .., n$, $A[i] \leq chave$ na linha 5 quando i vale $j - 1$. Desta forma, como $t_j = 1$ para $j = 2, .., n$, o custo do melhor caso é dado por:

$$\begin{aligned} T(n) &= c_2.n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) \\ &= (c_2 + c_3 + c_4 + c_5 + c_9)n - (c_3 + c_4 + c_5 + c_9) \end{aligned}$$

Que pode ser expressada como $an + b$ para constantes a e b dependentes dos custos de instrução c_i. Assim, este custo é uma `função linear` de n.

Por outro lado, caso o arranjo estiver ordenado em ordem inversa, ou seja, em ordem descrescente, ocorre o pior caso. Neste caso, é necessário comparar cada elemento $A[j]$ com todos os elementos do subarranjo ordenado $A[1..j-1]$, portanto $t_j = j$ para $j = 2, .., n$. Observando que

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$
e
$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

o tempo de execução do pior caso é dado por:

$$\begin{aligned} T(n) &= c_2.n + c_3(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_9(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_2 + c_3 + c_4 + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + c_9\right)n - (c_3 + c_4 + c_5 + c_9) \end{aligned}$$

Que pode ser expressada por $an^2 + bn + c$ para constantes a, b e c dependentes dos custos de instrução c_i. Assim, este custo é uma `função quadrática` de n.

No restante desta disciplina serão abordadas apenas a análise para cálculo do tempo de execução do pior caso, ou seja, determinarems o tempo de execução mais longo para qualquer entrada de tamanho n. Três razões para isto:

- O tempo para o pior caso de um algoritmo estabelece um limite superior para o tempo de execução de qualquer entrada.
- Para alguns algoritmos o pior caso ocorre com bastante frequencia.
- Na maioria dos algoritmos, o caso médio é quase tão ruim quanto o pior caso.

Bibliografia

[CRLS] CORMEN, T. H. et al. Algoritmos: Teoria e Prática. Elsevier, 2012. 3a Ed. Capítulo 2 (Dando a Partida), Seções 2.1 e 2.2.