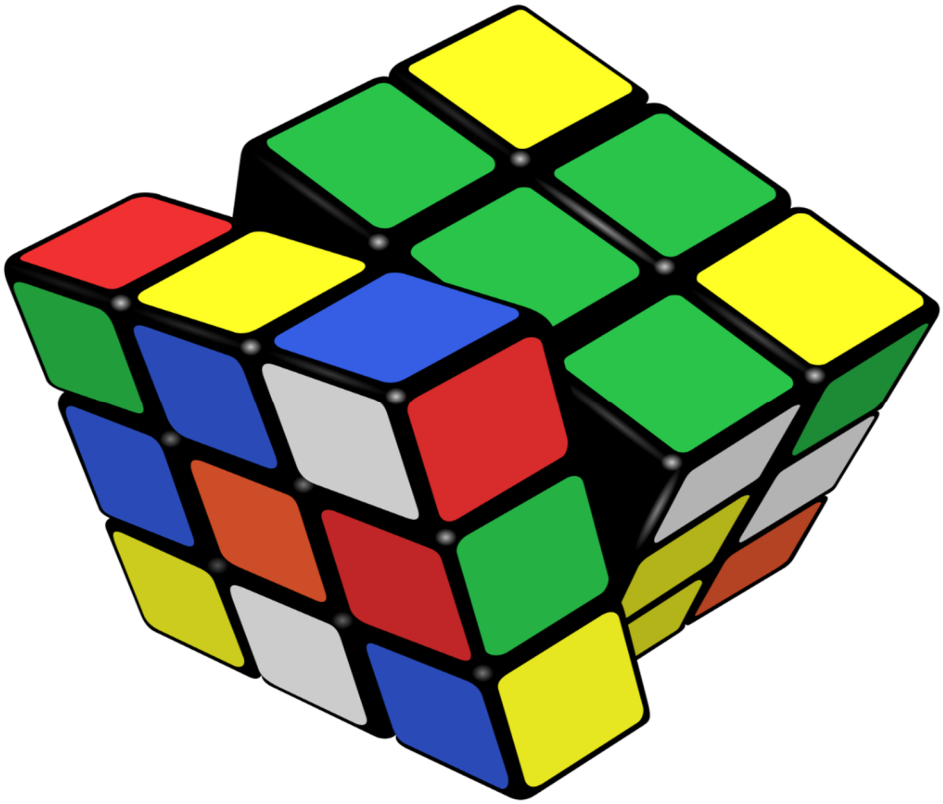
ESTRUTURAS DE DADOS



FABRÍCIO GALENDE MARQUES DE CARVALHO

Sumário

PREFÁCIO 2
1. INTRODUÇÃO 2
1.1. OBJETIVOS DA UNIDADE 2
1.2. REVISÃO GERAL DE ALGORITMOS 2
1.3. COMO OS ALGORITMOS PODEM SER REPRESENTADOS 3
1.4. BLOCOS DE CONSTRUÇÃO DE ALGORITMOS 4
1.4.1. COMENTÁRIOS 4
1.4.2. DECLARAÇÃO DE VARIÁVEIS E TIPOS 5
1.4.3. BLOCOS DE CONTROLE DE FLUXO (DECISÃO) 6
1.4.4. BLOCOS DE REPETIÇÃO (ITERAÇÃO) 7
1.4.5. FUNÇÕES 8
1.4.6. CLASSES, MÉTODOS E OBJETOS 10
1.5. EQUIVALÊNCIA DE ALGORITMOS 11
EXERCÍCIOS E PROBLEMAS 12
2. ALGORITMOS RECURSIVOS 16
2.1. INTRODUÇÃO 16
2.2. RECURSÃO 17
2.3. CUIDADOS AO SE UTILIZAR UM ALGORITMO RECURSIVO 19
EXERCÍCIOS E PROBLEMAS 21
REFERÊNCIAS BIBLIOGRÁFICAS 24
ANEXO I 25

PREFÁCIO

Esse material tem como objetivo fornecer ao estudante dos cursos superiores em computação uma visão geral de algoritmos e estruturas de dados.

Ao final de cada unidade são apresentados exercícios e problemas (prefixo P) que focam na aplicação prática dos fundamentos estudados na unidade. Exercícios de terminologia e conceitos (TC) são também propostos para fomentar a fixação dos termos técnicos e conceitos previamente estudados.

Apesar de ser um curso introdutório, esse material pressupõe que o estudante cursou prelo menos um semestre de alguma disciplina básica relacionada à programação e algoritmos.

1. INTRODUÇÃO

1.1. OBJETIVOS DA UNIDADE

- ✓ Revisar os conceitos básicos relacionados à construção de algoritmos e programas elementares
- ✓ Ilustrar equivalências entre algoritmos diferentes em termos de entrada e saída.

1.2. REVISÃO GERAL DE ALGORITMOS

O QUE SÃO ALGORITMOS?

Algoritmos correspondem a um conjunto de passos a serem efetuados para a resolução de um problema.

Os passos, ou etapas, definidos em um algoritmo devem ser finitos, ou seja, o algoritmo possui início e fim bem definidos.

Esquematicamente, um algoritmo pode ser considerado como um bloco funcional que transforma um conjunto de entradas em um conjunto de saídas.

A entrada são os dados disponíveis para o problema e a saída corresponde à solução.

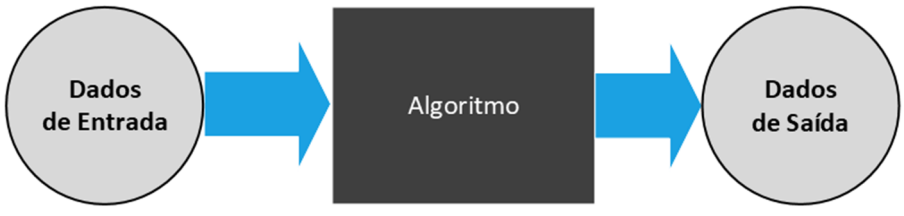


Figura 1. Diagrama de blocos de um algoritmo.

1.3. COMO OS ALGORITMOS PODEM SER REPRESENTADOS

Algoritmos podem ser representados de diferentes formas. Algumas delas incluem:

- ☑ Linguagem natural.
- ☑ Pseudocódigo.
- ☑ Diagramas ou esquemas visuais.
- ☑ Utilização de linguagens de programação.

Exemplos:

a) Linguagem natural:

```
1. Ler dois números
2. Calcular a soma desses números
3. Retornar a soma obtida.
```

b) Pseudocódigo:

```
ler("Primeiro número:")
ler(numero_1: inteiro)
escrever("Segundo número")
ler(numero_2: inteiro)
resultado: inteiro ← numero_1 + numero_2
escrever(resultado)
```

c) Diagramas ou esquemas visuais

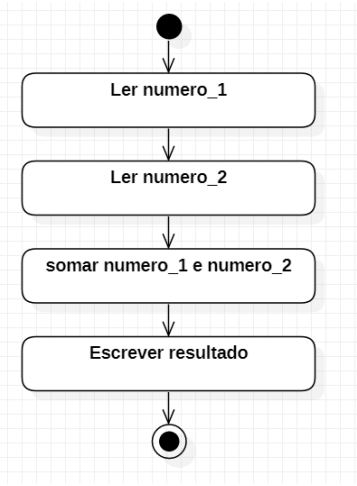


Figura 2. Diagrama de atividades representativo de um algoritmo.

d) Utilização de linguagens de programação

```
import * as prompt_sync from 'prompt-sync';
var numero_1: number;
var numero_2: number;
var resultado: number;

const prompt = prompt_sync();
numero_1 = parseFloat(prompt("Primeiro número: "));
numero_2 = parseFloat(prompt("Segundo número: "));
resultado = numero_1 + numero_2;
console.log(resultado);
```

Independentemente da maneira como um algoritmo é representado é importante que haja consistência e uniformidade de modo a deixar claro quais passos são executados, sua ordem e demais aspectos relevantes. Esse aspecto deve depender o mínimo possível de notações ou representações que não sejam de comum conhecimento dos desenvolvedores envolvidos no projeto.

1.4. BLOCOS DE CONSTRUÇÃO DE ALGORITMOS

De um modo geral, algoritmos podem ser visualizados como sendo um agrupamento de blocos que efetuam operações específicas desde o início da resolução de um problema (ex. leitura dos dados), até a obtenção da resposta final (solução).

Esses blocos possuem finalidades específicas e podem ser descritos tal como feito a seguir.

1.4.1. COMENTÁRIOS

Servem para esclarecer aspectos importantes de um algoritmo.

Comentários em geral são dispensáveis quando o algoritmo é escrito em linguagem natural ou pseudocódigo.

Caso sejam utilizadas linguagens de programação, são ignorados pelo compilador, sendo tipicamente substituídos por espaços em branco (ausência de instruções) durante o processo de compilação.

Exemplo: Código-fonte, em TypeScript, que exemplifica comentários.

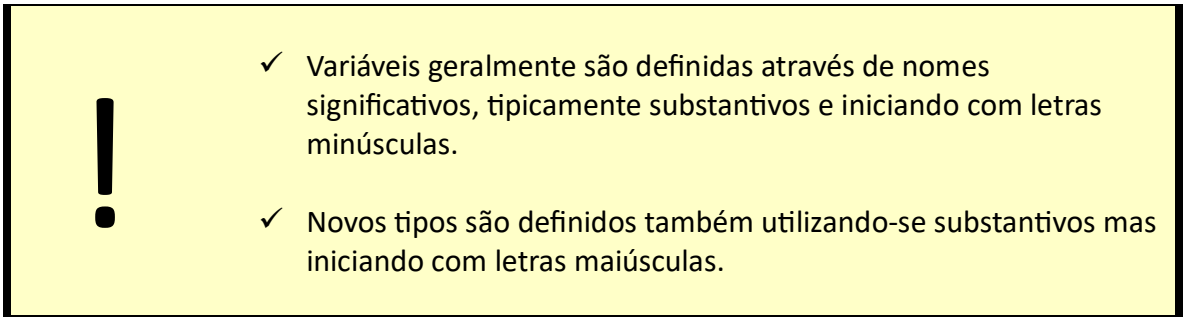
```
/* Exemplo de comentário em várias linhas
**
** Autor: Fabrício G. M. de Carvalho
*/

var a: number = 10; // um número qualquer.
console.log(a);
```

1.4.2. DECLARAÇÃO DE VARIÁVEIS E TIPOS

Declaram nomes que posteriormente serão associados a valores (dados) operados pelos algoritmos.

São essas declarações são conhecidas como ***name bindings*** (vínculos a nomes).

Quando uma declaração renomeia um tipo já existente, dizemos que ocorre um ***alias*** (apelido).



- ✓ Variáveis geralmente são definidas através de nomes significativos, tipicamente substantivos e iniciando com letras minúsculas.
- ✓ Novos tipos são definidos também utilizando-se substantivos mas iniciando com letras maiúsculas.

Exemplo: Pseudocódigo para declaração de variáveis e tipos.

```
a: inteiro ← 10
b: Array[1..10] de caracteres
b[0]← 'x'
Tipo Cachorro:
     porte: cadeia de caracteres
     pelagem: cadeia de caracteres
     raca: cadeia de caracteres
fim-tipo
c: Cachorro
c.porte ← 'pequeno'
```

Exemplo: Código-fonte, em TypeScript, contendo declaração de variáveis e definição de tipos.

```typescript
/* tipo lógico */
var logico: boolean;
logico = true;
console.log(logico);

/* definindo um novo tipo: */
type Cachorro = {
    raca: string;
    porte: string;
    pelagem: string;
}
```

```typescript
var cachorro_1: Cachorro = {
    porte: "pequeno",
    pelagem: "longa",
    raca: "Shih-Tzu"
}
console.log(typeof(cachorro_1));
console.log(cachorro_1.raca);
```

1.4.3. BLOCOS DE CONTROLE DE FLUXO (DECISÃO)

São blocos que permitem que determinados passos do algoritmo sejam executados de modo condicional, ou seja, caso uma determinada condição seja satisfeita.

A condição sempre deve ser algo que possa ser expressa como verdadeira ou falsa.

Exemplo: Pseudocódigo do bloco Se-então-Senão.

```
a: inteiro
b: inteiro ← 1
Se b > 0 então
      a ← 1
Senão
      a ← 0
Fim-se
imprimir(a)
```

Exemplo: Código-fonte, em TypeScript, do bloco if-else.

```
/* bloco if-else */
let a: number;
let b: number = 1;
if (b > 0 ){
    a = 1;
} else {
    a = 0;
}
console.log(a);
```

Notar que b > 0 é uma expressão que pode ser avaliada como verdadeira ou falsa, dependendo do valor de b.

Exemplo: Pseudocódigo do bloco Selecione-Caso(s).

```
c:inteiro ← 10
Selecione valor de c:
      caso 10:
            Escrever("c é igual a 10")
      caso 11:
            Escrever("c é igual a 11")
      outros casos:
            Escrever("c possui outro valor")
Fim-selecione
```

Exemplo: Código-fonte, em TypeScript, do bloco switch-case.

```
let c: number = 10;
switch (c) {
    case 10:
        console.log("c é igual a 10");
        break;
    case 11:
        console.log("c é igual a 11");
        break;
    default:
        console.log("c possui outro valor");
}
```

Notar que, diferentemente do bloco if, o switch opera sobre valores que não necessitam ser verdadeiro ou falso somente. Tipicamente esse bloco de controle, nas linguagens de programação de alto nível, opera sobre valores que são inteiros ou tipos que podem ser representados por inteiros (tais como caracteres ou cadeias de caracteres).

1.4.4. BLOCOS DE REPETIÇÃO (ITERAÇÃO)

São blocos que são executados de modo repetido dependendo do fato de uma determinada condição ser ou não satisfeita.

Dependendo do tipo de bloco e da condição, pode ser executado uma, muitas ou nenhuma vez.

Exemplo: Pseudocódigo do bloco Enquanto ... Faça.

```
a : inteiro ← 1
Enquanto (a < 10)
      imprimir("Valor de a: ", a)
      a ← a+1
Fim-enquanto
```

Exemplo: Código-fonte, em TypeScript, do bloco while.

```
var a: number = 1;
while(a < 10){
    console.log("valor de a: ", a);
    ++ a;
}
```

Exemplo: Pseudocódigo do bloco Faça ... Enquanto.

```
b : inteiro ← 1
Faça
      imprimir("Valor de b: ", b)
      b ← b+1
Enquanto ( b < 10)
```

Exemplo: Código-fonte, em TypeScript, do bloco do-while.

```typescript
var b: number = 1;
do{
    console.log("valor de :b", b);
    b = b + 1;
} while( b< 10)
```

Exemplo: Pseudocódigo do bloco para.

```
c:inteiro
Para c ← 1, 2 .. 9
 imprimir("Valor de b: ",b)
Fim-para
```

Exemplo: Código-fonte, em TypeScript, para bloco for.

```typescript
for(let c: number = 1; c< 10; ++c){
    console.log("valor de c: ", c);
}
```

1.4.5. FUNÇÕES

Funções são utilizadas na definição de conjuntos de passos ou etapas que podem se repetir em diferentes pontos de um algoritmo. Ou seja, sempre que os mesmos passos são executados (potencialmente com dados diferentes) em diversos pontos de um algoritmo é conveniente a definição de uma função.

Uma função recebe valores através de seus **parâmetros formais.** Diz-se então que a função é um bloco parametrizado.

Os valores que substituem os parâmetros, durante a chamada da função, chamam-se **argumentos**.

Uma função define um **escopo**, que pode ser entendido como um limitador de até onde certas variáveis podem ser acessadas.

Toda função pode ser associada a um **tipo de dado**. Por definição, o tipo associado a uma função em geral corresponde ao **tipo de dado que ela retorna**, ou seja, o valor que "sai da função".

O nome dado à função é seu **identificador**.

Esquematicamente, uma função pode ser representada pelo seguinte pseudocódigo:

```
identificador(parâmetros formais): tipo retornado
      passo 1
      passo 2
      passo 3
      retorne valor_retornado
Fim-função
```

O identificador de uma função, bem como seus parâmetros formais e seu tipo retornado são denominados de **interface da função** ou sua **assinatura** (***function interface / function signature***).

Em documentações de sistemas de software, as descrições de interfaces de funções são conhecidas como **interface de programação de aplicações** (API – Application Programming Interface(s)). Esse mesmo termo é empregado para descrever funcionalidades de serviços da internet (*web services*).

Exemplo: Pseudocódigo de uma função que soma dois números.

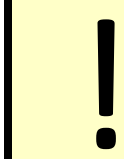
```
somar(a:inteiro, b:inteiro): inteiro
      resultado: inteiro
      resultado ← a + b
      retorne resultado
Fim-função
```

Exemplo: Código-fonte, em TypeScript, de uma definição de função que soma dois números.

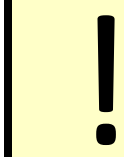
```typescript
function somar(a: number, b:number){
    let resultado: number = a + b;
    return resultado;
}
```

Exemplo: Código-fonte, em TypeScript, que **invoca** a função que soma dois números.

```
var x: number = 1;
var y: number = 2;
var z: number;
z = somar(x,y);
console.log("z vale: ", z);
```



- ✓ Funções são um bloco construtor de algoritmos que ajudam a torná-los modulares e com baixo acoplamento.
- ✓ Um código-fonte não acoplado é um código que pode ser reutilizado e movido sem causar "efeitos colaterais" em outras partes do programa/sistema de software.



- ✓ Uma vez que funções representam operações, é usual que sejam nomeadas utilizando-se verbos. Exemplos frequentes de identificadores de função incluem: ler, escrever, salvar, buscar, apagar, mover, alterar, ordenar, etc. (em inglês: ***read, write, save, search, delete, move, update, sort, etc.***).

1.4.6. CLASSES, MÉTODOS E OBJETOS

Quando se define um tipo que possui **propriedades**, que podem assumir diferentes **valores** e, além disso, outras que são **funções** e que geralmente podem operar sobre tais valores, tem-se aquilo que se chama de **classe**.

Uma classe modela "coisas" do mundo real em termos de **propriedades/atributos** e **operações/funções**.

Operações/funções atreladas a objetos são tipicamente denominadas de **métodos**.

Exemplo: Um modelo de classe para um cachorro:

```
Classe Cachorro:
     raça: cadeia de caracteres
     latir: função
Fim-classe
```

Exemplo: Um modelo de classe para um cachorro em TypeScript:

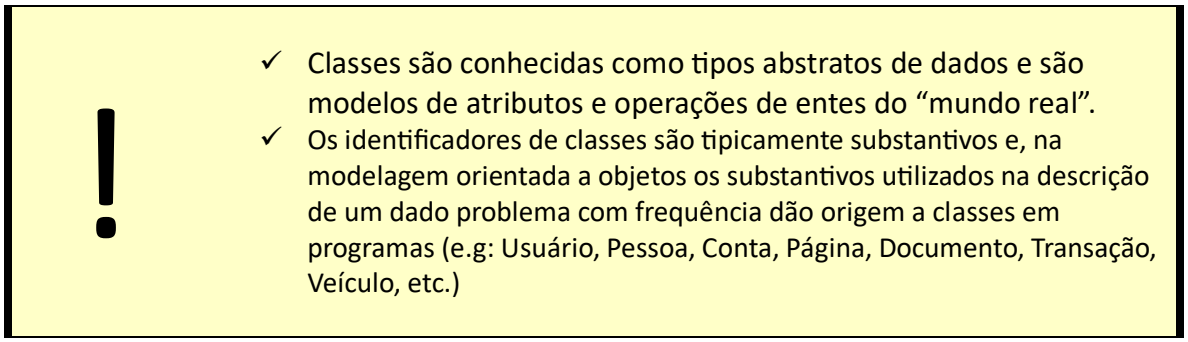
```
class Cachorro{
    raca: string;
    constructor(raca: string){
        this.raca = raca;
    }
    latir(): string{
        return "au au au";
    }
}
```

Quando uma classe é utilizada para criar uma variável e é utilizada em um programa, seja para acessar suas propriedades ou se executar suas operações, tem-se uma **instância de classe** ou **objeto**.

Exemplo: Instanciando um objeto da classe Cachorro, em TypeScript.

```
var cachorro_1 = new Cachorro("Lhasa");
console.log(cachorro_1.raca);
console.log(cachorro_1.latir());
```

Em linguagens de programação que dão suporte à orientação a objetos (ou seja, que permitem a definição e a instanciação de classes), é comum que haja uma operação que sempre é chamada assim que um objeto é instanciado e passe a ser vinculado a uma variável. Essa operação é denominada de **construtor** e geralmente possui uma regra típica de definição ou palavra reservada da linguagem de programação.



1.5. EQUIVALÊNCIA DE ALGORITMOS

A rigor, algoritmos equivalentes seriam aqueles que executam os mesmos passos, na mesma ordem, desde a entrada dos dados até a obtenção da solução.

Se considerarmos a equivalência sob o ponto de vista somente da correspondência entrada/saída, podemos dizer, até certo ponto, que algoritmos equivalentes seriam aqueles que conseguem obter as mesmas correspondências entrada/saída considerando as mesmas entradas.

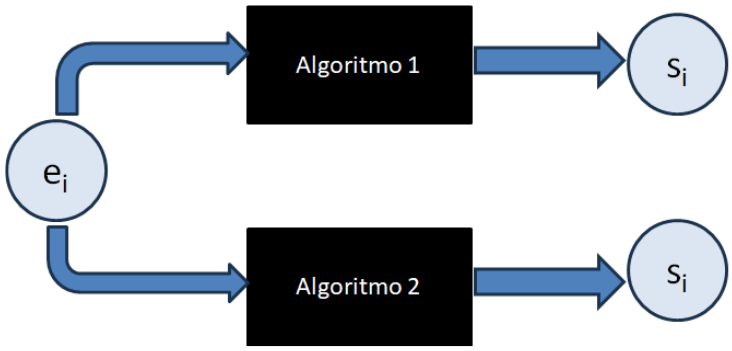


Figura 3. Algoritmos equivalentes em termos de entrada e saída.

Essa noção de equivalência é importante para que algoritmos desenvolvidos possam ser analisados, avaliados e melhorados, de acordo com um certo objetivo (e.g: melhoria de desempenho, economia de memória, melhor legibilidade, etc.)

Exemplo: Algoritmos equivalentes em termos de entradas e saídas.

```
x: inteiro ← 1
Se x =1 então
    escrever("x é igual a 1")
Senão Se x=2 então
          escrever("x é igual a 2")
     Senão
          escrever("x possui outro valor")
     Fim-se
Fim-se
```

```
x:inteiro ← 1
Selecione valor de c:
     caso 1:
          escrever("x é igual a 1")
     caso 2:
          escrever("x é igual a 2")
     outros casos:
          escrever("x possui outro valor")
Fim-selecione
```

EXERCÍCIOS E PROBLEMAS

PRÁTICA DE ALGORITMOS E PROGRAMAÇÃO

P.1.1. Crie um programa em typescript e defina:
a) Uma variável que tenha anotação explícita de tipo numérico.
b) Uma variável que tenha anotação explícita de tipo string.

c) Uma variável que não tenha anotação explícita de tipo mas que receba um valor lógico.
d) Uma variável que seja um JSON contendo propriedades de tipo numérico e de tipo string.
e) Um array numérico com anotação explícita de tipo.

Execute cada um desses programas e mostre na saída os valores e os tipos utilizando a função **`typeof()`**

P.1.2. Para os itens do exercício **P.1.1**, qual a diferença com relação à exibição dos tipos quando você passa o cursor sobre a variável durante a sua declaração?

P.1.3 Escreva um programa, em TypeScript, que solicite que o usuário digite dois números e imprima o maior deles.

P.1.4. Escreva um programa, em TypeScript, que solicite que o usuário digite duas letras e diga qual delas vem antes e qual vem depois no alfabeto.

P.1.5. Escreva um programa, em TypeScript, que solicite que o usuário digite duas palavras e diga qual delas aparece antes da outra no dicionário. O programa não deve solicitar nenhuma informação adicional por parte do usuário e supõe que as palavras são escritas somente com caracteres de 'a' a 'z'. Para esse caso, especifique, também, o algoritmo em pseudocódigo, conforme notação explicada em sala de aula. Utilize o método de string charCodeAt().

P.1.6. Repita o exercício P.1.5 mas utilizado operadores relacionais ao invés do método charCodeAt().

P.1.7. Escreva um programa que exiba um menu contendo 3 alternativas: "1. Dúvidas", "2.Reclamações", "3.Sair". O usuário deve digitar a palavra correspondente à opção do menu e, dependendo da opção, deve ser fornecida uma orientação ao usuário. Exemplo: Caso o usuário digite "Dúvidas" exiba: "Suas dúvidas devem ser encaminhadas para o email duvidas@email.com". Esse programa deve ser escrito em TypeScript e deve fazer uso do bloco switch.

P.1.8. Escreva um programa, utilizando o bloco for, que imprima todos os múltiplos de 3 contidos entre 0 e 100, inclusive. Os valores devem ser impressos em ordem crescente.

P.1.9. Repita o exercício P.1.8 mas utilizando um bloco while, imprimindo os números em ordem decrescente.

P.1.10. Repita o exercício P.1.9 utilizando um bloco do while.

P1.11. Escreva dois programas equivalentes, em TypeScript, que solicitem que o usuário digite dois números, n1 e n2. O programa deve dizer se o n1 é "menor ou igual a n2" ou se é "maior do que n2". Um programa deve utilizar o bloco switch e o outro deve utilizar if-else. Qual dificuldade surgiria caso se desejasse que o programa discriminasse 3 condições (>, < ou =)?

P.1.12. Repita o exercício P.1.11 utilizando uma chamada à função e uma estrutura if de comparação interna entre os números. A entrada para a função deve ser os números e a saída deve ser uma string contendo uma mensagem informativa da relação entre esses números (>, <

ou =). Assegure-se de que a interface da função não é quebrada e de que o código é devidamente modularizado. A função deve ser definida em um arquivo e invocada em outro (use export e import).

P.1.13. Modele, utilizando orientação a objetos, um usuário de um sistema que tenha preenchido as seguintes informações em uma interface de cadastro: nome, ano de nascimento, cpf e gênero. Esse usuário deve possuir um método chamado *equals*, que compara uma instância da classe com outra passada como argumento para o método *equals* e outro método chamado *speak_name* que retorna a string representativa do nome do usuário. Demonstre a execução de um programa que faça uso dessa classe, exibindo resultados no console.

P.1.14. Demonstre como uma variável declarada com escopo de bloco pode levar a uma obtenção errada de resultado quando existe uma função ou método que faz uso de um valor armazenado em uma variável global. Ilustre isso com uma função simples e diga como esse tipo de construção de programa viola um bom projeto de função. Como esse tipo de problema poderia ser resolvido?

P.1.15. Crie um programa que contenha o seguinte: Num arquivo que contenha modelo, defina uma variável do tipo cadeia de caracteres, uma variável do tipo numérico, uma variável do tipo array de cadeia de caracteres, uma variável do tipo array de números, um objeto que modele uma pessoa (que tenha nome e idade). Nesse mesmo arquivo, inicialize valores específicos para cada uma dessas variáveis e exporte essa variáveis para outro arquivo de mesmo nome mas com sufixo _view. Crie uma função que receba cada uma dessas variáveis como argumentos (parâmetros formais da função) e, internamente, altere o valor de cada um dos parâmetros passados. Imprima os valores das variáveis, no arquivo _view, antes e depois de chamar a função. Qual sua conclusão a respeito? OBS: não viole as interfaces de entrada e saída da função, conforme instruído em sala de aula.

P.1.16. Considere a seguinte situação: Criação de um menu de opções numérico, onde as opções são as seguintes: 1 – Criar um cadastro; 2 – Excluir um Cadastro; 3- Atualizar um Cadastro; 4 – Listar Cadastros. Represente esse menu de opções, desde o momento em que o sistema solicita a opção para o usuário, a leitura da opção e a impressão de uma mensagem informando qual opção o usuário selecionou. Para a seleção deve ser utilizado um bloco do tipo Selecione-Caso. Utilize para sua representação:

a) Linguagem Natural
b) Pseudocódigo
c) Diagrama de atividades da UML.

Implemente o menu da situação do problema utilizando a Linguagem de Programação TypeScript.

TERMINOLOGIA E CONCEITOS

TC.1.1. Complete os seguintes parágrafos com termos utilizados na área de algoritmos e estruturas de dados.

Um algoritmo corresponde a um conjunto de _____(1)_____ a serem efetuados para a resolução de um problema. Os _____(2)_____ ou _____(3)_____ definidos em um algoritmo devem ser _____(4)_____, ou seja, o algoritmo deve possuir início e fim bem

definidos. Algoritmos podem ser vistos também como _____(5)_____ funcionais que transformam um conjunto de _____(6)_____ em um conjunto de _____(7)_____.

Os blocos fundamentais que podem constituir um algoritmo incluem tipicamente _____(8)_____, _____(9)_____, _____(10)_____, _____(11)_____, _____(12)_____ e _____(13)_____.

Sob o ponto de vista de entrada e saída, pode-se dizer que algoritmos _____(14)_____ são aqueles que obtém uma mesma correspondência _____(15)_____considerando as mesmas _____(16)_____.

TC.1.2. Utilizando diagramas de atividades da UML, represente os seguintes blocos de programas:

a) Uma estrutura seleção do tipo selecione-caso contendo 3 opções, onde cada opção, ao ser ativada, é executada sem que as opções seguintes sejam acionadas.
b) Uma estrutura de seleção do tipo selecione-caso contendo 3 opções, onde cada opção, ao ser ativada, é executada e as demais opções são também executadas (isso corresponde à ausência do *break* ao final do caso.
c) Uma estrutura de seleção contendo 3 if-else aninhados.
d) Uma estrutura de seleção contendo 3 ifs independentes, sequenciais.
e) Uma estrutura de repetição do tipo while.
f) Uma estrutura de repetição do tipo do-while.

TC.1.3. Qual a finalidade dos elementos fork e join na representação de um fluxo de programa? Dê um exemplo prático.

2. ALGORITMOS RECURSIVOS

Algoritmos recursivos são amplamente utilizados em computação. Muitos dos problemas resolvidos de modo computacional são evidentes quando encarados considerando-se o prisma da recursão e muito mais difíceis de serem resolvidos quando são considerados utilizando-se métodos iterativos.

Nessa seção os algoritmos recursivos são brevemente estudados levando-se em conta os seguintes objetivos de aprendizagem:

- ✓ Definir, de modo geral, um algoritmo recursivo
- ✓ Ilustrar a resolução de problemas computacionais através da utilização de algoritmos recursivos.
- ✓ Entender os impactos da utilização de um algoritmo recursivo no desempenho de um programa.
- ✓ Ilustrar a técnica de *memory caching*.

2.1. INTRODUÇÃO

No estudo de algoritmos recursivos é importante que o profissional seja capaz de fazer uma clara diferenciação entre **problema** e **instância de problema**, sob o ponto de vista computacional.

Um problema computacional corresponde a um **questionamento mais geral** e demanda uma resposta também geral.

A resposta a tal questionamento serve como **solução geral ao problema**.

Exemplo: Problema de determinação do maior valor entre dois números informados.

Nesse caso, a resposta correspondente será um algoritmo que determine, entre dois números arbitrários, qual deles é o maior.

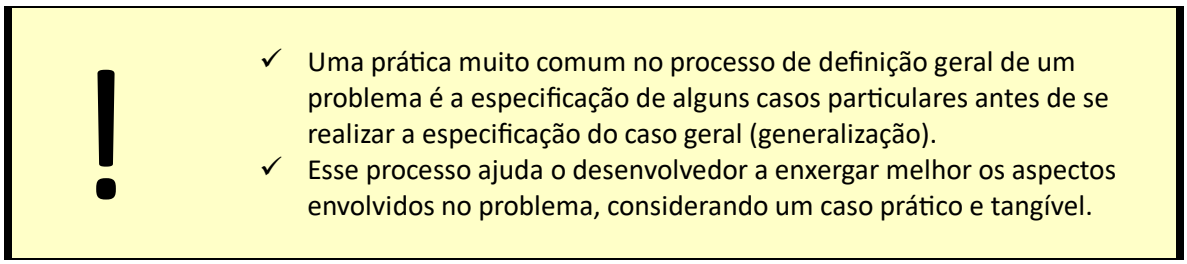
Note-se que, para este caso os números sequer são especificados, ou seja, não se sabe se são números positivos, negativos, se o zero está ou não incluído, se podem ser números contendo parte fracionária, etc.

Quando um problema, como o do exemplo anterior, é expresso considerando-se dados de entrada específicos, concretos (e.g.: Determinar qual dos números, entre 2 e 5, é o maior), tem-se aquilo que se chama de **instância do problema**.

Ou seja, uma instância de problema é um **caso particular do problema**.

O entendimento da diferença entre problema e instância de problema é fundamental para o entendimento da estratégia de resolução de problemas baseada em recursão. Cabe sempre ressaltar que a solução para uma instância particular de problema não necessariamente gerará uma solução geral ou conclusão para o problema. Um exemplo típico disso é a execução da divisão entre dois números. Por exemplo, considerando-se dois números inteiros, 4 e 2, é notório que o resultado da divisão nesse caso é exata e vale 2 (4/2 = 2). Um desenvolvedor desavisado poderia, por exemplo, afirmar que sempre a divisão de dois números inteiros resultaria em um

número inteiro, porém a simples modificação da instância do problema para uma divisão de 4 por 3 mostra que essa conclusão é falsa, pois 4/3 = 1,3333... . Não só a conclusão é falsa como o resultado da divisão dá origem a uma parte fracionária que se repete indefinidamente (uma dízima periódica) e que poderá ocasionar problemas de arredondamento.



- ✓ Uma prática muito comum no processo de definição geral de um problema é a especificação de alguns casos particulares antes de se realizar a especificação do caso geral (generalização).
- ✓ Esse processo ajuda o desenvolvedor a enxergar melhor os aspectos envolvidos no problema, considerando um caso prático e tangível.

Compreendida a diferença entre instância de problema e problema, pode-se passar à discussão de recursão e problemas recursivos.

2.2. RECURSÃO

Problemas recursivos são aqueles em que uma determinada instância do problema contém uma instância "menor" do mesmo problema.

Exemplo: Determinar o maior número contido em uma lista de números é o mesmo que determinar o maior número entre os maiores números de duas sublistas obtidas a partir da lista de números original.

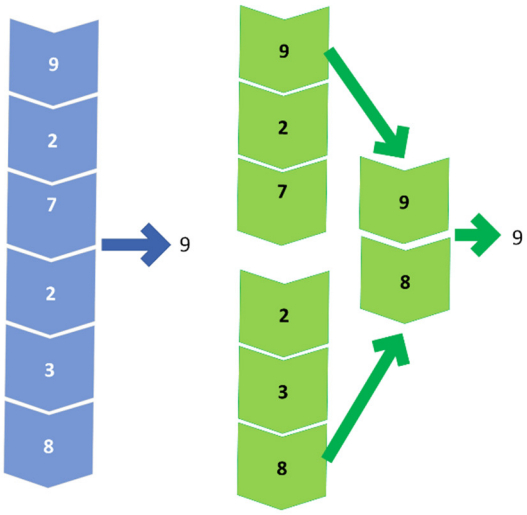


Figura 4. Determinação do menor número contido em uma lista de números. Duas abordagens, sem e com recursão.

Na imagem em azul, tem-se uma instância de problema resolvido de modo direto: maior elemento da lista com 6 elementos. Nesse caso, percorre-se a lista inteira e o resultado é que o número 9 é a solução encontrada.

Na imagem em verde, o mesmo problema foi quebrado em 2 subproblemas: maior elemento em uma lista contendo 3 elementos, seguido do maior elemento entre dois elementos resultantes da instância anterior (lista com 2 elementos). Nesse caso, o problema foi quebrado em instâncias menores com "tamanho" igual a "metade" do tamanho do problema original.

Note-se que essa foi uma escolha arbitrária para esse caso, pois o mesmo problema poderia ter sido desmembrado considerando-se uma lista contendo um único número e uma lista contendo os 5 números restantes.

Em essência, um problema recursivo é resolvido da seguinte forma:

```
Se a instância do problema é "pequena" Então
   Resolva o problema e retorne a solução
Senão
   Reduza a instância do problema
   Aplique a resolução à instância menor
   Volte à instância original (i.e., "maior") e
   combine a solução de modo a obter a solução
Fim-se
```

A instância "pequena" do problema é denominada de **caso base**. Tipicamente, o caso base é composto por um problema cuja solução é trivial ou direta.

Para o exemplo anterior de obtenção do menor número em uma lista de números, o caso base que poderia ser considerado seria uma lista contendo um único número.

Instâncias maiores do problema são compostas pelo **caso base combinado a outras instâncias do mesmo problema**.

A solução do problema será, portanto, uma combinação da solução de sucessivas reduções do problema até que o caso base seja atingido.

Exemplo: Determinação do maior elemento contido em um array.

- ✓ Identificação do Caso base (instância "pequena):

  Array contendo 1 único elemento.

- ✓ Identificação dos outros casos (instância "grande") :

  Array contendo 2 ou mais elementos

O pseudocódigo seguinte ilustra a definição de uma função que executa a determinação do menor elemento em um *array* utilizando uma abordagem recursiva.

```
Funcao maior_r(array: número): número
     Se tamanho(array) == 1 então
          retorne número
     Senão
          maior_restante = maior_r(array[1..N-1])
          Se array[0] >= maior_restante então
               retorne array[0]
          Senão
               retorne maior_restante
          Fim-se
     Fim-se
Fim-função
```

```
function maior_r(a: number[]): number{
    if (a.length == 1){
        console.log("Caso base atingido!")
        return a[0];
    } else {
        console.log("Chamada recursiva!");
        console.log("Invocando maior_r( ", a.slice(1, a.length), ");");
        let maior_restante = maior_r(a.slice(1, a.length))
        if (a[0] >= maior_restante){
            return a[0];
        }
        else{
            return maior_restante;
        }
    }
}
```

Figura 5. Código-fonte, em TypeScript, que implementa o algoritmo de determinação do maior número em um array utilizando recursão.

```
Array original:
[ 1, 4, 10, 20, -1 ]
Chamada recursiva!
Invocando maior_r(  [ 4, 10, 20, -1 ] );
Chamada recursiva!
Invocando maior_r(  [ 10, 20, -1 ] );
Chamada recursiva!
Invocando maior_r(  [ 20, -1 ] );
Chamada recursiva!
Invocando maior_r(  [ -1 ] );
Caso base atingido!
O maior número do array é:  20
```

Figura 6. Saída de execução do programa que implementa a determinação do maior elemento em um array de modo recursivo

Note-se que uma função ou método recursivo é caracterizado por:

- ✓ Possuir um caso base, frequentemente colocado em um bloco *if-else* e que determina a solução direta para o problema em sua menor instância.
- ✓ Possuir uma chamada a si mesmo, dentro do bloco condicional *if-else* em uma situação distinta daquela presente para o caso base.

Além disso, algoritmos recursivos em geral não necessitam de laços, pois a chamada recursiva é que desencadeia repetidos processamentos por parte da função.

2.3. CUIDADOS AO SE UTILIZAR UM ALGORITMO RECURSIVO

Todas as vezes que um algoritmo recursivo é invocado, criam-se cópias de dados na memória em uma região chamada pilha de execução (ou pilha de chamada/programa). Isso aumenta a chamada **complexidade espacial do algoritmo**.

A complexidade espacial considera os **dados de entrada** e os **dados auxiliares** requeridos pelo algoritmo.

Quando se considera o tempo de processamento (i.e., o tempo necessário para que o algoritmo retorne com a solução do problema), tem-se a chamada **complexidade temporal**.

Alterar um algoritmo/programa tipicamente afeta sua **complexidade espacial** e **temporal**.

Se uma função é chamada de modo recursivo um elevado número de vezes, isso pode gerar o estouro de pilha – stack overflow - (i.e., não há mais memória disponível para armazenar dados do programa em execução).

Quando é requerida a chamada frequente a uma função recursiva é comum a utilização de técnicas tais como cache em memória para que o desempenho do algoritmo seja melhorado.

Nesse caso, ao invés de se recorrer à chamada recursiva, examina-se o cache e verifica-se se já houve uma chamada à função com tal argumento, em caso positivo, retorna-se o cache ao invés de se invocar novamente a função.

Esquematicamente, o uso de cache em memória pode ser representado através do seguinte pseudocódigo:

```
Função recusiva(parâmetro, cache): tipo_retorno
      Se cache[parâmetro] ≠ vazio então
            retorne cache[parâmetro]
      Senão
            Se a instância do problema é "pequena" Então
                  Resolva o problema
                  cache[parâmetro] ← solução caso base
                  retorne solução caso base
            Senão
                  Reduza a instância do problema
                  Aplique a resolução à instância menor: invoque
                  recursiva(parâmetro_problema_reduzido ,cache)
                  Efetue a composição da solução
                  cache[parâmetro] ← solução
                  retorne solução
            Fim-se
      Fim-se
Fim-função
```

Note-se que a utilização de cache será eficaz quando a invocação da função de modo recursivo for substituída por um acesso direto ao cache. Essa situação é comum no cálculo de sequências numéricas onde há dependência do valor do k-ésimo termo com com os termos k-1, k-2,..., k-n. Em casos onde não há essa dependência, a utilização do cache só apresentará vantagens quando a mesma função for invocada mais de uma vez, considerando-se os mesmos argumentos ou argumentos que acabem por atingir chamadas recursivas com argumentos já utilizados.

- ✓ Caches são implementados utilizando-se estruturas tais como dicionários /mapas/ JSONs. Para esse caso, a chave é o argumento à função recursiva e o valor é o retorno correspondente (previamente calculado).

EXERCÍCIOS E PROBLEMAS

PRÁTICA DE ALGORITMOS E PROGRAMAÇÃO

P.2.1. Desenvolva um programa recursivo para calcular o menor elemento presente em um array não ordenado.
P.2.2. Desenvolva um programa recursivo que calcule o elemento com o maior valor absoluto presente em um array não ordenado.
P.2.3. Desenvolva um programa recursivo que calcule o fatorial do n-ésimo número. Faça a análise de desempenho do programa através da criação do cache para chamadas repetidas à função utilizando o mesmo argumento. Considere que a função é um método de uma classe denominada de Factorial. Crie um gráfico que mostre a evolução do tempo de execução para repetidas chamadas à mesma função, comparando a versão com e sem cache, aumentando-se o número de vezes que a função e invocada, fixando-se n (preferencialmente um valor alto que não gere estouro de pilha).

Número de vezes que a função é chamada, para n fixo	Recursiva (tempo ms)	Recursiva com cache (tempo ms)
1		
10		
1000		
.....		

P.2.4. Desenvolva um programa que calcule os elementos da sequência de Fibonacci e que exiba na tela a árvore de chamadas “aproximada”.
P.2.5. Para o item 2.4. Faça uma análise de desempenho para diferentes valores de n (termo da sequência) utilizando orientação a objetos e criação de um cache de valores. Crie um gráfico comparativo tal como descrito no exercício **P.2.3.** Para este exercício, utilize como cache uma estrutura de dados do tipo JSON.
P.2.6. Repita o problema **2.5.** mas utilizando uma instância da classe Map.
P.2.7. Defina uma classe denominada MyArray que contenha um método recursivo que retorne a soma de todos os elementos presentes no array.
P.2.8. Defina uma classe MyArray que contenha um método que imprima de modo recursivo todos os elementos do array e que contenha outro método, também recursivo, que retorne os elementos do array em ordem reversa.
P.2.9. Defina uma classe que modele uma caixa d’água. A caixa deve conter uma determinada quantidade de líquido (em litros). Defina um método, que retorna o valor total em litros, obtido de modo recursivo extraindo litro a litro a capacidade da caixa d´água até esgotar sua capacidade (esvaziar).

P.2.10. Utilizando um código recursivo desenvolvido durante as aulas ou em qualquer um dos exercícios anteriores, envolvendo passagem de array à função recursiva, crie um programa que imprima uma pilha de execução. Considere que cada dado presente na pilha de execução, para sucessivas chamadas usando array, é representado utilizando um asterisco "*". Compare o resultado considerando uma função que faz chamada recursiva simples (ex. fatorial ou progressão aritmética) com uma função que faz chamada recursiva dupla (ex. sequência de Fibonacci). A Figura abaixo mostra um exemplo de saída de execução esperado para a obtenção da pilha de execução do programa recursivo que obtém o máximo elemento em um array de 5 elementos.

```
Array original:
[ 1, 4, 10, 20, -1 ]
*****|
*********|
**********|
***********|
************|
**********|
**********|
******|
*|
O maior número do array é:  20
```

P.2.11. Desenvolva um programa recursivo que calcule o maior elemento presente em um array. O programa desenvolvido deverá sempre dividir o array ao meio e compor a solução considerando a obtenção da solução de cada uma das metades.

P.2.12. Desenvolva um programa recursivo que calcule o menor elemento presente em um array. O programa desenvolvido deverá sempre dividir o array ao meio e compor a solução considerando a obtenção da solução de cada uma das metades.

P2.13. Desenvolva um programa recursivo que calcule a soma dos elementos de um array dividindo o array ao meio. O caso base deve ser composto por um array de um único elemento ou um array vazio.

TERMINOLOGIA E CONCEITOS

TC.2.1. Complete o seguinte parágrafo com termos utilizados na área de estruturas de dados referente ao desenvolvimento de software utilizando recursão.

É importante, para qualquer profissional da área de computação que trabalhe com desenvolvimento, que os conceitos de ____(1)_____ e _____(2)_____ sejam bem diferenciados. Um ____(3)_____ corresponde a um _____(4)_____ geral e demanda uma ____(5)_____. Já uma ____(6)_____ corresponde a um _____(7)_____ do problema.

Um _____(8)_____ é resolvido desmembrando-se o problema em um problema menor até que se chegue a menor instância possível, chamada de _____(9)_____. A solução para o problema é então obtida combinando-se sucessivamente o _____(10)_____ a _____(11)_____ até que se chegue a instância original.

A ____(12)_____ diz respeito a quantidade de ____(13)_____ utilizada por um programa. A _____(14)_____ diz respeito às _____(15)_____ envolvidas no programa, já a _____(16)_____ diz respeito aos dados temporários que são necessários para que o programa inicie e termine sua execução. Alterar a maneira como um algoritmo é codificado para um programa pode _____(17)_____ ou _____(18)_____ sua complexidade _____(19)_____ e sua complexidade _____(20)_____.

TC.2.2. Dê um exemplo de algoritmo recursivo que apresente complexidade espacial superior a outro, também recursivo, porém distinto. Ilustre esboçando os dados de programa e entrada na pilha de execução. **OBS:** Seu exemplo não precisa conter código de programação mas sim explicar o porquê da complexidade computacional superior utilizando pseudocódigo para representar os dados analisados.

TC.2.3. Dê um exemplo de algoritmo recursivo que apresente complexidade espacial superior a outro em decorrência principalmente a dados internos da função e não como decorrência da entrada. **OBS:** Seu exemplo não precisa conter código de programação mas sim explicar o porquê da complexidade computacional superior utilizando pseudocódigo para representar os dados analisados.

TC.2.4. Dê um exemplo de algoritmo recursivo que apresente complexidade espacial superior a outro em decorrência principalmente da entrada ao invés de dados internos à função. Esboce o cálculo de complexidade de ambos os algoritmos para um certo número de chamadas recursivas.

TC.2.5. Explique o impacto da utilização de cache em termos de aumento ou diminuição de complexidade espacial na definição de um modelo de classe.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] CARVALHO, F. G.M. GitHub contendo códigos-fonte. Disponível em <https://github.com/fabriciogmc/algorithms_and_data_structures_ts/>

[2] BHARGAVA, Aditya Y. Entendendo algoritmos – um guia ilustrado para programadores e outros curiosos. São Paulo, Novatec, 2017.

[3] WENGROW, Jay. A common-sense guide to data structures and algorithms – level up your core programming skills. 2nd Ed. North Carolina, The Pragmatic Programmers, 2020.

[4] CORMEN, Thomas et al. Introduction to algorithms. 3rd Ed., Massachusetts, MIT Press, 2009.

ANEXO I

Instalação das dependências do TypeScript

1. **Compilador (transpiler) TypeScript:**

Digitar o seguinte comando no prompt de comando:

```
npm install -g typescript
```

2. **Executor de código TypeScript no node:**

Digitar o seguinte comando no prompt de comando:

```
npm install -g ts-node
```

3. **Instalação da dependência do prompt-sync para leitura de valores no prompt de comando:**

```
npm install prompt-sync

npm install @types/prompt-sync
```