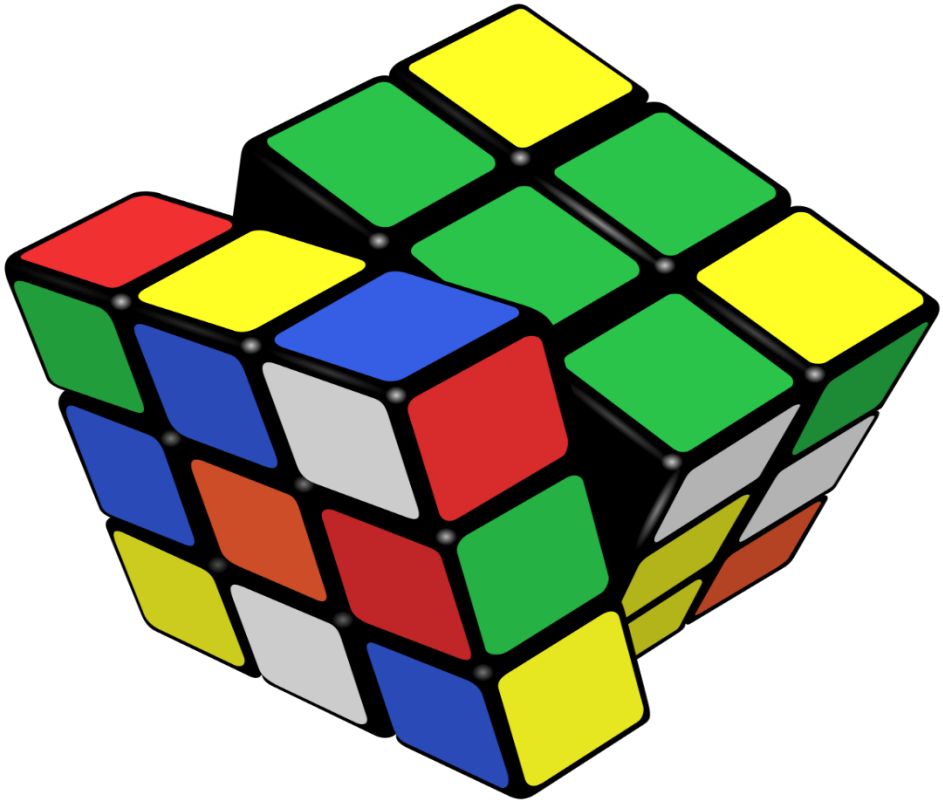
ESTRUTURAS DE DADOS



FABRÍCIO GALENDE MARQUES DE CARVALHO

Sumário

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.

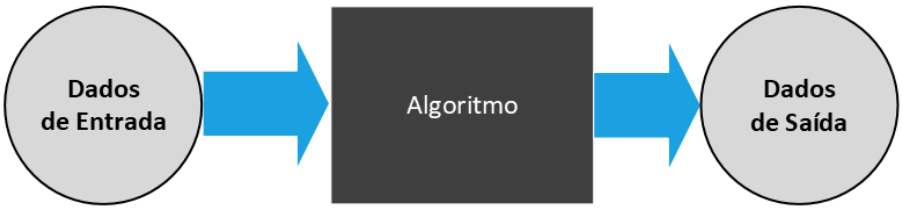


Figura 1. Diagrama de blocos de um algoritmo.

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

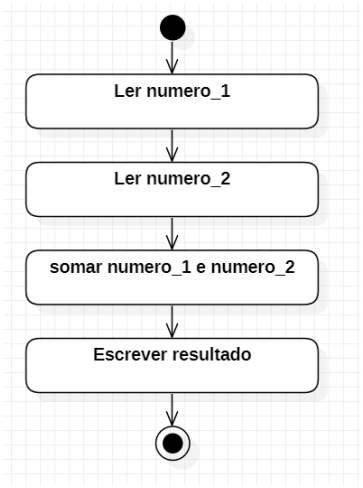


Figura 2. Diagrama de atividades representativo de um algoritmo.

d) Utilização de linguagens de programação

```typescript
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.

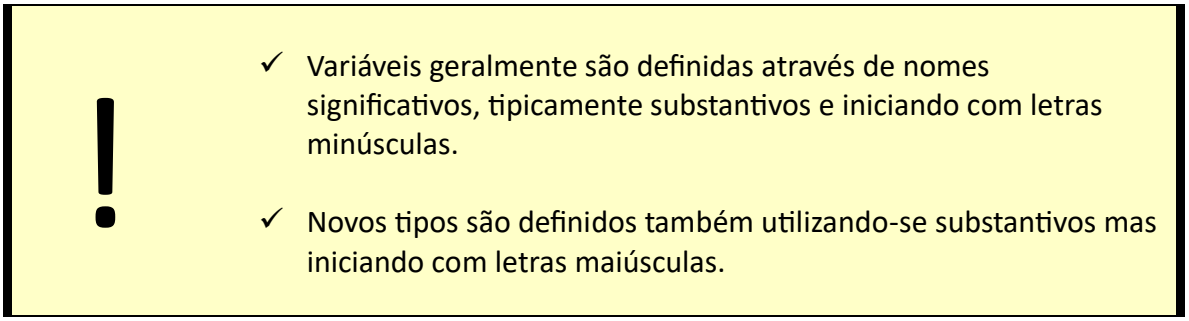
```typescript
/* Exemplo de comentário em várias linhas
**
** Autor: Fabrício G. M. de Carvalho
*/

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.

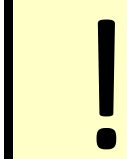
```
somar(a:inteiro, b:inteiro): inteiro
      resultado: inteiro
      resultado ← a + b
      retorne resultado
Fim-função
```

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

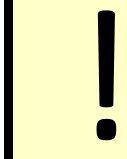
```typescript
function somar(a: number, b:number){
    let resultado: number = a + b;
    return resultado;
}
```

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:

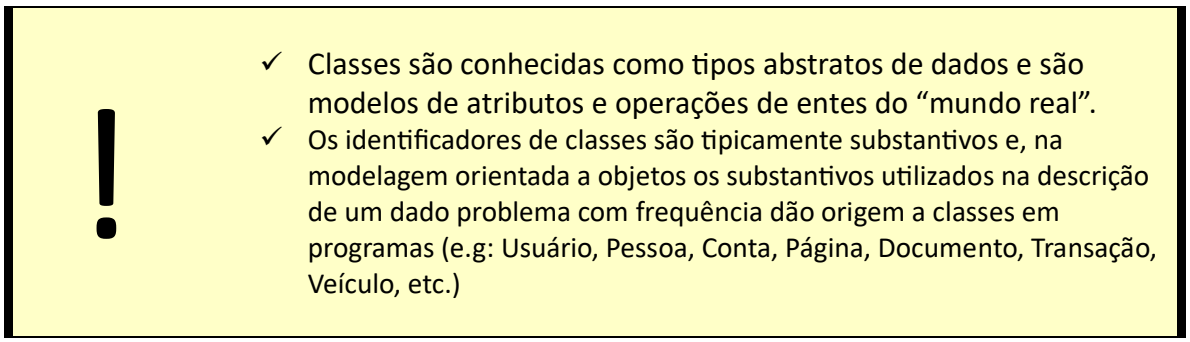
```
class Cachorro{
    raca: string;
    constructor(raca: string){
        this.raca = raca;
    }
    latir(): string{
        return "au au au";
    }
}
```

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.

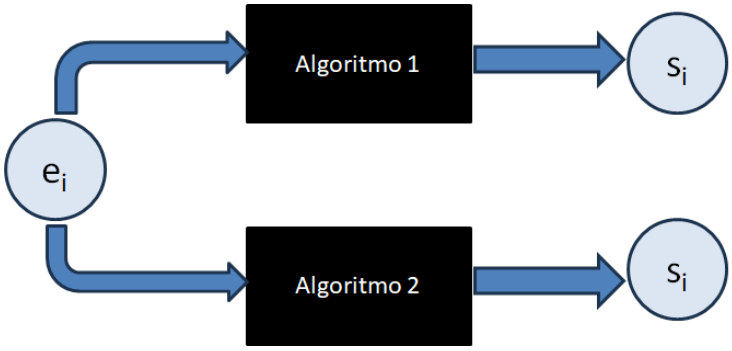


Figura 3. Algoritmos equivalentes em termos de entrada e saída.

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.

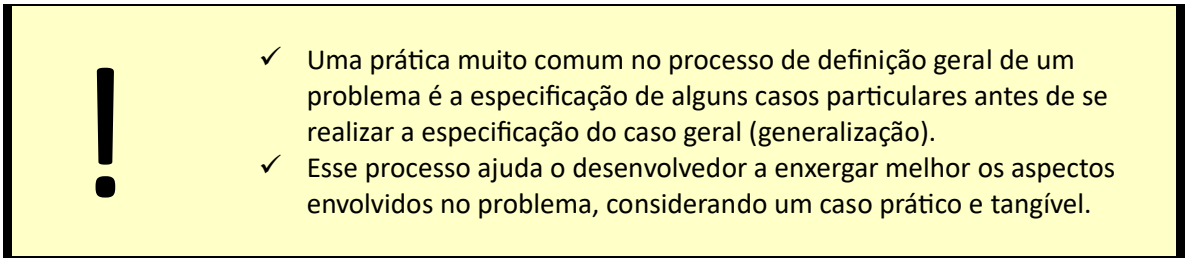
Note-se que, para este caso os números sequer são especificados, ou seja, não se sabe se são números positivos, negativos, se o zero está ou não incluído, se podem ser números contendo parte fracionária, etc.

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.

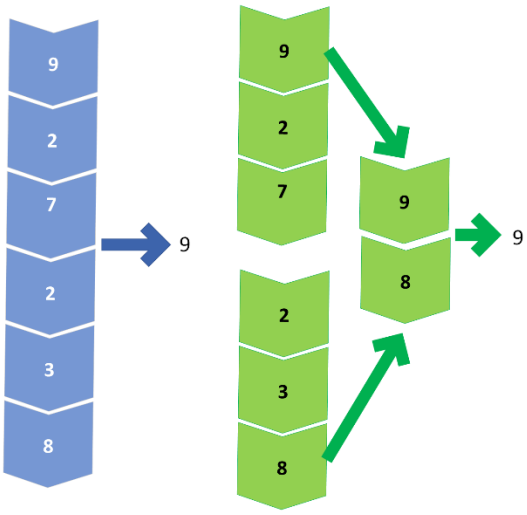


Figura 4. Determinação do menor número contido em uma lista de números. Duas abordagens, sem e com recursão.

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.

3. BUSCA E ORDENAÇÃO

3.1. BUSCA EM VETORES

A busca de dados armazenados constitui uma das operações básicas efetuadas nos mais variados tipos estruturados de dados, quer seja para sua simples leitura, alteração, exclusão ou adequada inserção.

Nessa seção são estudados algoritmos de busca aplicáveis a vetores contendo dados que podem ou não estar ordenados.

Observação: Considerar que os elementos buscados fazem parte de um vetor, significa que é possível acessar diretamente qualquer posição do vetor através do seu índice. Outras estruturas de dados dinâmicas, tais como listas ou pilhas não possuem essa propriedade. Portanto, a discussão efetuada nessa seção não é válida para estruturas que não possuam essa característica de acesso direto e indexado.

Em linhas gerais, a busca de um elemento em um vetor consiste em se resolver o seguinte problema:

Dado um elemento de tipo **`T`**, cujo valor é **`e`**, determinar, para um vetor **`v[0..N-1]`** de elementos do mesmo tipo de **`T`**, qual a posição de **`e`** em **`v`** ou, caso contrário, informar que o elemento não faz parte de **`v`**

Para resolver esse problema, dois algoritmos bastante utilizados são a **busca sequencial** e a **busca binária**.

3.1.1. BUSCA SEQUENCIAL

A busca sequencial é utilizada quando os dados não estão ordenados e, portanto, não se pode estabelecer uma relação de precedência entre tais dados e que poderia, de alguma forma, acelerar o processo de busca.

Além disso, a busca sequencial pode ser a única alternativa quando se está diante de um tipo estruturado que não permite acesso aleatório aos dados (e.g.: lista encadeada).

O algoritmo seguinte ilustra a busca sequencial:

```
busca_s(e:T, v:array[0..N-1] de T): inteiro
      pos: inteiro ← -1
      Para índice ← 0..(tamanho(v)-1)
            Se v[índice] = e
                  retorne índice
            Fim-se
      Fim-para
      retorne pos
```

Observar que a busca sequencial, para o pior caso, efetuará N comparações do elemento buscado com os elementos do vetor. Nesse caso, diz-se que o algoritmo apresenta complexidade de ordem n (indicado através de O(n)).

O melhor caso ocorre quando o elemento buscado está na primeira posição. Quando isso acontece, diz-se que o algoritmo é de ordem constante e indica-se por O(1).

No caso médio, a busca retornará o índice do elemento localizado no "meio" do vetor . Nesse caso, diz-se que o algoritmo é de ordem `n` e indica-se por O(n).

3.1.2. BUSCA BINÁRIA

A **busca binária** é utilizada quando se tem um vetor ordenado (em ordem crescente ou decrescente).

A cada "rodada" de busca compara-se o elemento procurado com o elemento localizado no meio do vetor de dados. Caso seja encontrado, retorna-se o índice do elemento no vetor. Caso não seja encontrado, reduz-se o espaço de busca à metade e procede-se com uma nova rodada até que não mais restem elementos a serem comparados.

O algoritmo seguinte ilustra uma possível descrição para a busca binária.

```
busca_binaria(e,v)
      inicio ← 0
      fim ← tamanho(v)-1
      meio ← ⌊(inicio+fim)/2⌋
      Enquanto (inicio ≤ fim)
            Se v[meio] = e  então
                  retorne meio
            Senão Se v[meio] > e   então
                        fim ← meio – 1
                  Senão
                        inicio ← meio + 1
                  Fim-se
            Fim-se
            meio ← ⌊(inicio+fim)/2⌋
      Fim-enquanto
      retorne -1
```

A cada "rodada" de busca o tamanho do vetor é sistematicamente reduzido à metade, ou seja, n, n/2, n/4, ..., $n/2^k$. Desta forma o número de operações básicas (comparações do elemento buscado com o elemento do meio) é proporcional a $\log_2 n$. Dessa forma, para o pior caso, que é quando o elemento não é encontrado no vetor, o algoritmo é de ordem $\log_2 n$ (indica-se por $O(\log_2 n)$). Cabe ressaltar que, como logaritmos podem ser expressos em diferentes bases, é comum que algoritmos dessa natureza sejam também indicados por O(log(n)), ou O(Log(n)), que utilizam a base 10 ou a base *e*.

Os algoritmos mostrados nessa seção são todos de natureza iterativa, porém, é possível defini-los de modo recursivo sendo que, para esses casos, haverá um aumento de complexidade espacial.

Exemplos de implementações recursivas e também envolvendo uso de sentinela, para o caso de busca sequencial, são ilustrados no repositório contendo código-fonte referenciado ao final desse documento.

3.2. BUSCA EM CADEIAS DE CARACTERES

Algoritmos de busca em cadeia de caracteres constituem um caso particular dos chamados "algoritmos de correspondência de cadeias".

Entre as muitas aplicações possíveis, podem ser citadas as buscas efetuadas por corretores ortográficos, busca de padrões em sequências de DNA, editores de substituição presentes em IDEs (*Integrated Development Environments*) que suportam diferentes linguagens de programação, de processamento de linguagem natural (PLN), etc.

O problema de busca em cadeias pode ser definido da seguinte forma:

Considerando dois arranjos (i.e., cadeias de caracteres) **Texto** = `t[0..(n-1)]` e **Padrão** = `p[0..(m-1)]`, com n ≥ m, diz-se que `p` ocorre com deslocamento d em `t` (ou começa em *d*) se

$$t[j+d] = p[j],\ \forall j \in \mathbb{Z}_+ |\ 0 \leq j \leq (m-1).$$

Visualmente, a ocorrência do padrão no texto pode ser ilustrada a seguir:

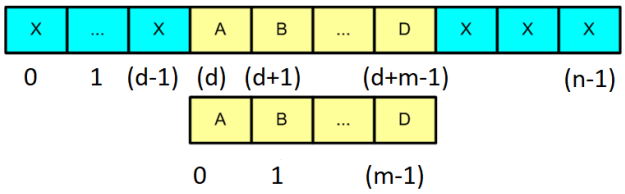


Figura 7. Ilustração da correspondência de p em t.

Se `p` ocorre com deslocamento `d` em `t`, então diz-se que `d` é um deslocamento válido. O problema então se resume a achar todos os deslocamentos válidos com os quais o padrão ocorre no texto.

3.2.1. ALGORITMO TRIVIAL

Neste algoritmo, todos os deslocamentos válidos são encontrados utilizando-se um laço que verifica a condição `p[0..(m-1)] = t[d .. (d+m-1)]` para cada um dos `(n-m+1)` valores possíveis para `d`.

```
trivial(p:string, t:string): array de inteiro
   diff ← tamanho(t)-tamanho(p)
   deslocamentos: array de inteiros
   Para d ← 0..diff
         corresponde ← verdadeiro
         Para j ← 0..(tamanho(p)-1)
               Se p[j] ≠ t[j+d] então
                     corresponde ← falso
                     interrompa
               Fim-se
         Fim-para
         Se corresponde então
               deslocamentos.adicionar(d)
         Fim-se
```

```
    Fim-para
  retorne(deslocamentos)
```

O pior caso do algoritmo ocorre quando o laço interno de comparação do padrão p é executado m vezes (i.e., todos os caracteres do padrão encontram correspondência). Neste caso, pode-se afirmar que o algoritmo é da ordem O((n-m+1).m). Se m = n, então o algoritmo será da ordem O(m), já se o texto tiver o dobro do tamanho do padrão, o algoritmo será da ordem $O(m^2)$

3.2.2. ALGORITMO DE BOYER MOORE

Trata-se de um algoritmo mais elaborado que faz uso de certas propriedades, do padrão `p` procurado, para a aceleração da busca. Nele, seguintes regras são aplicadas:

1. Efetuar a busca a partir do final da cadeia de caracteres;
2. Considerando uma não correspondência entre um caractere *c* do texto `t` e o padrão `p`, utilizar a última ocorrência de c em p para determinar o tamanho do "salto" na busca.

A figura seguinte ilustra esquematicamente os "saltos" efetuados pelo algoritmo de busca:

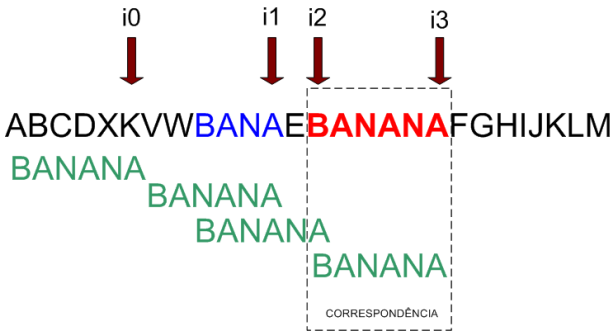


Figura 8. Esquema de "salto" do algoritmo de Boyer-Moore.

Para o adequado funcionamento do algoritmo, é necessário que se pré-processe o padrão procurado de modo que a última posição de um determinado caractere em um padrão seja armazenada. Essa posição é utilizada na determinação do "salto" quando não houve correspondência do caractere examinado no padrão no texto buscado.

```
construir_mapa_car(p:string): mapa de caracteres
   mapa ← {}
   Para i ← 0..tamanho(p)-1
        Mapa[p[i]] = i
   Fim-para
   retorne(deslocamentos)
```

Para esse algoritmo, uma chamada do tipo construir_mapa('cara') retornaria um mapa {'c':0, 'a':3, 'r':2}

Utilizando a função auxiliar, pode-se definir o algoritmo de Boyer-Moore como segue:

```
busca_boyer_moore(p:string, t:string): array de números
   mapa ← construir_mapa_car(p)
   dif ← tamanho(t) – tamanho(p)
   d ← 0
   deslocamentos ← array de números vazio
   Enquanto d ≤ dif
        corresponde ← verdadeiro
        i ← tamanho(p)-1
        Enquanto i ≥ 0
             Se p[i] ≠ t[i+d] então
                  corresponde ← falso
                  interrompa(Enquanto)
             Fim-se
             i ← i-1
        Fim-enquanto
        Se corresponde = falso então
             Se mapa[t[i+d]] ≠ vazio então
                  Se (i - mapa[t[i+d]]) > 1
                    salto ← i - mapa[t[i+d]]
                  Senão
                    salto ← 1
             Senão
                  salto ← i + 1
             Fim-se
             d ← d + salto
        Senão
             deslocamentos.adicionar(d)
             d ← d + 1
        Fim-se
   Fim-para
   retorne(deslocamentos)
```

Notar que o algoritmo apresentará o pior caso quando houver a correspondência total entre o texto procurado e o padrão. Para esse caso, o desempenho será idêntico ao algoritmo de busca sequencial que, para o caso quando $n = 2m$, fornece a complexidade $O(m^2)$. Para o melhor caso, onde não houver correspondência entre o padrão e o texto procurado, para nenhum dos caracteres, o número de saltos será dado por n/m, ou seja, a ordem do algoritmo será $O(n/m)$. Notar que se n cresce de maneira linear, em relação a m, então o tempo também crescerá de maneira linear pois o número de "saltos" será proporcional a n.

3.3. ORDENAÇÃO

Algoritmos de ordenação são algoritmos que dispõem os elementos de uma estrutura de dados de maneira que haja alguma relação de precedência. Ou seja, dependendo do fato de um elemento ser considerado antecessor ou sucessor a outro, segundo uma relação de ordem, o elemento é adequadamente posicionado na estrutura.

3.3.1. ALGORITMO DA BOLHA

Trata-se de um algoritmo de ordenação simples onde o vetor a ser ordenado é repetidamente percorrido e trocas entre os elementos adjacentes são efetuadas sempre que estão fora da ordem (crescente ou decrescente).

A cada vez que o vetor a ser ordenado é percorrido, o maior (ou menor) elemento é transferido para a última posição observada, tal como uma "bolha" emergindo em um tanque com água.

A figura seguinte ilustra a mecânica de operação do algoritmo da bolha.

```
Bubble sort:
Original array:  1 2 3 4 5 10 9 8 7 6
Array after pass #  1 :  1 2 3 4 5 9 8 7 6 10
Last swap position:  8
Array after pass #  2 :  1 2 3 4 5 8 7 6 9 10
Last swap position:  7
Array after pass #  3 :  1 2 3 4 5 7 6 8 9 10
Last swap position:  6
Array after pass #  4 :  1 2 3 4 5 6 7 8 9 10
Last swap position:  5
Array after pass #  5 :  1 2 3 4 5 6 7 8 9 10
Last swap position:  4
```

Figura 9. Ilustração de operação do algoritmo da bolha.

O algoritmo seguinte descreve uma possível definição de função que implementa o algoritmo da bolha:

```
Algoritmo bolha(v: vetor)
      esta_ordenado: booleano ← falso
      ultima_pos_troca:numero ← tamanho(v)-1
      Enquanto ( não esta_ordenado )
          esta_ordenado ← verdadeiro
          j ← 0
          Enquanto (j < ultima_pos_troca){
              Se (v[j] > v[j+1]) Então
                  aux ← v[j]
                  v[j] ← v[j+1]
                  v[j+1] ← aux
                  esta_ordenado = falso
              Fim-se
              j ← j+1
          Fim-enquanto
          ultima_pos_troca ← ultima_pos_troca -1
      Fim-enquanto
```

Em virtude do fato do algoritmo ser estruturado utilizando-se laços aninhados, apresenta, para o pior caso, complexidade da ordem de n^2 (ou seja, $O(n^2)$).

Vetores relativamente pequenos sob o ponto de vista computacional exigirão um elevado número de comparações e tornarão a utilização deste algoritmo inapropriada para elevados volumes de dados.

Para melhorar o desempenho para o caso médio, pode-se armazenar a última posição onde houve uma troca de modo a se economizar passos adicionais de comparação em parte do vetor já ordenado.

3.3.2. ALGORITMO DA INSERÇÃO

Trata-se de um algoritmo de ordenação que, apesar de não ser muito eficiente para o pior caso, é bastante simples e de fácil compreensão.

O algoritmo percorre o vetor uma única vez do sentido “esquerda para a direta” e, a medida em que encontra um elemento fora de sua posição, relativamente aos elementos anteriormente visitados, “volta à esquerda” e insere este elemento em sua posição correta, deslocando os demais elementos à direita.

É muito similar à maneira com que as pessoas costumam arrumar as cartas de baralho em suas mãos.

A figura seguinte ilustra esquematicamente o funcionamento do algoritmo de ordenação da inserção.

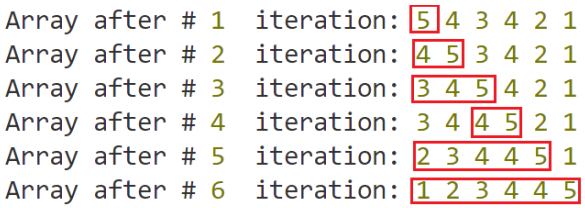


Figura 10. Exemplo esquemático de ordenação por inserção para um array contendo 6 elementos.

```
Algoritmo insercao(v: vetor de T)
      Para i:inteiro ← 0 .. tamanho(v)-1
      elemento:T ← v[i]
      pos: inteiro ← i
      Enquanto (pos>0 e v[pos-1] > elemento)
          v[pos] ← v[pos-1]
          pos ← pos-1
      Fim-enquanto
      v[pos] ← elemento
     Fim-para
```

Observando-se atentamente o algoritmo, percebe-se que, no pior caso (quando o array está em ordem reversa) serão efetuadas, a cada iteração, 1, 2, 3...N comparações e trocas de posição.

Avaliando-se o somatório do total de operações, em especial as trocas e comparações, conclui-se que o algoritmo apresenta complexidade $O(n^2)$.

3.3.3. ALGORITMO DA SELEÇÃO (SELECTION SORT)

Trata-se de um algoritmo que, a cada rodada (i.e., passagem pela estrutura) coloca o maior (ou menor) elemento na sua posição correta, definitiva, considerando a estrutura ordenada. A cada rodada esse algoritmo não efetua mais do que uma troca de posição entre os elementos.

Algoritmicamente, pode-se expressar a ordenação por seleção como segue:

```
Algoritmo selecao(v: vetor de T)
      passagem: inteiro ← 0
      Enquanto passagem < N
         e:T ← v[passagem]
         pos:número ← passagem
         Para p:numero = (passagem+1)..(N-1)
            Se v[p] < v[pos] Então
               pos ← p
         Fim-para
         v[passagem] ← v[pos]
         v[pos] ← e
         passagem ← passagem + 1
      Fim-enquanto
Fim-Algoritmo
```

A cada rodada, esse algoritmo percorre a estrutura de dados não ordenada e compara N, N-1, N-2, ..., 1 vezes, até que o algoritmo termina. Nesse caso, pode-se mostrar que o número de comparações efetuadas pelo algoritmo acarretará um desempenho da ordem de $O(n^2)$.

A figura abaixo ilustra o resultado da execução desse algoritmo para v = [7, 6, 5, 4, 3, 2, 1].

```
pass: # 0
Actual v: [ 7 6 5 4 3 2 1 ]
Swap positions:  0 and  6
pass: # 1
Actual v: [ 1 6 5 4 3 2 7 ]
Swap positions:  1 and  5
pass: # 2
Actual v: [ 1 2 5 4 3 6 7 ]
Swap positions:  2 and  4
pass: # 3
Actual v: [ 1 2 3 4 5 6 7 ]
Swap positions:  3 and  3
pass: # 4
Actual v: [ 1 2 3 4 5 6 7 ]
Swap positions:  4 and  4
pass: # 5
Actual v: [ 1 2 3 4 5 6 7 ]
Swap positions:  5 and  5
pass: # 6
Actual v: [ 1 2 3 4 5 6 7 ]
Swap positions:  6 and  6
```

3.3.4. ALGORITMO DA MESCLA (MERGE SORT)

Este algoritmo faz parte dos chamados “algoritmos de divisão e conquista”. Algoritmos desta natureza tipicamente dividem um problema em problemas menores (divisão) até que se tenha instâncias pequenas o suficiente para resolução direta (conquista) seguidas de posterior combinação das soluções menores de forma que seja obtida a solução para o problema maior original.

Em essência, este algoritmo possui três fases fundamentais:

1. Divisão do vetor a ser ordenado em dois outros vetores até que se obtenha o caso mais simples possível (i.e., um único elemento que, por definição, já está ordenado);
2. Ordenação direta dos casos mais simples através de intercalação de elementos;
3. Combinação dos casos mais simples ordenados até a obtenção do vetor completamente ordenado.

A figura seguinte ilustra a mecânica geral de operação do algoritmo de ordenação por mescla.

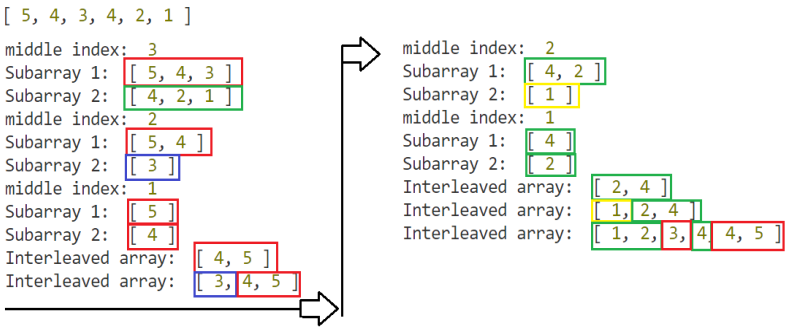


Figura 11. Esquema de funcionamento do algoritmo de ordenação por mescla.

O algoritmo que implementa a ordenação por mescla, em TypeScript, é descrito a seguir em duas partes. Na primeira parte é mostrado o método auxiliar de intercalação (`interleave`) que é o responsável por intercalar elementos pertencentes a dois vetores que já estão ordenados. A segunda parte ilustra o método da ordenação por mescla, fazendo uso do método de intercalação.

```
    interleave(v1:T[], v2:T[]):T[]{
        let v_i = new Array<T>();
        let p1:number = 0;
        let p2:number = 0;

        while(p1<v1.length && p2 < v2.length){

            if( v1[p1] < v2[p2]){
                v_i.push(v1[p1]);
                ++ p1
            } else {
                v_i.push(v2[p2]);
                ++ p2;
            }
        }

        v_i = Array.prototype.concat(v_i, v1.slice(p1,v1.length));
```

```
        v_i = Array.prototype.concat(v_i, v2.slice(p2,v2.length));
        return v_i;
    }
```

```
mergesort(v:T[]): T[]{
        if(v.length<2){
              return v;
        }
        let meio = Math.round(v.length/2)
        let v_l =this.sort(v.slice(0,meio));
        let v_r = this.sort(v.slice(meio, v.length));
        return this.interleave(v_l, v_r);
    }
```

A cada chamada recursiva, o algoritmo de ordenação merge sort reduz o espaço de ordenação à metade, ou seja, efetua $\log_2(n)$ reduções no espaço de ordenação. Adicionalmente, a cada redução é efetuada uma intercalação correspondente a **n** elementos do espaço de ordenação reduzido. Portanto, a complexidade do algoritmo é $O(\mathbf{n}.\log_2(n))$.

Conclusão: Este algoritmo possui desempenho temporal superior aos algoritmos da Bolha e da Inserção (ambos com complexidade $O(n^2)$ para o pior caso). Note-se, entretanto, que este algoritmo requer memória adicional para efetuar as divisões e intercalações.

3.3.5. ALGORITMO DA ORDENAÇÃO RÁPIDA (QUICK SORT)

Trata-se de um algoritmo de ordenação que utiliza a ideia de se ordenar gradualmente um vetor em termos de um elemento que é escolhido como referência: o pivô.

Elementos maiores passam a ser posicionados, para o caso da ordenação crescente, à direita do pivô e elementos menores passam a ser posicionados à esquerda do pivô.

A figura seguinte ilustra a mecânica de operação do algoritmo Quick Sort.

```
Original array: [ 7 6 5 4 8 9 10 3 2 1 0 ]
Quick sort detailed view:
left_array: [ 6 5 4 3 2 1 0 ], pivot:  7  right_array: [ 8 9 10 ]
left_array: [ 5 4 3 2 1 0 ], pivot:  6  right_array: [ ]
left_array: [ 4 3 2 1 0 ], pivot:  5  right_array: [ ]
left_array: [ 3 2 1 0 ], pivot:  4  right_array: [ ]
left_array: [ 2 1 0 ], pivot:  3  right_array: [ ]
left_array: [ 1 0 ], pivot:  2  right_array: [ ]
left_array: [ 0 ], pivot:  1  right_array: [ ]
left_array: [ ], pivot:  8  right_array: [ 9 10 ]
left_array: [ ], pivot:  9  right_array: [ 10 ]
Final ordered array: [ 0 1 2 3 4 5 6 7 8 9 10 ]
```

Figura 12. Mecânica de operação do algoritmo quick sort para um array contendo 11 elementos.

A seguir, é mostrada uma possível implementação recursiva do algoritmo em TypeScript.

```
  quick_sort(v:T[]): T[]{
    if(v.length<2){
        return v;
    }
    let pivot = v[0];
    let left_array: T[] = new Array<T>();
    let right_array: T[] = new Array<T>();
    for(let i = 1; i <v.length; ++i){
        if (v[i]< pivot)
            left_array.push(v[i]);
        if (v[i]>=pivot)
            right_array.push(v[i]);
    }
    return Array.prototype.concat(this.sort(left_array),
                                  pivot,
                                  this.sort(right_array));

}
```

Para o pior caso, este algoritmo possui complexidade $O(n^2)$. Entretanto, para o caso médio, a complexidade deste algoritmo é reduzida para $O(n.\log_2(n))$. Intuitivamente, basta pensar que, ao dividir o espaço de busca tomando-se em consideração o pivô, na média, metade dos elementos estará automaticamente ordenada em relação à outra metade. Além disso, a cada divisão, serão necessárias n comparações em relação a tal pivô para se reposicionar os elementos maiores e menores.

Assim como no caso do algoritmo merge sort, o quick sort requer memória adicional durante o particionamento do vetor em maiores, menores e iguais. Ou seja, há um aumento da complexidade espacial em relação às implementações iterativas mostradas para outros algoritmos, tais como o algoritmo da bolha.

EXERCÍCIOS E PROBLEMAS

PRÁTICA DE ALGORITMOS E PROGRAMAÇÃO

OBS: Em todos os exercícios resolvidos, utilizar as implementações fornecidas pelo professor para algoritmos de busca e ordenação, fazendo as devidas adaptações no mesmo arquivo. Não utilizar pacotes de software, bibliotecas, etc., prontos, implementar o algoritmo 'do zero', exceto pelas funções de leitura e escrita em arquivo.

P.3.1. Desenvolva um programa que leia os dados de um arquivo de texto e que armazene cada uma das palavras em um vetor (cada elemento do vetor contém uma palavra). Depois de carregar os dados no vetor, o programa solicita ao usuário a digitação de uma palavra e então faz a busca sequencial e informa a posição da palavra no vetor. Na sua implementação, utilizar o algoritmo de busca sequencial genérica, fornecido pelo professor no repositório do GitHub.

P.3.2. Desenvolva um programa que leia os dados de um arquivo de texto, contendo palavras em ordem alfabética, e que armazene cada uma das palavras em um vetor (cada elemento do vetor contém uma palavra). Depois de carregar os dados no vetor, o programa solicita ao usuário a digitação de uma palavra e então faz a busca binária e informa a posição da palavra no vetor. O programa também deve fornecer como saída o número de passos de comparação que foram efetuados até que a palavra seja encontrada. As entradas para o programa devem ser o arquivo de texto contendo as palavras em ordem alfabética e a palavra buscada. A saída o número de passos e em qual posição a palavra foi encontrada.

P.3.3. Desenvolva um programa que leia os dados de um arquivo de texto e que armazene os as palavras lidas em uma única string. Essa string deverá ser separada por algum delimitador configurável e que será uma entrada adicional ao programa (ex. vírgula, ponto e vírgula, espaço em branco, etc.). A seguir, o programa solicita ao usuário que digite uma palavra e, então, usando o algoritmo de busca sequencial, informa a quantidade de ocorrências da palavra na string que foi carregada do arquivo de texto e, também, os deslocamentos onde a palavra (padrão) foi encontrada. Ao ser encerrado, o programa informa também a quantidade de deslocamentos que foram utilizados para percorrer toda a string.

P.3.4. Repetir o Problema 3.3 mas utilizando o algoritmo de Boyer-Moore.

P.3.5. Desenvolva um programa que leia um arquivo de texto e que armazene cada uma das palavras em um vetor (uma palavra por célula do vetor). A seguir, o programa efetua a ordenação usando o algoritmo Bubble Sort e escreve, em um arquivo de saída (colocar o sufixo _ord.txt no nome do arquivo) as palavras em ordem lexicográfica.

P.3.6. Repetir o Problema 3.5 utilizando o Merge Sort.

P.3.7. Repetir o Problema 3.5. utilizando o Insertion Sort.

P.3.8. Repetir o Problema 3.5. utilizando o algoritmo Quick Sort.

P.3.9. Desenvolva um programa que leia um arquivo de texto e armazene cada palavra em um vetor. A seguir, utilizando o algoritmo de busca sequencial, determina, para cada palavra existente no vetor, o número de ocorrências dessa palavra ao longo do texto. Seu programa deverá ser "inteligente" o suficiente para ignorar caracteres de pontuação do tipo ",", ";" , "." e ":". Ou seja, a busca da palavra "adorei", deve retornar a posição 1 se a string de texto procurado

for composta por " adorei!" . Nesse caso, o seu programa deverá contabilizar os caracteres de pontuação de modo individual, segundo o exemplo seguinte:

Exemplo de conteúdo do arquivo de texto:
Eu gosto muito de Matemática. Eu Amo Matemática.
Saída: {Eu:2, gosto: 1, muito:1, de: 1, Matemática: 2, '.'=2}

P.3.10. Repita o Problema 3.9 mas armazenando os dados do arquivo de texto em uma única variável do tipo string e utilizando o algoritmo de Boyer-Moore.

P.3.11. Desenvolva um programa que leia um texto de um arquivo e um conjunto de verbos no infinitivo e suas formas flexionadas de outro arquivo. Utilizando o algoritmo de Boyer-Moore, gere um arquivo de saída (colocar o sufixo _out.txt) que seja idêntico ao arquivo original mas substituindo os verbos flexionados por suas formas no infinitivo. Considerar que tanto as entradas como as saídas estarão todas com letras minúsculas.

Exemplo de conteúdo do arquivo de texto de entrada:
Amamos café da manhã quando é bem servido.

Exemplo de arquivo de texto com verbos no infinitivo e suas formas flexionadas:
amar: amamos, amo, amas, amais
ser: sou, é, será
servir: sirvo, serves, servimos, servido

Exemplo de conteúdo de arquivo de saída:
Amar café da manhã quando ser bem servir.

OBS: Em sistemas de processamento de texto, essa técnica é denominada de lematização.

P.3.12. Um stemizador é um bloco de uma pipiline de processamento de linguagem natural que simplesmente "corta" palavras e obtém uma raiz que não necessariamente faz parte do léxico de uma linguagem. Por exemplo, a stemização da palavra "necessariamente" pode ser "necessária". Utilizando o algoritmo de Boyer-Moore, desenvolva um programa que sirva como stemizador. Seu stemizador deve receber como entradas: 1. Um arquivo de texto contendo o texto a ser stemizado; 2. O tamanho da preservação do stemizador (ex. 3, significa que somente os 3 primeiros caracteres de uma palavra serão mantidos, de forma que a palavra "ana" não seria alterada, mas a palavra "analfabeto" seria cortada e reduzida à "ana"). A saída deve ser um arquivo de texto contendo o texto original stemizado, sendo esse arquivo de extensão _out.txt.

P.3.13. Faça a análise comparativa dos algoritmos de ordenação da bolha não otimizado e bolha otimizado. Na sua análise monte uma tabela e um gráfico ilustrativo do tamanho do vetor, versus tempo de ordenação, considerando um vetor contendo 1.000, 10.000 e 100.000 elementos. Considere o pior caso (elementos ordenados em ordem reversa para ordenação em ordem direta) e o caso médio (gere elementos na faixa de valores do vetor utilizando um gerador pseudoaleatório.

P.3.14. Repita o problema P.3.13 mas comparando o algoritmo da seleção com o algoritmo da bolha.

P.3.15. Repita o problema P.3.13 mas comparando para o pior caso e para o caso médio o algoritmo da bolha não otimizado com o algoritmo merge sort. Para evitar estouro de pilha, faça ordenações repetidas sobre o mesmo array não ordenado (ex. ao invés de ordenar um array de 100.000 elementos, ordene um array de 1000 elementos 100 vezes).

P.3.16. Repita o problema P.3.13 mas comparando o algoritmo da bolha com o algoritmo quick sort. Siga as orientações do problema P.3.15.

P.3.17. Construa um sistema que ordene valores em uma lista de objetos contendo os atributos nome:string e idade:número. Seu sistema deve ordenar os valores primeiramente pelo nome e, posteriormente, pela idade. Note que seu algoritmo de ordenação deverá operar em 2 camadas. A primeira ordena pelo nome e a segunda ordena os objetos de mesmo nome mas utilizando o telefone. Faça a devida adaptação no algoritmo de ordenação da bolha fornecido pelo professor.

P.3.18. Repita o problema 3.17 mas utilizando o algoritmo da inserção.

P.3.19. Repita o problema 3.18 mas utilizando o algoritmo da mescla.

P.3.20. Repita o problema 3.19 mas utilizando o algoritmo de ordenação rápida.

TERMINOLOGIA E CONCEITOS

Para os problemas seguintes, considere os seguinte vetor não ordenado:

[0, 4, 3, 33, 22, -1, 4, 32, -3, -4, - 9, 31, 5, 2]

TC.3.21. Ilustre a operação do algoritmo de ordenação da bolha para cada “passagem ao longo do vetor.

TC.3.22. Ilustre a operação do algoritmo de ordenação por inserção para cada “passagem reversa” e direta ao longo do vetor. Mostre a posição do índice de comparação em ordem inversa.

TC. 3.23. Ilustre a operação do algoritmo de ordenação por mescla, informando todos os passos de divisão e conquista. Adicionalmente, indique a posição dos índices durante as intercalações, de modo a detalhar cada rodada de intercalação.

TC. 3.24. Ilustre a operação do algoritmo de ordenação rápida informando as definições dos pivôs, os subvetores da esquerda e a direita para cada rodada de chamada recursiva. Adicionalmente, mostre o apontador da posição que está selecionando o elemento que está sendo comparado ao pivô.

4. LISTAS ENCADEADAS, FILAS E PILHAS

4.1. INTRODUÇÃO

Listas, **filas** e **pilhas** são estruturas de dados que alocam memória dinamicamente, conforme os dados são acrescentados a essas.

Esses tipos de estruturas são ditas estruturas lineares e sequenciais, pois os dados são vinculados e acessados "um após o outro" de modo que só há um (ou nenhum) dado prévio e um (ou nenhum) dado seguinte.

A diferenciação entre essas três estruturas se dá em termos de **política de acesso aos dados**, ou seja**, como os dados podem ser acrescentados e removidos da estrutura**.

4.2. LISTAS ENCADEADAS

Listas encadeadas são **tipos estruturados** de dados que representam **sequências de dados** na memória do computador.

Em geral, as listas estão associadas a regiões de memória que são dinamicamente alocadas. Cada endereço de memória presente na lista não necessariamente é contíguo a outro endereço de memória presente na mesma lista.

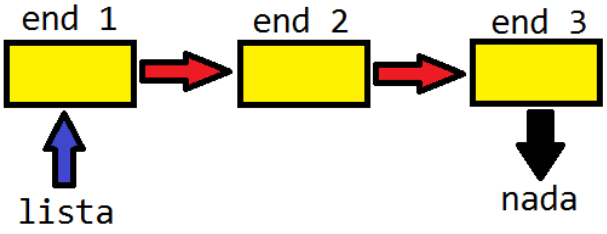


Figura 13. Esquema de vínculo entre elementos de uma lista.

Uma lista é composta por **nós** (em inglês o termo é node) ou **células**. É nos nós que tanto os dados como a referência aos demais nós da lista são armazenados.

Dependendo da relevância ou não dos dados armazenados no primeiro nó da lista, pode-se ter uma lista dita **"com cabeça"** ou **"sem cabeça"**. Listas "sem cabeça" são aquelas em que não há qualquer diferenciação entre os seus nós. Já listas com cabeça são aquelas onde o primeiro nó aponta para nada só se a lista estiver vazia e dado armazenado é irrelevante.

As operações básicas que podem ser realizadas com listas são a sua criação, inserção de elementos, remoção de elementos, alteração de elementos, busca de elementos e a exibição do conteúdo da lista.

- **Criação:** Criação de uma lista vazia;
- **Inserção:** Alocação de nova região de memória (novo nó), escrita de conteúdo e vinculação à lista;
- **Remoção:** busca sequencial do dado na lista seguida de sua desvinculação (eliminação do nó que contém o dado);
- **Alteração:** busca sequencial do dado seguida de alteração;
- **Exibição:** Caminhamento e impressão sequencial dos dados presentes em cada nó da lista.

O algoritmo seguinte ilustra a definição de um nó. Para essa classe `valor` corresponde ao dado que se deseja armazenar e `próximo` é uma referência a um outro nó, ou seja, a um outro objeto da mesma classe Nó. Por esse motivo, diz-se que um nó é uma **estrutura autorreferenciada**.

```
Classe Nó:
    valor: tipo
    próximo: Nó
    construtor(v:tipo): Nó
        valor ← v
        próximo ← nada
    fim-função
Fim-classe
```

Uma lista pode ser considerada como uma referência ao primeiro nó. Além disso, deve prover métodos de criação (inicialização), inserção, remoção, alteração e exibição/listagem. O algoritmo seguinte ilustra esquematicamente a definição de uma classe Lista, com o detalhamento do método de criação/inicialização que, para esse caso, é chamado de `construtor`. Além disso, são mostradas as interfaces para os métodos de inserção, remoção, alteração e listagem.

```
Classe Lista:
      início: Nó
      tamanho: inteiro
      construtor(): Lista
        início ← nada
        próximo ← 0
      fim-função

      inserir(n:Nó): nada

      remover(n:Nó): booleano

      alterar(n:Nó): booleano

      exibir(): nada


Fim-classe
```

Uma possível implementação dessa estrutura de dados pode ser encontrada em

https://github.com/fabriciogmc/data_structures_source_lectures_typescript/tree/main/listas_encadeadas

4.3. PILHAS

Assim como as listas e filas, pilhas são tipos estruturados dinâmicos (mas que podem possuir capacidade estaticamente limitada, dependendo da implementação) utilizados para armazenar

dados quando se deseja que a política de acesso seja do tipo *FILO* (do inglês First-In-Last-Out - primeiro a entrar, último a sair).

Pilhas são amplamente aplicadas no ramo da computação. Exemplos de sua utilização incluem a construção de compiladores, interpretadores de comandos, funções de gerenciamento de memória fornecidas por sistemas operacionais, etc.

A figura abaixo ilustra esquematicamente uma pilha.

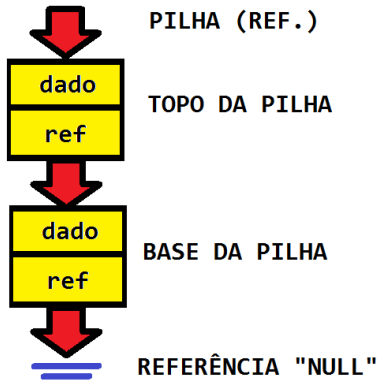


Figura 14. Esquema representativo de uma pilha.

Como se pode perceber pela figura, uma pilha possui a estrutura básica de um Nó como elemento constituinte.

Ao nos referirmos a uma pilha, tipicamente estamos falando de seu topo (ou seja, referência ao elemento localizado no topo).

O elemento localizado no topo foi o último elemento acrescentado e deve ser o primeiro a ser retirado.

Quando não há mais elementos na pilha, a referência da pilha não aponta para nada (ou seja, a pilha está vazia).

Quando uma pilha possui capacidade limitada e esta é excedida, diz-se que houve estouro de pilha ("*stack overflow*")

O algoritmo abaixo ilustra esquematicamente a definição de uma classe `Pilha`, considerando a mesma estrutura de `Nó` que foi definida para uma Lista. São mostrados os métodos de construção, verificação se a estrutura está vazia, empilhamento, desempilhamento e exibição.

```
Classe Pilha:
      topo: Nó
      tamanho: inteiro
      construtor(): Pilha
        topo ← nada
        tamanho ← 0
      fim-função

      esta_vazia(): booleano
            Se tamanho = 0 então
                  retorne verdadeiro
            Senão
```

```
                retorne falso
            Fim-Se
        Fim-função

        empilhar(n:Nó): vazio
              n.proximo ← topo
              topo ← n
              tamanho ← tamanho - 1
        Fim-função

        desempilhar(): Nó
          no ← nada
          Se (não esta_vazia()) então

               No ← topo;
               topo ← topo.próximo
               tamanho ← tamanho - 1
          Fim-se
          retorne no;
        Fim-função


        exibir(): vazio
              no_atual:Nó ← topo
              Enquanto ( no_atual ≠ nada )
                    imprimir( no_atual.valor )
                    no_atual ← no_atual.próximo
              Fim-Enquanto
        Fim-função

Fim-classe
```

Uma implementação da classe Pilha pode ser achada em:

https://github.com/fabriciogmc/data_structures_source_lectures_typescript/tree/main/pilhas

4.4. FILAS

Assim como as listas encadeadas, as filas são tipos estruturados de dados que são **dinâmicos** (i.e., alocação e liberação dinâmica de memória).

As filas possuem uma estrutura diferente das listas encadeadas em virtude de suas **restrições comportamentais**.

A figura seguinte esquematiza uma fila de sentido único não-circular.

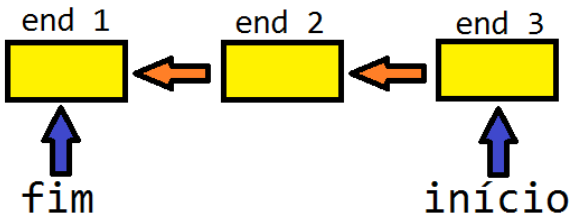


Figura 15. Esquema de referência e nós para uma fila unidirecional (não-circular).

Para uma fila, as seguintes restrições comportamentais devem ser observadas:

1. A solicitação de remoção de um elemento da fila sempre retorna o primeiro elemento que foi inserido (no **início**);
2. A solicitação de adição de um elemento à fila implica em uma adição ao seu **final**.

Estas restrições comportamentais criam uma política de acesso aos elementos da fila chamada de FIFO (do inglês, *First-In-First-Out*, o primeiro a entrar é o primeiro a sair).

O algoritmo seguinte ilustra a definição de uma classe Fila e o seu método de construção (criação de uma fila vazia). As assinaturas dos métodos de adição, remoção e verificação se está vazia também são mostrados.

```
Classe Fila:
      tamanho: número
      início : Nó
      fim: Nó
      construtor(): Fila
        início ← nada
        fim ← nada
        tamanho ← 0
      Fim-função

      esta_vazia(): booleano

      adicionar(n:Nó): vazio

      remover(): Nó

      exibir(): vazio

Fim-classe
```

Uma implementação da classe Fila pode ser encontrada em:

https://github.com/fabriciogmc/data_structures_source_lectures_typescript/tree/main/filas

EXERCÍCIOS

OBS: Em todos os exercícios resolvidos, utilizar as implementações fornecidas pelo professor e fazer as devidas adaptações no mesmo arquivo. Não utilizar pacotes de software, bibliotecas, etc., prontos. Implementar o algoritmo 'do zero', exceto pelas funções de leitura e escrita em arquivo.

PRÁTICA DE ALGORITMOS E PROGRAMAÇÃO

P.4.1. Desenvolva um programa que leia os dados de um arquivo de texto e que armazene cada uma das palavras em uma lista encadeada (cada nó da lista contém uma palavra). Depois de

carregar os dados na lista, o programa solicita ao usuário a digitação de uma palavra e então faz a busca sequencial e informa a posição da palavra na lista. Caso a palavra apareça mais de uma vez, o programa deve informar isso na saída.

P.4.2. Desenvolva um programa que leia os dados de um arquivo de texto e que armazene cada uma das palavras em uma lista encadeada (cada nó da lista contém uma palavra). Depois de carregar os dados na lista, o programa inverte a ordem dos elementos da lista (utilizando somente uma outra lista) e imprime a lista em ordem direta e em ordem reversa. Dependendo da escolha do usuário, o programa gera um novo arquivo de saída (colocar o sufixo _out.txt) que contém o texto idêntico ao original ou em ordem reversa (atentar para os espaços em branco e para os sinais de pontuação).

P.4.3. Adapte o algoritmo de ordenação da bolha para que receba como entrada uma lista encadeada e devolva na saída uma lista encadeada ordenada.

P.4.4. Desenvolva um algoritmo recursivo que imprima uma lista encadeada. Use como exemplo de lista encadeada a definição fornecida do repositório de referência da disciplina.

P.4.5. Desenvolva um algoritmo que popule uma lista com 100, 1.000, 10.000 e 100.000 elementos. Primeiramente gere nós com valores aleatórios, no intervalo [0, N-1], sendo N o tamanho da lista. Depois, efetue uma busca sequencial, adaptando o algoritmo de remoção fornecido pela implementação de lista no GitHub. Anote os tempos para que a busca seja encerrada (busque um valor considerado no meio da faixa de valores possíveis, um ao final da faixa e outro valor que não exista na lista). Repita esse procedimento, mas considerando a geração de números de modo ordenado (não aleatório) e adapte o algoritmo de tal forma que considere a ordenação na lista. Anote os tempos para que o algoritmo seja encerrado e faça uma tabela comparativa. Qual sua conclusão? Lembre que a busca retorna sempre os índices do elemento buscado e que a busca pode retornar mais de um índice para o caso de geração de elementos repetidos. Além disso, considere os mesmos números em ambas as listas, sendo que, no segundo caso, os números não estarão fora de ordem.

P.4.6. Desenvolva um programa que leia os dados de um arquivo de texto, armazene cada uma das palavras em uma pilha, conforme são lidas do arquivo original. A seguir, o programa gera um arquivo de saída (colocar o sufixo _out.txt) que contém o texto com as palavras e sinais de pontuação em ordem invertida.

P.4.7. Repita o problema **P.4.5** mas modificando a implementação da pilha para utilização de array para armazenamento de dados. Considere que o topo da pilha é a última posição indexável do array e que a base da pilha é a primeira posição. Se o topo aponta para um número negativo, considere que a pilha está vazia.

TERMINOLOGIA E CONCEITOS

TC.4.8. Qual a vantagem, em termos de algoritmo de busca sequencial, de se ter uma lista encadeada ordenada em relação a uma lista encadeada não ordenada? Dê um exemplo prático considerando uma lista contendo 10 elementos com valores numéricos e forneça argumentos

relacionados ao número de comparações necessárias ao algoritmo e, também, relacionados ao critério de parada.

5. ÁRVORES E GRAFOS

5.1. INTRODUÇÃO

Quando foram estudadas as estruturas de dados lineares listas, filas e pilhas, cada elemento (i.e., nó) estava associado a nenhum, um ou dois elementos do mesmo tipo (i.e., zero, um ou dois nós).

Esses tipos de estruturas são razoáveis para representar diversos problemas do mundo real (e.g.: pessoas aguardando em uma fila de banco). Entretanto, há problemas nos quais esses tipos de estruturas não são capazes de representar da melhor forma a associação existente entre os elementos (e.g.: possíveis rotas entre uma cidade e outras). Nesses casos, estruturas de dados mais gerais, denominadas de grafos, são mais convenientes.

5.2. GRAFOS

Um grafo é uma estrutura de dados formada por dois elementos básicos: **vértices** (ou nós) e **arestas** (ou conexões).

Um vértice *v* é um elemento simples que pode conter um nome e outros atributos (i.e., dados de interesse). O conjunto de vértices é representado por V.

Uma aresta *a* representa a conexão entre dois vértices. O conjunto de arestas é representado por A. Dessa forma, um grafo é descrito genericamente através de:

Grafo: G = (V, A)

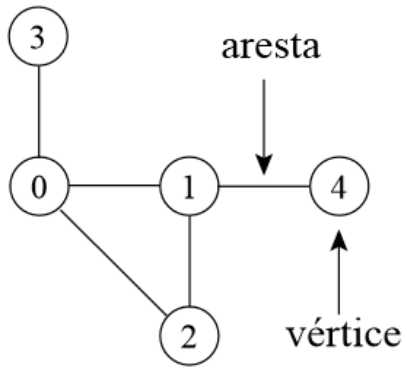


Figura 16. Exemplo visual de um grafo.

Um **grafo direcionado** é um conjunto de vértices e arestas G= (V,A) em que o conjunto V é finito e as arestas representam uma relação binária em V, ou seja, a aresta relaciona um vértice v_i a outro vértice v_j.

Uma aresta v_iv_j sai do vértice v_i e incide no vértice v_j e o vértice v_i é dito adjacente ao vértice v_j.

Uma aresta pode sair de um vértice e incidir nele mesmo.

A figura seguinte ilustra um grafo direcionado e os respectivos conjuntos de vértices e arestas:

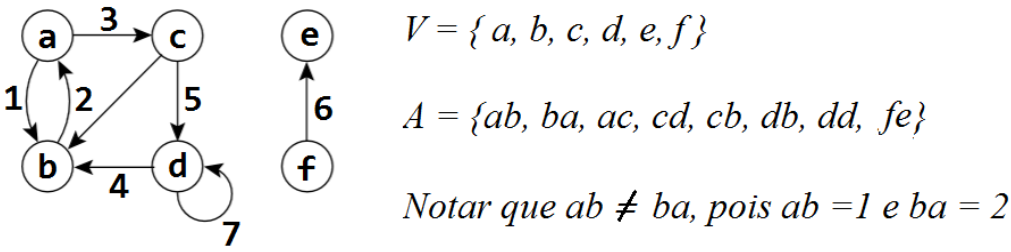


Figura 17. Exemplo de grafo direcionado.

Um **grafo não direcionado** é um conjunto de vértices e arestas G= (V,A) em que o conjunto V é finito e as arestas são representadas por pares de vértices não ordenados (i.e., ab = ba, ou seja, as arestas (a,b) e (b,a) são consideradas iguais).

Em um grafo não direcionado, se há uma aresta entre os vértices a e b, então a é adjacente a b e b é adjacente ao vértice a. Neste caso, graficamente não é necessária a utilização de setas na representação.

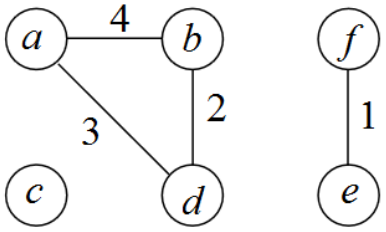


Figura 18. Exemplo de grafo não direcionado.

Considerando grafos não direcionados, um grafo $\bar{G}$ é complementar a um grafo G caso tenha sido obtido pela eliminação das arestas existentes em G e pela criação de arestas não existentes em G. A figura seguinte ilustra um exemplo de grafos complementares.

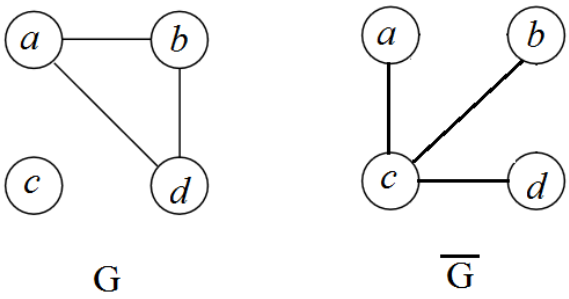


Figura 19. Exemplo de grafos complementares.

O **grau de um vértice v**, denotado por $d_G(v)$, em um grafo não direcionado, corresponde ao número de arestas que incide em v.

Um grafo é dito **conexo** se seus vértices são ligados dois a dois, ou seja, se para quaisquer vértices v_i e v_j existe um caminho com v_i e v_j em seus extremos.

As figuras seguintes ilustram exemplos de grafo conexo e não conexo e de graus de vértices.

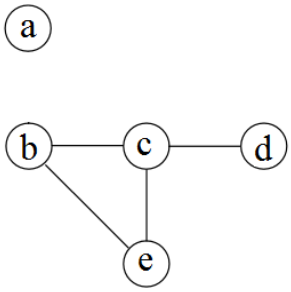


Figura 20. Exemplo de grafo não conexo, com $d_G(a) = 0$.

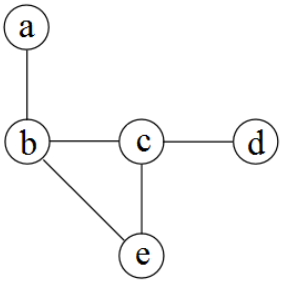


Figura 21. Exemplo de grafo conexo. $d_G(a) = 1$.

Um grafo não direcionado contendo *n* vértices é dito **completo** se cada vértice é adjacente aos demais.

Um grafo completo de n vértices é representado por K_n e $K_n = \bar{G} + G$.

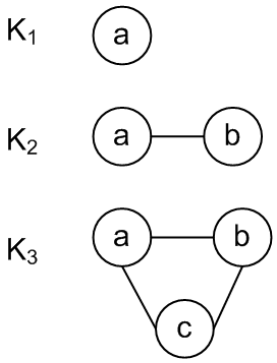
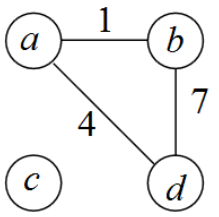


Figura 22. Exemplos de grafos completo contendo de 1 a 3 vértices.

Um grafo pode ser representado de várias formas. Duas das mais comuns são através das matrizes de adjacência e através de dicionários/documentos.

A figura seguinte ilustra um exemplo de representação de um grafo não direcionado utilizando matriz de adjacências.



	a	b	c	d
a	0	1	0	4
b	1	0	0	7
c	0	0	0	0
d	4	7	0	0

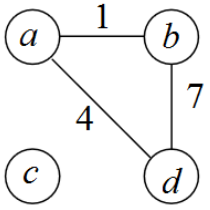
Figura 23. Exemplo de representação de grafo através de matriz de adjacências.

A representação através de matriz de adjacências possui como vantagem a fácil indexação (localização dos vértices) e verificação de existência de conexão. Como desvantagem, essa representação requer muita memória para o caso de grafos com muitos vértices e que sejam pouco conectados.

Um exemplo de implementação dessa representação, em TypeScript, pode ser encontrado no seguinte repositório:

https://github.com/fabriciogmc/data_structures_source_lectures_typescript/tree/main/grafos

A seguir, a próxima figura ilustra um exemplo de representação utilizando dicionários/documentos. Nessa representação, cada vértice corresponde a uma chave do dicionário ou propriedade do documento. Os valores são listas contendo pares de vértices ligados ao vértice da chave e as respectivas arestas com eventuais atributos adicionais.




```
G['a'] = [('b',1), ('d',4)]
G['b'] = [('a',1), ('d',7)]
G['c'] = [ ]
G['d'] = [('a',4), ('b',7)]
```

Figura 24. Exemplo de representação de grafo através de dicionários/documentos.

Para o grafo anterior, o dicionário/documento representativo do grafo seria:

G = {'a':[('b',1), ('d',4)], 'b': [('a', 1), ('d', 7)], 'c': [ ], 'd': [('a',4), ('b',7)]}

A representação através de documentos possui como vantagem a economia de memória para o caso de grafos pouco ligados. Como desvantagem, essa representação pode requerer um maior número de passos necessários à localização de elementos conectados.

5.3. ÁRVORES

Um grafo não direcionado é **acíclico**, se não apresenta circuitos, isto é, partindo-se de um vértice v não é possível retornar a v sem se repetir uma aresta já percorrida.

Um **grafo conexo e acíclico** (i.e., grafo onde se pode chegar a qualquer vértice partindo-se de qualquer outro e onde não há ciclos) é denominado de **árvore**.

Caso um vértice da árvore possua grau 1 e nele incida somente uma aresta, este vértice é dito ser uma **folha.**

Árvores são estruturas de dados que são comumente utilizadas para representar dados que guardam alguma relação hierárquica. Dessa forma, é comum também a utilização de uma notação que representa a hierarquia entre os vértices da árvore.

Em uma árvore, um dos vértices pode ser designado como **raiz**. A raiz pode ser considerada como o **vértice no "topo" da hierarquia**. Note-se que uma raiz, assim como uma folha, pode apresentar grau 1, porém, não há arestas incidentes na raiz.

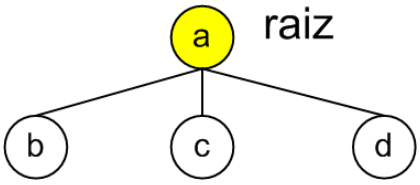


Figura 25. Representação esquemática de uma árvore. O nó a é a raiz e os nós b, c e d são folhas.

O **pai** de um vértice é o vértice que é adjacente a este e que pertence ao caminho que o leva à **raiz.** O **filho** de um vértice v_i é o vértice v_j o qual tem como pai v_i. Uma **folha** é um vértice sem filhos.

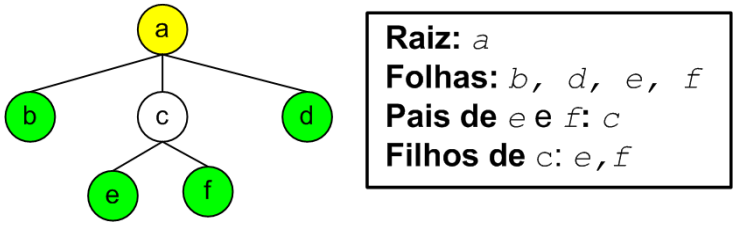


Figura 26. Exemplo de relação hierárquica em uma árvore.

5.4. ÁRVORES BINÁRIAS

Uma árvore binária é aquela em que qualquer vértice possui no máximo 3 vértices adjacentes, ou seja, o grau de cada vértice é menor ou igual a 3.

Para o caso de vértice com grau 3, uma das arestas necessariamente deve vir do nó pai.

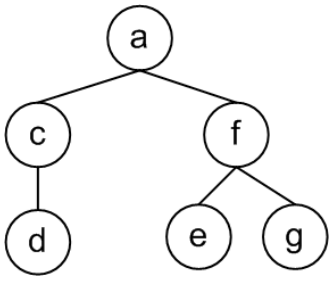


Figura 27. Exemplo de árvore binária.

Tipicamente a implementação das estruturas de dados em árvore envolve recursão tanto para a definição de atributos como para a implementação de algoritmos de caminhamento (***traversal algorithms***).

Exemplos de implementação de algoritmos de caminhamento em árvore binária podem ser encontrados no seguinte repositório:

https://github.com/fabriciogmc/data_structures_source_lectures_typescript/tree/main/arvores

5.5. ÁRVORES N-ÁRIAS

As generalizações para o caso das árvores binárias são as árvores ternárias, quaternárias, etc. Os casos mais gerais são ditos árvores n-árias. Para as árvores n-árias, cada vértice possui grau menor ou igual a (n-1).

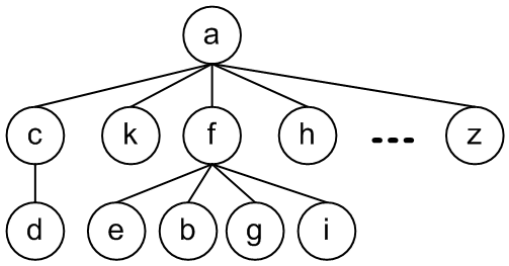


Figura 28. Exemplo de árvore n-ária.

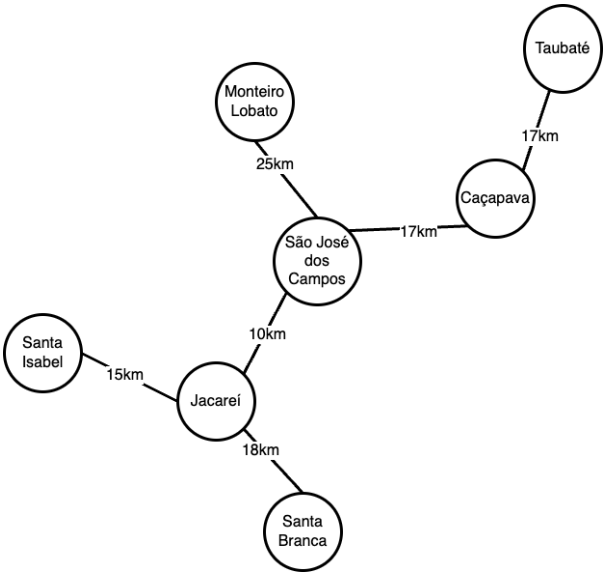
Uma das aplicações mais populares das árvores n-árias são os dicionários e os mecanismos de indexação para busca na Internet.

EXERCÍCIOS

OBS: Em todos os exercícios resolvidos, utilizar as implementações fornecidas pelo professor e fazer as devidas adaptações no mesmo arquivo. Não utilizar pacotes de software, bibliotecas, etc., prontos. Implementar o algoritmo 'do zero', exceto pelas funções de leitura e escrita em arquivo.

PRÁTICA DE ALGORITMOS E PROGRAMAÇÃO

P.5.1. Desenvolva um programa que represente, utilizando grafo com matriz de adjacências, a seguinte configuração representativa da distância entre os centros de 7 cidades:



P.5.2. Desenvolva um programa que ordene um vetor em ordem decrescente e que faça uso de uma árvore binária com algoritmo de inserção e caminhamento esquerda-raiz-direita.

TERMINOLOGIA E CONCEITOS

TC.5.3. Desenhe os gráficos K_4 e K_5.

TC.5.4. Escreva como ficaria a representação dos gráficos K2 e K3 utilizando:

a) Matriz de Adjacências

b) Dicionários/documentos

REFERÊNCIAS BIBLIOGRÁFICAS

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

[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.

[5] ALGORITMOS ORDENAÇÃO: DANÇAS FOLCLÓRICAS:

Bubble sort:

https://www.youtube.com/watch?v=Iv3vgjM8Pv4

Merge sort:

https://www.youtube.com/watch?v=dENca26N6V4

Insertion sort:

https://www.youtube.com/watch?v=EdIKIf9mHk0

Quick sort:

https://www.youtube.com/watch?v=3San3uKKHgg

Selection sort:

https://www.youtube.com/watch?v=Ns4TPTC8whw

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
```