ESTRUTURA DE DADOS

AULA 1

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

O objetivo desta aula é introduzir os principais conceitos inerentes a esta disciplina. Iniciaremos conceituando a palavra-chave de nossa disciplina: *estruturas de dados*, bem como os conceitos atrelados a ela.

Esta aula se inicia com as técnicas de análise de algoritmos. Você aprenderá como mensurar a complexidade e o desempenho de um código e verá como encontrar a *função custo* de um determinado algoritmo e o que é e como fazer uma *análise assintótica*.

A aula se encerra com o assunto de recursividade, que é uma técnica de programação que visa melhorar o desempenho de algoritmos em algumas situações.

Ao longo de toda a aula serão apresentados exemplos de algoritmos escritos tanto em pseudocódigo quanto em linguagem de programação.

TEMA 1 – DEFINIÇÃO: ESTRUTURA DE DADOS

Ao projetarmos um algoritmo, uma etapa fundamental é a especificação e o projeto de seus dados de entrada. Esses dados são projetados pensando-se na aplicação e apresentarão tipos distintos. Dentre os tipos primitivos de dados, citamos: inteiro, real, caractere e *booleano*. Para cada uma dessas categorias, o espaço ocupado na memória será diferente e dependerá da arquitetura da máquina, do sistema operacional e da linguagem de programação, bem como da forma de manipulação desses dados e do seu tratamento pelo programa. Os tipos de dados manipulados por um algoritmo podem ser classificados em duas categorias distintas: os dados primitivos (ou atômicos), que são dados indivisíveis, como inteiro, real, caractere ou lógico; dados complexos (ou compostos), que podem ser divisíveis em mais partes menores.

Todo o *dado atômico* é aquele no qual o conjunto de dados manipulados é indivisível, ou seja, você trata-os como sendo um único valor. Como exemplo de dados atômicos, veja o algoritmo da Figura 1. Nele, dois valores simples são manipulados por meio de uma soma cujo resultado é salvo em outra variável, também atômica. Todos os dados são do tipo inteiro e tratados, na memória, de forma não divisível.

Figura 1 – Exemplo de código com dados simples ou atômicos.

```
1  algoritmo "AULA1_Dados_Simples"
2  var
3     x, y, z: inteiro //DADOS SIMPLES
4  inicio
5     x = 5
6     y = 1
7     z = x + y  //MANIPULAÇÃO SIMPLES
8  fimalgoritmo
```

Fonte: Elaborado pelo autor.

Por sua vez, *dados complexos* são aqueles cujos elementos do conjunto de valores podem ser decompostos em partes mais simples. Se um dado pode ser dividido, isso significa que ele apresenta algum tipo de organização estruturada e, portanto, é chamado de *dado estruturado*, o qual faz parte de uma estrutura de dados.

As estruturas de dados podem ser homogêneas ou heterogêneas.

1.1 Estruturas de dados homogêneas

Estruturas de dados homogêneas são aquelas que manipulam um só tipo de dado. Você já deve ter tido contato com esse tipo de dado ao longo de seus estudos anteriores em programação. *Vetores* são estruturas homogêneas de dados**.** Um vetor será sempre unidimensional. Se desejarmos trabalhar com duas dimensões, trabalhamos com *matrizes* e, mais do que duas dimensões denominamos de *tensores*. Vamos relembrar a seguir um pouco sobre vetores e matrizes.

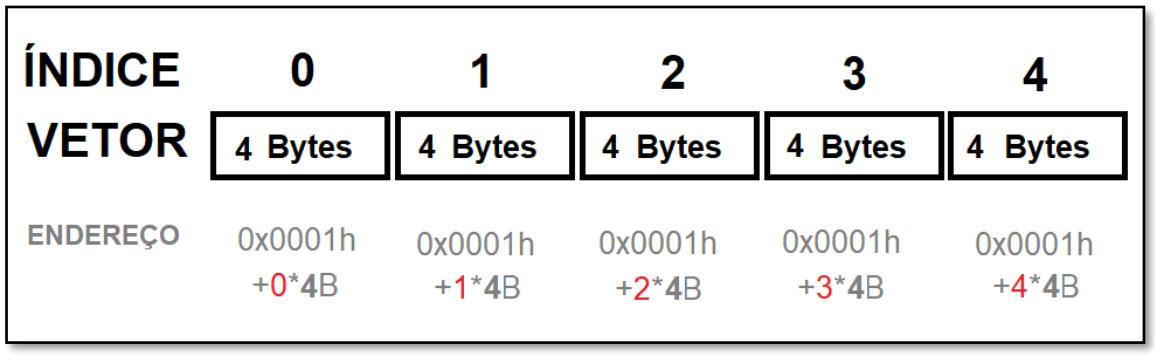
1.1.1 Vetores

É um tipo de estrutura de dados linear que necessita de somente um índice para indexação dos endereços dos elementos. O vetor contém um número fixo de células. Cada célula armazenará um único valor, sendo todos estes valores do mesmo tipo de dados.

Quando declaramos uma estrutura de vetor, na memória do programa ele é inicializado (alocado) a partir da primeira posição (endereço da primeira célula). Cada outra célula, a partir da segunda, possui um endereço de referência relativo à primeira célula endereçada. Esse endereço é calculado considerando a posição da primeira célula, acrescido do tamanho em *bytes* de cada célula, tamanho que depende do tipo de dado armazenado no vetor. Chamamos isto de *alocação sequencial.*

Observe na Figura 2 um vetor de valores inteiros de dimensão, ou comprimento, igual a 5, ou seja, contendo 5 células. Sua inicialização na memória é dada pela primeira posição desse vetor, nesse caso, no endereço 0x0001h. Assumimos que o vetor homogêneo é do tipo inteiro de tamanho 4 *bytes* por célula, considerando uma arquitetura Windows 64bits e um programa desenvolvido e compilado em linguagem C. A segunda posição (célula) está alocada, portanto, na posição da memória 0x0001h + 4 *bytes*. A terceira posição (célula) está alocada, portanto, na posição da memória 0x0001h + 2*4 *Bytes*. E assim por diante, até a última posição do vetor, o qual contém um tamanho fixo e conhecido previamente.

Figura 2 – Vetor de inteiros de tamanho 5



Fonte: Elaborado pelo autor.

Podemos generalizar a forma de cálculo de cada posição na memória de um vetor pela Equação 1:

$$Endereço_n = Endereço_0 + (Índice * Tamanho_{Bytes}), \quad (1)$$

em que $Endereço_0$ é o endereço conhecido da primeira posição do vetor, índice é a posição de cada célula e $Tamanho_{Bytes}$ é o tamanho de cada célula em *bytes,* neste exemplo, 4 *Bytes.*

1.1.2 Matrizes

Uma matriz é uma estrutura de dados homogênea, linear, com alocação sequencial e bidimensional.

Para cada uma das dimensões da matriz, é necessário utilizar um índice diferente. Considerando as duas dimensões, precisaremos de um índice, *i,* para tratar das linhas e um índice, *j,* para tratar as colunas da matriz. A analogia de *i* e *j* será adotada ao longo deste material para manter a igualdade com a simbologia adotada na matemática.

Cada célula desta matriz poderá ser acessada por meio de um endereço da memória obtido, sequencialmente, com base no sistema de indexação dupla. Considerando cada posição da matriz um arranjo *[i,j]* de linhas e colunas, a primeira posição desta matriz é a posição [0,0]. Podemos generalizar a forma de cálculo de cada posição na memória de uma matriz pela Equação 2:

$$Endereço_{i,j} = Endereço_{0,0} + \left(Índice_{linha} * C * Tamanho_{Bytes}\right) + \left(índice_{coluna} * Tamanho_{Bytes}\right), \quad (2)$$

em que $Endereço_{0,0}$ é o endereço conhecido da primeira posição da matriz, em que $Índice_{linha}$ é o índice da linha, $Índice_{coluna}$ é o índice da coluna, *C* é a quantidade de colunas por linhas e $Tamanho_{Bytes}$ é o tamanho de cada célula em *Bytes,* neste exemplo, 4 *Bytes.* É válido observar que o tamanho de cada célula depende do tipo de dados adotado, bem como da arquitetura do sistema e da linguagem de programação.

Também podemos afirmar que cada dimensão de uma matriz é na verdade um vetor. Em outras palavras, uma matriz é um vetor de vetor. Considerando nosso exemplo bidimensional, nossa matriz é composta, portanto, por dois vetores.

Saiba mais

Para um maior aprofundamento sobre este assunto siga a leitura do capítulo 1 do livro a seguir.

LAUREANO, M. **Estrutura de dados com algoritmos E C**. São Paulo: Brasport, 2008.

1.2 Estruturas de dados heterogêneas

Vimos, até então, somente estruturas de dados que manipulam um único tipo de dado, porém existem estruturas capazes de manipular mais do que um tipo, por exemplo, empregando dados inteiros e caracteres, simultaneamente em uma só estrutura. São as chamadas *estruturas de dados heterogêneas*, também conhecidas como *registros* (ou *structs* na linguagem de programação C).

Um exemplo de registro poderia ser um cadastro de pessoas em uma agenda de contatos. Cada pessoa deve conter um nome e um *e-mail* para contato (variáveis do tipo caractere) e também uma data de nascimento (variável numérica e inteira). Esses registros de contatos são conjuntos de posições que ficam armazenados em uma só variável referente ao cadastro de contatos. Porém, cada novo cadastro está em uma posição distinta da memória. Ainda, cada item do cadastro (nome, *e-mail* e data) é identificado por meio de um campo, que é um subíndice dentro de cada índice do registro. É válido ressaltar que é possível que tenhamos dentro de cada registro, até mesmo vetores e/ou matrizes, ou seja, estruturas homogêneas dentro de uma heterogênea.

1.3 Resumo e conclusões

Vetores, matrizes e registros são estruturas de dados já estudadas anteriormente ao longo do seu aprendizado em programação. Caso tenha necessidade de revisar estes conteúdos e relembrar como emprega-los em programação, recomendo que refaça a leitura de um destes três livros: (Ascencio, 2012), (Puga; Risseti 2016), (Mizrahi, 2008).

Ao longo deste curso você aprenderá a manipular estruturas de dados a seu favor. Verá como ordenar e classificar dados inseridos e também como pesquisar as informações desejadas (Tema 2 desta aula).

Além disso, aprenderá diversos algoritmos para resolver os problemas de manipulação de dados e aprenderá a encontrar qual deles será o mais eficiente para cada caso, empregando a chamada *análise assintótica de algoritmos*, ainda no Tema 1.

Existem também estruturas de dados mais complexas e com capacidade de solucionar problemas mais aprofundados, e que muitas vezes se utilizam de outras estruturas mais simples (como vetores e registros) para formar as estruturas de dados mais complexas.

TEMA 2 – ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

Quando desenvolvemos algoritmos, as possibilidades de solução do problema tendem a ser bastante grandes e distintas. É costumeiro dizermos que nenhum programador irá desenvolver um algoritmo exatamente igual a outro e, consequentemente, teremos programas com desempenhos diferentes.

Ao longo desta disciplina, iremos aprender diversos algoritmos distintos para solucionar problemas envolvendo manipulação de estruturas de dados. Desse modo, como poderemos saber qual deles é o mais eficiente para solucionar um determinado problema? Que parâmetros de desempenho devemos analisar?

Ao analisarmos o desempenho de um algoritmo, existem dois parâmetros que precisam ser observados:

- *Tempo de execução* – quando tempo um código levou para ser executado;
- *Uso de memória volátil* – a quantidade de espaço ocupado na memória principal do computador;

Acerca do tempo de execução, um fator bastante relevante nesse parâmetro é o tamanho do conjunto de dados de entrada. Vamos assumir que tenhamos uma estrutura de dados homogênea do tipo vetor preenchido aleatoriamente com valores do tipo inteiro. Agora, precisamos desenvolver um algoritmo capaz de ordenar esses algarismos do menor para o maior valor.

Caso nosso vetor tenha uma pequena dimensão (10, 50 ou 100 valores por exemplo), algoritmos que venham a ser desenvolvidos para fazer essa

ordenação terão pouco impacto no tempo de execução. Isso ocorre porque o nosso conjunto de dados de entrada é bastante pequeno.

Agora, caso extrapolarmos o tamanho do nosso conjunto de dados de entrada para, digamos, um milhão, ou um bilhão, e quisermos novamente ordenar esses dados no menor até o maior, o tempo de execução de nosso algoritmo aumentará bastante. Nesse caso, o impacto de um código mais eficiente resultará em um tempo de execução muito inferior.

Com essa análise, podemos concluir que, quanto maior nosso conjunto de dados de entrada, maior tenderá a ser o impacto de nosso algoritmo no tempo de sua execução, tornando essencial um algoritmo eficaz para a execução daquela tarefa.

2.1 Função de custo do algoritmo

O custo em tempo de execução de um algoritmo é o tempo que ele demora para encerrar a sua execução. Podemos medir de forma empírica esse tempo de execução. As linguagens de programação, e até o próprio compilador, fornecem recursos e ferramentas capazes de mensurar esses tempos.

Observe que fazer esse tipo de análise empírica pode ser trabalhosa e pouco confiável. Ao realizar esses testes empíricos, o conjunto de instruções do microprocessador está executando o código já compilado. A arquitetura da máquina, o ambiente em que o programa será executado e até a própria construção do compilador podem influenciar no tempo de execução.

Como forma de abstrair nossa análise do *hardware* e de *softwares* que são alheios ao nosso desenvolvimento, podemos encontrar matematicamente o *custo* de um algoritmo, encontrando uma equação que descreve o seu comportamento em relação ao desempenho do algoritmo. Encontrar esse custo é prever os recursos que o algoritmo utilizará. A função custo *T(n)* de um algoritmo qualquer pode ser dada como:

$$T(n) = T_{tempo} + T_{espaço}, \quad (3)$$

em que T_{tempo} é o custo em tempo de execução e $T_{espaço}$ é o custo em uso de memória pelo programa. Ao longo desta disciplina faremos a análise matemática acerca do custo em tempo de execução, desprezando o custo de

memória, optando por uma análise mais comum em ambientes de desenvolvimento.

Para encontrarmos esse custo em tempo de execução, consideramos as seguintes restrições em nossas análises:

- Nossos códigos estarão rodando em um, e somente um, microprocessador por vez;
- Não existirão operações concorrentes, somente sequenciais;
- Consideraremos que todas as instruções do programa contêm um custo unitário, fixo e constante.

Uma instrução será uma operação ou manipulação simples realizada pelo programa, como: atribuição de valores, acesso a valores de variáveis, comparação de valores e operações aritméticas básicas.

Considerando as restrições citadas, observemos o código a seguir, que é um recorte contendo uma atribuição de uma variável e um laço de repetição vazio do tipo PARA.

Figura 3 – Código exemplo para contagem de instruções

```
1  algoritmo "AULA1_Laço_Vazio"
2  var
3     i, n: inteiro
4  inicio
5     n = 10
6     para i de 0 até n faça
7        //LAÇO VAZIO
8     fimpara
9  fimalgoritmo
```

Fonte: Elaborado pelo autor.

Neste código temos diversas instruções que podem ser contadas de forma unitária para determinarmos a sua função de custo. A Tabela I resume todas as instruções unitárias, seus significados e respectiva linha em que aparecem.

Tabela 1 – Contagem de instruções unitárias do código apresentado na Figura 3.

Linha	*Motivo*	*Total de Instruções*
3	Atribuição de valor	1
4	Atribuição de valor (i=0) e comparação (i<n)	2
4-loop	Comparação (i<n) e incremento (i++)	2n

Fonte: Elaborado pelo autor.

Algumas dessas instruções ocorrem somente uma vez no programa. É o caso da linha 3, linha 4 (primeira vez). Todas estas linhas totalizam 3 instruções. Porém, na linha 4 temos o laço de repetição. Esse laço será executado enquanto sua condição for satisfeita. Para cada execução dele, perceba que teremos duas instruções executando, portanto serão $2n$, em que n é o número de vezes em que o laço executa menos um, pois, nesse caso, a primeira execução sempre acontecerá independentemente de qualquer coisa. Com isso, definimos que a função custo de tempo de execução do algoritmo da Figura 3 é dado pela expressão da Equação 4.

$$T(n) = 2n + 3 \tag{4}$$

Vamos agora analisar um segundo código um pouco mais complexo. O código da Figura 4 encontra o maior valor dentro de um vetor de dimensão variável.

Figura 4 – Código exemplo que busca o maior valor dentro de um vetor

```
1  algoritmo "AULA1_Maior_Valor"
2  var
3     i, Maior: inteiro
4     Valores: vetor [1..10] de inteiro
5  inicio
6
7     Maior = 0
8     para i de 0 até 10 faça
9        se (Valores[i] >= Maior) //Testa se o valor do vetor é maior
10          Maior = Valores[i] //Atribui o valor como sendo o maior
11       fimse
12    fimpara
13
14 fimalgoritmo
```

Fonte: Elaborado pelo autor.

Observe que temos duas linhas de código a mais que o exemplo da Figura 3: uma condicional (linha 9) e uma atribuição (linha 10), e que ambas as linhas são executadas *n* vezes dentro do laço de repetição, ou seja, estão atreladas a ele.

Nesse código, temos diversas instruções que podem ser contadas de forma unitária para determinarmos a função de custo deste. A Tabela II resume todas as instruções unitárias, seus significados e respectiva linha em que aparecem.

Tabela 2 – Contagem de instruções unitárias do código apresentado na Figura 4.

Linha	***Motivo***	***Total de Instruções***
7	Atribuição de valor	1
8	Atribuição de valor (i=0) e comparação (i<n)	2
8-loop	Comparação (i<n) e incremento (i++)	2n
9	Comparação	1
10	Atribuição de valor	1

Fonte: Elaborado pelo autor.

Observe que na linha 9 temos um teste condicional. Caso essa condição seja satisfeita, ou seja, sua resposta seja verdadeira, significa que a linha 10 será executada. Caso ela seja falsa, essa linha não será executada. Perceba então que a linha 10 pode, ou não, ser executada, e quanto menos vezes isso acontecer, menor o tempo de execução deste algoritmo (menos execuções de instruções).

E não só isso, mas a ordem em que os valores do conjunto de entrada de dados estiverem inseridos implicará diretamente o desempenho desse algoritmo. Vejamos duas possibilidades a seguir para um vetor de dimensão 4 para compreendermos melhor essas possibilidades:

a. **Vetor ordenado de forma crescente:** aplicando o algoritmo da Figura 4 com o vetor de entrada da Equação 5, crescente, este vetor de entrada implicará a condicional da linha 9 sendo verdadeira todas as vezes em que for testada, e consequentemente a linha 10 sendo executada sempre também. Portanto, considerando o nosso tempo de execução, e tendo uma linha de código a mais sendo executado todas as vezes (linha 10), essa situação pode ser chamada de *situação do pior caso* do algoritmo.

Essa situação é considerada o pior caso, pois é quando temos o maior número de instruções sendo executadas antes que o algoritmo se encerre, implicando diretamente o tempo de execução desse código.

$$Vetor = [1, 2, 3, 4]; \tag{5}$$

A função custo para essa situação é dada pela Equação 6.

$$T(n) = 4n + 3 \tag{6}$$

b. Vetor ordenado de forma decrescente: aplicando o algoritmo da Figura 4 com o vetor de entrada da Equação 7, implicará a linha 9 sendo falsa todas as vezes (exceto a primeira pois a variável *Maior* estará com o valor zero), e consequentemente a linha 10 nunca sendo executada, exceto uma vez. Portanto, considerando o tempo de execução e tendo uma linha de código que não é executada, essa situação pode ser chamada de *situação do melhor caso* do algoritmo. Esta situação é considerada o melhor caso, pois é quando temos o menor número de instruções sendo executadas antes que o algoritmo se encerre, implicando diretamente o tempo de execução deste código.

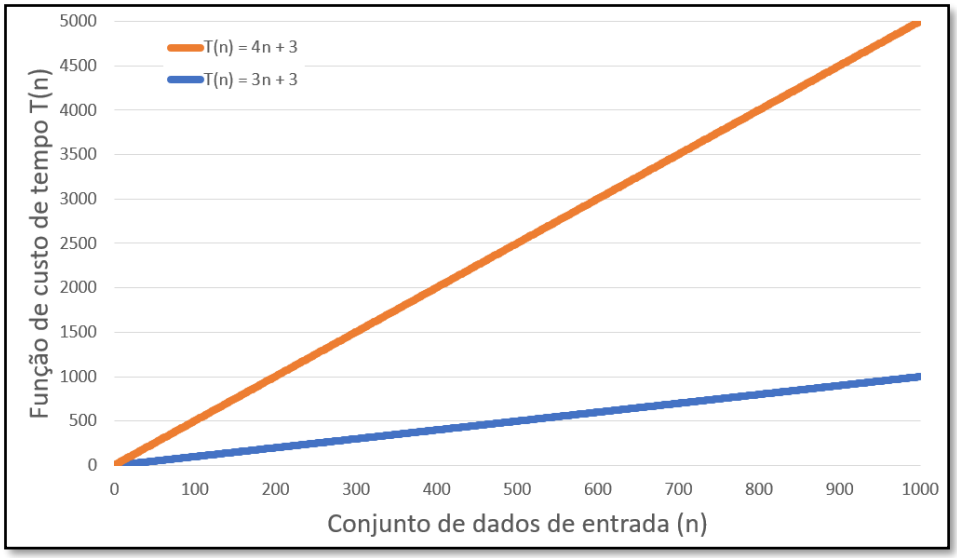
$$Vetor = [4, 3, 2, 1]; \tag{7}$$

A função custo para esta situação é dada pela Equação 8.

$$T(n) = 3n + 3 \tag{8}$$

Perceba que o *pior caso* e o *melhor caso* resultam em duas funções de custo diferentes. Se realizarmos a construção de ambas curvas, conforme a Figura 5, vemos a diferença no desempenho das duas situações. Na Figura 5, temos no eixo horizontal o tamanho do nosso conjunto de dados de entrada, que varia entre 0 até 1000 valores para nosso vetor. No eixo vertical, temos a resposta da nossa função custo. Quanto maior o custo, pior o desempenho.

Figura 5 – Comparativo gráfico da complexidade do algoritmo para o melhor e pior caso.



Fonte: Elaborado pelo autor.

O termo de maior grau em uma equação polinomial é aquele que possui o maior expoente. O impacto do coeficiente do termo de maior grau de ambas equações infere diretamente no desempenho de nossos algoritmos. Percebemos que quanto mais cresce nosso conjunto de dados de entrada, maior é a discrepância entre ambos os desempenhos.

TEMA 3 – ANÁLISE ASSINTÓTICA DE ALGORITMOS

Já entendemos um pouco sobre como mensurar de forma matemática a eficiência em tempo de execução de um algoritmo, e também alguns fatores que impactam no seu desempenho. Também vimos como é trabalhosa a realização da contagem manual de instruções até chegarmos às equações de custo apresentadas anteriormente.

Para nossa sorte, não iremos precisar fazer essa contagem minuciosa de instruções todas as vezes que precisarmos descobrir a função custo de tempo, porque podemos fazer a análise assintótica.

Nesse tipo de análise, encontraremos uma curva de tendência aproximada do desempenho de um algoritmo. A análise baseia-se na extrapolação do conjunto de dados de entrada, fazendo-os tenderem ao infinito

e permitindo que negligenciemos alguns termos de nossas equações. Em outras palavras, descartamos de nossas equações os termos que crescem lentamente à medida que nosso conjunto de dados de entrada tende ao infinito.

Para obtermos o comportamento assintótico de qualquer função, mantemos somente o termo de *maior grau* (maior crescimento) da equação, negligenciando todos os outros, inclusive o coeficiente multiplicador do termo de maior grau. Por exemplo, se pegarmos a Equação 6, negligenciamos o coeficiente de *n* e também o termo "+ 4", obtendo assim uma curva assintótica de primeiro grau (Equação 9) (Cormen, 2012).

$$T(n) = n \tag{9}$$

Na Tabela III temos funções custo de diferentes graus e seus respectivos comportamentos assintóticos. Perceba também que na terceira coluna dessa tabela foi colocado um resumo de como identificar cada um dos comportamentos somente por observação de um algoritmo. Por exemplo, um algoritmo sequencial, sem laços de repetição apresenta uma complexidade assintótico unitária, ou seja, independentemente do tamanho do conjunto de dados de entrada, seu desempenho não muda. Agora, para cada laço de repetição aninhado adicionado no seu código, um grau na complexidade é acrescido.

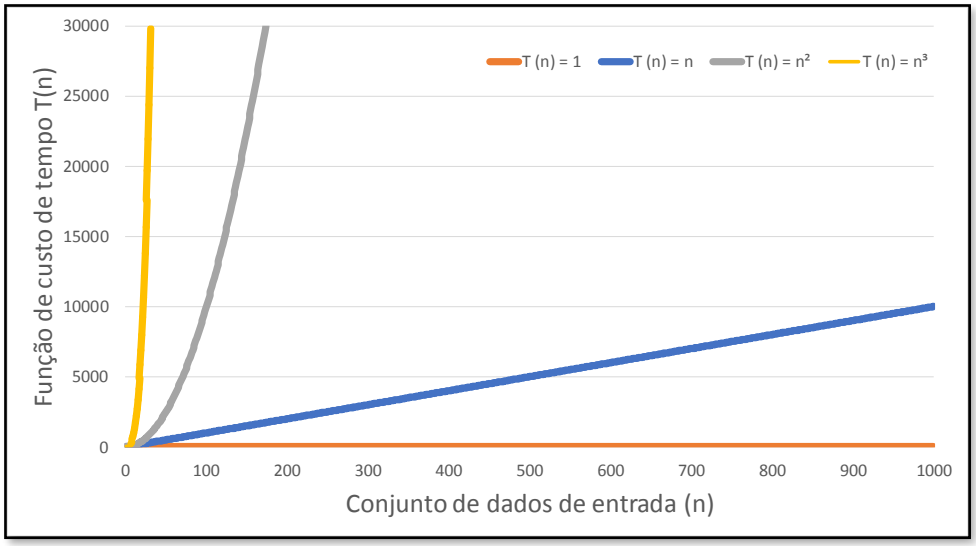
Tabela 3 – Exemplos de funções custo e seus respectivos comportamentos assintóticos.

Função Custo	Comportamento Assintótico	Algoritmo
T(n) = 10	1	Sequencial
T(n) = 10n + 2	n	1 laço
T(n) = $10n^2 + 5n + 2$	n^2	2 laços
T(n) = $10n^3 + 50n^2 + n + 1$	n^3	3 laços

Fonte: Elaborado pelo autor.

A Figura 6 ilustra diferentes comportamentos assintóticos e seus desempenhos. Quanto maior o grau de nosso algoritmo, pior tenderá a ser seu desempenho em tempo de execução. Perceba que um comportamento n^3, por exemplo, apresenta um crescimento bastante acentuado na piora de seu desempenho se comparado a graus menores, o que significa que pequenos conjuntos de dados tendem a impactar rapidamente na eficiência desse código.

Figura 6 – Comparativo da complexidade do algoritmo para diferentes casos assintóticos.



Nosso objetivo na análise de algoritmos é encontrar uma forma de construir o algoritmo de tal forma que o aumento do tempo de execução seja o menor possível em detrimento ao crescimento do conjunto de dados de entrada.

3.1 Notações da análise assintótica

Agora que já compreendemos como um comportamento assintótico é determinado e também que podemos ter diversos casos de desempenho para um mesmo algoritmo, vamos listar três notações presentes na literatura comumente utilizadas para comparar algoritmos (Ascencio, 2011). Essas notações determinam a complexidade assintótico de um código para diferentes situações, sendo estas situações dependentes do conjunto de dados.

a. GRANDE-O (*BIG-O*):

- Define o comportamento assintótico superior;
- É o pior caso de um algoritmo;
- Mais instruções sendo executadas.

b. GRANDE-ÔMEGA:

- Define o comportamento assintótico inferior;
- É o melhor caso do algoritmo (caso ótimo);
- Menos instruções sendo executadas.

c. GRANDE-TETA:

- Define o comportamento assintótico firme;
- Caso médio de um algoritmo.
- É o comportamento considerando a grande maioria dos casos.

Vamos trabalhar com o a *notação Big-O*, que é a notação mais empregada em análises de algoritmos e nos diz que um código nunca será pior do que a situação mostrada nesta notação, podendo, no entanto, ser melhor, dependendo do conjunto de dados de entrada.

Vejamos na Figura 7 um algoritmo em que temos 3 laços de repetição aninhados. Entre cada um deles, temos uma condicional. Abstraindo o teste da condicional e fazendo uma análise assintótica desses algoritmos para o melhor e o pior caso, podemos pensar na seguinte situação.

A pior situação desse algoritmo em desempenho é quando os dois testes condicionais são sempre verdadeiros, resultando nos três laços de repetição sendo executados todas as vezes. Sendo assim, teremos três laços encadeados, gerando uma complexidade de grau 3 (n^3 - Tabela III).

A melhor situação desse código é quando o primeiro teste condicional resulta em falso todas as vezes. Sendo assim, os dois laços internos nunca serão executados, sobrando somente o primeiro laço. Assim, teremos um grau 1 de complexidade, somente (n – Tabela III).

Figura 7 – Código exemplo para análise de notações.

```
1  algoritmo "AULA1_Complexidade"
2  var
3     i, j, k, n: inteiro
4  inicio
5
6     n = 10
7     para i de 0 até n faça
8
9        se (/*CONDIÇÃO 1*/)
10          para j de 0 até n faça
11
12             se (/*CONDIÇÃO 2*/)
13                para k de 0 até n faça
14                   //LAÇO VAZIO
15
16                fimpara
17             fimse
18          fimpara
19       fimse
20    fimpara
21
22 fimalgoritmo
```

Fonte: Elaborado pelo autor.

Retomando, para o pior caso, ou seja, para a notação Big-O, temos a complexidade assintótica:

$$O(n^3) \tag{10}$$

Agora, para um melhor caso, ou seja, para a notação Grande-Ômega, temos a complexidade assintótica:

$$\Omega(n) \tag{11}$$

Não esqueça de observar a Figura 6 e comparar a diferença para os dois casos no gráfico!

Um outro conceito relevante para a nossa disciplina de estrutura de dados é a recursividade. Esta técnica é muito empregada em programação e irá aparecer de forma recorrente ao longo de toda a nossa disciplina.

A recursão é o processo de definição de algo em termos de si mesmo (Laureano, 2008). Um algoritmo recursivo é aquele que utiliza a si próprio para atingir algum objetivo, ou obter algum resultado. Em programação, a recursividade está presente quando uma *função* realiza chamadas a si mesma.

Um algoritmo recursivo é muito empregado no conceito chamado “dividir para conquistar” para resolver um problema. Ele divide um problema maior em partes menores. Estas pequenas partes são solucionadas de forma mais rápida e mais eficiente. Todas as pequenas soluções são, portanto, agregadas para obter a solução maior e final para o problema.

Segundo Laureano (2008), “Se um problema em questão é grande, deve ser reduzido em problemas menores de mesma ordem, aplicando-se o método sucessivas vezes até a solução completa ser atingida”.

4.1 Complexidade do algoritmo recursivo

Para chegarmos ao entendimento da complexidade recursiva, vamos partir da observação do vetor de dados abaixo e analisar o algoritmo *iterativo*, que realiza a busca de um valor nesse vetor (Figura 8). O algoritmo iterativo é aquele que varre (perpassa), posição por posição, de nosso vetor até encontrar o valor desejado. Perceba-se, portanto, que como temos somente um laço de iteração, temos a complexidade Big-O como sendo **O(n)**.

$$Vetor = [-5, -2, -1, 0, 1, 2, 4] \quad (11)$$

Figura 8 – Algoritmo que busca um valor dentro de um vetor de forma sequencial

```
1  algoritmo "AULA1_Busca_Sequencial_Iterativa"
2  var
3     i: inteiro
4     Numeros: vetor [0..6] de inteiro
5  inicio
6
7     para i de 0 até 7 faça
8        //TESTA SE O VALOR DO VETOR É O DESEJADO
9     fimpara
10
11 fimalgoritmo
```

Fonte: Elaborado pelo autor.

Devemos fazer uma pergunta: podemos melhorar o desempenho deste algoritmo da Figura 8? Neste caso, a resposta é sim. Basta organizar o pensamento de outra forma.

Assumindo que o vetor está ordenado, conforme Equação 11, em vez de testarmos um valor por vez desde a posição 0, checaremos diretamente o valor do meio de nosso vetor e testaremos se esse valor é maior, menor ou igual ao desejado. Se o valor do meio por menor que o valor desejado, isso significa que nosso valor está em alguma posição do lado direito do ponto central; caso contrário, estará do lado esquerdo.

Considerando o vetor da Equação 11, caso queiramos buscar o valor 2 nele, saberemos que ele está em um dos últimos três valores, pois 2 é maior do que 0 (valor central). Assim, podemos, em uma segunda tentativa, pegar somente os últimos três valores, obter seu valor do meio e comparar novamente, encontrando já na segunda tentativa o valor 2. Assim, muito mais rapidamente o valor desejado é atingido sem precisarmos testar um valor de cada vez desde o começo.

O que foi feito neste segundo algoritmo foi o uso do recurso de “dividir para conquistar”, ou seja, dividimos nosso problema em partes menores, obtivemos nossos resultados menores e alcançamos nosso objetivo mais rapidamente.

Para encontrarmos matematicamente a complexidade de um algoritmo que trabalhe com esse recurso, vamos assumir um problema de tamanho n.

Cada problema menor (divisão) de nosso algoritmo será a metade do problema anterior $(n/2)$. Podemos fazer estas subdivisões até que cada pequena parte tenha tamanho 1. Assim, a cada subdivisão do nosso vetor na metade, teremos o que é mostrado na Tabela IV.

Tabela 4 – A matemática da divisão do problema em partes menores.

Problema original	n	$\boldsymbol{n/2^x}$
Passo 1	$n/2$	$n/2^1$
Passo 2	$n/4$	$n/2^2$
Passo 3	$n/8$	$n/2^3$
Passo 4	$n/16$	$n/24$

Fonte: Elaborado pelo autor.

A cada iteração vamos dividindo ainda mais nosso problema. Obtemos, de forma genérica, a partir da terceira coluna da Tabela IV, a Equação 12:

$$\frac{n}{2^k} \tag{12}$$

Considerando que quando reduzimos um algoritmo à sua menor parte possível, sua complexidade será sempre unitária (cada parte terá um custo). Podemos igualar a Equação 12 com o valor 1 (Equação 13 e 14):

$$\frac{n}{2^k} = 1 \tag{13}$$

$$n = 2^k \tag{14}$$

Agora, precisamos isolar a variável *k* em função de *n* na Equação 14 e encontrarmos a complexidade deste algoritmo. Para tal, fazemos o logaritmo de ambos os lados da equação (Equação 15) e isolamos o *k* (Equação 17):

$$\log n = \log 2^k \tag{15}$$

$$\log n = \mathrm{k}.\log 2 \tag{16}$$

$$k = \frac{\log n}{\log 2} \tag{17}$$

Agora, aplicamos uma propriedade de logaritmos (Equação 18) na Equação 17 para simplificarmos nossa equação:

$$\log_n m = \frac{\log_x m}{\log_x n} \tag{18}$$

Obtemos, portanto:

$$k = \frac{\log n}{\log 2} \rightarrow k = \log_2 n \tag{19}$$

Assim, a complexidade assintótica Big-O para este algoritmo será:

$$O(\log n) \tag{20}$$

TEMA 5 – EXEMPLO CLÁSSICO DE RECURSIVIDADE: FATORIAL

Agora que vimos que a recursividade trabalha com o conceito de "dividir para conquistar" e que sua complexidade para o pior caso é $O(logn)$, vamos analisar como construímos de fato um algoritmo recursivo e comparar seu desempenho sem a recursividade.

Um exemplo empregado para o entendimento da construção de um algoritmo recursivo é o cálculo de um fatorial. Vamos observar o código de uma função de cálculo de fatorial utilizando somente iteratividade (sem a recursão). Na Figura 9, temos uma função que realiza essa lógica. Na função, *Fatorial_Iterativo* é passado como parâmetro um valor n, inteiro. Este é o valor para o qual desejamos calcular o fatorial. Na linha 25, temos a variável que receberá o calculado da fatorial sendo inicializada com o valor. Isso ocorre porque o menor valor para uma fatorial será sempre um ($0! = 1$ e também $1! = 1$). Já nas linhas 26 e 27, temos de fato o cálculo da fatorial, com seu valor retornando na linha 27. Considerando uma lógica iterativa, a complexidade deste algoritmo será *O(n)*.

Figura 9 – Algoritmo fatorial iterativo

```
21  Fatorial_Iterativo (n: inteiro): inteiro
22  var
23     fat: inteiro
24  iniciofuncao
25     fat = 1
26     para i de 1 até n faça
27        fat = fat * i
28     fimpara
29
30     retorne fat
31  fimfuncao
```

Fonte: Elaborado pelo autor.

Na Figura 10, temos uma função que novamente a fatorial, mas agora de forma recursiva, *Fatorial_Recursivo*. Na linha 29, perceba que temos o cálculo da fatorial agora sem nenhum laço de repetição. Nesta linha, temos uma nova chamada para a função do fatorial. Isso caracteriza a *recursividade*, uma função chamando a si mesma.

Figura 10 – Algoritmo fatorial recursivo

```
21  Fatorial_Recursivo (n: inteiro): inteiro
22  var
23     fat: inteiro
24  iniciofuncao
25     fat = 1
26     se (n == 0) então
27        retorne fat
28     senão
29        fat = n * Fatorial_Recursivo (n - 1)
30     fimse
31  fimfuncao
```

Fonte: Elaborado pelo autor.

Imaginemos que queremos calcular o fatorial de 5. Quando a função de cálculo fatorial é chamada pela primeira vez, o valor é passado como parâmetro para a função. Ao chegar na linha 29, a função atual é interrompida (mas não é encerrada) e uma nova instância desta função é aberta no programa. Isso

significa que temos duas funções fatoriais abertas e alocadas na memória do programa.

Nessa segunda chamada, o valor 4 é passado como parâmetro e a função é executada novamente até chegar à linha 29. Mais uma vez, a função fatorial é chamada de *forma recursiva*, caracterizando uma terceira instância da função sendo aberta.

Seguindo essa mesma sequência, a função recursiva é chamada até que o valor 0 seja passado como parâmetro. Quando isso acontecer, isso significará que a última instância da função foi aberta. Agora, o programa começa a finalizar e encerrar cada uma das funções abertas e retornando o resultado desejado para a função que a chamou. Quando a última instância (que é a primeira aberta) é encerrada, temos a resposta de fatorial de 5 sendo mostrada na função *main* do programa. Para o exemplo citado (fatorial de 5), teremos 6 instâncias da mesma função abertas de forma simultânea (Tabela V).

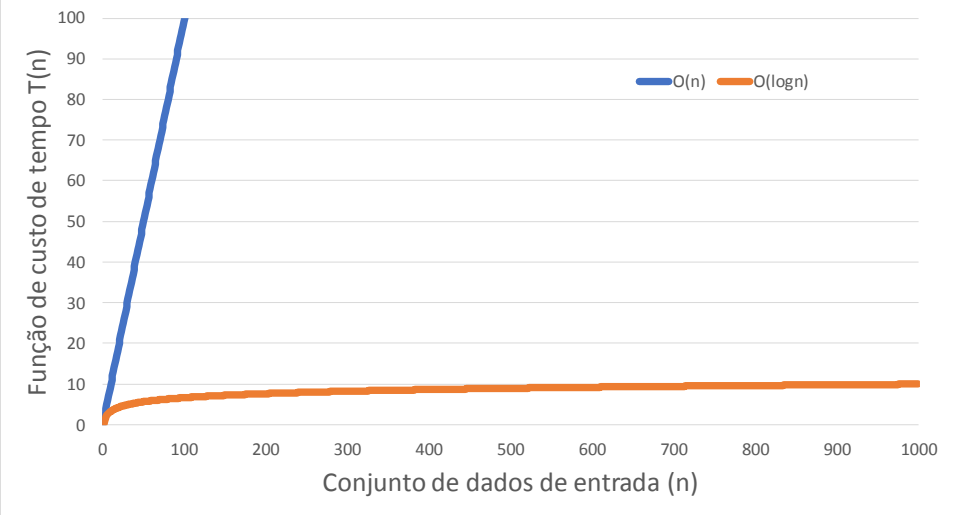
Tabela 5 – Todas as chamadas da função recursiva do fatorial

Chamada	LINHA 29
1	$Fatorial\_Recursivo(5)$
2	$5 * Fatorial\_Recursivo(4)$
3	$5 * 4 * Fatorial\_Recursivo(3)$
4	$5 * 4 * 3 * Fatorial\_Recursivo(2)$
5	$5 * 4 * 3 * 2 * Fatorial\_Recursivo(1)$
6	$5 * 4 * 3 * 2 * 1 * Fatorial\_Recursivo(0)$

Fonte: Elaborado pelo autor.

Por fim, podemos ainda observar a diferença no desempenho da fatorial iterativa *O(n)* para a fatorial recursiva *O(logn)*. A Figura 11 ilustra esta diferença. Perceba que, em tempo de execução, uma função recursiva tende a apresentar um desempenho bastante superior a uma não recursiva.

Figura 11 – Comparativo da complexidade do algoritmo fatorial iterativo com o recursivo



Fonte: Elaborado pelo autor.

Percebemos o interessante desempenho da função recursiva, porém, como o algoritmo recursivo está abrindo muitas instâncias de uma mesma função, e para cada nova instância temos suas variáveis locais sendo alocadas novamente, isso significa um aumento do uso de memória desse programa. Portanto, deve-se tomar cuidado com a possibilidade de estouro da pilha de memória em aplicações recursivas

FINALIZANDO

Nesta aula aprendemos os alicerces de nossa disciplina. Vimos o que são estruturas de dados e alguns de seus tipos. Diversas outras estruturas de dados irão aparecer no decorrer do nosso curso.

Vimos também o conceito de análise de algoritmos, notação Big-O e de recursividade e sua complexidade.

REFERÊNCIAS

ASCENCIO, A. F. G. **Estrutura de dados:** algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson, 2011.

_____. **Fundamentos da programação de computadores:** algoritmos, Pascal, C/C++ (padrão ANSI) JAVA. 3. ed. São Paulo: Pearson, 2012.

CORMEN, T. H. **Algoritmos:** teoria e prática. 3. ed. Elsevier, 2012.

LAUREANO, M. **Estrutura de dados com algoritmos E C**. São Paulo: Brasport, 2008.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Peason, 2008.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.