Estratégia Concursos

# Aula 00

*CNU (Bloco 2 - Tecnologia, Dados e Informação) Passo de Conhecimentos Específicos - Eixo Temático 4 - Desenvolvimento de Software - 2024 (Pós-Edital)*

Autor:

05 de Março de 2024

# Lógica de Programação

Sumário



## Conteúdo

Lógica de Programação. Conceitos básicos, Tipos de dados, Estruturas de decisão, Estruturas de seleção, Funções e Procedimentos, Recursividade.

# Análise Estatística

Inicialmente, convém destacar o percentual de incidência do assunto, dentro da disciplina **Desenvolvimento e Programação de Sistemas** em concursos/cargos similares. Quanto maior o percentual de cobrança de um dado assunto, maior sua importância.

Obs.: *um mesmo assunto pode ser classificado em mais de um tópico devido à multidisciplinaridade de conteúdo*.

| Assunto | Relevância na disciplina em concursos similares |
|---|---|
| Linguagens de programação | **38.2 %** |
| 1. Python | **14.5 %** |
| 2. JavaScript | **12.7 %** |
| 3. Java | **3.6 %** |
| 4. VB Script | **1.8 %** |
| 5. C Sharp | **1.8 %** |
| 6. R | **1.8 %** |
| Web | **5.5 %** |
| Linguagens de marcação | **5.5 %** |
| 1. XML (Extensible Markup Language) | **3.6 %** |
| 2. HTML (HyperText Markup Language) | **1.8 %** |
| CSS (Cascading Style Sheets) | **3.6 %** |
| Conceitos básicos de programação | **1.8 %** |
| Frameworks Java | **1.8 %** |
| 1. Hibernate | **1.8 %** |

# GLOSSÁRIO DE TERMOS

*Faremos uma lista de termos que são relevantes ao entendimento do assunto desta aula. Caso tenha alguma dúvida durante a leitura, esta seção pode lhe ajudar a esclarecer.*

Programação: A atividade de escrever código para criar programas de computador que implementam desejos ou instruções específicas.

Lógica de Programação: O processo de usar uma linguagem de programação para resolver problemas. Isso envolve a aplicação de sequências lógicas e estruturas de controle, como loops e condicionais.

Algoritmo: Uma sequência finita e bem definida de passos que são seguidos para resolver um problema ou realizar uma tarefa.

Linguagem de Programação: Uma linguagem formal usada para expressar instruções de computador que podem ser traduzidas em código de máquina e executadas por um computador.

Pseudocódigo: Uma representação de alto nível de um algoritmo ou programa de computador que utiliza a estrutura de uma linguagem de programação, mas é destinado para leitura humana ao invés de execução de máquina.

Variável: Um local de memória nomeado usado para armazenar um valor que pode ser alterado durante a execução de um programa.

Constante: Um valor que não muda durante a execução de um programa.

Expressão: Uma combinação de variáveis, constantes, funções e operadores que produz um valor quando avaliada.

Comando: Uma instrução que o computador pode interpretar e executar.

Tipo de dado elementar: Um tipo de dado básico que não pode ser decomposto em tipos de dados mais simples. Exemplos incluem inteiros, reais, caracteres e lógicos.

Tipo de dado estruturado: Um tipo de dado complexo que é construído a partir de tipos de dados elementares. Exemplos incluem arrays, registros, listas e árvores.

Operador aritmético: Um símbolo que representa uma operação matemática específica, como adição (+), subtração (-), multiplicação (*) e divisão (/).

Operador lógico: Um símbolo que representa uma operação lógica, como E (AND), OU (OR) e NÃO (NOT).

Estrutura de decisão: Uma instrução de controle em programação que permite ao computador executar diferentes ações com base em uma condição, como IF-THEN-ELSE e SWITCH-CASE.

Estrutura de repetição: Uma instrução de controle em programação que permite ao computador executar o mesmo bloco de código várias vezes, como FOR, WHILE e DO-WHILE.

Função: Um bloco de código nomeado que realiza uma tarefa específica e retorna um valor.

Procedimento: Semelhante a uma função, mas não retorna um valor.

Recursão: O processo de uma função ou procedimento se chamar a si mesmo para resolver um problema que pode ser dividido em subproblemas menores de mesma natureza.

# ROTEIRO DE REVISÃO

*A ideia desta seção é apresentar um roteiro para que você realize uma revisão completa do assunto e, ao mesmo tempo, destacar aspectos do conteúdo que merecem atenção.*

## Introdução

Programação é o processo de escrever, testar, depurar e manter o código-fonte de programas computacionais. O código-fonte é escrito em uma linguagem de programação e ele é formado por um conjunto de instruções que são usadas para produzir vários tipos de saída. Programação envolve atividades como análise, desenvolvimento de algoritmos, verificação da precisão de algoritmos, codificação de

algoritmos em uma linguagem de programação específica e teste, depuração e manutenção do código.

A lógica de programação é importante na programação porque é a base para a definição do processo a ser seguido para resolver um problema. Ela permite expressar soluções de problemas de uma maneira que um computador possa interpretar e executar. Também nos ajuda a entender e analisar a sequência de operações e a avaliar condições para determinar a ordem de execução das instruções.

Por fim, algoritmo é uma sequência de passos para resolver um problema específico em um número finito de etapas. Os algoritmos são a base de qualquer processo de resolução de problemas relacionado à programação. Eles podem ser expressos em quase qualquer linguagem, desde a linguagem natural até linguagens de programação. O algoritmo é uma parte fundamental da lógica de programação e fornece a base sobre a qual o código de um programa é construído, e pode ser pensado como uma "receita de bolo" para a construção de programas.


Pseudocódigo e Fluxogramas

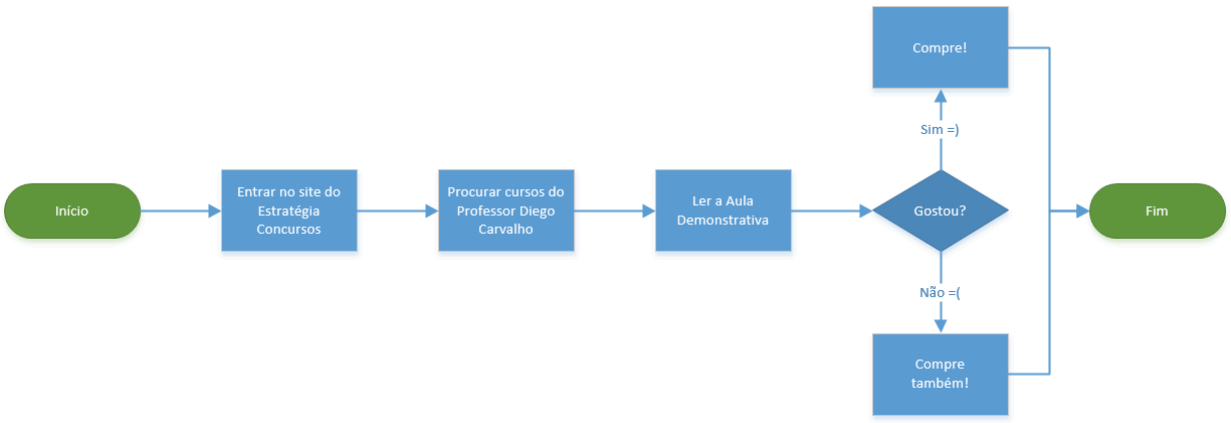
Pseudocódigo é uma descrição de alto nível de um algoritmo que usa as convenções estruturais de uma linguagem de programação, mas destina-se a ser legível para humanos, não para uma máquina. Ele é utilizado para planejar e visualizar a lógica de um algoritmo antes de implementá-lo em uma linguagem de programação específica.

O pseudocódigo não é executado em computadores. Em vez disso, é usado por programadores para esboçar a estrutura do programa e descrever suas características de maneira simples. Ele não segue a sintaxe de nenhuma linguagem de programação específica, o que o torna acessível para qualquer pessoa, independentemente da linguagem de programação que conheça.

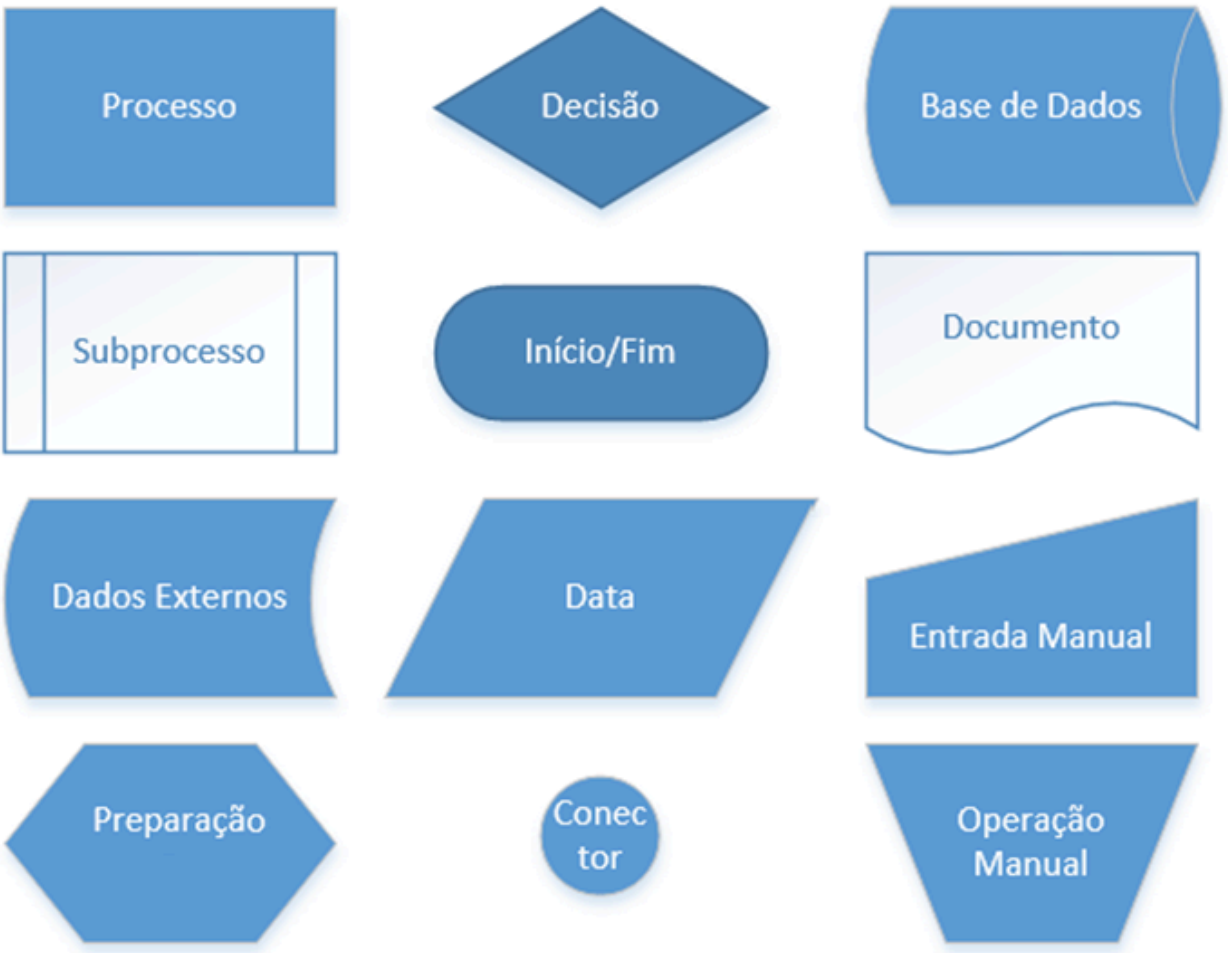
Por exemplo, um algoritmo simples para somar dois números em pseudocódigo poderia ser:

```
Início
    Receber número1
    Receber número2
    soma = número1 + número2
    Mostrar soma
Fim
```

Uma outra maneira de representar o fluxo de um programa ou algoritmo é por meio de Fluxogramas. Um fluxograma é uma espécie de diagrama utilizado para documentar processos, ajudando o leitor a visualizá-los, compreendê-los mais facilmente e encontrar falhas ou problemas de eficiência, como mostra a imagem abaixo:




Os principais símbolos de um Fluxograma são:




Podemos, então, representar algoritmos por meio de **Linguagens de Programação, Pseudocódigos ou Fluxogramas**! Em geral, os fluxogramas são mais utilizados para leigos; pseudocódigo para usuários um pouco mais avançados; e linguagens de programação para os avançados.

## Variáveis, Constantes, Expressões e Comandos

Entre as estruturas fundamentais de programação estão variáveis, constantes, expressões e comandos em geral. Vamos estudar esses conceitos em maiores detalhes.

### Variáveis

Variáveis são locais de armazenamento na memória do computador. Cada variável possui um nome único (identificador) e um tipo de dados, que determina o tipo de informação que a variável pode conter, como números inteiros, reais, caracteres ou booleanos. Variáveis são chamadas assim porque o valor que elas armazenam pode variar ao longo do tempo.

Por exemplo, em pseudocódigo, poderíamos ter:

```
numero = 10
```

O conteúdo de uma variável pode ser alterado, consultado ou apagado diversas vezes durante a execução de um algoritmo, porém o valor apagado é perdido.

## Constantes

Constantes são semelhantes às variáveis no sentido de que também são locais de armazenamento na memória do computador. No entanto, ao contrário das variáveis, o valor de uma constante não muda uma vez que é definido.

Por exemplo, em pseudocódigo, poderíamos ter:

```
const PI = 3.14
```

Constantes são dados que simplesmente não variam com o tempo, i.e., possuem sempre um valor fixo invariável. Por exemplo: a constante matemática π é (sempre foi e sempre será) igual a 3.141592 – esse valor não mudará.

## Expressões e Comandos

Expressões são combinações de uma ou mais constantes, variáveis, operadores e funções que a linguagem de programação interpreta (calcula) para produzir outro valor. Por exemplo, a expressão "5 * (numero + 3)" é uma expressão que envolve o número 5, a variável "numero", o número 3 e os operadores de adição e multiplicação.

Comandos são instruções que especificam ação. A ação pode ser aritmética, como adição ou multiplicação, pode ser manipulação de dados, como atribuir um valor a uma variável, ou pode ser de controle, como comandos de decisão (se, então, senão)

ou comandos de loop (para, enquanto). Um programa é, em essência, uma lista de comandos a serem executados em ordem.

Por exemplo, em pseudocódigo, um comando de decisão pode ser:

```
se (numero > 10) então
    mostrar "O número é maior que 10"
senão
    mostrar "O número é menor ou igual a 10"
fim
```

## Tipos de Dados

Em geral, tipos de dados se dividem em dois grupos: Dados Elementares e Dados Estruturados. Os dados elementares também podem ser chamados de simples, básicos, nativos ou primitivos. Já os dados estruturados também podem ser chamados de compostos.

### Tipos de dados elementares

Tipos de dados elementares são os blocos de construção básicos com os quais os programas de computador manipulam a informação. Eles representam valores simples que, normalmente, correspondem diretamente a tipos de dados que a máquina subjacente manipula. Tipos de dados elementares comuns incluem:

1. Inteiro: Representa números inteiros, que podem ser tanto positivos quanto negativos. Por exemplo, -10, 0, 5, 100.
2. Real: Representa números de ponto flutuante, ou seja, números que possuem uma parte fracionária. Por exemplo, 3.14, -0.01, 0.0, 100.12.
3. Caractere: Representa um único caractere. Isso pode incluir letras, números e símbolos. Por exemplo, 'a', 'Z', '3', '?'.
4. Lógico: Representa valores verdadeiros ou falsos (também conhecidos como booleanos). Em muitas linguagens de programação, esses são representados como 'true' (verdadeiro) ou 'false' (falso).

Tipos de dados estruturados

Tipos de dados estruturados, por outro lado, são tipos de dados mais complexos que são construídos a partir dos tipos de dados elementares. Eles podem representar uma coleção ou estrutura de valores. Exemplos comuns de tipos de dados estruturados incluem:

1. Cadeia de caracteres (String): Uma string é uma sequência de caracteres. Ela é usada para representar e manipular texto. Por exemplo, "Olá, Mundo!" é uma string.

Além das strings, existem outros tipos de dados estruturados, como:

1. Arrays (Vetores): Um array é uma coleção ordenada de elementos (geralmente do mesmo tipo), acessíveis por um índice.
2. Listas: Semelhantes aos arrays, mas com operações mais flexíveis, como inserção e remoção de elementos.
3. Dicionários (ou Mapas): Uma coleção de pares chave-valor que permite a recuperação eficiente de um valor com base em sua chave.
4. Registros (ou Estruturas): Coleção de valores, potencialmente de diferentes tipos, agrupados em uma única unidade.

Cada linguagem de programação suporta diferentes tipos de dados e fornece mecanismos para criar novos tipos de dados a partir dos existentes.

## Operadores

Operadores geralmente são divididos em Aritméticos, Relacionais e Lógicos.

Operadores Aritméticos: são utilizados para obter resultados numéricos, preocupando-se com a priorização[1].

| Operador | Símbolo | Prioridade |
|---|---|---|
| Multiplicação | * | 2º |
| Divisão | / | 2º |

[1] Em operadores que possuem a mesma prioridade, o que aparecer primeiro deve ser priorizado! Além disso, parênteses possuem sempre a maior prioridade!

| | | |
|---|---|---|
| Adição | + | 3º |
| Subtração | - | 3º |
| Exponenciação | ^ | 1º |

Operadores Relacionais: são utilizados para comparar números e literais, retornando valores lógicos.

| Operador | Símbolo |
|---|---|
| Igual a | = |
| Diferente de | <> ou != |
| Maior que | > |
| Menor que | < |
| Maior ou igual a | >= |
| Menor ou igual a | <= |

Operadores Lógicos: servem para combinar resultados de expressões, retornando valores lógicos (verdadeiro ou falso).

| Operador/Símbolo |
|---|
| E/And |
| Ou/Or |
| Não/Not |

# Estruturas de Decisão (Seleção)

Estruturas de decisão permitem que um programa decida qual bloco de instruções executar com base em uma condição lógica. Elas ajudam a implementar a lógica do programa de maneira controlada. As duas estruturas de decisão mais comuns são IF/ELSE e SWITCH/CASE.

1. IF, ELSE

A estrutura de decisão IF, ELSE é usada quando queremos escolher entre duas opções de execução - um bloco de instruções será executado se a condição for verdadeira e outro bloco será executado se a condição for falsa.

Veja um exemplo em pseudocódigo:

```
se (condição) então
    // Bloco de código a ser executado se a condição for verdadeira
senão
    // Bloco de código a ser executado se a condição for falsa
fim
```

2. SWITCH/CASE

A estrutura SWITCH/CASE é usada quando queremos escolher entre várias opções de execução. A expressão dentro da instrução SWITCH é avaliada uma vez, e o seu valor é comparado com os valores de cada CASE. Se houver uma correspondência, o bloco de instruções correspondente é executado.

Veja um exemplo em pseudocódigo:

```
escolher (expressão)
    caso valor1:
        // Bloco de código a ser executado se a expressão for igual a valor1
    caso valor2:
        // Bloco de código a ser executado se a expressão for igual a valor2
    ...
    caso contrário:
        // Bloco de código a ser executado se a expressão não for igual a ner
fim
```

Perceba que a instrução SWITCH/CASE é uma alternativa mais limpa para um longo encadeamento de IF, ELSE IF quando estamos comparando a mesma variável ou expressão com diferentes valores.

## Estruturas de Repetição

Estruturas de repetição, também conhecidas como loops, permitem que um bloco de código seja executado várias vezes, enquanto uma determinada condição for verdadeira. Essas estruturas são fundamentais na programação, pois muitas operações precisam ser repetidas até que uma certa condição seja atingida. Aqui estão as três estruturas de repetição mais comuns:

1. WHILE

A estrutura de repetição WHILE executa um bloco de código enquanto uma determinada condição for verdadeira. Se a condição for falsa quando o loop WHILE for alcançado, o bloco de código dentro do loop não será executado.

Por exemplo:

```
enquanto (condição) faça
    // Bloco de código a ser executado enquanto a condição for verdadeira
fim
```

2. DO-WHILE

A estrutura DO-WHILE é semelhante à estrutura WHILE, mas com uma diferença importante: o bloco de código dentro do loop DO-WHILE será executado pelo menos uma vez, porque a condição só é avaliada após a execução do bloco de código.

Por exemplo:

```
faça
    // Bloco de código a ser executado
enquanto (condição)
```

3. FOR

A estrutura de repetição FOR é usada quando sabemos exatamente quantas vezes queremos que um bloco de código seja executado. O loop FOR geralmente é usado para iterar através de uma sequência de valores, como elementos de uma matriz ou intervalo de números.

```
para i = 1 até 10 faça
    // Bloco de código a ser executado 10 vezes
fim
```

Em todos esses exemplos, é crucial garantir que a condição do loop eventualmente se torne falsa. Caso contrário, o loop continuará infinitamente, resultando em um erro conhecido como loop infinito.

## Funções e Procedimentos

Funções e procedimentos são ambos subprogramas usados na programação para agrupar um conjunto de instruções que realizam uma tarefa específica. Eles ajudam a melhorar a estrutura do código e a reutilização de código, além de tornar o programa mais fácil de entender e gerenciar. A principal diferença entre eles reside em como eles tratam a saída.

Funções

Uma função é um subprograma que recebe zero ou mais valores (chamados parâmetros) e retorna um valor. As funções são normalmente usadas quando uma operação precisa ser realizada várias vezes e pode produzir um resultado diferente dependendo dos valores de entrada.

Por exemplo:

```
função calcularMédia(numero1, numero2)
    retorno (numero1 + numero2) / 2
fim
```

Neste exemplo, a função "calcularMédia" recebe dois números como parâmetros e retorna a média deles.

Procedimentos

Um procedimento, por outro lado, é um subprograma que realiza uma tarefa, mas não retorna um valor. Um procedimento pode ainda receber parâmetros, que são valores de entrada que afetam a execução do procedimento.

Por exemplo, considere um procedimento que imprime uma mensagem de boas-vindas a um usuário:

```
procedimento mostrarBoasVindas(nome)
    mostrar "Olá, " + nome + "! Bem-vindo ao nosso sistema."
fim
```

Neste exemplo, o procedimento "mostrarBoasVindas" recebe um parâmetro "nome" e imprime uma mensagem de boas-vindas, mas não retorna nenhum valor.

A principal diferença entre funções e procedimentos é que **as funções retornam um valor e os procedimentos não**. Ambos podem receber parâmetros para influenciar seu comportamento.

## Passagem de parâmetros: por Valor ou por Referência

Quando chamamos funções ou procedimentos, geralmente passamos informações para eles na forma de parâmetros. A maneira como esses parâmetros são passados pode ter implicações significativas em nossos programas. Existem duas maneiras principais de passar parâmetros: por valor e por referência.

Passagem por Valor

Na passagem por valor, uma cópia do valor do argumento é passada para o parâmetro da função. Isso significa que modificar o parâmetro dentro da função não afeta o argumento original.

Aqui está um exemplo em pseudocódigo:

```
procedimento alterarValor(x)
    x = 10
fim

variável a = 5
alterarValor(a)
mostrar a // Ainda será 5, pois a mudança na função não afeta 'a'
```

Neste exemplo, o valor de 'a' é passado para a função alterarValor. Dentro da função, 'x' é alterado para 10, mas isso não afeta o valor de 'a', pois 'x' é apenas uma cópia de 'a'. Portanto, quando 'a' é exibido, ainda é 5.

Passagem por Referência

Na passagem por referência, o endereço da variável (não o valor real) é passado para a função. Isso significa que a função tem acesso à localização real do argumento na memória, e quaisquer mudanças feitas no parâmetro da função afetarão o argumento original.

Aqui está um exemplo em pseudocódigo:

```
procedimento alterarValor(ref x)
    x = 10
fim

variável a = 5
alterarValor(a)
mostrar a // Agora será 10, pois a mudança na função afeta 'a'
```

Neste exemplo, uma referência a 'a' é passada para a função alterarValor. Isso significa que quando 'x' é alterado para 10 dentro da função, 'a' também é alterado, pois 'x' é uma referência para 'a'. Portanto, quando 'a' é exibido, é 10.

# Recursividade

A recursividade é um conceito fundamental na programação e na matemática onde uma função ou procedimento se chama a si mesmo. Uma função recursiva resolve um problema grande dividindo-o em subproblemas menores que são mais gerenciáveis. O ponto crítico de uma função recursiva é ter uma condição de término para evitar que ela entre em um loop infinito.

Por exemplo, aqui está uma função recursiva simples em pseudocódigo que calcula o fatorial de um número:

```
função fatorial(n)
    se n == 0 então
        retorno 1
    senão
        retorno n * fatorial(n - 1)
    fim
fim
```

Neste exemplo, a função fatorial chama a si mesma para calcular o fatorial de n - 1, multiplicando o resultado por n. A condição de término é quando n é igual a 0, nesse caso a função retorna 1.

Exemplos de Recursividade na Vida Real

1. Fractais: Fractais são formas que se repetem infinitamente em diferentes escalas. Cada parte do fractal é uma cópia reduzida do todo. Este é um exemplo de recursividade na natureza.
2. Ancestrais: Se você considerar a questão "Quem são seus antepassados?", isso é recursivo por natureza. Seus antepassados são seus pais, os antepassados dos seus pais, os antepassados dos antepassados dos seus pais, e assim por diante.
3. Bonecas Matryoshka: As famosas bonecas russas que se abrem para revelar uma boneca menor dentro, que por sua vez se abre para revelar uma boneca ainda menor, e assim por diante, é um exemplo simples de recursividade. Cada boneca é um problema menor do mesmo tipo que a boneca original.

# Linguagem de Programação

Uma linguagem de programação é uma linguagem formal projetada para expressar instruções que podem ser traduzidas em código de máquina e executadas por um computador ou outra máquina. Elas permitem que programadores especifiquem precisamente quais operações o computador deve realizar.

Linguagens de programação podem ser classificadas em vários tipos, cada um com sua própria sintaxe e semântica:

1. Linguagens de Programação de Baixo Nível: Estas são mais próximas do código de máquina e são difíceis de ler e entender para os humanos. Elas permitem ao programador um controle detalhado sobre o hardware do computador. A linguagem Assembly é um exemplo disso.
2. Linguagens de Programação de Alto Nível: Estas são mais próximas das linguagens humanas e são mais fáceis de ler e escrever. Elas abstraem muitos detalhes do hardware do computador. Exemplos incluem Python, Java, C++, JavaScript, etc.
3. Linguagens de Programação Orientadas a Objetos: Estas permitem a modelagem de problemas do mundo real em termos de "objetos" que possuem dados (atributos) e comportamentos (métodos). Exemplos incluem Java, C++, Python, etc.
4. Linguagens de Programação Funcionais: Estas tratam a computação como a avaliação de funções matemáticas e evitam mudanças de estado e dados mutáveis. Exemplos incluem Haskell, Erlang, Clojure, etc.

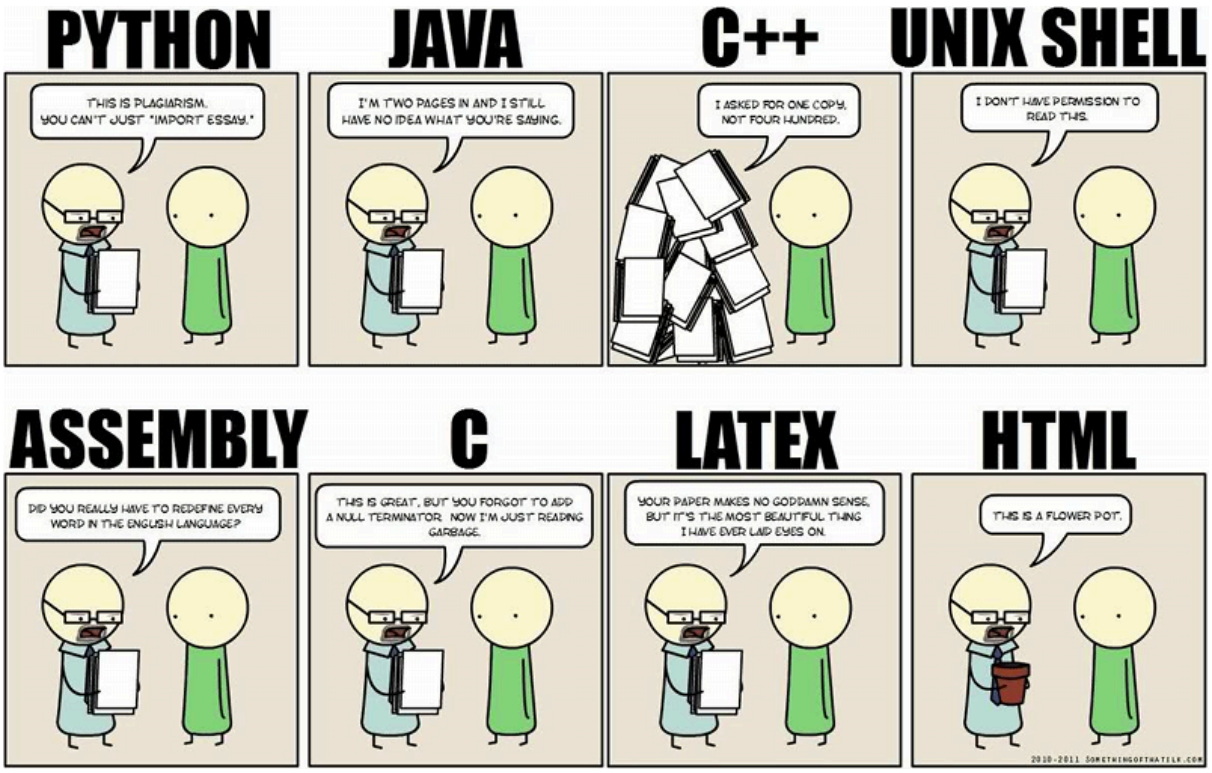

Um breve histórico de linguagens de programação

Linguagens de programação têm uma história rica que remonta aos primeiros dias da computação. Vamos explorar uma visão geral dos principais marcos.

1. Assembly (1949): As linguagens de Assembly foram as primeiras a serem criadas. Elas são consideradas linguagens de baixo nível porque estão muito próximas do hardware do computador. Cada instrução em Assembly corresponde diretamente a uma instrução em código de máquina, que é o que a CPU entende.
2. FORTRAN (1957): A primeira linguagem de alto nível foi a FORTRAN (FORmula TRANslation), desenvolvida pela IBM para computação científica. Ela introduziu uma sintaxe mais legível e a capacidade de expressar problemas científicos de forma mais direta.
3. COBOL (1959): COBOL (COmmon Business-Oriented Language) foi desenvolvido por um comitê de empresas e o governo dos EUA. Foi projetado para aplicações comerciais e é conhecido por sua legibilidade, facilitando a manutenção de programas.
4. LISP (1958): LISP (LISt Processing) foi criada por John McCarthy no MIT e é a segunda mais antiga linguagem de programação de alto nível ainda em uso hoje (depois de FORTRAN).
5. ALGOL (1958-1968): A família ALGOL (ALGOrithmic Language) teve um impacto significativo na ciência da computação e influenciou muitas outras linguagens, incluindo Pascal, C e Java.

6. C (1972): A linguagem C foi desenvolvida nos Bell Labs para o desenvolvimento do sistema operacional Unix. É conhecida por sua eficiência e controle de baixo nível sobre o hardware do computador.
7. Pascal (1970) e Ada (1980): Ambas foram criadas com o objetivo de incentivar boas práticas de programação e são conhecidas por sua clareza sintática e segurança.
8. C++ (1985): C++ foi uma extensão da linguagem C que adicionou suporte para programação orientada a objetos, permitindo a criação de software complexo e reutilizável.
9. Java (1995): Desenvolvida pela Sun Microsystems, a linguagem Java é orientada a objetos e foi projetada para ter poucas dependências de implementação, permitindo aos desenvolvedores "escrever uma vez, rodar em qualquer lugar".
10. Python (1991): Python é uma linguagem de alto nível conhecida por sua legibilidade e simplicidade. É usada em uma variedade de domínios, incluindo web, ciência de dados, aprendizado de máquina e muito mais.
11. JavaScript (1995): JavaScript foi originalmente desenvolvido para adicionar interatividade aos sites. Hoje, é usado tanto no front-end quanto no back-end do desenvolvimento web.
12. Swift (2014): Swift é uma linguagem de programação desenvolvida pela Apple para o desenvolvimento de aplicativos iOS e macOS.

Poderíamos citar várias outras linguagens (existem literalmente milhares!), mas essa lista dá uma boa ideia de como elas evoluíram em mais de sete décadas!



# QUESTÕES ESTRATÉGICAS

*Nesta seção, apresentamos e comentamos uma amostra de questões objetivas selecionadas estrategicamente: são questões com nível de dificuldade semelhante ao que você deve esperar para a sua prova e que, em conjunto, abordam os principais pontos do assunto.*

*A ideia, aqui, não é que você fixe o conteúdo por meio de uma bateria extensa de questões, mas que você faça uma boa revisão global do assunto a partir de, relativamente, poucas questões.*

1. (FGV – 2018 – Câmara de Salvador-BA) Expressões lógicas são frequentemente utilizadas em linguagens de programação. Por exemplo, um comando if com a expressão

if not (A and B)

pode ser reescrito, para quaisquer valores lógicos de A e B, com a expressão:

a) A or B
b) not A or not B
c) not A or B
d) not (not A or not B)
e) A and B

Comentários:
Neste caso, utilizamos a propriedad que é aplicar a negação às proposições, e inverter disjunção para conjunção, ou o inverso:

não (A e B) = não A ou não B

não (A ou B) = não A e não B

Portanto, not (A and B) = not A or not B.

Gabarito: Letra B.

2. (FGV – 2018 – Câmara de Salvador-BA) Observe o trecho de pseudocódigo exibido a seguir.

```
a := 1;
b := 3;
c := 5;
while b <> a and c < 20
{
     if a > c {
        c := c - 2
     }
     else {
          c := c + 2;
          if a + b < c {
               a := b - a;
               b := b + 2
          }
     }
}
print a, b, c;
```

Numa hipotética execução desse código, os valores exibidos seriam:

a) 2, 5, 7;
b) 6, 13, 15;
c) 6, 13, 19;
d) 7, 15, 21;
e) 7, 17, 23

Comentários:

Inicia o programa definindo os valores para as variáveis: a = 1, b = 3, c = 5

Chega no while.

Iteração: 1

Realiza o teste: b <> a and c < 20

3 <> 1 and 5 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 1 > 5

Falso. Entra no "else".

Realiza o teste: if a + b < c. 1 + 3 < 7.

Verdadeiro. Entra dentro do if.

a := b - a = 3 - 1 = 2

b := b + 2 = 3 + 2 = 5

Fim do loop. Valores: a = 2, b = 5, c = 7

Iteração: 2

Realiza o teste: b <> a and c < 20

5 <> 2 and 7 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 2 > 7

Falso. Entra no "else".

Realiza o teste: if a + b < c. 2 + 5 < 9.

Verdadeiro. Entra dentro do if.

a := b - a = 5 - 2 = 3

b := b + 2 = 5 + 2 = 7

Fim do loop. Valores: a = 3, b = 7, c = 9

Iteração: 3

Realiza o teste: b <> a and c < 20

7 <> 3 and 9 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 3 > 9

Falso. Entra no "else".

Realiza o teste: if a + b < c. 3 + 7 < 11.

Verdadeiro. Entra dentro do if.

a := b - a = 7 - 3 = 4

b := b + 2 = 7 + 2 = 9

Fim do loop. Valores: a = 4, b = 9, c = 11

Iteração: 4

Realiza o teste: b <> a and c < 20

9 <> 4 and 11 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 4 > 11

Falso. Entra no "else".

Realiza o teste: if a + b < c. 4 + 9 < 13.

Falso. Não entra dentro do if.

Fim do loop. Valores: a = 4, b = 9, c = 13

Iteração: 5

Realiza o teste: b <> a and c < 20

9 <> 4 and 13 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 4 > 13

Falso. Entra no "else".

Realiza o teste: if a + b < c. 4 + 9 < 15.

Verdadeiro. Entra dentro do if.

a := b - a = 9 - 4 = 5

b := b + 2 = 9 + 2 = 11

Fim do loop. Valores: a = 5, b = 11, c = 15

Iteração: 6

Realiza o teste: b <> a and c < 20

11 <> 5 and 15 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 5 > 15

Falso. Entra no "else".

Realiza o teste: if a + b < c. 5 + 11 < 17.

Verdadeiro. Entra dentro do if.

a := b - a = 11 - 5 = 6

b := b + 2 = 11 + 2 = 13

Fim do loop. Valores: a = 6, b = 13, c = 17

Iteração: 7

Realiza o teste: b <> a and c < 20

13 <> 6 and 17 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 6 > 17

Falso. Entra no "else".

Realiza o teste: if a + b < c. 6 + 13 < 19.

Falso. Não entra dentro do if.

Fim do loop. Valores: a = 6, b = 13, c = 19

Iteração: 8

Realiza o teste: b <> a and c < 20

13 <> 6 and 19 < 20

Verdadeiro. Entra dentro do loop.

if a > c {

Realiza o teste: 6 > 19

Falso. Entra no "else".

Realiza o teste: if a + b < c. 6 + 13 < 21.

Verdadeiro. Entra dentro do if.

a := b - a = 13 - 6 = 7

b := b + 2 = 13 + 2 = 15

Fim do loop. Valores: a = 7, b = 15, c = 21

Realiza o teste: b <> a and c < 20

15 <> 7 and 21 < 20. Falso

Fim do programa. Valores: a = 7, b = 15, c = 21

3. (FGV – 2018 – SEFIN-RO) Analise o trecho de pseudocódigo a seguir.

```
a := 2;
b := a * 10;
while a < 10 and b > 14
begin
  if a <> b
    begin
        if a > 5
            print (a, b)
        else
            a := a + 3;
    end
  else
    begin
        a := b - 2;
        print (a);
    end;
  a := a + 3
end;
```

Assinale a opção que exibe o conteúdo integral do resultado que seria produzido numa hipotética execução desse código.

a) 2 20
   5 20
   8 20
b) 2 20
c) 8 20

d) 2
5
8

e) 2 20
17
5 20
14
8 20
11

Comentários:

Vamos seguir o algoritmo.

Define o valor de a = 2

Depois, define o valor de b = a * 10 = 2*10 = 20.

Então, faz o teste do while.

Teste: a < 10 and b > 14. 2 < 10 and 20 > 14. Verdadeiro. Entra no while.

Depois, vem um if: if a <> b. Realiza o teste: a <> b. 2 <> 20.

Verdadeiro. Entra no if, e cai em um outro if: if a > 5. Realiza o teste: a > 5. 2 > 5.

Falso. Entra no "senão", e executa a = a + 3 = 2 + 3 = 5

Fora dos ifs, mas ainda dentro do loop, executa a = a + 3 = 5 + 3 = 8.

Fim do loop. Valores: a = 8, b = 20. Volta pro teste do while.

Teste: a < 10 and b > 14. 8 < 10 and 20 > 14. Verdadeiro. Entra no while.

Depois, vem um if: if a <> b. Realiza o teste: a <> b. 8 <> 20.

Verdadeiro. Entra no if, e cai em um outro if: if a > 5. Realiza o teste: a > 5. 8 > 5.

Verdadeiro. Entra no "então", e imprime "8 20"

Fora dos ifs, mas ainda dentro do loop, executa a = a + 3 = 8 + 3 = 11.

Fim do loop. Valores: a = 11, b = 20. Volta pro teste do while.

Falso. Sai do while.

Note que, em todo o processo, somente houve uma impressão: 8 20.

Gabarito: Letra C.

4. (FGV – 2018 – Prefeitura de Niterói – RJ) Sabendo-se que a função retorna o número de elementos de um array e que L assume o tipo de um array de inteiros, indexados a partir de zero, analise o pseudocódigo a seguir.

```
L := {10,2,40,53,28,12};
trocou := True;
while trocou {
    trocou := False;
    for k := 0 to len(L) - 2 {
        if L[k] > L[k+1] {
            L[k] = L[k+1];
            L[k+1] = L[k];
            trocou = True;
        }
    }
}
print L
```

Esse algoritmo deveria ordenar os elementos do array em ordem crescente, mas há problemas no código que produzem resultados errôneos. Assinale a opção que indica o que é de fato printado ao final da execução do código mostrado.

a) {10,2,40,53,28,12}
b) {2,10,12,28,40,53}
c) {53,40,28,12,10,2}
d) {2,2,12,12,12,12}
e) {2,10,10,10,10,12}

Comentários:
Primeiro, algumas considerações.

- As posições no array são contadas a partir do 0 neste algoritmo. Ou seja, a primeira posição é o 0, a segunda é o 1, e assim por diante.

- Era para o algoritmo ser o Bubble Sort, um famoso algoritmo de ordenação. O erro do algoritmo é não usar uma variável de troca na hora de inverter os valores.
  L[k] = L[k+1];
  L[k+1] = L[k];

  Deveria ser:

  aux = L[k];
  L[k] = L[k+1];
  L[k+1] = aux;

  Como resultado, ele acaba pegando o valor da próxima posição e colocando na atual; mas o valor da próxima posição se mantém intacto.

  Dito isso, vamos analisar o algoritmo passo-a-passo.

  Inicializa o array L = 10,2,40,53,28,12.

  Inicializa a variável trocou = true.

  Então, entra no while while trocou.

  Realiza o teste. trocou é true, então entra no while.

  Iteração do while número 1

  Define trocou = false. Agora, executa o for k := 0 to len(L) - 2. Vai rodar um loop armazenando na variável k a cada iteração o valor de 0 até 4.

  Para k = 0:

  Testa L[k] > L[k+1], ou seja, L[0] > L[1], ou seja, 10 > 2.

  Verdadeiro. Entra no if.

  L[k] = L[k+1], ou seja, L[0] = L[1], ou seja, L[0] = 2.

  L[k+1] = L[k], ou seja, L[1] = L[0], ou seja, L[1] = 10.

  Define trocou = true.

  Valor de L = 2,2,40,53,28,12.

  Para k = 1:

  Testa L[k] > L[k+1], ou seja, L[1] > L[2], ou seja, 2 > 40.

Falso. Não entra no loop.

Valor de L = 2,2,40,53,28,12.

Para k = 2:

Testa L[k] > L[k+1], ou seja, L[2] > L[3], ou seja, 40 > 53.

Falso. Não entra no loop.

Valor de L = 2,2,40,53,28,12.

Para k = 3:

Testa L[k] > L[k+1], ou seja, L[3] > L[4], ou seja, 53 > 28.

Verdadeiro. Entra no if.

L[k] = L[k+1], ou seja, L[3] = L[4], ou seja, L[3] = 28.

L[k+1] = L[k], ou seja, L[4] = L[3], ou seja, L[4] = 53.

Define trocou = true.

Valor de L = 2,2,40,28,28,12.

Para k = 4:

Testa L[k] > L[k+1], ou seja, L[4] > L[5], ou seja, 28 > 12.

Verdadeiro. Entra no if.

L[k] = L[k+1], ou seja, L[4] = L[5], ou seja, L[4] = 12.

L[k+1] = L[k], ou seja, L[5] = L[4], ou seja, L[5] = 28.

Define trocou = true.

Valor de L = 2,2,40,28,12,12.

Volta para o while.

Realiza o teste. trocou é true, então entra no while.

Iteração do while número 2

Define trocou = false. Agora, executa o for k := 0 to len(L) - 2. Vai rodar um loop armazenando na variável k a cada iteração o valor de 0 até 4.

Para k = 0:

Testa L[k] > L[k+1], ou seja, L[0] > L[1], ou seja, 2 > 2.

Falso. Não entra no loop.

Valor de L = 2,2,40,28,12,12.

Para k = 1:

Testa L[k] > L[k+1], ou seja, L[1] > L[2], ou seja, 2 > 40.

Falso. Não entra no loop.

Valor de L = 2,2,40,28,12,12.

Para k = 2:

Testa L[k] > L[k+1], ou seja, L[2] > L[3], ou seja, 40 > 28.

Verdadeiro. Entra no if.

L[k] = L[k+1], ou seja, L[2] = L[3], ou seja, L[2] = 28.

L[k+1] = L[k], ou seja, L[3] = L[2], ou seja, L[3] = 40.

Define trocou = true.

Valor de L = 2,2,28,28,12,12.

Para k = 3:

Testa L[k] > L[k+1], ou seja, L[3] > L[4], ou seja, 28 > 12.

Verdadeiro. Entra no if.

L[k] = L[k+1], ou seja, L[3] = L[4], ou seja, L[3] = 12.

L[k+1] = L[k], ou seja, L[4] = L[3], ou seja, L[4] = 28.

Define trocou = true.

Valor de L = 2,2,28,12,12,12.

Para k = 4:

Testa L[k] > L[k+1], ou seja, L[4] > L[5], ou seja, 12 > 12.

Falso. Não entra no loop.

Valor de L = 2,2,28,12,12,12.

Volta para o while.

Realiza o teste. trocou é true, então entra no while.

Iteração do while número 3

Define trocou = false. Agora, executa o for k := 0 to len(L) - 2. Vai rodar um loop armazenando na variável k a cada iteração o valor de 0 até 4.

Para k = 0:

Testa L[k] > L[k+1], ou seja, L[0] > L[1], ou seja, 2 > 2.

Falso. Não entra no loop.

Valor de L = 2,2,28,12,12,12.

Para k = 1:

Testa L[k] > L[k+1], ou seja, L[1] > L[2], ou seja, 2 > 28.

Falso. Não entra no loop.

Valor de L = 2,2,28,12,12,12.

Para k = 2:

Testa L[k] > L[k+1], ou seja, L[2] > L[3], ou seja, 28 > 12.

Verdadeiro. Entra no if.

L[k] = L[k+1], ou seja, L[2] = L[3], ou seja, L[2] = 12.

L[k+1] = L[k], ou seja, L[3] = L[2], ou seja, L[3] = 28.

Define trocou = true.

Valor de L = 2,2,12,12,12,12.

Para k = 3:

Testa L[k] > L[k+1], ou seja, L[3] > L[4], ou seja, 12 > 12.

Falso. Não entra no loop.

Valor de L = 2,2,12,12,12,12.

Para k = 4:

Testa L[k] > L[k+1], ou seja, L[4] > L[5], ou seja, 12 > 12.

Falso. Não entra no loop.

Valor de L = 2,2,12,12,12,12.

Volta para o while.

Realiza o teste. trocou é true, então entra no while.

Iteração do while número 4

Define trocou = false. Agora, executa o for k := 0 to len(L) - 2. Vai rodar um loop armazenando na variável k a cada iteração o valor de 0 até 4.

Para k = 0:

Testa L[k] > L[k+1], ou seja, L[0] > L[1], ou seja, 2 > 2.

Falso. Não entra no loop.

Valor de L = 2,2,12,12,12,12.

Para k = 1:

Testa L[k] > L[k+1], ou seja, L[1] > L[2], ou seja, 2 > 12.

Falso. Não entra no loop.

Valor de L = 2,2,12,12,12,12.

Para k = 2:

Testa L[k] > L[k+1], ou seja, L[2] > L[3], ou seja, 12 > 12.

Falso. Não entra no loop.

Valor de L = 2,2,12,12,12,12.

Para k = 3:

Testa L[k] > L[k+1], ou seja, L[3] > L[4], ou seja, 12 > 12.

Falso. Não entra no loop.

Valor de L = 2,2,12,12,12,12.

Para k = 4:

Testa L[k] > L[k+1], ou seja, L[4] > L[5], ou seja, 12 > 12.

Falso. Não entra no loop.

Valor de L = 2,2,12,12,12,12.

Note que, nesta iteração, ele não entrou no if nenhuma das vezes. Logo, não definiu trocou = true.

Então, ele volta pro while com a variável trocou = false, e então sai do while.

Valor final de L = 2,2,12,12,12,12.

Gabarito: Letra D.

5. (VUNESP – 2015 – TCE/SP – Analista de Sistemas) Um usuário implementou uma rotina de um programa, denominada Fatorial, e passou para essa rotina um parâmetro com o valor 6, mas deseja receber, após a execução da rotina, nesse mesmo parâmetro, o valor 6! (seis fatorial). Para isso, a passagem de parâmetro deverá ser por:

a) escopo.
b) hashing.
c) módulo.
d) referência.
e) valor.

Comentários:

Galera, observem a pegadinha da questão: ele manda 6 (seis) como parâmetro e no retorno da rotina o valor é 6! (seis fatorial). Observe que o valor foi modificado, logo não pode ter sido uma passagem por valor - foi uma passagem por referência. Caso o retorno fosse 6 (seis), a passagem provavelmente seria por valor.

Gabarito: Letra D

6. (VUNESP – 2014 – SP/URBANISMO – Analista Administrativo) Analise o algoritmo a seguir, apresentado na forma de uma pseudolinguagem (Português Estruturado). Esse algoritmo deverá ser utilizado para responder às questões.

```
Início
   Inteiro x1, x2, x3, i;
   Leia x1, x2, x3;
   Para i de 1 até x1 faça
   [
      x2 ← x1 + x3;
      x3 ← x1 – x2;
   ]
Imprima (x2 + x3);
Fim
```

Considere que os valores lidos para x1, x2 e x3 tenham sido, respectivamente, 5, 4 e 3. É correto afirmar que o valor impresso ao final da execução do algoritmo é igual a:
a) –3
b) 0
c) 5
d) 8
e) 11

Comentários:

Sabendo que x1 recebe 5, x2 recebe 4 e x3 recebe 3. Executa-se o loop do algoritmo da forma a seguir.
Quando i = 1, temos que:
x2 <- x1 + x3 = 5 + 3 = 8 (x1, x2, x3 = 5, 8, 3)
x3 <- x1 - x2 = 5 - 8 = -3 (x1, x2, x3 = 5, 8, -3)

Quando i = 2, temos que:
x2 <- x1 + x3 = 5 - 3 = 2 (x1, x2, x3 = 5, 2, -3)
x3 <- x1 - x2 = 5 - 2 = 3 (x1, x2, x3 = 5, 2, 3)

Quando i = 3, temos que:

x2 <- x1 + x3 = 5 + 3 = 8 (x1, x2, x3 = 5, 8, 3)
x3 <- x1 - x2 = 5 - 8 = -3 (x1, x2, x3 = 5, 8, -3)

Quando i = 4, temos que:
x2 <- x1 + x3 = 5 - 3 = 2 (x1, x2, x3 = 5, 2, 3)
x3 <- x1 - x2 = 5 - 2 = 3 (x1, x2, x3 = 5, 2, 3)

Quando i = 5, temos que:
x2 <- x1 + x3 = 5 + 3 = 8 (x1, x2, x3 = 5, 8, 3)
x3 <- x1 - x2 = 5 - 8 = -3 (x1, x2, x3 = 5, 8, -3)
Logo, x2 + x3 = 8 - 3 = 5.

Gabarito: Letra C

## QUESTIONÁRIO DE REVISÃO E APERFEIÇOAMENTO

*A ideia do questionário é elevar o nível da sua compreensão no assunto e, ao mesmo tempo, proporcionar uma outra forma de revisão de pontos importantes do conteúdo, a partir de perguntas que exigem respostas subjetivas.*

*São questões um pouco mais desafiadoras, porque a redação de seu enunciado não ajuda na sua resolução, como ocorre nas clássicas questões objetivas.*

*O objetivo é que você realize uma auto explicação mental de alguns pontos do conteúdo, para consolidar melhor o que aprendeu ;)*

*Além disso, as questões objetivas, em regra, abordam pontos isolados de um dado assunto. Assim, ao resolver várias questões objetivas, o candidato acaba memorizando pontos isolados do conteúdo, mas muitas vezes acaba não entendendo como esses pontos se conectam.*

*Assim, no questionário, buscaremos trazer também situações que ajudem você a conectar melhor os diversos pontos do conteúdo, na medida do possível.*

*É importante frisar que não estamos adentrando em um nível de profundidade maior que o exigido na sua prova, mas apenas permitindo que você compreenda melhor o assunto de modo a facilitar a resolução de questões objetivas típicas de concursos, ok?*

*Nosso compromisso é proporcionar a você uma revisão de alto nível!*

*Vamos ao nosso questionário:*

## Perguntas

1. O que é lógica de programação e por que é importante?
2. Como a pseudocódigo é usado na programação?
3. O que é uma variável em programação?
4. Qual a diferença entre uma constante e uma variável?
5. O que é uma estrutura de decisão e quais são os exemplos comuns?
6. Como uma estrutura de repetição funciona em programação?
7. Qual a diferença entre uma função e um procedimento?
8. Como a passagem de parâmetros por valor difere da passagem por referência?
9. O que é recursividade na programação?
10. Como um tipo de dado elementar difere de um tipo de dado estruturado?
11. Quais são os operadores aritméticos comuns em programação?
12. Como os operadores lógicos são usados em programação?
13. O que é uma linguagem de programação?
14. Qual é o propósito de uma linguagem de Assembly?
15. Como a linguagem de programação C influenciou as linguagens subsequentes?
16. Como a linguagem de programação Java difere das linguagens anteriores?
17. O que é uma expressão em programação?
18. O que é um comando em programação?

## Perguntas e Respostas

1. O que é lógica de programação e por que é importante?

Resposta: A lógica de programação é o processo de usar uma linguagem de programação para resolver problemas. Isso envolve a aplicação de sequências lógicas e estruturas de controle, como loops e condicionais. É importante porque é a base para a criação de programas de computador que executam tarefas desejadas.

2. Como a pseudocódigo é usado na programação?

Resposta: O pseudocódigo é uma representação de alto nível de um algoritmo ou programa de computador. Ele utiliza a estrutura de uma linguagem de programação, mas é destinado para leitura humana ao invés de execução de máquina. É frequentemente usado para planejar e comunicar a lógica de um algoritmo antes da implementação do código.

3. O que é uma variável em programação?

Resposta: Uma variável é um local de memória nomeado usado para armazenar um valor que pode ser alterado durante a execução de um programa.

4. Qual a diferença entre uma constante e uma variável?

Resposta: A diferença entre uma constante e uma variável é que o valor de uma constante não muda durante a execução de um programa, enquanto o valor de uma variável pode ser alterado.

5. O que é uma estrutura de decisão e quais são os exemplos comuns?

Resposta: Uma estrutura de decisão é uma instrução de controle em programação que permite ao computador executar diferentes ações com base em uma condição. Exemplos comuns incluem IF-THEN-ELSE e SWITCH-CASE.

6. Como uma estrutura de repetição funciona em programação?

Resposta: Uma estrutura de repetição é uma instrução de controle em programação que permite ao computador executar o mesmo bloco de código várias vezes. A execução continua até que uma condição especificada seja atendida. Exemplos comuns incluem FOR, WHILE e DO-WHILE.

7. Qual a diferença entre uma função e um procedimento?

Resposta: Ambos são blocos de código nomeados que realizam uma tarefa específica. A principal diferença é que uma função retorna um valor, enquanto um procedimento não.

8. Como a passagem de parâmetros por valor difere da passagem por referência?

Resposta: Na passagem por valor, uma cópia do valor é passada para a função ou procedimento. As alterações feitas no valor dentro da função ou procedimento não afetam o valor original. Na passagem por referência, um endereço de memória é passado, portanto, as alterações feitas no valor dentro da função ou procedimento afetam o valor original.

9. O que é recursividade na programação?

Resposta: A recursividade é o processo de uma função ou procedimento se chamar a si mesmo para resolver um problema que pode ser dividido em subproblemas menores de mesma natureza.

10. Como um tipo de dado elementar difere de um tipo de dado estruturado?

Resposta: Um tipo de dado elementar é um tipo de dado básico que não pode ser decomposto em tipos de dados mais simples, como inteiros, reais, caracteres e lógicos. Um tipo de dado estruturado é um tipo de dado complexo que é construído a partir de tipos de dados elementares, como arrays, registros, listas e árvores.

11. Quais são os operadores aritméticos comuns em programação?

Resposta: Os operadores aritméticos comuns em programação incluem adição (+), subtração (-), multiplicação (*) e divisão (/).

12. Como os operadores lógicos são usados em programação?

Resposta: Os operadores lógicos, como E (AND), OU (OR) e NÃO (NOT), são usados em programação para realizar operações lógicas. Eles são frequentemente usados em estruturas de decisão para combinar ou inverter condições.

13. O que é uma linguagem de programação?

Resposta: Uma linguagem de programação é uma linguagem formal usada para expressar instruções de computador que podem ser traduzidas em código de máquina e executadas por um computador.

14. Qual é o propósito de uma linguagem de Assembly?

Resposta: As linguagens de Assembly são consideradas linguagens de baixo nível porque estão muito próximas do hardware do computador. Cada instrução em Assembly corresponde diretamente a uma instrução em código de máquina, que é o que a CPU entende.

15. Como a linguagem de programação C influenciou as linguagens subsequentes?

Resposta: A linguagem C foi desenvolvida nos Bell Labs para o desenvolvimento do sistema operacional Unix. É conhecida por sua eficiência e controle de baixo nível sobre o hardware do computador. Muitas linguagens subsequentes, incluindo C++, C#, Objective-C e muitas outras, foram influenciadas ou diretamente baseadas na linguagem C.

16. Como a linguagem de programação Java difere das linguagens anteriores?

Resposta: Java é uma linguagem de programação orientada a objetos que foi projetada para ter poucas dependências de implementação, permitindo aos desenvolvedores "escrever uma vez, rodar em qualquer lugar". Isso significa que um programa Java deve ser capaz de ser executado em qualquer dispositivo ou sistema operacional que possua uma máquina virtual Java (JVM), sem a necessidade de recompilação.

17. O que é uma expressão em programação?

Resposta: Uma expressão é uma combinação de variáveis, constantes, funções e operadores que produz um valor quando avaliada.

18. O que é um comando em programação?

Resposta: Um comando é uma instrução que o computador pode interpretar e executar.

## Lista de Questões Estratégicas

1. (FGV – 2018 – Câmara de Salvador-BA) Expressões lógicas são frequentemente utilizadas em linguagens de programação. Por exemplo, um comando if com a expressão

if not (A and B)

pode ser reescrito, para quaisquer valores lógicos de A e B, com a expressão:

a) A or B
b) not A or not B
c) not A or B
d) not (not A or not B)
e) A and B

2. (FGV – 2018 – Câmara de Salvador-BA) Observe o trecho de pseudocódigo exibido a seguir.

```
a := 1;
b := 3;
c := 5;
while b <> a and c < 20
{
    if a > c {
       c := c - 2
    }
    else {
        c := c + 2;
        if a + b < c {
            a := b - a;
            b := b + 2
        }
    }
}
print a, b, c;
```

Numa hipotética execução desse código, os valores exibidos seriam:

a) 2, 5, 7;
b) 6, 13, 15;
c) 6, 13, 19;
d) 7, 15, 21;
e) 7, 17, 23

3. (FGV – 2018 – SEFIN-RO) Analise o trecho de pseudocódigo a seguir.

```
a := 2;
b := a * 10;
while a < 10 and b > 14
begin
  if a <> b
    begin
        if a > 5
            print (a, b)
        else
            a := a + 3;
    end
  else
    begin
        a := b - 2;
        print (a);
    end;
  a := a + 3
end;
```

Assinale a opção que exibe o conteúdo integral do resultado que seria produzido numa hipotética execução desse código.

a) 2 20
   5 20
   8 20
b) 2 20
c) 8 20
d) 2
   5
   8
e) 2 20
   17
   5 20
   14
   8 20
   11

4. (FGV – 2018 – Prefeitura de Niterói – RJ) Sabendo-se que a função retorna o número de elementos de um array e que L assume o tipo de um array de inteiros, indexados a partir de zero, analise o pseudocódigo a seguir.

```
L := {10,2,40,53,28,12};
trocou := True;
while trocou {
    trocou := False;
    for k := 0 to len(L) - 2 {
        if L[k] > L[k+1] {
            L[k] = L[k+1];
            L[k+1] = L[k];
            trocou = True;
        }
    }
}
print L
```

Esse algoritmo deveria ordenar os elementos do array em ordem crescente, mas há problemas no código que produzem resultados errôneos. Assinale a opção que indica o que é de fato printado ao final da execução do código mostrado.

a) {10,2,40,53,28,12}
b) {2,10,12,28,40,53}
c) {53,40,28,12,10,2}
d) {2,2,12,12,12,12}
e) {2,10,10,10,10,12}

5. (VUNESP – 2015 – TCE/SP – Analista de Sistemas) Um usuário implementou uma rotina de um programa, denominada Fatorial, e passou para essa rotina um parâmetro com o valor 6, mas deseja receber, após a execução da rotina, nesse mesmo parâmetro, o valor 6! (seis fatorial). Para isso, a passagem de parâmetro deverá ser por:

a) escopo.
b) hashing.
c) módulo.
d) referência.
e) valor.

6. (VUNESP – 2014 – SP/URBANISMO – Analista Administrativo) Analise o algoritmo a seguir, apresentado na forma de uma pseudolinguagem (Português Estruturado). Esse algoritmo deverá ser utilizado para responder às questões.

```
Início
   Inteiro x1, x2, x3, i;
   Leia x1, x2, x3;
   Para i de 1 até x1 faça
   [
      x2 ← x1 + x3;
      x3 ← x1 - x2;
   ]
Imprima (x2 + x3);
Fim
```

Considere que os valores lidos para x1, x2 e x3 tenham sido, respectivamente, 5, 4 e 3. É correto afirmar que o valor impresso ao final da execução do algoritmo é igual a:

a) –3

b) 0
c) 5
d) 8
e) 11

## Gabaritos

1. B
2. D
3. C
4. D
5. D
6. C

# ESSA LEI TODO MUNDO CONHECE:
# PIRATARIA É CRIME.

**Mas é sempre bom revisar o porquê e como você pode ser prejudicado com essa prática.**

1. 

Professor investe seu tempo para elaborar os cursos e o site os coloca à venda.

2. 

Pirata divulga ilicitamente (grupos de rateio), utilizando-se do anonimato, nomes falsos ou laranjas (geralmente o pirata se anuncia como formador de "grupos solidários" de rateio que não visam lucro).

3. 

Pirata cria alunos fake praticando falsidade ideológica, comprando cursos do site em nome de pessoas aleatórias (usando nome, CPF, endereço e telefone de terceiros sem autorização).

4. 

Pirata compra, muitas vezes, clonando cartões de crédito (por vezes o sistema anti-fraude não consegue identificar o golpe a tempo).

5. 

Pirata fere os Termos de Uso, adultera as aulas e retira a identificação dos arquivos PDF (justamente porque a atividade é ilegal e ele não quer que seus fakes sejam identificados).

6. 

Pirata revende as aulas protegidas por direitos autorais, praticando concorrência desleal e em flagrante desrespeito à Lei de Direitos Autorais (Lei 9.610/98).

7. 

Concurseiro(a) desinformado participa de rateio, achando que nada disso está acontecendo e esperando se tornar servidor público para exigir o cumprimento das leis.

8. 

O professor que elaborou o curso não ganha nada, o site não recebe nada, e a pessoa que praticou todos os ilícitos anteriores (pirata) fica com o lucro.

**Deixando de lado esse mar de sujeira, aproveitamos para agradecer a todos que adquirem os cursos honestamente e permitem que o site continue existindo.**