Complexidade teórica de algoritmos[1][2]

1 Complexidade teórica

A análise teórica da complexidade de algoritmos parte de alguns princípios básicos:

1. **Modelo abstrato** – para uma análise idônea, é necessário o uso de modelos que abstraiam detalhes técnicos como (i) a capacidade do programador que implementa o algoritmo, (ii) o nível de velocidade oferecido pela linguagem adotada para a implementação do algoritmo[3] e (iii) o poder de processamento oferecido pelo *hardware* onde se executa o algoritmo. Para atender a estes requisitos, a análise teórica de complexidade é feita com pseudocódigos, simulando sua execução em uma máquina abstrata[4].

2. **Equivalência de instruções** – a principal característica do modelo de máquina abstrata adotada na análise teórica de complexidade é a equivalência entre os tempos de instrução. Especificamente, supõe-se que operações que na prática apresentam variações consideráveis de duração apresentem tempos similares no modelo teórico. Assim, operações como atribuição, comparação e aritmética podem ser analisadas como equivalentes em termos de tempo computacional.

3. **Análise em função da entrada** – uma segunda característica do modelo de máquina adotado é a abstração do tempo computacional necessário para uma instrução. Mais precisamente, o alvo da análise teórica não é mensurar o tempo computacional necessário para a execução de um algoritmo, mas descrever a quantidade de instruções necessárias em função do tamanho da entrada apresentada. Além disso, é importante considerar os diferentes tipos de entrada aos quais um mesmo algoritmo pode ser apresentado. Assim, a análise téorica de complexidade é feita para diferentes casos, conhecidos como o *melhor*, o *pior* e o caso *médio*[5].

2 Complexidade assintótica

A análise da complexidade teórica de algoritmos faz uso da noção matemática de crescimento assintótico de funções. Em síntese, dados dois algoritmos A_1 e A_2 e um dos casos que se deseja analisar (melhor ou pior), identifica-se primeiro as funções que descrevem a quantidade de instruções que cada um requer para sua execução, parametrizadas pelo tamanho da entrada (isto é, $f_{A_1}(n)$ e $f_{A_2}(n)$). Em seguida, adota-se a noção de crescimento assintótico para comparar a velocidade com a qual estas funções crescem à medida que o valor de n aumenta.

As principais notações assintóticas utilizadas neste tipo de análise são descritas a seguir.

Notação O (limite superior), utilizada para delimitar superiormente o crescimento de funções. Diz-se que uma função $f \in O(g)$, isto é, que ela tem seu crescimento limitado superiormente por g, se, e somente se, existirem uma constante c e um limite positivo n_0 tais que, para quaisquer valores $n > n_0$, $f(n) \leq c \cdot g(n)$.

[1] Roteiro de estudo fornecido na disciplina de Estruturas de Dados Básicas 1 (EDB1) da Universidade Federal do Rio Grande do Norte (UFRN).

[2] Autor: Leonardo Bezerra.

[3] Linguagens *compiladas* tradicionalmente executam mais rápido que linguagens *interpretadas*, uma vez que o processo de tradução para a linguagem de máquina é feito separadamente da execução do algoritmo.

[4] O modelo RAM, que será estudado em mais detalhes em outras disciplinas.

[5] A análise de caso médio será tópico de outras disciplinas, dadas as ferramentas matemáticas necessárias para sua execução.

Notação Ω (limite inferior), utilizada para delimitar inferiormente o crescimento de funções. Diz-se que uma função $f \in \Omega(g)$, isto é, que ela tem seu crescimento limitado inferiormente por g, se, e somente se, existirem uma constante c e um limite positivo n_0 tais que, para quaisquer valores $n > n_0$, $f(n) \geq c \cdot g(n)$.

Notação Θ (crescimento equivalente), utilizada quando uma função tem seu crescimento similar à outra, isto é, ela pode ser delimitada superiormente e inferiormente pela mesma função. Diz-se que uma função $f \in \Theta(g)$, isto é, que ela tem seu crescimento limitado superiormente e inferiormente por g ($f \in O(g)$ e $f \in \Omega(g)$), se, e somente se, existirem constantes c_1 e c_2 e limites positivos n_1 e n_2 tais que, para quaisquer valores $n > n_1$, $f(n) \leq c_1 \cdot g(n)$ e, para quaisquer valores $n > n_2$, $f(n) \geq c_2 \cdot g(n)$.

3 Padrões de crescimento de funções

A análise da complexidade assintótica de algoritmos é particularmente importante para o projeto e análise comparativa de algoritmos. Uma vez que os padrões de crescimento de funções mais comuns são bem conhecidos, torna-se possível examinar a eficiência de algoritmos e prever seu comportamento em diferentes tamanhos de instâncias. A Tabela 1.1 mostra as principais ordens de crescimento em ordem crescente, sendo a complexidade constante a ideal e a fatorial a mais elevada.

Função	**Ordem**
1	Constante
$log\, n$	Logarítmica
n	Linear
$n\, log\, n$	Logarítmica linear
n^2	Quadrática
n^3	Cúbica
2^n	Exponencial
$n!$	Fatorial

Table 1.1: Padrões de crescimento de funções.

4 Utilizando os conceitos de complexidade assintótica

Neste roteiro, você deverá por em prática os conhecimentos discutidos em sala e neste documento. Mais especificamente, solicita-se que você (i) apresente o pseudocódigo de um algoritmo projetado para o problema enunciado abaixo, (ii) apresente sua análise assintótica de complexidade pro pior e pro melhor caso, e (iii) implemente o seu algoritmo, adicionando-o ao seu repositório *U01* desta disciplina.

4.1 Enunciado do problema

Sejam os parâmetros de entrada de um problema de remoção de elementos em vetores de inteiros:

- v, um vetor de inteiros não-ordenado e cuja ordem dos elementos não importa,
- c, uma chave a ser removida, caso esteja presente em v, e
- n, o tamanho de v.

Apresente um algoritmo que remova, de forma eficiente, a chave c do vetor v de tamanho n, e retorne o tamanho do novo vetor.

4.2 Análise assintótica

Para a análise teórica da complexidade do algoritmo apresentado, identifique primeiro os casos que levam o algoritmo ao seu melhor e pior desempenho. A análise deverá ser enviada em PDF pelo SIGAA juntamente com o pseudocódigo do algoritmo, e deverá conter os seguintes passos:

Para cada caso (melhor e pior),

1. Descreva as características da entrada;
2. Apresente a função de complexidade do algoritmo, isto é, a função que descreve a quantidade de instruções que o mesmo requer em função do tamanho da entrada;
3. Identifique o comportamento assintótico desta função, utilizando as três notações apresentadas em sala e neste documento (O, Ω e Θ).

4.3 Detalhes da implementação

Seguindo a organização do projeto apresentada nas aulas anteriores, seu procedimento deverá ser implementado em três arquivos:

1. **include/remocao.h** – o arquivo-cabeçalho, contendo a assinatura (protótipo) da função.
2. **src/remocao.c** – o arquivo-fonte, contendo a implementação da função.
3. **test/t_remocao.c** – o arquivo-teste, contendo os testes unitários desta função.

Estes arquivos deverão ser adicionados ao repositório *U01* e sincronizados com o GitLab até o término da aula.