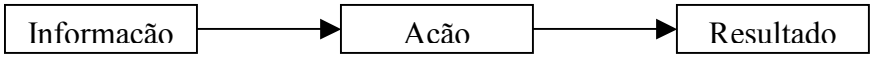
ALGORÍTMOS
INTRODUÇÃO A LÓGICA DE PROGRAMAÇÃO

1 - INTRODUÇÃO E DEFINIÇÃO

1.1 Introdução a Lógica de Programação

É um método pelo qual se aplica o fundamento do Raciocínio Lógico em desenvolvimento de programas de computador, fazendo uso ordenado dos elementos básicos suportados por um dado estilo de programação.

Usar o Raciocínio Lógico no desenvolvimento de programas consiste em:
- ✓ Ler atentamente o enunciado
- ✓ Retirar do enunciado a relação das entradas de dados
- ✓ Retirar do enunciado a relação das saídas de dados
- ✓ Determinar as ações que levarão a atingir o resultado desejado



1.2 Algoritmos

Podemos pensar num algoritmo como um "mecanismo" de transformação de entradas em saídas. Assim, um algoritmo ao ser "executado", receberá algumas entradas, que serão processadas e nos devolverá saídas esperadas.

ALGO*: *Lógica ***RÍTMO*: *Estudo*** → ***ALGORÍTMO* =** ESTUDO DA LÓGICA

➔ Aplicação computacional do Estudo da Lógica para se alcançar um objetivo.

➔ Linguagem em alto nível que permite ao usuário/programador preparar e programar o computador para resolver problemas através de uma seqüência lógica de ações que deverão ser executados passo-a-passo e que seguirão os conceitos da programação estruturada.

Um algoritmo é um texto estático, onde temos vários passos que são lidos e interpretados de cima para baixo. Para que venhamos a obter o(s) resultado(s) deste algoritmo, necessitamos "executá-lo", o que resulta em um processo dinâmico.

No fluxo de controle identificamos em cada passo da execução qual é o próximo comando a ser executado.

A compreensão da lógica de programação de um algoritmo está diretamente ligada a compreensão de seu fluxo de controle. A partir de uma compreensão correta, podemos traçar as diversas execuções possíveis de um algoritmo. Se testarmos todas essas possibilidades, e obtivermos resultados corretos, podemos ter certeza de estar entregando um produto final confiável.

Antes de escrever um algoritmo que deverá atingir uma solução, deve-se conhecer profundamente o problema em questão e também planejar como se deve resolve-lo.

Exemplo de uma solução para um problema:

Problema em questão: TRABALHAR
Ponto de Partida: DORMINDO
Solução em linguagem natural (alto nível)

1. ACORDAR — Ponto de partida do problema
2. SAIR DA CAMA
3. IR AO BANHEIRO
4. LAVAR O ROSTO E ESCOVAR OS DENTES
5. COLOCAR A ROUPA
6. SE CHOVER, ENTÃO USAR GUARDA-CHUVA, SENÃO USAR BONÉ
7. IR AO PONTO DE ONIBUS
8. ANDAR DE ONIBUS ATÉ ELE CHEGAR AO SERVIÇO
9. BATER O CARTÃO DE PONTO
10. COMEÇAR O EXPEDIENTE — Obtenção da solução

1.3 Método para desenvolvimento de algoritmos

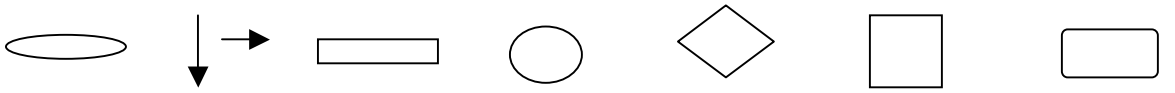
1. Faça uma leitura de todo o problema até o final, a fim de formar a primeira impressão. A seguir, releia o problema e faça anotações sobre os pontos principais.
2. Verifique se o problema foi bem entendido. Questione, se preciso, ao autor da especificação sobre suas dúvidas. Releia o problema quantas vezes for preciso para tentar entendê-lo.
3. Extraia do problema todas as suas entradas (informações, dados).
4. Extraia do problema todas as suas saídas (resultados).
5. Identifique qual é o processamento principal.
6. Verifique se será necessário algum valor intermediário que auxilie a transformação das entradas em saídas. Esta etapa pode parecer obscura no início, mas com certeza no desenrolar do algoritmo, estes valores aparecerão naturalmente.
7. Teste cada passo do algoritmo, com todos os seus caminhos para verificar se o processamento está gerando os resultados esperados.
8. Crie valores de teste para submeter ao algoritmo.
9. Reveja o algoritmo, checando as normas de criação.

1.4 O Pseudocódigo ou Português Estruturado ou Portugol

É uma linguagem natural e informal que ajuda os programadores a desenvolver o algoritmo a fim de se comunicar com a máquina. Consiste principalmente na utilização de uma seqüência estruturada de ações.

1.5 O Fluxograma ou Diagrama de Bloco

É a utilização de símbolos pré-definidos para representar o fluxo de dados durantes a execução dos procedimentos.



Inicio e Fim	Fluxo de dados	Sentenças e Comandos	Conectores de fluxos	Estrutura de Seleção	Estruturas de Repetição	Processo Alternativo

2. Estrutura de Controle

Também conhecida como programação estruturada. Divide-se em 3 estruturas:

- 2.1 Seqüencial
- 2.2 Seleção ou Condicional
- 2.3 Repetição

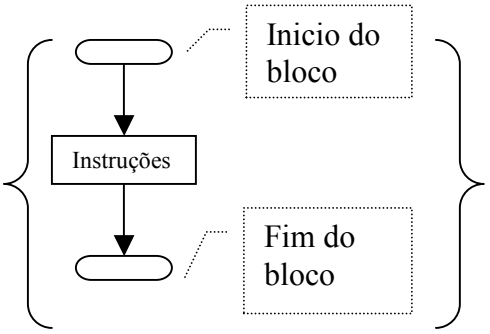
Esta estrutura não são independentes, por isso interagem entre si

2.1 Programação Estruturada Seqüencial

Uma programação seqüencial é aquela cujas ações são executadas uma após a outra (TOP-DOWN) e identificadas pelo início e fim de cada bloco de instruções.

Exemplo de Programação Seqüencial:

```
INICIO
  |
  {Corpo do Algoritmo}
  |
  v
FIM
```



FLUXOGRAMA

É sempre importante utilizar técnicas de parametrização e identação na hora de se escrever os algoritmos. Isto facilita na leitura de seus procedimentos e melhor apresenta sua organização.

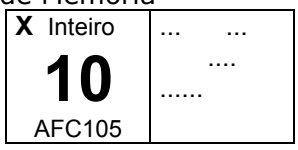
A Programação Estruturada é definida por uma seqüência de ações que seguem a seguinte composição:

- 2.1.1 Constantes e Variáveis
- 2.1.2 Símbolos
- 2.1.3 Operadores
- 2.1.4 Funções
- 2.1.5 Comandos
- 2.1.6 Sentenças

2.1.1 Constantes e Variáveis

Constantes - Vaz referências a dados que não sofrem alterações ao longo do algoritmo.
Variáveis - Vaz referências aos dados do problema e que deverão ser armazenadas na memória do computador e sofrem alterações ao longo do algoritmo. Toda variável deve ser classificada a um tipo exclusivo de informação pela qual a memória da máquina será preparada para armazenar o dado, e este tipo será incluído já no inicio dos procedimentos.

Modelo de uma Célula de Memória
Nome: X
Tipo: Inteiro
Conteúdo: 10
Endereço: AFC105



Modelo de armazenamento de variáveis na memória do computador

Composição de uma variável

- Nome da Variável:
  Nome que será utilizado para identificar a informação e determinar onde a mesma se encontra na memória

- Tipo de dados:
  INTEIRO → Dados numéricos com valores inteiros. Ex. IDADE
  REAL → Dados numéricos e que suportam casas decimais. Ex. ALTURA, PESO
  CARACTER → Dados texto (conjunto de caracteres não numéricos ou para fins numéricos) Ex. NOMES, ENDERECOS, CEP, PALAVRAS.
  DIMENSIONADAS → Conjunto dinâmico de dados armazenados em forma de tabelas ou seqüências limitadas homogêneas (VETORES e MATRIZES) ou heterogêneas (CAMPOS e REGISTROS)
  LÓGICOS → Assumem valores únicos e distintos: V (verdade), F (falso).

- Conteúdo:
  Valor atribuído à variável e que será colocado na memória.

Tabela de Operadores Lógicos		
V e V = V	V ou V = V	não V = F
V e F = F	V ou F = V	não F = V
F e V = F	F ou V = V	
F e F = F	F ou F = F	

- Endereço:
  Endereço físico de armazenamento que será gerenciado pelo sistema operacional do computador e este evitará que outras informações invadam a área restrita a uma variável

Declaração das Variáveis:

A declaração é dada sempre no topo do algoritmo.

Exemplo:

```
INICIO

    INTEIRO A, X, idade;
    REAL peso, L;
    CARACTER nome, cep, K;
    LOGICO resposta;
    VETOR notas[10];

FIM.
```

2.1.2 Símbolos

Representação por símbolos que o computador identifica e interpreta a fim de satisfazer um procedimento.

Os principais e mais utilizados são:

- ← atribuição de dados a variáveis
- (início de um conjunto dados
-) fim de um conjunto de dados
- " inicio e fim de uma expressão caracter
- // comentário
- ; terminadores de linhas de instruções
- . fim do algoritmo
- , separador de conjunto de dados

Exemplo:

```
INICIO
    // variáveis
        INTEIRO A, X, idade;
        REAL peso, L;
        CARACTER nome, cep, K;
        LOGICO resposta;
    // principal
        A ← 5;
        Peso ← 65.4;
        nome ← "FEPI" ;
        resposta ← falso;
FIM.
```

2.1.3 Operadores

Símbolos aritméticos que geram processamento e retornam resultados.
São classificados em:

- Aritméticos (cálculos e processamentos)

Operador	Descrição	Exemplo	Resultado
+	Adição	10 + 15	25
-	Subtração	20 – 10	10
*	Multiplicação	3 * 5	15
/	Divisão (resultado será um número real)	5 / 2	2,5
^, **	Exponenciação	5 ^ 2 ou 5**2	25

- Relacionais (equivalência ou igualdade entre dois ou mais valores)

Operador	Descrição	Exemplo	Resultado
=	Igualdade	10 = 15	Falso
>	Maioridade	10 > 15	Falso
<	Menoridade	10 < 15	Verdade
>=	Maioridade e Igualdade	10 >= 15	Falso
<=	Menoridade e Igualdade	10 <= 15	Verdade
<>	Diferenciação	10 <> 15	Verdade

Exemplos de uso:

```
INICIO
        // variáveis
                INTEIRO A, B, C;
        // principal
                A ← 5; B ← 4;
                C ← A + B;
                C > A;          ( condição verdadeira )
                B = A;          ( condição falsa )
                A <> B;         ( condição verdadeira )
FIM.
```

2.1.4 Funções

Rotinas matemáticas prontas no processador que geram processamento sobre um determinado valor especificado e retornam resultados.

Principais funções:

- SEN(x) → retorna o seno do valor especificado entre os parentes
- COS(x) → retorna o cosseno do valor especificado entre os parentes
- MOD(x) → retorna a o resto da divisão de um valor especificado por parentes
- ABS(x) → retorna o valor absoluto do valor especificado entre os parentes
- SQRT(x) → retorna a raiz quadrada do valor especificado entre os parentes

Exemplos de uso:

```
INICIO
// variáveis
        INTEIRO A, B, C;
// principal
        A ← 25;
        B ← -4;
        C ← SQRT(A);        ( c = 5 )
        C ← ABS(B);         ( c = 4 )
FIM.
```

2.1.5 Comandos

Palavras chaves ou instruções pré-definidas que são interpretadas pelo processador e passam produzir iterações entre o usuário e a máquina.

Principais comandos:

- LEIA(x) → permite ao usuário informar um valor para a variável durante a execução do algoritmo
- ESCREVA(x) → mostra na tela do computador o valor da variável em questão
- IMPRIMA(x) → envia para impressora o valor da variável em questão

Exemplo

```
INICIO
// variáveis
        INTEIRO A, B, C;
// principal
        Leia(A);
        Escreva(" Informe um valor para variável B: ");
        Leia(B);
        C ← A / B;
        Escreva(A);
        Escreva(" Resultado da divisão = ", C);
        Imprima(" O resultado da divisão de ",A, " por ", B, " é  ",C);

FIM.
```

2.1.6 Sentenças

É a mistura das todas as estruturas afim de se obter um resultado ou uma solução para um problema. Formam uma sentença o conjunto de regras, comandos, variáveis, funções e símbolos, agrupados de forma lógica permitindo que o computador possa processar e interagir com os dados.

Exemplo

```
INICIO
// variáveis
        INTEIRO A, B, C;
// principal
        Leia(A);
        Leia(B);
        C ← A / ABS(B);
        Escreva( "o valor de C é :" , C);
FIM.
```

Todas linhas de ações por mais simples que sejam, formam uma sentença.

Lista de exercícios...

Entregar em: _____/_____/_____

2.2 Programação Estruturada de Seleção ou Condicional

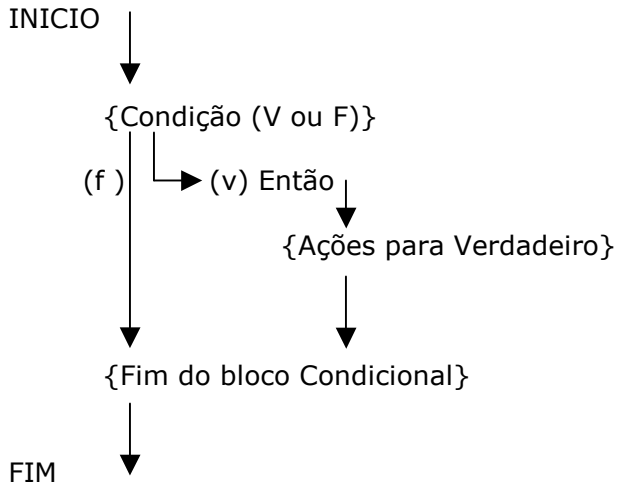
Uma estrutura de Seleção ou Condicional é aquela cujas ações são determinadas por uma verificação de entrada. Uma análise de condição será posta em questão e o computador através de um processamento de dados contidos na memória reconhecerá se a condição é verdadeira ou falsa e assim determinará quais ações serão executadas.

→ Existem 3 tipos de estruturas de seleção:

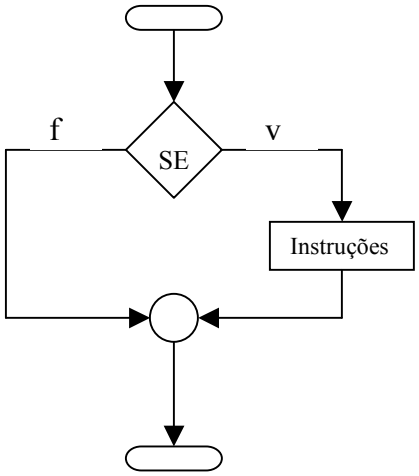
2.2.1 Seleção Simples (SE-ENTÃO)
2.2.2 Seleção Composta (SE-ENTÃO-SENÃO)
2.2.3 Seleção Múltipla (CASO)

2.2.1 Seleção Simples ou Única (SE ENTÃO): O bloco de instruções somente será executado se a verificação de entrada for verdadeira, caso isto não ocorra, o bloco condicional não será executado.

Pseudocódigo:



Fluxograma:



Exemplo Prático: (utilizando o português estruturado)

a.
```
Ir para escola
  Procurar minha agenda
  Se  (encontrar a agenda)
      Então
      Levá-la para escola
  Fim Se
  Pegar o ônibus
  Chegar na escola
```

b.
```
Ir a um baile
  Chegar a portaria do clube
  Mostrar a identidade
  Se  (idade menor que 18 anos)
      Então
      Apresentar acompanhante
  Fim Se
  Entrar no baile
  Curtir a festa
```


Exemplo Prático: (utilizando algoritmo)

a.
```
Inicio
// variáveis
    inteiro idade;
    caracter resposta;
// principal
    Escreva("Informe a idade");
    Leia(idade);
    Se (idade >= 18)
        Então
        Resposta ← "Maioridade";
        Escreva(resposta);
    Fim se;
Fim.
```

b.
```
Inicio
// variáveis
        inteiro idade;
        caracter resposta;
// principal
        Escreva("Informe a idade");
        Leia(idade);
        Se (idade>10) e (idade<30)
            Então
            Resposta ← "Jovem";
        Escreva(resposta);
        Fim se;
Fim.
```

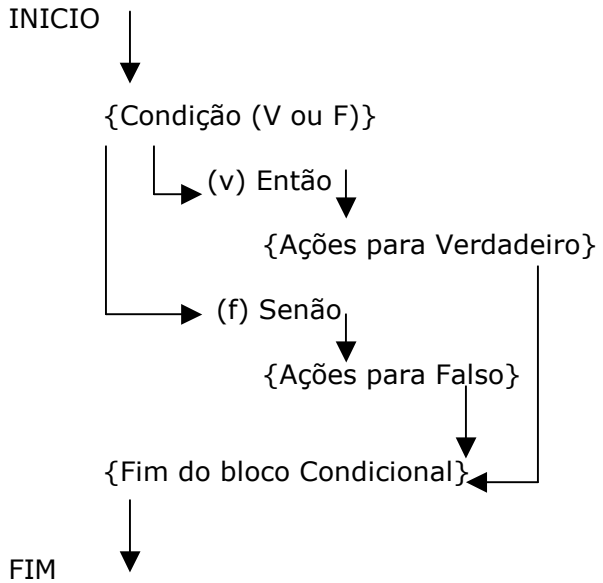
c.
```
Inicio
// variáveis
    inteiro idade;
    caracter resposta;
// principal
    Escreva("Informe a idade");
    Leia(idade);
    Se (idade=70)ou (idade=80)
        Então
        Resposta ← "3ª idade";
        Escreva(resposta);
    Fim se;
Fim.
```

d.
```
Inicio
// variáveis
        inteiro idade;
        caracter resposta;
// principal
        Escreva("Informe a idade");
        Leia(idade);
        Se ((idade>1) e (idade<9))
            ou (idade<1) Então
            Resposta ← "Criança";
            Escreva(resposta);
        Fim se;
    Fim.
```

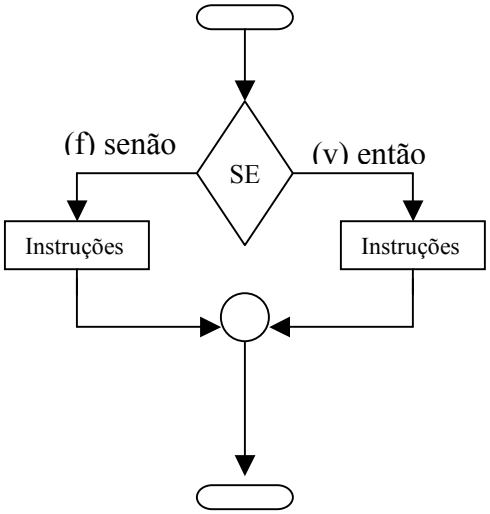
2.2.2 Seleção Composta e/ou Seleção Dupla (SE ENTÃO SENÃO) neste caso passa a existir mais de um bloco de instruções porém apenas um será executado, e isto se dará pelo resultado obtido na verificação de entrada. Quando esta verificação resultar em caminhos distintos para verdadeiro ou para falso, chamamos de Seleção Dupla, mas quando na verificação existir várias possibilidades de respostas então a chamamos de composta.

Modelo de Seleção Dupla

Pseudocódigo:



Fluxograma:



Exemplo Prático: (utilizando o português estruturado)

a.
```
Ir para escola
Se (fazer Frio)
     Então
          Vestir blusa
     Senão
          Vestir Camiseta
Fim se
Chegar a escola
```

b.
```
Sair a noite
Chegar a portaria do clube
Mostrar a identidade
Se  (idade >= 18)
   Então
        Entrar no baile
        Curtir a festa
   Senão
        Não entrar no baile
Fim Se
Voltar para casa
```

c.
```
Fazer a prova
Obter a nota
Se (nota < 30) Então
     Reprovado
Senão
     Se (nota >= 30) e (nota < 60) Então
          Recuperação
     Senão
          Aprovado
     Fim se
Fim se
```


Exemplo Prático: (utilizando algoritmo)

a.
```
Início
    // variáveis
        inteiro hora;
        caracter Ação1, Ação2, Ação3, Ação4, Ação5;
    // principal
        leia(hora);
        Se hora > 6 e hora < 18 então
                Ação1 = Levantar, tomar  café, trabalhar, almoçar, trabalhar e  ir embora;
                Se hora  > 6 e hora < 12 então
                        Ação2 = Levantar, tomar  café, trabalhar e almoçar;
                        Se hora > 6 e hora < 9 então
                                Ação3 = Levantar, tomar  café e trabalhar;
                                Se hora > 6 e hora < 8 então
                                        Ação4 = Levantar e  tomar  café ;
                                        Se hora > 6 e hora < 7 então
                                                Ação5 = Levantar;
                                        Fim se;
                                Fim se;
                        Fim se;
                Fim se;
        Fim se;
    Fim.
```

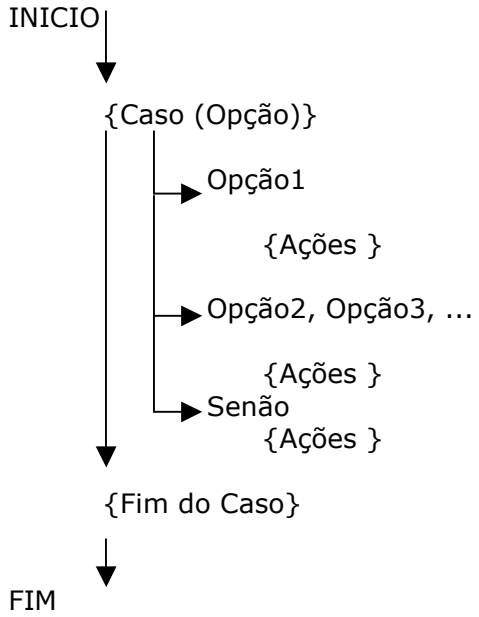
b.
```
Início
    // variáveis
        inteiro hora;
        caracter período;
    // principal
        leia(hora);
        Se hora > 6 e hora < 12 então
                período = manha;
        Senão
                Se hora  > 12 e hora < 18 então
                        período = tarde;
                Senão
                        Se hora > 18 e hora < 24 então
                                período = noite;
                        Senão
                                Se hora > 0 e hora < 6 então
                                        período = madrugada;
                                Fim se;
                        Fim se;
                Fim se;
        Fim se;
    Fim.
```

c.
```
Início
        inteiro hora;
        caracter período;
        leia(hora);
        Se hora > 6 e hora < 18 então          período = dia;
        Fim se;
        Se hora  > 6 e hora < 12 então         período = manha;
        Fim se;
        Se hora > 18 e hora < 24 então         período = noite;
        Fim se;
        Se hora > 12 e hora < 18 então         período = tarde;
        Fim se;
    Fim.
```

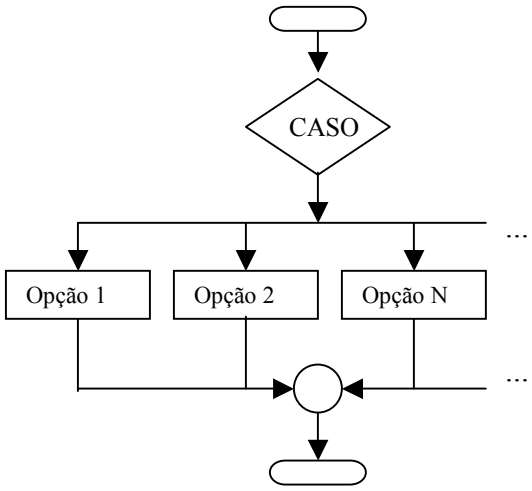

2.2.3 Seleção Múltipla (CASO) Utilizada quando temos muitas possibilidades para uma determinada situação, onde a aplicação da estrutura se...então...senão, tornaria o algoritmo muito complexo.

Modelo de Seleção Multipla

Pseudocódigo:



Fluxograma:




```
Caso <expressão>
    valor1 :  <comando 1>;
    valor2 : valor5 : <comando 2>;
    ...
    senão <comando n>;
fim-caso;
```

As opções podem apresentar valores individuais ou uma faixa de valores.

Exemplo Prático: (utilizando o português estruturado)

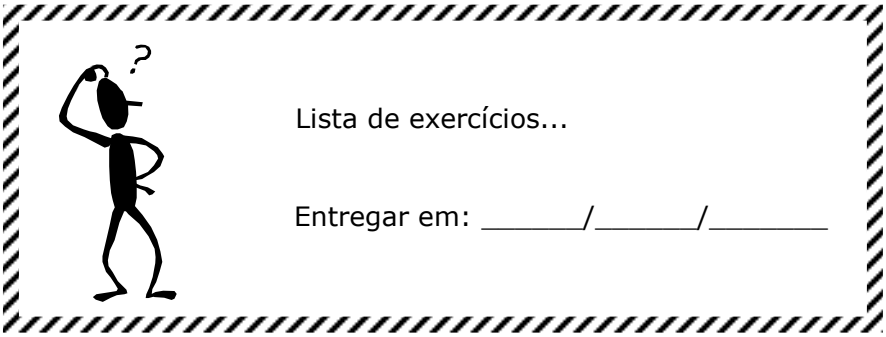
a.
```
Cumprimentar alguém
Olhe as Horas
Caso(Horas)
   >=6   e <11: Bom Dia
   >=12 e <18: Boa Tarde
   >=18 e <24: Boa Noite
Fim do caso
```

b.
```
Caixa Eletrônico
Informe a opção
Caso(opção)
      saque:  Agência, nº.conta, senha,
              valor. Retirar dinheiro
      extrato:Informar Agência,nº.conta,
              senha. Retirar extrato
      deposito:Informar Agência,nº.conta,
              valor. Retirar comprovante
Fim do caso
```

Exemplo Prático: (utilizando o algorítmo)

a.
```
Inicio
  // declaração de variáveis
  Inteiro Numero;
  Caracter Extenso;
  // Principal
  leia(Numero);
  caso(Numero)
        1: Extenso ← 'Um';
        2: Extenso ← 'Dois';
        3: Extenso ← 'Três';
        4: Extenso ← 'Quatro';
        5: Extenso ← 'Cinco';
        6: Extenso ← 'Seis';
        7: Extenso ← 'Sete';
        8: Extenso ← 'Oito';
        9: Extenso ← 'Nove';
        senão:  Extenso ← 'Erro';
  fim-caso;
Fim.
```

b.
```
Inicio
  // declaração de variáveis
  Inteiro A,B, Opcao;
  Real C;
  // Principal
     A ← 3;
     B ← 2;
     escreva(" Escolha sua opção:");
     escreva(" 1 – Somar.");
     escreva(" 2 – Subtrair.");
     escreva(" 3 – Multiplicar.");
     escreva(" 4 – Dividir.");
     leia(Opcao);
     caso(Opcao)
          1: C ← A+B;
          2: C ← A-B;
          3: C ← A*B;
          4: C ← A/B;
          senão: escreva("Inválido");
     fim-caso;
     escreva("O resultado será: ",C);
Fim.
```



2.3 Programação Estruturada de Repetição

Uma estrutura de Repetição é aquela cujas ações são executadas repetidamente, enquanto uma determinada condição permanece válida. O fator que diferencia os vários modelos de estrutura de repetição é o ponto em que será colocada a condição.

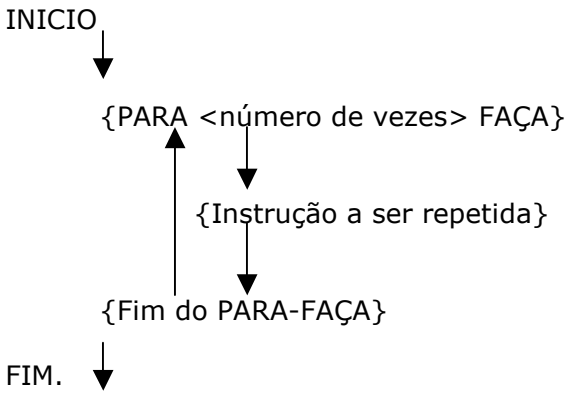
→ Existem 3 tipos de estruturas de repetição:

2.3.1 Para-Faça
2.3.2 Enquanto-Faça
2.3.3 Repita-Até

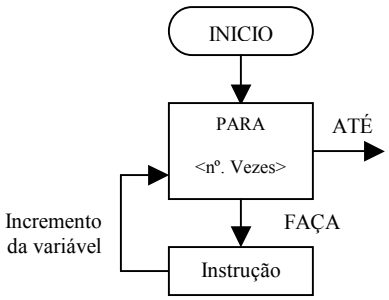
2.3.1 Para-Faça Usamos a estrutura Para-Faça quando precisamos repetir um conjunto de comandos em um número pré-definido de vezes. É conhecida como estrutura de repetição por laços (loops) definidos, pois se utiliza uma variável de controle que é incrementada em um número determinado de unidades de um valor inicial até um valor final.

Modelo de Repetição Para-Faça

Pseudocódigo:



Fluxograma:




```
PARA  Inicio  ATÉ  Fim  FAÇA
        <Sentença 1>;
        <Sentença 2>;
        <Sentença n>;
Fim-Para;
```

Execução enquanto Início for menor ou igual a Fim

Quando o algoritmo encontra a instrução fim-para, incrementa a variável INICIO em 1 unidade (default) ou mais. Cada vez que é completada o bloco de instrução, ele retorna a linha da estrutura PARA e testa se INICIO é menor ou igual a FIM, se for menor ou igual o processo continua no laço (loop), caso não, o processo é abandonado.
Obs: O valor da variável INICIO não pode ser alterado no interior da estrutura.

Exemplo Prático: (utilizando o português estruturado)

Ir de elevador do primeiro ao quinto Andar
Chamar o elevador
Entrar no elevador
Informar o andar
Para andar do primeiro até o quinto faça
    mover ao próximo andar
Fim do Movimento
Sair do elevador

Exemplo Prático: (utilizando o algoritmo)

a.
```
Inteiro var, resultado;
para var ← 1 até 10 faça
        resultado ← 2 * var;
fim-para;
escreva(resultado);
```

b.
```
Inteiro var, resultado;
para var ← 1 até 10 faça
        resultado ← 2 * var;
        escreva(resultado);
fim-para;
```

c.
```
Inteiro var, resultado;
para var←1 até 5 passo 2 faça
        resultado ← 2 * var;
        escreva(resultado);
fim-para;
```

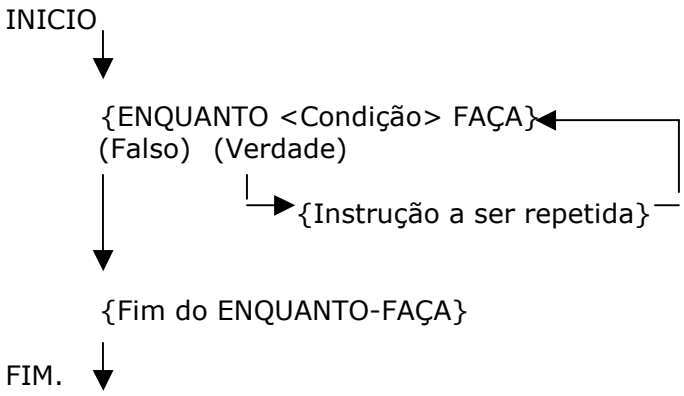
:: Exercícios propostos ::

```
Dadas as informações a seguir escreva a série
Variavel  x
Valor inicial  1
Valor final  30
Regra de ciclo x<-x+1
Formula padrao x+(x*2)/(x^2)
```

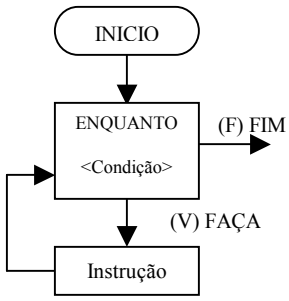
2.3.2 Enquanto-Faça Utilizada quando não sabemos o número de repetições e quando possuímos uma expressão que deve ser avaliada para que os comandos da estrutura sejam executados. Assim, enquanto o valor da <condição> for verdadeiro, as ações dos comandos são executadas. Quando for falso, a estrutura é abandonada, passando a execução para a próxima linha após o comando FIM-ENQUANTO. Se já da primeira vez o resultado for falso, os comandos não serão executados.

Modelo de Repetição Enquanto-Faça

Pseudocódigo:



Fluxograma:




```
Enquanto <condição> Faça
    <comando 1>;
    <comando 2>;
    <comando n>;
Fim-Enquanto;
```

execução enquanto a condição for verdadeira

É sempre importante observar que primeiro se analisa a condição para depois dependendo do resultado obtido executar o bloco a ser repetido. Caso a condição não seja satisfeita nada será feito e a próxima linha após o fim-enquanto será requisitada. Também é necessário caso a condição seja verdadeira permitir o incremento para a variável em condição (se necessário) para que a estrutura não entre em loop infinito.

Exemplo Prático: (utilizando o português estruturado)

```
Ir de elevador do primeiro ao quinto Andar
Chamar o elevador
Entrar no elevador
Informar o Andar
Enquanto Andar atual  for menor que 5 faça
    mover o elevador para cima
    Andar passa para o próximo
Fim do Movimento
Sair do elevador
```


Exemplo Prático: (utilizando o algoritmo)

```
a.
aux ← 1;
enquanto (aux <= 10) faça
        resultado ← 5 * aux;
        aux ← aux + 1;
fim-para
```

:: Exercícios propostos ::

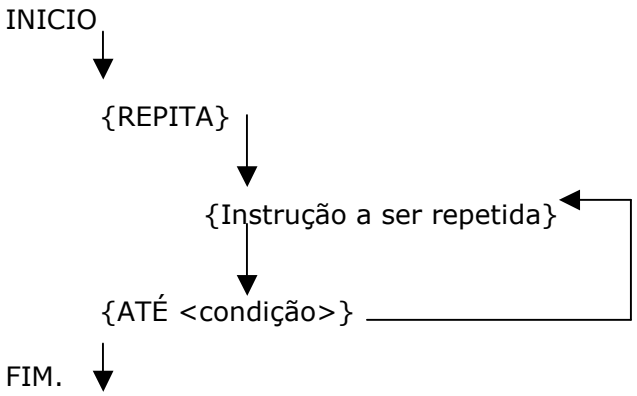
Analisando Séries co estruturas de repetição

S = 3 / 6
Variáveis envolvidas:
Valores de Inicio:
Fórmula Padrão:
Regra de Ciclo:

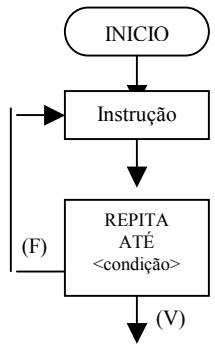
2.3.3 Repita-Até Utilizada quando não sabemos o número de repetições e quando os comandos devem ser executados pelo menos uma vez, antes da expressão ser avaliada. Assim, o programa entra na estrutura Repita...Até que executa seus comandos pelo menos uma vez. Ao chegar no fim da estrutura, a expressão será avaliada. Se o resultado da expressão for verdadeiro, então o comando é abandonado.

Modelo de Repetição Repita-Até

Pseudocódigo:



Fluxograma:



Repita

<comando 1>;
<comando 2>;
<comando n>;

Até <condição>;

É executado pelo menos uma vez

Exemplo Prático: (utilizando o português estruturado)

a.
Ir de elevador do primeiro ao quinto Andar
Chamar o elevador
Entrar no elevador
Informar o Andar
Repita
    mover o elevador para cima
    Andar passa para o próximo
Até Andar atual igual a 5
Sair do elevador

b.
Executar um aplicativo com senha
Abrir o Windows
Abrir a pasta do aplicativo
Executar o arquivo principal
Repita
    Digitar a senha
Até senha ser válida
Usar o aplicativo

Exemplo Prático: (utilizando o algoritmo)

```
a.
   aux ← 1;
   repita
         resultado ← 5 * aux;
         escrever resultado;
         aux ← aux + 1;
   até (aux > 10);
```

:: Exercícios propostos ::

```
inicio
      inteiro x,  y;
      x <- 0;
      y <- 0;
      enquanto x > 0 faça
            y <- y + 3;
            x <- x + 1;
      fim-enquanto;
fim.
```

```
inicio
      inteiro x,  y;
      x <- 0;
      y <- 0;
      repita
            y <- y + 3;
            x <- x + 1;
      até x > 0;
fim.
```

3. Aplicação de variáveis dimensionais homogêneas

3.1 Vetores – Variáveis Unidimensionais

Os vetores são estruturas de dados que permitem o armazenamento de um conjunto de dados de mesmo tipo. Por este motivo, são chamadas de estruturas homogêneas. Os vetores são unidimensionais, pois cada elemento do vetor é identificado por um índice.

Similarmente, podemos definir vetores como posições de memória, identificadas por um mesmo nome, individualizadas por índices e cujo conteúdo é de mesmo tipo.
Para acessarmos um elemento de um vetor, referimo-nos ao nome do vetor acompanhado pelo seu índice que virá entre colchetes ([e]). Pense num prédio com 120 apartamentos.

Para enviar uma correspondência a um determinado apartamento, devemos colocar no endereço de destinatário, o número do prédio mais o número do apartamento. O vetor funciona de forma similar.

Veja a sintaxe da declaração de um vetor:

Tipo Básico de Dados: Nome do vetor[nº de elementos]

Para fazermos referência a um elemento do vetor, colocamos:

Nome do vetor[elemento]

Cada elemento de um vetor é tratado como se fosse uma variável simples.

Exemplo:

Supondo que pedíssemos para criar um algoritmo para ler o nome de 5 pessoas, e mostrasse esses nomes na ordem inversa de leitura. A princípio, vocês pensariam em cinco variáveis: nome1, nome2, nome3, nome4 e nome5.

Veja como ficaria a solução, nesse caso:

```
Inicio
    caracter: nome1, nome2, nome3, nome4, nome5;
    escreva('Informe o nome de 5 pessoas: ');
    leia(nome1);    //ANA
    leia(nome2);    //PAULA
    leia(nome3);     //CRISTINA
    leia(nome4);    //GUSTAVO
    leia(nome5);    //ANTONIO
    escreva('Ordem Inversa de Leitura ');
    escreva(nome5);
    escreva(nome4);
    escreva(nome3);
    escreva(nome2);
    escreva(nome1);
Fim
```

Assim, na memória teríamos ...

Nome1	Nome2	Nome3	Nome4	Nome5
ANA	PAULA	CRISTINA	GUSTAVO	ANTONIO

Todavia, se alterássemos esse algoritmo para ler o nome de 100 pessoas, a solução anterior se tornaria inviável. Para casos como este, podemos fazer uso de vetores. Se tivéssemos criado 100 variáveis, teríamos que declarar e usar: nome1, nome2, nome3, ..., nome99, nome100. Com o vetor passamos a ter: nome[1], nome[2], nome[3], nome[99], nome[100], onde a declaração do vetor se limita à linha: caracter: nome[100].

Veja que para todos os elementos nos referimos ao mesmo nome de vetor "Nome".

Assim, veja a solução do algoritmo anterior com o uso de vetores:

```
Início
    Caracter: nome[5];
    Inteiro: aux;
    para aux ← 1 até 5 faça
       escreva('Informe o Nome ', aux);
       leia(nome[aux]);
    fim-para;

   escreva('Ordem Inversa de Leitura ');

    para aux ← 5 até 1 faça
       escreva (nome[aux]);
    fim-para
fim
```

Veja a representação da memória:

NOME				
1	2	3	4	5
ANA	PAULA	CRISTINA	GUSTAVO	ANTONIO

3.2 Matrizes – Variáveis Multidimensionais

As matrizes são estruturas de dados que permitem o armazenamento de um conjunto de dados de mesmo tipo, mas em dimensões diferentes. Os vetores são unidimensionais, enquanto as matrizes podem ser bidimensionais (duas dimensões) ou multidimensionais.

Similarmente podemos conceituar matrizes como um conjunto de dados referenciado por um mesmo nome e que necessitam de mais de um índice para ter seus elementos individualizados.

Para fazer referência a um elemento da matriz serão necessários tantos índices quantas forem as dimensões da matriz.

Veja a sintaxe da declaração de uma matriz:

Tipo básico de Dados : Nome da matriz [Li, Ls, Nq]

onde:

Li – Limite inferior *
Ls – Limite superior *
Nq – Número de quadrantes (somente para matrizes multidimensionadas)
* Valores obrigatórios

Para fazermos referência a um elemento da matriz, colocamos:

Nome da matriz [linha, coluna]

O número de dimensões de uma matriz pode ser obtido pelo número de vírgulas (,) da declaração mais 1. O número de elementos pode ser obtido através do produto do número de elementos de cada dimensão.

Obs: Quando você desejar percorrer uma matriz, linha por linha, crie uma estrutura de repetição, fixando a linha e variando a coluna. Para percorrer uma matriz, coluna por coluna, fixe a coluna e varie a linha.

Vamos pensar numa estrutura onde as colunas representem os cinco dias úteis da semana, e as linhas representem as três vendedoras de uma loja. Na interseção de cada linha x coluna, colocaremos o faturamento diário de cada vendedora.

	(Segunda) **COLUNA 1**	(Terça) **COLUNA 2**	(Quarta) **COLUNA 3**	(Quinta) **COLUNA 4**	(Sexta) **COLUNA 5**
(SANDRA) **LINHA 1**	1050,00	950,00	1241,00	2145,00	1256,00
(VERA) **LINHA 2**	785,00	1540,00	1400,00	546,00	0,00
(MARIA) **LINHA 3**	1658,00	1245,00	1410,00	245,00	1546,00

A representação desta tabela em forma de matriz, seria:

VendasDiarias : matriz [3, 5] de real;

Veja como ficaria o algoritmo para ler esses valores:

```
Algoritmo LeVendasDiarias;
Inicio
      real: VendasDiarias[3,5];
      inteiro: ndLinha, indColuna, i ;


   //Variando o número de linhas - Vendedoras
   Para indLinha ← 1 até 3 faça
         escrever ('Vendedora :', indLinha);

      //Variando o número de colunas – Dias da Semana
      Para indColuna ← 1 até 5 faça
            escreva ('Faturamento do Dia : ', indColuna);
            leia (VendasDiarias[indLinha, indColuna]);
         Fim-para;
   Fim-para;

Fim
```

Poderíamos melhorar o algoritmo acima, trabalhando com um vetor que contivesse os nomes dos dias da semana e das vendedoras. Assim, a comunicação do programa com o usuário ficaria mais clara. Veja:

```
Algoritmo LeVendasDiariasVersao2;

Inicio
      real: VendasDiarias[3,5];
      caracter: Vendedoras[3];
      caracter: DiasSemana[5];
      inteiro: indLinha, indColuna;

      Vendedoras[1] ← 'Sandra';
      Vendedoras[2] ← 'Vera';
      Vendedoras[3] ← 'Maria';
      DiasSemana['Segunda', 'Terça', 'Quarta', 'Quinta', 'Sexta');

      //Variando o número de linhas - Vendedoras
      Para indLinha ← 1 até 3 faça
            escreva('Vendedora : ', Vendedoras[indLinha]);
            //Variando o número de colunas – Dias da Semana
            Para indColuna ← 1 até 5 faça
                  escreva('Fatur.do Dia:', DiasSemana[indColuna]);
                  leia(VendasDiarias[indLinha, indColuna]);
            Fim-para;
      Fim-para;
Fim
```

Um algoritmo que apenas lê e nada faz com esses resultados, não serve para grande coisa, certo ?! Por isso, vamos melhorar esse algoritmo e apresentar como resultado o faturamento diário de todas as vendedoras.

```
Algoritmo LeVendasDiariasVersao3;
Início
      real: VendasDiarias[3,5];
      caracter: Vendedoras [3], DiasSemana[5];
      inteiro: indLinha, indColuna;
      real: FaturaDia;

      Vendedoras[1] ← 'Sandra';
      Vendedoras[2] ← 'Vera';
      Vendedoras[3] ← 'Maria';

      DiasSemana[1] ← 'Segunda';
      DiasSemana[2] ← 'Terça';
      DiasSemana[3] ← 'Quarta';
      DiasSemana[4] ← 'Quinta';
      DiasSemana[5] ← 'Sexta';

      //Variando o número de linhas – Vendedoras }
      para indLinha ← 1 até 3 faça
            escreva('Vendedora : ', Vendedoras[indLinha]);
            //Variando o número de colunas – Dias da Semana
            para indColuna ← 1 até 5 faça
                  escreva('Fatur.do Dia : ', DiasSemana[indColuna]);
                  leia(VendasDiarias[indLinha, indColuna]);
            fim-para;
      fim-para;

      //Vamos começar variando a coluna, para poder obter o //faturamento
      de cada dia da semana
      Para indColuna ← 1 até 5 faça
            //A cada novo dia a variável que recebe faturamento é
            // zerada
            FaturaDia ← 0;

            //Vamos variar a linha, para obter os valores
            //faturados de cada vendedora
            para indLinha ← 1 até 3 faça
                  FaturaDia←FaturaDia + _
                              VendasDiarias[indLinha,indColuna];
            fim-para

            escreva("Faturamento de : ", DiasSemana[indColuna]);
            escreva(FaturaDia);

      fim-para;

fim
```

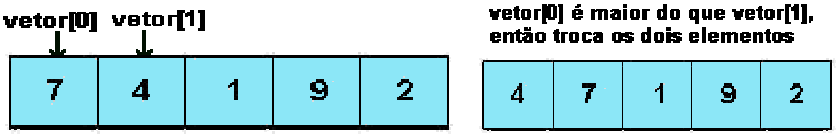
4. Análise de algoritmos para ordenação de vetores

4.1 O Algoritmo Bubble Sort (Método da Bolha)

Uma introdução ao sort da bolha

Um dos métodos de ordenação é o sort da bolha. O sort da bolha funciona movendo repetidamente o maior elemento para a posição de maior índice do vetor. Ao invés de percorrer todo o vetor para localizar o maior elemento, o sort da bolha se concentra em sucessivos pares de elementos adjacentes no vetor.

Ele compara os dois elementos,e então ou os troca (se o elemento com o menor índice é maior que o outro elemento) ou os deixa do jeito que estão (se o elemento com o menor índice é menor do que o outro elemento). Em ambos os casos, após tal passo, o maior dos dois elementos estará na posição de maior índice.



O foco então se move para a próxima maior posição, e o processo é repetido. Quando o foco alcança o final do vetor, o maior elemento será movido de qualquer que tenha sido sua posição original para o fim ou para a posição de maior índice no vetor. O processo então é repetido, fazendo com que o segundo maior elemento seja movido para a posição de maior índice - 1. Este processo é refeito até que todos os elementos do vetor tenham sido ordenados.

Veja um exemplo mais detalhado de um sort da bolha.

Um algoritmo genérico para o sort da bolha

O algoritmo do sortda bolha tem muitas variantes. No entanto, o algoritmo dado abaixo é uma bse para todos eles.

```
Procedure BubbleSort(item[1..n])

for j:=n downto 1 do
    for k:=1 to (j-1)
      if item[k] > item[k+1]
      then Troca(item,k,k+1);
Return

Procedure Troca(A,i,j)
Aux = A[i]
A[i] = A[j]
A[j] = Aux
Return
```

Examinaremos agora como este algoritmo funciona.
Loop Principal

- Compara dois elementos adjacentes nas posições "k" "k + 1"
- Se o elemento na posição "k" é maior do que o elemento "k + 1" então troca a posição dos dois valores
- Uma iteração move o maior valor para a última posição do vetor
- Repete o loop
- Finaliza o loop principal

Um algoritmo para o sort da bolha também pode ser descrito passo a passo como se segue:

1. Repete o passo 4 num total de n-1 vezes.
2. Repete o passo 3 para os elementos na parte não ordena do vetor.
3. Se o elemento atual do vetor é maior do que o próximo elemento então troca os elementos.
4. Se nenhuma mudança é feita, então retorne, caso contrário reduz o tamanho da parte não ordenada em 1.

Eficiência do Sort da Bolha

O sort da bolha é composto de dois loops. O número de vezes que o loop externo iterage (é repetido) é determinado pela natureza da lista que vai ser ordenada. O loop interno é repitido uma vez a menos do que o número de de elementos no vetor e é novamente chamado (do começo) toda vez que se se repete o loop externo. Então a ordem do loop interno é n-1 ou O(n-1).

Melhor Caso
Neste caso, a lista já está ordenada quando o algoritmo do sort da bolha é chamado. O loop interno irá se repetir sem que a condição do if seja verdadeira. O loop externo irá terminar após uma repetição. Portanto, no melhor caso, o loop interno irá se repetir num total de n-1 vezes, a condição do if nunca será verdadeira, o procedure de troca nunca será chamado e o número de trocas é 0.

Pior Caso
Neste caso, a análise se torna um pouco mais difícil. O pior caso para o algoritmo do sort da bolha, ocorre com uma lista invertida (9 8 7 ... 1). O maior elemento se encontra no início da lista e portanto tem que ser movido para o final para que a lista fique em ordem. Na primeira iteração do loop externo, ele será trocado com o segundo elemento, na segunda iteração do loop, será trocado com o terceiro elemento. Continuando este raciocínio, este elemento se moverá uma posição até alcançar o final da lista a cada iteração do loop externo. Para ele se mover do início até o final da lista, ele terá que ser trocado n-1 vezes.
Todos os outros elementos da lista serão movidos uma posição até alcançar o início ou até o final da lista a cada iteração. Como os elementos no início ou no final da lista têm a distância mais longa a percorrerem, todos os outros elementos estarão em suas posições corretas quando o elemento que estiver no final da lista tenha sido movido para o início. Uma última iteração será necessária para determinar que a lista esteja ordenada. Portanto, o número total de iterações do loop externo para se ordenar uma lista invertida é n e o número total de iterações do loop interno é n(n-1), o qual é aproximadamente igual a n2 ou O(n2).

Vantagens e desvantagens do sort da bolha
Uma das principais vantagens do sort da bolha é que ele é extremamente fácil de implementar (por volta de 5 linhas de código). É também bastante fácil de se entender em se tratando de um algoritmo de ordenação.

Infelizmente, o sort da bolha é um algoritmo muito lento, levando O(n2) para finalizar a ordenação e portanto, não deve ser usado em tabelas muito grandes.

4.2 O Algoritmo do Método da Inserção

Método da Inserção
Um dos métodos mais simples de se ordenar um vetor é o Sort da inserção. Um exemplo do Sort da Inserção ocorre em nosso dia a dia toda a vez que jogamos baralho. Para ordenar as cartas que temos na mão, extraímos uma carta, movemos as cartas restantes e colocamos a carta extraída no seu lugar correto. Este processo se repete até que todas estejam na sequência correta. Ambos médio e pior caso têm uma complexidade de O(n2).

Funcionamento

O sort da inserção é como o sort da bolha, só que mais eficiente. O sort da inserção pega o primeiro elemento e o compara com o elemento próximo a ele. Se este for maior,os dois elementos são trocados. Então ele vai percorrendo todo o vetor comparando todos os outros elementos com o primeiro elemento. Se encontrar um valor menor do que ele, o algoritmo o troca com o primeiro elemento.Quando o final do vetor é alcançado, se move para o segundo elemento e o compara com os elementos que o seguem no vetor. E faz uma troca se encontrar algum menor que ele. Percorre todo o resto do vetor dessa maneira até comparar o penúltimo elemento com o último elemento. Ele usa menos comparações e isso o torna mais rápido que o sort da bolha.
Começando pelo topo do vetor na Figura 1(a), extraímos o 3. Então os elementos acima são movidos para baixo até acharmos o lugar correto para inserirmos o 3. Este processo é repetido na Figura 1(b) com o próximo número. Finalmente, na Figura 1(c), completamos a ordenação colocando o 2 em seu devido lugar.

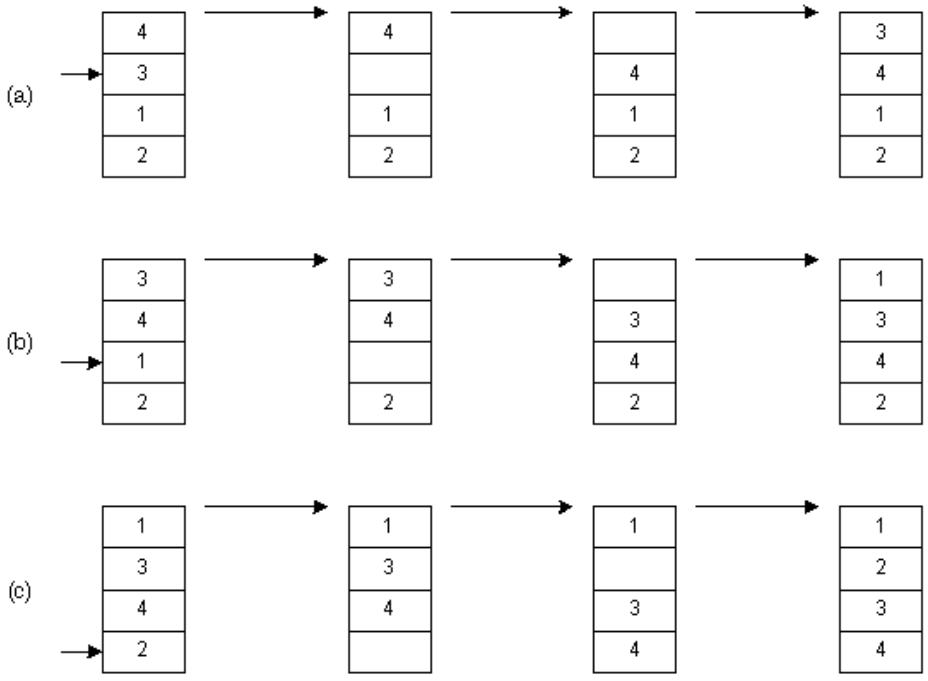


Figura 1: Sort da Inserção

Assumindo que temos n elementos no vetor, temos que passar por n - 1 entradas. Para cada entrada, temos que examinar e mover n - 1 outras entradas, resultando num algoritmo de complexidade O(n2). O sort da inserção não usa memória extra, pois a ordenação é feita no próprio vetor de origem.O sort da inserção também é estável. Sorts estáveis mantém a ordem original das chaves quando chaves idênticas estão presentes nos dados de entrada.

Um algoritmo para o sort da inserção

Segue abaixo o algoritmo do sort da inserção:

```
algorithm insertionsort
begin
  for i= n-1 down to 1 begin
    temp = xi
    j = i+1
    while(j <= n and xj < temp) begin
       xj - 1 = xj
      j=j+1
    end
    xj - 1 = temp
  end
end
```

Para ver o que está acontecendo, a tabela abaixo mostra o algoritmo trabalhando na lista (4, 3, 6, 1, 5, 2).

i = 5	4	3	6	1	5	2	temp = 5
i = 4	4	3	6	1	2	5	temp = 1
i = 3	4	3	6	1	2	5	temp = 6
i = 2	4	3	1	2	5	6	temp = 3
i = 1	4	1	2	3	5	6	temp = 4
I = 0	1	2	3	4	5	6	temp =

O algortimo passo a passo pode ser descrito como se segue:

1. De um até o comprimento do vetor -1 faça os passos de 2 a 4
2. Em seguida ao primeiro loop até o final do vetor-1 faça o passo 3
3. Se vetor[variável] é maior do que vetor[variável + 1] então troque os elementos
4. Repita isto até que o penúltimo elemento seja comparado com o último elemento

4.3 O Algoritmo do Método da Seleção

Uma introdução ao Método da Seleção

O método da seleção é um dos meios mais fáceis de se ordenar dados. Ao invés de trocar números vizinhos a toda hora como o Sort da Bolha , este algoritmo identifica o menor elemento do vetor e o compara com o elemento da primeira posição do vetor. Depois disso, ele reexamina os elementos restantes no vetor para achar o segundo menor elemento. O elemento encontrado é comparado com o elemento da segunda posição do vetor. Este processo se repete até que todos os elementos sejam ordenados. A ordem pode ser definida pelo usuário (i.e. decrescente ou crescente).

Um algoritmo genérico para o método da seleção
Segue abaixo algoritmo da seleção em pseudocódigo:

```
Seleção ( A, N )
    For i = 1 to n-1 do
        min = i
```

```
        For j =i+1 to n do
            if A[j] < A[min] then
                    min = j
                  A[min] = a[ i ]
           End If
        Next j
    Next i

    Return
```

Um algoritmo de seleção também pode ser descrito passo a passo como se segue :

1. Repita os passos de 1 a 5 no total de n-1 vezes.
2. Grave a parte do vetor que já está ordenada.
3. Repita o passo 4 para os elementos na parte não ordenada do vetor.
4. Grave a posição do menor elemento da parte não ordenada do vetor
5. Compare o primeiro elemento do vetor não ordenado com o menor elemento

Eficiência do método da seleção

Para um determinado número de elementos, o método da seleção passa por um número de comparações e trocas, portanto sua performance é previsível O número de iterações entre os elementos depende o quão ordenado está o vetor. Durante cada passo apenas uma troca é essencial, portanto o número máximo de trocas para este algoritmo é n-1

O número total de comparações é proporcional a n2 ou O(n2).

Pior caso

O pior caso do método da seleção é n2/4.

Vantagens e desvantagens do método da seleção

O método da seleção é fácil de entender e isto facilita a correta implementação do algoritmo. E faz com que o processo de ordenação seja fácil de escrever. No entanto, há algumas razões pelas quais os programadores se recusam a usar este algoritmo:

1. Este método usa ordenação interna. Isto significa que todo o vetor teria que ser carregado na memória principal, o que requereria uma memória muito grande. No caso de grandes bancos de dados comerciais, mais memória do que a disponível.
2. O performance O (n2) do algoritmo tornaria-se muito lenta em grandes volumes de dados.Se você tiver um vetor com 100.000 elementos, o método da seleção não seria a escolha correta.
3. Atualmente, muitas base de dados de aplicações possuem listas ordenadas dos dados e essas listas são atualizadas regularmente. Na maioria das vezes estas base de dados estão, em sua maior parte, em ordem. Um método ideal seria capaz de reconhecer este fato e trabalharia apenas com os itens não ordenados. O método da seleção é incapaz de fazer isso

4.4 O Algoritmo MergeSort (Método da Recursividade)

MergeSort é um algoritmo de ordenação recursivo que usa O(n log n) comparações para o pior caso. Para ordenar um vetor de n elementos, nós executamos os três passos seguintes em sequência:

Se n < 2, então o vetor já está ordenado.

De outra forma, se n > 1, nós executamos os três passos:

1. Ordenamos a metade esquerda do vetor.
2. Ordenamos a metade direita do vetor.
3. Intercalamos as metades.

Complexidade de Tempo

Para se ter uma ideia de quanto tempo leva a execução de um algoritmo MergeSort, nós levamos em consideração o número de comparações que ele faz para o pior caso.

O cálculo do consumo de tempo de um algoritmo recursivo consiste, essencialmente, na solução de uma relação de recorrência. Usaremos uma função T(n), onde n é o tamanho do vetor que queremos ordenar.

Ordenação por intercalação

Rearranjar um vetor A[p..r] de modo que ele fique em ordem crescente.
Eis um algoritmo recursivo que resolve o problema.
Ele supõe que p <= r e adota o caso p = r como base da recursão.

```
MERGESORT (A, p, r)
  se p < r então
    q := chão((p+r)/2)
    MERGESORT (A, p, q)
    MERGESORT (A, q+1,
r)
    INTERCALA (A, p, q, r)
```

P			Q	q+1		r
1	3	5	9	9	2	8

Observe que p <= q < r. Com isso, tanto A[p..q] quanto A[q+1..r] são estritamente menores que A[1..n].

O procedimento INTERCALA recebe A[p..q..r] tal que A[p..q] e A[q+1..r] são crescentes e rearranja o vetor de modo que A[p..r] fique crescente.

```
INTERCALA (A, p, q, r)
   para i := p até q faça
         B[i] := A[i]
   para j := q+1 até r faça
         B[r+q+1-j] := A[j]
   i := p
   j := r
   para k := p até r faça
         se B[i] <= B[j]
            então A[k] = B[i]
                  i = i+1
            senão A[k] = B[j]
                  j := j-1
```

Analisando-se esse procedimento, percebe-se que a complexidade de tempo de Intercala é O(n).

Então, qual o consumo de tempo do MERGESORT no pior caso?

Uma Expressão para T(n)

Como o MergeSort tem dois casos, a descrição de T(n) também terá dois casos. O 1° Caso é a base da recursão:

T(n) = 0, se n > 2.

Para o caso do vetor ter n elemento com n > 1,o número de comparações usadas para ordenar n elementos é no máximo a soma do número de comparações para cada uma das metades.

T(n) = T(n/2) + T(n/2) + n, se n > 1.

Observando a expressão acima, temos que o 1° termo da expressão indica o n° de comparações usadas para ordenar a metade esquerda do vetor.
O segundo termo da expressão indica o n° de comparações usadas para ordenar a metade direita do vetor.
O último termo n, indica o n° de comparações usadas no procedimento Intercala.

Como encontrar a Complexidade de Tempo

Para tanto, deve-se primeiramente enunciar o seguinte Teorema:

Teorema: Sejam a,b,c,k constantes não negativas com a >0, b > 1, a solução para T(n) = aT(n/b) + cnk é :

T(n) :

O(n logab), a > bk
O(nk logn), a = bk
O(nk), a < bk

No nosso caso, T(n) = 2T(n/2) + n, temos que :

a = 2
b = 2
cnk = n, logo k = 1

Temos a seguinte relação : a = bk <--> 2 = 21

Dessa relação temos que T(n) é O(nk logk), ou seja,

O algoritmo MergeSort é O(n log n)

4.5 O Algoritmo QuickSort (Método Rápido)

Algoritmo Básico

Quicksort trabalha particionando um arquivo em duas partes e então as ordenando separadamente.
Pode ser definido recursivamente:

```
quicksort (tipoInfo : a[], inteiro : esq, dir)
    inteiro i;
    início
        se dir > esq então
            i <- particione(a, esq, dir);
            quicksort(a, esq, i-1);
            quicksort(a, i+1, dir);
        fim se
    fim
```

Os parâmetros dir e esq delimitam os subarquivos dentro do arquivo original, dentro dos quais a ordenação ocorre.
A chamada inicial pode ser feita com quicksort(a, 1, N);
O ponto crucial é o algoritmo de partição.

Particionamento em Quicksort

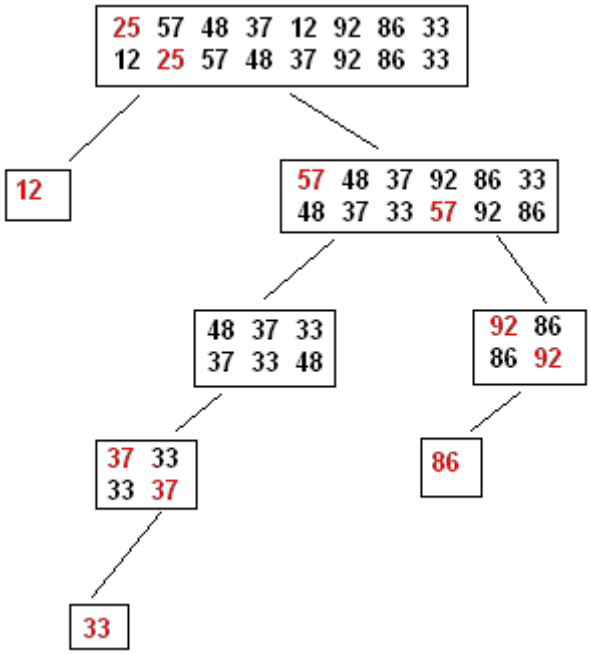
- O procedimento de particionamento deve rearranjar o arquivo de maneira que as seguintes condições valham:

1. o elemento a[i] está em seu lugar final no arquivo para um i dado,
2. nenhum dos elementos em a[esq],...,a[i-1] são maiores do que a[i],
3. nenhum dos elementos em a[i+1],...,a[dir] são menores do que a[i].

Exemplo

Ordenação do vetor inicial 25 57 48 37 12 92 86 33.
Se o primeiro elemento (25) for colocado na sua posição correta, teremos: 12 25 57 48 37 92 86 33.
Neste ponto:
todos os elementos abaixo de 25 serão menores e
todos os elementos acima de 25 serão maiores que 25.
Como 25 está na sua posição final, o problema foi decomposto na ordenação dos subvetores: (12) e (57 48 37 92 86 33).
O subvetor (12) já está classificado.
Agora o vetor pode ser visualizado: 12 25 (57 48 37 92 86 33).
Repetir o processo para a[2]...a[7] resulta em: 12 25 (48 37 33) 57 (92 86)
Se continuarmos particionando 12 25 (48 37 33) 57 (92 86), teremos:
12 25 (37 33) 48 57 (92 86)
12 25 (33) 37 48 57 (92 86)
12 25 33 37 48 57 (92 86)
12 25 33 37 48 57 (86) 92
12 25 33 37 48 57 86 92

Visão da Recursividade do Quicksort como Árvore:



O subproblema da partição

O coração do QUICKSORT está no seguinte subproblema, que formularemos de maneira propositalmente vaga: rearranjar A[p..r] de modo que todos os elementos pequenos fiquem na parte esquerda do vetor e todos os elementos grandes fiquem na parte direita. Este é o subproblema da partição . O ponto de partida para a solução do subproblema é a escolha de uma "chave", digamos x: os elementos do vetor que forem maiores que x serão considerados grandes e os demais serão considerados pequenos. A dificuldade está em escolher x de tal modo que cada uma das duas partes do vetor rearranjado seja estritamente menor que o vetor todo.

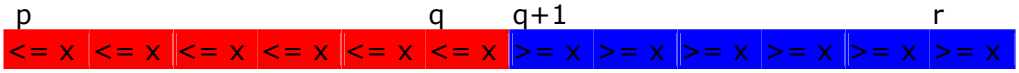
O seguinte algoritmo resolve o subproblema da separação da seguinte maneira: supondo p < r, o algoritmo rearranja os elementos de A[p..r] e devolve um índice q tal que

p <= q e A[i] <= x para cada i em p..q e

q < r e A[j] >= x para cada j em q+1..r

para algum x.

Essa versão do algoritmo adota como chave x o valor inicial de A[p].




```
     PARTICIONE (A, p, r)
1      x := A[p]
2      i := p-1
3      j := r+1
4      enquanto 0 = 0
5        faça repita j := j-1
6             até que A[j] <= x
7           repita i := i+1
8             até que A[i] >= x
9           se i < j
10            então troque A[i] :=: A[j]
11            senão devolva j
```

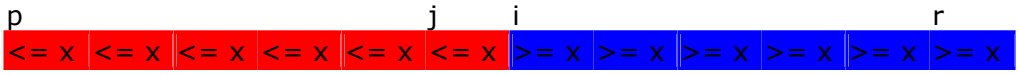
Para entender como e por que o algoritmo funciona de acordo com sua especificação, observe que no início de cada nova iteração do loop que começa na linha 4 temos as seguintes propriedades:

A[p..i] <= x , i < j , A[j..r] >= x .

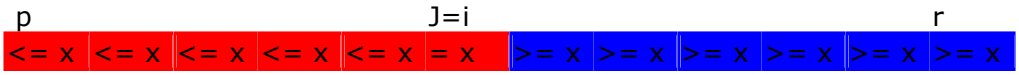
Na última passagem pela linha 4 o vetor A[i+1..j-1] consiste em

- zero ou mais elementos < x seguidos de zero ou mais elementos > x ou
- zero ou mais elementos < x , seguidos de exatamente um elemento igual a x , seguido de zero ou mais elementos > x .

No primeiro caso, o algoritmo chega à linha 11 com j = i-1.



No segundo caso, o algoritmo chega à linha 11 com j = i.



Não é difícil perceber agora que o algoritmo faz o que prometeu; em particular, que p <= j < r na linha 11.

O desempenho do algoritmo de partição

Quanto tempo o algoritmo PARTICIONE consome? Comecemos contando o número total de execuções da linha 5 (note que entre duas execuções da linha 5 podem ocorrer execuções das linhas 7, 8, 9 e 4). O valor de j diminui a cada execução da linha 5. O valor inicial de j é r+1 e o valor final é pelo menos i-1, que vale pelo menos p-2. Logo, o número total de execuções da linha 5 não passa de r-p+3. É claro que o número total de execuções da linha 6 também não passa de r-p+3.

Uma análise semelhente mostra que o número total de execuções da linha 7 não passa de r-p+3. O mesmo se pode dizer das linhas 8 e 9. O número de execuções da linha 4 também não pode passar de r-p+3.

Portanto, se a execução de cada linha consome 1 unidade de tempo, o consumo total de tempo não passa de 7(r-p)+10, ou seja, de

7n + 3 ,

onde n denota r-p+1, ou seja, o número de elementos do vetor A[p..r]. Esta delimitação vale sob a hipótese de 1 unidade de tempo por linha. Na realidade, diferentes linhas podem consumir diferentes quantidades de tempo, mas cada uma consome uma quantidade de tempo que não depende dos dados do problema. Se refizermos os cálculos levando isso em conta, veremos que o consumo de tempo de nosso algoritmo é

O(n) ,

Não é difícil verificar que o consumo de tempo do PARTICIONE também é Omega(n).

O algoritmo Quicksort

O algoritmo QUICKSORT recebe um vetor A[p..r] e rearranja o vetor em ordem crescente.

```
        QUICKSORT (A, p, r)
1         se p < r
2           então q := PARTICIONE (A, p, r)
3                QUICKSORT (A, p, q)
4                QUICKSORT (A, q+1, r)
```

(Note que q >= p e q < r; portanto os vetores A[p..q] e A[q+1..r] são estritamente menores que o vetor original A[p..r].)

Desempenho do Quicksort no pior caso

Quanto tempo o algoritmo consome no pior caso? Digamos que PARTICIONE consome não mais que 7n+3 unidades de tempo, onde n = r-p+1. Então o consumo de tempo do QUICKSORT no pior caso, digamos T(n), satisfaz a recorrência

T(1) = 1
T(n) <= 7n + 4 + maxk (T(k) + T(n-k)) , para n = 2, 3, . . .

onde o máximo é tomado sobre todos os possíveis valores de k no intervalo 1..n-1. Assim, por exemplo, T(2) <= 14+4+T(1)+T(1) = 20. Outro exemplo: T(3) <= 21+4+T(1)+T(2) <= 46. Mais um exemplo: T(4) <= 28+4+T(1)+T(3) <= 79, uma vez que T(1)+T(3) = 47 > 40 = T(2)+T(2). Em geral, vamos mostrar que

T(n) <= 7n2

para n = 2, 3, . . . A desigualdade é certamente verdadeira quando n = 2. Agora suponha que n > 2 e suponha que a desigaldade vale para T(k) sempre que k < n. Teremos então

T(n) <= 7n + 4 + maxk (T(k) + T(n-k))

<= 7n + 4 + 7 maxk (k2 + (n-k)2)

<= 7n + 4 + 7 (1 + (n-1)2)

<= 7n2 - 7n + 18

<= 7n2

pois n >= 3. Observe que o pior caso ocorre quando k vale 1 ou n-1, ou seja, quando PARTICIONE devolve p ou devolve r-1.

Se deixarmos de lado a hipótese grosseira de que cada linha "simples" do algoritmo consome 1 unidade de tempo, as conclusões ainda serão as mesmas: a recorrência será

T(n) <= max0 < k < n (T(k) + T(n-k)) + O(n) ,

e daí se deduz, imitando o que já fizemos acima, que T(n) = O(n2).

Por outro lado, não é difícil verificar que T(n) = Omega(n2): basta supor que o vetor A[p..r] que o algoritmo recebe já está em ordem estritamente crescente. Nesse caso, PARTICIONE devolve q = p e a recorrência se reduz a T(n) = T(1) + T(n-1) + O(n). Ademais, tanto A[p..q] quanto A[q+1..r] são crescentes.

Conclusão: O desempenho de pior caso do QUICKSORT é decepcionante. Por que então o algoritmo é tão popular?

Desempenho médio do Quicksort: preliminares

O consumo de tempo médio do QUICKSORT é bem melhor que O(n2). A verificação rigorosa deste fato não é fácil, mas a idéia intuitiva é simples: o consumo de tempo só chega perto de n2 quando o valor de q devolvido por PARTICIONE está sistematicamente muito próximo de p ou de r-1. A intuição sugere que isso deve ser raro.

Vamos fazer alguns cálculos grosseiros para adquirir alguma intuição sobre o comportamento médio do QUICKSORT. Suponha inicialmente que PARTICIONE sempre devolve um índice que

está a meio caminho entre p e r, ou seja, o vetor A[p..r] é dividido na proporção 1-para-1. Então o consumo de tempo, digamos T(n), do algoritmo satisfaz a recorrência

```
T(1)  =  1
T(n)  <=  T(chão(n/2)) + T(teto(n/2)) + 7n + 4      para n = 2, 3, . . .
```

Para simplificar, vamos resolver uma recorrência semelhante, que não tem chão nem teto. Para compensar a ausência do chão e teto é preciso estender a função a todos os número racionais positivos:

```
f(n)  =  1      para n <= 1
f(n)  <=  f(n/2) + f(n/2) + 7n + 4      para n > 1.
```

É fácil verificar que f(n) <= 10 n log2(n) para n >= 2. Ou seja, f(n) = O(n log(n)).

Agora suponha que PARTICIONE divide o vetor na proporção 8-para-1. Então o consumo de tempo do QUICKSORT obedece uma recorrência semelhante à seguinte:

```
g(n)  =  1      para n <= 1
g(n)  <=  g(n/9) + g(8n/9) + 7n + 4      para n > 1.
```

(Estamos sendo forçados a trabalhar com todos os valores racionais de n para que a recorrência faça sentido.) Por exemplo, g(9/8) <= g(1/8)+g(1)+63/8+4 = 1+1+63/8+4 = 111/8. Eu gostaria de mostrar que

```
g(n)  <=  13 n log9/8(n)
```

para todo n >= 9/8. (A base dos logaritmos é 9/8 apenas para simplificar alguns cálculos.) A desigualdade de fato vale para n = 9/8 pois g(9/8) = 111/8 enquanto 13 n log9/8(n) = 117/8. Agora suponha que n > 9/8. Então, por hipótese de indução, e usando sempre log na base 9/8,

5. Modularização

Para construirmos grandes programas, necessitamos fazer uso da técnica de modularização. Esta técnica faz com que dividamos um grande programa em pequenos trechos de código, onde cada qual tem uma função bem definida. Assim, além da facilidade em lidar com trechos menores, ainda podemos fazer uso da reutilização de código, já que estes trechos devem ser bem independentes.

Assim, definimos módulo como um grupo de comandos, constituindo um trecho de algoritmo, com uma função bem definida e o mais independente possível em relação ao resto do algoritmo.

A maneira mais intuitiva de trabalharmos com a modularização de problemas é definir-se um módulo principal de controle e módulos específicos para as funções do algoritmo. Módulos de um programa devem ter um tamanho limitado, já que módulos muito grandes são difíceis de serem compreendidos.

Os módulos são implementados através de procedimentos ou funções.

5.1 Procedimentos

Determina um trecho de algoritmo/programa que será executado somente no momento que o módulo principal necessitar.

Sintaxe de definição de um procedimento:

```
Procedimento Nome_do_Procedimento [ (parâmetros) ];

        <declaração das variáveis locais>

        sentença 1;
        sentença 2;
        sentença n;

Retorno
```

Os parâmetros podem ser passados **por valor** ou **por referência**. Um parâmetro passado por valor, não pode ser alterado pelo procedimento. Os parâmetros por referência são identificados usando-se o simbolo * antes de sua declaração.
Sintaxe da chamada do procedimento:

```
Nome_do_Procedimento (<lista de arâmetros>);
```

Os valores passados como parâmetros na chamada de um procedimento, devem corresponder sequencialmente à ordem declarada.

5.2 Função

Semelhante ao procedimento, todavia sua diferença consiste no fato de que um procedimento não retorna valor quando é chamado para execução, enquanto que a função retorna um valor.
Definição de uma função:

```
Função Nome_Função [ (parâmetros) ] : valor_retorno;

        < declaração de variáveis locais >

        sentença 1;
        sentença 2;
        sentença n;

Retorno variável;
```

Chamada da função:

```
Nome_Função (<lista de parâmetros>);

                Ou

Variável ← Nome_Função (<lista de parâmetros>);
```

Exemplos:

a) Procedimento

```
INICIO
        Inteiro: A, B;
        Leia(A,B);
        Maior(A,B);
FIM.
Procedimento Maior(X,Y)
        Inteiro: X, Y;
        Se X > Y então
              Escreva(X);
        Senão
              Escreva(Y);
        Fim-se;
Retorno;
```

b) Função

```
INICIO
        Inteiro: R, S;
        Leia(S);
        R ← Dobro(S);
        Escreva(R);
FIM.

Função Dobro(Z)
        Inteiro W, Z;
        W ← Z * 2;
Retorno W;
```

6. Variáveis Locais, Globais e Ponteiros

Determinam o modo na qual serão disponibilizadas as variáveis (conteúdo e/ou endereço) durante o processo de modularização do algoritmo/programa.

6.1 Variáveis Locais

Somente possuem seu valores dentro do procedimento/função que forma declaradas.
Exemplos:

```
INICIO
      Inteiro: R;
      R ← 5;
      Leitura();
      Escreva(R, X);
FIM.
Procedimento Leitura()
      Caracter: R, X;
      Leia(R,X);
      Escreva(R,X);
Retorno;
```

```
INICIO
      Inteiro: R;
      R ← 5;
      Leitura();
      Escreva(R, X);
FIM.
Função Leitura()
      Caracter: R, X;
      Leia(R,X);
      Escreva(R,X);
Retorno X;
```

6.2 Variáveis Globais ou Públicas:

Uma vez declarada, seus valores são mantidos por todo algoritmo/programa e não precisam ser passadas a outros procedimentos/função por parâmetro.
Exemplo:

```
INICIO
      Inteiro Global: R;
      Leitura();
      Escreva(R);
FIM.


Procedimento Leitura()
      Leia(R);
Retorno;
```

```
INICIO
      Real: B;
      B ← 5;
      Teste();
      B ← B + D;
      Escreva(B);
FIM.
Procedimento Teste()
      Real Global: D;
      Leia(D);
Retorno;
```

6.3 Variáveis Ponteiros

6.3.1 Definição de Ponteiros

Ponteiro é uma variável que contêm o endereço de outra variável ou de uma posição de memória conhecida. Se uma variável contém o endereço de uma outra variável, dizemos que a primeira aponta para a segunda.

Através de uma variável do tipo ponteiro pode-se ter acesso ao conteúdo de endereço de memória representado por tal variável.

Os ponteiros fornecem maneiras pelas quais as funções podem modificar os argumentos recebidos. Você poderá substituí-los por matrizes em muitas situações para aumentar a eficiência.

Além dos ponteiros serem uma das características mais fortes da linguagem C, também é a mais perigosa. Os ponteiros usados incorretamente podem provocar erros difíceis de serem encontrados.

É importante observar que uma variável do tipo ponteiro deve ser definida com o mesmo tipo de dado que ela irá endereçar, isto é, apontar. Significa dizer que uma variável ponteiro, do tipo real, por exemplo, só deverá apontar para dados também definidos como real.

6.3.2 Como Funcionam os Ponteiros

Os int guardam inteiros. Os float, double guardam números de ponto flutuante. Os char guardam caracteres, ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Voce anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.

Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de tipos diferentes.

No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro int aponta para um inteiro, isto é, guarda o endereço de um inteiro.

6.3.3 Operadores de Ponteiros

Operadores de ponteiros são importantes em C. Eles permitem que objetos sejam passados para funções, e que as funções modifiquem tais objetos.

Os dois operadores de ponteiros são: & e *.

6.3.3.1 & (operador de endereço)

Retorna o endereço da variável à que precede, não o valor da variável. Por exemplo, se o inteiro num está localizado no endereço 1000, então

y = #

coloca em y o endereço de memória da variável num. O símbolo " & " pode ser lido como " o endereço de ". A declaração y = &num pode ser lida como : "coloque o endereço de num em y ", ou então, "y recebe o endereço de num.

O algoritmo abaixo declara uma variável do tipo inteiro e uma variável do tipo real e exibe o valor e endereço de ambas :

Por exemplo:

```
Inicio
        inteiro  a;
        real  b;
        a ← 1;
        b ← 3.5;
        escreva ( " Valor = ", a , " Endereço =  ", &a );
        escreva ( " Valor = ", b , " Endereço =  ", &b );
Fim
```

6.3.3.2 * (variável ponteiro)

O valor da variável à que precede e usa este símbolo é classificada como endereço da informação na memória.

Por exemplo :

```
y = *num;
```

coloca o valor no endereço de num em y.
Pode ser lido como : "y recebe o valor no endereço num"

```
y = &num;
*y = 1995;
```

coloca o valor 1995 em num. O símbolo * pode ser lido como " no endereço ". Este exemplo poderia ser lido como : " coloque o valor 1995 no endereço y ". O operador * também pode ser usado do lado direito da declaração. Por exemplo,

```
y = &num;
*y = 1995;
v = *y / 10;
```

coloca o valor de 199 em v.

Os operadores & e * são chamados operadores de ponteiros por serem designados para trabalhar em variáveis ponteiro.

O comando : px = &x;
utiliza o operador & que fornece o endereço. Podemos interpretar este comando como :
- o operador & pode ser aplicado à variáveis
- o operador * trata seu operando com um endereço e acessa esse endereço para buscar o conteúdo.

A sequência :

```
x = &x;
y = *x;
```

atribui a y o mesmo valor atribuído no comando : y = x.

6.3.3.3 Declarando e Utilizando Ponteiros

Para declarar um ponteiro temos a seguinte forma geral:

tipo_do_ponteiro *nome_da_variável;

É o asterisco (*) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;
char *temp, *pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstânicias pode levar a um travamento do micro, ou a algo pior. O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado! Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador &. Veja o exemplo:

```
int count=10;
int *pt;
pt=&count;
```

Criamos um inteiro count com o valor 10 e um apontador para um inteiro pt. A expressão &count nos dá o endereço de count, o qual armazenamos em pt. Simples, não é? Repare que não alteramos o valor de count, que continua valendo 10.

Como nós colocamos um endereço em pt, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de count usando pt. Para tanto vamos usar o operador "inverso" do operador &. É o operador *. No exemplo acima, uma vez que fizemos pt=&count a expressão *pt é equivalente ao próprio count. Isto significa que, se quisermos mudar o valor de count para 12, basta fazer *pt=12.

Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador &. Ou seja, o operador & aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador * ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

7. Aplicação de variáveis dimensionadas heterogêneas

7.1 Registros e Campos

O conceito de registro visa facilitar o agrupamento de variáveis que não são do mesmo tipo, mas que guardam estreita relação lógica. Assim, registros correspondem a uma estrutura de dados heterogênea, ou seja, permite o armazenamento de informações de tipos diferentes. São localizados em posições de memória, conhecidos por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições, os campos. Vamos pensar que precisamos fazer um algoritmo que leia os dados cadastrais dos funcionários de uma empresa. Somente com os campos nome e salário já temos uma divergência de tipos – caracter e real. Assim, precisamos de uma estrutura de dados que permita armazenar valores de tipos de diferentes – estamos diante dos registros.
Nas matrizes, a individualização de um elemento é feita através de índices, já no registro cada elemento é individualizado pela referência do nome do campo.
Veja a sintaxe da declaração de um registro:

```
Registro: Nome_Registro
          Tipo: Nome_do_Campo;
          Tipo: Nome_do_Campo;
          ...
          Tipo: Nome_do_Campo;
Fim-registro;
```

Para trabalharmos com um registro, devemos primeiramente criar um tipo registro. Depois, declaramos uma variável cujo tipo será este tipo registro.

A sintaxe que representa o acesso ao conteúdo de um campo do registro é:

```
nome_registro.nome_campo;
```

A atribuição de valores aos registros é feita da seguinte forma:

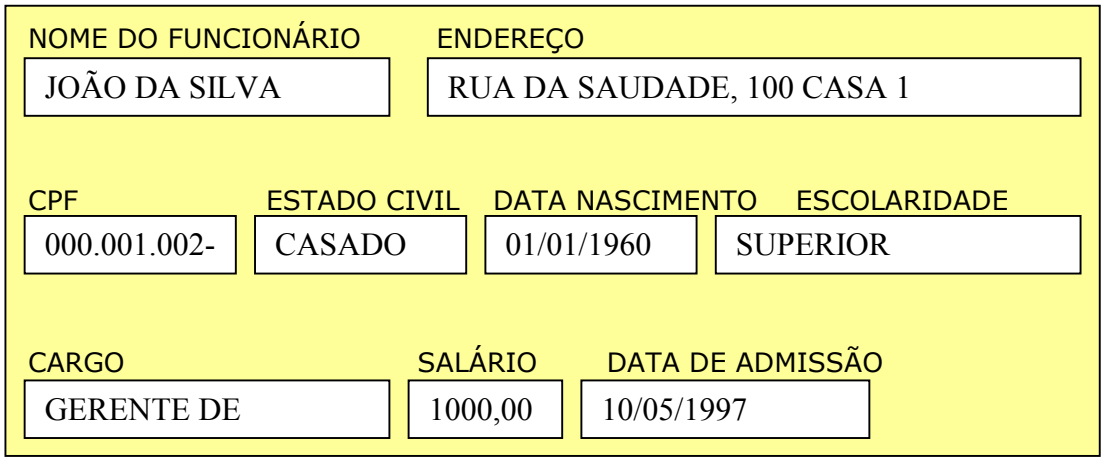
```
nome_registro.nome_campo ← valor;
```

Exemplo:

```
Registro: data
        Inteiro: dia;
        Caracter: mes;
        Inteiro: ano;
Fim-registro

Escreva ("Qual é o dia?");
Leia (data.dia);
Escreva ("Qual é o mes?");
Leia (data.mes);
Escreva ("Qual é o ano?");
Leia (data.ano);
Escreva("A data informada é ",data.dia,"/", data.mes,"/", data.ano)
```

Suponha uma aplicação onde devemos controlar os dados de funcionários da empresa. Imagine que tenhamos fichas onde os dados estão logicamente relacionados entre si, pois constituem as informações cadastrais do mesmo indivíduo.



Cada conjunto de informações do funcionário pode ser referenciável por um mesmo nome, como por exemplo, FUNCIONARIO. Tais estruturas são conhecidas como registros e aos elementos do registro dá-se o nome de campos.
Assim, definiríamos um registro da seguinte forma:

```
Registro: Funcionario
          Caracter: nome;
          Caracter: endereco;
          Caracter: cpf;
          Caracter: estado_civil;
          Caracter: data_nascimento;
          Caracter: escolaridade;
          Caracter: cargo;
          Real: salario;
          Caracter: data_admissao;
Fim-registro;
```

8. Arquivos

Na maioria das vezes, desejaremos desenvolver um algoritmo de forma que os dados manipulados sejam armazenados por um período longo de tempo, e não somente durante o tempo de execução do algoritmo. Como a memória principal do computador é volátil, ou seja, ao ser desligado o computador, todos os dados da memória são perdidos, necessitamos de uma memória auxiliar que seja permanente, como por exemplo, um disquete ou o disco rígido (HD).

Assim, passamos a ter um novo conceito no mundo computacional – o de arquivos. Arquivo é um conjunto de registros armazenados em um dispositivo de memória auxiliar (secundária). Por sua vez, um registro consiste de um conjunto de unidades de informação logicamente relacionadas – os campos. Assim, podemos definir que um registro corresponde a um conjunto de campos de tipos heterogêneos. Veja que neste momento estamos tratando de registros físicos, ao contrário, do que vimos no item anterior, que são os registros lógicos.
O fato do arquivo ser armazenado em uma memória secundária, o torna independente de qualquer algoritmo, isto é, um arquivo pode ser criado, consultado, processado e eventualmente removido por algoritmos distintos.

Sendo o arquivo uma estrutura fora do ambiente do algoritmo, para que este tenha acesso aos dados do arquivo é necessária a operação de leitura do registro no arquivo. As operações básicas que podem ser feitas em um arquivo através de um algoritmo são: obtenção de um registro, inserção de um novo registro, modificação ou exclusão de um registro.

A disposição dos registros no arquivo – organização – oferece ao programador formas mais eficientes e eficazes de acesso aos dados. Vamos considerar, aqui, as duas principais formas de organização de arquivos: a seqüencial e a direta.

- **Organização Seqüencial**: A principal característica da organização seqüencial é a de que os registros são armazenados um após o outro. Assim, tanto a leitura quanto a escrita, são feitas seqüencialmente, ou seja, a leitura de um determinado registro só é possível após a leitura de todos os registros anteriores e a escrita de um registro só é feita após o último registro.

- **Organização Direta**: A principal característica da organização direta é a facilidade de acesso. Para se ter acesso a um registro de um arquivo direto, não é necessário pesquisar registro a registro, pois este pode ser obtido diretamente – acesso aleatório. Isto é possível porque a posição do registro no espaço físico do arquivo é univocamente determinada a partir de um dos campos do registro (chave), escolhido no momento de criação do arquivo.

O acesso a um arquivo dentro do algoritmo é feito através da leitura e escrita de registros.
No algoritmo, o arquivo deve ser declarado e aberto, antes que tal acesso possa ser feito.
No final do algoritmo, ou quando houver necessidade, o arquivo deve ser fechado.

A sintaxe da declaração de arquivos é :

arquivo: nome_arquivo (nome_registro, organização);

onde:

nome_registro – nome do registro lógico que será usado para se ter acesso aos registros físicos do arquivo.

organização – indica o tipo de organização do arquivo, que pode ser seqüencial ou direta.

Exemplo:

Registro: Registro_Endereco
Caracter: rua, bairro, cidade, uf, cep;
Inteiro: numero;
Fim-registro;
Arquivo: Agenda (***Registro_Endereço***, *Seqüencial*);

8.1 Abertura de Arquivos

A declaração do arquivo é a definição, para o algoritmo, do modelo e dos nomes que estarão associados à estrutura de dados, isto é, ao arquivo.
A sintaxe da declaração de arquivos é :

Abrir (nome_ arquivo, tipo_abertura) ;

onde:

tipo_abertura → especifica se o arquivo será usado para:
- leitura
- escrita
- leitura/escrita (caso não seja especificado o tipo_abertura este será automaticamente acionado)

Exemplo:

Abrir (Agenda, leitura);
Abrir (Agenda, escrita);
Abrir (Agenda);
Abrir (Agenda, Contas_Pagar);

8.3 Fechamento de Arquivos

Para se desfazer a associação entre o modelo e o arquivo físico, usa-se o comando de fechamento de arquivos, cuja sintaxe está a seguir:

Fechar (nome_ arquivo);

8.4 Visualização dos dados de Arquivos

Para ver os dados armazenados dentro de um arquivo usa-se a sintaxe a seguir:

Escrever (nome_arquivo);

8.5 Inserindo dados em Arquivos

Para gravar as informações dos registros dentro de um arquivo de dados usa-se a sintaxe de comando seguir:

Inserir/Gravar (nome_arquivo);

Usamos o comando Inserir quando os dados ainda não existem no arquivo, e Gravar quando estamos alterando algo que já existe dentro do arquivo.

8.6 Pesquisa de registro num arquivo

Para encontrar informações sobre registros dentro de um arquivo de dados usa-se a sintaxe de comando seguir:

Pesquisar (nome_arquivo.nome_campo, valor);

Exemplo:

```
Algoritmo Pesquisa1

Inicio

    // declarações

    Inteiro: mat;
    Lógico: Encontrou;
    Registro: Dados_Aluno
                Inteiro: matricula;
                Caracter: nome, data_nascimento, endereco;
    fim-registro;
    Arquivo: Alunos( Dados_Alunos, Seqüencial );

    Abrir (Alunos, leitura);

    Encontrou ← Falso;
    Escreva (" Qual o número de  matricula que deseja procurar? ");
    Leia ( mat )
    Se Pesquisar( Alunos.Matricula , mat ) então
              Escrever('Aluno encontrado');
    Fim-se;

    Fechar Alunos;
Fim
```

9. Tabela ASCII

Tabela ASCII

Caracteres normais

Binário	Decimal	Hex	Gráfico
0010 0000	32	20	(vazio) (sp)
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binário	Decimal	Hex	Gráfico
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binário	Decimal	Hex	Gráfico
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~
0111 1111	127	7F	Delete

10. Referências Bibliográficas

SEBESTA, Robert W. Conceitos de linguagem de programação, 4ª edição. Editora Bookman, ano 2000.

FARRER, Harry, BECKER, Christiano G., FARIA, Eduardo C., MATOS, Helton Fábio de, SANTOS, Marcos Augusto dos, MAIA, Miriam Lourenço. Algoritmos Estruturados. Editora Guanabara, 1989.

GUIMARÃES, Angelo de Moura, LAGES, Newton A de Castilho. Algoritmos e estruturas de dados. Rio de Janeiro. LTC – Livros Técnicos e Científicos Editora, 1985.

SALVETTI, Dirceu Douglas, BARBOSA, Lisbete Madsen. Algoritmos. Editora Makron Books, 1998.

SILVA, Joselias Santos da. Concursos Públicos – Raciocínio Lógico. R&A Editora Cursos e Materiais Didáticos, 1999.

WIRTH, Niklaus. Algoritmos e Estruturas de Dados. Editora Prentice-Hall do Brasil, 1986.

SILVA, Cristiano Vieira. Apostila de Algoritmos – Introdução a Lógica de Programação, Colégio Universitas – Cursos Técnico em Informática. Edição particular, publicação ©2004 – A utilização deste material será somente sob autorização e a fonte deverá ser citada.