
# DESCRIÇÃO

Conceitos de análise de algoritmos, os tipos de estruturas de dados homogêneas, heterogêneas e ponteiros, análise da complexidade dos algoritmos, notação O, como avaliar a complexidade dos algoritmos.

# PROPÓSITO

Apresentar os conceitos básicos para o entendimento da construção de algoritmos, empregando os tipos de estrutura de dados básicos. Compreender a importância da análise de algoritmos, permitindo a construção de programas com desempenho adequado à necessidade dos usuários. Empregar a notação O e utilizar exemplos práticos para entender a análise de algoritmo.

# PREPARAÇÃO

Antes de iniciar o conteúdo deste tema, tenha em mãos um livro de Matemática do ensino médio que apresente os conceitos de funções matemáticas, como função lineares, funções quadráticas, exponenciais e logarítmicas.

# OBJETIVOS

## MÓDULO 1

Definir os conceitos básicos para construção de algoritmos

## MÓDULO 2

Definir as estruturas de dados manipuladas pelos algoritmos

## MÓDULO 3

Definir a notação O e suas aplicações práticas

## MÓDULO 4

Empregar a análise da complexidade dos algoritmos

# INTRODUÇÃO

Algoritmos são a estrutura básica para a criação de soluções para problemas computacionais. A modularização de algoritmos é a principal forma para diminuir a complexidade desses problemas.

Os módulos ou subprogramas passam a tratar partes menores da complexidade do problema, facilitando a compreensão e futuras manutenções que serão necessárias durante o ciclo de vida de um software. A modularização também diminui o retrabalho, pois permite que trechos de códigos sejam reutilizados em outros locais no mesmo sistema.

Após a definição de um algoritmo, é necessário passar por uma otimização. Por mais que, atualmente, os computadores possuam uma capacidade computacional bastante poderosa, principalmente comparando com os recursos computacionais de décadas atrás, a otimização permite que o sistema possa ter um desempenho muito mais adequado, oferecendo ao usuário uma melhor experiência de uso.

A complexidade dos problemas cresceu tão rápido quanto a capacidade computacional, portanto é fundamental a capacidade de analisar diversos algoritmos para um problema complexo para se chegar

ao algoritmo mais otimizado para o problema. Um algoritmo otimizado deverá executar em um menor espaço de tempo possível ocupando o menor espaço possível de memória.

Apresentaremos na introdução do Módulo 1 a construção de algoritmos com suas respectivas estruturas de dados utilizadas, os conceitos de análise de algoritmos, explicando como analisar o pior caso de um algoritmo e se chegar a uma ordenação dos algoritmos para a solução de um determinado problema.

# MÓDULO 1

---

- Definir os conceitos básicos para construção de algoritmos

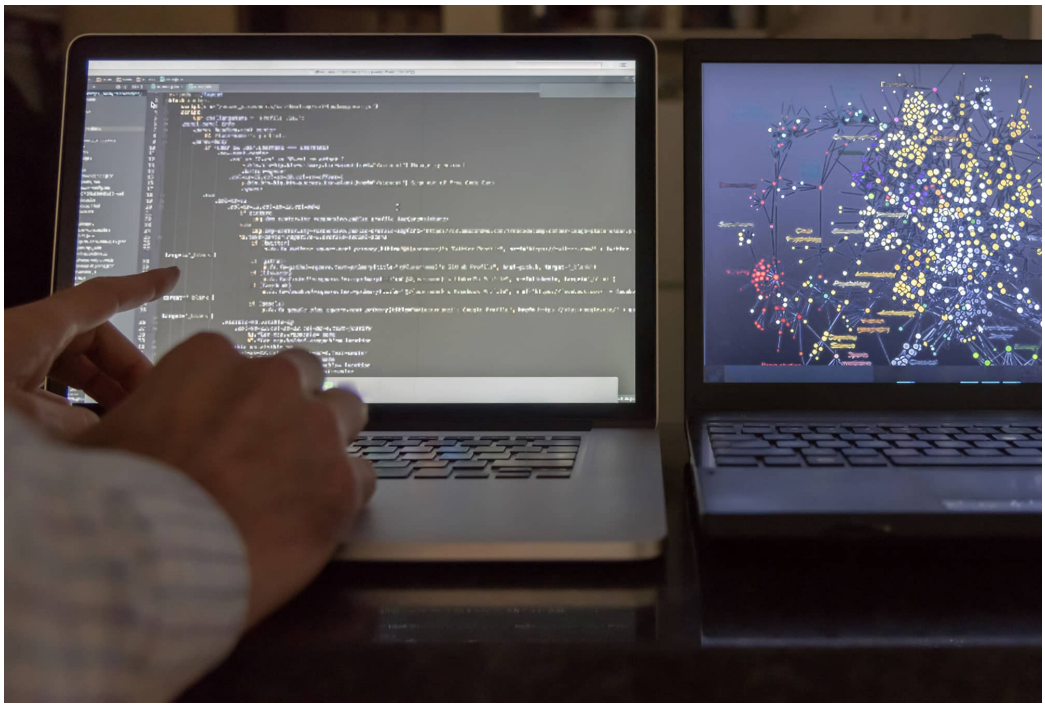
# INTRODUÇÃO

Segundo Forbellone e Eberspacher (2005), um problema matemático é tão mais complexo quanto maior for a quantidade de variáveis a serem tratadas, enquanto um problema algorítmico é tão mais complexo quanto maior for a quantidade de situações diferentes que precisam ser tratadas.

## SAIBA MAIS

Um algoritmo é base de tudo que é programado para um computador e pode ser definido como uma sequência finita de etapas, perfeitamente definidas, que é utilizada para solucionar um problema.

Cada algoritmo tem uma complexidade que está intimamente associada à complexidade do problema a ser resolvida. Quanto maior for a variedade de situações a serem tratadas, maior será a complexidade.



Fonte: Shutterstock

A complexidade pode ser reduzida, reduzindo-se a variedade. E a variedade pode ser reduzida, dividindo problemas maiores em problemas menores. Os problemas menores são tratados através do emprego de sub-rotinas, que terão complexidade menor e poderão ser implementadas de uma forma mais fácil.

Uma técnica que será estudada para a decomposição de problemas é conhecida como top-down, a qual será estudada mais adiante, e que irá definir as sub-rotinas que devem ser criadas para a resolução dos problemas.

## SUB-ROTINAS

Sub-rotinas, também chamadas de subprogramas, são blocos de instruções que realizam tarefas específicas. É um trecho de programa com atribuições específicas, simplificando o entendimento do programa principal, proporcionando ao programa menores chances de erro e de complexidade. Elas são utilizadas para diminuir a complexidade de problemas complexos.

Os programas, de acordo com Ascêncio (2012), tendem a ficar menores e mais organizados, uma vez que o problema pode ser subdivido em pequenas tarefas. Em resumo, são pequenos programas para resolver um problema bem específico. Uma sub-rotina não deve ser escrita para resolver muitos problemas, senão ela acabará perdendo o seu propósito.

## OBJETIVOS

Uma sub-rotina possui os seguintes objetivos:

Dividir e estruturar um algoritmo em partes logicamente coerentes.

Facilidade em testar os trechos em separado.

Aumentar a legibilidade de um programa.

Evitar que uma certa sequência de comandos necessária em vários locais de um programa tenha que ser escrita repetidamente nestes locais, diminuindo também, o código fonte.

O programador poderá criar sua própria biblioteca de funções, tornando sua programação mais eficiente, uma vez que poderá fazer uso de funções por ele escritas em vários outros programas com a vantagem de já terem sido testadas.

## VANTAGENS

Uma sub-rotina possui as seguintes vantagens:

Clareza e legibilidade no algoritmo.

Construção independente.

Testes individualizados.

Simplificação da manutenção.

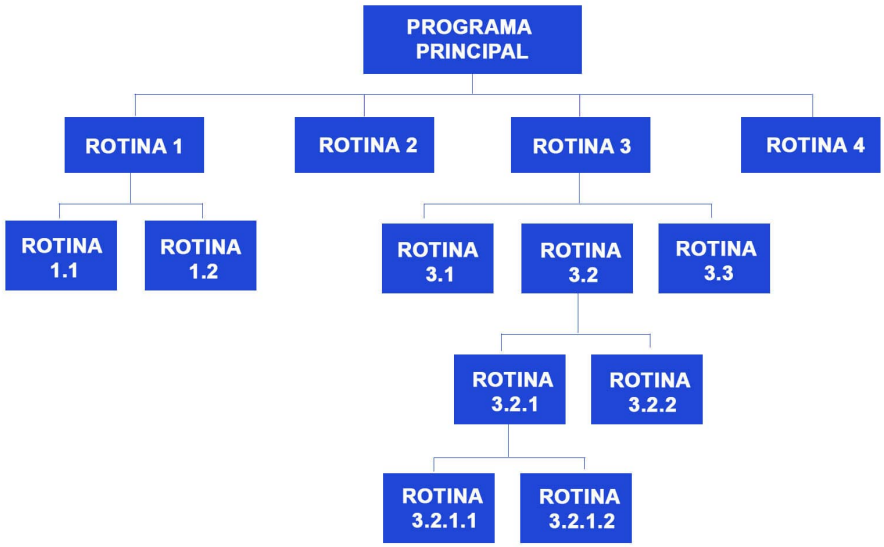
Reaproveitamento de algoritmos.

# DECOMPOSIÇÃO DE PROBLEMAS

Um método adequado para a programação estruturada de um computador é trabalhar com o conceito de programação estruturada, pois a maior parte das linguagens de programação utilizadas atualmente também são adequadas a este tipo de paradigma, o que facilita a aplicação deste processo de trabalho. O método mais adequado para a programação estruturada, segundo Manzano e Oliveira (2016).

Este método permite que o programa tenha uma estrutura semelhante a um organograma.

A figura mostra um exemplo da estrutura top-down.




Fonte: O Autor

A utilização do método top-down permite que seja realizada uma divisão de problemas em refinamentos sucessivos, da seguinte forma:


Fonte: O Autor

Cada divisão obtida com a técnica de refinamentos sucessivos pode ser convertida uma parte do algoritmo. Essas partes são conhecidas como módulos ou sub-algoritmos.

# DECLARAÇÃO

Uma **sub-rotina** é um bloco contendo início e fim, sendo identificada por um nome, pelo qual será referenciada em qualquer parte e em qualquer momento do programa.

# SUB-ROTINA

A sub-rotina serve para executar tarefas menores, como ler, calcular, determinar o maior/menor valor entre uma lista de valores, ordenar, converter para maiúsculas, entre outras.

Como uma sub-rotina é um programa, ela poderá efetuar diversas operações computacionais, como entrada, processamento e saída, da mesma forma que são executadas em um programa.

A sintaxe genérica de uma sub-rotina é a seguinte:

sub-rotina < nome_da_sub-rotina > [(tipo dos parâmetros : < sequência de declarações de parâmetros >)] : < tipo de retorno >

var
< declaração de variáveis locais >;
**Início**
comandos que formam o corpo da sub-rotina
retorne(< valor >) ; /* ou retorne; ou nada */
**Fim sub-rotina**

Onde:

| Glossário | | |
|---|---|---|
| tipo de retorno | Sequência de declarações de parâmetros | corpo da sub-rotina |
| nome da sub-rotina | **início** | retorne ( .. ) |
| parâmetros | variáveis locais | **Fim sub-rotina** |

**Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

## TIPO DE RETORNO

Tipo de dado que a sub-rotina dará retorno.

## SEQUÊNCIA DE DECLARAÇÕES DE PARÂMETROS

Declarações de variáveis da sub-rotina (tipo e nome).

## CORPO DA SUB-ROTINA

Sequência de comandos.

## NOME DA SUB-ROTINA

Segue as mesmas regras de declaração de variáveis da linguagem utilizada.

## INÍCIO

Início da sub-rotina.

## RETORNE ( .. )

O que será retornado ou não para o algoritmo.

## PARÂMETROS

Nomes das variáveis, seguem as mesmas regras de declaração de variáveis da linguagem utilizada.

## VARIÁVEIS LOCAIS

Declarações de variáveis que serão utilizadas dentro da sub-rotina (tipo e nome).

## FIM SUB-ROTINA

Fim da sub-rotina.

A declaração da sub-rotina deve estar entre o final da declaração de variáveis e a linha de início do programa principal.

Segue, abaixo, um exemplo de um algoritmo para somar dois números inteiros.

```
Algoritmo somaInteiros;
var
n, m: inteiro;
//declaração de sub-rotina
sub-rotina soma(inteiro: a,b): inteiro
var
```

```
resultado: inteiro;
início
resultado <- a + b;
retorne resultado;
fim sub-rotina
início
n <- 8;
m <- 4;
escreva(soma(n, m)); //chamada da sub-rotina (soma)
fim Algoritmo.
```

# CHAMADA DE SUB-ROTINA

Uma sub-rotina pode ser acionada de qualquer ponto do algoritmo principal ou de outra sub-rotina. O acionamento de uma sub-rotina é conhecido por **chamada** ou **ativação**.

Quando ocorre uma chamada, o fluxo de controle é desviado para a sub-rotina, quando ela é ativada no algoritmo principal. Ao terminar a execução dos comandos da sub-rotina, o fluxo de controle retorna ao comando seguinte àquele onde ela foi ativada, exatamente conforme o seguinte algoritmo:

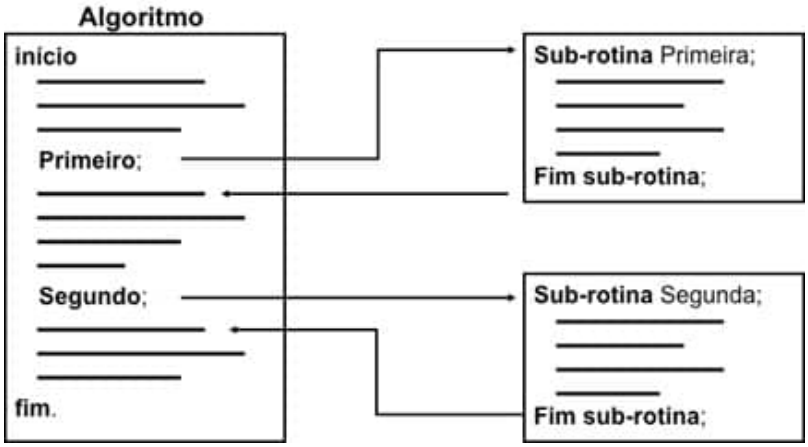
```
algoritmo()
{...
...
< chamada da sub-rotina >
...
...
}
sub-rotina(...)
{ ...
retorne(...);
}
```

Pode ocorrer que um algoritmo tenha mais de uma sub-rotina. Desta forma, teremos várias chamadas a partir do programa principal a cada uma das sub-rotinas.

## ★ EXEMPLO

O exemplo abaixo mostra um algoritmo com duas sub-rotinas sendo chamadas, veja que o fluxo de execução passa para a sub-rotina chamada (primeira sub-rotina). Ao final da execução da primeira sub-rotina, o controle retorna para o programa principal.

O mesmo procedimento ocorre com a chamada para a segunda sub-rotina, ao final da execução o controle retorna para o programa principal que segue o seu fluxo normal de execução, conforme a figura:




Fonte: Autor

# PARAMETRIZAÇÃO DE SUB-ROTINAS

É possível tornar uma sub-rotina mais genérica e, portanto, mais reutilizável. Isso pode ser feito através da utilização de parâmetros. Parâmetros são os valores que uma sub-rotina pode receber antes de iniciar a sua execução.

## 💬 COMENTÁRIO

A ativação do módulo precisa indicar os argumentos (valores constantes ou variáveis que serão passados para a sub-rotina). E a passagem dos valores ocorre a partir da correspondência (ordem) argumento X parâmetro.

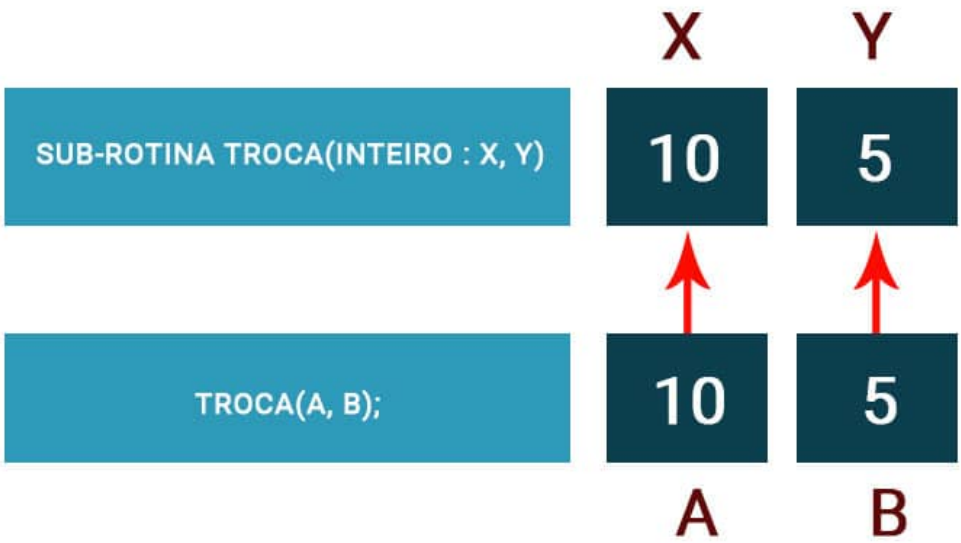
Podemos ter, por exemplo, uma sub-rotina para exibir a tabuada de oito. Essa sub-rotina poderia ser mais genérica caso fosse possível exibir a tabuada de qualquer número utilizando um parâmetro. Desta forma, a sub-rotina poderia ser executada para calcular a tabuada de qualquer valor passado como parâmetro.

O exemplo abaixo utiliza o conceito de parâmetros para trocar o valor de dois números recebidos como parâmetros.

```
1 Algoritmo trocaValores;
2 var
3 a, b : inteiro;
4 //declaração de sub-rotina
5 sub-rotina troca(inteiro : x, y)
6 var
7 auxiliar: inteiro;
8 início
9 auxiliar <- x;
10 x <- y;
11 y <- auxiliar;
12 escreva(´valor de a: ´, x, ´valor de b:´, y);
13 fim sub-rotina
14 inicio
15 a <- 10;
16 b <- 5;
17 troca (a, b); //chamada da sub-rotina
18 fim Algoritmo.
```

No algoritmo acima, a definição da sub-rotina é feita das linhas 05 a 13 e a chamada de função é realizada na linha 17.

Os valores das variáveis **a** e **b** são passados na sequência para as variáveis **x** e **y** na sub-rotina troca (inteiro : x, y), ou seja, o valor da variável **a** é passado por valor para a variável **x**, enquanto que o valor da variável **b** também é passado por valor para a variável **y**.




Fonte: O Autor

Na passagem de parâmetros por valor, o valor da variável na chamada é copiado para a variável da função. As alterações realizadas dentro da função não são refletidas na variável original.

Portanto, a substituição de valores na sub-rotina troca (inteiro: x, y) ocorrerá apenas dentro da sub-rotina, não alterando os valores das variáveis a e b.

Outra forma de passar valores por parâmetros é a passagem por referência, onde os endereços de memória das variáveis são acessados diretamente e alterações nos valores são refletidas nas variáveis originais.

Se a passagem de parâmetros da sub-rotina troca (inteiro: x, y) fosse por referência, os valores das variáveis teriam sido comutados entre si.

Existem sub-rotinas que podem receber parâmetros e retornar um valor para o programa principal ou ainda sub-rotinas que recebem parâmetros e não retornam nenhum valor para o programa principal.

As primeiras são chamadas de funções enquanto as últimas, além de receberem a denominação de funções, são também chamadas de procedimentos em algumas linguagens de programação.

Seguem dois exemplos de sub-rotinas, uma função com retorno e um procedimento. Os dois exemplos utilizam uma estrutura de repetição para somar os valores de 1 a 10, mostrando no final o valor da soma.

Vamos ver o primeiro exemplo com retorno.

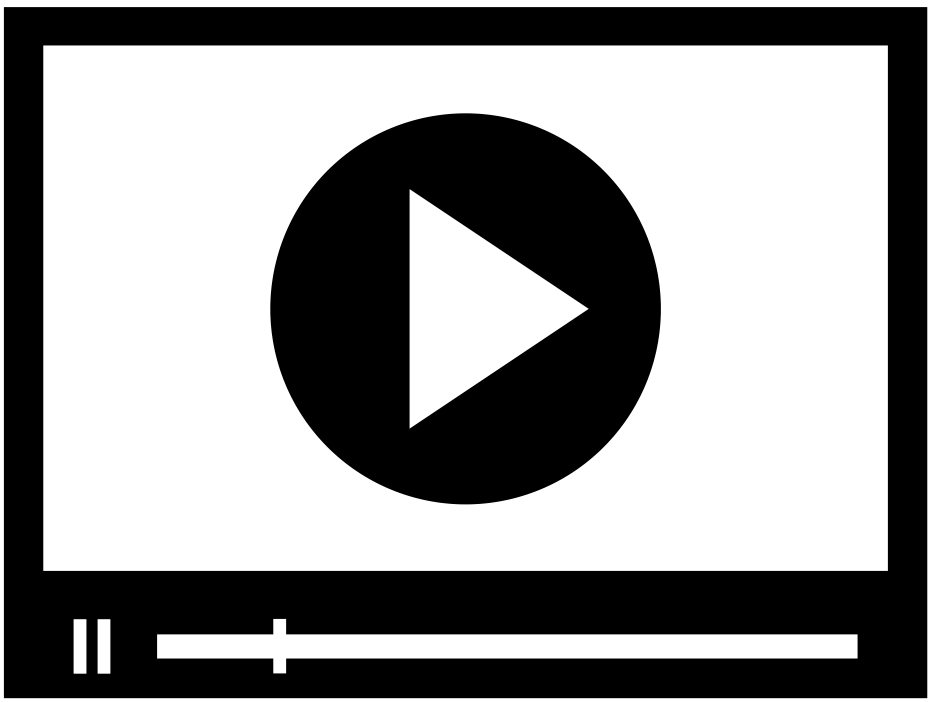
```
Algoritmo somaValoresComRetorno;
var
resultado : inteiro;
// declaração de sub-rotina
// utilizei o nome procedimento, poderia ser utilizado sub-rotina
procedimento somaValores( )
var
soma : inteiro;
inicio
soma <- 0;
para i de 1 até 10 faça
soma <- soma + i;
fim-para
retorne soma;
fim procedimento;
inicio
resultado <- somaValores( );
escreva(“ A soma dos valores é “, soma);
fim Algoritmo.
```

Agora, vamos ver o segundo exemplo sem retorno.

```
Algoritmo somaValoresSemRetorno;

// declaração de sub-rotina
// utilizei o nome função, também poderia ser utilizado sub-rotina
função somaValores( )
var
soma : inteiro;
inicio
soma <- 0;
para i de 1 até 10 faça
soma <- soma + i;
fim-para
escreva(“ A soma dos valores é “, soma);
fim procedimento;

inicio
somaValores( );
fim Algoritmo.
```



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

# VERIFICANDO O APRENDIZADO

**1. FUNÇÕES SÃO USADAS PARA CRIAR PEQUENOS PEDAÇOS DE CÓDIGOS SEPARADOS DO PROGRAMA PRINCIPAL. ELAS SÃO IMPORTANTES PORQUE ELAS RETORNAM VALORES, AJUDAM A FRAGMENTAR O CÓDIGO EM PARTES MENORES - MAIS FÁCEIS DE LIDAR - E AINDA POR CIMA PODEM SER UTILIZADAS MAIS DE UMA VEZ NO MESMO PROGRAMA, POUPANDO PRECIOSOS MINUTOS DE PROGRAMAÇÃO E INÚMERAS LINHAS DE CÓDIGO. NA DEFINIÇÃO DE UMA FUNÇÃO, PRECISAMOS ESCREVER SUAS PARTES, SENDO ALGUMAS OBRIGATÓRIAS E OUTRAS, FACULTATIVAS.**

**SUB-ROTINA < NOME_DA_SUB-ROTINA > [( TIPO DOS PARÂMETROS : <**

**SEQUÊNCIA DE DECLARAÇÕES DE PARÂMETROS > )] : < TIPO DE RETORNO >**

**VAR**

**< DECLARAÇÃO DE VARIÁVEIS LOCAIS >;**

**INÍCIO**

**COMANDOS QUE FORMAM O CORPO DA SUB-ROTINA**

**RETORNE(< VALOR >) ; /* OU RETORNE; OU NADA */**

**FIM SUB-ROTINA**

**DAS OPÇÕES APRESENTADAS ABAIXO, MARQUE A QUE APRESENTA APENAS PARTES OBRIGATÓRIAS DE FORMA QUE A FUNÇÃO EXECUTE ALGUMA TAREFA, SEM LEVAR EM CONSIDERAÇÃO OS SÍMBOLOS (CHAVES, PARÊNTESES, PARÂMETROS, DENTRE OUTROS).**

**A)** Tipo de retorno, nome e corpo da função

**B)** Tipo de retorno, parâmetro e corpo da função

**C)** Tipo de retorno, nome e parâmetros

**D)** Tipo de retorno, nome, parâmetros e corpo da função

## 2. A MODULARIZAÇÃO DE ALGORITMOS É IMPORTANTE PARA ORGANIZAR MELHOR O CÓDIGO, FACILITAR A MANUTENÇÃO, ENTRE OUTRAS COISAS. SOBRE FUNÇÕES E PROCEDIMENTOS, ASSINALE A ALTERNATIVA CORRETA SOBRE A MODULARIZAÇÃO:

**A)** O procedimento sempre retorna um valor ao programa.

**B)** A função retorna um valor ao programa.

**C)** As variáveis definidas no escopo de cada função são acessíveis em todo o programa.

**D)** As variáveis locais são declaradas no escopo do programa inteiro.

---

## GABARITO

**1. Funções são usadas para criar pequenos pedaços de códigos separados do programa principal. Elas são importantes porque elas retornam valores, ajudam a fragmentar o código em partes menores - mais fáceis de lidar - e ainda por cima podem ser utilizadas mais de uma vez no mesmo programa, poupando preciosos minutos de programação e inúmeras linhas de código. Na definição de uma função, precisamos escrever suas partes, sendo algumas obrigatórias e outras, facultativas.**

```
sub-rotina < nome_da_sub-rotina > [( tipo dos parâmetros : < sequência de declarações de parâmetros > )] : < tipo de retorno >
var
   < declaração de variáveis locais >;
início
     comandos que formam o corpo da sub-rotina
 retorne(< valor >) ; /* ou retorne; ou nada */
Fim sub-rotina
```

**Das opções apresentadas abaixo, marque a que apresenta apenas partes obrigatórias de forma que a função execute alguma tarefa, sem levar em consideração os símbolos (chaves, parênteses, parâmetros, dentre outros).**

A alternativa **"A "** está correta.

Os parâmetros em funções não são obrigatórios. Em uma função são obrigatórios: o tipo de retorno, nome e corpo da função.

**2. A modularização de algoritmos é importante para organizar melhor o código, facilitar a manutenção, entre outras coisas. Sobre funções e procedimentos, assinale a alternativa correta sobre a modularização:**

A alternativa **"B "** está correta.

As funções sempre retornam um valor ao programa principal. Os procedimentos não retornam um valor ao programa principal.

# MÓDULO 2

---

⦿ Definir as estruturas de dados manipuladas pelos algoritmos

# INTRODUÇÃO

Os tipos primitivos (inteiro, real, caractere e lógico) não são suficientes para representar todos os tipos de informação, particularmente quando temos mais de uma informação relacionada, por exemplo, uma lista dos nomes dos alunos de uma sala, endereço de alguém, dentre outros. Os tipos primitivos serão utilizados para construir outras estruturas de dados mais complexas, como vetores, matrizes e registros.

Fonte: Shutterstock

A partir de agora, apresentaremos uma técnica de programação que permitirá trabalhar com o agrupamento de várias informações dentro de uma mesma variável. Vale salientar que este agrupamento ocorrerá obedecendo sempre ao mesmo tipo de dado, a menos que se trabalhe com uma estrutura de dados do tipo registro.

# ARRANJO UNIDIMENSIONAL HOMOGÊNEO

O tipo de dado homogêneo recebe diversos outros nomes, tais como: variáveis indexadas, variáveis compostas, variáveis subscritas, arranjos, **vetores**, tabelas em memória ou arrays.

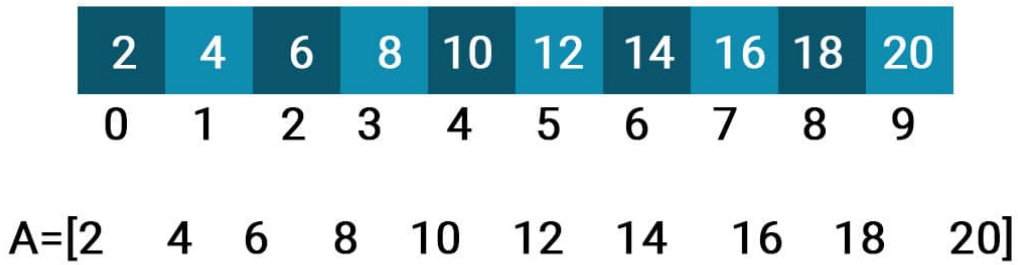
## VETORES

Os vetores são tipos de dados que são construídos à medida que forem necessários, pois não é sempre que os tipos básicos (real, inteiro, caractere ou lógico) e/ou variáveis simples são suficientes para representar a estrutura de dados utilizada em um programa.

Um vetor é uma variável homogênea que repete um tipo de dado um número específico de vezes. Ou seja, é um arranjo de elementos armazenados na memória principal, um após o outro, todos com o

mesmo nome. A ideia é a mesma de uma matriz linha da matemática, isto é, várias colunas e uma linha. Este tipo de estrutura, em particular, é também denominado como arranjo unidimensional.

Um exemplo de vetor de valores inteiros está representado abaixo.




Fonte: O Autor

Esse é um vetor de 10 elementos, isto é, tem 10 variáveis, todas com o mesmo nome e diferentes por sua posição dentro do arranjo que é indicada por um índice.

Quando se tem somente uma linha, podemos omiti-la e colocar somente a coluna. Em algoritmos, expressamos da seguinte forma:

Fonte: O Autor

# DECLARAÇÃO

A declaração de um vetor caracteriza-se por ser definida uma única variável dimensionada com um determinado tamanho. A dimensão de um vetor é constituída por constantes inteiras e positivas. Os nomes dados aos vetores seguem as mesmas regras de nomes utilizados em variáveis simples.

tipo < nome_do_vetor > [tamanho] ;

Uma variável simples somente pode conter um valor por vez. No caso dos vetores, estes poderão armazenar mais de um valor por vez, pois são dimensionados exatamente para este fim. Lembrando que a manipulação dos elementos de um vetor ocorrerá de forma individualizada, pois não é possível efetuar a manipulação de todos os elementos do conjunto ao mesmo tempo.

Tanto a entrada como a saída de dados manipuladas em um vetor são processadas passo a passo, um elemento por vez. Estes processos são executados com o auxílio de estruturas de repetição, um looping.

Com o objetivo de apresentar o benefício do emprego de vetores, vamos imaginar que é necessário criar um algoritmo para calcular a média geral de uma turma de dez alunos. Na implementação abaixo, vamos utilizar um algoritmo sem empregar um vetor.

```
Algoritmo media_1;
var
n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, media: real;
Início
escreva(“Entre com a nota do 1º aluno: “);
ler(n1);
escreva(“Entre com a nota do 2º aluno: “);
ler(n2);
escreva(“Entre com a nota do 3º aluno: “);
ler(n3);
escreva(“Entre com a nota do 4º aluno: “);
ler(n4);
escreva(“Entre com a nota do 5º aluno: “);
ler(n5);
escreva(“Entre com a nota do 6º aluno: “);
ler(n6);
escreva(“Entre com a nota do 7º aluno: “);
ler(n7);
escreva(“Entre com a nota do 8º aluno: “);
ler(n8);
escreva(“Entre com a nota do 9º aluno: “);
ler(n9);
escreva(“Entre com a nota do 10º aluno: “);
ler(n10);
media ← (n1 + n2 + n3 + n4 + n5 + n6 + n7 + n8 + n9 + n10)/10;
escreva(“A média da turma é: ”, media);
Fim.
```

## COMENTÁRIO

Percebe-se que, neste algoritmo, para calcular a média da turma, foi preciso declarar dez variáveis contendo a média de cada aluno.

Com tanta variável, fica difícil a manipulação delas. E se no caso, a média fosse de uma turma de 100 alunos? Vamos reescrever esse algoritmo usando a estrutura de um vetor.

**Algoritmo** media_2;
**var**
notas[10], soma, media : **real;**
i: **inteiro;**
**Início**
soma ← 0;
**Para** i ← 0 **até** 9 **faça**
**início**
**escreva**(“Digite a”’, i+1 , “ª. nota:”);
**leia**(notas[I]);
soma ← soma + notas[i];
**fim;**
media ← soma / 10;
**escreva**(“A média dos alunos é :”, media);
**Fim.**

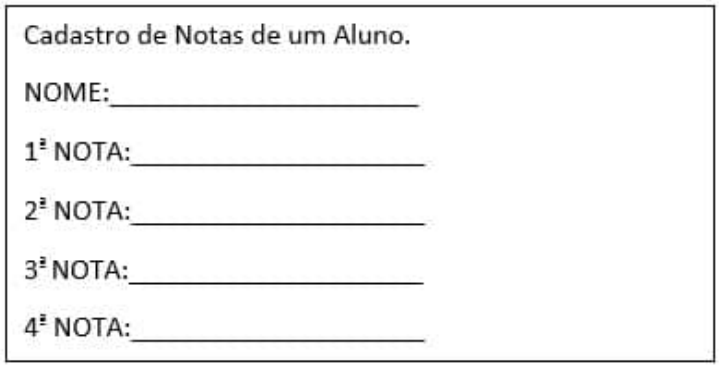
Observe que agora foi utilizada uma única variável com 10 valores diferentes armazenados e identificados com a ajuda de um índice, tornando o programa menor e mais fácil o seu entendimento.

# REGISTROS

A utilização da estrutura de dados registro consiste em trabalhar vários dados de tipos diferentes (campos), em uma mesma estrutura. Por esta razão, este tipo de dado é considerado heterogêneo. Essa múltipla alocação de variáveis é chamada de dados compostos heterogêneos, estruturas ou registros, na qual os elementos de um registro são alocados em posições de memória adjacentes.

Para poder informar o nome de um aluno e de suas quatro notas, isto nos obrigaria a utilizar dois vetores: um para armazenar os nomes dos alunos e o outro para conter as notas (os valores armazenados em ambos são de tipos diferentes).

A utilização dos registros vem facilitar essa representação, conforme a Figura:




Fonte: Autor

📷 Figura – Estrutura de um registro. Fonte: O Autor.

Essa estrutura do tipo registro é formada pelos campos: nome, 1ª nota, 2ª nota, 3ª nota, 4ª nota, e poderia ser chamado de Aluno. Para selecionar e guardar os dados de um determinado aluno (na memória), seriam necessárias tantas variáveis quantos campos existam no formulário de cadastro. E o tempo para manipular tais dados será muito maior se não se dispuser de uma forma simplificada de serem acessados e manipulados.

# DEFINIÇÃO

Os registros são conjuntos de informações relacionadas entre si, que podem ser referenciadas como uma unidade (o Registro) e, normalmente, são conjuntos de informações (campos) de tipos diferentes.

Assim sendo, um registro é uma estrutura de dados composta e heterogênea, ou seja, formada por um conjunto de variáveis (campos) que podem assumir tipos diferentes de dados, inclusive os tipos compostos (vetores, matrizes e registros) .

Ao contrário das variáveis já estudadas, uma variável do tipo registro, quando definida, pode ocupar muitas posições da memória principal, que ficam reservadas para o uso.

## 📢 ATENÇÃO

Logo, ao se definir uma variável do tipo registro, deve-se definir também quais serão as partes componentes desta variável (os campos), por exemplo: nome, idade, sexo, altura, junto ao seu tipo básico.

Um registro poderá ter tantos campos quanto o Analista deseje. Para facilitar, recomenda-se um máximo de 10 campos por registro. Quando forem necessárias estruturas de dados maiores, pode-se utilizar

registros previamente definidos dentro de outros registros.

# DECLARAÇÃO

Um registro deve ser declarado da seguinte forma:

tipo r = registro
{Descrição dos campos e seus tipos}
fim registro;
r = REG;

Tomando como exemplo a proposta de se criar um registro denominado Aluno, será declarado da seguinte forma:

tipo r = registro
texto: NOME;
real: NOTA1;
real: NOTA2;
real: NOTA3;
real: NOTA4;
fim registro;
r = ALUNO.

Vejamos agora a definição de uma estrutura registro com outra estrutura registro embutida.

FUNC

| Nome | | |
|---|---|---|
| Endereço | | |
| RUA | NRO | CEP |
| Cidade | | Estado |
| Salario | | |

**Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

Será declarado da seguinte forma:

Tipo r = registro
texto: NOME;
ender: ENDEREÇO;
texto: CIDADE;
texto: ESTADO;
real: SALARIO;
fim registro;
r = FUNC;

Observe que o tipo **ender** ainda não foi definido (não é tipo básico), logo a definição está incompleta. Portanto, devemos definir o tipo **ender** que está embutido no registro r, da seguinte forma:

Tipo ender = registro
texto: RUA;
inteiro: NRO;
inteiro: CEP;
fim registro;

# ATRIBUIÇÃO

As operações realizadas sobre uma variável registro são as mesmas de uma variável comum. A atribuição em registros será feita da seguinte forma, considerando o registro FUNC:

FUNC . NOME ← “Joana Curadora”;
FUNC . ENDEREÇO . RUA ← “Avenida das Americas”;
FUNC . ENDEREÇO . NRO← 4200;
FUNC . ENDEREÇO . CEP← 22640102;
FUNC . CIDADE ← “Rio de Janeiro”;
FUNC . ESTADO ← “RJ”;
FUNC . SALARIO ← 1.00;

A utilização do ponto ( . ) indica que a variável FUNC possui campos e que NOME é um deles. Desta forma, devemos indicar o nome da variável seguido de um ponto ( . ) e logo após segue-se o nome do campo.

# PONTEIROS

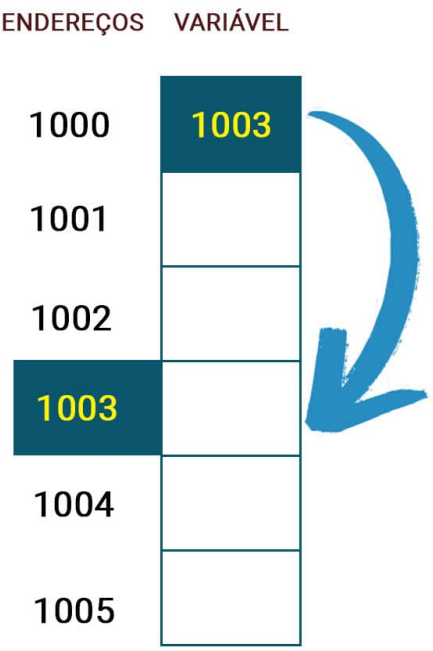
Ponteiro é um tipo especial de variável para conter o endereço de memória de outra variável. O endereço de memória é a localização de uma outra variável que já foi declarada anteriormente no programa.

## 💬 COMENTÁRIO

Dizemos que uma variável aponta para outra variável quando a primeira contém o endereço de memória ocupado pela segunda. Seu endereço indica em que parte da memória do computador a variável está alocada, mais especificamente, o endereço do primeiro byte ocupado.

Os ponteiros proporcionam um modo de acesso à variável sem referenciá-la diretamente (modo indireto de acesso).

Podemos perceber que a posição de memória 1000 possui o valor 1003, que é uma posição de memória. Considere, então, que você declare a variável PONT, que ocupa a posição de memória 1000. O conteúdo da variável PONT será a posição de memória 1003, que seria, por exemplo, da variável A. Então, dizemos que a variável PONT aponta para a variável A.




Fonte: O Autor

Figura – Referências de ponteiros. Fonte: O Autor.

Seguem algumas razões para a utilização de ponteiros:

Fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem (passagem por referência).

Passam matrizes e strings mais convenientemente de uma função para outra (usá-los no lugar de matrizes).

Manipular os elementos de matrizes mais facilmente, por meio da movimentação de ponteiros, no lugar de índices entre colchetes.

Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, em que um item deve conter referências a outro.

Alocar e desalocar memória dinamicamente do sistema.

Passar para uma função o endereço de outra função.

# DECLARAÇÃO

Quando se declara um ponteiro, deve-se declará-lo com o mesmo tipo (int, char, etc.) do bloco a ser apontado. Por exemplo, se queremos que um ponteiro aponte para uma variável int (bloco de 4 bytes em alguns ambientes), devemos declará-lo como int também.

## ❓ VOCÊ SABIA

Cada informação é representada por um certo conjunto de bytes, de acordo com as características da arquitetura e do compilador empregados.

Por exemplo, podemos ter em uma determinada arquitetura os seguintes valores para os tipos abaixo:

char:1 byte;

int:4 bytes;

float: 4 bytes;

short: 2 bytes;

long: 4 bytes;

double: 8 bytes.

Cada um destes conjuntos de bytes, que chamaremos de **bloco**, tem um nome e um **endereço** de localização específico na memória.

A sintaxe da declaração de um ponteiro é a seguinte:

tipo_ptr *nome_ptr1;

Ou

tipo_ptr *nome_ptr1, *nome_ptr2, ...;

onde:

*tipo_ptr*: é o tipo de bloco para o qual o ponteiro apontará;
*: é um operador que indica que nome_ptr1 é um ponteiro;
nome_ptr_1, nome_ptr_2, ...: são os nomes dos ponteiros;

## ★ EXEMPLO

Veja as seguintes instruções utilizando a linguagem C como referência:

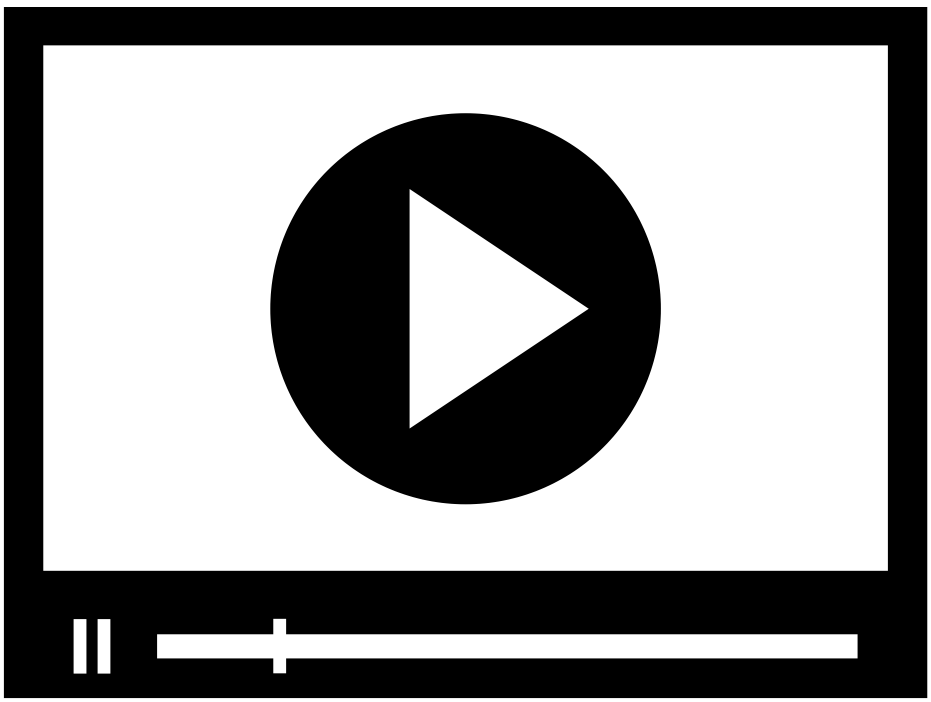
```
int *p;

float *ptr1, *ptr2;
```

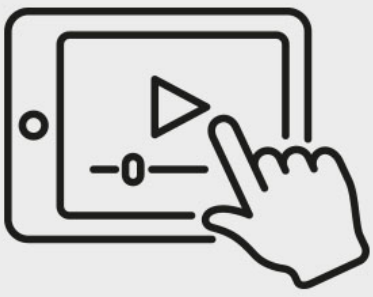
A primeira instrução declara um ponteiro chamado p que aponta para um inteiro. Este ponteiro aponta para o **primeiro** endereço de um bloco de quatro bytes.

Sempre é necessário declarar o tipo do ponteiro, mesmo que ele aponte para qualquer tipo de dado. Neste exemplo, dizemos que declaramos um ponteiro tipo int;

A segunda instrução declara dois ponteiros (ptr1 e ptr2) do tipo float, que apontará para o primeiro endereço da posição de memória ocupada por cada variável.



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# VERIFICANDO O APRENDIZADO

**1. ANALISE O TRECHO DE CÓDIGO ABAIXO E SELECIONE ENTRE AS ALTERNATIVAS O QUE SERÁ EXIBIDO APÓS A EXECUÇÃO.**

```
ALGORITMO QUESTÃO;
VAR
   A, B, D : INTEIRO;
   V[4] : INTEIRO;
INICIO
      A <- 3;
       B <- 2;
       V[4] <- {6,7,8,9}; // INICIALIZAÇÃO DO VETOR, DE FORMA CORRETA.
       A = (V[2] – V[0]) + A;
       D <- A * V[B]; //O CARACTERE * SIGNIFICA MULTIPLICAÇÃO
       ESCREVA(“ A SAÍDA É: “, D, “ “, V[1], A+2);
FIM.
```

**A)** A saída é: 24 75

**B)** A saída é: 40 77

**C)** A saída é: 21 67

**D)** A saída é: 40 75

**2. EM RELAÇÃO AOS PONTEIROS NAS LINGUAGENS DE PROGRAMAÇÃO, SELECIONE, DAS OPÇÕES SEGUINTES, AQUELA QUE JUSTIFICA SUA APLICAÇÃO:**

**A)** Flexibilidade de endereçamento e controle do gerenciamento de armazenamento dinâmico.

**B)** Aumento da legibilidade dos programas.

**C)** Facilidade de implementação no gerenciamento dinâmico.

**D)** Dificuldade na implementação de tipos primitivos.

## GABARITO

**1. Analise o trecho de código abaixo e selecione entre as alternativas o que será exibido após a execução.**

```
Algoritmo questão;
var
   a, b, d : inteiro;
   v[4] : inteiro;
inicio
      a <- 3;
       b <- 2;
       v[4] <- {6,7,8,9}; // inicialização do vetor, de forma correta.
       a = (v[2] – v[0]) + a;
       d <- a * v[b]; //o caractere * significa multiplicação
       escreva(“ a saída é: “, d, “ “, v[1], a+2);
fim.
```

A alternativa **"B "** está correta.

As variáveis a e b são declaradas e inicializadas, sendo a = 3 e b = 2.

V[4] = {6, 7, 8, 9} => O vetor v é declarado e inicializado com o conjunto de valores {6, 7, 8, 9}.
Configuração do vetor v:

| | | | |
|---|---|---|---|
| 6 | 7 | 8 | 9 |

Índices => 0 1 2 3

a = (v[2] – v[0]) + a; => a = (8-6) + 3 = 5

d = a * v[b]; => d = 5*8 = 40

escreva(“ a saída é: “, d, “ “, v[1], a+2); => exibe a saída: 40 77

Desta forma, a resposta correta é: **b. 40 77**

**2. Em relação aos ponteiros nas linguagens de programação, selecione, das opções seguintes, aquela que justifica sua aplicação:**

A alternativa **"A "** está correta.

Ponteiros são muito utilizados para gerenciar o armazenamento de endereços de memória. Sempre acessamos o endereço de um ponteiro, facilitando o acesso à sua área de memória.

# MÓDULO 3

---

- Definir a notação O e suas aplicações práticas

# INTRODUÇÃO

Considerando um determinado problema que deve ser resolvido computacionalmente, é necessário criar um algoritmo que precisa ser implementado em um computador como um conjunto de passos para a resolução do problema.

Fonte: Shutterstock

A estratégia para a definição de um algoritmo deve se basear nas seguintes atividades:

Especificar um algoritmo através das suas propriedades.

Definir sua arquitetura através das suas estruturas de dados.

Definir sua complexidade considerando o tempo de execução e espaço ocupado na memória.

Implementação em uma linguagem de programação.

Execução de testes caixa-branca para submeter entradas e verificar se as saídas obtidas estão de acordo com as propriedades especificadas de forma que se prove a corretude do algoritmo.

Segundo Moacir (2002), um algoritmo definido para um problema deve possuir as seguintes características:

## DESEMPENHO

Crucial para qualquer software.

## SIMPLICIDADE

Menor chance de gerar erros na implementação.

## CLAREZA

Escrito de forma clara e documentada para facilitar a manutenção.

## SEGURANÇA

Deve ser seguro para manipulação de dados.

## FUNCIONALIDADES

Implementar as funcionalidades requeridas.

## MODULARIDADE

Permite manutenção e reuso.

## INTERFACE AMIGÁVEL

Importante para a usabilidade dos usuários.

A corretude é uma característica fundamental para algoritmos, segundo Cormen *et al*. (2002): Um algoritmo é dito correto se, para cada instância de entrada, ele para de executar com a saída correta (ou informa que não há solução para aquela entrada). É desejável que um algoritmo termine e seja correto.

A corretude levanta algumas questões como: o algoritmo é correto, porém pode levar 3 anos para finalizar uma determinada execução. Isto pode levar a uma classe de problemas conhecidos como intratáveis, pois não existe um algoritmo que solucione o problema com demanda de recursos e tempo adequado. Esses problemas são conhecidos como NP-Difícil ou NP-Completo.

## RESUMINDO

Analisando as atividades do desenvolvimento de algoritmos, pode-se chegar à conclusão de que a análise de algoritmos se baseia na avaliação de algoritmos para a resolução de um problema que possua a melhor relação entre o tempo de execução e o espaço em memória utilizado para a execução do algoritmo.

Adicionalmente, a análise de algoritmos se preocupa com a corretude do algoritmo.

# O QUE É EFICIÊNCIA?

Segundo a Lei de Moore (1965), o poder computacional das máquinas dobra a cada 18 meses. Portanto, a princípio, pode-se considerar que um computador poderia resolver qualquer problema em um espaço de tempo curto. No entanto, os problemas crescem mais rápido que a capacidade computacional existente.

## ★ EXEMPLO

Por exemplo, conforme Enrooth (2020), o Google indexa bilhões de páginas através de seu indexador e esse volume de páginas cresce diariamente. Com o crescente volume, o algoritmo do Google deve conseguir indexar as páginas em um tempo adequado, ou seja, eficiente.

A eficiência pode ser associada aos recursos computacionais da seguinte forma:

A quantidade de espaço de armazenamento que utiliza.

A quantidade de tráfego que gera em uma rede de computadores.

A quantidade de dados que precisam ser movidos do disco ou para o disco.

# COMPLETE A FRASE

alterado

No entanto, para a maior parte dos problemas, a eficiência está relacionada

crescimento

indexação

tamanho

tempo

ao [ ] de execução em função do [ ] da entrada a ser processada. Considerando o exemplo de [ ] do Google, o tamanho da entrada é grande e em constante [ ]. Por esse motivo, de acordo com Enrooth (2020), o algoritmo de indexação do Google é [ ] de 350 a 400 vezes em um ano.

# RESPOSTA

A sequência correta é:

No entanto, para a maior parte dos problemas, a eficiência está relacionada ao **tempo** de execução em função do **tamanho** da entrada a ser processada. Considerando o exemplo de **indexação** do Google, o tamanho da entrada é grande e em constante **crescimento**. Por esse motivo, de acordo com Enrooth (2020), o algoritmo de indexação do Google é **alterado** de 350 a 400 vezes em um ano.

De acordo com Toscani e Veloso (2012), nos algoritmos de multiplicação de matrizes, pode-se ver claramente que um algoritmo mais simples pode ser uma escolha ineficiente com o crescimento muito rápido no tempo de execução conforme ocorre o aumento do tamanho do problema.

A tabela 1 mostra o desempenho desses dois algoritmos que calculam o determinante de uma matriz n x n, considerando tempos de operações de um computador real (Toscani e Veloso, 2012).

| N | Método de Cramer (n!) | Método de Gauss ($n^3$) |
|---|---|---|
| 2 | 22 $\mu s$ | 50 $\mu s$ |

| | | |
|---|---|---|
| 3 | 102 $\mu s$ | 159 $\mu s$ |
| 4 | 456 $\mu s$ | 353 $\mu s$ |
| 5 | 2,35s | 666 $\mu s$ |
| 10 | 1,19min | 4,95s |
| 20 | 15225 séculos | 38,63s |
| 40 | 5.1033 séculos | 0,315min |

**Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

Tabela 1 – Tamanho do problema x tamanho de execução. Adaptado de Laira e Toscano, 2012.

# COMPLETE A FRASE

Analisando a tabela 1, percebe-se que o Método de Cramer possui um algoritmo mais [ ], porém se torna rapidamente [ ] para tamanhos de entrada a partir de [ ] (ordem da matriz), enquanto o Método de Gauss se mantém um tempo [ ]

de resolução para um crescimento até

na ordem da matriz.

20

40

estável

**RESPOSTA**

inviável

A sequência correta é:

simples

Analisando a tabela 1, percebe-se que o Método de Cramer possui um algoritmo mais **simples** , porém se torna rapidamente **inviável** para tamanhos de entrada a partir de **20** (ordem da matriz), enquanto o Método de Gauss se mantém um tempo **estável** de resolução para um crescimento até **40** na ordem da matriz.

Outra motivação para o estudo é que mesmo com toda a evolução tecnológica da capacidade computacional das máquinas, não diminui a importância da análise do algoritmo mais eficiente.

Na tabela 2, é apresentado um algoritmo executando em uma máquina X na coluna 2 e em outra máquina Y com 10 vezes o poder de processamento da máquina:

| Complexidade de tempo | Tamanho máximo de problema resolvível na máquina lenta | Tamanho máximo de problema resolvível na máquina rápida |
|---|---|---|
| $\log_2 n$ | $X_0$ | $(X_0)^{10}$ |
| N | $X_1$ | $10X_1$ |
| N . $\log_2 n$ | $X_2$ | 10 $X_2$ (p/ $X_2$ grande) |
| $N^2$ | $X_3$ | 3,16 $X_3$ |

| | | |
|---|---|---|
| $N^3$ | $X_4$ | 2,15 $X_4$ |
| $2^n$ | $X_5$ | $X_5$ + 3,3 |
| $3^n$ | $X_6$ | $X_6$ + 2,096 |

**Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

Tabela 2 – Complexidade do algoritmo x tamanho máximo de problema resolvível Adaptado de Toscani e Veloso, 2012.

# COMPLETE A FRASE

10 elementos

baixa

décima potência

elementos

lenta

rápida

Analisando a tabela 2, para um problema com complexidade [ ], como $\log_2 n$, a máquina mais [ ] quase resolve uma entrada elevada à [ ] mais rápido que uma máquina mais [ ].

Em valores práticos, enquanto a máquina lenta resolve uma entrada de [ ], a máquina mais rápida consegue executar $10^{10}$ [ ] no mesmo tempo.

## RESPOSTA

A sequência correta é:

Analisando a tabela 2, para um problema com complexidade **baixa**, como $\log_2 n$, a máquina mais **rápida** quase resolve uma entrada elevada à **décima potência** mais rápido que uma máquina mais **lenta**.

Em valores práticos, enquanto a máquina lenta resolve uma entrada de **10 elementos**, a máquina mais rápida consegue executar $10^{10}$ **elementos** no mesmo tempo.

Para um problema com complexidade polinomial, a máquina mais lenta resolve uma entrada para 10 elementos, a máquina mais rápida consegue executar 3,16 x 10 elementos. O ganho na execução já não está diretamente ligado à capacidade da máquina (10 x mais rápida).

Analisando o caso de um problema com complexidade exponencial, $3^n$, a máquina mais lenta executa uma entrada de 10 elementos e a máquina mais rápida executa apenas 10 + 2,096 elementos.

A conclusão é que se a escolha do algoritmo for errada, uma máquina 10 vezes mais rápida não consegue ser eficiente proporcionalmente à sua capacidade computacional. Isto demonstra a importância da análise de algoritmos para se encontrar o mais eficiente possível, mesmo com máquinas poderosas computacionalmente.

# ANÁLISE DE ALGORITMOS

O objetivo principal é ser capaz de, dado um problema, mapeá-lo em uma classe de algoritmos e encontrar a melhor escolha entre os algoritmos, com base em sua eficiência.

## COMO PODE SER FEITA A ANÁLISE DE UM ALGORITMO PARA SE CHEGAR À CONCLUSÃO SE É EFICIENTE OU NÃO?

# RESPOSTA

O cálculo é feito através do número de passos básicos necessários em função do tamanho da entrada que o algoritmo recebe.

Como vimos na seção anterior, o desempenho do algoritmo deve ser desassociado do desempenho da máquina que executará o algoritmo. O foco principal é a redução da análise do número de operações realizadas em função do tamanho de entrada dos elementos.

Um tamanho de entrada geralmente é relativo ao número de elementos da entrada que são processados pelo algoritmo, como, por exemplo, o número de elementos em um arranjo, lista, árvore etc. Em alguns algoritmos, pode ser também o tamanho de um inteiro que é passado por parâmetro.

Os passos básicos se referem às operações primitivas utilizadas pela máquina para realizar uma determinada atividade. As atividades básicas são:

- Operações aritméticas.
- Comparações.
- Atribuições.
- Resolver um ponteiro ou referência.
- Indexação em um arranjo.
- Chamadas e retornos de funções e métodos.

No seguinte algoritmo que pode ser utilizado para se encontrar um número em uma tabela, existem 8 passos básicos.

I = 0; achou = F Passo Básico 1
Repita
I = i + 1 Passo Básico 2
Se tab[i] = ch Passo Básico 3
então achou = V Passo Básico 4
até achou ou i = n Passo Básico 5
se achou Passo Básico 6
então pos = i fim-se Passo Básico 7
retorne-saida (achou, pos) Passo Básico 8
fim-procedimento

A análise da complexidade de um algoritmo se baseia em dois tipos de complexidade:

Espacial - Este tipo de complexidade representa, por exemplo, o espaço de memória usado para executar o algoritmo.

Temporal - Este tipo de complexidade é o mais usado e pode ser dividido em dois grupos:

Tempo (real) necessário à execução do algoritmo.

Número de instruções necessárias à execução.

A análise temporal é a mais utilizada para estudo dos algoritmos.

# ANÁLISE ASSINTÓTICA

Para se realizar a análise assintótica de um algoritmo, deve-se considerar que para um algoritmo com tamanho de entrada n:

Cada operação (passo básico) leva o mesmo tempo constante.

A memória da máquina é eficiente.

A eficiência de um algoritmo é representada por uma função f(n) para uma entrada de tamanho n. A eficiência assintótica descreve a eficiência de um algoritmo quando n torna-se grande.

## ★ EXEMPLO

Neste caso, o exemplo de um bot de indexação de páginas é interessante, pois deve-se analisar a complexidade de um algoritmo para indexação de páginas web. Este algoritmo possui um tamanho de entrada de bilhões de elementos. Não faz sentido analisar a complexidade desse algoritmo para um tamanho de entrada de dez elementos.

Para comparar algoritmos, determinamos suas ordens de crescimento (eficiência assintótica). O algoritmo com a menor ordem de crescimento deverá executar mais rápido para tamanhos de entradas maiores.

O resultado da análise assintótica é a geração da complexidade assintótica do algoritmo, que é definida pelo crescimento da complexidade para entradas suficientemente grandes.

O fundamental da análise de algoritmos é a preocupação da análise de valores extremamente grandes, ou seja, tendendo ao infinito. Conforme apresentado, a computação está preocupada com a resolução de problemas com tamanho de entrada grande, que é a realidade atual dos softwares.

Então, um algoritmo assintoticamente mais eficiente é melhor para todas as entradas, exceto para entradas relativamente pequenas. Estas entradas pequenas são desprezadas para a análise de algoritmos. Na figura, é apresentado um exemplo do descarte de entradas pequenas.

Considerando as funções $f(n) = n + 4$ e $g(n) = n^2$;

Para: $n = 0, 1, 2$ temos $f(n) > g(n)$;

mas para $n >= 4$, temos $f(n) = n + 4 <= n + n = 2 . n <= n^2 = g(n)$.

Fonte: O Autor

Figura – Análise assintótica. Fonte: O Autor.

# ANALISANDO A FIGURA, QUAL O ALGORITMO MAIS EFICIENTE PARA A ÁREA DE ANÁLISE DE ALGORITMOS?

## RESPOSTA 1 RESPOSTA 2

## RESPOSTA 1

Por exemplo, considerando a entrada de tamanho 2 elementos, a quantidade de operações realizadas por f(n) = (6) é maior que a quantidade de operações realizadas por g(n) = (4).

# RESPOSTA 2

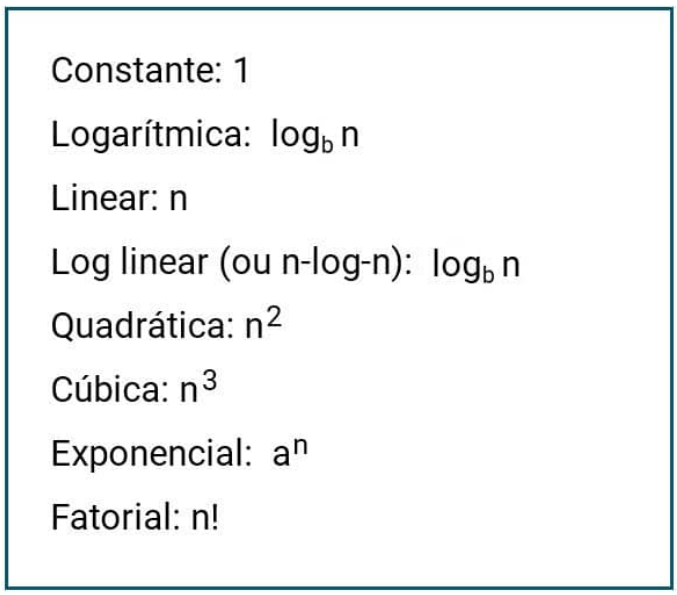
No entanto, considerando a entrada de tamanho 5 elementos, a quantidade de operações realizadas por f(n) = (9) é menor que a quantidade de operações realizadas por g(n) = (25). Portanto, para valores muito grandes (ordem assintótica), f(n) é mais eficiente que g(n).

Em outras palavras, respeitando a análise assintótica, o algoritmo f(n) tem ordem de crescimento menor para tamanhos grandes de entrada n.

Como realizar uma comparação assintótica entre funções?

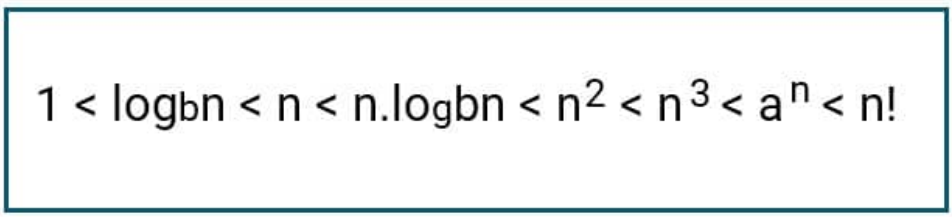
Para tornar isto possível, é preciso introduzir uma simplificação do modo de comparar funções.

Essa comparação só leva em conta a “velocidade de crescimento” das funções. Assim, ela despreza fatores multiplicativos (pois a função $2n^2$, por exemplo, cresce tão rápido quanto $10n^2$) e despreza valores pequenos do argumento (a função $n^2$ cresce mais rápido que 100n, embora $n^2$ seja menor que 100n quando n é pequeno).




autor

A análise assintótica de algoritmos gera um conjunto de funções, conforme a figura




autor/shutterstock

Considerando a forma de comparação assintótica de funções, podemos ordenar as funções, conforme a figura

Considerando a figura, nessa matemática, as funções são classificadas em "ordens" e todas as funções de uma mesma ordem são "equivalentes". Então, por exemplo, $100\ n^2$, $100 + 3\ n^2$, $80 + n^{2/3}$ são equivalentes. Todas as funções possuem a ordem $n^2$.

# NOTAÇÃO O (BIG OH)

Existem formas de se analisar cada função matemática de um algoritmo para se provar que uma função é assintoticamente superior a outra. Desta forma, pode-se definir qual algoritmo é melhor que o outro para se resolver um determinado problema.

Cada algoritmo possui um comportamento para cada entrada de dados e sempre existirá determinado conjunto de dados de entrada em que o algoritmo se comportará da pior forma possível. A pior forma possível, segundo Cormen *et al*. (2012), indica que o algoritmo realizará um conjunto maior de passos básicos para se chegar ao resultado correto.

Um exemplo prático são os algoritmos ordenação, como, por exemplo, o Quicksort (Cormen *et al., 2012*). Se for inserido para o algoritmo do Quicksort um conjunto de entrada com os dados quase ordenados, este se comporta da pior forma possível, que, para a área de análise de algoritmos, é chamado do pior caso para o algoritmo.

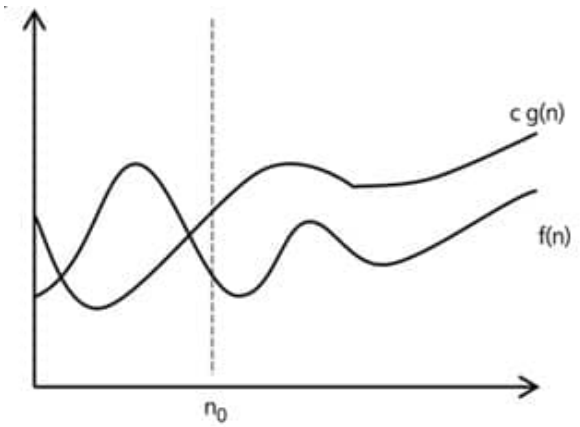
A notação O é a forma matemática de se prever o pior caso para cada algoritmo para determinada solução de um problema.

# DEFINIÇÃO

Dadas as funções assintoticamente não negativas f e g, dizemos que f está na ordem O de g e escrevemos f = O(g) se f(n) ≤ c · g(n) para algum c positivo e para todo n suficientemente grande.

Em outras palavras, existe um número positivo c e um número n0 tais que f(n) ≤ c · g(n) para todo n maior que n0. (Cormen, 2012).

Podemos dizer que f (n) pertence O(g(n)), mas em geral se escreve f (n) = O(g(n)). Para n suficientemente grande, g(n) é cota superior para f(n) com um fator constante.




Fonte: Autor

Figura – Definição Notação O – f(n) = O(g(n)). Fonte: Cormen et al., 2002.

Para todos os valores de n à direita de n0, o valor de f (n) reside em c * g(n) ou abaixo desse. Formalmente, a função g(n) é um limitante assintótico superior para f (n).

Exemplo: $4n^2 = O(n^3)$

Analisando o exemplo acima, pode-se inferir nessa equação como sendo $4n^2 \leq O(n^3)$. A taxa de crescimento de $4n^2$ é menor ou igual à taxa de $n^3$, ou seja, $n^3$ é assintoticamente superior a $4n^2$.

# EXEMPLOS DE FUNÇÕES O

Para todos os exemplos, de acordo com a definição, é preciso encontrar c que seja sempre maior ou igual a n para todo valor de n0. Assim, eu provo que g(n) é limitante superior à função f(n).

# TEORIA NA PRÁTICA

**Exemplo 1:** Provar que f(n)= $n^2$ é O(n)

## RESOLUÇÃO

$n^2 \leq c \cdot n \Rightarrow n \leq c$ é impossível, pois c deve ser constante

**Conclusão:** Não existe um c multiplicado por g(n) que torne g(n) superior assintoticamente a f(n)

**Exemplo 2:** $3n^3 + 20n^2 + 5$ é $O(n^3)$

## RESOLUÇÃO

É preciso encontrar $c > 0$ e $n >= 1$ tais que $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ para $n \geq n_0 \Leftrightarrow$ como $3n^3 + 20n^2 + 5$ $(3 + 20 + 5)\ n^3$, podemos tomar $c = 28$ e qualquer $n_0 > 1$

**Conclusão:** Existe um c (28) que multiplicado por g(n)($n^3$) torna g(n) superior assintoticamente a f(n) a partir de $n_0 = 1$

**Exemplo 3:** $2^{n+2}$ é $O(2^n)$

## RESOLUÇÃO

É preciso $c > 0$ e $n0 > 1$ tais que $2^{n+2} \leq c * 2^n$ para todo $n \geq n0$ note que $2^{n+2} = 2^{2 \cdot 2n} = 4 * 2n$ ó assim, basta tomar, por exemplo, $c = 4$ e qualquer n

**Conclusão:** Existe um c (4) que multiplicado por g(n) ($n^{2n}$) torna g(n) superior assintoticamente a f(n) a partir de $n_0 = 1$

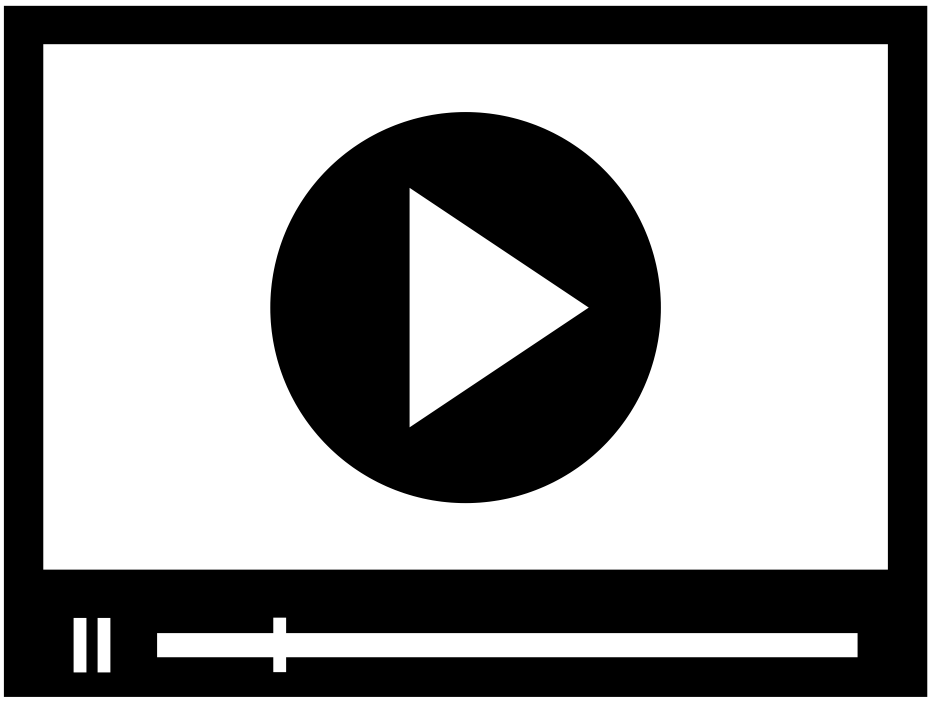
# COTA SUPERIOR DE PROBLEMAS

Seja dado um problema, por exemplo, multiplicação de duas matrizes quadradas n x n. Conhecemos um algoritmo para resolver este problema (pelo método trivial) de complexidade $O(n^3)$. Sabemos, assim, que a complexidade deste problema não deve superar $O(n^3)$, uma vez que existe um algoritmo que o resolve com esta complexidade.

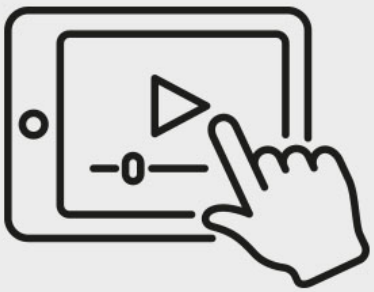
## ★ EXEMPLO

Uma cota superior ou limite superior (upper bound) deste problema é $O(n^3)$. A cota superior de um problema pode mudar se alguém descobrir outro algoritmo melhor. Outro exemplo são os problemas de ordenação de dado cuja cota superior do problema é O(n log n). Caso, por exemplo, seja descoberto um algoritmo que resolva em n, o problema mudará sua cota superior, poderá ser alterada O (n).

Resumidamente, o objetivo é se conseguir, através de análise matemática de diversos algoritmos, qual o algoritmo mais eficiente para a resolução do problema.



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# VERIFICANDO O APRENDIZADO

**1. UTILIZAMOS A NOTAÇÃO *O* PARA CLASSIFICAR O DESEMPENHO DE UM ALGORITMO. DIZEMOS QUE F=O(G) QUANDO PARA QUALQUER VALOR DE N MAIOR QUE N0 A FUNÇÃO CG(N)>F(N).**

**A ANÁLISE DA COMPLEXIDADE COMPUTACIONAL DE UM ALGORITMO É SEMPRE FEITA NA VISÃO DO PESSIMISTA, ISTO É, NA ANÁLISE DO PIOR CASO APRESENTADO AO ALGORITMO.**
**O MÉTODO DA BOLHA É UM ALGORITMO DE ORDENAÇÃO BASEADO EM TROCAS. A ANÁLISE DE COMPLEXIDADE DE PIOR CASO DO MÉTODO DA BOLHA DIZ QUE A COMPLEXIDADE COMPUTACIONAL DO ALGORITMO É ON2, CLARO, PARA O PIOR CASO. . É CONHECIDO QUE NOS MELHORES CASOS, O MÉTODO DA BOLHA EXECUTA EM UM TEMPO PROPORCIONAL À ON, PORÉM, NÃO LEVAMOS ISTO EM CONSIDERAÇÃO PARA ANÁLISE DO COMPLEXIDADE COM A VISÃO DO PESSIMISTA. UM OUTRO MÉTODO DE ORDENAÇÃO É O MERGE SORT, SUA COMPLEXIDADE COMPUTACIONAL É ON LOGN PARA TODAS AS INSTÂNCIAS, OU SEJA, TODOS AS INSTÂNCIAS SUBMETIDAS AO MERGE SORT SÃO DO PIOR CASO. COM BASE NESTES FATOS, INTERPRETE AS AFIRMAÇÕES ABAIXO SEGUNDO SEU VALOR VERDADE.**

**I - PODE-SE AFIRMAR QUE O MERGE SORT SEMPRE EXECUTA MAIS RÁPIDO QUE O MÉTODO DA BOLHA, PARA A MESMA INSTÂNCIA SUBMETIDA AOS DOIS ALGORITMOS.**
**II - PODE-SE AFIRMAR QUE O MERGE SORT EXECUTA MAIS RÁPIDO QUE O MÉTODO DA BOLHA QUANDO INSTÂNCIAS DO PIOR CASO DO MÉTODO DA BOLHA SÃO SUBMETIDAS A AMBOS OS ALGORITMOS.**
**III - EXISTE A POSSIBILIDADE DE EXISTIREM INSTÂNCIAS QUE EXECUTAM MAIS RÁPIDO NO MÉTODO DA BOLHA DO QUE NO MERGE SORT. ENTRETANTO, ESTAS INSTÂNCIAS NÃO SÃO LEVADAS EM CONSIDERAÇÃO PORQUE NÃO DEVEM FAZER PARTE DA ANÁLISE DO PESSIMISTA.**

**A)** Somente I e II são verdadeiras.

**B)** Somente II e III são verdadeiras.

**C)** Somente I é verdadeira

**D)** Somente II é verdadeira.

**E)** Somente III é verdadeira.

**2. CONSIDERE UMA EMPRESA DESENVOLVEDORA DE SOFTWARE DE ALTA COMPLEXIDADE E CRÍTICOS PARA GERENCIAMENTO DA MALHA AÉREA DO PAÍS. NO ENTANTO, A EMPRESA DEMITIU SEU ARQUITETO DE SOFTWARE ESPECIALIZADO EM OTIMIZAÇÃO DOS ALGORITMOS DA EMPRESA. DURANTE O SEU PERÍODO DE DEMISSÃO, O ARQUITETO LHE DEIXOU UMA TABELA COM OS CÁLCULOS QUE ELE FEZ PARA A EMPRESA COM O INTUITO DE PROVAR A**

**IMPORTÂNCIA DA COMPLEXIDADE PARA A DETERMINAÇÃO DO DETERMINANTE DE MATRIZES. ANALISANDO A TABELA, MARQUE A OPÇÃO CORRETA SOBRE A ANÁLISE.**

| N | ALGORITMO RESOLUÇÃO MÉTODO DE CRAMER | ALGORITMO RESOLUÇÃO MÉTODO DE GAUSS ($N^3$) |
|---|---|---|
| 2 | 22MS | 50MS |
| 3 | 102MS | 159MS |
| 4 | 456MS | 353MS |
| 5 | 2,35MS | 666MS |
| 10 | 1,19MIN | 4,95MS |
| 20 | 15 225 SÉCULOS | 38,63MS |
| 40 | $5.10^{33}$ SÉCULOS | 0,315S |

**ATENÇÃO! PARA VISUALIZAÇÃOCOMPLETA DA TABELA UTILIZE A ROLAGEM HORIZONTAL**

**A)** O método de Gauss é o melhor algoritmo para todos os conjuntos de entrada de dados possíveis para o cálculo do determinante de uma matriz.

**B)** O método de Cramer é o melhor algoritmo para todos os conjuntos de entrada de dados possíveis para o cálculo do determinante de uma matriz.

**C)** O método de Cramer e Gauss são equivalentes, pois possuem esforço semelhante para os casos mais comuns de ordem de matrizes.

**D)** O algoritmo de Gauss é melhor cálculo de determinantes para matrizes com ordem maior ou igual a 4.

---

## GABARITO

**1. Utilizamos a notação *O* para classificar o desempenho de um algoritmo. Dizemos que f=O(g) quando para qualquer valor de n maior que n0 a função cg(n)>f(n).**
**A análise da complexidade computacional de um algoritmo é sempre feita na visão do pessimista, isto é, na análise do pior caso apresentado ao algoritmo.**
**O método da bolha é um algoritmo de ordenação baseado em trocas. A análise de complexidade de pior caso do método da bolha diz que a complexidade computacional do algoritmo é On2, claro, para o pior caso. . É conhecido que nos melhores casos, o método da bolha executa em um tempo proporcional à On, porém, não levamos isto em consideração para análise do complexidade com a visão do pessimista. Um outro método de ordenação é o merge sort, sua complexidade computacional é On logn para todas as instâncias, ou seja, todos as instâncias submetidas ao merge sort são do pior caso. Com base nestes fatos, interprete as afirmações abaixo segundo seu valor verdade.**

**I - Pode-se afirmar que o merge sort SEMPRE executa mais rápido que o método da bolha, para a mesma instância submetida aos dois algoritmos.**
**II - Pode-se afirmar que o merge sort executa mais rápido que o método da bolha quando instâncias do pior caso do método da bolha são submetidas a ambos os algoritmos.**
**III - Existe a possibilidade de existirem instâncias que executam mais rápido no método da bolha do que no merge sort. Entretanto, estas instâncias não são levadas em consideração porque não devem fazer parte da análise do pessimista.**

A alternativa **"B "** está correta.

Justificativa: A afirmativa I é falsa, a análise de complexidade do pior caso, como o próprio nome diz, leva em consideração o pior comportamento do algoritmo, isto é, a configuração da instância que demanda o maior número de operações para a solução do problema. Isto não quer dizer que não existam instâncias que demandem muito menos operações. O pior caso do método da bolha configura-se por instâncias ordenadas em ordem reversa a desejada. Neste caso o algoritmo executa em um tempo de On2, porém, o melhor caso do método da bolha é a submissão de sequências já ordenadas e, neste caso, o algoritmo demanda On operações de comparação para concluir que a sequência já está ordenada. O merge sort é um algoritmo que sempre executa em um tempo proporcional a On log(n).

Portanto, em instâncias pertencentes ao melhor caso do método da bolha executam mais rápido quando aplicadas ao algoritmo da bolha.

II – Verdadeiro, no pior caso, o método da bolha demanda On2 operações para ordenar o vetor e o merge sort On log(n).

III – Verdadeiro, as instâncias do melhor caso, que executam em On.

**2. Considere uma empresa desenvolvedora de software de alta complexidade e críticos para gerenciamento da malha aérea do país. No entanto, a empresa demitiu seu arquiteto de software especializado em otimização dos algoritmos da empresa. Durante o seu período de demissão, o arquiteto lhe deixou uma tabela com os cálculos que ele fez para a empresa com o intuito de provar a importância da complexidade para a determinação do determinante de matrizes. Analisando a tabela, marque a opção correta sobre a análise.**

| n | Algoritmo resolução Método de Cramer | Algoritmo resolução Método de Gauss ($n^3$) |
|---|---|---|
| **2** | **22ms** | **50ms** |
| **3** | **102ms** | **159ms** |
| **4** | **456ms** | **353ms** |
| **5** | **2,35ms** | **666ms** |
| **10** | **1,19min** | **4,95ms** |
| **20** | **15 225 séculos** | **38,63ms** |
| **40** | **$5.10^{33}$ séculos** | **0,315s** |

**Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

A alternativa **"D "** está correta.

A questão é importante para fixação do conceito matemático de complexidade x tempo de execução dos algoritmos. O algoritmo de Gauss tem desempenho melhor a partir da matriz de ordem 4.

# MÓDULO 4

---

- Empregar a análise da complexidade dos algoritmos

# INTRODUÇÃO

A análise prática de algoritmos está associada diretamente ao cálculo do desempenho de cada algoritmo definido para a solução de um determinado problema. Esse cálculo irá gerar um big O para cada algoritmo. Desta forma, deverá ser escolhido o algoritmo que gera o menor big O de acordo com a ordenação apresentada na figura.

Fonte: Shutterstock

Segundo Moacir (2010), algumas premissas importantes podem ser utilizadas para análise prática, como:

Se f (n) for um polinômio de grau d, então f (n) é O ($n^d$ ), devem ser desprezados os termos de menor ordem e desprezados os fatores constantes. Por exemplo, uma função como f(n) = 3 $n^3$+ 2 $n^2$ + 10, seguindo a premissa, transformar-se-á em O($n^3$).

Use a menor classe de funções possível, para simplificação. Por exemplo, a função f(n)=2n é O(n), ao invés de 2n ser O(2n).

Use a expressão mais simples possível, para simplificação. Por exemplo, a função f(n)= 3n + 5 é O(n), ao invés de f(n) = 3n + 5 possuir O(3n).

Ao comparar dois algoritmos com tempo de execução: f (n) = $10^{100n}$, e g(n) = 10n log n. Pela análise assintótica, o primeiro é mais eficiente No entanto, $10^{100}$ é o número estimado (por alguns astrônomos) como o limite superior para a quantidade de átomos no universo observável 10n log n > $10^{100n}$ apenas para n > $(2^{10})^{99}$

# PRINCÍPIOS BÁSICOS

A complexidade do algoritmo é baseada na quantidade de passos básicos executados para o resultado final do algoritmo.

Um algoritmo geralmente tem componentes que são formados por passos básicos que dão suas contribuições para o desempenho e para a complexidade do algoritmo.

Conforme Laira (2010), algumas componentes de um algoritmo são sempre executadas. Outras componentes definem alternativas, que são executadas conforme o caso. Além disso, pode ser conveniente encarar uma componente como a restrição de um algoritmo a certas entradas (fornecidas por outras componentes).

## COMPONENTES CONJUNTIVAS

Uma componente de um algoritmo é dita conjuntiva quando ela é sempre executada em qualquer execução do algoritmo.

Consideremos um algoritmo a com duas componentes conjuntivas f(n) e g(n), o desempenho do algoritmo A sobre a entrada n é dado por O(h(n)) = O(f(n)) + O(g(n)).

Considerando o princípio das componentes conjuntivas, se duas componentes são conjuntivas, deve ser considerada a soma da cota superior entre os dois componentes conforme a fórmula:

$$O(h(n)) = O(f(n)) + O(g(n)).$$

**Atenção!** Para visualização completa da equação utilize a rolagem horizontal

Resumidamente, como se trata do big O, complexidade do pior caso, deve ser considerado o pior caso dos dois componentes. Como são sempre executados, deve ser a soma do O das duas componentes.

# COMPONENTES DISJUNTIVAS

Uma componente de um algoritmo é dita disjuntiva quando são executadas dependendo do conjunto entrada de valores do algoritmo. Consideremos um algoritmo

A com duas componentes disjuntivas f(n) e g(n), o desempenho do algoritmo A sobre entrada n é dado por O(h(n)) = max(O(f(n)),O(g(n))).

Considerando o princípio das componentes disjuntivas, se duas componentes são disjuntivas, deve ser considerada a maior cota superior entre os dois componentes conforme a fórmula:

$$O(h(n)) = max(O(f(n)), O(g(n))).$$

**Atenção!** Para visualização completa da equação utilize a rolagem horizontal

Resumidamente, como se trata do big O, complexidade do pior caso, deve ser considerado o pior caso da execução entre os dois componentes que não são obrigatoriamente executados a cada execução.

# PRINCÍPIO DA ABSORÇÃO

Se f é absorvida por g, sua soma pontual f + g é O( g ).

O máximo assintótico em ordem, max (f(n), g(n)) de duas funções f e g deve ser sua cota superior - em sentido assintótico - para as ordens de ambas.

# ETAPA 1

Por exemplo, duas componentes disjuntivas f(n) = O(1) e g(n) = O(n),

## ETAPA 2

Que geram a soma de componentes O(1) + O(n).

## ETAPA 3

Pelo princípio da absorção, a complexidade final será O(n), ou seja o max(O(1),O(n)).

Frequentemente, toma-se como máximo assintótico a soma ou o máximo pontual, que foi o caso do exemplo.

# COMPLEXIDADES PESSIMISTAS DE ESTRUTURAS ALGORÍTMICAS

Para realizar a análise do pior caso, ou a chamada complexidade pessimista, algumas regras básicas devem ser consideradas conforme tabela 3:

| |
|---|
| f(n) = O(f(n)) |
| c * O(f(n)) = O(f(n)) , c = constante |
| O(f(n)) + O(f(n)) = O(f(n)) |
| O(O(f(n)) = O(f(n)) |

| |
|---|
| O(f(n)) + O(g(n)) = O(max(f(n),g(n))) |
| O(f(n))O(g(n)) = O(f(n)g(n)) |
| f(n)O(g(n)) = O(f(n)g(n)) |

**Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

Tabela 3 – Regras da complexidade de O. Fonte: Barrere, 2020.

A seguir, serão analisadas as complexidades para algumas operações.

**Atribuição:** O tempo de execução é o tempo do comando da atribuição. Se a operação de atribuição for dentro de uma estrutura de repetição, deve ser multiplicada pelo número de vezes que é executada.

# EXEMPLO 1: A COMPLEXIDADE DA ATRIBUIÇÃO É O(1)

(1) a = a * i (1)

## CONCLUSÃO

## CONCLUSÃO

Na linha (1) é executada uma vez a atribuição de valores para a, isto gera O(1).

# EXEMPLO 2: A COMPLEXIDADE DA ATRIBUIÇÃO NESTE CASO É O(N)

(1) VetA[] = VetI[];

## CONCLUSÃO

### CONCLUSÃO

Na linha (1) é executado O(n) vezes para a atribuição de todos os elementos (i) do VetI para o VetA. Isto gera O(n) passos para essa atribuição.

**Repetições:** O tempo de execução é pelo menos o tempo dos comandos dentro da repetição multiplicada pelo número de vezes que é executada.

# EXEMPLO 1: A COMPLEXIDADE DA REPETIÇÃO É O(N)

(1) para i de 1 ate n faca (n)

(2)a = a * i (1)

## CONCLUSÃO

### CONCLUSÃO

Na linha (1) é executado O(n) vezes na linha(2) o tempo do comando interno (1). Isto gera O(n).O(1) para essa repetição número de execuções internas.

# EXEMPLO 2: A COMPLEXIDADE DA REPETIÇÃO É O(N)

(1) para i de 1 ate n faca (O(n) - repetição)

(2) a = a * i (O(1) - Atribuição)

(3) c = c + a (O(1) - Atribuição)

## CONCLUSÃO

## CONCLUSÃO

Na linha (1) é executado O(n) vezes o tempo do comando interno de atribuição (2) e o tempo do comando interno de atribuição (3). Isto gera O(n) . O(1) . O(1) que equivale a O(n) repetições da execução interna.

**Repetições aninhadas:** Análise feita de dentro para fora, o tempo total é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições internas (aninhadas).

# EXEMPLO 1: A COMPLEXIDADE DA REPETIÇÃO É $O(N^2)$

(1) para i de 1 ate n faca – O(n)

(2) para j de 0 ate n-1 faca – O(n-1)

(3) a = a*(i+j) – O(1)

## CONCLUSÃO RESOLUÇÃO

## CONCLUSÃO

Na linha (1) é executado O(n) vezes a estrutura de repetição. Na linha (2) é executada O(n – 1) vezes a estrutura de repetição. Na linha (3) é O(1) a atribuição de atribuição. A complexidade final será a multiplicação da complexidade a linha (3) pela linha (2) e pela linha (1) – de dentro para fora.

## RESOLUÇÃO:

**Resolução:** $O(O(1).O(n).O(n)) \Leftrightarrow O(1.n.n) \Leftrightarrow O(n^2)$.

Comentário: Por que O(n – 1) se transforma em O(n)? Devido às regras de análise assintótica, o mais importante é considerar a ordem desprezando os fatores multiplicadores e as constantes. Com isso, O(n -1) possui a ordem n, portanto se simplifica em O(n).

# EXEMPLO 2: A COMPLEXIDADE DA REPETIÇÃO É $O(N^3)$

(1) para i de 1 ate n faca – O(n)

(2) para j de 0 ate n-1 faca – O(n-1)

(3) para k de 0 ate n-2 faca – O(n-2)

(4) a = a*(i+j) – O(1)

## CONCLUSÃO RESOLUÇÃO

## CONCLUSÃO

Na linha (1) é executado O(n) vezes a estrutura de repetição. Na linha (2) é executada O(n – 1) vezes a estrutura de repetição. Na linha (3) é executada O(n – 2) vezes a estrutura de repetição. Na linha (4) é O(1) a atribuição de atribuição. A complexidade final será a multiplicação da complexidade a linha (4), pela linha (3), pela linha (2) e pela linha (1) – de dentro para fora.

## RESOLUÇÃO

**Resolução:** O(O(1).O(n).O(n).O(n)) ⇔ O(1.n.n.n) ⇔ O(n3).

Comentário: Por que O(n – 2) se transforma em O(n)? Devido às regras de análise assintótica, o mais importante é considerar a ordem desprezando os fatores multiplicadores e as constantes. Desta forma, O(n-2) possui a ordem n, portanto se simplifica em O(n).

**Condições:** A complexidade nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos dentro do bloco do “então” e do “senão”. A complexidade gera a seguinte fórmula: O(O(condicional) + max (O(então), O(senão))

# EXEMPLO 1: A COMPLEXIDADE DA CONDIÇÃO É O(1)

(1) se (a < b) então - O(1)

(2) a = a + 1 - O(1)

## DETALHAMENTO: SOLUÇÃO

## DETALHAMENTO:

Na linha (1) é executada a comparação com complexidade constante O(1). A linha (2) é executada somente caso a condição na linha (1) seja verdadeira. Se for executada, possui complexidade constante O(1) por ser um comando de atribuição.

## RESOLUÇÃO:

O(O(condicional) + max (O(então), O(senão)) ⇔ O(O(condicional) + O(então)) ⇔ O( O(1) + (O(1)) ⇔ O(1)

# EXEMPLO 2: A COMPLEXIDADE DA CONDIÇÃO É O(N)

(3) se (a < b) então - O(1)

(4) a = a + 1 - O(1)

(5) senao

(6) para i de 1 ate n-1 faca - O(n - 1)

(7) a = a*i - O(1)

## DETALHAMENTO RESOLUÇÃO

## DETALHAMENTO:

Na linha (1) é executada a comparação com complexidade constante O(1). A linha (2) é executada somente caso a condição na linha (1) seja verdadeira. Se for executada, possui complexidade constante O(1) por ser um comando de atribuição. A linha (4) é executada somente caso a condição na linha (1) seja falsa. A linha (4) possui complexidade O(n - 1) por ser uma repetição de

n – 1 passos básicos. A linha (5) possui complexidade constante O(1) por ser um comando de atribuição.

## RESOLUÇÃO:

O(O(condicional) + max (O(então), O(senão)) ⇔ O(senão) = O(n). O(1) ⇔ O(senão) = O(n) ⇔ O( O(1) + max(O(1),O(n)) – ⇔ O(O(1) + O(n)) ⇔ **O(n)**

Comentários:

1 - Por que o max(O(1), O(n)) resultou em O(n)? Esse é um caso de componentes disjuntivas, conforme visto em Princípio da absorção . Como está sendo tratado sempre o pior caso, precisa-se obter o pior caso entre o “então” e o “senão”, no caso é O(n).

2 – Por que O(O(1) + O(n)) resultou em O(n)? Pelo princípio da absorção apresentado em Componentes disjuntivas, o O(1) pode ser absorvida pelo O(n) pela maior ordem.

**Repetições indefinidas:**A complexidade nunca é maior do que o tempo do teste da condição multiplicado pelo número máximo possível de execução da iteração e pela complexidade de cada iteração A complexidade gera a seguinte fórmula: O(O(condicionalrepetição) + O(O(númerorepeticoes).O(comandosrepeticao)))

# EXEMPLO 1: A COMPLEXIDADE DA REPETIÇÃO É O(N)

(1) enqto a < n faça - O(1) comparação e O(n) da repetição

(2) a = a*(i+j) – O(1)

## DETALHAMENTO RESOLUÇÃO

## DETALHAMENTO:

Na linha (1) é executada a comparação da repetição como valor constante O(1) e O(n) vezes o máximo de execução da estrutura de repetição. Na linha (2) é executada a atribuição com valor constante O(1).

## RESOLUÇÃO

$O(O(1) + O(O(n).O(1))) \Leftrightarrow O(1 + O(n)) \Leftrightarrow O(1 + n) \Leftrightarrow O(n)$

**Chamadas às sub-rotinas:** A sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa que a chamou.

# EXEMPLO 1: A COMPLEXIDADE DA CHAMADA É O(N)

(1) a = a + 1 - O(1)

Soma (a,b) - O(n)

## DETALHAMENTO RESOLUÇÃO

## DETALHAMENTO

Na linha (1) é executada a atribuição com complexidade constante O(1). Na linha (2) é executada a chamada da sub-rotina Soma(a,b) com complexidade O(n).

## RESOLUÇÃO

$O(O(\text{linha1}) + O(\text{Soma}(a,b)) \Leftrightarrow O(O(1) + (O(n)) \Leftrightarrow \mathbf{O(n)}$

# EXEMPLO DE CÁLCULO DE COMPLEXIDADES

A partir das regras definidas para cada estrutura algorítmica da Complexidades pessimistas de estruturas algorítmicas, serão apresentados diversos exemplos práticos para o cálculo do O para a complexidade de pior caso dos algoritmos.

## EXEMPLO FUNÇÃO MINMAX

**Veja esse algoritmo para o cálculo dos valores mínimo e máximo de um vetor, foi adaptado de Laira, 2013.**

```
Funcão MinMax(tabela : Inteiros)
(1) p = 0
(2) min = tabela [p]
(3) max = tabela [p]
(4) para i de p + 1 até q faça
(5) se tabela[ i ] > max
(6) então max = tabela[ i ]
(7) se tabela[ i ] < min
(8) então min = tabela[ i ]
(9) fim-para
(10) retorne MinMax(min,max)
```

O ideal é dividir por componentes do algoritmo para se obter o valor da complexidade.

| |
|---|
| **O(O(linha1), O(linha2), O(linha3) = O(1) + O (1) + O (1) ⇔ O (1 + 1 + 1) ⇔ O(3) ⇔ O(1)** |
| O(linha5) + O(linha6) = O(1) + O(1) ⇔ O(2) ⇔ O(1) |
| O(linha7) + O(linha8) = O(1) + O(1) ⇔ O(2) ⇔ O(1) |
| **O(para linha4 … linha8) = O( q - p) . O(linha5...linha8) ⇔ O (q – p) equivale a O(n) pois são n passos da iteração ⇔ O(n).O(1) ⇔ O(n)** |
| O(linha10) = O(1) |

O(minxmax) = O(para linha1...linha3) + O(para linha4 … linha8) + O(linha10) ⇔ O(1) + O(n) + O(1) ⇔ O(1+n+1) ⇔ O(n+ 2) ⇔ O(n)

**Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

O(O(linha1), O(linha2), O(linha3) = O(1) + O (1) + O (1) ⇔ O (1 + 1 + 1) ⇔ O(3) ⇔ O(1) – Complexidade constante de atribuição.

O(linha5) + O(linha6) = O(1) + O(1) ⇔ O(2) ⇔ O(1) – A linha 5 é uma condição com complexidade constante; A linha 6 é uma atribuição com complexidade constante.

O(linha7) + O(linha8) = = O(1) + O(1) ⇔ O(2) ⇔ O(1) – A linha 7 é uma condição com complexidade constante; A linha 8 é uma atribuição com complexidade constante.

O(para linha4 … linha8) = O( q - p) . O(linha5...linha8) ⇔ O (q – p) equivale a O(n) pois são n passos da iteração ⇔ O(n).O(1) ⇔ O(n) - É uma repetição de n passos como duas condicionais.

O(linha10) = O(1) – Retorno da função possui complexidade constante.

O(minxmax) = O(para linha1...linha3) + O(para linha4 … linha8) + O(linha10) ⇔ O(1) + O(n) + O(1) ⇔ O(1+n+1) ⇔ O(n+ 2) ⇔ O(n) – São componentes conjuntivos que são sempre executados.

# OPERAÇÃO COM MATRIZES

Procedimento OperacaoMatriz()

(1)i, j: inteiro

(2)A: vetor inteiro de n posicoes

(3)i = 1

(4)enquanto (i < n) faca

(5)A[i] = 0

(6)i = i + 1

(7)para i = 1 ate n faca

(8)para j = 1 ate n faca

(9)A[i] = A[i] + (i*j)

(10)fim

# CÁLCULO DA COMPLEXIDADE

| |
|---|
| O(linha3) = O(1) |
| O(para linha4...linha6) – O(linha4) . (O(linha5) + O(linha6)) ⇔ O(linha4) = O(n) ⇔ O(linha5) + O(linha6) ⇔ O(1) + O(1) ⇔ O(1) ⇔ O(n). |
| O(1) ⇔ O(n) |
| O(para linha7...linha9) – O(linha7) . O(linha8) . O(linha9) ⇔ O(linha7) = O(n) O(linha8) = O(n) ⇔ O(linha9) = O(1) ⇔ O(n) . O(n) . O(1) ⇔ O(n.n.1) ⇔ $O(n^2)$ |
| O(operacaomatriz) = O(linha3) + O(para linha4...linha6) + O(para linha7...linha9) ⇔ O(1) + O(n) + $O(n^2)$ ⇔ $O(1 + n + n^2)$ |

**Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

| |
|---|
| O(linha3) = O(1) – complexidade constante de uma atribuição |
| O(para linha4...linha6) – O(linha4) . (O(linha5) + O(linha6)) ⇔ O(linha4) = O(n) – A repetição é executada n vezes ⇔ O(linha5) + O(linha6) ⇔ O(1) + O(1) ⇔ O(1) ⇔ O(n). |
| O(1) ⇔ O(n) – Estrutura de repetição executada n vezes com complexidade constante para cada |

repetição.

O(para linha7...linha9) – O(linha7) . O(linha8) . O(linha9) $\Leftrightarrow$ O(linha7) = O(n) $\Leftrightarrow$ O(linha8) = O(n) $\Leftrightarrow$ O(linha9) = O(1) – Atribuição complexidade constante $\Leftrightarrow$ O(n) . O(n) . O(1) $\Leftrightarrow$ O(n.n.1) $\Leftrightarrow$ $O(n^2)$

O(operacaomatriz) = O(linha3) + O(para linha4...linha6) + O(para linha7...linha9) $\Leftrightarrow$ O(1) + O(n) + $O(n^2)$ $\Leftrightarrow$ $O(1 + n + n^2)$ $\Leftrightarrow$ Considerando apenas a variável com maior ordem, o resultado é $O(n^2)$.

# FUNÇÕES IMPORTANTES NA COMPLEXIDADE

Após a análise do algoritmo, é importante entender as complexidades obtidas para se buscar algoritmos mais otimizados para o problema. Sempre considerar n como o tamanho das entradas.

## CONSTANTE: ≈ 1

O algoritmo, independentemente do tamanho de n, operações executadas um número fixo de vezes.

## LOGARÍTMICA: ≈ $LOG_B$ N

Essa complexidade, segundo Cormen (2012), é típica de algoritmos que resolvem um problema transformando-o em problemas menores (dividir para conquistar). Para dobrar $\log_2$ n é preciso fazer $\log_2 n^2$. A base também muda pouco os valores: $\log_2$ n = 20 e $\log_{10}$ n = 6 para n = 1.000.000.

## *LINEAR*: ≈ *N*

Algoritmos com essa complexidade, em geral, executam uma certa quantidade de operações sobre cada um dos elementos de entrada. A melhor situação para quando é preciso processar n elementos de entrada e obter n elementos de saída.

## LOG LINEAR (OU N-LOG-N): N.$LOG_B$ N

Essa complexidade é comum em algoritmos que resolvem um problema transformando-o em problemas menores, resolvem cada um de forma independente e depois juntam as soluções.

## QUADRÁTICA: ≈ $N^2$

Essa complexidade ocorre frequentemente quando os dados são processados aos pares, com laços de repetição aninhados. Sempre que n dobra, o tempo de execução é multiplicado por 4. Esses algoritmos podem ser úteis para resolver problemas de tamanho relativamente pequeno.

## CÚBICA ≈ $N^3$

Ocorre em multiplicações de matrizes, com três estruturas de repetição aninhadas. Sempre que n dobra, o tempo de execução é multiplicado por 8. Podem ser úteis para resolver problemas de tamanho relativamente pequeno (ou quando não se tem outra opção!).

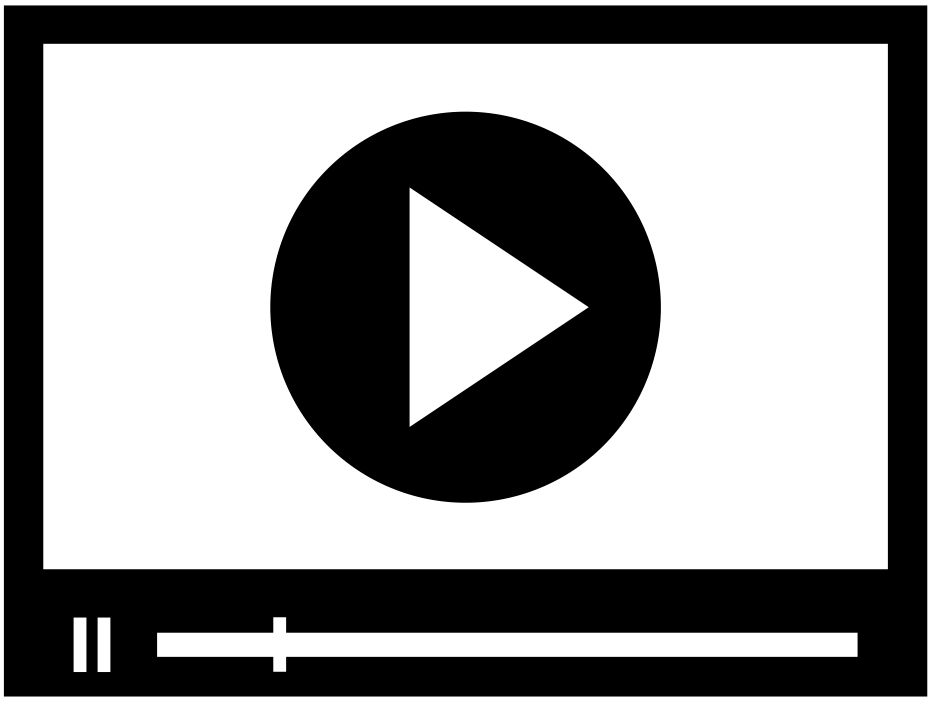
## *EXPONENCIAL*: ≈ $A^N$

Essa complexidade geralmente ocorre quando se usa um algoritmo como uma solução de força bruta.

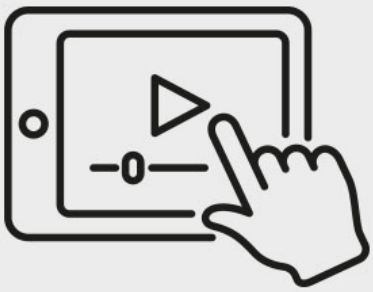
Para o caso $2^n$, sempre que n dobra, o tempo de execução é elevado ao quadrado. Esses algoritmos não têm utilidade do ponto de vista prático.

## *FATORIAL*: ≈ *N*!

Normalmente, é apresentado como complexidade exponencial, apesar de o fatorial ter comportamento muito pior. Geralmente, ocorre quando se usa uma solução de força bruta. Para n = 20, o fatorial de n! é igual $2.4^{1018}$, para o dobro n = 40, o fatorial de n! é igual $8.2^{1047}$. Definitivamente, não são úteis do ponto de vista prático.



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# VERIFICANDO O APRENDIZADO

**1. NUMA COMPETIÇÃO DE PROGRAMAÇÃO, GANHAVA MAIS PONTOS O TIME QUE APRESENTASSE O ALGORITMO MAIS EFICIENTE PARA RESOLVER O PIOR CASO DE UM DETERMINADO PROBLEMA. A COMPLEXIDADE ASSINTÓTICA (NOTAÇÃO BIG O) DOS ALGORITMOS ELABORADOS ESTÁ ILUSTRADA NA TABELA ABAIXO. ESCOLHA A ALTERNATIVA QUE INDIQUE A COLOCAÇÃO DOS TIMES NA COMPETIÇÃO:**

| TIME | COMPETIÇÃO |
| --- | --- |

| | |
|---|---|
| **BRANCO** | **$O(N^{20})$** |
| **AMARELO** | **O(NLOGN)** |
| **AZUL** | **O(1)** |
| **VERDE** | **O(N!)** |
| **VERMELHO** | **$O(2^N)$** |

**ATENÇÃO! PARA VISUALIZAÇÃOCOMPLETA DA TABELA UTILIZE A ROLAGEM HORIZONTAL**

**A)** Amarelo, Azul, Branco, Vermelho, Verde

**B)** Vermelho, Verde, Branco, Amarelo

**C)** Azul, Amarelo, Vermelho, Branco, Verde

**D)** Azul, Amarelo, Branco, Vermelho, Verde

**2. GERE A COMPLEXIDADE PESSIMISTA DO ALGORITMO DE ORDENAÇÃO ABAIXO:**

```
ORDENA(INT VET[], INT N){
(1) INT I, POS, AUX;
(2) PARA (I = 0;I < N; I++) {
(3) POS = I;
(4) PARA (J = I + 1;J < N;J++ )
(5) SE (VET [POS] > VET [J])
(6) POS = J;
```

**(7) SE (POS <> I) {**

**(8) AUX = VET[I];**

**(9) VET[I] = VET[POS];**

**(10) VET[POS] = AUX;**

**}**

**}**

**}**

**A)** O(n)

**B)** $O(n^2)$

**C)** $O(n^3)$

**D)** O(nlogn)

---

## GABARITO

**1. Numa competição de programação, ganhava mais pontos o time que apresentasse o algoritmo mais eficiente para resolver o pior caso de um determinado problema. A complexidade assintótica (notação Big O) dos algoritmos elaborados está ilustrada na tabela abaixo. Escolha a alternativa que indique a colocação dos times na competição:**

| Time | Competição |
|---|---|
| **Branco** | **$O(n^{20})$** |
| **Amarelo** | **O(nlogn)** |
| **Azul** | **O(1)** |
| **Verde** | **O(n!)** |

| | |
|---|---|
| **Vermelho** | **$O(2^n)$** |

**Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

A alternativa **"D "** está correta.

A questão aborda um conceito importante da ordem da complexidade assintótica das funções. Pelo resultado, as complexidades assintóticas funções ordenadas ficaram da seguinte forma: O(1) < O(nlogn) < O(n20) < O(2n) < O(n!).

**2. Gere a complexidade pessimista do algoritmo de ordenação abaixo:**

```
Ordena(int vet[], int n){
(1) int i, pos, aux;
(2) para (i = 0;i < n; i++) {
(3) pos = i;
(4) para (j = i + 1;j < n;j++ )
(5) se (vet [pos] > vet [j])
(6) pos = j;
(7) se (pos <> i) {
(8) aux = vet[i];
(9) vet[i] = vet[pos];
(10) vet[pos] = aux;
}
}
}
```

A alternativa **"B "** está correta.

A questão 2 aborda o cálculo da complexidade pessimista, Big O, para um algoritmo de ordenação. São duas repetições aninhadas nas linhas (2) e linha (4), gerando uma complexidade $O(n^2)$.

# CONCLUSÃO

# CONSIDERAÇÕES FINAIS

Os algoritmos são considerados a estrutura base para o aprendizado do desenvolvimento de software. Um bom desenvolvedor conhece bem algoritmos e pode traduzir para qualquer linguagem de programação.

As estruturas de dados apresentadas são importantes para resolver uma grande parte de problemas reais, além de terem fortes aplicações para a resolução de problemas matemáticos. São estruturas de fácil aprendizado e de implementação, tornando-se uma ferramenta poderosa para a solução de problemas computacionais.

A análise de algoritmos é uma área largamente utilizada para o apoio a soluções de problemas computacionalmente complexos, como escolha da melhor rota entre diversos pontos, menor caminho entre dois pontos em um mapa, busca de soluções para equações, ordenação de grandes conjuntos de elementos, entre outros.

A partir da análise de algoritmos, pode-se chegar, em um tempo adequado, à solução ótima para o problema, considerando parâmetros como tempo e espaço de memória consumidos. Importante ressaltar que a análise de algoritmos objetiva estudar a solução de um problema para quantidade muita alta de elementos de entrada. Como, por exemplo, um algoritmo para escolher a melhor rota para um ônibus, considerando dez bairros com mais de trinta ruas em cada bairro de uma metrópole.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.

# REFERÊNCIAS

Ascencio, A. F. G.; Campos, E. A. V. de. **Fundamentos da Programação de Computadores**: Algoritmos, Pascal, C, C++. 3. ed. São Paulo: Pearson, 2012.

Barrere, E. **Análise e Projeto de Algoritmos,** Notas de Aula. *In*: UFJF. Consultado em meio eletrônico em: 24 set. 2020.

Cormen, T. H. *et al*. **Algoritmos**: Teoria e Prática. 3. ed. São Paulo: LTC, 2017, cap. 1 – 3.

Enrooth, E. **How often does Google update search results?**. *In*: HDWebPros. Consultado em meio eletrônico em: 24 set. 2020.

Feofiloff, P. **Análise de Algoritmos**. *In*: IME. Consultado em meio eletrônico em: 24 set. 2020.

Forbellone, A. L; Eberspacher, H. **Lógica de Programação**: A construção de Algoritmos e Estruturas de Dados. 3. ed. São Paulo: Pearson, 2005.

Manzano, J. A. N. G.; Oliveira, J. F. de. **Algoritmos** – Lógica para Desenvolvimento de Programação de Computadores. 28. ed. São Paulo: Érica, 2016.

Ponti, M. **Análise de Algoritmos**, Notas de Aula. *In*: ICMC.USP. Consultado em meio eletrônico em: 24 set. 2020.

QCONCURSOS. **Questões de Concursos**. *In*: QConcursos. Consultado em meio eletrônico em: 24 set. 2020.

Toscani, L.; Veloso, P. *et al*. **Complexidade de Algoritmos**. (Coleção Livros Didáticos UFRGS, v. 13). 3. ed. São Paulo: Bookman, 2012, cap. 1 – 3.

---

# EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, leia:

Sobre algoritmos aplicados a uma linguagem de programação, Projeto de Algoritmos com Implementações em Java e C++, na página DCC.UFMG.

Sobre o cálculo de complexidade dos algoritmos, listas de exercícios dos livros Algoritmos: Teoria e Prática, de Cormen *et al*.  e Complexidade de Algoritmos, de Toscani e Veloso.

Sobre pior caso, melhor caso e caso médio, os livros de Cormen *et al*.  e Toscani e Veloso utilizados neste tema.

---

# CONTEUDISTA

Marcelo Nascimento Costa

# 🔗 CURRÍCULO LATTES