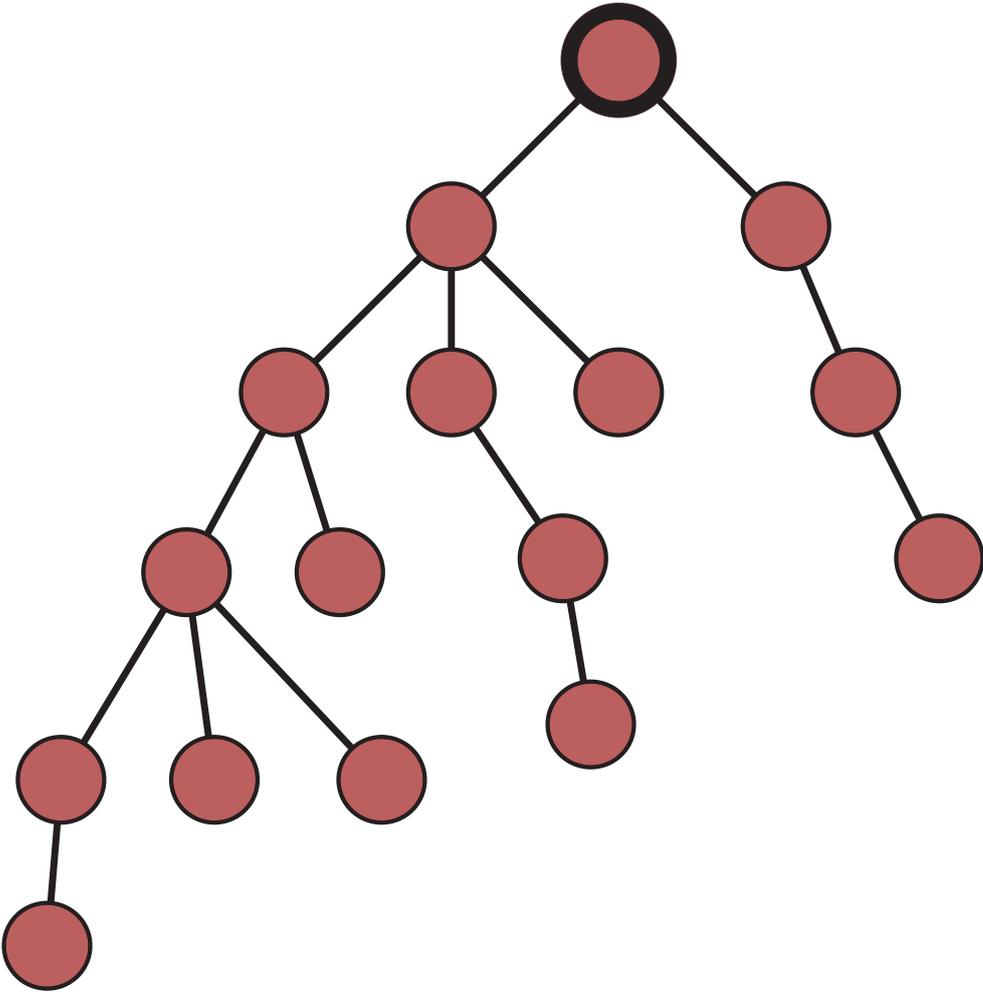


Informática
&
Técnicas de Programação (VBA)
V01.2

Alexandre Checoli Choueiri



20 de abril de 2023

Conteúdo

1	Introdução	5
1.1	Algoritmos	5
1.2	Representação de Algoritmos	6
1.2.1	Fluxogramas	6
1.2.2	Memória e variáveis	8
1.2.3	Como o computador executa as ações & Pseudocódigos	9
1.3	Entrada e saída de dados	20
1.4	Teste de mesa	21
1.5	Quadro resumo	21
1.6	Exercícios	21
1.6.1	Variáveis	21
1.6.2	Condicionais	21
1.6.3	Pseudocódigos/Fluxogramas, input/output	22
1.6.4	Laços de repetição e Estruturas de dados	24
2	VBA Inicial	29
2.1	Configuração	29
2.2	O primeiro código	30
2.3	Declaração de variáveis, comentários e verificação imediata	31
2.4	Operadores matemáticos e de comparação numérica	33
2.5	Condicionais e operadores lógicos	34
2.6	Laços de repetição	35
2.6.1	Laço For	35
2.6.2	Laço While	36
2.7	Input/Output	37
2.7.1	Input: Inputbox	37
2.7.2	Input: Cells e Range	37
2.7.3	Input: Arquivo de texto	39
2.7.4	Output: MsgBox	41
2.7.5	Output: Cells e Range	42
2.7.6	Output: Arquivos	43
2.8	Estrutura de dados: arrays	43
2.8.1	Estáticos vs dinâmicos	44
2.8.2	Vetores multidimensionais	45
2.8.3	Métodos mais usados	46
2.9	Subrotinas	48
2.9.1	Conceitos gerais	48
2.10	Subrotinas em VBA	51
2.10.1	Subs	51
2.10.2	Function	52
2.11	Exercícios	53
2.11.1	Condicionais	53
2.11.2	Input/Output	53

2.11.3	Vetores	55
2.11.4	Matrizes	56
2.11.5	Subrotinas - subs vs functions	58
3	VBA Intermédiario	61
3.1	Estruturas de dados	61
3.1.1	Collections	61
3.1.2	Dictionaries	62
3.1.3	Type	63
3.2	Input de dados	64
3.3	Classes	65
3.3.1	Criação e utilização	65
3.3.2	O problema da cópia	67
3.3.3	O problema da cópia/atribuição	69
3.3.4	Criando cópias	69
3.3.5	Criando métodos de cópia	70
4	Trabalhos	73
4.1	Calculo/Algebra	73
4.1.1	Cálculo de integrais	73
4.1.2	Cálculo de derivadas	74
4.1.3	Cálculo de raízes de funções	74
4.1.4	Multiplicação de matrizes	76
4.1.5	Matriz inversa	76
4.1.6	Resolução de sistemas lineares	76
4.2	Eng. de produção	76
4.2.1	Cálculo do BOM (Bill of Materials)	76
4.2.2	Agrupamento de máquinas (Algoritmo ROC)	77
4.3	Pesquisa Operacional/ Otimização	78
4.3.1	O algoritmo Simplex	78
4.3.2	O problema do caixeiro viajante - (<i>TSP - Traveling Salesman Problem</i>)	78
4.3.3	O problema do roteamento de veículos - (<i>VRP - Vehicle Routing Problem</i>)	80
4.3.4	O problema da mochila - <i>Bin Packing Problem</i>	80
4.3.5	O problema das p-mediana (<i>k-means</i>)	80
4.4	Miscelânea	80
4.4.1	Ordenação Bubble Sort	80

Lista de Figuras

1.1	Fluxo de um algoritmo	5
1.2	Componentes do fluxograma	6
1.3	Algoritmo compras no mercado	7
1.4	Algoritmo compras no mercado 2	7
1.5	Algoritmo compras no mercado 3	8
1.6	Endereços e memória em um computador	8
1.7	Etapas na memória do algoritmo de soma	9
1.8	Variáveis e memória RAM	9
1.9	Fluxograma Custo maçãs	13
1.10	Fluxograma Custo maçãs	13
1.11	Memória alocada para notas de alunos	16
1.12	Memória alocada para um vetor com 5 elementos (os elementos são contíguos na memória)	17
1.13	Resumo do capítulo	27
2.1	Área inicial VBA	30
2.2	Área inicial VBA com módulo	31
2.3	Índices células Excel	38
2.4	Range "A1:A7"	38
2.5	Range "A1:F4"	39
2.6	Lendo matriz linha a linha	40
2.7	Lendo arquivo com indicação de linhas a serem lidas	41
2.8	Execução da subrotina	48
2.9	Execução da subrotina	49
2.10	Passagem de valores à subrotina	50
2.11	Alterando o valor do parâmetro	50
2.12	Alterando o valor do parâmetro	50
2.13	Alterando o valor do parâmetro	51
2.14	Quebra cabeça Excel	54
2.15	Lendo matriz dos dados	54
2.16	Lendo e somando coordenadas	54
2.17	Leitura de vetor com 5 elementos	55
2.18	Leitura de coordenadas	57
2.19	Matriz de distâncias	57
2.20	Swap de elementos	58
2.21	Separação do problema em 3 partes	59
3.1	Seleção da aba <i>Microsoft Scripting Runtime</i>	61
3.2	Arquivo a ser lido, separado por espaços	65
3.3	Classes para a solução do VRP	66
3.4	Programa com os módulos de classes	67
3.5	Solução criada no exemplo	68
3.6	Solução criada no exemplo sem usar o set	68

4.1	Integral de f	73
4.2	Integral de f	74
4.3	Derivada de f	75
4.4	Raiz de f	75
4.5	Árvore de produto	76
4.6	Solução para o caixeiro viajante	78
4.7	Solução infactível para o caixeiro-viajante	79
4.8	Rotas factíveis para o mesmo PCV	79

Lista de Tabelas

1.1	Artigos com algoritmos explicados por Fluxograma	7
1.2	Tipos de dados	10
1.3	Artigos com algoritmos explicados por pseudocódigo	10
1.4	Operadores matemáticos	14
1.5	Operadores de comparação numérica	14
1.6	Operadores lógicos	15
1.7	Tabela-verdade And	15
1.8	Tabela-verdade Or	15
1.9	Tabela-verdade Xor	15
1.10	Preço da ton. de chapa	24
2.1	Operadores matemáticos	33
2.2	Operadores de comparação numérica	33
2.3	Operadores lógicos	35
2.4	Botões MsgBox e valores numéricos	42
2.5	Dedução de custos compra do aço	58
2.6	Selo concedido à empresa	59

Capítulo 1

Introdução

1.1 Algoritmos

Existem muitas definições para algoritmos, de forma geral, *um algoritmo pode ser descrito como um passo-a-passo para a resolução de um problema*. Note que um algoritmo não precisa ser executado por um computador. A Figura 1.1 ilustra os passos genéricos de um algoritmo, pela definição de Knuth.

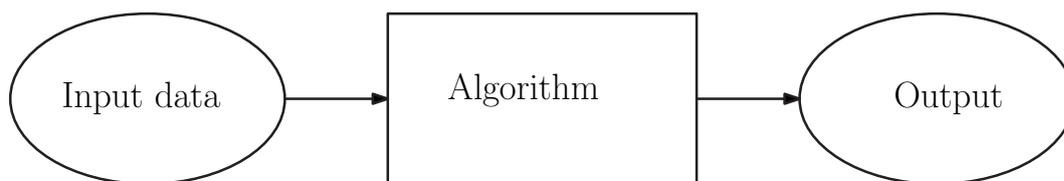


Figura 1.1: Fluxo de um algoritmo

Definição formal de um algoritmo:

Um algoritmo é um procedimento finito, definido e eficaz, que produz uma saída.

Donald E. Knuth (1968)

- Finito: Deve haver um fim para ele, em um tempo considerável.
- Definido: Precisamente definido, em termos claros e sem ambiguidades.
- Eficaz: Os passos descritos precisam ser passíveis de execução.
- Procedimento: A sequência de passos definidos.
- Saída: Se não houver algum tipo de saída, o resultado é indeterminado.

Propriedades desejadas de um bom algoritmo:

- a Ele resolve o problema de forma **correta**.
- b Pode ser executado (consideravelmente) **rápido**.
- c Requer (consideravelmente) **pouco espaço**.
- d É simples.

Os objetivos acima são, na maioria das vezes, conflitantes, de forma que é necessário encontrar um *trade-off* entre velocidade e memória utilizada.

Os algoritmos estão presentes em todos os lugares, não somente na computação. Imagine os problemas abaixo e tente enxergá-los por uma visão algorítmica, ou seja: entradas, transformação e saídas.

1. Realizar compras em um mercado

2. Preparar um bolo
3. Encontrar o menor entre dois números
4. Encontrar o menor entre três números
5. Encontrar as raízes de uma equação de segundo grau
6. Encontrar a soma de dois números

Para cada um dos exemplos acima, desenhe um fluxograma como o da 1.1 expondo as entradas e saídas de todos os problemas (sem a definição do algoritmo em si).

1.2 Representação de Algoritmos

Para que possamos criar algoritmos, primeiro precisamos aprender como representá-los. Podemos representar um algoritmo de 3 formas diferentes, por descrições narrativas, fluxogramas e pseudo-códigos.

Descrição narrativa: Consiste em analisar o enunciado do problema e escrever, utilizando linguagem natural, os passos a serem seguidos para sua resolução (receita de bolo).

Fluxograma: Consiste em analisar o enunciado do problema e escrever, utilizando símbolos gráficos, os passos a serem seguidos para sua resolução.

Pseudocódigo: Consiste em analisar o enunciado do problema e escrever, por meio de regras predefinidas (uma *sintaxe* própria), os passos a serem seguidos para sua resolução.

1.2.1 Fluxogramas

A Figura 1.2 apresenta os principais componentes para se utilizar fluxogramas na descrição de algoritmos (existem muitos outros).

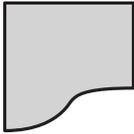
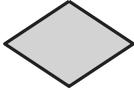
	Início e fim do algoritmo
	Sentido do fluxo de dados
	Calculos e atribuição de valores
	Entrada de dados
	Saída de dados
	Decisão

Figura 1.2: Componentes do fluxograma

Uma representação possível para o algoritmo de realizar compras no mercado por um fluxograma pode ser vista na Figura 1.3.

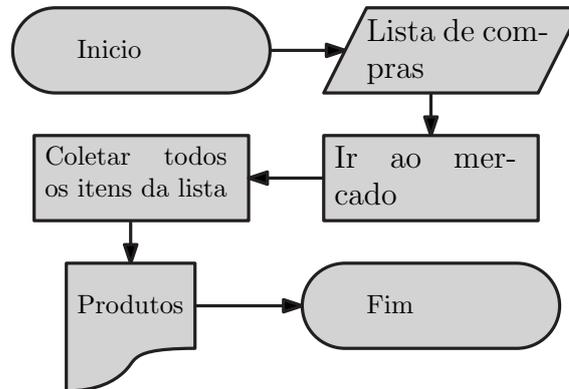


Figura 1.3: Algoritmo compras no mercado

Ainda podemos refinar o algoritmo, a etapa "Coletar todos os itens da lista" está muito simplificada, pode ser que alguns itens não sejam encontrados. Para modelar esta situação, precisa-se, de alguma forma, representar todos os itens da lista individualmente, e ainda definir quais ações devem ser tomadas no caso de um item não ser encontrado.

A Figura 1.4 apresenta um fluxograma detalhando mais o processo de compras, neste caso, todos os itens da lista são verificados individualmente. [Cormen et al., 2001]

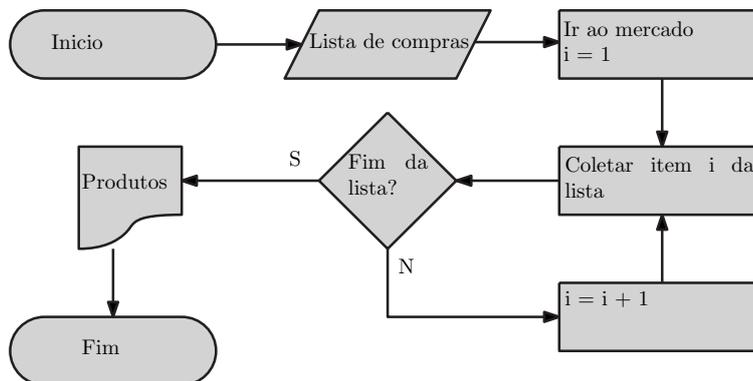


Figura 1.4: Algoritmo compras no mercado 2

Podemos deixar o algoritmo mais específico ainda, se especificarmos o que a pessoa deve fazer caso um produto da lista não esteja disponível, o fluxograma da Figura 1.5 representa o cenário em que, quando a pessoa não encontra um item da lista, ela deve procurar outro equivalente:

A Tabela 1.1 a seguir apresenta alguns artigos científicos que exemplificam seus algoritmos sob a forma de Fluxogramas:

Artigo	Algoritmo
[George and Robinson, 1980]	Carregamento de contêineres
[Pongcharoen et al., 2004]	Sequenciamento de itens multi-nível (<i>scheduling</i>)
[Tuzun and Burke, 1999]	Localização e roteirização

Tabela 1.1: Artigos com algoritmos explicados por Fluxograma

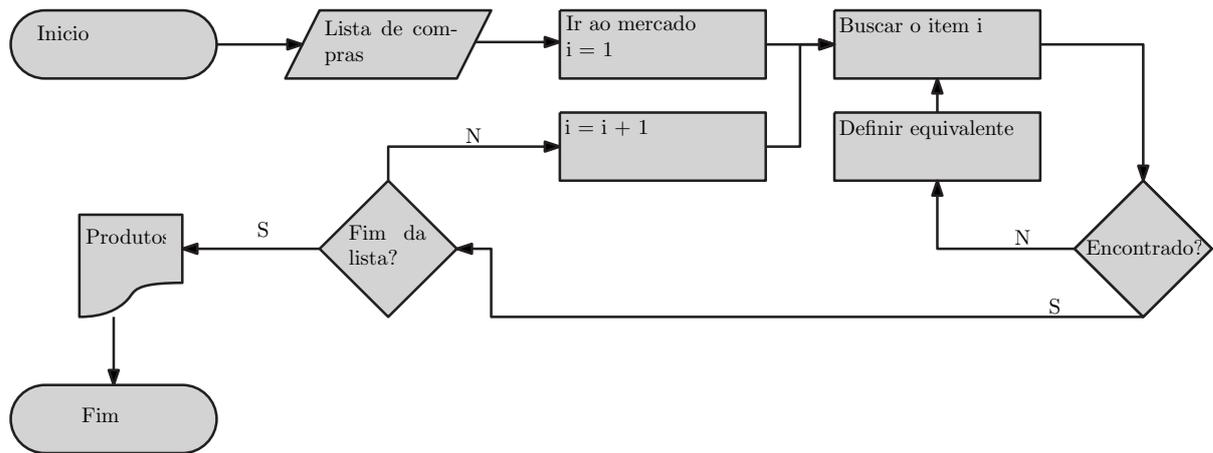


Figura 1.5: Algoritmo compras no mercado 3

1.2.2 Memória e variáveis

Para que possamos manipular dados nos algoritmos, utilizamos da memória do computador. A memória é o local onde os dados são armazenados e pode ser entendida como uma sequência de células nas quais se pode armazenar um valor correspondente a um dado.

Cada parte da memória de acesso aleatório (RAM - random access memory) tem o seu *endereço* próprio e individual. Um endereço de memória é exatamente como um endereço de uma casa, ou seria, se as seguintes condições fossem verdadeiras:

1. Cada casa é numerada em ordem
2. Não há números pulados ou repetidos
3. A cidade inteira consiste em uma rua comprida

Assim, o primeiro endereço poderia ser 0X1000, o segundo 0x1001 e assim por diante. OBS: Por convenção os endereços de memória são expressos em hexadecimais. Veja a Figura 1.6.

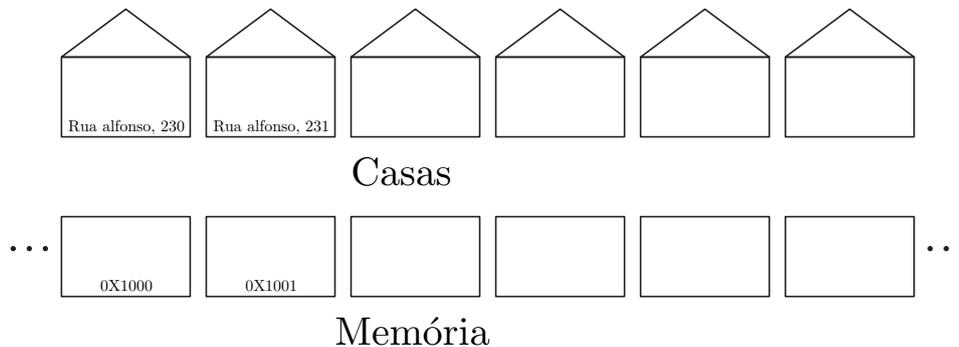


Figura 1.6: Endereços e memória em um computador

Cada endereço de memória tem um *espaço* pré definido para que dados sejam armazenados, geralmente a menor parte de endereçamento de memória é de um *byte*. Esse é o tamanho de um número inteiro. Todas as operações que realizarmos em nossos algoritmos que dependam de dados, precisam usar a memória do computador, de forma que precisamos requisitar a utilização para o computador (quem é o "gerente" da memória nos computadores é o sistema operacional).

Suponha que vamos escrever um programa que armazene dois números, realize a soma dos mesmos e em seguida armazene a soma em um novo espaço da memória. A sequência ficaria assim:

1. Por favor computador, reserve o endereço de memória 0X1000 e aloque o número 10 neste espaço.

2. Por favor computador, reserve o endereço de memória 0X1002 e aloque o número 20 neste espaço.
3. Por favor computador, reserve o endereço de memória 0X1003 e aloque o número contido no endereço 0x1001 somado ao número contido no endereço 0X1002.

Um esquema do algoritmo é mostrado na Figura 1.7.

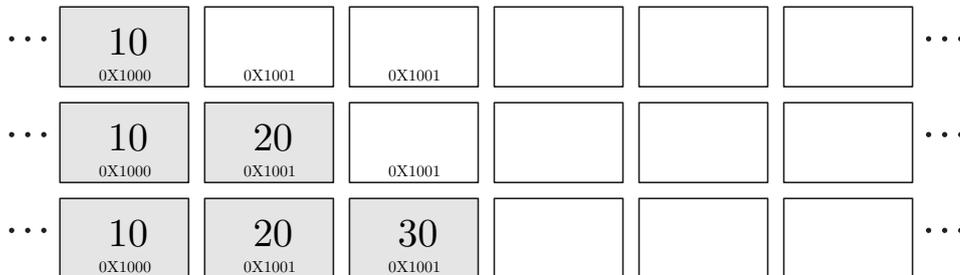


Figura 1.7: Etapas na memória do algoritmo de soma

Felizmente, não precisamos conhecer o endereço da memória quando requisitamos seu uso. Todas as linguagens de programação possuem o recurso chamado de *variáveis*. A variável é uma abstração da célula de memória, com um nome da escolha do programador. Dessa forma podemos nomear o tipo de dado que queremos armazenar (por exemplo, um inteiro), e o sistema operacional se encarrega de encontrar um espaço de memória que caiba o tipo que requisitamos, e dar o nome que queremos para ele.

No caso do exemplo acima, podemos fornecer nomes sugestivos para as variáveis: o primeiro número pode se chamar `num1`, o segundo `num2` e o resultado da soma dos dois `soma`. O algoritmo fica então:

Algoritmo: Soma de números

1. Reserve `num1`, do tipo inteiro
2. `num1 = 10`
3. Reserve `num2`, do tipo inteiro
4. `num2 = 20`
5. Reserve `soma`, do tipo inteiro
6. `soma = num1 + num2`

A memória fica então como na Figura 1.8:



Figura 1.8: Variáveis e memória RAM

Todas as linguagens de programação permitem a alocação de memória com o nome de variáveis. Esse processo é chamado de definição (ou declaração) de variáveis. Como podemos salvar diversos tipos de informação (no caso acima só usamos números inteiros), e cada informações requer um número diferente de espaços na memória, muitas linguagens de programação (VBA, C++, JAVA) exigem que o *tipo* do dado seja especificado. Alguns tipos de dados usados na linguagem VBA são mostrados na Tabela 1.2:

1.2.3 Como o computador executa as ações & Pseudocódigos

Por meio de uma linguagem de programação escrevemos as ações que o computador deve executar. Ele não faz nada que não está escrito, e segue a execução iniciando em um ponto de partida, e

Dados	Espaço de armazenamento	Explicação
Double	8 bytes	Números reais
Single	4 bytes	Números reais
Integer	2 bytes	Números inteiros
Date	8 bytes	Datas
Integer	2 byte	Números inteiros
Boolean	2 byte	Verdadeiro ou Falso (True/False)
String	10 bytes + tamanho da cadeia de caracteres	Palavras
Variant	-	Aceita qualquer tipo de dados

Tabela 1.2: Tipos de dados

executa linha a linha do código, até que o programa seja finalizado. Por exemplo, o Algoritmo 1.2.2 (Soma de números) seria executado na ordem de numeração das linhas. Note que o Algoritmo não está escrito em nenhuma linguagem de programação, mas sim em pseudocódigo.

Pseudocódigos

Pseudocódigos são um intermediário entre as linguagem de programação de computadores e a linguagem dos humanos. Elas servem para estruturarmos um código que será programado para um computador, de forma menos "compromissada". Uma vez com o pseudo criado, o mesmo pode ser implementado em qualquer linguagem de programação.

Para se escrever um algoritmo em pseudo-código, primeiro deve-se definir (ou utilizar) uma notação para as palavras-chave da pseudo linguagem e as regras de utilização das mesmas (sintaxe). As palavras chave da pseudo-linguagem utilizada neste material serão mostradas ao longo dos exemplos.

A Tabela 1.3 apresenta alguns artigos científicos que usam pseudocódigos para explicar seus algoritmos.

Artigo	Algoritmo
[Henn and Wäscher, 2012]	"Order batching problem"
[Cohen, 1995]	Inferência de regras de decisão
[Nagata, 2007]	Roteirização de veículos

Tabela 1.3: Artigos com algoritmos explicados por pseudocódigo

Em nosso pseudocódigo utilizaremos a seguinte *sintaxe* para definir variáveis:

Dim <nome_da_variavel> As <tipo_da_variavel>

Podemos pensar na palavra "Dim" como "reserve um espaço na memória" de nome <nome_da_variavel> e do tipo <tipo_da_variavel>. Existem algumas regras que devem ser seguidas para se nomear as variáveis, são elas:

1. Você deve usar uma letra como o primeiro caractere
2. Não é possível usar um espaço, ponto (.), ponto de exclamação (!) @ ou os caracteres , &, \$ e no nome.

Podemos declarar diversas variáveis em uma mesma linha, separando-as por vírgula.

EXEMPLO: Considere o pseudocódigo a seguir, você consegue identificar qual será o valor final da variável A?

Algoritmo: Descubra A

1. Dim **A** as Integer, **B** as Integer, **C** as Integer
2. **A** = 10
3. **B** = 5
4. **C** = -10
5. **A** = **A** + 3
6. **B** = **B** - 2
7. **A** = **A** - **B**

Condicionais

Podemos ainda inserir *condicionais* em nosso pseudocódigo. As condicionais permitem definir partes do código que serão ou não executadas, de acordo com alguma condição. Por exemplo, considere o pseudocódigo a seguir. Qual o valor de A no fim deste código?

Algoritmo: Descubra A condicional

1. Dim **A** as Integer, **B** as Integer, **C** as Integer
2. **A** = 10
3. **B** = 5
4. **B** = **B** - 2
5. Se (**B** < -10) faça:
 - 5.1. **A** = **A** - **B**
6. **C** = -10

Note que a instrução 5.1 só será executada se a condição (**B** < -10) for verdadeira, **caso contrário o programa pula a linha 5.1** e passa direto para a execução da linha 6.

Podemos ainda incrementar as condicionais, indicando o que o programa deve fazer no caso de uma condição ser falsa. Podemos encadear diversas condicionais, porém se todas estiverem no mesmo nível, sempre que uma condição for aceita o programa pula para o fim de todas as condicionais! Considere o exemplo a seguir (Idade mínima para beber), o que será armazenado na variável *mensagem* na linha 7?

Algoritmo: Idade mínima para beber

1. Dim **idade** as Integer
2. Dim **mensagem** as String
3. **idade** = 15
4. Se (**idade** < 5) faça:
 - 4.1. **mensagem** = "Você ainda é uma criança!"
5. Senão, se (**idade** < 18) faça:
 - 5.1. **mensagem** = "Tão perto e ao mesmo tempo tão longe..."
6. Senão faça:
 - 6.1. **mensagem** = "Parabéns! Pegue um copo e fique a vontade"
7. Imprima **mensagem**

Note que as linhas 4,5 e 6 fazem parte da mesma condicional encadeada, ou seja, se o programa executar 4.1 por exemplo, ele sai de todas as condicionais encadeadas no mesmo nível (4, 5 e 6) e executa a linha 7.

Podemos ainda aninhar condicionais, ou seja, uma condicional que será avaliada somente se outra condicional, um nível acima dela tiver sido avaliada como verdadeira. Considere o seguinte problema:

EXEMPLO: Um mercado vende 2 tipos de maçãs, M1 e M2, se o cliente comprar menos de 6 maçãs do tipo M1 o custo unitário/maçã é de R\$0.50, caso compre 6 ou mais o custo unitário passa a ser de R\$0.30. A mesma coisa acontece com as maçãs do tipo M2, porém abaixo de 10 unidades o

preço é de R\$0.40 e a partir de 10 passa a ser de R\$0.20. OBS: Considere que uma pessoa só possa comprar maçãs de um dos dois tipos. O algoritmo abaixo calcula o custo da compra de 10 maçãs do tipo M1.

Algoritmo: Custo maçãs

1. Dim `tipo_maca` as String
2. Dim `num_maca` as Integer
3. Dim `custo` as Integer
4. `tipo_maca = "M1"`
5. `num_maca = 10`
6. Se (`tipo_maca = "M1"`) faça:
 - 6.1. Se (`num_maca < 6`) faça:
 - 6.1.1. `custo = 0.5 * num_maca`
 - 6.2. Senão
 - 6.2.1. `custo = 0.3 * num_maca`
7. Senão, se (`tipo_maca = "M2"`) faça:
 - 7.1. Se (`num_maca < 10`) faça:
 - 7.1.1. `custo = 0.4 * num_maca`
 - 7.2. Senão
 - 7.2.1. `custo = 0.2 * num_maca`

Novamente, note que existem condicionais em 2 níveis, um na linha 6 e 7, e dentro de cada um destes um condicional aninhado, 6.1 e 6.2 aninhado a 6 e 7.1 e 7.2 aninhado a 7. Lembre-se sempre da regra: se uma condição for verdadeira, somente o trecho desta condição é executado. Podemos simplificar o código da seguinte forma para entendermos o que está acontecendo (escondemos os níveis mais aninhadas de condicionais e vamos entendendo por partes):

Algoritmo: Custo maçãs simplificado

1. Dim `tipo_maca` as String
2. Dim `num_maca` as Integer
3. Dim `custo` as Double
4. `tipo_maca = "M1"`
5. `num_maca = 10`
6. Se (`tipo_maca = "M1"`) faça:
 - 6.1. PARTE 1
7. Senão, se (`tipo_maca = "M2"`) faça:
 - 7.1. PARTE 2

Agora, entendido o código simplificado, podemos substituir PARTE1 e PARTE2 pelos códigos originais. Esta é uma prática muito comum para algoritmos muito complexos, deixamos uma "indicação" do que será pensado e implementado no futuro. A Figura 1.9 mostra um fluxograma do algoritmo Custo maçãs:

Já a Figura 1.10 mostra onde estão as partes simplificadas do algoritmo Custo maçãs simplificado.

Operadores

Note que neste algoritmo usamos *operadores matemáticos* e *operadores de comparação*. Operadores matemáticos são aqueles usados para realizar operações matemática. São mostrados na Tabela 1.4:

As operações matemáticas têm a seguinte ordem de precedência:

1. Multiplicações, Divisões e Exponenciais
2. Módulo (operador para encontrar o resto da divisão)
3. Somas e Subtrações

E as operações são executadas na ordem \rightarrow .

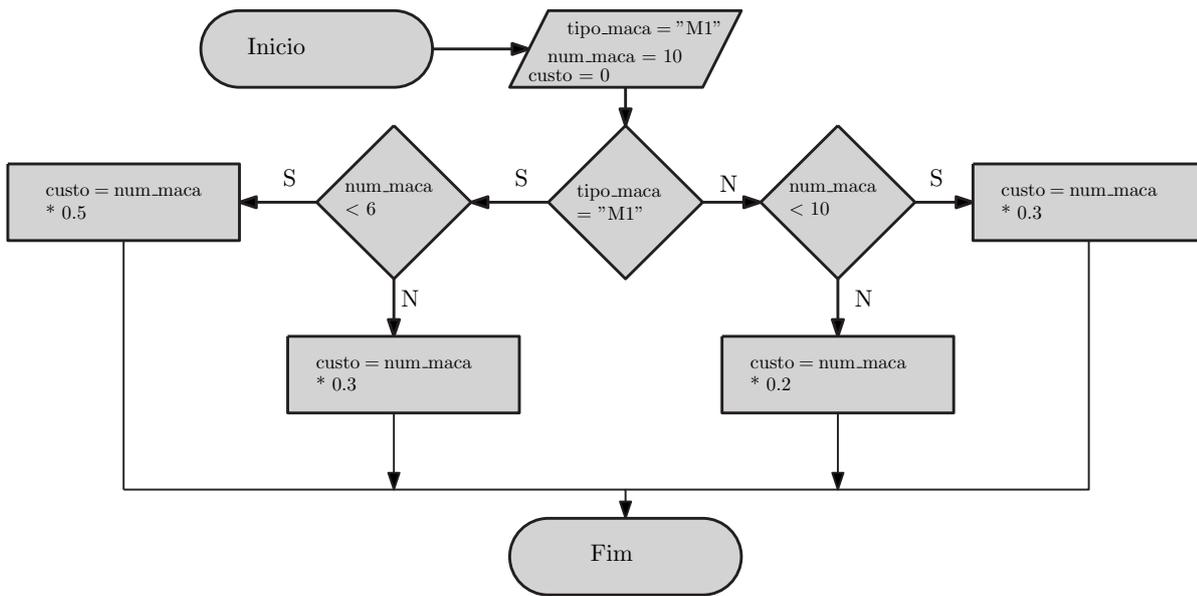


Figura 1.9: Fluxograma Custo maçãs

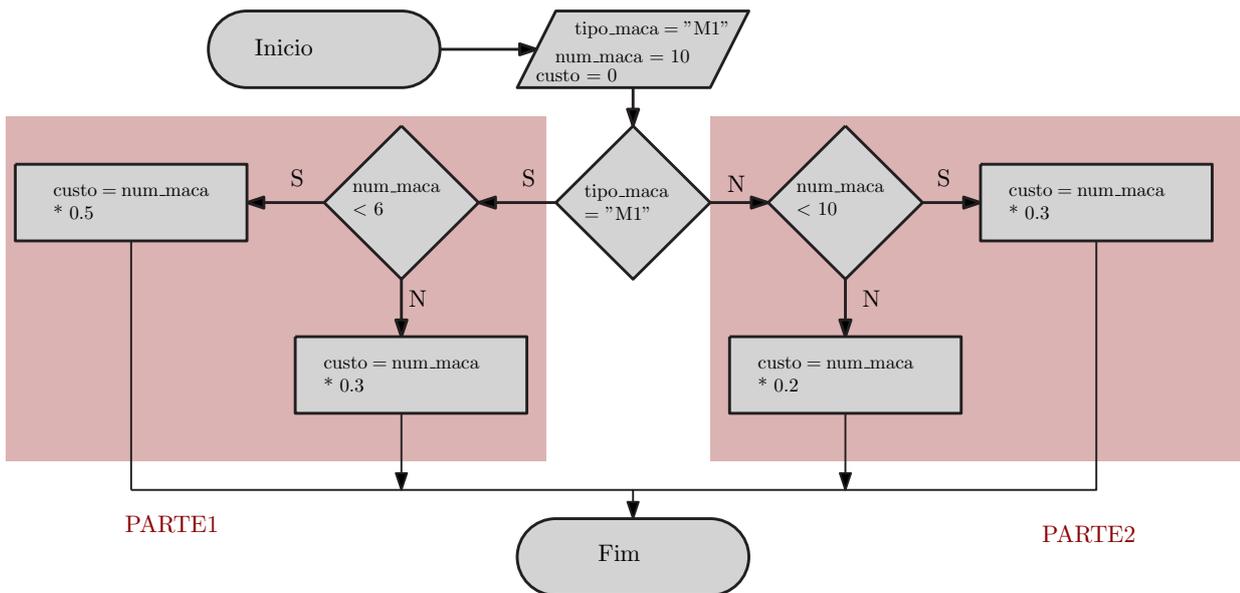


Figura 1.10: Fluxograma Custo maçãs

Quando a multiplicação e a divisão ocorrem juntas em uma expressão, cada operação é avaliada à medida que ocorre da esquerda para a direita. Quando a adição e subtração ocorrem juntas em uma expressão, cada operação é avaliada em ordem de aparência da esquerda para a direita.

Parênteses podem ser usados para substituir a ordem de precedência e forçar algumas partes de uma expressão a serem avaliadas antes de outras. Operações entre parênteses são sempre executadas antes das externas. No entanto, entre parênteses, a precedência do operador é mantida.

Considere a seguinte expressão:

$$3 + 6 * 2/3$$

A primeira ação executada é $6*2$ (12), em seguida $12/3$ (4), e finalmente $3 + 4$.

Operador	Nome	Exemplo
+	Soma	10 + 10
-	Subtração	10 - 5
*	Multiplicação	2*2
/	Divisão	10/5
Mod	Resto	10 Mod 3
^	Exponencial	2^2

Tabela 1.4: Operadores matemáticos

Nos exemplos anteriores realizamos operações de comparação nas condicionais, por exemplo verificando se a idade é < 15 , ou ainda se o tipo de maçã comprada = "M1". Esses são os operadores de comparação (de valores ou numérico), que comparam duas expressões e retornam um Boolean (verdadeiro ou falso) que representa a relação de seus valores. Os operadores de comparação numéricos e de valores são mostrados na Tabela 1.5:

Operador	Condição testada
=	O valor da primeira expressão é igual ao valor da segunda?
<>	O valor da primeira expressão é diferente do valor da segunda?
<	O valor da primeira expressão é menor que o valor da segunda?
<=	O valor da primeira expressão é menor ou igual ao valor do segundo?
>	O valor da primeira expressão é maior que o valor do segundo?
>=	O valor da primeira expressão é maior ou igual ao valor do segundo?

Tabela 1.5: Operadores de comparação numérica

OBS1: O operador de igualdade (=) é usado tanto para comparações (em condicionais) quanto para realizar atribuições, ou seja, determinada variável recebe um valor, por exemplo $A = 10$.

Considerando as seguintes variáveis,

A = 10
 B = -3
 C = True
 D = False

Avalie cada uma das condições abaixo como verdadeira ou falsa.

1. Se ($A < 2$)
2. Se (C)
3. Se ($A < B * A$)
4. Se (D)
5. Se ($2 > (A - 3)$)

Ainda, existem operadores lógicos, usados para comparar expressões booleanas (verdadeiro ou falso) como um todo. Os operadores são mostrados na Tabela 1.6.

As chamadas Tabelas-verdade mostram os resultados possíveis para todos os operadores lógicos, considerando as duas condições. As tabelas avaliam a seguinte expressão genérica:

$$COND_1 \text{ OPERADOR } COND_2 \quad (1.1)$$

Em que COND1 e COND2 são Booleanos (Verdadeiro ou False), e OPERADOR é o operador aplicado. As Tabelas () para todos os operadores são mostradas abaixo:

EXEMPLO: Considere os mesmo valores de variáveis do exemplo anterior. Avalie as seguintes condições como verdadeiras ou falsas:

Operador	Descrição
And	se ambas as condições forem verdadeiras, a Expressão é verdadeira
Or	Se qualquer uma das condições for verdadeira, resultado verdadeiro
Not	Reverte o estado lógico do operando
Xor	Se apenas uma das expressões for verdadeira, o resultado é Verdadeiro

Tabela 1.6: Operadores lógicos

COND1	COND2	Avaliação
True	True	True
True	False	False
False	True	False
False	False	False

Tabela 1.7: Tabela-verdade And

COND1	COND2	Avaliação
True	True	True
True	False	True
False	True	True
False	False	False

Tabela 1.8: Tabela-verdade Or

COND1	COND2	Avaliação
True	True	False
True	False	True
False	True	True
False	False	False

Tabela 1.9: Tabela-verdade Xor

1. Se $((A > 2) \text{ And } (B < 10)) \rightarrow \text{True}$
2. Se $((A > 2) \text{ And } (B < -10)) \rightarrow \text{False}$
3. Se $((A > 2) \text{ Or } (B < -10)) \rightarrow \text{True}$
4. Se $((A > 2) \text{ Or } (B < 10)) \rightarrow \text{True}$
5. Se $(\text{Not}(A > 2) \text{ And } (B < 10)) \rightarrow \text{False}$
6. Se $((A > 2) \text{ And } (\text{Not } B < 10)) \rightarrow \text{False}$
7. Se $\text{Not}((A < 0) \text{ And } (B > 10)) \rightarrow \text{True}$
8. Se $(\text{Not}(A < 0) \text{ And } \text{Not } (B > 10)) \rightarrow \text{True}$

Estruturas de dados

Para resolver determinados problemas, podemos precisar de mais espaços na memória do computador. Por exemplo, imagine que você precisa armazenar as notas de 50 alunos da turma do curso de informática do curso de Eng. de produção, para realizar algumas operações sobre elas (por exemplo, calcular quem quais notas são maiores do que 7, média, etc...).

Poderíamos criar um espaço na memória para a nota de cada aluno individualmente, com o

seguinte código (Salvando notas dos alunos):

Algoritmo: Salvando notas dos alunos

1. Dim *n1* as Integer
2. Dim *n2* as Integer
3. Dim *n3* as Integer
4. ... (até 50)
5. *n1* = 7
6. *n2* = 9.2
7. *n3* = 2
8. ... (até 50)
9. (CALCULA A MÉDIA DOS VALORES)

Uma esquematização das variáveis e dos endereços na memória ficaria (Figura 1.11):



Figura 1.11: Memória alocada para notas de alunos

OBS: Note que os endereços não estão na mesma ordem em que as variáveis foram declaradas no algoritmo (fisicamente). Não somos nós que escolhemos os endereços, mas sim o sistema operacional. Por isso a memória é chamada RAM (**R**andom **A**ccess **M**emory), ou seja, qualquer endereço pode ser acessado, não necessariamente precisam estar em sequência

Mantido dessa forma, o código pode se tornar obsoleto em muitas situações... imagine que 3 novos alunos são transferidos de outra universidade. Tanto a declaração de variáveis quanto o cálculo da média devem ser alterados no código, e isso deverá ser feito sempre que o número de alunos mudar. Para que isso não ocorra, podemos usar **estruturas de dados**, que armazenam os dados na memória de forma mais eficiente ¹

Duas estruturas de dados muito utilizadas são os *vetores* e as *matrizes*. Os vetores armazenam muitas informações de um mesmo tipo, pode ser pensado como um conjunto de endereços na memória. Podemos declarar um vetor com um número específico de elementos na memória. Ainda, podemos acessar os elementos (para realizar atribuições e consultas) usando o parenteses com o *índice* do elemento. O índice é um número inteiro iniciado em 0, que numera os elementos de um vetor. Considere o pseudocódigo a seguir:

Algoritmo: Salvando notas dos alunos

1. Dim *V(4)* as Integer
2. *V(0)* = 10
3. *V(1)* = 12
4. *V(2)* = -3
5. *V(3)* = 0
6. *V(4)* = 0

O código acima declara um vetor com 5 posições e logo em seguida faz uma atribuição de valor para cada um dos 5 elementos. Note que a declaração para 5 elementos é *V(4)*, pois o 4 indica o último índice do vetor, como os vetores começam no índice 0, *V(4)* tem 5 posições (0,1,2,3 e 4). O vetor declarado ficaria da seguinte forma na memória do computador (Figura 1.12):

No caso dos vetores, o endereço do primeiro elemento (*V(0)*) é encontrado pelo sistema operacional, e todos os outros são localizados de forma contígua.

¹Na realidade, a eficiência de uma estrutura de dados depende muito da aplicação que se deseja fazer, todas possuem vantagens e desvantagens, como se verá mais a frente



Figura 1.12: Memória alocada para um vetor com 5 elementos (os elementos são contíguos na memória)

Como os vetores, podemos definir matrizes de n dimensões. O código a seguir declara uma matriz de 2 linhas e 2 colunas, e atribui 0 a todos os elementos:

Algoritmo: Declarando uma matriz 2x2

1. Dim $V(1,1)$ as Integer
2. $V(0,0) = 0$
3. $V(0,1) = 0$
4. $V(1,0) = 0$
5. $V(1,1) = 0$

A declaração é similar ao vetor, porém determinamos primeiro o índice da última linha, e em seguida o da última coluna. A matriz declarada no código é uma representação de:

$$V(1,1) = \begin{bmatrix} V(0,0) & V(0,1) \\ V(1,0) & V(1,1) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

EXEMPLO: Como podemos escrever um código que realize a soma de dois vetores $A = [1,2,3]$ e $B = [3,4,2]$, atribuindo o resultado em um terceiro vetor C ?

Algoritmo: Soma de vetores

1. Dim $A(2)$ as Integer
2. Dim $B(2)$ as Integer
3. Dim $C(2)$ as Integer
4. $A(0) = 1$
5. $A(1) = 2$
6. $A(2) = 3$
7. $B(0) = 3$
8. $B(1) = 4$
9. $B(2) = 2$
10. $C(0) = A(0) + B(0)$
11. $C(1) = A(1) + B(1)$
12. $C(2) = A(2) + B(2)$

OBS: O que aconteceria se tentássemos acessar o elemento $A(5)$?

A vantagem de se usar vetores e matrizes é poder automatizar a atribuição, verificação e cálculos com seus elementos (como no caso das notas dos alunos), para isso, porém, precisamos de uma forma mais ágil para percorrer os elementos das estruturas de dados, e não realizar as atribuições elemento a elemento como no código acima. Para isso usamos os laços de repetição.

Laços de repetição

Laços de repetição, também conhecidos como laços de iteração ou simplesmente *loops*, são comandos que permitem iteração de código, ou seja, que comandos presentes no bloco sejam repetidos diversas vezes. Através de laços de repetição é possível criar programas que percorram vetores, analisando individualmente cada elemento, e até mesmo criar trechos de código que sejam repetidos até que certa condição estabelecida seja cumprida.

Existem diversos tipos de laços de repetição, um deles é o laço *for* (para). O laço *for* permite que uma variável contadora seja testada e incrementada a cada iteração, sendo essas informações definidas na chamada do comando. Considere o código abaixo:

Algoritmo: Laço for

1. For i = 0 to 8
 - 1.1. OUTPUT(i)
2. Next i
3. OUTPUT("Fim")

A sintaxe do laço *for* está nas linhas 1 e 2, a linha 1 é o início do código que deverá ser repetido e a linha 2 é o fim. Tudo que estiver entre essas linhas (1.1) será iterado, no caso acima, o número *i* será impresso. Mas quantas vezes isso ocorrerá?

Inicialmente o laço *for* declara uma variável e atribui um valor a ela, no caso a variável chamada *i* recebe o valor 0. Em seguida tudo que está dentro do **bloco for** é executado, no caso imprimir o valor de *i*. Quando o código chega no fim do laço *for* na linha 2, a variável é incrementada em uma unidade, ou seja, *i* passa a ter o valor de 1 ($0 + 1$). Em seguida, o código volta até o início do laço (linha 1) e realiza a verificação de uma condicional: o valor atual de *i* é \leq ao valor após a palavra **to**? Se a condição for verdadeira, o trecho do bloco é repetido, se não, o código sai do loop e executa o que estiver depois dele, no caso imprimir a mensagem "Imprima fim".

Podemos agora usar o laço *for* para realizar a atribuição de elementos em um vetor. Considere o código abaixo, que declara um vetor de 100 posições, e atribui o valor de 10 a todos os elementos:

Algoritmo: Atribuição de valores a vetor com laço for

1. Dim V(99) as Integer
2. For i = 0 to 99
 - 2.1. V(i) = 0
3. Next i

A cada iteração do laço *for* o contador *i* aumenta de valor, de forma que acessamos os elementos do vetor na posição de *i*. Assim, a cada iteração um novo endereço do vetor recebe o valor de 0.

EXEMPLO: Considere os códigos abaixo, qual o valor final do vetor V?

Algoritmo: Atribuição de valores a vetor com laço for II

1. Dim V(4) as Integer
2. For i = 0 to 4
 - 2.1. V(i) = i
3. Next i

RESPOSTA

V = [0,1,2,3,4]

Algoritmo: Atribuição de valores a vetor com laço for 2

1. Dim V(4) as Integer
2. For i = 0 to 4
 - 2.1. V(i) = i
3. Next i
4. For i = 0 to 4
 - 4.1. V(i) = V(i)^2
5. Next i

RESPOSTA

No primeiro laço $V = [0,1,2,3,4]$, no segundo $V = [0,1,4,9,16]$

EXEMPLO: Escreva um pseudocódigo que realize a soma dos 100 primeiros inteiros e atribua o resultado em uma variável chamada *resul*.

RESPOSTA

1. Dim *resul* as Integer
2. *resul* = 0
3. For *i* = 0 to 100
 - 3.1. *resul* = *resul* + *i*
4. Next *i*

Tamanho de estruturas de dados

Usualmente o tamanho das estruturas de dados é determinado enquanto o código é executado, de forma que não temos essa informação de antemão. Isso dificulta a definição do tamanho dos loops, de forma que precisamos de uma método para "perguntar" à estrutura de dados (vetor, por exemplo) a posição do seu último elemento. Para isso usamos a expressão

$$UBOUND(arrayname, [dimension]) \quad (1.2)$$

Em que *arrayname* é o nome dado ao vetor e *dimension* especifica sobre qual dimensão queremos saber o último elemento, sendo que este parâmetro é opcional, e quando não fornecido o valor 1 é o padrão. Considere o código abaixo:

Algoritmo: Atribuição de valores a vetor com laço for 2

1. Dim $V(4)$ as Integer
2. For *i* = 0 to $UBOUND(V)$
 - 2.1. $V(i) = i$
3. Next *i*

Note que não especificamos o valor final no laço for, usamos a função $UBOUND(V)$. Como o último elemento de V é 4, a função $UBOUND(V)$ retorna o valor 4. Desta forma, não importaria o tamanho do vetor pois a função é genérica. O código abaixo faz a iteração em uma matriz (que nada mais é do que um vetor de 2 dimensões):

EXEMPLO: Escreva um pseudocódigo que realize a soma dos 100 primeiros inteiros e atribua o resultado em uma variável chamada *resul*.

Algoritmo: Declarando uma matriz 2x2

1. Dim $V(2,1)$ as Integer
2. For *i* = 0 to $UBOUND(V,1)$
 - 2.1. For *j* = 0 to $UBOUND(V,2)$
 - 2.1.1. $V(i,j) = i + j$
 - 2.2. Next *j*
3. Next *i*

No código dimensionamos uma matriz de 3 linhas e cada uma com 2 colunas. O primeiro for da linha 2 usa o $UBOUND$ da primeira dimensão, ou seja, a última posição em relação ao número de linhas, neste caso o retorno é 2. Já o for interno, da linha 2.1, usa o $UBOUND$ em relação à segunda dimensão da matriz (colunas), de forma que a função retorna o valor de 1.

1.3 Entrada e saída de dados

Lembre-se de que um algoritmo funciona no formato input/output, ou seja, deve existir alguma forma de fornecer as entradas para o computador, e este de mostrar as saídas. Isso pode ser feito de diversas formas, por exemplo:

- Arquivos de texto
- Bancos de dados
- Interface com o usuário

Em nossos pseudocódigos utilizaremos uma abstração para o input de dados com a notação

$$variavel = INPUT("Mensagem")$$

Sendo *variavel*, a variável declarada que recebera o input. A função INPUT() pode ler dados de um arquivo, de um banco de dados, ou mesmo abrir uma caixa de input para o usuário. Se for um input para o usuário, uma mensagem pode ser mostrada para o mesmo (porém não é obrigatório). O que o usuário digitar, ou estiver nos arquivos, bancos de dados, etc...será atribuído à variável. Alguns exemplos são mostrados abaixo:

Algoritmo: Soma dois números fornecidos pelo usuário

1. Reserve **N1,N2**, **soma** do tipo inteiro
2. **N1** = INPUT("Digite o primeiro número")
3. **N2** = INPUT("Digite o segundo número")
4. **soma** = **A** + **B**

Algoritmo: Soma os elementos de um vetor fornecido pelo usuário

1. Dim **V1(4)** as Integer
2. Dim **soma** as Integer
3. **soma** = 0
4. **V1** = INPUT("Digite os 5 valores do vetor")
5. For i = 0 to 4
 - 5.1. **soma** = **soma** + **V(i)**
6. Next i

Já para o output usaremos a abstração

$$OUTPUT("Mensagem") \tag{1.3}$$

Sendo que o conteúdo entre parênteses é mostrado ao usuário. Podemos reescrever os pseudocódigos acima imprimindo os valores calculados para o usuário:

Algoritmo: Soma dois números fornecidos pelo usuário

1. Dim **N1** as Integer, **N2** as Integer, **soma** as Integer
2. **N1** = INPUT("Digite o primeiro número")
3. **N2** = INPUT("Digite o segundo número")
4. **soma** = **A** + **B**
5. OUTPUT("O resultado da soma é : ")
6. OUTPUT(**soma**)

Primeiramente a mensagem "O resultado da soma é : " seria mostrado e em seguida o valor associado à variável **soma**.

1.4 Teste de mesa

O teste de mesa é uma técnica manual, que serve para simularmos a execução de um código de computador e verificarmos se ele está correto. Alguns códigos podem ficar muito complexos, de forma que analisá-los somente na linguagem de programação pode ser confuso. Em um teste de mesa vamos seguindo a ordem de execução do código escrito, e anotando os valores das variáveis a medida em que são "executadas". Considere o código abaixo que aloca valores a um vetor de 5 elementos.

Algoritmo: Atribuição de valores a vetor com laço for II

1. Dim V(4) as Integer
2. For i = 0 to 4
 - 2.1. V(i) = i
3. Next i

Podemos criar um teste de mesa acompanhando o valor da variável i e os valores alocados ao vetor. O teste ficaria então (considere que no momento em que o vetor é declarado todos os seus elementos têm o valor 0 por padrão):

i	V[]
0	[0,0,0,0,0]
1	[0,1,0,0,0]
2	[0,1,2,0,0]
3	[0,1,2,3,0]
4	[0,1,2,3,4]

1.5 Quadro resumo

Considere a Figura 1.13 com o resumo dos conceitos do capítulo.

1.6 Exercícios

1.6.1 Variáveis

Considere as declarações de variáveis abaixo, quais estão corretas e quais estão erradas? Explique o porque das erradas

1. Dim A as Integer
2. Dim 1x as Double
3. Dim A, B, C as String
4. Dim A as Integer, Dim B as String
5. _N1 as Integer

1.6.2 Condicionais

1. Para cada uma das condicionais abaixo, verifique se a avaliação da sentença é verdadeira ou falsa, considerando:

- A = 10
- B = 5
- C = True
- D = False

1. EXEMPLO: Se $(A + B) > 15 \rightarrow \text{False}$
2. Se $(A + B) \geq 15$
3. Se $(A \geq 15) \text{ And } (B \leq 15)$
4. Se $(A \geq 15) \text{ Or } (B \leq 15)$
5. Se $\text{Not}((A \geq 15) \text{ Or } (B \leq 15))$
6. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } C$
7. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } (\text{Not } C)$
8. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } (A = \text{True})$
9. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } (\text{Not } A)$
10. Se $(A \text{ Mod } 2) <> 0$

RESPOSTA

1. EXEMPLO: Se $(A + B) > 15 \rightarrow \text{False}$
2. Se $(A + B) \geq 15 \rightarrow \text{True}$
3. Se $(A \geq 15) \text{ And } (B \leq 15) \rightarrow \text{False}$
4. Se $(A \geq 15) \text{ Or } (B \leq 15) \rightarrow \text{True}$
5. Se $\text{Not}((A \geq 15) \text{ Or } (B \leq 15)) \rightarrow \text{False}$
6. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } C \rightarrow \text{True}$
7. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } (\text{Not } C) \rightarrow \text{False}$
8. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } (A = \text{True}) \rightarrow \text{True}$
9. Se $((A \geq 15) \text{ Or } (B \leq 15)) \text{ And } (\text{Not } A) \rightarrow \text{False}$
10. Se $(A \text{ Mod } 2) <> 0 \rightarrow \text{False}$

1.6.3 Pseudocódigos/Fluxogramas, input/output

1. Para cada um dos algoritmos abaixo, determine o valor que é mostrado ao usuário (OUTPUT):

Algoritmo 1

1. Dim **A** as Double, **B** as Double, **C** as Double, **D** as Double, **X** as Double
2. $A = 30$
3. $B = 5$
4. $C = 3$
5. $D = -1$
6. Se $(A = 2) \text{ And } (B < 7)$ faça:
 - 6.1. $X = (A + B) * (B - 2)$
7. Senão faça:
 - 7.1. $X = (A + B) / D * (C + D)$
8. OUTPUT(X)

Algoritmo 2

1. Dim **A** as Double, **B** as Double, **X** as Double
2. $A = 30$
3. $B = 5$
4. Se $(A = 2) \text{ Or } (B < 7)$ faça:
 - 4.1. $X = B + (B \text{ Mod } 2)$
5. Senão faça:
 - 5.1. $X = (A / 2) \text{ Mod } 2$
6. OUTPUT(X)

Algoritmo 3

1. Dim **A** as Double, **B** as Double, **X** as Double
2. Dim **C,D** as Boolean
3. **A** = 30
4. **B** = 5
5. **C** = True
6. **D** = False
7. Se (**A***3 + **B***5) = 2) Or (**B** < 7) faça:
 - 7.1. Se (Not **D**) And (**C**) faça:
 - 7.1.1. **X** = **A** / ((**A** + **B**)/5)
 - 7.2. Senão faça:
 - 7.2.1 **X** = **X** + 3
8. Senão faça:
 - 8.1. Se (**D**) Or (Not **D**) faça:
 - 8.1.1 **X** = (**A** + 3) / ((**A** + **B**)/5)
 - 8.2 Senão faça:
 - 8.2.1 **X** = 4
9. OUTPUT(**X**)

RESPOSTAS

Algoritmo 1: -17.5

Algoritmo 2: 6

Algoritmo 3: 4.28

2. Represente cada um dos 3 algoritmos acima como um fluxograma.
3. Para cada um dos problemas abaixo, represente a solução como um fluxograma e escreva um pseudocódigo do algoritmo:
 - (a) Leia as dimensões de um retângulo (base e altura), calcule e escreva a área do retângulo.
 - (b) Escreva um algoritmo que armazene o valor 10 em uma variável **A** e o valor 20 em uma variável **B**. A seguir (utilizando apenas atribuições entre variáveis) troque os seus conteúdos fazendo com que o valor que está em **A** passe para **B** e vice-versa. Ao final, escrever os valores que ficaram armazenados nas variáveis. (OBS: É necessário uma terceira variável)
 - (c) Leia 3 valores numéricos fornecidos pelo usuário e informe qual foi o maior valor digitado.
 - (d) Leia 3 dimensões fornecidas pelo usuário, representando os lados de um triângulo, e informe se ele é retângulo ou não.
 - (e) Leia um número fornecido pelo usuário e informe se é par ou impar.
 - (f) Uma equação de grau 2 é dada por:

$$ax^2 + b + c \quad (1.4)$$

Em que a, b e c são constantes quaisquer. *Encontrar as raízes* de uma equação significa encontrar os valores de x que, quando substituídas na equação resultam em 0. As equações de grau 2 podem possuir 2 raízes diferentes, 2 raízes iguais, ou ainda não possuir raízes. Quem determina isso é o delta (Δ):

$\Delta > 0$: 2 raízes diferentes.

$\Delta = 0$: 2 raízes iguais.

$\Delta < 0$: sem raízes.

Com

$$\Delta = b^2 - 4ac \quad (1.5)$$

Caso existam raízes, as mesmas são dadas por:

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad (1.6)$$

$$x_2 = \frac{-b - \sqrt{\Delta}}{2a} \quad (1.7)$$

Escreva um algoritmo que leia os valores a, b e c de uma equação de grau 2 do usuário. Em seguida informe se existe uma, duas ou nenhuma raiz, e caso existam, imprima seus valores (Considere no pseudocódigo que a raiz quadrada de um número n é dada por $\text{Sqrt}(n)$, por exemplo, o código abaixo calcula a raiz quadrada de 16 e armazena na variável X).

Algoritmo Raiz

1. Dim X as Double
2. X = Sqrt(16)

- (g) O departamento de compras de uma indústria precisa comprar chapas de aço. Independentemente da quantidade comprada, existe um custo fixo de pedido de R\$200,00. Os preços das chapas variam de acordo com as quantidades compradas, conforme a Tabela 1.10.

Quantidade (ton)	Preço (R\$)
$0 < T \leq 2$	1800
$2 < T \leq 6$	1500
$T > 6$	1200

Tabela 1.10: Preço da ton. de chapa

Escreva um algoritmo que leia a quantidade de aço comprada pelo usuário e informe o custo da compra. OBS: Os preços se mantêm para as suas respectivas faixas, ou seja, se um usuário comprar 5 toneladas o preço do aço não é $5 * \text{R}\$1500$, mas sim $2 * \text{R}\$1800 + 3 * \text{R}\1500 , pois as primeiras duas toneladas custaram R\$1800 e as outras 3 R\$1500.

1.6.4 Laços de repetição e Estruturas de dados

1. Considere os pseudocódigos abaixo, existe algum erro no algoritmo? Se sim, explique o que.

Algoritmo 1

1. Dim X as Double
2. X = Sqrt(16)

Algoritmo 2

1. Dim X(2) as Double
2. X(0) = 3
2. X(1) = 3
2. X(2) = 4
2. X(3) = 100

Algoritmo 2

1. Dim X(10) as Integer
2. For i = 0 to 11
 - 2.1. X(i) = i + 1
3. Next i

Algoritmo 3

1. Dim X(10) as Integer
2. For i = 0 to UBOUND(X) + 1
 - 2.1. X(i) = i + 1
3. Next i

Algoritmo 4

1. Dim X(10) as Integer
2. For i = 0 to UBOUND(X)
 - 2.1. X(i + 1) = 3
3. Next i

2. Para cada pseudocódigo abaixo, determine, por meio de um teste de mesa, quais os valores finais do vetor x:

Algoritmo 1

1. Dim X(6) as Integer
2. For i = 0 to UBOUND(X)
 - 2.1. X(i) = i
3. Next i

Algoritmo 2

1. Dim X(6) as Boolean
2. For i = 0 to UBOUND(X)
 - 2.1. Se X(i) Mod 2 = 0
 - 2.1.1. X(i) = True
 - 2.2 Senão
 - 2.2.1. X(i) = False
3. Next i

Algoritmo 3

1. Dim X(6) as Integer
2. X(0) = 1
3. For i = 1 to UBOUND(X)
 - 3.1. X(i) = X(i-1) + 2
4. Next i

Algoritmo 4

1. Dim X(6), cont as Integer
2. cont = UBOUND(X)
3. For i = 0 to UBOUND(X)
 - 3.1. X(cont) = i
 - 3.2. cont = cont - 1
4. Next i

Algoritmo 5

1. Dim A(3) as Integer, B(3) as Integer, X(4) as Integer
2. For i = 0 to UBOUND(A)
 - 2.1. Se i Mod 2 = 0
 - 3.1.1 A(i) = i
 - 3.1.2 B(i) = 0
 - 3.2. Senão
 - 3.2.1 A(i) = 0
 - 3.2.2 B(i) = i
4. Next i
3. For j = 0 to UBOUND(A)
 - 3.1. X(i) = A(i) + B(i)
4. Next i
5. X(UBOUND(X)) = 90

3. Escreva um algoritmo que realize a soma dos 100 primeiros números inteiros.
4. Escreva um algoritmo que realize a soma de todos os números pares entre 0 e 100.
5. Escreva um algoritmo que realize a soma de todos os números ímpares entre 0 e 100.
6. Escreva um algoritmo que armazene a seguinte matriz:

$$M = \begin{bmatrix} 2 & 3 & 6 \\ 4 & 5 & 8 \end{bmatrix}$$

E realize a soma dos seus elementos (use um laço **for** e a função **UBOUND()**)

Tipos de dados

Dados	Espaço de armazenamento	Explicação
Double	8 bytes	Números reais
Integer	2 bytes	Números inteiros
Date	8 bytes	Datas
Integer	2 byte	Números inteiros
Boolean	2 byte	Verdadeiro ou Falso (True/False)
String	10 bytes + tamanho da cadeia de caracteres	Palavras

Operadores matemáticos

Operador	Nome	Exemplo
+	Soma	10 + 10
-	Subtração	10 - 5
*	Multiplicação	2*2
/	Divisão	10/5
Mod	Resto	10 Mod 3
^	Exponencial	2^2

Operadores de comparação numérica

Operador	Condição testada
=	O valor da primeira expressão é igual ao valor da segunda?
<>	O valor da primeira expressão é diferente do valor da segunda?
<	O valor da primeira expressão é menor que o valor da segunda?
<=	O valor da primeira expressão é menor ou igual ao valor do segundo?
>	O valor da primeira expressão é maior que o valor do segundo?
>=	O valor da primeira expressão é maior ou igual ao valor do segundo?

Operadores lógicos

Operador	Descrição
And	se ambas as condições forem verdadeiras, a Expressão é verdadeira
Or	Se qualquer uma das condições for verdadeira, resultado verdadeiro
Not	Reverte o estado lógico do operando
Xor	Se apenas uma das expressões for verdadeira, o resultado é Verdadeiro

Sintaxe pseudocódigos	
Definição de variável	Dim A as Integer
Atribuição de valor a variável	A = 10
Definição de vetor com 4 elementos	Dim V(3) as Integer
Atribuição de valor a elemento do vetor	V(0) = 10
Condicional 1	Se A > 10 faça A = A + 1
Condicional 2	Se A > 10 faça A = A + 1 Senão faça: A = A + 3
Condicional 3	Se A > 10 faça A = A + 1 Senão, se A < 5 faça: A = A + 3 Senão faça: A = A + 3
Laço for	For i = 0 to 10 A = A + i Next i

Figura 1.13: Resumo do capítulo

Capítulo 2

VBA Inicial

O Visual Basic for Applications (VBA) é uma implementação do Visual Basic da Microsoft incorporada em todos os programas do Microsoft Office - dentro desse pacote estão os famosos aplicativos Office (Word, Excel, Power Point, Access, etc), bem como em outras aplicações da Microsoft, como o Visio, e que foi também incorporada pelo menos parcialmente em outros programas de terceiros como o AutoCAD, Mathcad e WordPerfect. Ele substitui e estende as capacidades de anteriormente existentes linguagens de programação de macros específicas para as aplicações e pode ser usado para controlar a quase totalidade dos aspectos da aplicação anfitriã, incluindo a manipulação de aspectos do interface do usuário tais como menus e barra das ferramentas e o trabalho com formulários desenhados pelo usuário ou com caixas de diálogo ("Wikipedia")

2.1 Configuração

Para configurar o Excel para a programação em VBA significa habilitar as "macros".

- **Excel 2007** Botão do Microsoft Office >> Opções do Excel >> Central de Confiabilidade >> Configurações da Central de Confiabilidade >> Configurações de Macro >> Habilitar todas as Macros.
- **Excel 2013** Botão do Microsoft Office >> Opções do Excel >> Central de Confiabilidade >> Configurações da Central de Confiabilidade >> Configurações de Macro >> Habilitar todas as Macros.

Ainda, para deixarmos disponível de forma rápida a aba de programação, chamada "Desenvolvedor":

- Arquivo >> Opções do Excel >> Personalizar >> Em personalizar faixa de opções >> Guias principais >> marque a caixa de seleção Desenvolvedor.

Para iniciar um novo programa:

- Aba desenvolvedor >> Visual Basic.

A seguinte janela deve aparecer, como na Figura 2.1.

Para iniciar um programa inserimos os chamados "Módulos":

- Clicar com o botão direito em VBA Project (canto esquerdo)

Com o módulo, o programa fica como na imagem

A área central (em branco) é o local onde os códigos serão colocados.

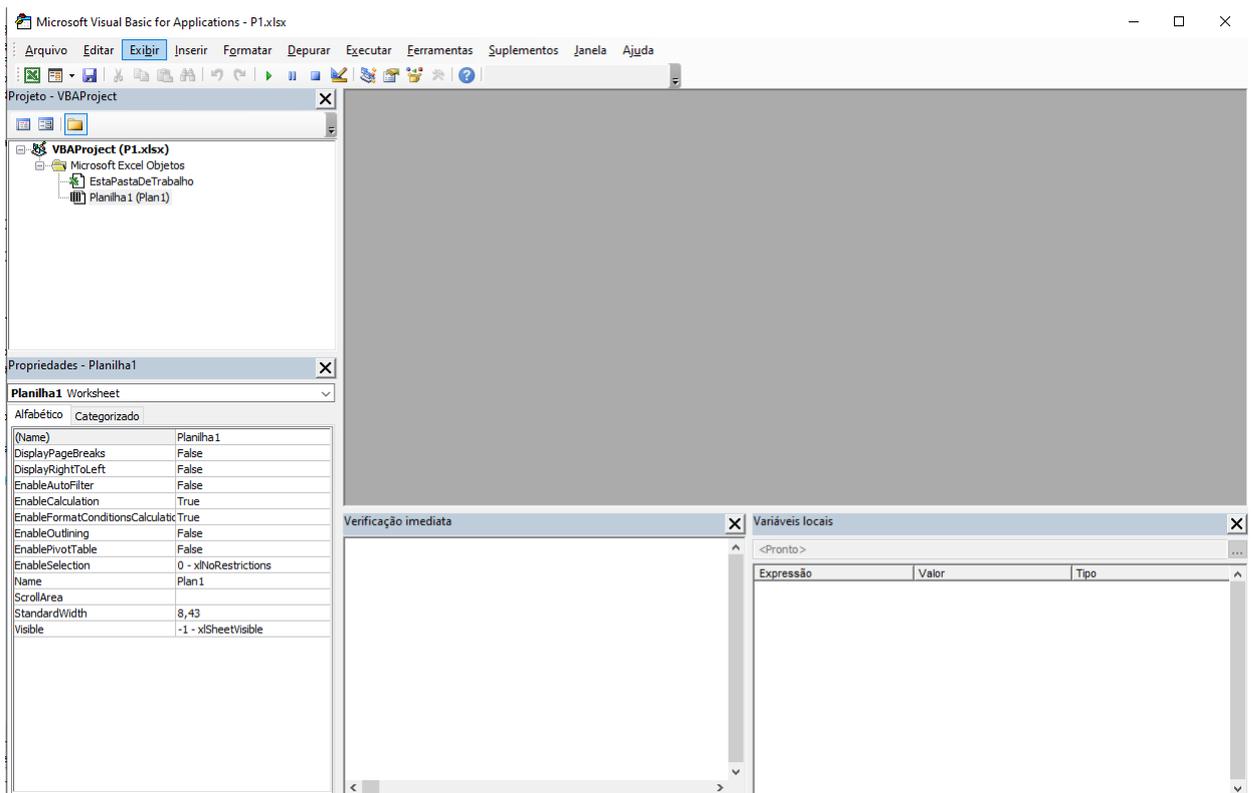


Figura 2.1: Área inicial VBA

2.2 O primeiro código

Todo o código que criado deve estar em um "bloco" de código, com início e fim. Com o bloco criado podemos selecioná-lo e executá-lo. Por enquanto chamaremos esses blocos de **Sub's** (como subrotinas), o código abaixo exemplifica a sintaxe para a criação de uma Sub:

```
Sub nome_sub ()
End Sub
```

Em que "nome_sub" é o nome dado a Subrotina que será executada. Para executar a subrotina podemos apertar F5 ou clicar sobre o ícone de "rodar" (ícone verde na parte superior). Como Sub do código acima não têm nada implementado, nada será executado. Considere o código abaixo e o execute para ver o que acontece!

```
Sub primeira_sub ()
    MsgBox ("Ola mundo!")
End Sub
```

Ao criarmos mais Subs o excel pode não saber qual ele deve executar, nesses casos nós devemos informar. Uma tela é aberta com todas as subs do módulo e nós escolhemos quais devem ser executadas.

```
Sub primeira_sub ()
    MsgBox ("Ola mundo!")
End Sub

Sub segunda_sub ()
    MsgBox ("Ola mundo 2!")
End Sub
```

OBSERVAÇÃO IMPORTANTE! Ao salvar a planilha, a mesma deve ser salva como pasta habilitada para macros do excel! Caso contrário tudo programado no módulo será perdido.

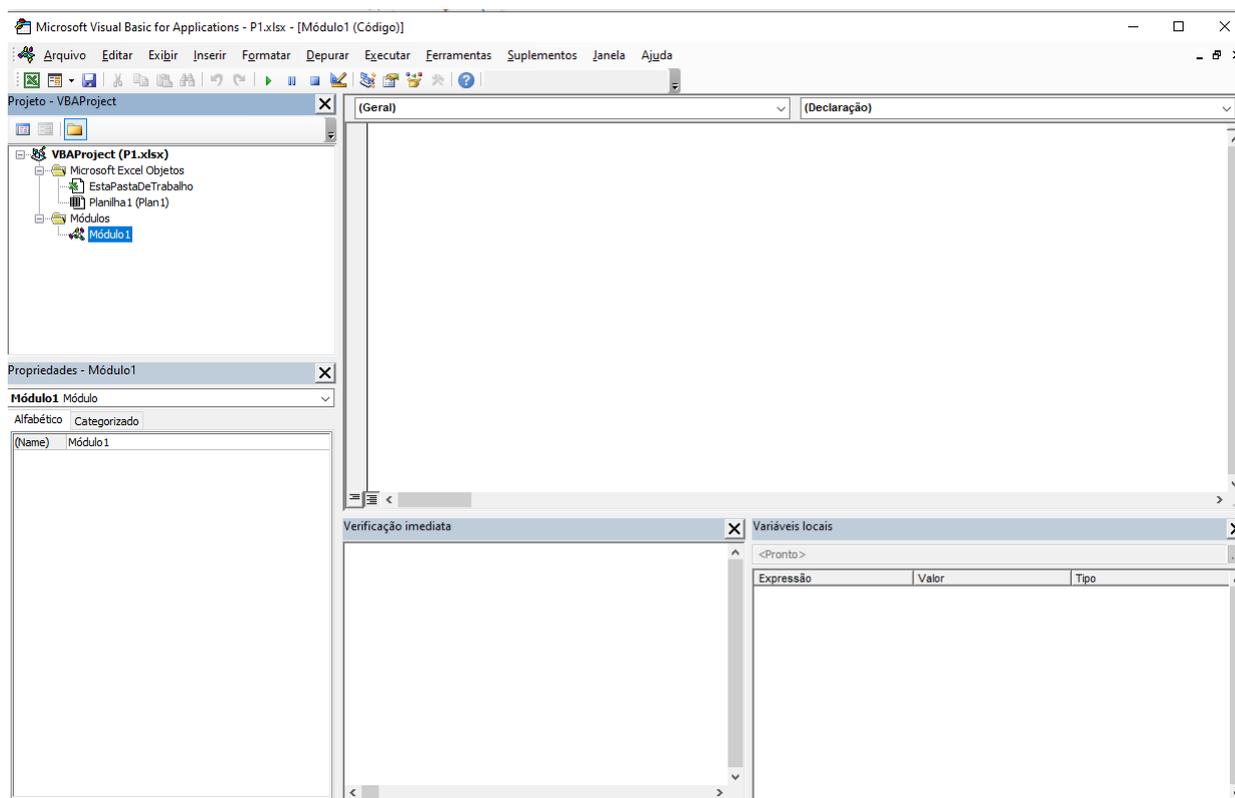


Figura 2.2: Área inicial VBA com módulo

2.3 Declaração de variáveis, comentários e verificação imediata

A declaração de variáveis no VBA segue a mesma sintaxe dos pseudocódigos utilizados no capítulo 1. O código abaixo exemplifica a criação de uma variável do tipo inteiro, real, booleana e string, bem como a atribuição de valores às mesmas:

```
Sub declara_variaveis ()
    Dim a As Integer
    Dim b As Double
    Dim c As Boolean
    Dim d As String

    a = 10
    b = 10.5
    c = True
    d = "Ola"
End Sub
```

Note que a string é declarada entre aspas. Podemos declarar diversas variáveis em uma mesma linha, separando-as por vírgula. Considere o código abaixo em que a, b e c são do tipo inteiro, e todas declaradas em uma única linha:

```
Sub declara_variaveis ()
    Dim a As Integer, b As Integer, c As Integer
End Sub
```

Em VBA também podemos utilizar variáveis sem declará-las explicitamente (infelizmente). Considere o código abaixo:

```
Sub nao_declarada ()
    a = 10 / 3
End Sub
```

Para a maioria dos efeitos, essa não é uma boa prática, de forma que podemos forçar o excel a não aceitar variáveis que não estejam declaradas, basta inserirmos a opção *Option Explicit* na primeira linha do código:

```
Option explicit
Sub nao_declarada()
    a = 10 / 3
End Sub
```

Podemos inserir *comentários* nos códigos. Os comentários não são executados como códigos, de forma que podemos escrever anotações pessoais explicando alguns trechos, deixar indicações do que deve ser feito mais tarde, ou ainda comentar uma parte de código que não queremos apagar ainda. A sintaxe para os comentários em VBA é marcado pelo '. Tudo escrito a frente disso não será executado. O código abaixo exemplifica:

```
Sub comentario()
    Dim a As Integer, b As Integer

    ' Essa parte do codigo e MUITO importante!
    a = 10

    ' Essa nem tanto...
    Dim st As String
End Sub
```

Os tipos de variáveis são mostrados na Tabela 1.2.

Uma parte muito importante de qualquer ambiente de programação (independentemente da linguagem), é a possibilidade de "ver" o que está acontecendo com o código. Quando realizamos o design de algoritmos, como mostrado no capítulo anterior, lançamos mão dos testes de mesa para verificar se o mesmo está fazendo o que deveria. Podemos usar "outputs" nos nossos programas para mostrar ao usuário o que está acontecendo, e de certa forma implementar um teste de mesa. Nesta seção veremos somente o recurso da *janela de verificação imediata*, porém existem muitos outros que serão vistos nos capítulos vindouros.

A janela de verificação imediata pode ser acessada por:
Exibir >> Janela de verificação imediata

Ela serve para que possamos "exportar" informações do nosso código. Para isso, usamos o comando **Debug.Print**, e tudo que estiver após o comando será exportado para a janela de verificação. Considere o código abaixo:

```
Sub imprime_mensagem()
    Dim a As String
    a = "Esta mensagem sera exibida na janela de verificacao"
    Debug.Print a
End Sub
```

A janela pode ser usada para imprimir mais um valor ao mesmo tempo, basta separar os itens por ponto-vírgula ou vírgulas:

```
Sub imprime_mensagem()
    Dim a As String, b As String
    a = "Esta mensagem sera exibida "
    b = "na janela de verificacao"
    Debug.Print a, b
End Sub
```

O código abaixo realiza a soma de dois valores e imprime o resultado na janela de verificação.

```
Sub imprime_mensagem()
    Dim a As Double, b As Double, c As Double
    a = 10.111
    b = 20.256
    c = a + b
    Debug.Print c
End Sub
```

2.4 Operadores matemáticos e de comparação numérica

Também os operadores matemáticos e de comparação numérica em VBA possuem a mesma sintaxe do pseudocódigo do capítulo 1. As Tabelas 2.1 e 2.2 exemplificam:

Operador	Nome	Exemplo
+	Soma	10 + 10
-	Subtração	10 - 5
*	Multiplicação	2*2
/	Divisão	10/5
Mod	Resto	10 Mod 3
^	Exponencial	2^2

Tabela 2.1: Operadores matemáticos

As operações matemáticas têm a seguinte ordem de precedência:

1. Multiplicações, Divisões e Exponenciais
2. Módulo (operador para encontrar o resto da divisão)
3. Somas e Subtrações

E as operações são executadas na ordem \rightarrow .

Quando a multiplicação e a divisão ocorrem juntas em uma expressão, cada operação é avaliada à medida que ocorre da esquerda para a direita. Quando a adição e subtração ocorrem juntas em uma expressão, cada operação é avaliada em ordem de aparência da esquerda para a direita.

Parênteses podem ser usados para substituir a ordem de precedência e forçar algumas partes de uma expressão a serem avaliadas antes de outras. Operações entre parênteses são sempre executadas antes das externas. No entanto, entre parênteses, a precedência do operador é mantida.

Operador	Condição testada
=	O valor da primeira expressão é igual ao valor da segunda?
<>	O valor da primeira expressão é diferente do valor da segunda?
<	O valor da primeira expressão é menor que o valor da segunda?
<=	O valor da primeira expressão é menor ou igual ao valor do segundo?
>	O valor da primeira expressão é maior que o valor do segundo?
>=	O valor da primeira expressão é maior ou igual ao valor do segundo?

Tabela 2.2: Operadores de comparação numérica

EXEMPLO: Avalie os pares de expressões a seguir e diga se elas produzem o mesmo resultado, em seguida confirme se você está correto calculando pelo VBA e imprimindo os resultados na janela de verificação:

1. $A = (4/2)+(2/4)$ e $A = 4/2+2/4$
2. $A = 4/(2+2)/4$ e $A = 4/2+2/4$
3. $A = (4+2)*2-4$ e $A = 4+2*2-4$

EXEMPLO: Reescreva as expressões abaixo com a menor quantidade de parênteses possível (quando isso for factível), **mas sem alterar o resultado**. Da mesma forma, valide as suas respostas no VBA imprimindo os valores das expressões, com e sem parênteses:

1. $A = 6*(3+2)$
2. $A = 2+(6*(3+2))$
3. $A = 2+(3*6)/(2+4)$

2.5 Condicionais e operadores lógicos

Em VBA os condicionais simples (uma condição) seguem a seguinte sintaxe:

```
Sub condicional0()
  Dim A As Integer, B As Integer
  A = 10
  B = 30

  If A < B Then
    ' BLOCO EXECUTADO SE A CONDICAO FOR VERDADEIRA
  End If
End Sub
```

Logo após o **If** colocamos a condição a ser testada, se a mesma for avaliada como verdadeira o bloco após o **Then** e antes do **End if** é executado. Caso contrário o bloco não é executado e o código segue para as próximas instruções abaixo do **End if**.

Podemos designar uma condição alternativa nas condicionais, ou seja, o que fazer caso a condição não seja verdadeira. Isso é feito com a instrução **Else**.

```
Sub condicional1()
  Dim A As Integer, B As Integer
  A = 10
  B = 30

  If A < B Then
    ' BLOCO EXECUTADO SE A CONDICAO FOR VERDADEIRA
  Else
    ' BLOCO EXECUTADO SE A CONDICAO NAO FOR VERDADEIRA
  End If
End Sub
```

Dessa forma, o **Else** é equivalente ao **Senão**. A instrução pode ser lida como "Execute essa instrução **SE** a condição for verdadeira, **SENÃO**, execute esta outra instrução".

Podemos ainda especificar diversas condições em um único **If**, usando a sintaxe **ElseIf**.

```
Sub condicional2()
  Dim A As Integer, B As Integer
  A = 10
  B = 30

  If A < B Then
    ' BLOCO EXECUTADO SE A CONDICAO FOR VERDADEIRA
  ElseIf A > B Then
    ' BLOCO EXECUTADO SE A PRIMEIRA CONDICAO FOR FALSE MAS A SEGUNDA VERDADEIRA
  End If
End Sub
```

O **ElseIf** é equivalente ao **Senão se**.

ATENÇÃO: Em um bloco **if** de mesmo nível hierárquico no máximo **UMA** condição é executada, não importa se **If**, **ElseIf** ou **Else**, se qualquer uma delas for verdadeira, após o bloco ser executado o programa passa para a linha após o **End if**. Ou seja, mesmo que mais de uma condição seja verdadeira, a primeira avaliada é a que será executada.

EXEMPLO: Considere o código abaixo. O que será impresso na janela de verificação?

```
Sub condicional3()
  Dim A As Integer, B As Integer
  A = 10
  B = 30

  If A < 100 Then
    Debug.Print "A < 100"
  ElseIf B < 100 Then
    Debug.Print "B < 100"
  End If
End Sub
```

Note que mesmo que a segunda condição seja verdadeira ($B < 100$) ela não é executada, pois o bloco da primeira condição já foi executado. Sempre lembrar de que, em um mesmo nível hierárquico, somente uma condição é executada.

Operador	Descrição
And	se ambas as condições forem verdadeiras, a Expressão é verdadeira
Or	Se qualquer uma das condições for verdadeira, resultado verdadeiro
Not	Reverte o estado lógico do operando
Xor	Se apenas uma das expressões for verdadeira, o resultado é Verdadeiro

Tabela 2.3: Operadores lógicos

Diversas condições podem ser concatenadas usando os operadores lógicos, como no pseudocódigo. A Tabela 2.3 mostra os operadores que podem ser utilizados:

EXEMPLO: Considere o código abaixo. O que será impresso na janela de verificação?

```
Sub condicional4()
    Dim A As Integer, B As Integer, C As Boolean, D As Boolean
    A = 10
    B = 30
    C = True
    D = False

    If (A < 100) And (D = True) Then
        If (A = 10) Then
            Debug.Print "Caminho 1"
        Else
            Debug.Print "Caminho 2"
        End If
    ElseIf (B > 100) Or (C = D) Then
        If (B = 30) Or (C = False) Then
            Debug.Print "Caminho 3"
        Else
            Debug.Print "Caminho 4"
        End If
    Else
        Debug.Print "Caminho 5"
    End If
End Sub
```

EXEMPLO: Implemente os algoritmos do capítulo anterior "Idade mínima para beber" (1.2.3) e "Custo maçãs" (1.2.3). Teste se eles estão funcionando com as idades: 10, 15, 18 e 50, e com os dois tipos de maçãs, com quantidades 2 e 10. O que acontece se o usuário escolher uma maçã que não existe (M3 por exemplo)? Se possível, melhore o código para se adequar a esta situação.

2.6 Laços de repetição

Existem diversos laços de repetição no VBA (como em todas as linguagens de programação). Alguns deles são mostrados a seguir.

2.6.1 Laço For

O laço **for** funciona como explicado no capítulo anterior. O laço **for** permite que uma variável contadora seja testada e incrementada a cada iteração, sendo essas informações definidas na chamada do comando. Considere o código a seguir:

```
Sub main()
    'Laco for que imprime os inteiros de 0 a 10
    For i = 0 To 10
        Debug.Print i
    Next i
End Sub
```

Tudo entre **For** e **Next** é considerado o bloco do laço, e será repetido de acordo com o contador, no caso acima **i**. Após executar o bloco o programa incrementa o contador **i** e volta para a linha do

For, onde a condição é testada: $i \leq 10$? Se sim, o bloco é executado novamente, se não, o bloco é ignorado e o código continua da linha após o **Next**.

Podemos ainda sair de um bloco **For** antes do seu término, com a opção **Exit For**. O código abaixo executa o bloco até que i seja igual a 8, e em seguida sai do bloco.

```
Sub main()
    For i = 0 To 10
        If i = 8 Then
            Exit For
        Else
            Debug.Print i
        End If
    Next i
End Sub
```

Até agora o contador foi incrementado de forma unitária, ou seja, a cada iteração uma unidade é aumentada de i . Podemos alterar esse incremento, ou seja, o "passo" do laço **For** com a opção **Step**. O código abaixo soma todos os números pares de 0 a 100, para isso o passo é de 2 unidades, para que todo i seja par.

```
Sub main()
    Dim soma_pares As Integer
    For i = 0 To 100 Step 2
        soma = soma + i
        Debug.Print i
    Next i
    Debug.Print soma
End Sub
```

Se estivermos com alguma estrutura de dados que possua um iterador, podemos usar o laço **For each** para acessar os elementos da estrutura, sem a necessidade de um contador. O código abaixo declara um vetor (serão vistos mais a frente), atribui alguns valores à seus elementos, imprimindo os valores em um laço **For each**.

```
Sub main()
    Dim v(3) As Integer
    v(0) = 3
    v(1) = 2
    v(2) = 6
    v(3) = 8

    For Each e In v
        Debug.Print e
    Next e
End Sub
```

Nesse caso **e** já representa os elementos do vetor, e não os índices

2.6.2 Laço While

O laço while (enquanto) executa um bloco de comandos até o momento em que uma determinada condição se torne false. O código abaixo exemplifica a sua sintaxe.

```
Sub main()
    Dim a As Integer
    a = 0

    Do While a < 20
        a = a + 1
        Debug.Print a
    Loop
End Sub
```

Nó código acima, o bloco é determinado após a linha **Do While** até **Loop**. Esse bloco será repetido enquanto a condição após a palavra **While** for satisfeita, neste caso $a < 20$. A cada iteração incrementamos o valor de **a** em uma unidade, de forma que após 20 iterações **a** valerá 21, fazendo a condição ser falsa e portanto parando de executar o laço.

2.7 Input/Output

Como vimos, algoritmos transformam um *input* em um *output*, por meio de operações bem definidas. Dessa forma, é muito importante conhecer métodos para fazer o input de informações no programa, e também como receber os outputs. Nesta seção veremos alguns deles, além da janela de visualização rápida, já apresentado nas seções anteriores.

2.7.1 Input: Inputbox

InputBox() é uma função de *input* para interação com o usuário por meio de uma interface gráfica, na própria planilha do excel. *A função exibe um prompt em uma caixa de diálogo, aguarda o usuário inserir texto ou clicar em um botão e retorna uma cadeia de caracteres com o conteúdo da caixa de texto.* Ou seja, uma caixa de diálogo é aberta, e o usuário pode digitar algum valor que pode ser usado no programa. A caixa pode ser personalizada de diversas formas, para mais detalhes ver a <https://docs.microsoft.com/pt-br/office/vba/language/reference/user-interface-help/inputbox-function>.

Em sua forma mais simples, considere o programa abaixo:

```
Sub caixa_de_entrada()  
    Dim a As Integer  
    a = InputBox("Digite um numero")  
    Debug.Print "O numero digitado foi : " a  
End Sub
```

O código acima simplesmente declara uma variável inteira (a), e em seguida atribui um valor a ela pelo InputBox e imprime na janela de verificação, ou seja, o que o usuário digitar na interface será atribuído a variável. Sempre devemos atribuir o que o usuário digita em uma variável, caso contrário não há sentido em usar InputBox. O código abaixo funciona, porém não conseguimos usar o que o usuário digitou dentro do nosso programa.

```
Sub caixa_de_entrada()  
    InputBox("Digite um numero")  
End Sub
```

OBSERVAÇÃO: O que acontece quando declaramos uma variável inteira para receber o input do usuário, mas o mesmo digita uma string? Como podemos corrigir esse comportamento? (DICA: Conversão de variáveis).

EXEMPLO Escreva um algoritmo que leia 3 valores numéricos fornecidos pelo usuário utilizando a função InputBox, e informe qual foi o maior valor digitado.

EXEMPLO Escreva um algoritmo que leia 3 valores numéricos fornecidos pelo usuário utilizando a função InputBox, representando os lados de um triângulo, e informe se ele é retângulo ou não.

EXEMPLO Escreva um algoritmo que leia um número fornecido pelo usuário utilizando a função InputBox, e informe se o número é par ou ímpar.

2.7.2 Input: Cells e Range

Uma das grandes vantagens de se utilizar o VBA, é a interação que podemos ter com os dados da própria planilha.

Uma das funções para acessar elementos das Células de uma planilha é justamente **Cells(l,c)**, em que **l** é a linha e **c** a coluna. Usando esta função, pensamos na planilha como uma matriz bidimensional com linhas e colunas (ambas começando em índices 1). A Figura 2.3 mostra como as células são numeradas.

O código abaixo lê os valores das células (1,1) e (1,2), imprimindo os seus conteúdos (usamos **.Value** para coletar o valor contido nas células).

```
Sub lendo_da_planilha()  
    Dim s1 As String, s2 As String  
    s1 = Cells(1, 1).Value  
    s2 = Cells(1, 2).Value  
    Debug.Print "Os valores das celulas sao : "; s1; s2  
End Sub
```

	A	B	C	D	E	F
1	(1,1)	(1,2)	(1,3)	(1,4)		Linha 1
2	(2,1)	(2,2)	(2,3)	(2,4)		Linha 2
3	(3,1)	(3,2)	(3,3)	(3,4)		Linha 3
4						
5	Coluna 1	Coluna 2	Coluna 3			
6						

Figura 2.3: Índices células Excel

Podemos ainda ler valores de abas diferentes, basta usar a função **Sheets(aba)**. antes de Cells, para primeiro selecionar de qual planilha os dados devem ser lidos. O argumento **aba** pode ser um índice ou mesmo o nome da planilha (entre aspas). O código abaixo lê os valores da célula (1,1) da aba1 e da aba2, um pelo índice e o outro pelo nome da aba:

```
Sub lendo_da_planilha_abas_diferentes()
    Dim a1 As String, a2 As String
    a1 = Sheets("Plan1").Cells(1, 1).Value
    a2 = Sheets("Planilha1").Cells(1, 1).Value
End Sub
```

A função **Cells** permite o acesso de células individuais da planilha, para acessarmos faixas (ou *ranges*) usamos a função **Range(C1,C2)**. **C1** e **C2** são strings que vão determinar o intervalo, sendo que **C2** é opcional (caso uma única célula seja selecionada). As strings de **C1** e **C2** são escritas pela notação da própria planilha, e não linhas e colunas. (por exemplo célula "A1"). O código abaixo lê e imprime na janela de verificação o valor contido na célula "A1".

```
Sub main()
    Dim a As String
    a = range("A1").Value
    Debug.Print a
End Sub
```

Usando o argumento da **Cell2** com um ponto-circula, conseguimos selecionar um intervalo. O código abaixo seleciona e imprime o conteúdo das células de "A1" até "F1". (A Figura 2.4 mostra o intervalo na planilha).

```
Sub main()
    Dim a As String
    For Each e In range("A1:F1").Value
        Debug.Print e
    Next e
End Sub
```

	A	B	C	D	E	F
1	1	2	3	4	5	6
2						

Figura 2.4: Range "A1:A7"

Com o range podemos selecionar conjuntos de células que não fazem parte da mesma linha. Nesses casos determinamos um retângulo de valores, em que **C1** representa o canto superior esquerdo e **C2** o canto inferior direito do retângulo. O código abaixo delimita o retângulo formado por "A1:F4" (a Figura 2.5 apresenta o intervalo, com **C1** e **C2** destacados em azul).

```
Sub main()
    For Each e In range("A1:F4").Value
        Debug.Print e
    Next e
End Sub
```

	A	B	C	D	E	F	G
1	1	2	3	4	5	6	
2	3	4	5	6	7	8	
3	3	1	2	3	4	5	
4	1	2	3	4	5	6	
5							

Figura 2.5: Range "A1:F4"

2.7.3 Input: Arquivo de texto

Em algumas situações a informação que desejamos usar está em arquivos de texto externos. Existem diversas formas para ler arquivos de texto, uma delas é ler o conteúdo linha a linha fazendo o *parse* das informações. O código abaixo mostra um *template* para ler o conteúdo de um arquivo .txt chamado Matriz e imprimir o conteúdo na janela imediata. OBS: Muitos dos códigos que utilizamos são "receitas de bolo", ou seja, não precisamos entender completamente o que o código está fazendo, mas sim saber utiliza-lo. Esse é o caso da leitura de arquivos, em muitos projetos precisamos ler arquivos de input externos, nessa situação, basta copiar e colar a "receita", sabendo adaptá-la para se adequar aos novos dados lidos.

```
Sub lendo_arquivos()
    Dim arquivo As String, linha As String
    arquivo = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\3 - Programacao (VBA)\Matriz.txt"

    ' Abre o arquivo
    Open arquivo For Input As #1
    ' Itera sobre todas as linhas do arquivo, imprimindo-as
    Do Until EOF(1)
        Line Input #1, linha
        Debug.Print linha
    Loop
    Close #1
End Sub
```

As 2 primeiras linhas definem a string que contém a caminho do arquivo a ser lido e a linha. Por este método os conteúdos do arquivo são lidos linha a linha (se o template não funcionar, veja observação ao final da subseção). A partir de **Open** o arquivo é aberto com o nome #1. O laço **Do** itera sobre todas as linhas do arquivo até o fim EOF (End of File), sendo que a cada iteração o conteúdo da linha fica salvo na variável **linha**, e é impresso na janela de verificação imediata.

Para usar o template, basta substituir o valor da variável 'arquivo' pelo caminho do arquivo que se deseja ler.

Somente ler e imprimir as linhas de um arquivo não é muito útil, dado que uma linha pode conter diversas informações. Veremos mais adiante manipulação de strings e vetores, mas uma forma simples de se separar as strings do texto lido é utilizar a função **Split(string, delimiter)**, em que **string** é o texto que queremos separar e **delimiter** o delimitador que queremos utilizar (espaço, ponto e vírgula por exemplo). Essa função retorna um vetor de strings, em que a cada posição do vetor uma string separada do texto original é armazenada. O código abaixo lê uma matriz separada por espaços de um arquivo de texto e em seguida separa os valores em um vetor.

```

Sub lendo_matriz()
Dim arquivo As String, linha As String
arquivo = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\3 - Programacao (VBA)\Matriz.txt"
Dim v_resultado() As String

' Abre o arquivo
Open arquivo For Input As #1
' Itera sobre todas as linhas do arquivo, imprimindo-as
Do Until EOF(1)
Line Input #1, linha
v_resultado() = Split(linha, " ") ' Quebra a linha por espacos, e atribui a um
vetor
Debug.Print v_resultado(0), v_resultado(1) ' imprime duas posicoes do vetor
Loop
Close #1
End Sub

```

Um exemplo de arquivo .txt é mostrado na Figura 2.6).

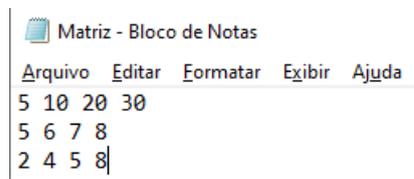


Figura 2.6: Lendo matriz linha a linha

EXEMPLO: Algumas vezes o próprio arquivo de texto mantém informações sobre seus dados. Por exemplo, considere um arquivo que contenha uma lista de vendas de um produto, porém o primeiro número do arquivo se refere à quantidade de vendas que o arquivo contém. Assim, não podemos usar o laço **Do Until EOF**. Dessa forma, inicialmente lemos a primeira linha do arquivo e salvamos em um inteiro, em seguida usamos um laço **For** para iterar os itens até o número lido. O código abaixo faz essa leitura e imprime os valores na janela de verificação imediata. Um exemplo de arquivo .txt dessa forma pode ser visto na Figura 2.7

```

Sub main()

Dim arquivo As String, textline As String
arquivo = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\3 - Programacao (VBA)\vendas.txt"
Dim v_resultado() As String

Open arquivo For Input As #1
Line Input #1, textline
v_resultado() = Split(textline)
Debug.Print v_resultado(0)
Dim n As Integer
n = Int(v_resultado(0))
For i = 1 To n
Line Input #1, textline
v_resultado() = Split(textline, " ")
Debug.Print v_resultado(0)
Next i
Close #1
End Sub

```

OBSERVAÇÃO: O método *Line Input #* busca a separação das linhas por Chr(13) ou uma sequência Chr(13)Chr(10). Alguns sistemas operacionais geram arquivos em que a quebra de linha é feita somente por Chr(10), de forma que *Line Input #* vai ler o arquivo inteiro sem identificar a quebra por linhas. Nesses casos, podemos ler o arquivo e forçar uma quebra pelo Chr(10).

O código abaixo faz a leitura de um arquivo de texto em que isso acontece (o arquivo faz parte da biblioteca de instâncias do problema do caixairo viajante, disponível em <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/>).

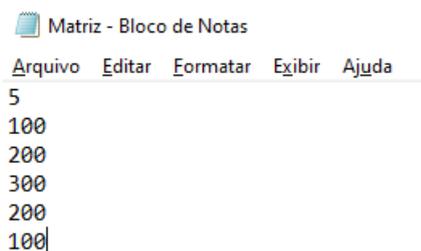


Figura 2.7: Lendo arquivo com indicação de linhas a serem lidas

```

Sub lendo_arquivos()
    Dim arquivo As String, linha As String
    arquivo = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\3 - Programacao (VBA)\Exemplos\
        InstanciasTSP\att48.TSP"

    'Abre o arquivo
    Open arquivo For Input As #1
        Line Input #1, linha

        ' Verifica se existe o caracter Chr(10) no texto lido
        If InStr(linha, Chr$(10)) > 0 Then
            MsgBox "A linha é separada por Chr(10)"
        End If

        Dim v_linhas_separadas() As String
        v_linhas_separadas = Split(linha, Chr$(10))

        For i = 0 To UBound(v_linhas_separadas)
            MsgBox v_linhas_separadas(i)
        Next i
    Close #1
End Sub

```

Existe uma condicional que verifica que de fato existem Chr(10) no conteúdo lido. Em seguida ele é separado por Chr(10) e todos os valores das linhas mostrados ao usuário por meio de **Msgbox**.

2.7.4 Output: MsgBox

A função **Msgbox()** é o equivalente do **Inputbox()** exibe uma mensagem para o usuário. Em sua forma mais simples, a **Msgbox()** apresenta uma string com uma mensagem para o usuário e um botão de "ok":

```

Sub msg_box()
    MsgBox "mensagem a ser exibida"
End Sub

```

Podemos alterar os botões da caixa de mensagem no segundo argumento. Abaixo a mensagem mostra o botão de "ok" e "cancelar".

```

Sub msg_box_cancelar_ok()
    MsgBox "Deseja continuar?", vbOKCancel
End Sub

```

Uma mensagem com os botões "sim" e "não" fica:

```

Sub msg_box_sim_nao()
    MsgBox "Sim ou nao?", vbYesNo
End Sub

```

Para usarmos as informações digitadas nos botões pelo usuário, precisamos usar os números referentes a cada botão. Cada botão tem um número inteiro que o representa, de forma que armazenamos esse número em uma variável, verificamos qual é o número, e tratamos a informação da forma desejada. A Tabela 2.4 mostra todos os botões, suas constantes e seus números associados:

O código abaixo utiliza os botões sim e não, ou seja "vbYes" e "vbNo". Dependendo do que o usuário digitou, uma nova mensagem aparece informando a decisão:

Botão clicado	Constante	Número
Ok	vbOk	1
Cancel	vbCancel	2
Abort	vbAbort	3
Retry	vbRetry	4
Ignore	vbIgnore	5
Yes	vbYes	6
No	vbNo	7

Tabela 2.4: Botões MsgBox e valores numéricos

```

Sub msg_box_cancelar_ok()
    Dim escolha As Integer
    ' O numero vai para a variavel OBS: NOTE QUE MSGBOX ESTA COM PARENTESSES!
    escolha = MsgBox("mensagem a ser exibida", vbYesNo)

    If escolha = 6 Then
        MsgBox ("Escolheu SIM")
    Else
        MsgBox ("Escolheu NAO")
    End If
End Sub

```

Note que quando usamos o valor digitado pelo usuário, os parâmetros da **MsgBox** devem estar entre parênteses! Veremos isso mais a frente, mas se trata do comportamento de Funções vs Subs.

2.7.5 Output: Cells e Range

Da mesma forma que usamos **Cells** e **Range** para ler informações das planilhas, podemos da mesma forma usa-los para preencher as planilhas. O código abaixo preenche as células (1,1) até (10,1) com valores numéricos (usando **Cells**):

```

Sub preenche_cells()
    For i = 1 To 10
        Cells(i, 1).Value = i
    Next i
End Sub

```

Já o código abaixo preenche o retângulo formado pelas células "A1" até "I14" com a palavra "OK", usando o **Range** (para lembrar como o retângulo é formado veja a seção 2.7.2).

```

Sub preenche_cells_range()
    For each e in Range("A1:I14")
        e.Value = "OK"
    Next e
End Sub

```

Além de textos com a propriedade **.Value** em **Cells** e **Range**, podemos alterar e inserir qualquer coisa nas células, como se estivéssemos formatando diretamente na planilha. O código abaixo usa a propriedade **.Interior.Color** para atribuir a cor verde às células do retângulo ("A1:I14"), e também insere a fórmula "=SOMA(J1:K14)":

```

Sub altera_cor_formula()

    For Each e In range("A1:I14")
        e.Interior.Color = vbGreen 'Pinta as celulas
        e.Formula = "=SOMA(J1:K14)" 'Insere as formulas
    Next e
End Sub

```

EXEMPLO: Crie um código que peça ao usuário um número de linhas e em seguida um número de colunas. Considerando essas linhas e essas colunas, pinte as células das linhas intercalando duas cores diferentes (verde e azul).

```

Sub pinta_celulas()

    Dim linhas As Integer, colunas As Integer

    linhas = InputBox("Digite um numero de linhas :")
    colunas = InputBox("Digite um numero de colunas :")

    For i = 1 To linhas
        For j = 1 To colunas
            If i Mod 2 = 0 Then
                Cells(i, j).Interior.Color = vbBlue
            Else
                Cells(i, j).Interior.Color = vbGreen
            End If
        Next j
    Next i

End Sub

```

2.7.6 Output: Arquivos

Da mesma forma que podemos abrir e ler arquivos de texto, também podemos gerar e exportar informações para eles. O código a seguir é um *template* para escrever em um arquivo chamado "ArquivoDeTexto.txt". Para utilizar, basta mudar o caminho e o nome do arquivo.

```

Sub escreve_arquivo()

    Dim caminho As String
    Dim FileNumber As Integer

    caminho = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\3 - Programacao (VBA)\ArquivoDeTexto.txt"
    FileNumber = FreeFile

    Open caminho For Output As FileNumber
        Print #FileNumber, "Escrevendo"
        Print #FileNumber, "um"
        Print #FileNumber, "arquivo!"
    Close FileNumber

End Sub

```

Todo o conteúdo após **Print #FileNumber**, será impresso no arquivo. Note que desta forma os dados são impressos em linha, ou seja, a cada **Print #FileNumber** uma nova linha é criada. Podemos escrever tudo na mesma linha colocando um ponto-e-vírgula após a string:

```

Sub escreve_arquivo()

    Dim caminho As String
    Dim FileNumber As Integer

    caminho = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\3 - Programacao (VBA)\ArquivoDeTexto.
txt"
    FileNumber = FreeFile

    Open caminho For Output As FileNumber
        Print #FileNumber, "Tudo ";
        Print #FileNumber, "na ";
        Print #FileNumber, "mesma ";
        Print #FileNumber, "frase ";
    Close FileNumber

End Sub

```

2.8 Estrutura de dados: arrays

Em VBA um *array* é uma estrutura de dados similar ao vetor de álgebra linear (para uma introdução, veja o capítulo 1). Os vetores servem para armazenar conjuntos de valores, que são acessados por seus índices na estrutura. Ao se declarar um vetor, definimos as suas dimensões, ou seja, uma matriz nada mais é do que um vetor com duas dimensões.

2.8.1 Estáticos vs dinâmicos

Existem 2 tipos de vetores em VBA, os **estáticos** e os **dinâmicos**. Os vetores estáticos têm um número fixo predeterminado de elementos que podem ser armazenados. Não se pode alterar o tamanho do tipo de dados de um *array* estático. Os vetores estáticos devem ser usados nas situações em que devemos armazenar um conjunto de valores que temos certeza que não irão mudar, por exemplo, dias da semana.

A sintaxe para se declarar um vetor estático é mostrada no código abaixo:

```
Sub declara_array()
' Declara um vetor com 5 elementos (de 0 a 5), do tipo real
Dim vetor(4) As Integer
End Sub
```

Note que o número 4 se refere ao último índice do vetor. Como o índice dos vetores é iniciado em 0, vetor(4) implica que existem 5 elementos no vetor (0,1,2,3 e 4) do tipo real (Double).

Para se alterar ou acessar os elementos do vetor, basta usar parênteses com o índice desejado. O código abaixo faz a atribuição de valores ao vetor e salva esses valores do vetor na planilha:

```
Sub declara_array()
Dim vetor(4) As Double 'declara o vetor
' atribui valores
vetor(0) = 10.5
vetor(1) = 10.4
vetor(2) = 10.5
vetor(3) = 18.5
vetor(4) = 5.5

' acessa os valores e salva na planilha
Cells(1, 1).Value = vetor(0)
Cells(2, 1).Value = vetor(1)
Cells(3, 1).Value = vetor(2)
Cells(4, 1).Value = vetor(3)
Cells(5, 1).Value = vetor(4)
End Sub
```

Já os vetores **dinâmicos** não têm um número pré-definido de elementos, de forma que podem ser redimensionados durante a execução do código. A sintaxe para se declarar um vetor dinâmico é a seguinte (idêntica aos estáticos, porém sem determinar o número de elementos):

```
Sub declara_array_dinamico()
Dim vetor_dinamico() As Double
End Sub
```

Para que possamos utilizar o vetor, no entanto, devemos definir o seu tamanho. Isso é feito com a função **ReDim** (de redimensionamento). A sintaxe é mostrada abaixo:

```
Sub declara_array_dinamico()
Dim vetor_dinamico() As Double 'declaramos sem o numero de elementos

ReDim vetor_dinamico(4)

vetor_dinamico(0) = 10.5
vetor_dinamico(1) = 10.4
vetor_dinamico(2) = 10.5
vetor_dinamico(3) = 18.5
vetor_dinamico(4) = 5.5

Cells(1, 1).Value = vetor_dinamico(0)
Cells(2, 1).Value = vetor_dinamico(1)
Cells(3, 1).Value = vetor_dinamico(2)
Cells(4, 1).Value = vetor_dinamico(3)
Cells(5, 1).Value = vetor_dinamico(4)
End Sub
```

Os vetores dinâmicos são perfeitos para acomodar dados que são variáveis, como dados fornecidos pelo usuário, por exemplo. O código abaixo declara um vetor dinâmico e pede ao usuário que determine o seu tamanho. Em seguida o vetor do tamanho estipulado é redimensionado e valores sequenciais são atribuídos a ele, para em seguida serem impressos na planilha do excel.

```

Sub tamanho_vetor()
Dim tamanho As Integer
Dim V() As Integer

' Pede um inteiro para o usuario
tamanho = InputBox("Digite o tamanho do vetor") ' note o -1!

'Redimensiona o vetor
ReDim V(tamanho - 1)

' Atribui valores
For i = 0 To tamanho - 1
    V(i) = i
Next i

'Imprime valores
For i = 0 To tamanho - 1
    Cells(i + 1, 1).Value = V(i)
Next i
End Sub

```

Note que o vetor é dimensionado com o número digitado pelo usuário - 1, isso se dá pois o vetor começa a sua indexação em 0.

O vetor pode ser redimensionado quantas vezes forem necessárias, porém a cada vez que o **ReDim** é usado, todos os valores do vetor são apagados.

EXEMPLO: Escreva um código que leia um número genérico de elementos de um arquivo de texto, sendo que o primeiro número é um indicador de quantos números existem no arquivo. Todos os números devem ser salvos em um vetor, e em seguida o vetor deve ser percorrido e todos os números pares impressos na planilha do excel.

```

Sub tamanho_vetor()
Dim tamanho As Integer
Dim V() As Integer

Dim arquivo As String, textline As String
arquivo = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\3 - Programacao (VBA)\Matriz.txt"
Dim v_resultado() As String

Open arquivo For Input As #1
Line Input #1, textline
v_resultado() = Split(textline)
tamanho = CInt(v_resultado(0))
ReDim V(tamanho - 1)
For i = 0 To tamanho - 1
    Line Input #1, textline
    v_resultado() = Split(textline, " ")
    V(i) = CInt(v_resultado(0))
Next i
Close #1

Dim indice_celula As Integer
indice_celula = 1

For i = 1 To tamanho
    If V(i - 1) Mod 2 = 0 Then
        Cells(indice_celula, 1).Value = V(i - 1)
        indice_celula = indice_celula + 1
    End If
Next i
End Sub

```

2.8.2 Vetores multidimensionais

Como dito, os vetores em VBA podem ter mais de uma dimensão (no máximo 32). As dimensões dos vetores são especificadas da sua declaração (ou redimensionamento) separado por vírgulas. O código abaixo declara um vetor de duas dimensões (ou seja, uma matriz) e faz a atribuição de alguns valores aos seus elementos.

```

Sub multidim_vetor()

    Dim matriz(2, 2) As Integer

    matriz(0, 0) = 0
    matriz(0, 1) = 0
    matriz(0, 2) = 0
    matriz(1, 0) = 0
    matriz(1, 1) = 0
    matriz(1, 2) = 0
    matriz(2, 0) = 0
    matriz(2, 1) = 0
    matriz(2, 2) = 0

End Sub

```

Cada número no dimensionamento do vetor indica quantos elementos aquela dimensão em específico terá, no caso acima, `matriz(2,2)` indica 3 elementos na primeira dimensão e 3 na segunda (lembrando que 2 se refere ao último índice, portanto 3 elementos). Os elementos são acessados pelos índices de cada dimensão; `matriz(0,1)` significa matrix no índice 0 da dimensão 1 e no índice 1 da dimensão 2.

Também podemos criar vetores multidimensionais dinâmicos, a única diferença é que quando fizermos o redimensionamento com **ReDim** todas as dimensões devem ser especificadas. O código abaixo declara um vetor dinâmico e em seguida o redimensiona como uma matriz 2x3 (duas linhas por 3 colunas).

```

Sub multidim_vetor_dinamico()
    ' Declarando um vetor dinamico (identico ao com uma dimensao)
    Dim matriz() As Integer

    ' redimensionando: duas dimensoes
    ReDim matriz(1, 2)

    ' fazendo uma atribuicao
    matriz(1, 1) = 100
    Debug.Print matriz(1, 1)
End Sub

```

2.8.3 Métodos mais usados

Existem algumas funções (ou métodos próprios) que são muito usados em vetores. Por exemplo a função **UBound(v,d)**. Essa função toma um vetor como argumento e retorna o maior índice de acesso disponível. **UBound()** é muito utilizado para realizar a iteração nos elementos do vetor, quando não sabemos o seu tamanho. O código abaixo declara um vetor de uma dimensão com 3 elementos e mostra para o usuário o retorno da função **UBound** por meio de uma **Msgbox**.

```

Sub ubound_vetor()
    Dim v(3) As Integer
    Dim tamanho As Integer
    tamanho = UBound(v)
    MsgBox tamanho
End Sub

```

O segundo argumento da função é opcional, e se refere à qual dimensão do vetor queremos saber o índice. Se nenhum valor é passado é assumido 1 (ou seja, a primeira dimensão do vetor). O código abaixo declara uma matriz 2x5 e mostra ao usuário o último índice da primeira dimensão (1) e em seguida da segunda (4).

```

Sub ubound_vetor_multidimensional()
    Dim v(1, 4) As Integer
    MsgBox UBound(v, 1) ' ultimo indice da primeira dimensao
    MsgBox UBound(v, 2) ' ultimo indice da segunda dimensao
End Sub

```

Como dito, **UBound** é muito útil para iterar sobre os elementos de um vetor. O código abaixo declara um vetor de duas dimensões (uma matriz 5x10), e em seguida percorre todos os elementos das linhas e das colunas por meio do laço **For**, usando os limites de **UBound** das duas dimensões.

```

Sub iteracao_vetores_ubound()
    Dim matriz(4, 9) As Integer
    For i = 0 To UBound(matriz, 1)
        For j = 0 To UBound(matriz, 2)
            matriz(i, j) = i + j
        Next j
    Next i

    ' imprime os dados da matriz criada na planilha
    For i = 0 To UBound(matriz, 1)
        For j = 0 To UBound(matriz, 2)
            Cells(i + 1, j + 1).Value = matriz(i, j)
        Next j
    Next i
End Sub

```

Se necessitarmos alterar as dimensões de um vetor dinâmico mantendo os elementos que já estão salvos, usamos a instrução **Redim Preserve**. Se o **Redim Preserve** for usado diminuindo o número de elementos do vetor, os elementos com índices acima do desejado serão deletados.

```

Sub redim_preserve()
    Dim v() As Integer

    ReDim v(2)
    v(0) = 10
    v(1) = 10
    v(2) = 10

    Debug.Print "Antes o redimensionamento"
    For i = 0 To UBound(v)
        Debug.Print v(i)
    Next i

    ReDim Preserve v(3)
    v(3) = 99

    Debug.Print "Apos o redimensionamento"
    For i = 0 To UBound(v)
        Debug.Print v(i)
    Next i
End Sub

```

Não é possível alterar a dimensão de um vetor (por exemplo transformar um vetor de uma dimensão em um de duas), e também só é possível alterar os valores da última dimensão dos vetores. Por exemplo, em uma matriz de duas dimensões, podemos alterar somente a segunda com **Redim Preserve**.

Muitas vezes não sabemos a dimensão do vetor, nem mesmo em tempo de execução, sendo que os elementos vão sendo adicionados quando necessário. É possível usar vetores para essas situações, juntamente com o **ReDim Preserve** (a cada novo item adicionado um novo redimensionamento é feito com uma posição a mais no vetor).

Considere o código abaixo. Um vetor dinâmico é declarado e a cada iteração do laço for, o mesmo é redimensionado para acomodar um elemento a mais (**UBound(v) + 1**), e esse elemento é adicionado no último índice disponível (**UBound(v)**).

```

Sub redim_preserve_iteracao()
    Dim v() As Integer

    ReDim Preserve v(0)
    v(0) = 0

    For i = 0 To 10
        ReDim Preserve v(UBound(v) + 1)
        v(UBound(v)) = i
    Next i

    For i = 0 To UBound(v)
        Debug.Print v(i)
    Next i
End Sub

```

2.9 Subrotinas

O conceito de subrotinas está presente em todas as linguagens de programação, e deve ser bem compreendido para que códigos grandes possam ser criados de forma limpa e eficiente. De forma geral, uma subrotina pode ser pensada como uma *parte* do código principal que está escrita fora do código principal (em outro arquivo, por exemplo). Pode parecer contra-intuitivo, mas existem duas razões principais para essa prática:

1. **Reutilização:** Imagine que você implementou o algoritmo para resolver uma equação de grau 2, e esse código tomou 15 linhas de programa. Em uma aplicação maior, essa funcionalidade de resolver a equação deve ser usada 30 vezes. Uma opção é copiar e colar as 15 linhas de código a cada vez que for necessário, mas isso deixaria o código muito maior do que o necessário. Para isso, criamos uma *subrotina* em outra parte do arquivo, com o código que resolve a equação, e sempre que precisarmos, somente a usamos (*chamamos* a subrotina). Dessa forma é possível reutilizar muitos trechos de código que são repetidos.
2. **Organização:** Em alguns momentos, mesmo que o código seja utilizado somente uma vez, é uma boa prática criar um local separado para ele. Imagine testar um código de 2000 linhas, com muitas funcionalidades complexas. Podemos separar o código em diversas partes menores, e definir o que cada parte deve fazer, em seguida testamos cada parte (subrotina) separadamente, o que deixa o código muito mais legível e organizado.

2.9.1 Conceitos gerais

Nesta primeira parte será explicado o conceito de subrotinas inerente à qualquer linguagem de programação. Para isso, outros tópicos subjacentes também serão abordados. Em seguida os conceitos serão elucidados pela linguagem de programação VBA (as sintaxes serão diferentes para as diferentes linguagens, porém os conceitos sempre serão os mesmos - salvo raras exceções...)

Como a subrotina é executada?

Primeiramente é preciso entender a *ordem* em que as subrotinas são executadas. Suponha que existe um código principal e uma subrotina escrita em outro lugar. O programa principal *chama* a subrotina em um determinado ponto, e partir deste momento, a execução do programa vai para a primeira instrução da subrotina. Quando a mesma for finalizada, o programa volta para a execução do código principal. Considere a Figura 2.8, em que a ordem de execução das instruções é numerada em vermelho.

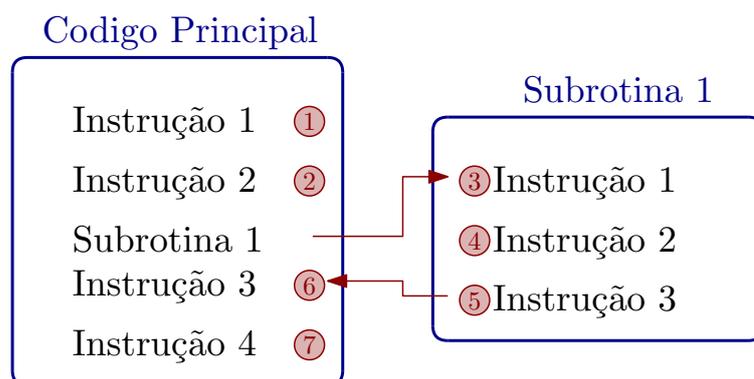


Figura 2.8: Execução da subrotina

Após a segunda instrução o programa passa imediatamente a executar a instrução 1 da subrotina, e só volta quando a subrotina estiver finalizada.

Via de regra, todos os códigos são subrotinas e depende de nós definirmos qual é o "código principal". De qualquer forma, após definido, é aí que o programa começa a sua execução. Como

não existe diferenciação entre subrotinas, uma subrotina pode chamar outra quantas vezes forem necessárias, e esta, por sua vez, também pode usar outras subrotinas! Considere a Figura 2.9 em que isso acontece.

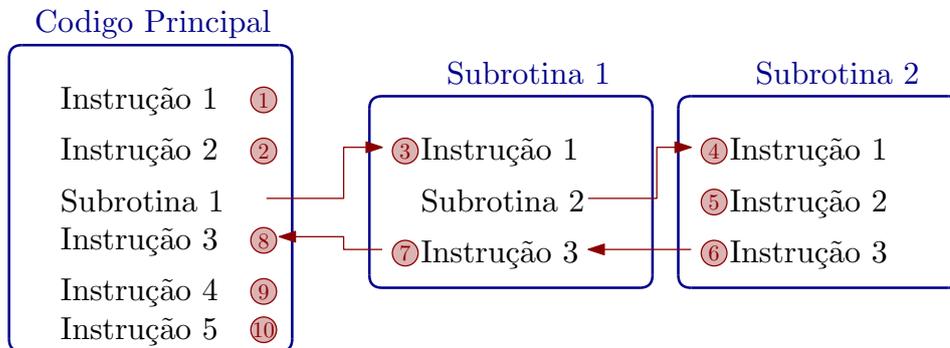


Figura 2.9: Execução da subrotina

OBS: O que aconteceria se uma subrotina chamasse a si mesma?

Escopo de variáveis

Definido como a subrotina é executada, surge uma dúvida: podemos usar as variáveis e estruturas de dados que *declaramos em uma subrotina na outra?* Para responder a essa pergunta precisamos conhecer o conceito de escopo das variáveis.

O escopo de variáveis determina onde as mesmas podem ser usadas. Como cada variável tem um "custo" para o computador, uma variável que pode ser usada em menos lugares é mais barata computacionalmente. Da forma como vinhamos declarando as variáveis, o escopo delas era somente na subrotina em que elas foram criadas, ou seja, em qualquer lugar entre as palavras **Sub** e **End Sub** poderíamos utilizá-la, porém em outras subrotinas elas não seriam válidas.

Podemos alterar o escopo de uma variável para que seja visível em todas as subrotinas, porém na maioria das vezes essa não é uma boa prática.

Uma outra pergunta que surge é a seguinte: existe alguma forma de, ao chamar uma subrotina de um código principal, passar algumas variáveis para que elas sejam processadas na subrotina?

Passando variáveis para as subrotinas

Imagine um código em que em uma primeira parte muitas informações são coletadas do usuário, bancos de dados, internet, para em seguida serem processadas. O código ficou muito grande e você deseja separar a aplicação em duas partes: a primeira que faz a coleta das informações e a segunda que faz todo o processamento e cálculos. Como passar todas essas informações entre uma subrotina e outra? Isso é feito com a *passagem de parâmetros*.

Os parâmetros de uma subrotina são variáveis passadas à mesma quando chamada. Dessa forma, quando a rotina principal for chamar a subrotina, todos os dados coletados serão passados e estarão disponíveis para processamento na própria subrotina.

Considere a Figura 2.10, em que a subrotina principal coleta dois números do usuário e armazena-os nas variáveis A e B. Em seguida chama a subrotina **Soma A,B**, faz o processamento das variáveis A e B e mostra o resultado ao usuário. Dessa forma *passamos os dados A e B como parâmetros* da subrotina Soma.

Dado que conseguimos passar valores de uma subrotina a outra, uma nova pergunta surge: o que acontece se alteramos os valores dos parâmetros dentro da subrotina? No exemplo anterior as variáveis que foram passadas (A e B) não foram alteradas. Considere o código da Figura 2.11, o que será mostrado ao usuário, 10 ou 20?

Essa pergunta vai depender da linguagem de programação a ser utilizada, porém existem duas possibilidades: ao passar o valor A para a subrotina, uma *cópia* é feita (automaticamente), de forma

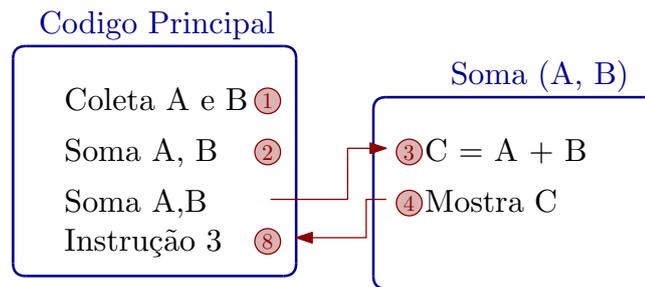


Figura 2.10: Passagem de valores à subrotina

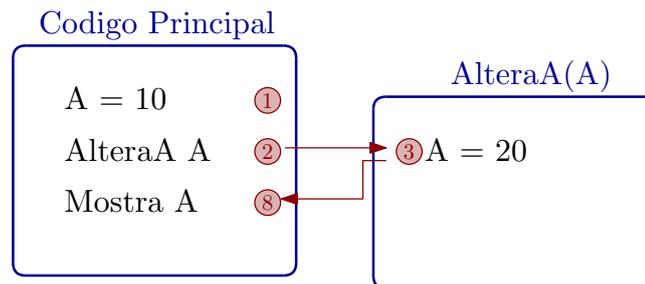


Figura 2.11: Alterando o valor do parâmetro

que qualquer alteração de A dentro da subrotina não é ecoado até a rotina principal (nesse caso o usuário veria o valor 10). A segunda possibilidade é que o valor passado *não é copiado*, de forma que a alteração dentro da sub altera o valor da variável fora, neste caso o valor mostrado ao usuário seria 20.

Este tópico é conhecido como *passagem de argumentos para funções por valor ou por referência*. Na linguagem VBA podemos escolher como desejamos passar os argumentos, se por *valor*, então uma cópia é feita dentro da subrotina, se por referência, a variável passada é de fato uma referência a original, de forma que qualquer alteração dentro da subrotina afetará a variável fora dela.

Retornando variáveis das subrotinas

Já vimos como mandar variáveis de uma rotina principal para que possam ser usadas em uma subrotina. Podemos também estar interessados em coletar algum valor da subrotina para usar na rotina principal. Isso se chama *retorno de variável*. Imagine o caso em que coletamos dois números do usuário na rotina principal, passamos esses valores para uma rotina de soma, e em seguida queremos mostrar o resultado para o usuário pela rotina principal. Para isso, retornamos a soma da subrotina e a atribuímos a uma variável na rotina principal. Considere a Figura 2.12 que exemplifica o caso.

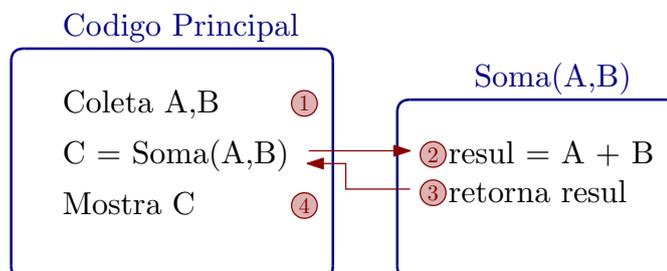


Figura 2.12: Alterando o valor do parâmetro

As variáveis A e B são somadas e atribuídas à variável resul dentro da subrotina Soma(A,B),

em seguida, a instrução "retorna resul" indica que o valor está sendo jogado para fora, ou seja, está sendo retornado da subrotina. Esse retorno é capturado pela variável C que está no código principal.

2.10 Subrotinas em VBA

Nesta Seção veremos como implementar os diversos casos vistos na seção anterior. Em VBA existe uma nomenclatura diferente para as subrotinas, dependendo das suas características. A Figura 2.13 abaixo exemplifica os casos.

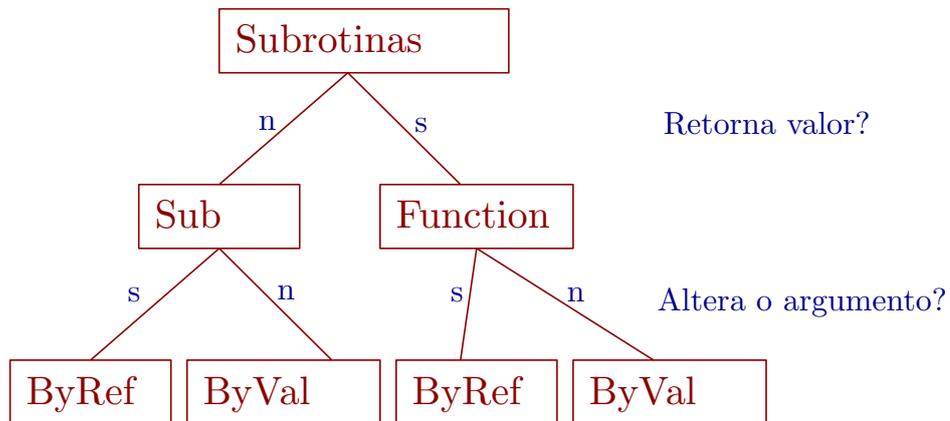


Figura 2.13: Alterando o valor do parâmetro

Basicamente existe uma primeira distinção entre as rotinas que retornam valor, chamadas **Functions**, e as que não retornam valor, chamadas **Subs** (usadas até agora). A segunda diferenciação (que vale tanto para Subs quanto para Functions) se refere a alteração ou não dos argumentos das funções. A seguir serão mostrados exemplos para cada caso.

2.10.1 Subs

As **Subs** são determinadas em blocos entre as palavras **Sub** e **End Sub**. Para chamarmos uma Sub que não possui parâmetros, basta escrevermos o seu nome. O código abaixo exibe uma mensagem de texto na sub principal e em seguida chama uma outra Sub, que exibe uma outra mensagem, finalmente uma terceira mensagem é exibida novamente na sub principal.

```

Sub sub_principal()
    MsgBox "Estou na principal!"
    sub_secundaria
    MsgBox "E agora voltei para a principal!"
End Sub

Sub sub_secundaria()
    MsgBox "Vim para a secundaria..."
End Sub
  
```

Agora adicionamos um argumento à sub_secundaria. Para iso, precisamos definir após os parênteses o tipo de variável do argumento, e se ele será alterado ou não (pelas palavras **ByRef** ou **ByVal**). OBS: Podemos não escrever ByRef ou ByVal, de forma que o padrão será assumido ByRef.

```

Sub sub_principal ()
    Dim a1 As String
    a1 = "Esta mensagem e um parametro"
    sub_secundaria a1
End Sub

Sub sub_secundaria(ByVal arg1 As String)
    MsgBox arg1
End Sub

```

Note que o nome do parâmetro definido na sub_secundaria *não precisa ser o mesmo da variável que será passada* (no caso acima o parâmetro se chama arg1 e a variável passada a1). Quando passada, usamos a1 dentro da subrotina como se ela se chamasse arg1! Ainda, quando chamamos a Sub colocamos o argumento a1 logo em seguida.

O código abaixo altera o valor do parâmetro dentro da subrotina, e depois imprime novamente uma mensagem ao usuário com o valor alterado (note que agora foi usado **ByRef** na definição do parâmetro).

```

Sub sub_principal ()
    Dim a1 As String
    a1 = "Nao quero ser alterada..."
    MsgBox "Mensagem antes da alteracao : " & a1
    sub_secundaria a1
    MsgBox "Mensagem apos a alteracao : " & a1
End Sub

Sub sub_secundaria(ByRef arg1 As String)
    'alterando a1
    arg1 = "Mas infelizmente eu fui..."
End Sub

```

Podemos colocar quantos argumentos precisarmos, no chamamento eles devem ser separados por virgulas. Considere o código abaixo que recebe dois valores do usuário e usa a Sub Soma(A,B) para somar os valores e mostrar o resultado ao usuário.

OBS: Note que para chamar uma sub a sintaxe é o nome dela seguido dos (possíveis) parâmetros, separados por virgulas.

```

Sub main()
    Dim A As Integer, B As Integer
    A = InputBox("Digite o primeiro numero")
    B = InputBox("Digite o segundo numero")
    Soma A, B
End Sub

Sub Soma(ByRef A As Integer, ByRef B As Integer)
    Dim resul As Integer
    resul = A + B
    MsgBox "O resultado da soma e : " & resul
End Sub

```

2.10.2 Function

Em VBA as **Functions** são Subs que retornam algum valor. Ou seja, uma sub chama uma Function e esta "joga" um valor para fora dela, que pode ser armazenado em uma variável na sub principal.

Para retornar o valor de uma Function escrevemos o próprio nome da Function seguida de "igual" e valor a ser retornado. O código abaixo é composto de uma sub principal e uma Function Soma(A,B). Na sub principal dois números são coletados do usuário. Em seguida esses números são passados para uma Function que realiza a soma dos mesmos e **retorna** o valor para a sub principal. Esse valor retornado é armazenado na sub principal e em seguida mostrado ao usuário.

```

Sub main()
    Dim A As Integer, B As Integer, resul As Integer
    A = InputBox("Digite o primeiro numero")
    B = InputBox("Digite o segundo numero")

    resul = Soma(A, B)
    MsgBox "O valor da soma e : " & resul
End Sub

Function Soma(ByRef A As Integer, ByRef B As Integer) as Integer
    Dim resul As Integer
    resul = A + B
    Soma = resul
End Function

```

Note que a sintaxe para escrever **Functions** é muito similar às subs, porém deve-se escrever **Function** e **End Function**, sendo que o tipo do retorno é definido com a palavra **As** [tipo retornado]. Ainda, uma outra diferença é no chamamento: em Functions deve-se usar parênteses após o nome da função, e se existem parâmetros, estes devem ficar dentro dos parênteses separados por vírgula.

A sintaxe para a passagem de argumentos (**ByRef** e **ByVal**) é a mesma das Subs.

2.11 Exercícios

2.11.1 Condicionais

1. Reescreva as expressões abaixo com a menor quantidade de parêntese possível (quando isso for factível), **mas sem alterar o resultado**. Valide as suas respostas no VBA imprimindo os valores das expressões, com e sem parênteses:

- (a) $A = 2 * (8 / (3 + 1))$
- (b) $A = 3 + (16 - 2) / (2 * (9 - 2))$
- (c) $A = (6 / 3) + (8 / 2)$
- (d) $A = ((3 + (8 / 2)) * 4) + (3 * 2)$
- (e) $A = (6 * (3 * 3) + 6) - 10$
- (f) $A = (((10 * 8) + 3) * 9)$
- (g) $A = ((-12) * (-4)) + (3 * (-4))$

2. Implemente os 3 algoritmos do exercício 1 (Algoritmo 1, Algoritmo 2 e Algoritmo 3), da seção 1.6.3 do capítulo 1.

2.11.2 Input/Output

1. Considere o código abaixo. Existe algum erro? Qual?

```

Sub caixa_de_entrada()
    Dim nome As Integer
    nome = InputBox("Digite seu nome")
    Debug.Print "Muito prazer, : "; nome
End Sub

```

2. Implemente os algoritmos dos exercícios 3f e 3g (equação de grau 2 e produção de chapas), usando a função InputBox para coletar os dados do usuário.
3. Considerando a Figura 2.14 e o algoritmo abaixo, determine o que será mostrado na janela de verificação imediata.

```

Sub quebra_cabeca()
    Debug.Print Cells(1, 1).Value
    Debug.Print Cells(3, 2).Value
    Debug.Print Cells(3, 4).Value
    Debug.Print Cells(5, 3).Value
End Sub

```

	A	B	C	D	E
1	você	errada	ou	a	
2	nós	aula	agora	baixa	
3	correta	conseguiu		imprimir	
4	porque	somente	nota	cadeira	
5	é	VBA	isso	alta	
6	mensagem	sem	programa	sentido	
7					
8					

Figura 2.14: Quebra cabeça Excel

4. Crie um algoritmo que leia as informações das células (1,1) e (1,2) da planilha do Excel. A célula (1,1) contém o número de linhas de uma matriz e célula (1,2) o número de colunas. A partir da linha 2 a matriz é escrita (uma matriz de inteiros). Os números da matriz devem ser lidos e impressos na janela de verificação rápida. A Figura 2.15 mostra um exemplo para a leitura de uma matriz 2x3. OBS: A leitura da matriz deve ser para um tamanho genérico de linhas e colunas.

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

Figura 2.15: Lendo matriz dos dados

5. Considere agora um arquivo de entrada com coordenadas (X,Y) no plano. A primeira linha informa quantos pares de coordenadas existem, e a partir da segunda linha, 2 valores reais, a coordenada X e a coordenada Y, tudo separado por espaços. Crie um código que leia este tipo de arquivo e salve na planilha do excel os pares, bem como a soma das duas coordenadas. A Figura 2.16 mostra um exemplo de arquivo de entrada e do resultado esperado na planilha:

Matriz - Bloco de Notas

Arquivo Editar Formatar

5

10 20

30 40

50 60

70 80

90 100

	A	B	C
1	10	20	30
2	30	40	70
3	50	60	110
4	70	80	150
5	90	100	190

(a) Arquivo de entrada
(b) Resultados

Figura 2.16: Lendo e somando coordenadas

6. Escreva um código que formate as células do Excel como um jogo tabuleiro de xadrez, ou seja, um quadrado de 8x8 células quadradas (procure qual propriedade das células alteram o a altura e largura), sendo que deve haver uma intercalação na coloração delas (DICA: Procure as propriedades **RowHeight** e **ColumnWidth**).

```

Sub pinta_xadres ()
Dim soma As Integer
soma = 0
For i = 1 To 8
For j = 1 To 8
soma = soma + 1
If soma Mod 2 = 0 Then
Cells(i, j).Interior.Color = vbBlack
Else
Cells(i, j).Interior.Color = vbGreen
End If
Next j
Rows(i).RowHeight = 10
Columns(j).ColumnWidth = 10
soma = soma + 1 'para alterar a ordem na proxima linha
Next i
End Sub

```

7. Escreva um algoritmo que leia a idade de uma pessoa (usando **Inputbox**) expressa em anos, meses e dias e escreva a idade dessa pessoa expressa apenas em dias (usando um **Msgbox**). Considerar ano com 365 dias e mês com 30 dias.
8. Escreva um algoritmo para ler o número total de eleitores de um município, o número de votos brancos, nulos e válidos. Calcular e escrever o percentual que cada um representa em relação ao total de eleitores. Use a função **Inputbox** para ler os dados e escreva os resultados na planilha usando **Cells**.
9. A jornada de trabalho semanal de um funcionário é de 40 horas. O funcionário que trabalhar mais de 40 horas receberá hora extra, cujo cálculo é o valor da hora regular com um acréscimo de 50%. Escreva um algoritmo que leia o número de horas trabalhadas em um mês, o salário por hora e escreva o salário total do funcionário, que deverá ser acrescido das horas extras, caso tenham sido trabalhadas (considere que o mês possua 4 semanas exatas). Use **Inputbox** e **Msgbox** para entrada e saída dos valores.
10. Escreva um algoritmo que leia uma quantidade genérica de números de um arquivo de texto. Imprima no excel usando a função **Range**, quantos números estão entre cada um dos intervalos abaixo:
 - (a) $(-\infty \leq N < 25)$
 - (b) $(25 \leq N < 50)$
 - (c) $(50 \leq N < 75)$
 - (d) $(75 \leq N < \infty)$

2.11.3 Vetores

1. Para cada um dos exercícios abaixo, considere a leitura de um vetor da planilha em que seu tamanho é definido na célula (1,1), e seus elementos escritos a partir da linha 2 como um vetor linha, como a Figura 2.17, e faça o que se pede:

	A	B	C	D	E
1	5				
2	2	3	4	2	3
3					
4					
5					

Figura 2.17: Leitura de vetor com 5 elementos

- (a) Some os elementos do vetor.

- (b) Calcule o número de elementos positivos, negativos e nulos.
- (c) Calcule a média dos elementos.
- (d) Encontra quantos valores estão acima e quantos estão abaixo da média.
- (e) Encontra o maior elemento.
- (f) Encontra o menor elemento.
- (g) Normaliza o vetor (encontra um vetor em que cada elemento é igual ao seu próprio valor dividido pelo maior elemento do vetor). OBS: Para esse caso considere somente vetores com elementos positivos.
- (h) Encontre o desvio-padrão amostral ($\hat{\sigma}$) dos valores. Seja x um vetor, x_i o seu i -ésimo elemento, \bar{x} a média dos elementos e n o número de elementos do vetor. O desvio-padrão $\hat{\sigma}$ é dado por:

$$\hat{\sigma} = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}} \quad (2.1)$$

- (i) **DESAFIO** Encontre a frequência de ocorrência de cada um dos elementos do vetor.
 - (j) **DESAFIO** Crie um algoritmo que ordene o vetor de forma crescente.
2. (MINI PROJETO COVID) Considere os dados da planilha [daily-covid-deaths.xlsx](#). A coluna *Daily new confirmed deaths due to COVID-19* contabiliza o número de mortes em um dia, referente ao país definido na coluna *Entity*. Considere que todos os dados de um mesmo país estão em sequência, e todos os dias uma nova planilha atualizada será passada. Deseja-se extrair algumas informações sobre os dados do Brasil de forma genérica, ou seja, que funcionem em todas as vezes que novas planilhas forem usadas. Para isso, realize as seguintes tarefas:
 - (a) Aloque todos os dados de mortes referentes ao Brasil em um vetor. (DICA: primeiro identifique as linhas de início e término referentes aos dados do Brasil)
 - (b) Usando o vetor, calcule a média de ocorrências de mortes por COVID no Brasil.
 - (c) Calcule uma estimativa de mortes para os próximos 2 dias usando o método da média móvel com 5 elementos.
 3. Para cada um dos exercícios abaixo, o(s) vetor(es) podem ser lidos de qualquer forma.
 - (a) Dados dois vetores de tamanho n , faça uma subrotina que diga se eles são iguais ou diferentes.
 - (b) Faça um programa que, dados dois vetores de mesmo tamanho, gere um terceiro vetor, cujos valores deverão ser compostos pelos elementos intercalados dos dois outros vetores.
 - (c) Escreva uma subrotina que, dado um vetor, inverta os elementos desse vetor. Faça a inversão no próprio vetor, ou seja, não crie um vetor auxiliar.
 4. Considere as seguintes sequências matemáticas. Para cada uma delas, escreva os 100 primeiros termos em um vetor:
 - $\{n\}, n = 1, \dots, 100$
 - $\left\{\frac{n}{n+1}\right\}, n = 1, \dots, 100 = \left\{\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \dots\right\}$
 - $\left\{\frac{(-1)^n(n+1)}{3^n}\right\}, n = 1, \dots, 100 = \left\{-\frac{2}{3}, \frac{3}{9}, -\frac{4}{27}, \dots\right\}$

2.11.4 Matrizes

1. Para cada um dos exercícios de matrizes abaixo, considere a leitura dos dados feita como no exercício 4 (armazenando os valores em uma matriz).
 - (a) Calcule o número total de elementos da matriz.
 - (b) Calcule a soma de todos os elementos.
 - (c) Calcule o menor e o maior elementos da matriz.
 - (d) Calcule a média de todos os elementos.
 - (e) Calcule o número de elementos que estão acima e abaixo da média.
 - (f) Calcule a média dos elementos por linha (um vetor de médias).
 - (g) Calcule a média dos elementos por coluna (um vetor de médias).

2. Para cada um dos exercícios de matrizes abaixo, considere a leitura de uma matriz quadrada (número de linhas igual ao número de colunas).
- (a) Calcule a soma da diagonal principal da matriz. A diagonal principal de uma matriz $A_{m \times m}$, com elementos a_{ij} se refere aos elementos em que $i = j$. Assim, a soma da diagonal principal é dada por:
- $$\sum a_{ii} \quad (2.2)$$
- (b) Calcule a soma dos elementos acima da diagonal principal.
 (c) Calcule a soma dos elementos abaixo da diagonal principal.
3. Considere coordenadas (x,y) na planilha, sendo que a célula A1 contém o número de coordenadas que devem ser lidas, e a partir de A2 as coordenadas são escritas (uma ao lado da outra, ou seja, [A2,B2],[A3,B3],...), como na Figura 2.18.

	A	B	
1	3		
2	10	5	
3	4	4	
4	-1	10	
5			
6			

Figura 2.18: Leitura de coordenadas

Considerando essas coordenadas, calcule:

- (a) A matriz de distâncias D , ou seja, a matriz com as distâncias de todos os pontos para todos os outros. Considere a distância d_{p_1,p_2} entre os pontos $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ como sendo:

$$d_{p_1,p_2} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2.3)$$

Um exemplo de matriz de distancias para um conjunto de pontos é mostrado na Figura.

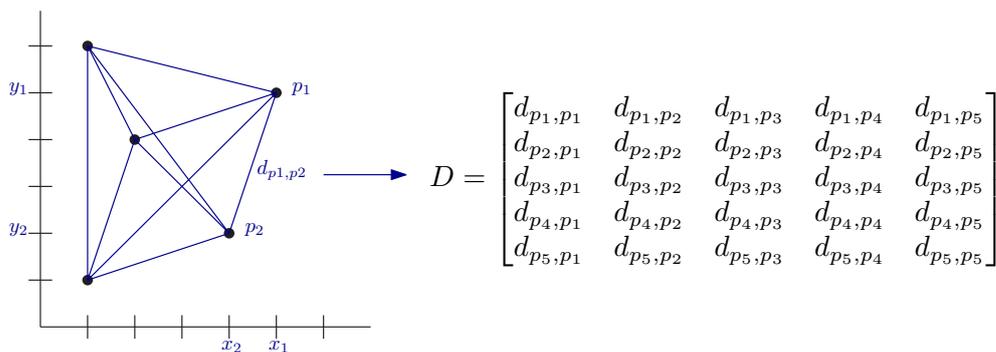


Figura 2.19: Matriz de distâncias

- (b) Com a matriz de distâncias, podemos criar e avaliar soluções para problemas de roteamento. Considere um problema que deve sair de um ponto (qualquer), visitar todos os outros pontos e retornar ao ponto inicial. Podemos armazenar essa rota como uma ordem pela qual os pontos devem ser visitados. Por exemplo a rota $r_1 = [1, 2, 3, 4, 5]$ parte do ponto 1, vai para o ponto 2 e assim por diante (considera-se que após o último ponto não é necessário reescrever o primeiro novamente). Esse tipo de solução pode ser armazenada como um vetor de inteiros, em que cada inteiro representa um ponto. Considerando a matriz e um vetor com uma rota, crie um algoritmo para calcular a distancia total da

rota. Por exemplo, a distância total da rota r_1 mostrada acima é composta pela distancias $d_{p_1,p_2} + d_{p_2,p_3} + \dots d_{p_5,p_1}$.

- (c) Ainda com base na matriz, crie diversas rotas de forma aleatória, calcule e imprima as rotas e as distancias. DICA: Uma forma de criar uma rota aleatória é gerar uma rota sequencial e fazer o swap aleatório de diversos elementos. A Figura abaixo ilustra o processo:

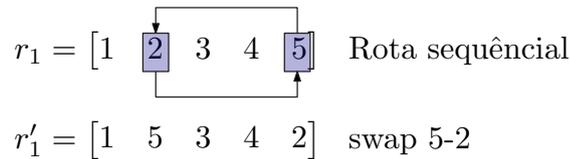


Figura 2.20: Swap de elementos

2.11.5 Subrotinas - subs vs functions

1. Para que serve uma subrotina (sub)?
2. Considere um código que realiza 3 tarefas: 1 - calcula a média de 2 valores fornecidos pelo usuário; 2 - encontra a soma dos n primeiros inteiros (com n fornecido pelo usuário); e 3 - escreve uma mensagem na célula A1 da planilha. Atualmente o código está como abaixo:

```

Sub ex_2_12_5_2()

    Dim n1 As Integer, n2 As Integer, media As Double

    n1 = InputBox("Digite o primeiro numero")
    n2 = InputBox("Digite o segundo numero")
    media = (n1 + n2) / 2
    MsgBox "A media dos numeros é : " & media

    Dim soma_n_inteiros As Integer, n As Integer, i As Integer
    n = InputBox("Digite o numero n a ser somado :")
    For i = 0 To n
        soma_n_inteiros = soma_n_inteiros + i
    Next i
    MsgBox "A soma dos numeros é " & soma_n_inteiros

    Sheets("Plan1").Cells(1, 1).Value = "Mensagem qualquer"

End Sub
    
```

Percebemos que existem tarefas que podem ser separadas. Como você reescreveria esse código para ficar mais organizado, usando subs?

3. O que é uma função em VBA? No que ela se diferencia de uma sub?
4. Considere o exercício sobre a compra de metais (ex 3g). Após a definição do preço de compra do aço (pelo departamento de compras), o mesmo é passado para a contabilidade, que consegue realizar o cálculo de deduções de imposto. As quantidades deduzidas de acordo com o preço P são mostradas na Tabela 2.5

Preço (R\$)	Dedução
$0 < P \leq 3500$	2.0% de P + 100
$3500 < P \leq 8500$	2.2% de P + 120
$8500 < P$	2.5% de P + 150

Tabela 2.5: Dedução de custos compra do aço

Ainda, a empresa está inscrita em um projeto governamental que lhe dá acesso a um "selo de contabilidade", de acordo com a quantidade de imposto que ela conseguiu deduzir pela compra

do aço. Os tipos de selo que ela recebe em função das deduções são mostradas na Tabela 2.6:

Dedução	Selo
$0 < I \leq 200$	"Sem selo"
$200 < I \leq 800$	"Selo prata"
$I > 800$	"Selo ouro"

Tabela 2.6: Selo concedido à empresa

Crie um programa que receba a quantidade de aço comprada (em toneladas) e mostre o tipo de selo que a empresa vai ganhar. DICA: Percebemos que o problema pode ser dividido em três partes, sendo que o cálculo de uma etapa é usado no cálculo da próxima, como mostrado na Figura 2.21.

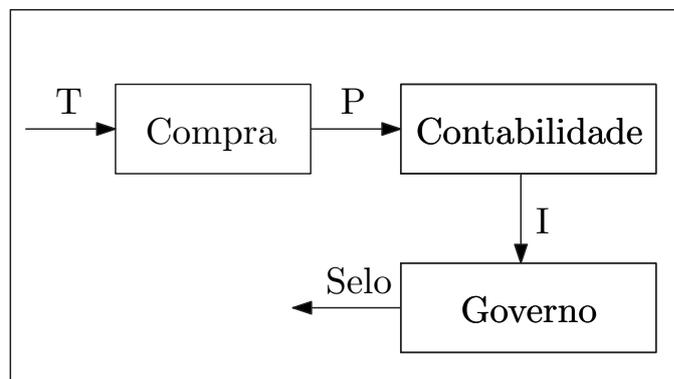


Figura 2.21: Separação do problema em 3 partes

5. Crie funções (uma separada da outra) para as seguintes operações (as funções devem retornar o resultado) da operação:
 - (a) soma(a,b): soma a e b.
 - (b) subtracao(a,b): subtrai b de a.
 - (c) multiplicacao(a, b): multiplica a por b.
 - (d) divisao(a,b): divide a por b.
 - (e) potenciacao(a,b): eleva a a potência de b
 - (f) fat(n): calcule o fatorial de n.
6. Crie um aplicativo que funcione como uma calculadora para o usuário: o programa mostra um 'menu' com as seguintes alternativas para o usuário:
 - (a) A: soma(a,b).
 - (b) B: subtracao(a,b).
 - (c) C: multiplicacao(a, b).
 - (d) D: divisao(a,b)
 - (e) E: potenciacao(a,b).
 - (f) F: fat(n).

O usuário pode digitar qualquer uma das letras do menu, sendo que em seguida os parâmetros para a execução do cálculo devem ser coletados, e o resultado mostrado ao usuário. (use as funções calculadas no exercício anterior). OBS: Para escrever algo em um **MsgBox** ou **Inputbox** e pular uma linha, digite **vbCrLf** separando as strings com aspas duplas. Olhe o exemplo abaixo:

```

Sub msg_pula_linhas ()
  MsgBox "Isso na linha 1" & vbCrLf & "Isso na linha 2" & vbCrLf & "E isso na linha 3"
End Sub
  
```

7. Ainda considerando o aplicativo anterior. Adicione uma opção G: Sair. Assim, após o usuário executar o cálculo, o menu aparece novamente, e somente se ele digitar a letra "G" o programa deve parar de funcionar.

Capítulo 3

VBA Intermédiario

Diversas funcionalidades desta seção só funcionam ao adicionar o *Microsoft Scripting Runtime*. Portanto, antes de continuar, faça como mostra o exemplo: no VBA, abra ferramentas → referências, selecione a opção *Microsoft Scripting Runtime*, como mostrado na Figura 3.1.

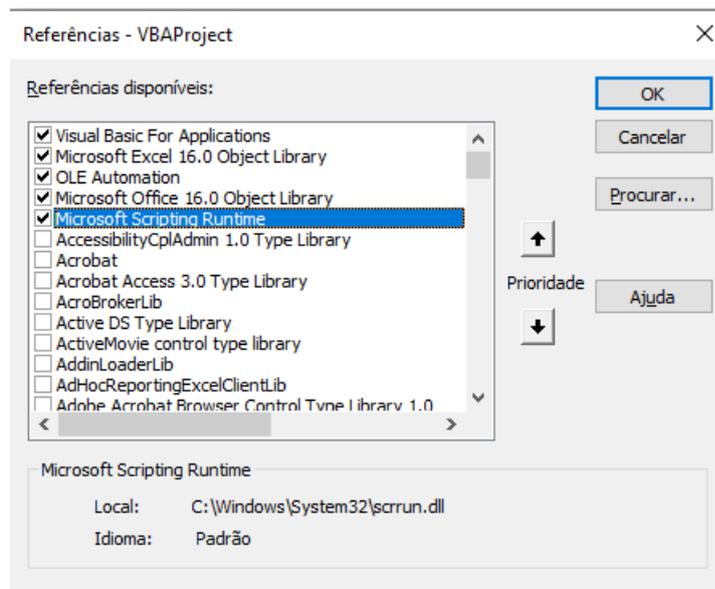


Figura 3.1: Seleção da aba *Microsoft Scripting Runtime*

3.1 Estruturas de dados

3.1.1 Collections

As **collections** são estruturas de dados que podem armazenar diversos elementos, como os vetores, porém não é necessário definir o seu tamanho antes de usar. Existem 4 métodos básicos nas coleções: **Add**, **Index**, **Remove** e **Count**, que serão descritos mais abaixo. Inicialmente veja a declaração de uma lista no código abaixo, que é um pouco diferente dos vetores.

```
Sub collection ()  
  
Dim list As Collection  
Set list = New Collection  
  
End Sub
```

Métodos

Uma diferença entre as collections e os vetores é que a collection começa na índice 1. Para se adicionar elementos usamos o método **Add**. O código abaixo mostra a inserção de 10 elementos na lista:

```
Sub collection()
    Dim list As Collection
    Set list = New Collection

    For i = 0 To 9
        list.Add i
    Next i

End Sub
```

O método **Count** informa o número de elementos da collection, assim, podemos usar **Count** de forma equivalente ao **UBound**. Ainda, para acessarmos os itens usamos o método **Item()** com o índice do elemento. O código abaixo insere 10 elementos e os imprime na janela de verificação imediata.

```
Sub collection()
    Dim list As Collection
    Set list = New Collection
    For i = 0 To 9
        list.Add i
    Next i

    For i = 1 To list.Count 'Note que o índice começa em i
        Debug.Print list.Item(i)
    Next i
End Sub
```

Também é possível remover qualquer elemento da collection com o método **Remove**. O código abaixo usa **Remove()** para remover o último elemento da collection.

```
Sub teste()
    Dim list As Collection
    Set list = New Collection

    For i = 0 To 9
        list.Add i
    Next i
    list.Remove (list.Count)

    For i = 1 To list.Count
        Debug.Print list.Item(i)
    Next i
End Sub
```

Além dos índices, podemos incluir uma 'chave' (string) ao inserir um valor em uma collection, de forma que quando quisermos acessar esse elemento, podemos usar a chave ao invés do índice. Todas as chaves em uma collection devem ser distintas umas das outras, caso contrário um erro ocorrerá. O código abaixo insere dois valores com duas chaves em uma collection, e em seguida imprime esses valores usando as mesmas chaves.

```
Sub collection()
    Dim list As collection
    Set list = New collection

    list.Add 3, "Chave1"
    list.Add 5, "Chave2"

    Debug.Print list.Item("Chave1")
    Debug.Print list.Item("Chave2")

End Sub
```

3.1.2 Dictionaries

Um dicionário é uma coleção que armazena pares no formato chave:valor (adicione o *Microsoft Runtime*, como mostrado nas collections acima). Podemos acessar os valores pelas chaves. Não

podem existir chaves duplicadas em um dicionário, se uma chave for adicionada que já existir, seu valor será sobrescrito pelo novo adicionado. O código abaixo realiza as principais operações em dicionários:

1. Declara
2. Inserir elementos
3. Iterar pelos elementos
4. Verificar se uma chave já existe
5. Remover elementos

```
Sub dicionario()  
  
    Dim v(2) As Integer  
    Dim i As Integer  
  
    ' Declarando o dicionario  
    Dic dic As New Dictionary  
  
    v(0) = 2  
    v(1) = 3  
    v(2) = 4  
  
    ' Adicionando chaves, valores (se inserir em um valor que ja existe vai substituir)  
    dic.Add "vec", v      ' vetor  
    dic.Add "b", 10      ' inteiro  
    dic.Add "c", "Cairo" ' string  
  
    ' Imprimindo o elemento da chave  
    Debug.Print dic("b")  
  
    ' Iterando no elemento vetor da chave 'vec'  
    For i = 0 To UBound(dic("vec"))  
        Debug.Print dic("vec")(i)  
    Next i  
  
    ' Iterando pelos elementos  
    Dim k As Variant  
    For Each k In dic.Keys()  
        Debug.Print k  
    Next k  
  
    ' Verificando se a chave "vec" existe  
    If dic.Exists("vec") Then  
        Debug.Print "Chave vec existe"  
    Else  
        Debug.Print "Chave vec não existe"  
    End If  
  
    ' Removendo um par chave/valor (pela chave)  
    dic.Remove "vec"  
  
    ' Verificando se a chave "vec" existe novamente  
    If dic.Exists("vec") Then  
        Debug.Print "Chave vec existe"  
    Else  
        Debug.Print "Chave vec nao existe"  
    End If  
  
End Sub
```

3.1.3 Type

Podemos usar diversas variáveis no VBA, inteiros, strings, etc... a estrutura Type nos permite criar um tipo de variável nosso. Por exemplo, imagine que você deseja criar uma variável do tipo cliente, em que cada elemento individual de cliente será composto de 3 variáveis; uma string com o nomes, um inteiro com o valor de compra e outra string com o número do RG.

Podemos criar essa nova "variável" como uma Type. O código abaixo cria este tipo de variável (note que ainda não estamos usando a variável, somente criando-a):

```
Option Explicit
Type cliente
    nome As String
    compras As Integer
    RG As String
End Type
```

Atenção: a criação das Types devem ser feitas no início do código, acima de todas as subs.

Com a Type criada, podemos usá-la nas subrotinas como se fosse uma nova variável. O código abaixo declara e preenche uma variável do tipo Type:

```
Sub adiciona_clientes()

    Dim c1 As cliente
    c1.nome = "Alexandre"
    c1.compras = 10
    c1.RG = "47585658214"

End Sub
```

Imagine que queremos salvar os valores de 3 clientes. Também podemos salvar as Types em um vetor, em que cada elemento será um cliente.

```
Sub adiciona_clientes()

    Dim v_clientes(3) As cliente

    Dim c1 As cliente
    c1.nome = "Michael"
    c1.compras = 10
    c1.RG = "47585658214"

    Dim c2 As cliente
    c2.nome = "Dwight"
    c2.compras = 10
    c2.RG = "47585658214"

    Dim c3 As cliente
    c3.nome = "Homer"
    c3.compras = 10
    c3.RG = "47585658214"

    v_clientes(0) = c1
    v_clientes(1) = c2
    v_clientes(2) = c3

End Sub
```

3.2 Input de dados

Podemos fazer a leitura de dados externos (arquivos) usando o *FileSystemObject* do *Microsoft Scripting Runtime*. O código abaixo lê um arquivo de texto com caminho "caminho" linha a linha. A cada iteração do laço while uma linha é lida e armazenada em uma string, e em seguida a string é impressa na janela de verificação.

```
Sub le_arquivo_de_texto()

    Dim fso As Object
    Set fso = CreateObject("Scripting.FileSystemObject")
    Dim arquivo As Object
    Dim caminho As String

    caminho = "C:\Users\Usuario\Documents\Alexandre\Nomes.txt"
    Dim line As String
    Set arquivo = fso.OpenTextFile(caminho, ForReading) 'abre o arquivo

    While Not arquivo.AtEndOfStream ' le linha a linha do arquivo
        line = arquivo.ReadLine ' toda linha e armazenada em uma string 'linha'
        Debug.Print line ' imprimindo a linha lida
    Wend
    arquivo.Close 'fecha o arquivo

End Sub
```

Da forma como está, todo o conteúdo da linha é armazenado na variável `linha`. Na maioria das vezes precisamos fazer mais um "processamento" na linha para obtermos os valores separados. Isso pode ser feito usando a função **Split(s1,sep)**. Essa função recebe dois argumentos, uma string (`s1`) que contém o texto a ser separado, e uma segunda string com um separador (`sep`). O retorno é um vetor de strings, em que cada elemento é uma parte separada da string `s1`.

Por exemplo, se tivéssemos um arquivo com um nome, uma idade e um RG, todos separados por espaços, como mostrado na Figura 3.2:

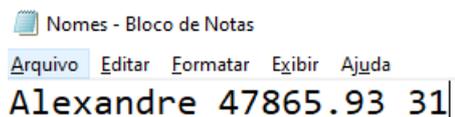


Figura 3.2: Arquivo a ser lido, separado por espaços

E um vetor de quebra de string chamado `v_split`, após a quebra o vetor teria os seguintes elementos:

$$v_split = [\text{"Alexandre"} \quad \text{"47865.93"} \quad \text{"31"}]$$

O código abaixo mostra a leitura das linhas, e a cada leitura os dados são separados por espaço e adicionados ao vetor `v_split`.

```
Sub le_arquivo_de_texto_vetor()

    Dim fso As Object
    Set fso = CreateObject("Scripting.FileSystemObject")
    Dim arquivo As Object
    Dim caminho As String
    Dim v_split() As String

    caminho = "C:\Users\Usuario\Documents\Alexandre\Nomes.txt"
    Dim line As String
    Set arquivo = fso.OpenTextFile(caminho, ForReading) 'abre o arquivo

    While Not arquivo.AtEndOfStream ' le linha a linha do arquivo
        line = arquivo.ReadLine ' toda linha é armazenada em uma string 'linha'
        v_split = Split(line, " ") ' o vetor v_split com os elementos separados da linha
    Wend
    arquivo.Close 'fecha o arquivo

End Sub
```

3.3 Classes

3.3.1 Criação e utilização

Classes são componentes muito importantes de diversas linguagens de programação, e viabilizam a chamada programação orientada à objetos (POO). Não vamos nos aprofundar nas classes a ponto de entendermos toda a filosofia da POO, mas sim as usaremos para criar estruturas de dados mais complexas.

Dessa forma, podemos pensar nas classes como um tipo de *type*, que vimos na seção 3.1.3. O problema das *types* é que não podemos fazer composições com as estruturas dinâmicas, como as listas (*collections*). Imagine que criamos uma *type* com dados de funcionários, nome, rg, idade, etc...e queremos ler dados de diversos funcionários de um arquivo externo, porém não sabemos quantos dados o arquivo contém. Uma solução seria criar uma lista em que cada elemento é uma

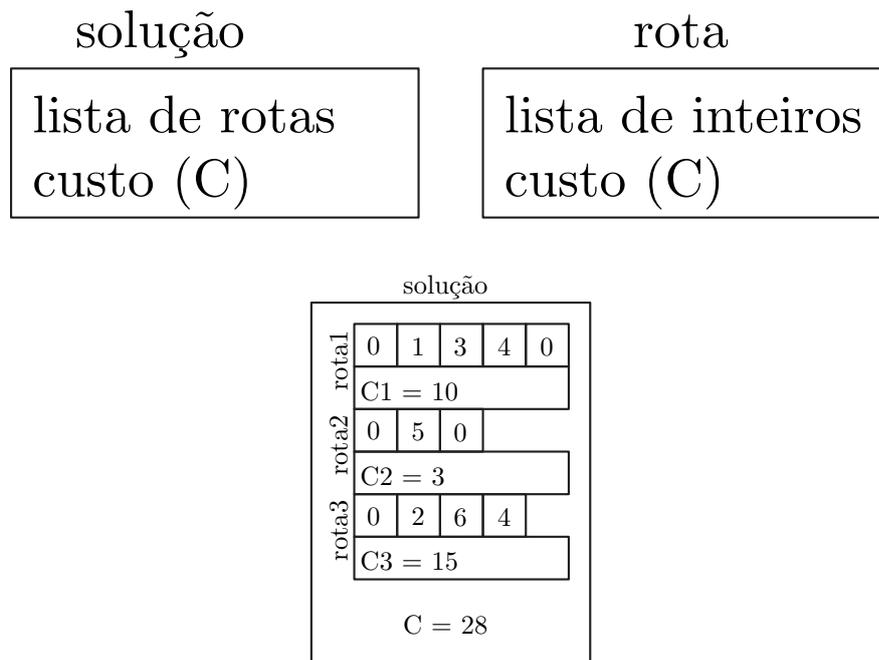


Figura 3.3: Classes para a solução do VRP

type diferente, e vamos adicionando novas *types* na lista a medida que vamos lendo os valores do arquivo de texto. Essa abordagem não é possível, pois não conseguimos alocar uma *type* em uma lista, somente em um vetor.

Uma solução seria **usar classes** no lugar de *types*, sendo que essas podem ser adicionadas em listas.

Vamos implementar a estrutura de dado necessária para resolver o problema da roteirização de veículos. Nesse problema uma solução é composta por diversas rotas que os veículos devem fazer, cada rota tem um custo, e o custo da solução é a soma do custo de todas as rotas. A estrutura pode ser esquematicamente visualizada como na Figura 3.3. Os dois primeiros retângulos (solução e rota) são as classes necessárias.

1. Uma **solução** tem um custo e várias rotas.
2. **Cada rota** tem uma lista de inteiros (dos clientes que são visitados) e o custo dessa rota.

O terceiro retângulo mostra um exemplo da classe preenchida. A solução possui 3 rotas (na lista de rotas), e tem custo total = 28 (10 + 3 + 15). Cada rota possui uma lista que sai e retorna ao ponto zero (depósito), bem como o seu custo.

Para criarmos essa solução, precisamos adicionar dois módulos de classes: com o botão direito no nome do projeto → inserir → módulo de classe. Alterando o nome das classes para rota e solution o vba fica como a Figura 3.4.

Abaixo seguem os códigos que devem ser inseridos nos módulos:

```
' ##### Classe solution #####
Option Explicit

Public custo As Long
Public rotas As New Collection ' lista da classe rota

' ##### Classe rota #####
Option Explicit

Public rota As New Collection ' lista de inteiros
Public custo As Long
```

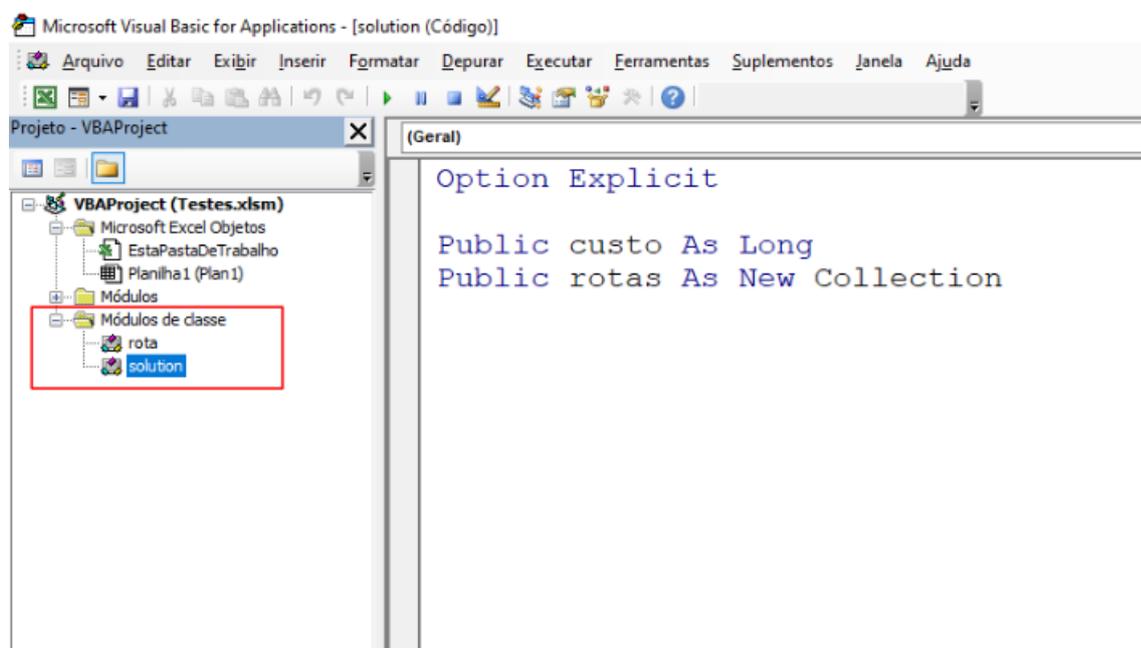


Figura 3.4: Programa com os módulos de classes

Com as duas classes, o código abaixo (que está em um módulo) mostra como podemos criar e alterar uma solução. Note que devemos usar a palavra **New** para instanciar um objeto (como fizemos com as listas e dicionários).

```
Option Explicit

Sub main()
    Dim sol As New solution ' Cria um objeto do tipo solution
    Dim i As Integer

    Dim r As rota ' Cria uma rota (nao usa a palavra new)
    For i = 0 To 2
        Set r = New rota ' Cria uma NOVA rota a cada iteracao, por isso "Set"
        r.rotas.Add 0
        r.rotas.Add 4
        r.rotas.Add 2
        r.rotas.Add 0
        r.custo = 100 ' Adiciona um custo para a rota
        sol.rotas.Add r ' Adiciona a rota a lista "rotas"
    Next i
    sol.custo = 28 ' Adiciona um custo para a solucao
    altera_rotas sol ' Passa a solucao para ser alterada em um sub
End Sub

Sub altera_rotas(ByRef sol As solution)
    sol.rotas(2).rotas.Add 16
End Sub
```

Como temos 2 classes, uma dentro da outra, precisamos criar o objeto `sol` como `solution`, e também os objetos `rotas` que estarão na lista de rotas dentro da `solution`. O código acima cria uma solução e adiciona 3 rotas à ela, cada uma com os valores (0-4-2-0), em seguida chama uma sub e altera somente a segunda rota, adicionando um elemento = 16. Ao fim do processo a estrutura fica semelhante à Figura 3.5.

3.3.2 O problema da cópia

Como sabemos existe uma diferença entre variáveis `ByRef` e `ByVal` no VBA. As cópias e criações de novos objetos do tipo classe mantêm o comportamento por padrão `ByRef`. Isso pode afetar a nossa

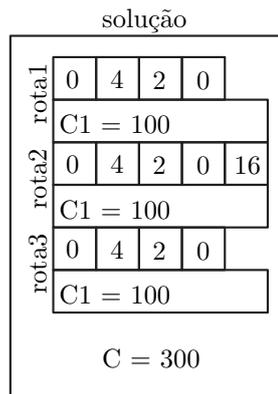


Figura 3.5: Solução criada no exemplo

utilização quando precisamos criar diversos objetos de uma classe por um laço for por exemplo.

Considere o último código apresentado, criamos 2 rotas (que são classes), usando o método for, porém repare que não usamos a declaração padrão (variável as `New` classe). Considere o código abaixo, que criaria todas as rotas da forma usual, sem usar o `Set`.

```
Option Explicit

Sub main()
    Dim sol As New solution ' Cria um objeto do tipo solution
    Dim i As Integer
    For i = 0 To 2
        Dim r As New rota ' Cria uma rota sem usar o Set
        r.rota.Add 0
        r.rota.Add 4
        r.rota.Add 2
        r.rota.Add 0
        r.custo = 100 ' A
        sol.rotas.Add r ' Adiciona a rota a lista "rotas"
    Next i
    sol.custo = 300
    altera_rota sol ' Passa a solucao para ser alterada em um sub
End Sub
```

O código acima cria as rotas mostradas na Figura 3.6

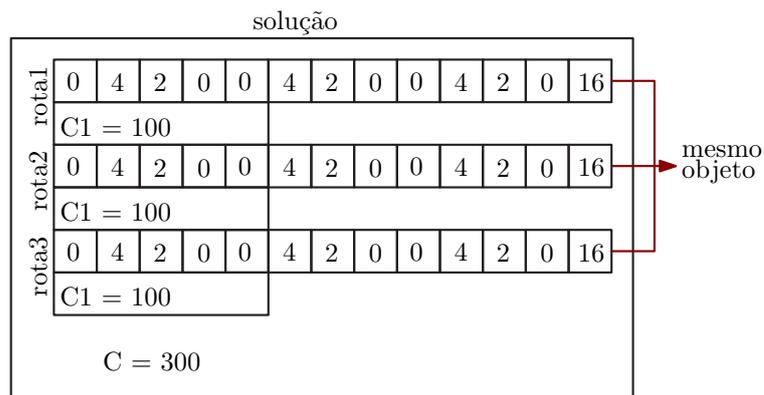


Figura 3.6: Solução criada no exemplo sem usar o set

Observe que embora esteja em um laço for, a variável `r` criada na primeira iteração **é a mesma que as outras duas**, pois ela é criada `ByRef`. Dessa forma toda vez que adicionamos um valor na lista, esse valor é adicionado na lista de todas as rotas criadas (que na verdade são a mesma). Para

evitar esse problema use o `Set`, ele garante que um novo objeto e uma nova referência estão sendo criados.

3.3.3 O problema da cópia/atribuição

O mesmo problema da criação ByRef acontece na atribuição, ou seja, quando queremos que 2 objetos sejam iguais. Quando lidamos com inteiros, as atribuições são feitas ByVal, ou seja, uma cópia. Considere o código abaixo:

```
Sub atribuicao()  
  
    Dim a As Integer, b As Integer  
    a = 10  
    b = a  
    b = 11  
    Debug.Print a, b  
  
End Sub
```

Como esperado, a atribuição `b = a` não quer dizer `a` e `b` são a mesma coisa, mas sim: `b` recebe uma cópia do conteúdo de `a`. O mesmo não ocorre com classes (tudo que precisa da palavra `New`). A própria atribuição (usar o sinal de `=`) não é permitida para as classes. Considere o código abaixo com 2 listas:

```
Sub atribuicao()  
  
    Dim l1 As New Collection  
    Dim l2 As New Collection  
  
    l1 = l2 'ERRO, nao e permitido  
  
End Sub
```

Não é possível fazer a atribuição dessa forma. Podemos usar o `Set` da seguinte forma:

```
Sub atribuicao()  
  
    Dim l1 As New Collection '  
    Dim l2 As Collection  
  
    l1.Add 3  
    Set l1 = l2  
  
End Sub
```

Porém note o comportamento de `l1` quando adicionamos 2 elementos em `l2`:

```
Sub atribuicao()  
  
    Dim l1 As New Collection '  
    Dim l2 As New Collection  
  
    l1.Add 3  
    Set l2 = l1  
  
    l2.Add 5  
    l2.Add 6  
  
    Dim i As Integer  
    For i = 1 To l1.Count  
        Debug.Print l1(i)  
    Next i  
  
End Sub
```

Ambas as listas possuem os valores (3,5,6), ou seja, **novamente estamos tratando do mesmo objeto**. O que podemos fazer para de fato criarmos uma cópia da lista `l1` em `l2`? VBA não têm uma operação de atribuição (`=`) padrão para classes, de forma que temos que criar nosso próprio método de cópia.

3.3.4 Criando cópias

O código abaixo cria um método para copiar listas. A sub adiciona todos os elementos da `l1` em `l2`.

```

Sub atribuicao()
    Dim l1 As New Collection
    Dim l2 As New Collection
    Dim i As Integer
    l1.Add 3

    copia_lista l2, l1 'faz a copia de l1 em l2
    l2.Add 5 ' adiciona um elemento em l2 que nao esta em l1
    Debug.Print "-----l2-----"
    For i = 1 To l2.Count
        Debug.Print l2(i)
    Next i
    Debug.Print "-----l1-----"
    For i = 1 To l1.Count
        Debug.Print l1(i)
    Next i
End Sub

Sub copia_lista(ByRef l_copia As Collection, ByRef l_original As Collection)
    Dim i As Integer
    For i = 1 To l_original.Count
        l_copia.Add (l_original(i))
    Next i
End Sub

```

Dessa forma temos que $l2 = (3,5)$ e $l1 = (5)$, como esperado. Se queremos copiar as nossas próprias classes, devemos usar a mesma abordagem: **copiar elemento a elemento de todas as estruturas dentro das mesmas**. Para isso podemos criar subs e funções dentro das próprias classes (chamados de métodos).

3.3.5 Criando métodos de cópia

Considere novamente as classes solution e rota do início da seção. Os códigos abaixo atualizam as classes com a sub Copy, que realiza a cópia elemento a elemento de um argumento e adiciona nas variáveis internas.

```

' ----- CLASSE ROTA -----
Option Explicit
Public rota As New Collection
Public custo As Long

Sub Copy(ByRef r1 As rota)

    custo = r1.custo
    Dim i As Integer
    For i = 1 To r1.rota.Count
        rota.Add (r1.rota(i))
    Next i

End Sub

```

```

'-----CLASSE SOLUTION-----
Option Explicit

Public custo As Long
Public rotas As New Collection

Public Sub Copy(ByRef c1 As solution)
    ' Copia os valores de c1 nas variaveis internas (custo e rotas)
    custo = c1.custo
    ' Copiando as rotas pelo próprio método de copia das rotas
    Dim i As Integer
    For i = 1 To c1.rotas.Count
        Dim r1 As rota
        Set r1 = New rota
        r1.Copy c1.rotas(i)
        Debug.Print c1.rotas(i).custo
        rotas.Add r1
    Next i
End Sub

```

E o código abaixo cria uma solução (s1) com uma rota, e em seguida faz uma cópia de sol em best_sol, adicionando várias rotas em best_sol.

```
Sub copia_solucão()  
  
    ' Dimensionando a solução 1  
    Dim s1 As New solution  
    Dim r1 As New rota  
    r1.custo = 10  
    r1.r.Add 3  
    r1.r.Add 4  
    s1.custo = 300  
    s1.rotas.Add r1  
  
    ' Copiando s1 em best_sol  
    Dim s_best As New solution  
    s_best.Copy s1  
  
    Dim i As Integer, j As Integer  
    For i = 4 To 10  
        Dim nova_rota As Collection  
        Set nova_rota = New Collection  
        For j = 0 To 5  
            nova_rota.Add i  
        Next  
        s_best.rotas.Add nova_rota  
    Next  
End Sub
```


Capítulo 4

Trabalhos

Nesta seção são apresentados alguns problemas mais complexos que podem ser resolvidos por meio da computação.

4.1 Cálculo/Algebra

4.1.1 Cálculo de integrais

A integral de uma função de uma variável, representa a área da curva abaixo do gráfico da função, como apresentado na Figura 4.1.

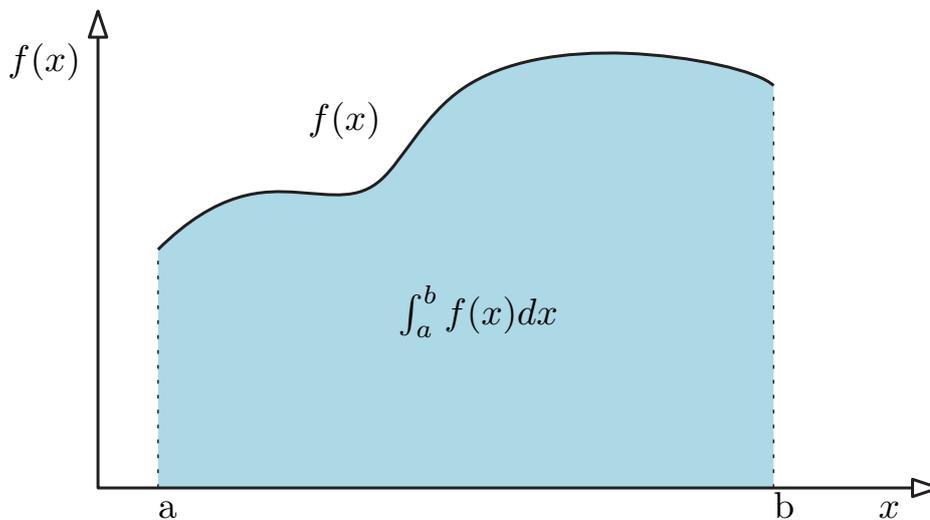


Figura 4.1: Integral de f

Um forma de se estimar o valor de integrais numericamente é pelo método da soma dos trapézios. Como mostrado na Figura 4.2.

Exercício: Desenvolva um código que calcule a integral de uma função genérica, o cálculo deve receber 3 parametros:

a: Limite inferior de integração

b: Limite superior de integração

n: Número de trapézios utilizados para a aproximação

Estime o valor da integral para três funções diferentes, e para cada uma, use 20 valores de *n* diferentes. Apresente as funções, os valores da integral e o número *n* utilizado.

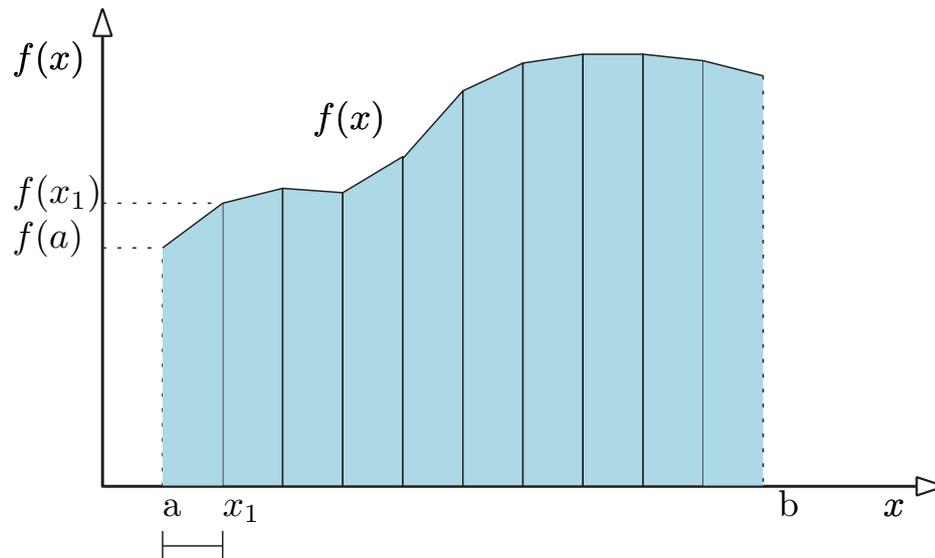


Figura 4.2: Integral de f

4.1.2 Cálculo de derivadas

A derivada de uma função qualquer em um ponto determina a inclinação da reta tangente a este ponto. A derivada $f'(x_0)$ de uma função $f(x)$ no ponto x_0 é:

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h} \quad (4.1)$$

Da definição, se $h \neq 0$ é pequeno (não muito pequeno para evitar o cancelamento catastrófico), é esperado que uma aproximação para a derivada no ponto x_0 seja dada por:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \quad (4.2)$$

A interpretação geométrica da derivada é mostrada na Figura 4.3

Exercício: Desenvolva um código que calcule a derivada de uma função genérica, o cálculo deve receber 2 parâmetros:

x_0 : Ponto em que a derivada será calculada

h : Distância utilizada para a aproximação

Estime o valor da derivada para três funções diferentes, e para cada uma, use 20 valores de h diferentes. Apresente as funções, os valores das derivadas, o erro de estimação e o número h utilizado.

4.1.3 Cálculo de raízes de funções

A raiz de uma função $f(x)$, representa os valores da equação que deixam a equação na forma $f(x) = 0$. O método mais básico para se encontrar uma raiz de uma função $f(x)$ é chamado método da bisseção (outros métodos com melhor convergência também existem, como o método de Newton-Raphson). O método da bisseção parte da premissa de que, se existe uma raiz em um dado intervalo $[a, b]$ no domínio da função, a função em $f(a)$ e em $f(b)$ devem ter sinais opostos (uma positiva e outra negativa). A partir desta premissa, um intervalo inicial deve ser escolhido de forma a obter os valores da função nos extremos, com sinais opostos.

A partir disso, o método divide o intervalo na metade e calcula o valor da função neste ponto, se o sinal da função neste ponto for positivo, o valor do intervalo inicial $[a, b]$ que for positivo é substituído, caso contrário o valor negativo é substituído. Esse processo é repetido até o momento em que o número de iterações máximo seja atingido, ou uma precisão da raiz seja encontrada.

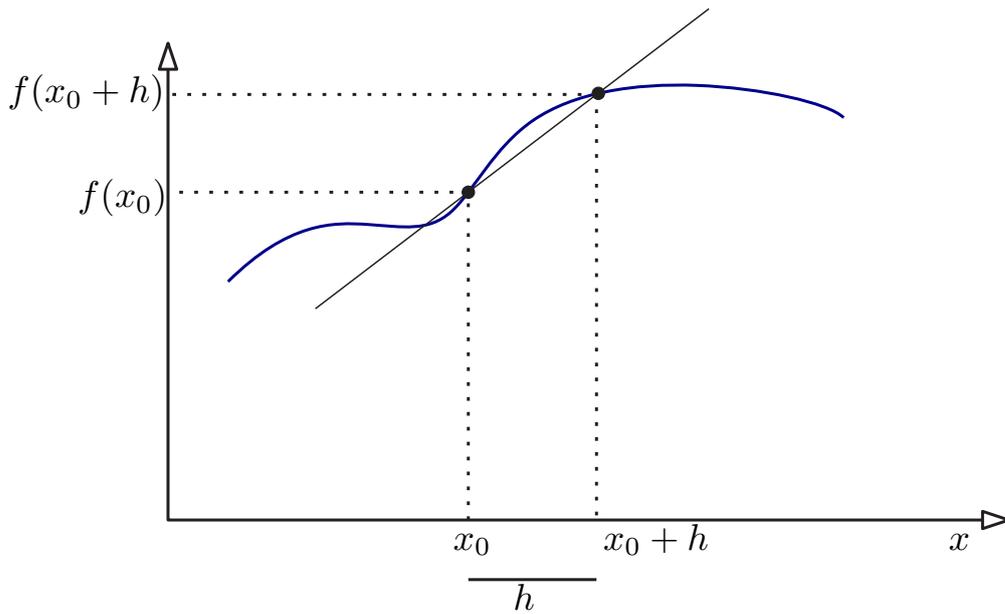


Figura 4.3: Derivada de f

A Figura 4.4 exemplifica o método, e como os intervalos vão diminuindo a cada iteração, se aproximando cada vez mais da raiz da função.

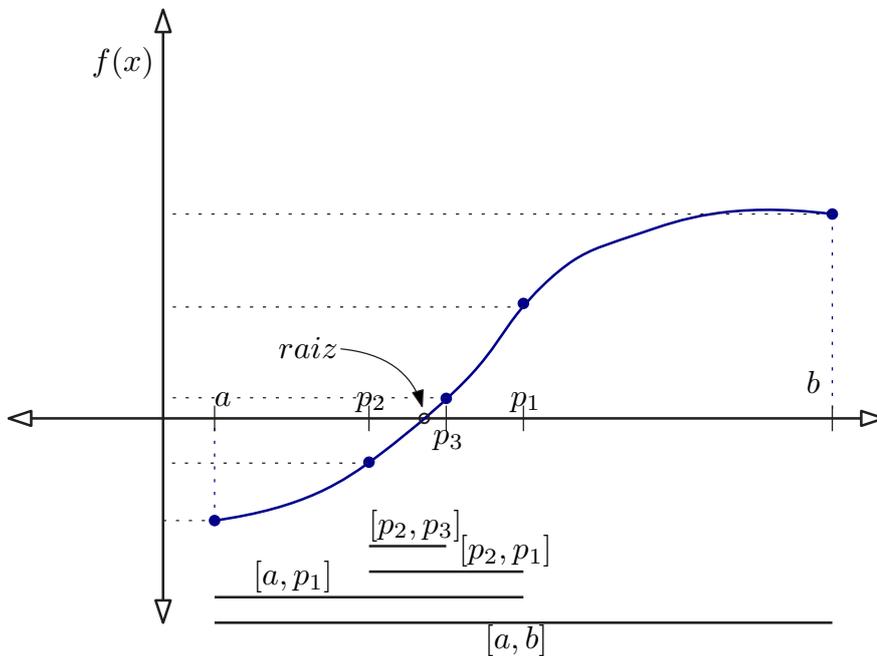


Figura 4.4: Raiz de f

Exercício: Desenvolva um código que calcule a raiz de uma função qualquer pelo método da bissecção. O método recebe 1 parâmetro como entrada, *MaxIter*: o número total de iterações do algoritmo. Execute o código para 3 funções diferentes, com 20 valores diferente de *MaxIter* cada uma. Compare os valores estimados e os valores exatos das raízes

4.1.4 Mutiplicação de matrizes

Dadas duas matrizes $M1_{m1 \times n1}$ e $M2_{m2 \times n2}$, realiza a multiplicação das duas, resultando em uma matriz $M3_{m1 \times n2}$. Para que a multiplicação seja possível, a condição $n1 = m2$ deve ser satisfeita, e a matriz resultante terá $m1$ linhas e $n2$ colunas. O algoritmo abaixo ilustra o procedimento para multiplicação de matrizes.

Data: Matrizes $M1_{m1 \times n1}$ e $M2_{m2 \times n2}$
Result: Matriz $M1_{m1 \times n2}$ resultado da multiplicação $M1 * M2$

```

1 M3 ← Inicialize M3 com m1 linhas e n2 colunas. for toda l = 0 até l < M1.size() do
2   for toda c = 0 até c < M2[0].size() do
3     for toda k = 0 até k < M2[0].size() do
4       M3[l][c] ← M3[l][c] + M1[l][k] * M2[k][c]
5     end
6   end
7 end

```

Algorithm 1: Multiplica Matriz

4.1.5 Matriz inversa

4.1.6 Resolução de sistemas lineares

4.2 Eng. de produção

4.2.1 Cálculo do BOM (Bill of Materials)

O cálculo do BOM está associado a explosão dos componentes de um produto em forma de árvore. Seja a árvore de produto seguir (4.5), que demonstra as relações de precedência entre todos os componentes (dentro dos círculos o número do item e ao lado a quantidade necessária).

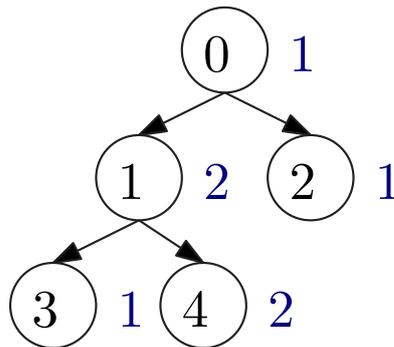


Figura 4.5: Árvore de produto

O problema de calcular o BOM é: Dada uma demanda do produto 0, quanto deverá ser produzido (ou comprado) para a produção de 0? No caso da árvore da Figura 4.5, para uma demanda de 2 unidades de 0, são necessárias as seguintes quantidades de 1,2,3 e 4:

- 1: 4
- 2: 2
- 3: 4
- 4: 8

As informações de uma árvore de produtos pode ser representada por uma matriz de incidências, a matriz abaixo representa a árvore da Figura 4.5.

$$AP_{p \times p} = \begin{bmatrix} 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Data: Matriz árvore de incidencia $AP_{p \times p}$, componente n_c , demanda d_c do componente
Result: Vetor BOM com quantidades dos componentes

```

1  $l = \emptyset$  // lista com os produtos
2  $d = \emptyset$  // lista com os fatores multiplicadores
3  $BOM \leftarrow$  inicialize vetor de tamanho  $p$ , com elementos 0
4 Adicionar  $p$  em  $l$ 
5 Adicionar  $d_c$  em  $d$ 
6 while  $l \neq \emptyset$  do
7    $NoAtual \leftarrow l[0]$ 
8    $FMult \leftarrow d[0]$ 
9   Remova  $l[0]$  de  $l$ 
10  Remova  $d[0]$  de  $d$ 
11  for toda coluna  $j \in AP$  do
12    if  $AP[NoAtual][j] > 0$  then
13      Adicionar  $j$  em  $l$ 
14      Adicionar  $FMult * AP[NoAtual][j]$  em  $d$ 
15    end
16  end
17   $BOM[NoAtual] += FMult$ 
18 end

```

Algorithm 2: Cálculo BOM

4.2.2 Agrupamento de máquinas (Algoritmo ROC)

O algoritmo ROC (Rank Order Clustering) é utilizado como uma técnica na área de tecnologia de grupos, para agrupar peças à máquinas, de forma a definir células de manufatura. A entrada para o algoritmo é uma matriz $M_{p \times m}$, em que p são os produtos da fábrica, e m as máquinas. A matriz é binária, de forma que:

$M_{ij} = 1$, se o produto i é processado pelo máquina j , e 0 caso contrário. A matriz a seguir exemplifica o conceito:

$$\begin{matrix} & M_0 & M_1 & M_2 & M_3 & M_4 & M_5 & M_6 \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} & P_0 \\ & & & & & & & P_1 \\ & & & & & & & P_2 \\ & & & & & & & P_3 \\ & & & & & & & P_4 \\ & & & & & & & P_5 \end{matrix}$$

No caso da matriz acima, o produto P_0 (linha 0 da matriz) é processado pelas máquinas M_0 , M_3 e M_5 (colunas). O algoritmo ROC opera nas linhas e colunas da matriz, de forma a agrupar os itens que são processados em máquinas semelhantes, desta forma criando as famílias de produtos e células de máquinas. Ao final, uma matriz da seguinte forma é gerada:

$$\begin{array}{cccccc}
 & M_3 & M_4 & M_0 & M_1 & M_2 & M_5 & M_6 \\
 \left(\begin{array}{cccccc}
 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 1 & 0 & 1
 \end{array} \right) & \begin{array}{l} P_1 \\ P_3 \\ P_2 \\ P_4 \\ P_6 \\ P_5 \end{array}
 \end{array}$$

Com a nova matriz, percebemos que os produtos P_1 e P_3 formam uma família, que são processados na célula de manufatura composta pelas máquinas M_3, M_4 e M_0 , e os produtos P_2, P_4, P_6 e P_5 formam outra família, que é processada na célula composta pelas máquinas M_1, M_2, M_5 e M_6 .

4.3 Pesquisa Operacional/ Otimização

4.3.1 O algoritmo Simplex

4.3.2 O problema do caixeiro viajante - (*TSP - Traveling Salesman Problem*)

O problema do caixeiro viajante (PCV) consiste em, dado um conjunto de pontos (cidades), determinar uma rota que:

1. Parte de uma cidade de origem
2. Passe por todas as cidades uma única vez
3. Retorne a cidade de origem

As Figuras 4.6 e 4.7 mostram uma solução factível e uma infactível para o problema do caixeiro viajante.

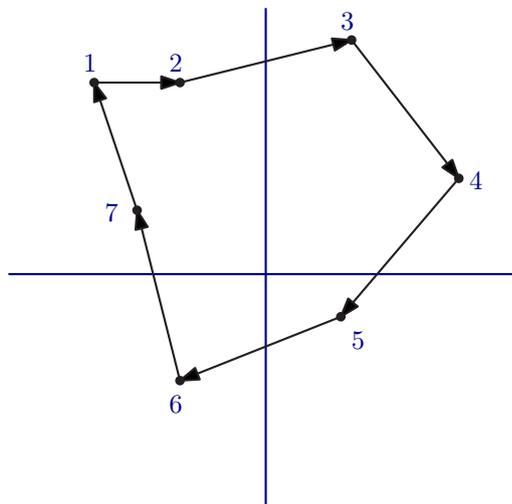


Figura 4.6: Solução para o caixeiro viajante

Existem diversas soluções factíveis para um mesmo problema, como mostrado na Figura

Como existem diversas soluções, existem diversos algoritmos que resolvem o problema. Abaixo segue um algoritmo guloso para a solução do PCV, chamado "Vizinho mais próximo".

Exercício: Implementar o algoritmo do vizinho mais próximo, lendo os pontos de um arquivo .txt (com número genérico de pontos), exportar a rota resultante juntamente com o custo total associado a mesma.

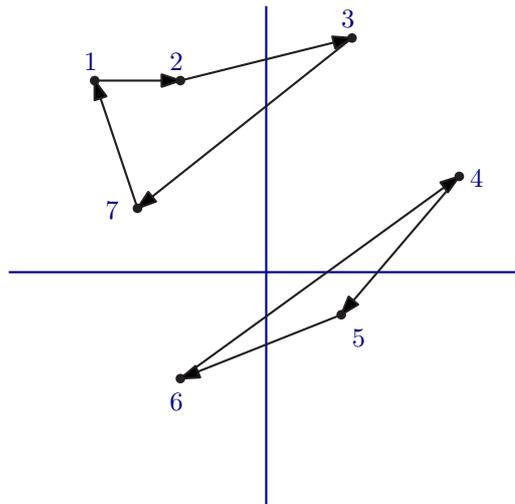


Figura 4.7: Solução infactível para o caixeiro-viajante

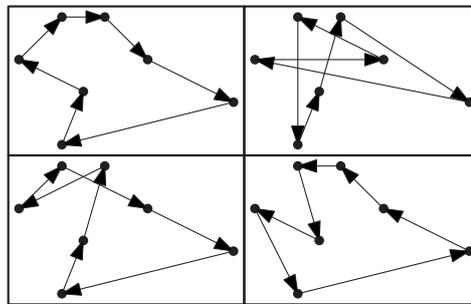


Figura 4.8: Rotas factíveis para o mesmo PCV

```

Data: Lista  $P$  de todos os pontos
Result: Rota  $R$ , Custo  $C$  da rota
1  $M_d \leftarrow$  CalcularMatrizDistancias( $P$ )
2 Adicionar um ponto  $p_0 \in P$  na rota  $R$ 
3 Remover  $p_0$  de  $P$ 
4 while  $P \neq \emptyset$  do
5    $p_u \leftarrow$  ultimo elemento de  $R$ 
6    $MinDist \leftarrow \infty$ 
7    $PontoMinDist \leftarrow 0$ 
8   for todo  $p_i \in P$  do
9      $D_{p_u p_i} \leftarrow M_d[p_u][p_i]$ 
10    if  $D_{p_u p_i} < MinDist$  then
11       $MinDist \leftarrow D_{p_u p_i}$ 
12       $PontoMinDist \leftarrow p_i$ 
13    end
14  end
15  Adicionar  $PontoMinDist$  em  $R$ 
16  Remover  $PontoMinDist$  de  $P$ 
17 end
18  $C \leftarrow$  CalculaCustoRota( $R$ )

```

Algorithm 3: Vizinho mais próximo

4.3.3 O problema do roteamento de veículos - (*VRP - Vehicle Routing Problem*)

4.3.4 O problema da mochila - *Bin Packing Problem*

4.3.5 O problema das p-medianas (*k-means*)

4.4 Miscelânea

4.4.1 Ordenação Bubble Sort

O algoritmo "*Bubble Sort*" é um algoritmo de ordenação de vetor, abaixo seu pseudocódigo:

```

Data: Vetor  $V$  de elementos a serem ordenados
Result: Vetor  $V$  de elementos ordenados
1 for todo  $i = 0$  até  $i < V.size()$  do
2   for todo  $j = 0$  até  $j < V.size() - 1 - i$  do
3     if  $V[j + 1] > v[j]$  then
4        $Aux \leftarrow V[j + 1]$ 
5        $V[j + 1] = V[j]$ 
6        $V[j] = Aux$ 
7     end
8   end
9 end

```

Algorithm 4: Bubble Sort

O algoritmo abaixo implementa um bubble sort em um vetor v .

```

For j = 0 To UBound(v) - 1
  For i = 0 To UBound(v) - 1 - j
    Debug.Print v(i)
    If v(i) < v(i + 1) Then
      Dim aux As Integer
      aux = v(i)
      v(i) = v(i + 1)
      v(i + 1) = aux
    End If
  Next i
Next j

```

Bibliografia

- [Cohen, 1995] Cohen, W. W. (1995). Fast effective rule induction. In *Machine Learning Proceedings 1995*, pages 115–123. Elsevier.
- [Cormen et al., 2001] Cormen, T., Leiserson, C. E., and Rivest, R. L. (2001). *Algoritmos: Teoria e Pratica*.
- [George and Robinson, 1980] George, J. A. and Robinson, D. F. (1980). A heuristic for packing boxes into a container. *Computers & Operations Research*, 7(3):147–156.
- [Henn and Wäscher, 2012] Henn, S. and Wäscher, G. (2012). Tabu search heuristics for the order batching problem in manual order picking systems. *European Journal of Operational Research*, 222(3):484–494.
- [Nagata, 2007] Nagata, Y. (2007). Edge assembly crossover for the capacitated vehicle routing problem. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 142–153. Springer.
- [Pongcharoen et al., 2004] Pongcharoen, P., Hicks, C., and Braiden, P. (2004). The development of genetic algorithms for the finite capacity scheduling of complex products, with multiple levels of product structure. *European Journal of Operational Research*, 152(1):215–225.
- [Tuzun and Burke, 1999] Tuzun, D. and Burke, L. I. (1999). A two-phase tabu search approach to the location routing problem. *European journal of operational research*, 116(1):87–99.