

N_EST DAD_A1 – Texto de Apoio

Site: [EAD Mackenzie](#)

Tema: ESTRUTURA DE DADOS {TURMA 03A} 2023/1

Livro: N_EST DAD_A1 – Texto de Apoio

Impresso por: FELIPE BALDIM GUERRA .

Data: terça, 14 fev 2023, 18:15

Índice

TIPO ABSTRATO DE DADOS

Características Fundamentais de um TAD

Características típicas de TADs

INTRODUÇÃO À ANÁLISE DE ALGORITMOS

Função de custo ou função de complexidade

Cenários

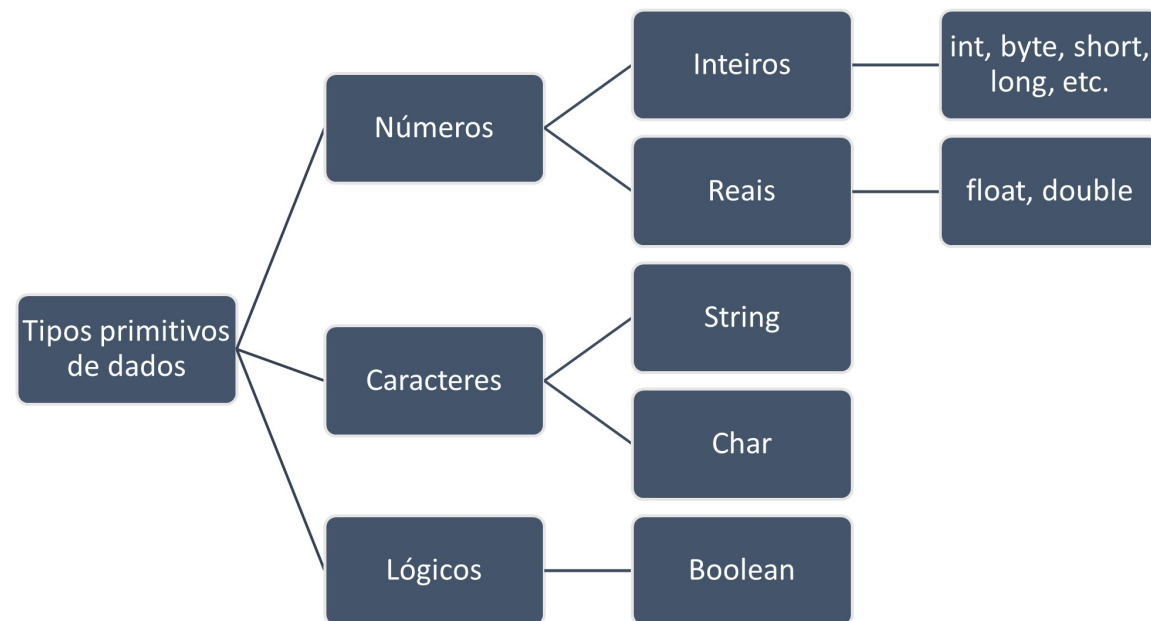
Análise assintótica

TIPO ABSTRATO DE DADOS

Os dados, em um programa, podem estar representados (estruturados) de diferentes maneiras. Normalmente, a escolha dessa representação é determinada pelas operações que serão utilizadas sobre esses dados.

Você já sabe que os dados têm um tipo, que se refere ao conjunto de valores que esses dados podem assumir ou gerar. A Figura 1 abaixo apresenta os tipos primitivos de dados ou os tipos escalares de dados. Esses tipos são manipulados pela maioria das linguagens de programação e, certamente, você já os conhece.

Figura 1 – Tipos primitivos de dados



Fonte: Elaborada pela autora.

Um dos grandes avanços ocorridos no desenvolvimento de software foi o conceito de **tipos abstratos de dados, no qual o programador pode definir seus próprios tipos de dados**. Essa ideia surgiu do trabalho de muitos pesquisadores, entre eles Ole-Johan Dahl (criador da linguagem Simula), Charles Antony Richard Hoare (que desenvolveu muitas das técnicas que utilizamos para trabalhar com tipos abstratos), David Parnas (que usou pela primeira vez o termo “escondendo informações”) e no MIT (Massachusetts Institute of Technology), Barbara Liskov e John Guttag (que realizaram um trabalho pioneiro a respeito da especificação de tipos abstratos e a respeito do suporte para tais tipos em linguagens de programação).

Assim, podemos definir que um Tipo Abstrato de Dados (TAD) é um conjunto bem definido de dados que serão armazenados e em paralelo. Definimos, então, um grupo de operações que podem ser aplicadas para manipular esses dados.

Vamos ver um primeiro exemplo?

Cadastro de funcionários

Que dados estão definidos “por trás” deste tipo?



Código, nome, CPF, data de admissão, cargo, salário bruto, comissões, descontos etc.




Quais operações podem manipular esses dados?


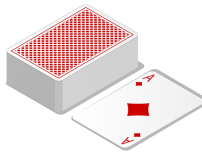


Calcular salário líquido, incluir funcionário, aumentar salário, calcular informe de rendimentos para Imposto de Renda etc.

Características Fundamentais de um TAD

As operações de um TAD implementam regras bem definidas para manipulação dos valores armazenados; os valores armazenados devem ser manipulados **EXCLUSIVAMENTE** pelas operações do TAD. A especificação de um TAD descreve quais dados podem ser armazenados e o que é possível fazer com esses dados por meio das operações do TAD. Mas a especificação do TAD não descreve como isso é ou será efetivamente implementado no programa. Isso pode ser definido em um segundo momento. Ou seja, um TAD é um modelo abstrato do armazenamento e manipulação de determinado conjunto de dados de um programa. Analise a tabela abaixo e identifique os dados e as operações em cada caso.

MUNDO REAL	DADOS	OPERAÇÕES	OBSERVAÇÕES
 Pessoa	Idade da pessoa	<ul style="list-style-type: none">• Nascer (idade recebe o valor zero).• Fazer aniversário (idade aumenta em 1).	Repare que as únicas operações possíveis com a idade de uma pessoa são: nascer, quando a idade é “zerada”, e fazer aniversário, quando a idade aumenta em 1. Não existe nenhuma operação para diminuir a idade, assim, isso nunca será possível.

 <p>Fila de espera</p>	<ul style="list-style-type: none"> • Quantidade de pessoas na fila. • Quem é a primeira pessoa da fila. • Quem é a última pessoa da fila. 	<ul style="list-style-type: none"> • Iniciar uma fila (fila vazia). • Entrar uma pessoa no final da fila. • Sair uma pessoa do início da fila. • Contar quantas pessoas têm na fila. 	<p>Elencamos apenas quatro operações para este TAD. Com esta definição, conseguimos começar uma fila “do zero”, fazer com que as pessoas entrem na fila (uma de cada vez), saiam da fila (uma de cada vez) e contar quantas pessoas estão na fila. Só temos quatro operações e nada, além disso, pode alterar esse TAD (por exemplo, tirar alguém do meio da fila).</p>
 <p>Pilha de cartas</p>	<ul style="list-style-type: none"> • Quantidade de cartas na pilha. • Carta do topo (naipe e valor). 	<ul style="list-style-type: none"> • Iniciar uma pilha de cartas (pilha vazia). • Colocar uma carta na pilha (no topo). • Tirar uma carta da pilha (do topo). • Mostrar a carta do topo. 	<p>As operações definidas nos permitem apenas inserir e remover novas cartas na pilha (sempre no topo), além de mostrar qual é a carta que está no topo. Não conseguimos, com essas operações, por exemplo, remover uma carta do meio da pilha.</p>

Características típicas de TADs

- Independe da linguagem de programação.
- É conceito básico para programação Orientada a Objetos.
- A manipulação do dado se faz unicamente por meio das operações definidas para ele.
- Não é preciso saber a forma exata de implementação do tipo ou de suas operações para conseguir utilizá-los.
- A definição é independente da implementação, podendo ser alterada sem que o uso se modifique.
- A aplicação não precisa (e não deve) ter acesso à forma de implementação. Portanto, somente após a definição estar completa é que devemos começar a pensar na implementação exata do tipo e suas operações.

INTRODUÇÃO À ANÁLISE DE ALGORITMOS

Você já sabe que um algoritmo é um conjunto finito de passos que devem ser seguidos e que atuam sobre um conjunto de dados de entrada para se chegar à solução de um problema. Esses problemas que têm um algoritmo definido são chamados de problemas computáveis. Nem sempre um problema computável é tratável, ou seja, os recursos computacionais utilizados são limitados (tempo de processamento e espaço de memória) e, portanto, podem impedir a execução de um determinado algoritmo. Por este motivo, é importante perceber que um mesmo problema pode ser resolvido por muitos algoritmos diferentes.

Se, para um mesmo problema, nós tivermos dois ou mais algoritmos, é sensato escolher aquele que trará uma solução no menor tempo possível e que utilizará o menor espaço para representação dos dados do problema (**custo do algoritmo**).

Para medir o custo de um algoritmo, podem ser usadas duas técnicas:

- executar o programa em uma máquina real; ou
- usar um modelo matemático.

No primeiro caso, pode-se perceber algumas desvantagens, uma vez que o compilador, o tipo de hardware e a quantidade de memória influenciam no tempo obtido.

Usar um modelo matemático significa analisar o conjunto de operações a serem executadas e atribuir um custo às operações mais relevantes. Desta forma, os recursos computacionais são separados do algoritmo e, assim, o custo é atribuído unicamente às instruções do mesmo.

Analisar um algoritmo significa estudar seu comportamento frente a diferentes tamanhos e valores de entrada. O que se pretende é estimar a quantidade de passos que o algoritmo deve executar frente a diferentes entradas. Desta forma, consegue-se obter a eficiência de um algoritmo.

Quando se analisa um determinado algoritmo, respostas são obtidas para as seguintes questões:

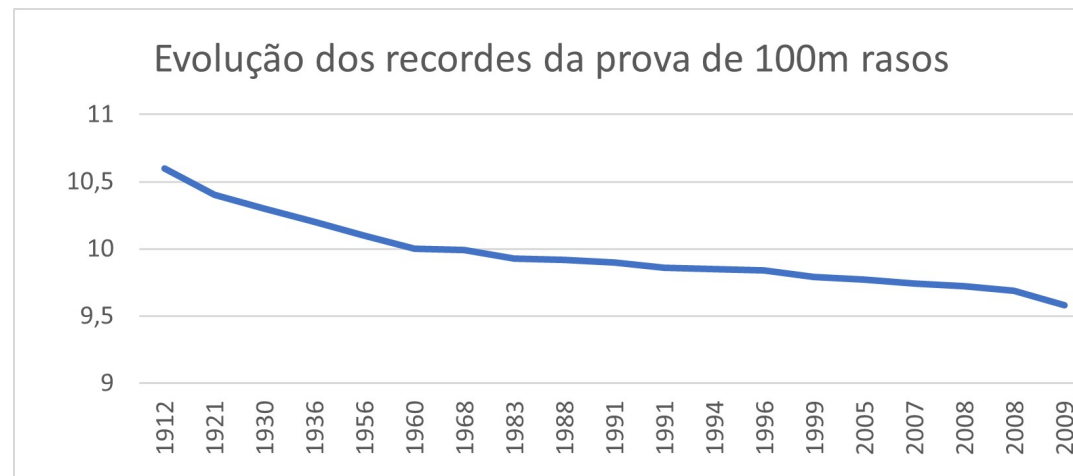
- Qual é o custo de usar um dado algoritmo para resolver um problema específico?
- Qual é o algoritmo de menor custo possível para resolver um problema particular? Exemplo: é possível estimar o número mínimo de comparações necessárias para ordenar n números por meio de comparações sucessivas?

Quando conseguimos determinar o menor custo possível (limite inferior) para resolver um problema, temos a medida da dificuldade intrínseca à resolução desse problema.

Vamos fazer uma analogia?

Você já deve ter ouvido que o recorde da prova olímpica dos 100 metros rasos vem diminuindo a cada ano. Isso acontece porque as técnicas de treinamento, os uniformes utilizados e a própria evolução da ciência têm contribuído nesse sentido. Veja o Gráfico 1 abaixo, com os tempos dessa prova, de 1912 (10s 6c) até 2009 (9s 58c), quando foi batido o último recorde.

Gráfico 1 – Evolução dos recordes da prova de 100 metros rasos de 1912 a 2009



Fonte: Elaborado pela autora.

Será que esse tempo sempre diminuirá? A resposta é não! Existe um limite inferior que é dado pela ciência e que o homem não conseguirá ultrapassar. Esse limite é de 9 segundos e 48 centésimos. Um ser humano nunca conseguirá realizar essa prova em um tempo inferior a esse. Ou seja, esse é o tempo intrínseco a esse tipo de prova (limite inferior). Quando um atleta realiza essa prova em um tempo próximo, ou igual ao limite inferior, dizemos que ele a realiza em um tempo ótimo!

Voltando para nossos problemas computacionais, cada problema tem um tempo de execução que é intrínseco ao seu tipo (limite inferior). Se você desenvolve um algoritmo para um determinado tipo de problema e o tempo (custo) é igual ao limite inferior, seu algoritmo tem um custo ÓTIMO!

Assim, dado um algoritmo A que resolve o problema P, qual função complexidade deste algoritmo? Se a complexidade de A for equivalente em ordem de grandeza ao limite inferior do problema, então, pode-se dizer que o algoritmo A é ótimo.

Função de custo ou função de complexidade

Chamamos de $f(n)$ a medida de tempo necessária para executar um algoritmo de tamanho n . Mas o que seria o tamanho de um algoritmo? É o tamanho de entrada do algoritmo que resolve o problema.

Exemplos:

- A busca em uma lista de N elementos ou a ordenação de uma lista de N elementos requerem mais operações à medida que N cresce.
- O cálculo do fatorial de N tem seu número de operações aumentado com o aumento de N .
- A determinação dos valores da série de Fibonacci envolve uma quantidade de adições proporcional ao valor de N .

É a complexidade de tempo. Neste caso, tempo, não é no sentido horário. É o número de vezes que determinada operação considerada relevante é executada.

Exemplo:

Considere o trecho do algoritmo abaixo, para encontrar o maior elemento de um vetor de inteiros $A[n]$, onde $n \geq 1$.

```
int n = A.length;
int temp = A[0];

for (int i = 1; i < n; i++) {
    if (A[i] > temp) {
        temp = A[i];
    }
}

int max = temp;
```

Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A , se A contiver n elementos. Logo:

$$f(n) = n - 1, \text{ para } n > 1$$

Provando que o algoritmo é ótimo:

Teorema: Qualquer algoritmo, para encontrar o maior elemento de um conjunto com n elementos, $n > 0$, faz pelo menos $n-1$ comparações.

Prova: Cada um dos $n-1$ elementos tem de ser verificado, por meio de comparações, que é menor do que algum outro elemento. Logo, $n-1$ comparações são necessárias. O custo é igual sobre todos os problemas de tamanho n .

Veja a figura a seguir, que ilustra esse algoritmo:

0	1	2	3	4	5	6	7	8	9
20	10	30	44	50	1	23	4	88	9

- Inicialmente, guardamos na variável TEMP, o valor que está armazenado na posição zero do vetor → TEMP = 20.
- Iniciamos o laço for, com $i=1$ e as comparações começam, em busca de algum valor que seja maior do que o que está atualmente em TEMP (20). A comparação $A[i] > temp$ é feita diversas vezes, até encerrar o laço for:
 - $10 > 20$ → falso → o maior valor ainda é 20
 - $30 > 20$ → verdadeiro → temp = 30
 - $44 > 30$ → verdadeiro → temp = 44
 - $50 > 44$ → verdadeiro → temp = 50
 - $1 > 50$ → falso → o maior valor ainda é 50
 - $23 > 50$ → falso → o maior valor ainda é 50
 - $4 > 50$ → falso → o maior valor ainda é 50
 - $88 > 50$ → verdadeiro → temp = 88
 - $9 > 88$ → falso → o maior valor ainda é 88
- Fim do laço e MAX recebe o valor 88 (o maior valor encontrado).
- Repare que, para um vetor com 10 elementos ($n=10$), tivemos de fazer nove comparações ($n-1$); e se o vetor tiver 20 elementos, quantas comparações serão necessárias? E se tiver 40 elementos?
- A função $f(n)=n-1$ nos dá a resposta (o custo) para qualquer valor de n :

n	Operações
10	9
20	19
40	39
n	n - 1

Acabamos de fazer nosso primeiro modelo matemático de representação do custo de um algoritmo. Ou seja, a partir dessa fórmula, conseguimos saber quantas operações serão realizadas, para qualquer valor de n . Inclusive, ao analisar a tabela anterior, percebemos que, se n dobra, o número de operações é quase igual ao dobro também.

Cenários

- **Melhor caso:** corresponde ao menor tempo de execução sobre todas as possíveis entradas de tamanho n . Desta forma, os dados de entrada favorecem a execução do algoritmo, implicando no menor consumo de tempo para execução.
- **Pior caso:** corresponde ao maior tempo de execução sobre todas as entradas de tamanho n . O custo de um algoritmo no pior caso é o limite superior da função $f(n)$. Desta forma, os dados de entrada desfavorecem a execução do algoritmo, implicando no maior consumo de tempo para execução.
- **Caso médio (caso esperado):** corresponde à média dos tempos de execução de todas as entradas de tamanho n . Neste caso, leva-se em consideração a distribuição de probabilidades, em que todas as entradas são igualmente possíveis.

Exemplo:

Considere o algoritmo para encontrar determinado valor dentro de um vetor de n elementos inteiros ($n \geq 1$). Utilizaremos o algoritmo da busca sequencial. Caso o elemento seja encontrado, é retornada a posição do vetor onde o elemento foi encontrado; caso o valor procurado não exista no vetor, é retornado o valor -1.

```
public static int buscaSeq(int[] vetor, int valor) {  
    for (int i = 0; i < vetor.length; i++) {  
        if (vetor[i] == valor) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Melhor caso: o valor procurado é o primeiro a ser consultado. Nesse caso, é feita uma única comparação → $f(n)=1$.

Pior caso: o valor procurado é o último a ser consultado, ou não está presente no vetor. Nesse caso, são feitas n comparações → $f(n)=n$.

Caso médio: aqui, temos o caso esperado, ou seja, contamos com a questão da probabilidade, em que o valor procurado, não é o primeiro nem é o último. Para estimar esse tempo, usamos a média entre os tempos anteriores:

$$f(n) = \frac{n+1}{2}, \text{ para } n > 0$$

Provando que o algoritmo é ótimo:

Teorema: Qualquer algoritmo, para encontrar um valor de um conjunto com n elementos por meio da busca sequencial, $n > 0$, faz pelo menos $(n+1)/2$ comparações.

Prova: Cada um dos elementos tem de ser verificado por meio de comparações e, se o valor for encontrado, é retornada a posição em que ele está armazenado, ou -1, se não for encontrado. O custo é igual sobre todos os problemas de tamanho n .

Veja a tabela abaixo, que ilustra esse algoritmo:

20	10	30	44	50	1	23	4	88	9
----	----	----	----	----	---	----	---	----	---

- Para ilustrar o MELHOR CASO, buscaremos o valor 20. É feita uma única comparação: o elemento que está na posição 0 do vetor é igual ao valor procurado? \rightarrow verdadeiro \rightarrow retorna a posição em que foi encontrado o valor $\rightarrow 0 \rightarrow f(n) = 1$.
- Para ilustrar o PIOR CASO, procuraremos o valor 9. Iniciamos o laço for, com $i=0$ e as comparações começam. A comparação $\text{vetor}[i] == \text{valor}$ é feita diversas vezes, até encerrar o laço for:
 - $20 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $10 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $30 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $44 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $50 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $1 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $23 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $4 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $88 == 9 \rightarrow$ falso \rightarrow incrementa i
 - $9 == 9 \rightarrow$ verdadeiro \rightarrow retorna a posição em que foi encontrado o valor $\rightarrow 9$
- Fim do laço e da função. Repare que para um vetor com 10 elementos ($n=10$), tivemos de fazer 10 comparações (n) $\rightarrow f(n) = n$.

- Para o CASO MÉDIO, podemos afirmar que o algoritmo realizará, em média, $(n+1)/2$ operações. Ou seja, para $n = 10$, em média, serão executadas cinco comparações (o valor foi arredondado para baixo). Utilizamos sempre o Caso Médio como referência, pois tem maior probabilidade de acontecer).

Análise assintótica

Ao ver uma expressão com $f(n)=n+10$ ou $f(n)=n^2+1$, a maioria das pessoas pensa automaticamente em valores pequenos de n . A análise de algoritmos faz exatamente o contrário: ignora valores pequenos e concentra-se nos valores enormes de n .

Para valores enormes de n , as funções n^2 , $(3/2)n^2$, $999 n^2$, $n^2/1000$, $n^2 + 100n$ crescem com a mesma velocidade e, portanto, são todas equivalentes. Esse tipo de matemática, interessada em valores enormes de n , é chamada de assintótica. Essas funções pertencem à mesma ordem e, portanto, são equivalentes.

A escolha do algoritmo não é um fator crítico para problemas de tamanho pequeno. O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce. O crescimento assintótico de $f(n)$ representa a velocidade com que a função tende ao infinito. Usualmente, é utilizada a notação O (Big O ou O grande) para denotar essa função $\rightarrow f(n) = O(n^2)$.

A análise assintótica permite comparar funções complexidades $f(n)$ de diferentes algoritmos, agrupando-as em classes de equivalência. Logo, dado um problema P que possua uma coleção de algoritmos A que o resolvem, é possível escolher aqueles que sejam mais eficientes com base na determinação de suas funções complexidade. Ou seja, a análise assintótica é a simplificação matemática que viabiliza a análise de algoritmos.

Principais classes de comportamento

$f(n) = O(1)$	Complexidade constante , ou seja, o custo do algoritmo independe do tamanho de n . As instruções do algoritmo são executadas um número fixo de vezes.
$f(n) = O(n)$	Complexidade linear , ou seja, se o valor de n dobrar, o número de operações também dobrará.
$f(n) = O(n^2)$	Complexidade quadrática , ou seja, se n dobrar, o tempo quadruplicará.
$f(n) = O(\log n)$	Complexidade logarítmica , ou seja, se n dobrar, o número de operações aumentará de uma constante.

$f(n) = O(n \log n)$	Complexidade logarítmica linear , ou seja, se n dobrar, o número de operações ultrapassa o dobro do tempo de n .
$f(n) = O(2^n)$	Complexidade exponencial , ou seja, se n dobrar, o número de operações é elevado ao quadrado.
$f(n) = O(n^3)$	Complexidade cúbica , ou seja, se n dobrar o tempo de execução será multiplicado por 8.

Os dois exemplos que estudamos anteriormente (buscar o maior elemento de um vetor e fazer a busca sequencial em vetor) pertencem à ordem Linear, uma vez que são $O(n)$.

Para se ter uma ideia, considere um algoritmo que manipule um vetor com $n=10^5$ ou 100.000 elementos. Considerando que 1 operação básica leva 1 microssegundo, e que 1 microssegundo = 10^{-6} segundos, veja a diferença entre as classes de algoritmos:

Ordem de complexidade	Tempo consumido	
$O(\log n)$	0,00002	segundo
$O(n)$	0,1	segundo
$O(n \log n)$	2	segundos
$O(n^2)$	3	horas
$O(n^3)$	32	anos
$O(2^n)$		séculos

Ao longo de nosso curso, estudaremos diversos TADS e faremos a análise de suas operações em termos de complexidade.

