

Algoritmos

Márcio Machado Pereira
Paulo César Rodacki Gomes

9 de outubro de 2020

Sumário

Lista de Figuras	ix
Lista de Quadros	xii
Prefácio	xvii
I Programação Python	1
1 Algoritmos e Programação de Computadores	3
1.1 Computação	3
1.2 Duas idéias fundamentais de ciência da computação	3
1.3 Algoritmos	4
1.3.1 Exemplo: problema de ordenação	6
1.3.2 A escolha de um algoritmo	6
1.4 Processamento de Informações	7
1.4.1 Exercício	7
1.5 Linguagens de Programação	8
1.5.1 Expressando algoritmos em linguagens de programação	9
1.5.2 Exemplo do algoritmo “Raiz Quadrada” em Python	9
1.6 Programa de Estudos	10
1.7 Porque aprender a programar (alguns exemplos)	11
1.8 O que será necessário	11
1.9 Extras	12
2 Computação e Computadores - Breve Histórico	13
2.1 A Estrutura de um Computador Moderno	15
2.1.1 Hardware	15
2.1.2 Software	16
3 A Linguagem de Programação Python	19
3.1 Programas	19
3.2 Objetos e tipos de dados	19
3.3 Expressões	20
3.4 Variáveis e Valores	20
3.5 Abstração de Expressões	21
3.5.1 Exercícios	22

4 Primeiros passos em Python	23
4.1 Imprimindo no console	23
4.2 Variáveis e Tipos	24
4.3 Operadores	25
4.3.1 Operadores Aritméticos	25
4.3.2 Precedência de Operadores	26
4.3.3 Exercícios	26
4.3.4 Operadores que atuam sob Strings	27
4.3.5 Comparações	27
4.3.6 Operadores Lógicos	29
4.4 Conversões de tipo (type cast)	29
4.4.1 Exercício:	30
4.5 Entrada de dados via teclado	30
4.6 Strings literais com formatação	30
4.7 A função format	31
5 Controle de Fluxo - Condicionais	33
5.1 Indentação do Código	33
5.2 Sentenças Condicionais	34
5.2.1 Exercícios:	35
5.3 Expressões condicionais	36
5.3.1 Valores True e False	36
5.3.2 Exemplos	37
5.3.3 Exercícios	38
6 Controle de Fluxo - Iterações	41
6.1 Laços while	41
6.1.1 Exercícios - while	42
6.2 Laços for	43
6.2.1 A função range	43
6.2.2 Exercícios - for	45
6.3 Escolhendo entre for e while	45
6.3.1 Exercícios	45
6.4 Variáveis Indicadoras	45
6.4.1 Exercício	46
6.5 Variáveis Contadoras	46
6.5.1 Exercícios	46
7 Números Binários, Laços Encaixados	49
7.1 Representações Binárias e Decimais	49
7.1.1 Conversão Binário-Decimal	49
7.1.2 Conversão Decimal-Binária	50
7.1.3 Exercício	51
7.2 Laços Encaixados	51
7.2.1 Equações Lineares	51
7.2.2 Mega-Sena	52
7.2.3 Exercícios	52

8 Introdução a Funções	55
8.1 Como escrever código?	55
8.2 Exercício de motivação	55
8.3 Decomposição e Abstração	56
8.4 Projetando funções	57
8.5 Documentação	58
8.6 Exercício: Fatorial	58
8.7 Exercício: Combinação	59
8.8 Escopo de Variáveis	59
8.9 Exercícios	62
8.10 Funções como argumentos	62
8.11 Funções Aninhadas e Funções como valores de retorno	64
8.12 Estudo de caso: algoritmo de Aproximações de Newton	65
8.13 Funções anônimas: lambda	66
9 Strings	71
9.1 Seqüências	71
9.2 Strings em Python	71
9.2.1 Comprimento de Strings	72
9.2.2 Acessando elementos de uma String	72
9.2.3 Slices de strings	73
9.2.4 Operador in	73
9.2.5 Método strip	73
9.2.6 Método find	74
9.2.7 Método startswith	74
9.2.8 Método replace	74
9.2.9 Strings e laços	74
9.2.10 Exercícios	75
10 Tuplas	77
10.1 Acessando Valores em tuplas	77
10.2 Contatenação de tuplas	78
10.3 Eliminando elementos de uma tupla	78
10.4 Operações básicas em tuplas	79
10.5 Atribuições com tuplas	79
10.6 Tuplas no retorno de funções	80
10.7 Tuplas como argumentos de comprimento variável de funções	80
11 Listas	83
11.1 Operações básicas: Pertinência, Concatenação e Repetição	84
11.2 Métodos de Listas	84
11.3 Listas e Strings	85
11.4 Referências de Listas	87
11.4.1 Alias e clone de Listas	88
11.4.2 Listas como Parâmetros	91
11.5 Processamento de Seqüências	93
11.5.1 Compreensão de Listas	93
11.5.2 Compreensão de Listas Aninhadas	95

11.5.3 Exercícios:	95
12 Arrays multidimensionais (Matrizes)	97
12.1 Biblioteca Numpy	98
12.2 Criação de Matrizes	99
12.3 Operações Básicas	100
12.4 Exercícios:	101
13 Conjuntos e Dicionários	103
13.1 Conjuntos	103
13.2 Dicionários	105
13.2.1 Motivação	105
13.2.2 Exemplo	106
13.2.3 Método <code>get</code>	107
13.2.4 A Função <code>dict</code>	107
13.3 Chaves e Conteúdos (valores) de Dicionários	108
13.3.1 Métodos <code>items</code> , <code>keys</code> e <code>values</code>	108
13.3.2 Método <code>copy</code>	108
13.3.3 Método <code>clear</code>	109
13.4 Compreensão de dicionários	109
13.5 Iterando em dicionários	109
13.5.1 Exercício: Contando letras em um String	110
13.6 Ordenando dicionários por chave e valor	110
13.7 Ordenação Customizada	110
13.7.1 Exercícios	110
14 Arquivos	113
14.1 Arquivos Texto	113
14.1.1 Escrevendo em um arquivo texto	114
14.1.2 Escrevendo números em um arquivo texto	114
14.1.3 Lendo dados de um arquivo texto	115
14.1.4 Lendo números de um arquivo texto	116
14.2 Tratamento de exceções	117
14.2.1 Tipos de Erros mais comuns	117
14.2.2 Tratando uma exceção	118
14.3 Editando Arquivos texto	119
14.3.1 Exemplo	119
14.4 Parâmetros do Programa	120
14.4.1 Exemplos	121
14.5 Arquivos Binários	123
14.5.1 Abrindo Arquivos Binários	124
14.5.2 Lendo e Escrevendo em Arquivos Binários	124
14.5.3 Exemplo	125
14.6 Acessando e manipulando arquivos e diretórios em disco	127
15 Pacotes e Módulos	131
15.1 Reuso de Código	131
15.1.1 Pacotes e Módulos	131

15.1.2 Um pouco sobre pacotes	132
15.1.3 Localizando Módulos	132
15.2 Módulos personalizados	132
15.2.1 O módulo vector.py	133
15.3 A Classe Vector	138
15.4 Definições de Métodos	138
15.4.1 O Método <code>__init__</code> :	138
15.4.2 O Método <code>__str__</code> :	138
15.4.3 Outros métodos especiais:	139
15.4.4 Exercícios	141
16 Tópicos complementares de Python	143
16.1 Iteradores e Geradores	143
16.1.1 Geradores	144
16.1.2 Iteradores	145
16.2 Polinômios como dicionários	147
16.2.1 Avaliação de Polinômios	148
16.3 Plotando funções	148
16.3.1 Matplotlib	149
16.3.2 Decorando o gráfico	150
16.3.3 Plotando múltiplas curvas	151
17 Expressões Regulares	153
17.1 Expressões Regulares	153
17.1.1 Método <code>search</code>	153
17.2 Objetos do tipo match	154
17.3 Outras Funções da biblioteca RE	154
17.4 Compilando REs	155
17.5 Regras básicas para escrita de uma RE	155
17.5.1 Backslash	155
17.5.2 Repetições	156
17.6 Classes de caracteres	156
17.7 Opcional	157
17.8 Outros Meta-Caracteres	158
17.9 Exemplo - Buscando um email	158
17.10 Exercícios	158
17.11 Referência	159
II Projeto de algoritmos	161
18 Ordenação	163
18.1 Selection sort	165
18.1.1 Exercício	166
18.2 Bubble sort	167
18.2.1 Exercício	167
18.3 Insertion sort	167
18.4 Exercícios	168

19 Algoritmos de Busca	171
19.1 Busca pelo Mínimo	171
19.2 Busca Linear ou Sequencial	172
19.2.1 Exercícios	172
19.3 Busca Binária em uma Lista	173
19.4 Versão recursiva da busca binária	174
19.4.1 Exercícios de Busca Binária	175
20 Recursividade	177
20.1 Função Fatorial	177
20.2 Torres de Hanoi	178
20.3 Palíndromo	180
20.4 Números de Fibonacci	181
20.4.1 Algoritmo Recursivo	181
20.4.2 Recursão com memorização	182
20.5 Recursão com Backtracking (Retrocesso)	183
20.6 O Problema das oito Damas do Xadrez	185
20.6.1 Exercício	186
21 Divisão e Conquista	187
21.1 Quick Sort	187
21.1.1 Análise de Complexidade	188
21.1.2 Algoritmo Recursivo	189
21.1.3 Algoritmo usando compreensão de listas	189
21.1.4 Exercícios	190
21.2 Merge Sort	190
21.2.1 Algoritmo	190
21.2.2 Análise de Complexidade	193
21.2.3 Solução alternativa para o Merge sort	193
21.2.4 Exercícios	194
III Tópicos avançados	195
22 Análise de algoritmos	197
22.0.1 Tamanho de entrada	198
22.0.2 Tempo de execução	198
22.1 Análise do algoritmo Selection Sort	198
22.2 Comparação de tempos de execução	202
22.3 Exercícios propostos	204
23 Ordens Assintóticas de Complexidade	205
23.1 Notação O	205
23.2 Notação Θ	207
23.3 Notação Ω	208
23.4 Análise do algoritmo Bubble Sort	209
23.5 Combinando notações assintóticas	213
23.5.1 Soma	213

23.5.2 Transitividade	214
23.5.3 Reflexividade	214
23.5.4 Simetria	214
23.6 Exercícios propostos	215
24 Análise de algoritmos recursivos	217
24.1 Definições recursivas	217
24.2 Algoritmos recursivos	218
24.3 Análise de algoritmos recursivos	221
24.3.1 Método da árvore de recorrência	221
24.3.2 Método de resolução por substituição	224
24.3.3 Método de resolução pelo teorema geral	224
24.4 Exercícios propostos	226

Listas de Figuras

2.1 Chambers's Mathematical Tables, New Edition, Londres, 1901	14
2.2 Principais componentes do hardware	15
2.3 Exemplo de terminal no macOS	17
3.1 Variáveis	22
4.1 Divisão floor	26
5.1 Esquema de indentação de blocos de código	33
5.2 Fluxograma do exemplo de comando condicional	34
5.3 Circunferência com centro na origem do sistema de coordenadas	38
5.4 Jogo pedra, papel e tesoura	39
5.5 Dica para jogo pedra, papel e tesoura	40
6.1 Fluxograma do comando while	42
6.2 Fluxograma do comando for	44
8.1 Visualização do frame global com Python Tutor	60
8.2 Visualização do frame global com Python Tutor	61
8.3 Método de Newton	66
8.4 Visualização de frames independentes global e local com Python Tutor	69
9.1 Imutabilidade de strings	72
9.2 Indexação de strings	72
11.1 Comparação de referências de strings	87
11.2 Comparação de referências de listas	88
11.3 Exemplo de alias de listas	89
11.4 Exemplo de clonagem com fatiamento	91
11.5 Exemplo passagem de lista como parâmetro de função	92
15.1 Exemplos de vetores em 2D	134
15.2 Vetores a serem adicionados	134
15.3 Adição de vetores	134
15.4 Subtração de vetores	135
15.5 Multiplicação de vetor por escalar	135
15.6 Divisão de vetor por escalar	135
15.7 Normalização de vetor	136
15.8 Criação de vetor de comprimento unitário	136

16.1 Exemplos de gráfico do matplotlib	149
16.2 Exemplos de gráfico do matplotlib com legendas	150
16.3 Exemplos de plotagem de múltiplas curvas	152
19.1 Exemplo de busca binária	174
20.1 Torres de Hanoi com três discos	178
20.2 Números de fibonacci	182
20.3 Árvore de recursão para fib(4)	182
20.4 Uma solução do problema das 8 rainhas	185
21.1 Ordem de execução das chamadas recursivas do Mergesort	191
21.2 Retorno do exemplo anterior	192
22.1 Exemplo de execução do algoritmo SelectionSort	199
22.2 Tempo de execução de algoritmo quadrático x algoritmo linear	202
23.1 Notação O, onde $f(n) = O(g(n))$	206
23.2 Notação Θ , onde $f(n) = \Theta(g(n))$	208
23.3 Notação Ω , onde $f(n) = \Omega(g(n))$	209
23.4 Exemplo de execução do algoritmo BubbleSort	210
24.1 Definição recursiva de árvores binárias	218
24.2 Exemplo de execução do algoritmo MergeSort	219
24.3 Exemplo de execução do procedimento Merge	227
24.4 Árvore de recorrência para $T(n) = 2T(n/2) + cn$	228
24.5 Árvore de recorrência para $T(n) = 3T(n/4) + cn^2$	229

Lista de Quadros

1.1	Exemplo de cálculo de raiz quadrada	8
3.1	Tipos de dados básicos em Python	20
21.1	Taxa de crescimento do trabalho de um algoritmo	187
22.1	Custo do SelectionSort	200
22.2	Tamanho máximo do problema em função do tempo	203
23.1	Custo do BubbleSort	210
23.2	Crescimento de funções	212

Lista de exemplos

1.1	Exemplo de algoritmo para cálculo de raiz quadrada	9
4.1	função print	23
4.2	função print com vírgulas	23
4.3	função print com separador	23
4.4	redefinindo o caracter de término do print	24

Prefácio

A maior parte deste material foi escrito pelo professor Marcio Machado Pereira, da Universidade Estadual de Campinas - UNICAMP. Inicialmente o material foi elaborado para a disciplina “Algoritmos e Programação de Computadores” daquela instituição. Porém, devido à semelhança com a ementa da disciplina “Algoritmos” do Bacharelado em Ciência da Computação do IFC – Campus Blumenau, eu entrei em contato com o Prof. Márcio, que, gentilmente permitiu que eu usasse seu material como base para a disciplina no IFC. Esta parceria com o prof. Márcio resultou no presente documento, que traz algumas contribuições de minha parte ao texto original para melhor atender às necessidades dos alunos do IFC. Fica aqui o meu agradecimento pessoal ao prof. Márcio e à Unicamp, por nos permitir utilizar seu material.

Paulo César Rodacki Gomes

Durante a elaboração deste caderno de lições, eu me beneficiei显著mente de materiais disponíveis na web e, claro, procuro dar aqui os devidos créditos. Embora tenha tentado ser preciso e correto, se esqueci de alguma citação ou se infringi alguma falha de propriedade intelectual, eles são única e exclusivamente de minha responsabilidade.

- *Composing Programs*, John DeNero, UC Berkeley, Creative Commons Attribution-ShareAlike 3.0 Unported License.
- *Introduction to Computation and Programming Using Python*, Ana Bell, Eric Grimson e John Guttag. MIT OpenCourseWare, Creative Commons License.
- *Explorations in Computing - An Introduction to Computer Science and Python Programming*, John S. Conery, University of Oregon, Creative Commons License.
- Livro *Think Python 2nd Edition*, Allen B. Downey, licença Creative Commons Atribuição-NãoComercial CC BY-NC 3.0.
- Python 3 Official Documentation. (<https://docs.python.org/3/>). PFS license, GPL-compatible.

Márcio Machado Pereira

Parte I

Programação Python

Capítulo 1

Algoritmos e Programação de Computadores

É difícil imaginar o nosso mundo sem computadores, embora não pensemos muito nos computadores reais. Também é difícil imaginar o que a civilização humana fez sem tecnologia de computadores nestes milhares de anos e que o mundo, como o conhecemos hoje, esteja tão envolvido com a tecnologia da informação nos últimos 25 anos.

Neste curso, que compreende a disciplina de “Algoritmos” do Bacharelado em Ciência da Computação do IFC – Campus Blumenau, vocês aprenderão sobre **ciência da computação**, que é o estudo da **computação** que tornou possível essa nova tecnologia e este novo mundo. Você também aprenderão a usar os computadores de forma eficaz e adequada para melhorar sua própria vida e a vida dos outros por meio da concepção, da implementação e de testes de algoritmos na linguagem de programação Python.

1.1 Computação

Uma **computação** é uma seqüência de operações bem definidas que levam de um ponto de partida inicial para um resultado final desejado. - note que esta definição não inclui a palavra “computador” - uma computação é um processo que pode ser realizado por uma pessoa ou máquina - a mesma computação pode ser realizada usando qualquer uma de várias tecnologias diferentes

1.2 Duas idéias fundamentais de ciência da computação

Como a maioria das áreas de estudo, a ciência da computação se concentra em um amplo conjunto de idéias inter-relacionadas. Duas das mais básicas são: **algoritmos** e **processamento de informações**. Essas idéias serão introduzidas de maneira informal, mas posteriormente vamos examiná-las com mais detalhes.

1.3 Algoritmos

A seqüência de etapas realizada durante uma computação é definida por um **algoritmo**. Um algoritmo pode ser pensado como uma “receita” do tipo: “Siga estas etapas e você irá resolver o problema”.

Um algoritmo deve incluir uma descrição completa dos seguintes ítems:

- o conjunto de insumos ou condições iniciais: uma especificação completa do problema a ser resolvido
- conjunto de saídas: descrições de soluções válidas para o problema
- seqüência de operações que acabará por produzir o resultado: as etapas devem ser simples e precisas

Pessoas já efetuavam cálculos muito antes da invenção de dispositivos de computação modernos, e muitos continuam a usar dispositivos de computação que consideramos primitivos. Por exemplo, considere a forma como os comerciantes efetuavam troco para clientes em mercados antes da existência de cartões de crédito, calculadoras de bolso ou caixas registradoras. Efetuar troco pode ser uma atividade complexa. Provavelmente levou algum tempo para aprender a fazê-lo, e é preciso algum esforço mental para fazê-lo correto sempre. Vamos considerar o que está envolvido neste processo.

O primeiro passo é calcular a diferença entre o preço de compra e a quantia de dinheiro que o cliente entrega ao comerciante. O resultado desse cálculo é o valor total que o comerciante deve retornar ao comprador. Por exemplo, se você comprar uma dúzia de ovos no mercado dos agricultores por R\$ 2,39 e você entrega ao agricultor uma nota de R\$ 10, ele deve retornar R\$ 7,61 para você. Para produzir esse valor, o comerciante seleciona, por exemplo, uma nota de R\$ 5,00, uma nota de R\$ 2,00 e as moedas de 50 centavos, 10 centavos e 1 centavo.

Poucas pessoas conseguem subtrair números de três dígitos sem recorrer a alguma ajuda manual, como lápis e papel. Como você aprendeu na escola primária, a subtração pode ser realizada com lápis e papel seguindo uma seqüência de etapas bem definidas. Você provavelmente já fez isso inúmeras vezes, mas nunca fez uma lista das etapas específicas envolvidas. Fazer essas listas para resolver problemas é algo que os cientistas da computação fazem o tempo todo. Por exemplo, a seguinte lista de etapas descreve o processo de subtrair dois números usando um lápis e papel:

Passo 1: Anote os dois números, com o número maior acima do número menor e seus dígitos alinhados nas colunas da direita para a esquerda.

Passo 2: Selecione a coluna mais à direita para iniciar o processo de subtração.

Passo 3: Anote abaixo a diferença entre os dois dígitos na coluna de dígitos atual, emprestando um 1 da próxima coluna do número superior à esquerda, se necessário.

Passo 4: Se não houver uma próxima coluna à esquerda, pare. Caso contrário, vá para a próxima coluna à esquerda e vá para o Passo 3.

Se o agente de computação (neste caso um ser humano) seguir cada uma dessas etapas simples corretamente, todo o processo resultará em uma solução correta para o problema dado. Assumimos no **Passo 3** que o agente já sabe como calcular a diferença entre os dois dígitos em qualquer coluna, emprestando da coluna à esquerda, se necessário.

Para realizar o troco, a maioria das pessoas pode selecionar a combinação de notas e moedas que representam a quantidade de troco correta, sem qualquer ajuda manual, além das moedas e notas. Mas os cálculos mentais envolvidos ainda podem ser descritos de forma semelhante aos passos anteriores, e podemos recorrer a escrevê-los no papel se houver uma disputa sobre a correção do troco. A sequência de etapas que descreve cada um desses processos computacionais é chamada de algoritmo. Informalmente, um algoritmo é como uma receita. Ele fornece um conjunto de instruções que nos diz como fazer algo, como fazer trocos, assar pão ou construir um automóvel. Mais precisamente, um algoritmo descreve um processo que termina com uma solução para um problema. O algoritmo também é uma das idéias fundamentais da ciência da computação.

Um algoritmo tem as seguintes características:

1. Um algoritmo consiste em um número finito de instruções.
2. Cada instrução individual em um algoritmo está bem definida. Isso significa que a ação descrita pela instrução pode ser realizada efetivamente ou ser executada por um agente de computação. Por exemplo, qualquer agente de computação capaz de cálculos aritméticos pode calcular a diferença entre dois dígitos. Portanto, um passo algorítmico que diz “calcular a diferença entre dois dígitos” seria bem definido. Por outro lado, um passo que diz “dividir um número por 0” não está bem definido, porque nenhum agente de computação poderia realizá-lo.
3. Um algoritmo descreve um processo que eventualmente pára depois de chegar a uma solução para um problema. Por exemplo, o processo de subtração pára depois que o agente de computação reduz a diferença entre os dois dígitos na coluna de dígitos mais à esquerda.
4. Um algoritmo resolve uma classe geral de problemas. Por exemplo, um algoritmo que descreve como fazer o troco deve funcionar para qualquer duas quantias de dinheiro cuja diferença seja maior ou igual a 0.

A capacidade de dividir uma tarefa em suas partes componentes é um dos principais trabalhos de um programador de computador. Uma vez que podemos desenvolver um algoritmo para resolver um problema, podemos automatizar a tarefa de resolver o problema. Os computadores podem ser projetados para executar um pequeno conjunto de algoritmos para realizar tarefas especializadas, como o funcionamento de um forno de microondas. Mas também podemos criar computadores, como o seu notebook, capazes de executar uma tarefa descrita por qualquer algoritmo. Esses computadores são verdadeiras máquinas de solução de problemas de propósito geral. Eles são diferentes de qualquer máquina que já construímos antes, e eles formaram a base do mundo completamente novo em que vivemos. Mais adiante nós iremos ver uma notação para expressar algoritmos para serem executados pelos computadores bem como algumas sugestões para projetar algoritmos. Você verá que os algoritmos e o pensamento algorítmico são fundamentos críticos de qualquer sistema de informação.

Em resumo, a palavra “algoritmo”, denota qualquer método especial para resolver um certo tipo de problema [17]. Informalmente podemos definir um algoritmo como um procedimento computacional bem definido que pega um valor, ou um conjunto de valores, como entrada e produz um valor, ou conjunto de valores, como saída. Portanto, um algoritmo é uma seqüência de passos computacionais não ambíguos que transforma a entrada em saída [9].

1.3.1 Exemplo: problema de ordenação

Um exemplo de problema para o qual existem vários algoritmos é o problema de ordenação de uma sequência de números em uma ordem não-decrescente. Formalmente, o problema de ordenação pode ser definido da seguinte forma:

Entrada: uma sequência de n números $\langle a_1, a_2, \dots, a_n \rangle$.

Saída: uma permutação (ou reordenamento) $\langle a'_1, a'_2, \dots, a'_n \rangle$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Dada uma sequência de entrada como $\langle 25, 37, 12, 48, 57, 92, 33, 86 \rangle$, por exemplo, um algoritmo de ordenação deve retornar uma sequência de saída igual a $\langle 12, 25, 33, 37, 48, 57, 86, 92 \rangle$. A sequência de entrada é chamada de uma instância do problema de ordenação. Uma instância de um problema corresponde então a todo o conjunto de entrada necessário para resolver o problema.

Um algoritmo é dito correto quando, para qualquer instância de entrada, produz uma saída correta, se forem fornecidos tempo e memória suficiente para sua execução [28]. Neste caso, dizemos que um algoritmo correto resolve um dado problema computacional. Um algoritmo incorreto pode produzir uma saída incorreta ou pode até não terminar sua execução.

O fato de um algoritmo teoricamente resolver determinado problema nem sempre significa que seja aceitável na prática, pois o tempo e a memória consumidos podem ser de extrema importância. Muitas vezes existem vários algoritmos diferentes para resolver um mesmo problema, e uma questão importante na Ciéncia da Computação é determinar qual o melhor algoritmo para resolver o problema. Nesse caso, uma regra geral é escolher o algoritmo que é mais fácil de entender, de implementar e de documentar [1]. Entretanto, normalmente o desempenho do algoritmo é um fator importante, então é necessário escolher um algoritmo que execute com rapidez e use os recursos computacionais disponíveis eficientemente. Por este motivo, é necessário considerar o problema de avaliar o tempo de execução de uma algoritmo e quais ações podem ser tomadas para solucionar problemas com mais rapidez.

1.3.2 A escolha de um algoritmo

A escolha de um algoritmo para resolver determinado problema computacional passa por uma série de critérios. Por exemplo, caso seja necessário implementar determinado software para ser utilizado somente uma vez, com um conjunto pequeno de dados, então deve-se escolher o algoritmo mais simples e fácil de implementar conhecido, para ter o código escrito e depurado o mais rápido possível. Porém, se o software vai ser usado várias vezes (e possivelmente mantido) por várias pessoas por um período de tempo mais longo, então outras questões se tornam importantes.

A primeira questão a ser considerada é a **simplicidade**. Um algoritmo mais simples é desejável por vários motivos. Em primeiro lugar, é mais fácil de implementar, e existe menor probabilidade de se cometer erros ao implementar algoritmos mais simples, em relação a algoritmos mais complicados.

Outra questão é a **clareza** na escrita do código fonte do algoritmo e o cuidado na documentação para que ele possa ser mantido posteriormente por outras pessoas. Um algoritmo simples sempre é mais fácil de ser descrito. Se ele for implementado de forma clara e com boa documentação, modificações posteriores no código fonte original podem ser feitas de forma mais fácil e segura por uma outra pessoa pois o autor original muitas vezes não está mais disponível para realizar essa tarefa.

Finalmente, chegamos à questão central de nosso estudo, a **eficiência** do algoritmo. Quando o algoritmo vai ser empregado várias vezes, sua eficiência torna-se importante. Geralmente a eficiência é associada ao tempo que um algoritmo leva para resolver uma instância de um problema, porém, outros recursos também podem ser considerados, tais como: **(i)** a quantidade de memória utilizada pelo algoritmo, **(ii)** tráfego gerado na rede quando o algoritmo é distribuído e **(iii)** quantidade de dados que deve ser lida e/ou escrita em disco, quando for o caso [9] [22].

1.4 Processamento de Informações

Desde que os seres humanos aprenderam a escrever, vários milhares de anos atrás, eles processaram a informação. A própria informação tomou muitas formas em sua história, das marcas impressas na argila na Mesopotâmia antiga, aos primeiros textos escritos na Grécia antiga, às palavras impressas nos livros, jornais e revistas produzidas em massa desde a Renascença, aos símbolos abstratos da matemática e da ciência modernas utilizados nos últimos 350 anos. Recentemente, no entanto, os seres humanos desenvolveram a capacidade de automatizar o processamento de informações construindo computadores. No mundo moderno dos computadores, a informação também é comumente referida como dados.

Mas o que é informação?

Como os cálculos matemáticos, o processamento da informação pode ser descrito com algoritmos. Em nosso exemplo anterior de efetuar o troco, as etapas de subtração envolveram a manipulação de símbolos usados para representar números e dinheiro. Ao levar a cabo as instruções de qualquer algoritmo, um agente manipula a informação. O agente de computação começa com algumas informações (conhecidas como entrada), transforma essas informações de acordo com regras bem definidas e produz novas informações, conhecidas como saída.

É importante reconhecer que os algoritmos que descrevem o processamento da informação também podem ser representados como informações. Os cientistas da computação conseguiram representar algoritmos de uma forma que pode ser executada de forma eficaz e eficiente pelas máquinas. Eles também criaram máquinas reais, chamadas de computadores digitais eletrônicos, que são capazes de executar estes algoritmos. Os cientistas da computação descobriram mais recentemente como representar muitas outras coisas, como imagens, audio e vídeo, como informação. Muitos dos meios de comunicação e dispositivos de comunicação que agora damos por certo seriam impossíveis sem esse novo tipo de processamento de informações.

1.4.1 Exercício

Descreva um algoritmo que seja capaz de calcular a raiz quadrada de um número inteiro positivo usando somente as operações básicas (adição, subtração, multiplicação e divisão).

Definição (declarativa, isto é, “o que é verdadeiro”):

$$\sqrt{x} = y, \text{ tal que } y \geq 0 \text{ e } y^2 = x$$

Solução (imperativa, isto é, “como fazer”):

Como exemplo de solução, vamos usar o método de Newton de Aproximações Sucessivas. Este método nos diz que sempre que tivermos um palpito y para o valor da raiz quadrada de um número x , podemos realizar uma manipulação simples para obter uma melhor adivinhação (próxima à raiz quadrada real) com a média entre y e x/y . Por exemplo, podemos calcular a raiz quadrada de 2 da seguinte maneira. Suponha que nosso palpito inicial seja 1:

Dividindo a tarefa em componentes:

Passo1: Comece com um palpito para y

Passo2: Se $y * y$ estiver próximo o suficiente de x , pare e diga que y é a resposta

Passo3: Caso contrário, melhore o palpito com base na média entre y e x/y , i.e., $\frac{(y+\frac{x}{y})}{2}$

Passo4: Usando este novo palpito, que novamente chamamos de y , repita o Passo2.

Quadro 1.1: Exemplo de cálculo de raiz quadrada

Palpite	Quociente	Média
1	$\frac{2}{1} = 2$	$\frac{(2+1)}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333$	$\frac{(1.3333+1.5)}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{(1.4167+1.4118)}{2} = 1.4142$
1.4142

Resumo

Para solucionar o problema nós definimos:

1. uma seqüência de etapas
2. fluxo de processo de controle que especifica quando cada etapa é executada
3. um meio de determinar quando parar

$1 + 2 + 3 = \text{um algoritmo!}$

1.5 Linguagens de Programação

Seria bom se pudéssemos abrir um telefone celular e dizer “enviar uma mensagem para Alex, Katia e Erica para ver se eles querem vir para estudar cálculo”. Poderíamos então trabalhar com outra coisa enquanto o telefone compõe uma mensagem de texto, envia para os telefones de nossos amigos e negocia com os outros telefones para encontrar um momento em que todos desejam se encontrar. Esse tipo de interação não é possível com computadores atuais, mas pesquisadores em um campo

conhecido como inteligência artificial estão tentando entender o que está envolvido nesses tipos de comunicação e desenvolver métodos computacionais para realizá-los. Hoje um assistente pessoal humano pode realizar esta tarefa, então este é um exemplo do tipo de problema que os humanos resolvem, mas os computadores (atualmente) não. É uma questão aberta se este problema está além dos limites da computação. Muito bem, pode ser possível que algum assistente pessoal digital no futuro realize essa tarefa.

As descrições das etapas de um algoritmo em português ou em outra linguagem humana, como no algoritmo para calcular a raiz quadrada de um número inteiro positivo, é suficiente para falar sobre o algoritmo, para descrever o processo para outra pessoa ou para tentar entender se o algoritmo funciona ou não. Mas, para executar o algoritmo em um computador, as etapas devem ser descritas com mais precisão. Nesta descrição, as etapas devem ser simples o suficiente para serem “entendidas” por uma máquina. Uma maneira de pensar o que uma máquina é capaz de fazer é pensar em termos de símbolos, como números ou letras. As etapas em um algoritmo são basicamente configurações de símbolos, como operações aritméticas simples ou comparações que determinam quais as palavras que vêm antes de outros no alfabeto.

Cientistas da computação resolveram esse problema criando notações para expressar cálculos de forma exata e inequívoca. Essas notações especiais são chamadas de linguagens de programação. Toda estrutura em uma linguagem de programação tem uma forma precisa (sua sintaxe) e um significado preciso (sua semântica). Uma linguagem de programação é algo como um código para escrever as instruções que um computador seguirá. De fato, os programadores geralmente se referem a seus programas como código de computador, e o processo de escrever um algoritmo em uma linguagem de programação é chamado de codificação. Python é um exemplo de uma linguagem de programação. É o idioma que usaremos neste curso. Você pode ter ouvido falar de outros idiomas, como C++, Java, Perl, Haskell, Swift ou PHP. Embora essas linguagens diferem em muitos detalhes, todas elas compartilham a propriedade de ter *sintaxe* e *semântica* bem definidas, inequívocas.

1.5.1 Expressando algoritmos em linguagens de programação

As linguagens de programação são naturalmente adequadas para se expressar algoritmos porque:

- Uma linguagem de programação fornece um conjunto de operações primitivas.
- Fornece expressões, que são combinações legais de primitivas.
- As Expressões e cálculos têm valores e significados.

1.5.2 Exemplo do algoritmo “Raiz Quadrada” em Python

O exemplo a seguir mostra a codificação em Python do algoritmo para cálculo de raiz quadrada pelo método de Newton mostrado no exercício 1.4.1.

Exemplo 1.1: Exemplo de algoritmo para cálculo de raiz quadrada

```
def sqrt(x):
    """ Calculate the square root of a positive integer """

    def average(y, x):
        return (y + x) / 2

    def improve(y, x):
```

```

        return average(y, x / y)

def abs(x):
    if x >= 0:
        return x
    else:
        return -x

def good_enough(y, x):
    return abs((y * y) - x) < 0.000001

def sqrt_iter(y, x):
    if good_enough(y, x):
        return y
    else:
        return sqrt_iter(improve(y, x), x)

return sqrt_iter(1.0, x)

```

Exemplo de execução

```

>>> sqrt(2)
>>> 1.4142135623746899

```

1.6 Programa de Estudos

Este curso irá abranger quatro aspectos principais da computação:

Noções básicas de programação: tipos de dados, estruturas de controle, desenvolvimento de algoritmos e design de programas com funções são idéias básicas que vocês precisarão dominar para resolver problemas com computadores. Este curso examinará estes tópicos fundamentais em detalhes e a parte prática permitirá sua compreensão deles para resolver uma ampla gama de problemas.

Processamento de dados e informações: os programas mais úteis dependem das estruturas de dados para resolver problemas. Essas estruturas de dados incluem seqüências imutáveis como tuplas e strings (cadeias de caracteres), sequências mutáveis como listas, conjuntos e dicionários, e tipos abstratos de dados (ADT) como pilhas, filas e árvores. Neste curso vocês irão usar, construir e avaliar o desempenho das estruturas de dados. O conceito geral de um tipo de dados abstrato será introduzido, assim como a diferença entre abstração e implementação. Você irão aprender a usar a análise de complexidade para avaliar os custos de espaço/tempo de diferentes implementações de ADTs. Sem o bom entendimento da eficiência, é possível levar até os computadores mais rápidos a uma parada de execução ao trabalhar com grandes conjuntos de dados.

Noções de programação orientada a objetos (OOP): A programação orientada a objetos é o paradigma de programação dominante usado para desenvolver grandes sistemas de software. Se houver tempo suficiente, nós iremos introduzir ao final do curso os princípios fundamentais da OOP.

Noções de análise de complexidade de algoritmos: A avaliação da eficiência de um algoritmo é chamada de **análise de um algoritmo**, e consiste fundamentalmente em estimar a quantidade de

recursos requeridos pelo algoritmo, em especial o **tempo** necessário para solucionar um problema. Em geral, ao se analizar vários algoritmos que solucionam um determinado problema, o mais eficiente é identificado. Além disso, esta análise pode indicar mais de um candidato e apontar vários algoritmos inferiores que devem ser descartados.

1.7 Porque aprender a programar (alguns exemplos)

- Como engenheiros ou cientistas da computação vocês deverão ser capazes de automatizar algum processo.
 - Vocês poderão criar programas para gerenciar e automatizar algum processo que hoje é manual.
- Vocês deverão ser capazes de desenvolver novas ferramentas ou protótipos.
 - Para criar ferramentas/protótipos vocês deverão fazer simulações computacionais para fazer testes preliminares.
- Vocês poderão enxergar situações onde uma solução computacional pode trazer benefícios.
 - Mesmo que vocês não implementem (programem) a solução vocês poderão propô-la e serem capazes de “conversar” com o pessoal de TI para implementar a solução.
- Como cientistas vocês devem propor uma hipótese e testá-la.
- Em vários casos onde os sistemas podem ser “modelados matematicamente”, são criados programas que fazem a simulação do sistema para checagem de uma hipótese.
- Vocês deverão resolver sistemas complexos de equações que não necessariamente podem ser resolvidos por softwares padrões (como MatLab).
- Vocês deverão implementar seus próprios resolvedores.
- Simulações.
- Muitos dos modelos propostos para explicar algum fenômeno são simulados computacionalmente. Implementar os modelos é uma tarefa básica.

1.8 O que será necessário

Você deverá ter acesso a um computador. Você vai precisar também ter acesso a uma aplicação chamada **Terminal**. Talvez seja a aplicação mais útil, porque você poderá usar o mesmo para dizer ao seu computador que faça praticamente tudo o que vocês quiserem. Usuários de Mac e Linux provavelmente estão familiarizados com este aplicativo. No Windows isso é um pouco mais complicado, já que a Microsoft é um tanto rebelde. Você podem instalar um emulador de terminal mais amigável do que o “Prompt de Comando” ou mesmo o “PowerShell” que acompanham o Windows. Eu sugiro instalar o Git Bash, a partir de <https://git-scm.com/downloads>. Durante a instalação, certifique-se de selecionar a opção “Use Windows’ default console window”. Uma outra opção, recomendado principalmente para quem quer iniciar no mundo Linux, é instalar o Shell Bash do Linux no Windows através do subsistema Windows para o Linux, presente nas versões mais atuais do Windows 10. Aqui vocês encontraram um tutorial de como fazer isto.

Para criar um programa, utilizamos um **editor de texto** para escrever o código do programa (e.g., Vim, Atom) ou um IDE – Ambiente de Desenvolvimento Integrado (e.g., PyCharm, WingIDE) e um **compilador/interpretador python**. O compilador é o que transforma o código em um programa executável. O interpretador é um programa que executa diretamente os comandos da

linguagem. Será preciso instalar o compilador/interpretador python da versão 3. Vocês poderão baixá-lo do site <https://www.python.org/downloads/>.

Uma outra opção para editar programas em Python é o Microsoft Visual Studio Code, ambiente gratuito e multiplataforma, disponível em <https://code.visualstudio.com>.

1.9 Extras

A seguir, temos algumas referências complementares que podem ser úteis:

- Livro “Practical Vim, second edition - Edit Text at the Speed of Thought” The Pragmatic Bookshelf
- Videos de Derek Wyatt sobre o editor Vim no Vimeo Derek Wyatt
- Livro (online) sobre o editor Atom Atom Flight Manual
- YouTube Video “Setting up a Python Development Environment in Atom” Corey Schafer
- Full Stack Python é um livro aberto que explica conceitos em linguagem simples e fornece recursos excelentes sobre esses tópicos.

Capítulo 2

Computação e Computadores - Breve Histórico

A resolução de problemas complexos pela execução sistemática de operações simples e diretas data de milhares de anos. Filósofos da antiga Grécia, Egito e China descobriram muitos fatos importantes sobre números e seus relacionamentos. Eles desenvolveram métodos que ainda hoje são usados para determinar se um número é primo ou como encontrar o maior denominador comum de um par de números.

Esta idéia de resolver computacionalmente um problema complexo pela execução repetida e sistemática de uma série de operações simples e diretas já estava bem estabelecida no século XIX. Os matemáticos desenvolveram técnicas para calcular entradas em tabelas usando apenas as operações mais básicas de aritmética, como adição e subtração, onde o valor em uma linha da tabela pode ser determinado usando valores de linhas preenchidas anteriormente. Muitas tabelas foram produzidas por grupos de pessoas que não tinham habilidades matemáticas avançadas, mas foram contratadas e treinadas para preencher uma tabela fazendo uma seqüência específica de adições e subtrações. Antes de meados do século XX, a palavra “computador” era um título de trabalho, referindo-se a qualquer pessoa envolvida no cálculo sistemático de valores como os encontrados em tabelas matemáticas. O matemático Babbage percebeu que as simples operações realizadas por computadores humanos eram de natureza mecânica, e ele sonhava com um dia onde uma máquina seria capaz de executar as etapas em uma computação de forma automática.

Antes de haver computadores para calcular funções matemáticas, se uma pessoa quisesse saber o valor de uma função trigonométrica, ela iria olhar em uma tabela de senos e cossenos. Por exemplo, para encontrar o valor de $\cos 30^\circ 20'$, a pessoa procurava a página de 30° , então buscava a linha de $20'$ e olhava na coluna rotulada como “coseno”. A figura 2.1 mostra um exemplo de publicação técnica que continha diversas tabelas matemáticas.

Hoje, a computação é tão familiar que nem pensamos como ela faz parte do dia a dia do ser humano nas formas mais corriqueiras. Um navegador, um topógrafo, um arquiteto ou qualquer outra pessoa que precise saber o valor de uma função matemática simplesmente insere um número em uma calculadora e pressiona um botão rotulado com o nome da função. A computação também está no cerne de aplicativos de computador que nos ajudam com tarefas comuns que, aparentemente, têm pouco ou nada a ver com a matemática, como usar um processador de texto para escrever um artigo, organizar uma biblioteca de música ou reproduzir uma música gravada.

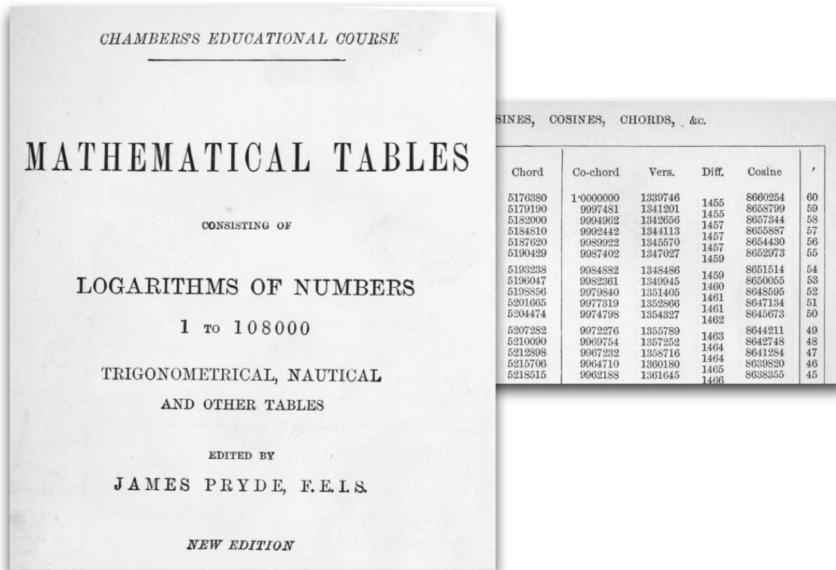


Figura 2.1: Chambers's Mathematical Tables, New Edition, Londres, 1901

Os primeiros computadores eletrônicos foram desenvolvidos durante a Segunda Guerra Mundial. Logo após o fim da guerra, a idéia de usar máquinas para automatizar os cálculos em ciência e negócios começou a se espalhar e várias empresas começaram a fabricar máquinas de computação. As máquinas eram muito grandes e muito caras, e foram encontradas apenas nas maiores empresas, agências governamentais ou laboratórios de pesquisas universitárias. Eles foram usados para tarefas tão diversas como calcular as trajetórias de foguetes e mísseis, prever o clima, e aplicativos de processamento de dados empresariais para folha de pagamento e contabilidade.

Nos últimos cinquenta anos a computação tornou-se uma parte essencial da vida moderna. Todos os dias, escrevemos emails, compartilhamos fotografias, reproduzimos músicas, lemos as notícias e pagamos contas usando nossos computadores pessoais. Os engenheiros usam computadores para projetar carros e aviões, as empresas farmacêuticas usam computadores para desenvolver novos medicamentos, os produtores de filmes geram efeitos especiais e, em alguns casos, filmes animados inteiros usando computadores e empresas de investimento usam modelos computacionais para decidir se as transações complexas são susceptíveis de sucesso. Não surpreendentemente, dado o papel que a astronomia desempenhou na história da computação, os astrônomos modernos também dependem fortemente da computação. Organizações como o Jet Propulsion Laboratory usam computadores para realizar os cálculos que acompanham as localizações de planetas, asteróides e cometas com o objetivo de se manter atento a possíveis ameaças de impactos, como o que envolveu o cometa Shoemaker-Levy e Jupiter em 1994.

A computação desempenha um papel muito mais extenso na ciência moderna do que o “cruzamento de dados” direto envolvido no cálculo de órbitas. A frase ciência computacional refere-se ao uso da computação para ajudar a responder a questões científicas fundamentais. Aqui, a palavra “computacional” é um adjetivo que descreve como a ciência é feita. A ciência computacional é, como as abordagens mais tradicionais da ciência teórica e da ciência experimental, uma maneira de tentar resolver importantes problemas científicos. Físicos computacionais usam computadores para estudar a formação de buracos negros, investigar teorias de como os planetas se formam e simular a colisão prevista, nos próximos três bilhões de anos, de nossa galáxia Via Láctea com a galáxia

Andrómeda.

A definição geralmente aceita de uma computação é que é uma seqüência de passos simples e bem definidos que levam à solução de um problema. O problema em si deve ser definido exatamente e sem ambigüidade, e cada passo na computação que resolve o problema deve ser descrito em termos muito específicos. Uma computação é um processo, uma seqüência de operações simples que leva de um estado inicial ao resultado final desejado. O processo pode ser realizado inteiramente por uma pessoa, ou por uma pessoa usando a ajuda de calculadoras mecânicas ou eletrônicas, ou completamente automatizadas por um computador. A escolha de qual tecnologia seria mais efetiva depende da situação.

2.1 A Estrutura de um Computador Moderno

Um sistema de computação moderno consiste em hardware e software. O hardware consiste nos dispositivos físicos necessários para executar algoritmos. O software é o conjunto desses algoritmos, representados como programas em linguagens de programação específicas. A seguir, nos concentraremos no hardware e no software encontrados em um sistema de computador de mesa típico (desktop, notebook), embora componentes semelhantes também sejam encontrados em outros sistemas de computação, como dispositivos móveis e caixas eletrônicas.

2.1.1 Hardware

Os componentes básicos do hardware de um computador são as memórias, a unidade de processamento central (CPU) e um conjunto de dispositivos de entrada e saída, conforme mostrado na Figura 2.2.

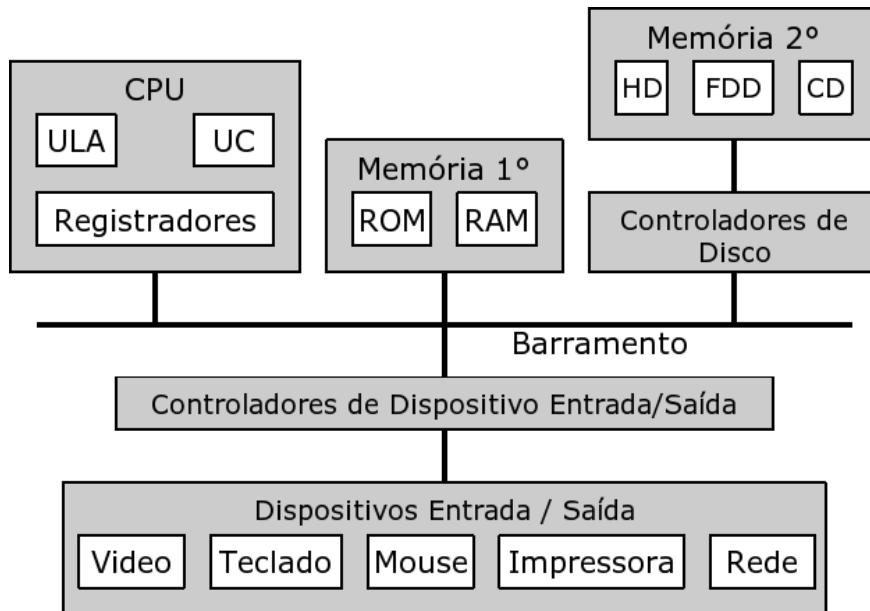


Figura 2.2: Principais componentes do hardware

O propósito da maioria dos dispositivos de entrada é converter informações que os seres humanos tratam, como texto, imagens e sons, em informações para o processamento computacional. A finalidade da maioria dos dispositivos de saída é converter os resultados desse processamento de volta para a forma utilizável para os humanos.

A memória do computador é configurada para representar e armazenar informações em formato eletrônico. Especificamente, as informações são armazenadas como padrões de dígitos binários (1s e 0s). As informações armazenadas na memória podem representar qualquer tipo de dados, como números, texto, imagens ou som, ou as instruções de um programa. Também podemos armazenar na memória um algoritmo codificado como instruções binárias para o computador. Uma vez que a informação é armazenada na memória, normalmente queremos fazer algo com isso - isto é, queremos processá-lo. A parte de um computador responsável pelo processamento de dados é a unidade central de processamento (CPU). Este dispositivo, que às vezes também é chamado de processador, consiste em interruptores eletrônicos dispostos para realizar operações lógicas, aritméticas (ULA) e de controle simples (UC). A CPU executa um algoritmo obtendo suas instruções binárias da memória, descodificando-as e executando-as. Executar uma instrução pode envolver a busca de outras informações binárias - a memória de dados também.

O processador pode localizar dados na memória principal (RAM) de um computador muito rapidamente. No entanto, esses dados existem apenas enquanto a energia elétrica entra no computador. Se a energia falhar ou estiver desligada, os dados na memória primária são perdidos. Claramente, é necessário um tipo de memória mais permanente para preservar os dados. Esse tipo de memória mais permanente é chamado de memória externa ou secundária, e vem em várias formas. Os meios de armazenamento magnéticos, como fitas e discos rígidos, permitem que padrões de bits sejam armazenados como padrões em um campo magnético. Os meios de armazenamento de semicondutores, como as memória flash, executam a mesma função com uma tecnologia diferente, assim como a mídia de armazenamento óptico, como CDs e DVDs. Alguns desses meios de armazenamento secundários podem conter quantidades muito maiores de informações do que a memória interna de um computador.

2.1.2 Software

Nós dissemos que um computador é uma máquina de solução de problemas de propósito geral. Para resolver qualquer problema computável, ele deve ser capaz de executar qualquer algoritmo. Como é impossível antecipar todos os problemas para os quais existem soluções algorítmicas, não há como “gravar” todos os algoritmos potenciais no hardware de um computador. Em vez disso, nós construímos algumas operações básicas no processador e demandamos que os algoritmos sejam projetados para usá-las. Os algoritmos são convertidos em forma binária e depois carregados, com seus dados, na memória do computador. O processador pode então executar as instruções dos algoritmos executando as operações mais básicas do hardware.

Todos os programas que estão armazenados na memória para que possam ser executados posteriormente são chamados de **software**. Um programa armazenado na memória do computador deve ser representado em dígitos binários, que também é conhecido como código de máquina. Carregando o código da máquina na memória do computador, um dígito por vez seria uma tarefa tediosa e propensa a erros para seres humanos. Seria conveniente se pudéssemos automatizar esse processo para recuperá-lo sempre. Por esse motivo, cientistas da computação desenvolveram outro programa, chamado de **loader**, para executar esta tarefa. Um **loader** leva um conjunto de instruções de

linguagem da máquina como entrada e as carrega nas posições de memória apropriadas. Quando o processo estiver concluído, o programa em linguagem de máquina está pronto para ser executado. Obviamente, o **loader** não pode se carregar na memória, então esse é um desses algoritmos que devem ser “gravados” na ROM do computador.

Agora que existe um **loader**, podemos carregar e executar outros programas que facilitam o desenvolvimento, a execução e o gerenciamento de programas. Este tipo de software é chamado de software de sistema. O exemplo mais importante do software de sistema é o **sistema operacional** de um computador. Você provavelmente já está familiarizado com pelo menos um dos sistemas operacionais mais populares, como Linux, macOS da Apple e Microsoft Windows. Um sistema operacional é responsável por gerenciar e agendar vários programas para executarem simultaneamente. Ele também gerencia a memória do computador, incluindo o armazenamento externo, e gerencia as comunicações entre a CPU, os dispositivos de entrada e saída e outros computadores em uma rede. Uma parte importante de qualquer sistema operacional é o seu sistema de arquivos, que permite a nós, usuários humanos, organizar seus dados e programas em armazenamento permanente. Outra função importante de um sistema operacional é fornecer interfaces de usuário - ou seja, maneiras para o usuário interagir com o software do computador. Uma interface baseada em terminal aceita entradas de um teclado e exibe saída de texto em uma tela do monitor. A figura 2.3 mostra um exemplo de terminal no sistema operacional macOS.

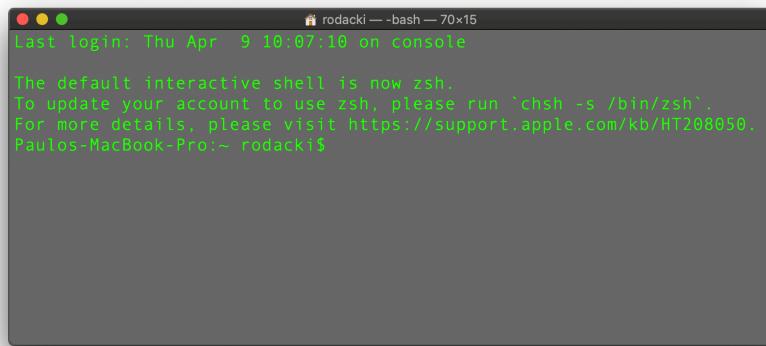


Figura 2.3: Exemplo de terminal no macOS

Uma interface gráfica moderna (GUI) organiza a tela do monitor em torno da metáfora de uma área de trabalho, com janelas contendo ícones para pastas, arquivos e aplicativos. Este tipo de interface de usuário também permite ao usuário manipular imagens com um dispositivo apontador, como um mouse.

Outro grande tipo de software é chamado de software aplicativo, ou simplesmente aplicativo. Um aplicativo é um programa projetado para uma tarefa específica, como editar um documento ou exibir uma página da Web. As aplicações incluem navegadores da Web, processadores de texto, planilhas, gerenciadores de banco de dados, pacotes de design gráfico, sistemas de produção de música e jogos, entre muitos outros.

No entanto, o hardware do computador pode executar apenas instruções que estão escritas em sua forma binária e no idioma da máquina. Escrever um programa de linguagem de máquina, no entanto, seria uma tarefa extremamente tediosa e propensa a erros. Para facilitar o processo

de redação de programas de computador, cientistas da computação desenvolveram linguagens de programação de alto nível para expressar algoritmos. Essas línguas se parecem com o inglês e permitem ao autor expressar algoritmos de forma que outras pessoas possam entender.

Um programador geralmente começa escrevendo declarações de linguagem de alto nível em um editor de texto. O programador executa então outro programa chamado **tradutor** (**compilador**) para converter o código do programa de alto nível em código executável. Como é possível que um programador cometa erros gramaticais, mesmo quando escreve um código de alto nível, o tradutor verifica erros de sintaxe antes de concluir o processo de tradução. Se detectar algum desses erros, o tradutor alerta o programador por meio de mensagens de erro. O programador então tem que rever o programa. Se o processo de transferência for bem-sucedido sem um erro de sintaxe, o programa pode ser executado pelo sistema de tempo de execução (**run-time system**). O sistema de tempo de execução pode executar o programa diretamente no hardware ou executar ainda outro programa chamado **interpretador** ou **máquina virtual** para executar o programa.

No próximo capítulo vamos iniciar o estudo da programação. Para isso, inicialmente vamos tomar conhecimento de alguns conceitos básicos e fundamentais da linguagem Python.

Capítulo 3

A Linguagem de Programação Python

Guido van Rossum inventou a linguagem de programação Python no início da década de 1990. Python é uma linguagem de programação de alto nível e de propósito geral para resolver problemas em sistemas de computação modernos. A linguagem e muitas ferramentas de suporte são gratuitos e os programas Python podem ser executados em qualquer sistema operacional.

Python é uma linguagem interpretada. Neste caso, o **compilador** traduz o código python para o que chamamos de *bytecode*. O *bytecode* é um código de baixo nível que é executado em uma máquina virtual python (PVM). Quando instalamos o compilador Python em um computador, na verdade estamos instalando tanto o compilador, quanto o interpretador e a máquina virtual onde os programas em python serão executados. O programa python é também um **interpretador** pois ele pode ser usado para executar diretamente os comandos em Python.

3.1 Programas

Comumente usamos o termo “programa” para nos referir a algoritmos, aplicativos ou a algum processo computacional. De qualquer forma, podemos elencar as seguintes informações a respeito de programas Python:

- um programa é uma seqüência de definições e comandos
 - definições são avaliadas
 - comandos são executados pelo interpretador Python em um *shell* (ou terminal)
- comandos (instruções) instruem o interpretador para fazer algo
- o programa Python pode ser digitado diretamente em um shell ou armazenado em um arquivo que é lido no shell e executado.

3.2 Objetos e tipos de dados

Os programas em Python manipulam objetos de dados. Os objetos têm um tipo (**class**) que define os tipos de operações que os programas podem fazer com eles. Os objetos são:

- escalares (não podem ser subdivididos). Ex.: int, float, bool, NoneType
- não escalares (têm uma estrutura interna que pode ser acessada). Ex: list, set, dict

Os primeiros tipos de dados (ou classes de objetos) que vamos estudar em Python são os escalares `int`, e `float`, que servem para manipular dados numéricos, o tipo `string` que representa informação textual e o tipo lógico, que pode assumir somente dois valores possíveis: `True` (verdadeiro) e `False` (falso). O quadro 3.1 mostra maiores detalhes sobre eles:

Quadro 3.1: Tipos de dados básicos em Python

tipo	nome python	conteúdo	exemplos
inteiro	<code>int</code>	números inteiros positivos e negativos	-5, 0, 43
real	<code>float</code>	números com ponto flutuante	3.14, -1.15, 43.0
lógico	<code>bool</code>	valores lógicos verdadeiro e falso	<code>True</code> , <code>False</code>
string	<code>str</code>	texto (cadeias de caracteres)	‘Bom dia’

Enquanto um tipo escalar representa um único dado, os tipos não escalares em Python servem para agrupar coleções de dados mantendo determinadas estruturas de organização da informação. Tipos não escalares serão tratados nos capítulos 9, 12 e 13.

3.3 Expressões

Python combina objetos e operadores para formar expressões. Uma expressão tem um valor e um tipo associado a ele.

Sintaxe para uma expressão simples é definida da seguinte forma:

`<objeto> <operador> <objeto>`

Podemos ter diferentes tipos de expressões. Expressões aritméticas executam operações matemáticas e produzem resultados numéricos. Expressões relacionais avaliam a relação entre objetos (por exemplo igualdade entre dois objetos) e produzem resultados lógicos. Expressões lógicas combinam diferentes expressões relacionais e produzem resultados lógicos.

3.4 Variáveis e Valores

Em programação, uma variável representa um espaço de memória que guarda uma valor ou objeto. Variáveis são criadas pelos programadores e possuem identificadores únicos. Podemos considerar que o identificador é o “nome” da variável. A linguagem Python faz diferenciação de caracteres maiúsculos e minúsculos para identificadores. Por exemplo `custo` e `Custo` seriam duas variáveis diferentes (porque Python considera identificadores diferentes).

É uma prerrogativa do programador nomear as variáveis da forma que melhor lhe convier. Normalmente escolhemos nomes que sejam representativos do significado daquele dado dentro do contexto do programa. Uma variável que armazenaria o resultado do cálculo da conta em uma caixa registradora poderia ser chamada de `total`, `valor_total` ou `total_compra`, por exemplo. Isto faria mais sentido do que chamar a variável de `a`, `xx` ou `teste`.

Existem algumas regras para criação de identificadores válidos. São elas:

- podem ser combinações de letras minúsculas (a – z) ou maiúsculas (A – Z) ou algarismos (0 a 9) ou underscore (_). Exemplos: `minhaNota`, `valor_do_desconto`, `x1`, `var_2`;
- o identificador não pode começar com um dígito. Exemplo: `1variavel1` é um identificador inválido; `variavel1` é válido;
- palavras reservadas da linguagem não podem ser identificadores. Por exemplo: `True` e `False` não são identificadores válidos;
- não podemos usar caracteres especiais (@,#,\$,%), etc);
- não pode conter espaços em branco;
- pode ser de qualquer tamanho.

O sinal de igual (=) é usado para fazer atribuição (*assignment* em inglês) do valor de uma expressão para um nome de variável (também chamado de vinculação ou *binding*). Ex:

- O valor é armazenado na memória do computador
- Uma atribuição vincula o nome ao valor
- Pode-se recuperar o valor associado ao nome ou variável invocando o nome.

Exemplo:

```
pi = 3.14
radius = 2.2
area = pi * (radius ** 2)
print(area)
radius = radius + 1
print(area)
```

Saída:

```
15.197600000000003
15.197600000000003
```

Observações:

- pode-se re-ligar nomes de variáveis usando novas instruções de atribuição
- o valor anterior ainda pode estar armazenado na memória, mas perdeu-se a vinculação com ele
- note que o valor de `area` (valor vinculado ao nome `area`) não muda até que você diga ao computador para fazer o cálculo novamente

Nós podemos usar a função `type()` para ver o tipo do objeto a qual o nome está vinculado no momento. Ex:

```
type(radius)
```

Saída: <class 'float'>

3.5 Abstração de Expressões

Por que dar nomes a valores de expressões?

- reutilizar nomes em vez de valores;
- mais fácil para mudar o código mais tarde.

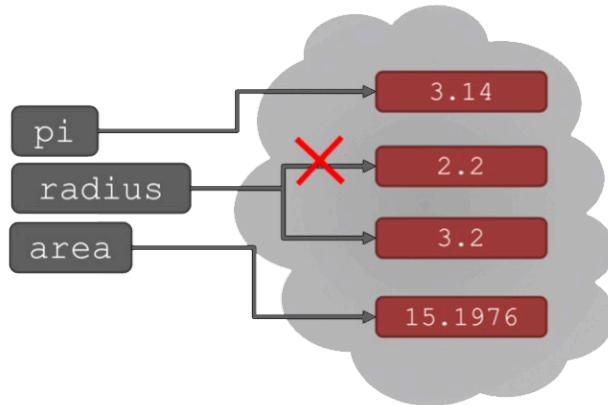


Figura 3.1: Variáveis

3.5.1 Exercícios

1. Qual o valor armazenado na variável *a* no fim do programa?

```
d = 3
c = 2
b = 4
d = c + b
a = d + 1
a = a + 1
print(a)
```

2. Você sabe dizer qual erro existe neste programa?

```
d = 3.0
c = 2.5
b = 4
d = b + 90
e = c * d
a = a + 1
print(a)
print(e)
```

Capítulo 4

Primeiros passos em Python

4.1 Imprimindo no console

A função **print** é a responsável por imprimir uma representação textual no console.

Exemplo 4.1: função print

```
print("Hello world!")
```

Saída

Hello world!

Estudaremos as funções em profundidade mais a frente, mas, por enquanto, precisamos aprender apenas o suficiente sobre elas para usar a função de impressão. Como todas as funções, a sintaxe para a função de impressão começa com o nome da função (que neste caso é **print**), seguida de uma lista de argumentos, incluída entre parathensis (). Nós vimos a impressão chamada com um único argumento acima, no entanto, é possível imprimir vários arguments (qualquer número, de fato), separado por vírgulas.

Exemplo 4.2: função print com vírgulas

```
print("Hello", "world!")
```

Saída

Hello world!

Observe que vários argumentos podem ser passados para a função de impressão, separados por vírgulas. Um único espaço é colocado entre cada argumento na saída. Você pode redefinir o caracter que é colocado entre cada argumento. No exemplo abaixo foi usado o caracter nulo :

Exemplo 4.3: função print com separador

```
print("Book", "mark", sep="")
```

Saída

Bookmark

Ao final a função print irá imprimir o caracter de controle de mudança de linha “\n”. Portanto, para imprimir uma linha em branco, basta chamar a função print() sem argumentos. Você pode ainda redefinir o caracter de término da função print. No exemplo abaixo é usado um espaço apenas:

Exemplo 4.4: redefinindo o caracter de término do print

```
print('You got', end=" ")
print('it!')
```

Saída

You got it!

4.2 Variáveis e Tipos

- **int** - inteiros: ..., -3, -2, -1, 0, 1, 2, 3, ...

```
x = 5
y = -3
print(x, type(x))
print(y, type(y))
```

Saída

5 <class 'int'>
-3 <class 'int'>

- **float** - números reais (de ponto flutuante), frações de decimal: -3.2, 1.5, 1e-8, 3.2e5

```
x = 5.0
y = -3.2
z = 2.2e6
print(x, type(x))
print(z, type(z))
```

Saída

5.0 <class 'float'>
2200000.0 <class 'float'>

- **str** - Strings (cadeia de caracteres)

- letras, caracteres especiais, espaços, dígitos
- entre aspas duplas ou aspas simples. Ex.: “IFC”, ‘python’

```
x = "IFC - Blumenau"
y = 'I love python'
print(x, type(x))
print(y, type(y))
```

Saída

```
IFC - Blumenau <class 'str'>
I love python <class 'str'>

print(type(4), type(4.0), type("4"))
```

Saída

<class 'int'> <class 'float'> <class 'str'>

- **bool** - Valores Booleanos: Verdadeiro (True) e Falso (False)

```
i_love_python = True
python_loves_me = False
print(i_love_python, type(i_love_python))
print(python_loves_me, type(python_loves_me))
```

Saída

```
True <class 'bool'>
False <class 'bool'>
```

4.3 Operadores

4.3.1 Operadores Aritméticos

- Adição:

```
4 + 6
```

Saída: 10

```
x = 5
4 + x
```

Saída: 9

```
x = 4.0 + 5
print(x, type(x))
```

Saída: 9.0 <class 'float'>

- Subtração:

```
x = 9
x - 3
```

Saída: 6

- Multiplicação:

```
x = 9
x * 3
```

Saída: 27

- Divisão em ponto flutuante (/):

trata-se da divisão comum, cujo resultado é um valor do tipo ponto flutuante.

```
10 / 3
```

Saída: 3.333333333333335

- Divisão floor (//)

A divisão floor arredonda o resultado para o valor do maior inteiro que é menor que o quociente da divisão. Observe a figura 4.1 para melhor compreender as operações mostradas a seguir

```
7.0//2      # 7.0/2 = 3.5: arredonda para 3.0, pois 3.0 < 3.5 < 4.0
-7.0//2    # -7.0/2 = -3.5: arredonda para -4.0, pois -4.0<-3.5<-3.0
-7.0//-2   # -7.0/-2 = 3.5: arredonda para 3.0, pois 3.0 < 3.5 < 4.0
-7// -2    # -7/-2 = 3.5: arredonda para 3, pois 3 < 3.5 < 4 (inteiros)
```

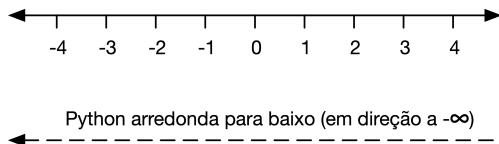


Figura 4.1: Divisão floor

- Potência:

```
2 ** 3, 2 ** 3.0, 3 ** 2
```

Saída: (8, 8.0, 9)

- Módulo (resto) da divisão inteira:

```
10 % 3
```

Saída: 1

4.3.2 Precedência de Operadores

Precedência é a ordem na qual os operadores serão avaliados quando o programa for executado. Em Python, os operadores são avaliados na seguinte ordem:

- $**$
- $*$, $/$, $//$, na ordem em que aparecerem na expressão.
- $\%$
- $+$, $-$, na ordem em que aparecerem na expressão.

Pode-se controlar a ordem com que as expressões são avaliadas com o uso de parênteses.

(expressão) também é uma expressão, que calcula o resultado da expressão dentro dos parênteses, para só então calcular o resultado das outras expressões:

```
print(5 + 10 % 3)
print((5 + 10) % 3)
```

Saída:

```
6
0
```

Você pode usar quantos parênteses desejar dentro de uma expressão. Procure usar sempre parênteses em expressões para deixar claro em qual ordem a mesma é avaliada!

4.3.3 Exercícios

1: Como você acha que o python avaliará as expressões abaixo? É o resultado que você esperava?

```
print(8 / 2 + 5 * 3)
print(5 + 10 % 3)
print(5 * 10 % 3)
```

2: O que acontece se deixarmos de fora um operando? Por exemplo, se digitarmos $3 + * 5$ em vez de $3 + 4 * 5$?

4.3.4 Operadores que atuam sob Strings

- Podemos concatenar strings usando o operador +:

```
"Hello" + " World"
```

Saída: 'Hello World'

- Podemos “n-plicar” Strings usando operador * (star):

```
"Bye" * 2
```

Saída: 'ByeBye'

- Strings vs. números:

```
4 + 5
```

Saída: 9

```
"4" + "5"
```

Saída: '45'

```
"4" + 5
```

Saída

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

```
4 + "5"
```

Saída

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

4.3.5 Comparações

- Menor que

```
5 < 4
```

Saída: False

- Maior que

```
5 > 4
```

Saída: True

- Maior ou igual

```
5 >= 4
```

Saída: True

```
4 >= 4
```

Saída: True

- Menor ou igual

```
4 <= 3
```

Saída: False

- Igual

```
5 == 4
```

Saída: False

```
5 == 5.0
```

Saída: True

```
5 == "5"
```

Saída: False

```
2 + 2 == 4
```

Saída: True

- Diferente (“not equal”)

```
5 != 4
```

Saída: True

- Comparando strings

```
x = "abc"  
y = 'abc'  
x == y
```

Saída: True

```
"abd" > "abc"
```

Saída: True

```
'abc' > 'abcd'
```

Saída: False

- Erro comum: inverter os caracteres dos operadores

```
2 => 3
```

Saída

```
File "<stdin>", line 1  
 2 => 3  
      ^
```

```
SyntaxError: invalid syntax
```

4.3.6 Operadores Lógicos

- negação (not):

```
print(not True)
a = 2 == 5
print(not a)
```

Saída

```
False
True
```

- e (and):

```
True and True
True and False
False and False
```

Saída

```
True
False
False
```

- ou (or):

```
True or true
True or False
```

Saída

```
True
True
```

4.4 Conversões de tipo (type cast)

Use as funções `int()`, `float()`, and `str()` para converter entre tipos (nós iremos falar sobre *funções* mais à frente):

```
x = "6"
print(x, type(x))
x = int("6")
print(x, type(x))
print(float("1.25"))
course = "intro" + str(2) + "cs"
print(course)
print("intro", 2, "cs", sep="")
```

Saída

```
6 <class 'str'>
6 <class 'int'>
1.25
intro2cs
intro2cs
```

4.4.1 Exercício:

3: Compute o valor do polinomio $y = ax^2 + bx + c$ para $x = -2$, $x = 0$, e $x = 2$ usando $a = 1$, $b = 1$, $c = -6$.

4.5 Entrada de dados via teclado

Usamos a função `input` para ler dados inseridos pelo usuário:

- Imprime a string no console
- O usuário escreve algo e pressiona a tecla <enter>
- a função então vincula este valor a uma variável. Ex:

```
text = input("Type anything... ")
print(5*text)
```

Execução

```
Type anything... teste
testetestetestetestetesteste
```

A função `input` fornece uma string. Portanto devemos fazer uma conversão de tipo (ou “type cast”) se quisermos trabalhar com números. Ex:

```
num = int(input("Type a number... "))
print(5*num)
```

Saída

```
Type a number... 10
50
```

4.6 Strings literais com formatação

Um literal de cadeia de caracteres formatada ou f-string é um string literal que é prefixado com ‘f’ ou ‘F’. Essas strings podem conter campos de substituição, que são expressões delimitadas por chaves { e }. Enquanto outros strings literais sempre têm um valor constante, as strings formatadas são realmente expressões avaliadas em tempo de execução. Exemplos:

```
name = "Paulo"
idade = 50
f'Seu Professor se chama {name} e tem {idade} anos de idade'
```

Saída

```
'Seu Professor se chama Paulo e tem 50 anos de idade'
```

Um uso interessante de literais com formatação é a possibilidade de definir, por exemplo, a precisão de números de ponto flutuante na saída do console, ou ainda a formatação que você deseja usar para uma data do calendário. Exemplos:

```
from math import pi
import datetime

print(pi)

width = 8
precision = 5
print(f'{pi:{width}.{precision}}')
print(f'{pi:.5}')

today = datetime.date(year=2018, month=3, day=1)
print("Data no formato Americano: ", today)
print(f"Data no formato Brasileiro: {today:%d/%m/%Y}")
```

Saída

```
3.141592653589793
3.1416
3.1416
Data no formato Americano: 2020-03-15
Data no formato Brasileiro: 15/03/2020
```

4.7 A função format

A função **format** converte o valor de um objeto para uma representação textual “formatada”, seguindo a especificação fornecida. A sintaxe e uso da função é o seguinte:

```
format(value[, spec])
print(format(value, spec))
```

Existe toda uma mini-linguagem que descreve a especificação da formatação para diferentes tipos de objetos aqui. Eis alguns exemplos de formatação para os objetos escalares **int** e **float** que vimos até agora:

```
pi = 3.141592653589793
# imprime um float com até 6 casas decimais (default)
print(format(pi, "f"))
# imprime um float truncando em 2 casas decimais
print(format(pi, ".2f"))
# float com 6 digitos e 2 casas decimais.
# Preenche de espaço a parte + significativa
print(format(pi, "6.2f"))
# imprime um int na sua representação decimal
print(format(123, "d"))
# idem, na sua representação hexadecimal
print(format(123, "x"))
```

Saída

```
3.141593
3.14
3.14
123
7b
```


Capítulo 5

Controle de Fluxo - Condicionais

5.1 Indentação do Código

Os programas Python são estruturados através de indentação, ou seja, os blocos de código são definidos pelo seu recuo. Ok, é o que esperamos de qualquer código de programa, não é? Sim, mas no caso de Python é um requisito de linguagem, não é uma questão de estilo. Este princípio facilita a leitura e a compreensão do código Python de outras pessoas.

Então, como isso funciona? Todas as declarações com a mesma distância à direita pertencem ao mesmo bloco de código, ou seja, as instruções dentro de uma linha de bloco verticalmente. O bloco termina em uma linha menos recuada ou no final do arquivo. Se um bloco tiver de ser mais profundamente aninhado, é simplesmente recuado mais para a direita. A figura 5.1 mostra esquematicamente como são organizados os blocos de código em Python por meio de indentação:



Figura 5.1: Esquema de indentação de blocos de código

Há outro aspecto da estruturação em Python, que vocês verão nos exemplos de declarações condicionais a seguir e de laços, mais a frente. As sentenças que iniciam o laço ou a declaração condicional terminam com dois pontos “:” – o mesmo é verdadeiro para funções e outras estruturas que introduzem blocos. Então, devemos dizer que as estruturas Python são definidas por dois pontos e indentação.

5.2 Sentenças Condicionais

É muito comum em um programa que certos conjuntos de instruções sejam executados de forma condicional, em casos como validar entradas de dados.

Sintaxe:

```
if <condição>:  
    <bloco de código>  
elif <condição>:          # O ou mais cláusulas elif  
    <bloco de código>  
else:                  # opcional  
    <bloco de código>
```

Na qual:

- <condição>: sentença que possa ser avaliada como verdadeira ou falsa.
- <bloco de código>: sequência de linhas de comando.
- As cláusulas **elif** e **else** são opcionais e podem existir vários **elifs** para o mesmo **if**, porém apenas um **else** ao final.

Exemplo:

```
x = int(input("Digite um valor para x: "))  
y = int(input("Digite um valor para y: "))  
if x == y:  
    print("x e y são iguais")  
elif x < y:  
    print("x é menor")  
else:  
    print("y é menor")  
print("Obrigado!")
```

A figura 5.2 mostra o fluxograma de execução do exemplo anterior.

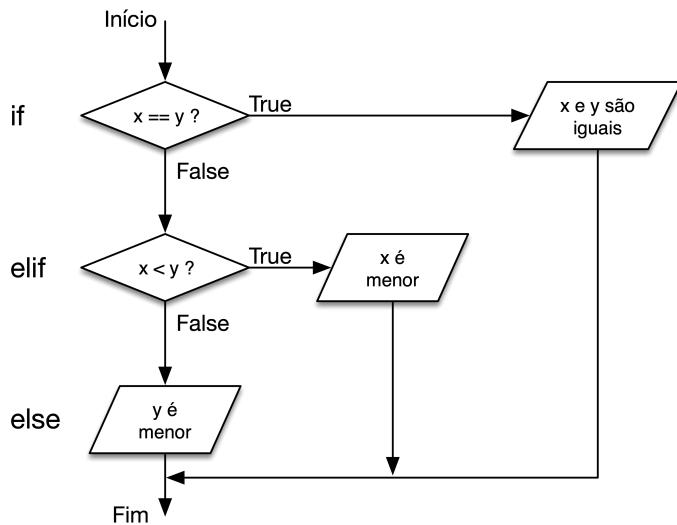


Figura 5.2: Fluxograma do exemplo de comando condicional

Um exemplo um pouco mais elaborado:

```
from datetime import date
import calendar

today = calendar.day_name[date.today().weekday()]
strike = "N"
my_lab = "Friday"

if today == "Tuesday":
    if strike == "Y":
        print("Stay home")
    else:
        print("Lecture in IFC!")
elif today == "Thursday":
    print("Another lecture in IFC!")
elif today == my_lab:
    print("Go to lab!")
elif today == "Monday" or today == "Friday" or today == "Saturday":
    print("no lecture in IFC")
else:
    print("playday!")
```

Se o bloco de código for composto de apenas uma linha, ele pode ser escrito após os dois pontos.
Exemplo:

```
temperatura = 40
if temperatura > 30: print("Derretendo...!")
```

5.2.1 Exercícios:

1. No trecho de código abaixo, quando o **comando_4** é executado?

```
if cond1:
    if cond2:
        comando_1
    else:
        comando_2
else:
    if cond3:
        comando_3
    else:
        comando_4
```

2. O que será impresso no código abaixo?

```
a = 9
if a > 3:
    if a < 7:
        print("a")
else:
    if a > -10:
        print("b")
    else:
        print("c")
```

```
print("finish!")
```

5.3 Expressões condicionais

A linguagem Python suporta a seguinte expressão:

```
<variável> = <valor_1> if <condição> else <valor_2>
```

Na qual `<variável>` receberá `<valor_1>` se `<condição>` for verdadeira e `<valor_2>` caso contrário. Por exemplo:

```
Weekend = True if (today == "Saturday" or today == "Sunday") else False
print(Weekend)
```

5.3.1 Valores True e False

Todo valor em python pode ser avaliado como True (Verdadeiro) ou False (Falso). A regra geral é que qualquer valor não-zero ou não vazio irá avaliar para Verdadeiro. Se você nunca tiver certeza, você pode abrir um terminal Python e escrever duas linhas para descobrir se o valor que você está considerando é True ou False. Dê uma olhada nos seguintes exemplos, mantenha-os em mente e teste qualquer valor que você tenha curiosidade.

```
if 0:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")

if 1:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")

# Test for an empty string
if '':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")

# any other string, including a space
if ' ':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

5.3.2 Exemplos

Exemplo 1

Escrever um programa que calcula a área de três tipos de objetos geométricos: quadrado, retângulo e círculo.

- Primeiramente deve ser lido um caractere que indica o tipo de objeto a ter a área calculada: 'q' para quadrado, 'r' para retângulo e 'c' para círculo.
- Em seguida deverá ser lido as dimensões do objeto:
 - Para um quadrado deve ser lido o tamanho de um lado.
 - Para um retângulo devem ser lidos os tamanhos de cada lado.
 - Para um círculo, deve ser lido o raio.
- Em seguida o programa faz o cálculo da área, imprime no terminal e finaliza o programa. Se o usuário digitar um caractere diferente de 'q', 'r', e 'c' o programa deverá imprimir uma mensagem de erro antes de finalizar o programa.

```
from math import pi

g = input("selecione o objeto - q, r, ou c: ")
entrada_valida = True
if g == "q":
    lado = float(input("forneça o lado do quadrado: "))
    area = lado * lado
    objeto = "quadrado"
elif g == "r":
    a = float(input("forneça o lado a do retangulo: "))
    b = float(input("forneça o lado b do retangulo: "))
    area = a * b
    objeto = "retangulo"
elif g == "c":
    r = float(input("forneça o raio do circulo: "))
    area = pi * r ** 2
    objeto = "circulo"
else:
    print("Entrada inválida!")
    entrada_valida = False
if entrada_valida:
    print(f"A área do {objeto} é {area}")
```

Exemplo 2

Quando ações são vendidas ou compradas por meio de um corretor, a comissão do corretor é muitas vezes calculada usando uma escala que depende do valor das ações negociadas. Escreva um programa que calcule o valor da comissão a partir do valor da transação informado pelo usuário, sabendo-se que o corretor cobra os valores indicados abaixo e que a comissão mínima é de R\$ 39,00.

- Até R\$ 2.500,00, comissão de R\$ 30,00 + 1,70%
- de R\$ 2.500,01 até R\$ 6.250,00, comissão de R\$ 56,00 + 0,66%
- de R\$ 6.250,01 até R\$ 20.000,00, comissão de R\$ 76,00 + 0,34%
- de R\$ 20.000,01 até R\$ 50.000,00, comissão de R\$ 100,00 + 0,22%

- de R\$ 50.000,01 até R\$ 500.000,00, comissão de R\$ 155,00 + 0,11%
- Acima de R\$ 500.000,00, comissão de R\$ 255,00 + 0,09%

```
# Cálculo de Comissão por volume de Vendas
venda = input("Forneça o valor total das vendas")
if not venda.isdecimal():
    print("Valor fornecido inválido")
else:
    if venda <= 2500:
        c = 30 + 0.017 * venda
        if c <= 39:
            c = 39
    elif venda <= 6250:
        c = 56 + 0.0066 * venda
    elif venda <= 20000:
        c = 76 + 0.0034 * venda
    elif venda <= 50000:
        c = 100 + 0.0022 * venda
    elif venda <= 500000:
        c = 155 + 0.0011 * venda
    else:
        c = 255 + 0.0009 * venda
    print(f"Sua comissão foi: R$ {c}")
```

5.3.3 Exercícios

Exercício 1

A figura 5.3 mostra uma circunferência de raio r com centro na origem do sistema de coordenadas, cuja equação é dada por $x^2+y^2 = r^2$. Escreva um algoritmo que, dado o raio r de uma circunferência e as coordenadas x_p e y_p de um ponto p qualquer, verifique se ele se encontra na circunferência, no seu interior ou no seu exterior.

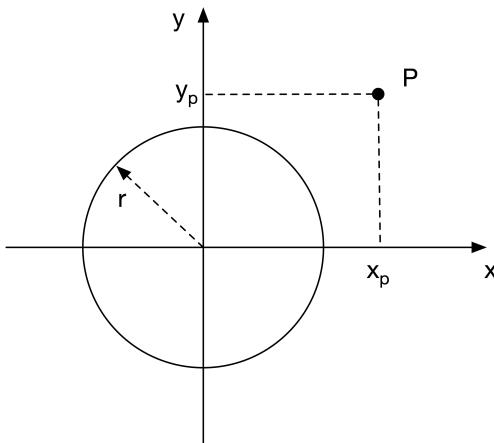


Figura 5.3: Circunferência com centro na origem do sistema de coordenadas

Exercício 2

Uma lata de leite em pó da marca A, com 400g custa R\$ 8,39. Um saco de leite em pó da marca B, com 1Kg custa R\$ 20,30. Qual a marca tem o melhor preço? Escreva um programa python que resolva este problema e mostre a resposta.

Exercício 3

Escreva um programa Python para resolver equações do 2º grau ($ax^2 + bx + c = 0$). Dados os coeficientes a, b, c, determine se há raízes reais e, caso positivo, quais são elas.

Exercício 4: Pedra, Papel e Tesoura

Escreva um programa que simula o jogo conhecido como “Pedra, Papel e Tesoura” de um jogador contra o computador. A figura 5.4 ilustra os lances que podem ser feitos por um jogador.

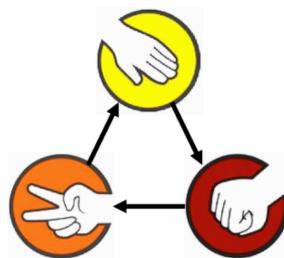


Figura 5.4: Jogo pedra, papel e tesoura

Associar objetos (pedra, papel, tesoura) a números é uma forma de **abstração**. Vamos usar essa abstração para simplificar o código do sorteio:

```
import random # Importa o módulo random
number = random.randrange(0, n) # Obtém um número aleatório entre 0 e n-1
```

Tal simplificação pode ser também aplicada ao teste da condição?

```
"""
Jogo Pedra, Papel e Tesoura
Vamos adotar a seguinte abstração:
- pedra = 0
- papel = 1
- tesoura = 2
Usaremos nos nomes:
- user para representar o jogador
- computer para o computador

Se user == computer : empate
user ganha quando:
    user == pedra e computer == tesoura, ou
    user == tesoura e computer == papel, ou
    user == papel e computer == pedra
"""

# Importa o módulo random
number = random.randrange(0, 3) # Obtém um número aleatório entre 0 e 2
```

```

    caso contrário computer ganha
"""
import random
pedra = 0
papel = 1
tesoura = 2
computer = random.randrange(0, 3)
user = int(input("Digite 0 p/ pedra, 1 p/ papel, ou 2 p/ tesoura: "))
if user == computer:
    print("It's a draw, try again")
elif (user == pedra and computer == tesoura) or \
    (user == tesoura and computer == papel) or \
    (user == papel and computer == pedra):
    print("You Win!")
else:
    print("Computer Wins!")

```

Escreva a solução usando aritmética com cálculo de resto da divisão de números inteiros. Veja a dica mostrada na figura 5.5:

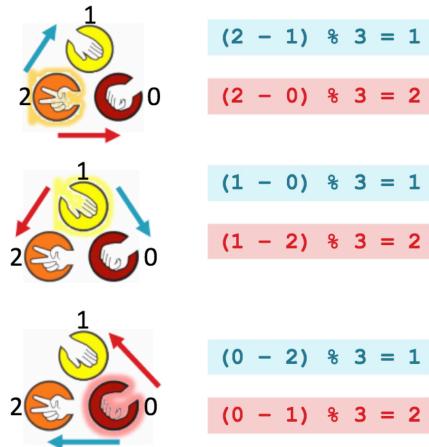


Figura 5.5: Dica para jogo pedra, papel e tesoura

Capítulo 6

Controle de Fluxo - Iterações

Até agora vimos como escrever programas capazes de executar comandos de forma linear, e, se necessário, tomar decisões com relação a executar ou não um bloco de comandos. Entretanto, eventualmente faz-se necessário executar um bloco de comandos várias vezes para obter o resultado esperado. A execução repetida de um conjunto de instruções é chamada de iteração. Python tem duas instruções para iteração - a declaração **for**, e a instrução **while**.

6.1 Laços while

Executa um bloco de código atendendo a uma condição.

Sintaxe:

```
while <condição>:  
    <bloco de código>  
    continue  
    break  
else:  
    <bloco de código>
```

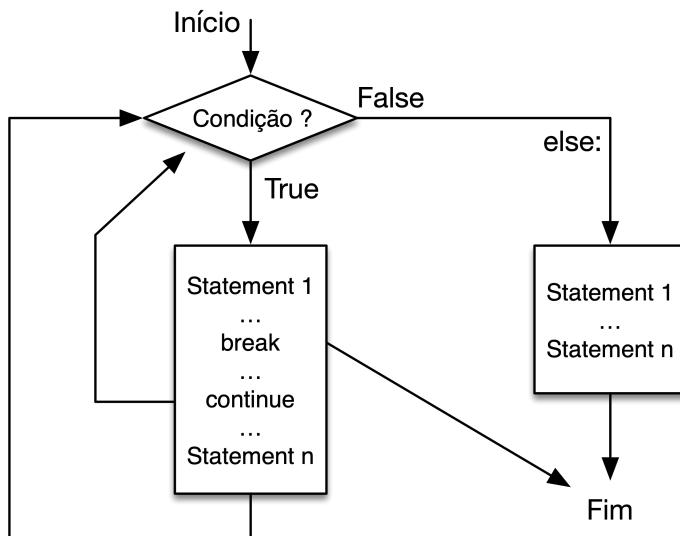
O bloco de código dentro do laço **while** é repetido enquanto a condição do laço estiver sendo avaliada como verdadeira. A figura 6.1 mostra o fluxograma dos laços de repetição com **while** em Python.

Exemplo: Calcule quantas vezes o dígito 0 aparece no número inteiro calculado abaixo.

```
num = 2**100  
print(num)
```

Saída: 1267650600228229401496703205376

```
count = 0  
  
while num > 0:      # 0 que acontece se mudarmos a condição para >=0?  
    if num % 10 == 0:  
        count = count + 1  
    num = num // 10
```

Figura 6.1: Fluxograma do comando `while`

```
print(count)
```

Saída: 6

Exemplo: Jogo de Adivinhação

```

import random           # Import the random module

number = random.randrange(1, 101) # Random number between 1 and 100
guesses = 0
guess = int(input("Adivinhe meu número entre 1 e 100:"))

while guess != number:
    guesses += 1
    if guess > number:
        print(guess, "está acima.")
    elif guess < number:
        print(guess, "está abaixo.")
    guess = int(input("tente novamente:"))

print("\nÓtimo, você acertou em", guesses, "tentativas!")
  
```

6.1.1 Exercícios - `while`

1. Repita o Jogo de adivinhação dando a opção do jogador de desistir, por exemplo, escolhendo o número 0. **Dica:** Use os comandos `break` e `else` dos laços `while`.
2. O que acontece se a condição no comando `while` for falsa na primeira vez?
3. O que acontece se a condição no comando `while` for sempre verdadeira?
4. Supondo que a população de um país A seja da ordem de 80000 habitantes com uma taxa anual de crescimento de 3% e que a população de B seja 200000 habitantes com uma taxa de crescimento de 1.5%. Faça um programa que calcule e escreva o número de anos necessários

para que a população do país A ultrapasse ou iguale a população do país B, mantidas as taxas de crescimento.

5. Altere o programa anterior permitindo ao usuário informar as populações e as taxas de crescimento iniciais. Valide a entrada e permita repetir a operação.

6.2 Laços for

É a estrutura de repetição mais usada no Python. A instrução aceita não só sequências estáticas, mas também sequências geradas por iteradores. Iteradores são estruturas que permitem iterações, ou seja, acesso aos itens de uma coleção de elementos, de forma sequencial. Nós estudaremos iteradores mais à frente.

Durante a execução de um laço *for*, a variável aponta para um elemento da sequência. A cada iteração, a variável é atualizada, para que o bloco de código do *for* processe o elemento correspondente.

As cláusula opcional *break* interrompe o laço e a cláusula opcional *continue* passa para a próxima iteração. O código dentro do *else* (opcional) é executado ao final do laço, a não ser que o laço tenha sido interrompido por *break*.

Sintaxe:

```
for <variable> in <sequência>:
    <bloco de código>
    continue
    break
else:
    <bloco de código>
```

A figura 6.2 mostra o fluxograma dos laços de repetição com o comando **for** em Python.

Exemplo: Vamos resolver o mesmo problema anterior usando o tipo **str** ao invés de **int**.

```
num = 2**100
count = 0
for digit in str(num):
    #print(digit, type(digit))
    if digit == "0":
        count = count + 1

print(count)
```

- a título de curiosidade a solução mais eficiente em python é usando funções builtin de str:

```
num = 2**100
count = str.count(str(num), "0")
print(count)
```

6.2.1 A função range

A função **range()** retorna uma série numérica no intervalo enviado como argumento. A série retornada é um objeto iterável tipo **range** e os elementos contidos serão gerados sob demanda. É

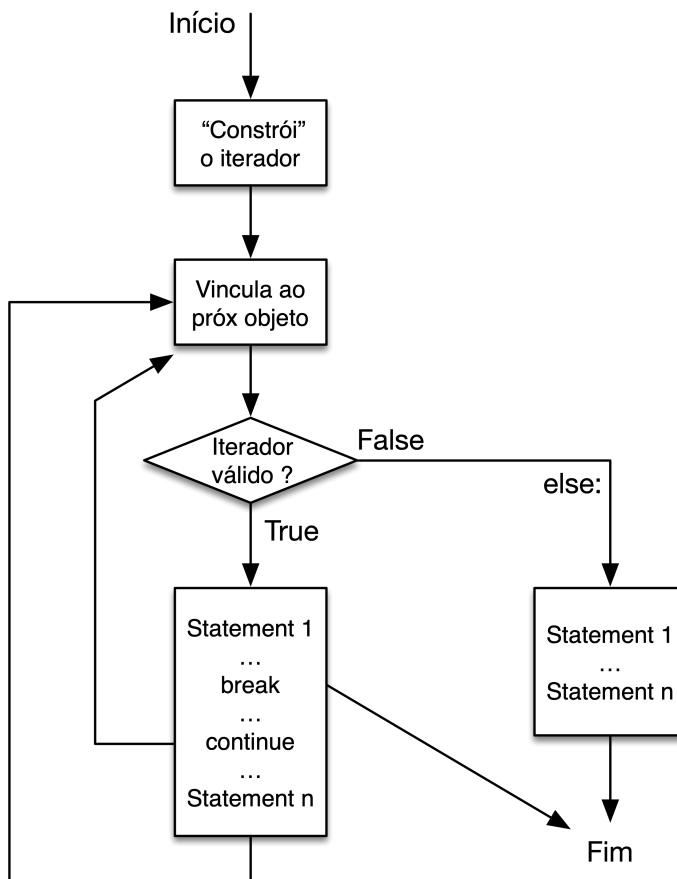


Figura 6.2: Fluxograma do comando for

comum o uso da função `range()` com laços for.

```
range( stop )                      # definição simplificada
range( [start], stop[, step] )      # definição completa
```

A função `range()` exige a definição do último elemento da sequência numérica. Por padrão, o parâmetro start será igual a 0 e o step igual a 1:

- start - valor onde o intervalo deve começar
- stop - valor (menos um) onde o intervalo deve finalizar
- step - passo (intervalo entre os elementos da sequência)

Note que o parâmetro stop possui o intervalo aberto, isto é, o número definido não estará contido na sequência numérica:

```
for num in range(5):
    print(num, end=" ")
```

Saída: 0 1 2 3 4

Exemplo: A soma dos números ímpares no intervalo de 0 a 100, usando uma variável **acumuladora**.

```
soma = 0
for impar in range(1, 100, 2):
    soma = soma + impar
```

```
print("A soma dos ímpares até 100 é: ", soma)
```

Saída: A soma dos ímpares até 100 é: 2500

6.2.2 Exercícios - for

1. Leia um inteiro positivo n , e imprima as potências: $2^0, 2^1, \dots, 2^n$. Dica: Use uma variável acumuladora que no início da i -ésima iteração de um laço, possui o valor 2^i . Imprima este valor e atualize a variável acumuladora para a próxima iteração, multiplicando esta variável por 2.
2. Faça um programa que mostre os n termos da Série a seguir:

$$S = 1/1 + 2/3 + 3/5 + 4/7 + 5/9 + \dots + n/m.$$
 Imprima no final a soma da série.

6.3 Escolhendo entre for e while

Use um laço **for**, se você souber, antes de iniciar o laço, o número máximo de vezes que você precisará executar o corpo do laço. Por exemplo, se você estiver percorrendo uma lista de elementos, você sabe que o número máximo de iterações do laço que você pode precisar é “todos os elementos da lista”. Problemas como: “itere este modelo de previsão de tempo para 1000 ciclos”, ou “pesquise esta lista de palavras”, ou ainda, “encontre todos os números primos até 10000”, sugerem que um laço **for** é o mais adequado.

Em contrapartida, se você precisar repetir alguma computação até que alguma condição seja atendida, e você não pode calcular antecipadamente quando isso acontecerá, como fizemos no “programa de advinhação”, você precisará de um laço **while**.

Chamamos de “iteração definida” o primeiro caso – temos alguns limites definidos para o que é necessário. O último caso é chamado de “iteração indefinida” – não temos certeza de quantas iterações precisamos – nem podemos estabelecer um limite superior!

6.3.1 Exercícios

1. **Potenciação:** Faça um programa que lê dois números inteiros positivos a e b . Utilizando laços (i.e., não utilizar $**$), o seu programa deve calcular e imprimir o valor a^b .
2. **A sequência** $3n+1$: também conhecida como Conjectura Collatz, esta sequência tem fascinado matemáticos por vários anos. A regra para criar a sequência é começar com um dado n , e gerar o próximo termo da seqüência a partir de n da seguinte forma: se n for par dividimos n por 2, ou senão, multiplicamos n por 3 e somamos 1 quando n for ímpar. A sequência termina quando n se tornar 1.

6.4 Variáveis Indicadoras

Um uso comum de laços é para a verificação se um determinado objeto, ou conjunto de objetos, satisfaz uma propriedade ou não. Um padrão que pode ser útil na resolução deste tipo de problema

é o uso de uma variável **indicadora**:

- Assumimos que o objeto satisfaz a propriedade (`indicadora = True`).
- Com um laço verificamos se o objeto realmente satisfaz a propriedade.
- Se em alguma iteração descobrirmos que o objeto não satisfaz a propriedade, então fazemos (`indicadora = False`).

Vamos usar este padrão na determinação se um número n é primo ou não.

- Um número é primo se seus únicos divisores são 1 e ele mesmo.

Dado um número n como detectar se este é ou não primo?

1. Leia o número n .
2. Faça a variável `indicadora = True`, assumindo que n é primo.
3. Teste se nenhum dos números entre 2 e $n - 1$ divide n . Lembre-se que o operador `%` retorna o resto da divisão. Portanto a é zero se e somente se b divide a .
4. Se o resto da divisão for igual a zero então faça `indicadora = False`. Com isto descobrimos que n não é primo.

6.4.1 Exercício

Escreva um programa que lê n números inteiros do teclado, e no final informa se os números lidos estão ou não em ordem crescente. Dica: Use duas variáveis, uma que guarda o número lido na iteração atual, e uma que guarda o número lido na iteração anterior. Os números estarão ordenados se a condição (`anterior <= atual`) for válida durante a leitura de todos os números.

6.5 Variáveis Contadoras

Considere ainda o uso de laços para a verificação se um determinado objeto, ou conjunto de objetos, satisfaz uma propriedade ou não. Um outro padrão que pode ser útil é o uso de uma variável **contadora**.

- Esperamos que um objeto satisfaça x vezes uma sub-propriedade. Usamos um laço e uma variável que conta o número de vezes que o objeto tem a sub-propriedade satisfeita.
- Ao terminar o laço, se a variável contadora for igual à x então o objeto satisfaz a propriedade.

Vamos revisitar o problema de determinar se um número n , inteiro, é primo ou não. Podemos usar uma variável que conta quantos números dividem n . Se o número de divisores for 0, então n é primo.

6.5.1 Exercícios

1. No exemplo dos números primos não precisamos testar todos os números entre $2, \dots, n - 1$, para verificar se dividem ou não n . Basta testarmos até $n/2$. Por que? Qual o maior divisor possível de n ?
2. Na verdade basta testarmos os números $2, \dots, \sqrt{n}$. Por que?

3. Escreva um programa Python para obter a série de Fibonacci entre 0 a 50.

Nota: A seqüência de Fibonacci é a série de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, onde cada número que segue é encontrado somando os dois números antes dele.

Capítulo 7

Números Binários, Laços Encaixados

7.1 Representações Binárias e Decimais

7.1.1 Conversão Binário-Decimal

Sabemos que um computador armazena todas as informações na representação binária. Portanto, é útil saber como converter valores binários em decimal e vice versa. Dado um número na sua representação binária $b_n b_{n-1} \dots b_2 b_1 b_0$, este corresponde na forma decimal a:

$$\sum_{i=0}^n b_i * 2^i$$

Exemplos:

$$101 = 2^2 + 2^0 = 5$$

$$1001110100 = 2^9 + 2^6 + 2^5 + 2^4 + 2^2 = 512 + 64 + 32 + 16 + 4 = 628$$

Vamos supor que lemos do teclado um inteiro na sua representação binária. Ou seja, ao lermos $n = 111$ assumimos que este é o número binário (e não cento e onze). Como transformar este número no correspondente valor em decimal (7 neste caso)?

Basta usarmos a expressão:

$$\sum_{i=0}^n b_i * 2^i$$

Mas, para tal, precisamos recuperar os dígitos individuais do número.

- Note que $n \% 10$ recupera o último dígito de n .
- Note que $n // 10$ remove o último dígito de n , pois ocorre a divisão inteira por 10.

Exemplo: Com $n = 345$, ao fazermos $n \% 10$ obtemos 5. E ao fazermos $n // 10$ obtemos 34.

O programa abaixo imprime cada um dos dígitos de n separadamente:

```
n = int(input("Digite um número:"))
while n != 0:
    digito = n % 10
    print(digito, end=" ")
    n = n // 10
print()
```

Para transformar um número em binário para decimal, devemos gerar as potências $2^0, \dots, 2^n$, e multiplicar cada potência 2^i pelo i-ésimo dígito. E, finalmente, para armazenar a soma $\sum_{i=0}^n b_i * 2^i$ usamos uma variável acumuladora:

```
n = int(input("Digite um número na representação binária:"))
soma = 0
potencia = 1
while n != 0:
    digito = n % 10
    soma += potencia * digito
    potencia *= 2
    n //= 10
print("Valor em decimal é: ", soma)
```

7.1.2 Conversão Decimal-Binária

Dado um número em decimal, vamos obter o correspondente em binário. Qualquer decimal pode ser escrito como uma soma de potências de 2:

$$5 = 2^2 + 2^0$$

$$13 = 2^3 + 2^2 + 2^0$$

Nesta soma, para cada potência 2^i , sabemos que na representação em binário haverá um 1 no dígito i . Exemplo: $13 = 1101$. O que acontece se fizermos sucessivas divisões por 2 de um número decimal?

- $13/2 = 6$ com resto 1
- $6/2 = 3$ com resto 0
- $3/2 = 1$ com resto 1
- $1/2 = 0$ com resto 1

Dado um número n na representação decimal, fazemos repetidas divisões por 2, obtendo os dígitos do valor em binário:

```
n = int(input("Digite um número:"))
while n != 0:
    digito = n % 2
    n = n // 2
    print(digito, end=" ")
print()
```

7.1.3 Exercício

Na conversão decimal-binário, modifique o programa para que este obtenha o valor binário em uma variável inteira, ao invés de imprimir os dígitos separados por espaço. Dica: Suponha $n = 7$ (111 em binário), e você já computou $x = 11$. Para "inserir" o último dígito 1 em x você deve fazer $x = x + 100$. Ou seja, você precisa de uma variável acumuladora que armazena as potências de $10 : 1, 10, 100, 1000$, etc.

7.2 Laços Encaixados

Também é possível ter laços dentro dos laços. Estes são chamados de laços aninhados (ou laços encaixados). Um uso comum de laços encaixados ocorre quando para cada um dos valores de uma determinada variável, precisamos gerar/checkar algo com valores de outras variáveis.

7.2.1 Equações Lineares

Problema: Determinar todas as soluções inteiras de um sistema linear como:

$$x_1 + x_2 = C$$

com $x_1 \geq 0, x_2 \geq 0, C \geq 0$ e todos inteiros.

Uma solução possível é: para cada um dos valores de $0 \leq x_1 \leq C$, teste todos os valores de x_2 possíveis e verifique quais deles são soluções.

```
C = int(input("Digite o valor da constante C:"))
for x1 in range(C + 1):
    for x2 in range(C + 1):
        if x1 + x2 == C:
            print(x1, " + ", x2, " = ", C)
```

Note que fixado x_1 , não precisamos testar todos os valores de x_2 , pois este é determinado como $x_2 = C - x_1$.

```
C = int(input("Digite o valor da constante C:"))
for x1 in range(C + 1):
    x2 = C - x1
    print(x1, " + ", x2, " = ", C)
```

Mas em um caso geral com n variáveis, $x_1 + x_2 + \dots + x_n = C$, será preciso fixar $n - 1$ variáveis para só então determinar o valor de x_n .

Exercício: Quais são as soluções de $x_1 + x_2 + x_3 = C$ com $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, C \geq 0$ e todas inteiras?

Note que fixado x_1 , o valor máximo de x_2 é $C - x_1$. Fixados x_1 e x_2 , o valor de x_3 é determinado como $C - x_1 - x_2$. Escreva a solução em Python, levando em conta as dicas acima.

```
C = int(input("Digite o valor da constante C:"))
for x1 in range(C + 1):
    for x2 in range(C + 1 - x1):
```

```
x3 = C - x1 - x2
print(x1, " + ", x2, " + ", x3, " = ", C)
```

7.2.2 Mega-Sena

Na Mega-Sena, um jogo consiste de 6 números distintos com valores entre 1 e 60. Problema: Imprimir todos os jogos possíveis da Mega-Sena.

Solução: Partindo da mesma idéia vista anteriormente, podemos gerar todos os possíveis valores para cada um dos 6 números do jogo. Assumindo que um possível jogo é sempre apresentado com os números em ordem crescente, fixamos o valor de d_1 , assim, d_2 necessariamente é maior que d_1 . E com d_1 e d_2 fixados, d_3 é maior que d_2 , etc.

```
nSolucoes = 0
for d1 in range(1, 61):
    for d2 in range(d1 + 1, 61):
        for d3 in range(d2 + 1, 61):
            for d4 in range(d3 + 1, 61):
                for d5 in range(d4 + 1, 61):
                    for d6 in range(d5 + 1, 61):
                        nSolucoes += 1
                        if nSolucoes <= 15:
                            print(d1, d2, d3, d4, d5, d6, sep=" ")
print("Total de soluções: ", nSolucoes)
```

7.2.3 Exercícios

- Faça um programa que leia um número n e imprima n linhas na tela com o seguinte formato (exemplo se $n = 6$):

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

- Faça um programa que leia um número n e imprima n linhas na tela com o seguinte formato (exemplo se $n = 6$):

```
+ * * * *
* + * * *
* * + * *
* * * + *
* * * * +
* * * * *
```

- Escreva um programa Python para construir o seguinte padrão, usando um laço aninhado.

```
*
```

```
* * *
* * * *
* * * * *
* * * *
* * *
*
*
```

- Um jogador da Mega-Sena é supersticioso, e só faz jogos em que o primeiro número do jogo é par, o segundo é ímpar, o terceiro é par, o quarto é ímpar, o quinto é par e o sexto é ímpar. Faça um programa que imprima todas as possibilidades de jogos que este jogador supersticioso pode jogar.

Capítulo 8

Introdução a Funções

8.1 Como escrever código?

- Até agora nós cobrimos alguns mecanismos da linguagem
- Sabemos como escrever diferentes trechos de código para cada computação
- Cada código é uma seqüência de instruções
- Problemas com esta abordagem
 - fácil para problemas em pequena escala
 - complicado para problemas maiores
 - difícil de acompanhar os detalhes
 - como você sabe que a informação certa é fornecida à parte correta do código?

8.2 Exercício de motivação

O número de combinações possíveis de **m** elementos em grupos de **n** elementos ($n \leq m$) é dada pela fórmula de combinação $m! / ((m-n)!n!)$.

Escreva um programa que lê dois inteiros **m** e **n** e calcula a combinação de **m**, **n** a **n**.

```
# leitura dos valores de entrada
m = int(input("Digite m: "))
n = int(input("Digite n: "))

# calcula o fatorial de m
k = m
k_fat = 1
cont = 1
while cont < k:
    cont += 1          # o mesmo que cont = cont + 1
    k_fat *= cont     # o mesmo que k_fat = k_fat * cont

m_fatorial = k_fat

# calcula o fatorial de n
k = n
```

```

k_fat = 1
cont = 1
while cont < k:
    cont += 1      # o mesmo que cont = cont + 1
    k_fat *= cont # o mesmo que k_fat = k_fat * cont

n_fatorial = k_fat

# calcula o fatorial de m - n
k = m-n
k_fat = 1
cont = 1
while cont < k:
    cont += 1      # o mesmo que cont = cont + 1
    k_fat *= cont # o mesmo que k_fat = k_fat * cont

mn_fatorial = k_fat

print("Comb(",m,",",",",n,") = ", format(mn_fatorial/(n_fatorial * n_fatorial)
, ".0f"))

```

8.3 Decomposição e Abstração

Uma linguagem de programação é mais do que apenas um meio para instruir um computador a executar tarefas. Ela também serve como um framework dentro do qual organizamos nossas idéias sobre processos computacionais.

Quando escrevemos um programa, devemos prestar especial atenção aos meios que a linguagem oferece para combinar idéias simples para formar idéias mais complexas. Toda linguagem poderosa possui três desses mecanismos:

- expressões e declarações primitivas, que representam os blocos de construção mais simples que a linguagem fornece,
- meios de combinação, através dos quais os elementos compostos são construídos a partir de mais simples, e
- meios de abstração, pelos quais elementos compostos podem ser nomeados e manipulados como unidades.

Na programação, lidamos com dois tipos de elementos: funções e dados. (Em breve, descobriremos que eles realmente não são tão distintos.) Informalmente, os dados são coisas que queremos manipular, e as funções descrevem as regras para manipular os dados. Assim, qualquer linguagem de programação poderosa deve ser capaz de descrever dados primitivos e funções primitivas, além de ter alguns métodos para combinar e abstrair funções e dados.

- **Decomposição** é o mecanismo que usamos para “dividir” o código em pedaços ou módulos, destinados a serem reutilizados. Com isso mantemos o código coerente e organizado. Nesta aula nós vamos realizar a decomposição do programa com funções. Ao final do curso, nós iremos realizar a decomposição com o mecanismo de classes.
- **Abstração** é o mecanismo na qual nós “escondemos” trechos de código como se fosse uma “caixa preta”. Nós não precisamos ver os detalhes do código, mas apenas conhecer sua

interface (input & output).

8.4 Projetando funções

As funções são um ingrediente essencial de todos os programas, grandes e pequenos, e servem como nosso mecanismo primário para expressar processos computacionais em uma linguagem de programação. Fundamentalmente, elas reforçam a idéia de que as funções são abstrações.

- Cada função deve realizar exatamente um trabalho. Esse trabalho deve ser identificável com um nome curto e caracterizável em uma única linha de texto. As funções que executam vários trabalhos em seqüência devem ser divididas em múltiplas funções.
- Se você se encontra copiando e colando um bloco de código, você provavelmente encontrou uma oportunidade para abstração funcional.

Lembre-se que essas simples diretrizes melhoram a legibilidade do código, reduzem o número de erros e muitas vezes minimizam a quantidade total de código escrito. Descompactar uma tarefa complexa em funções concisas é uma habilidade que leva tempo para dominar. Felizmente, o Python oferece vários recursos para suportar seus esforços.

Começamos examinando como expressar a idéia de “quadrado de um número”. Podemos dizer: “Para elevar um número ao quadrado, multiplicamos este número por si mesmo”. Isso é expresso em Python como:

```
def square(x):
    return x * x
```

que define uma nova função que recebeu o nome `square`. Esta função definida pelo usuário não está integrada ao interpretador python. Representa a operação composta de multiplicar algo por si mesmo. O *x* nesta definição é chamado de parâmetro formal, que fornece um nome para que um número seja multiplicado. A definição cria esta função definida pelo usuário e associa-a ao nome `square`.

Como definir uma função: As definições de funções consistem em uma declaração com a construtor `def` da linguagem e indica um **nome** e uma lista, separada por vírgulas, de **parâmetros formais**, além de uma declaração de retorno da função, presente no corpo da mesma, especificada por `return` seguido de uma expressão a ser avaliada sempre que a função é aplicada:

```
def <nome> (<parâmetros formais>):
    return <expressão de retorno>
```

A expressão de retorno não é avaliada imediatamente; Ela é armazenada como parte da função recém-definida e avaliada somente quando a função é eventualmente aplicada. Se o usuário não especificar uma expressão de retorno, o python implicitamente retorna o valor `None` do tipo `NoneType`.

Tendo definida a função `square`, podemos aplicá-la com uma expressão de chamada:

```
square(21)
square(2 + 5)
square(square(3))
```

Nós também podemos usar a função `square` como um bloco de construção na definição de outras funções. Por exemplo, podemos definir facilmente uma função `sum_squares` que, com os dois números como argumentos, retorna a soma de seus quadrados:

```
def sum_squares(x, y):
    return square(x) + square(y)

sum_squares(3, 4)
```

Saída: 25

8.5 Documentação

Uma definição de função geralmente inclui documentação que descreve a função, chamada docstring. Docstrings são convencionalmente inseridos entre três aspas duplas. A primeira linha descreve o objetivo da função em uma linha. As seguintes linhas podem descrever argumentos e esclarecer o comportamento da função:

```
def pressure(v, t, n):
    """Compute the pressure in pascals of an ideal gas.

    v -- volume of gas, in cubic meters
    t -- absolute temperature in degrees kelvin
    n -- particles of gas
    """
    k = 1.38e-23 # Boltzmann's constant
    return n * k * t / v
```

Quando você chama o `help` com o nome de uma função como argumento, você vê sua docstring.

```
help(pressure)
```

Saída

Help on function pressure in module __main__:

```
pressure(v, t, n)
    Compute the pressure in pascals of an ideal gas.

    v -- volume of gas, in cubic meters
    t -- absolute temperature in degrees kelvin
    n -- particles of gas
```

8.6 Exercício: Fatorial

Complete a função **fatorial** abaixo, que recebe como parâmetro um número inteiro k , $k \geq 0$, e retorna $k!$.

Escreva apenas o corpo da função. Observe que o código já inclui chamadas para a função **fatorial**, para que você possa testar a função.

```
def fatorial(k):
    """ Recebe um inteiro k e retorna o valor de k!
        Pre-condição: supõe que k é um numero inteiro não negativo.
    """

```

```

k_fat = 1
cont = 1
#
# complete a função fatorial
#
return k_fat

# testes
print("0! =", factorial(0))
print("1! =", factorial(1))
print("5! =", factorial(5))
print("17! =", factorial(17))

```

8.7 Exercício: Combinação

Usando a função do exercício **Fatorial**, escreva uma função que recebe dois inteiros, **m** e **n**, como parâmetros e retorna a combinação $m! / ((m-n)!n!)$.

```

def combinacao(m, n):
    """ Recebe dois inteiros m e n, e retorna o valor de m! / ((m-n)! n!)
    """

    # Complete a função combinação (não se esqueça de mudar o retorno).
    return None

# testes
print("Combinacao(4,2) =", combinacao(4,2))
print("Combinacao(5,2) =", combinacao(5,2))
print("Combinacao(10,4) =", combinacao(10,4))

```

8.8 Escopo de Variáveis

Nosso subconjunto de Python agora é complexo o suficiente para que o significado dos programas não seja óbvio. E se um parâmetro formal tiver o mesmo nome que uma função builtin? Duas funções podem compartilhar nomes sem confusão? Para resolver essas questões, devemos descrever os ambientes e escopo de nomes com mais detalhes.

Um ambiente em que uma expressão é avaliada consiste em uma seqüência de frames. Cada frame contém ligações (bindings), cada um dos quais associa um nome ao seu valor correspondente. Existe um único frame global. As instruções de atribuição (e importação) adicionam entradas ao primeiro frame do ambiente atual. Até agora, nosso ambiente consistiu apenas no frame global. Vamos visualizar o frame global para o exemplo de código a seguir usando a ferramenta Python Tutor¹.

```

a = 1
b = a
a = a + 1

```

¹O Python tutor está disponível em <http://www.pythontutor.com/>

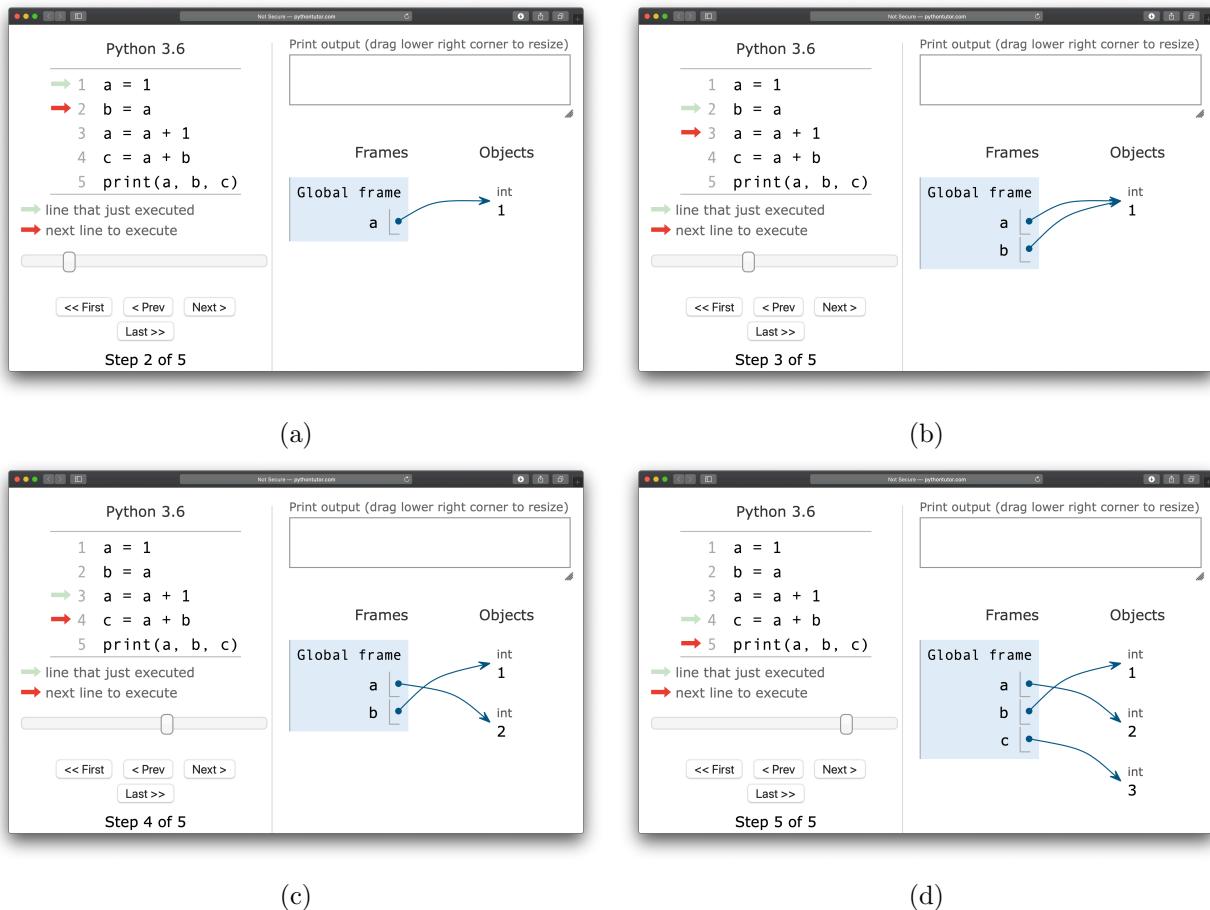


Figura 8.1: Visualização do frame global com Python Tutor

```

c = a + b
print(a, b, c)
  
```

Saída: 2 1 3

A figura 8.1 mostra passo a passo o estado do frame global para o exemplo anterior. A seta vermelha indica a próxima linha de código a ser executada, enquanto que a seta verde mostra a linha que acabou de executar.

Saída:

Para avaliar uma expressão de chamada cujo operador vincula uma função definida pelo usuário, o interpretador Python segue um processo computacional. Tal como acontece com qualquer expressão de chamada, o interpretador avalia as expressões operador e operando e, em seguida, aplica a função vinculada aos argumentos resultantes.

A aplicação de uma função definida pelo usuário apresenta um segundo frame local, que só é acessível a essa função. Para aplicar uma função definida pelo usuário para alguns argumentos, o interpretador:

- Vincula os argumentos aos nomes dos parâmetros formais da função em um novo frame local.
- Executa o corpo da função no ambiente que começa com este frame.

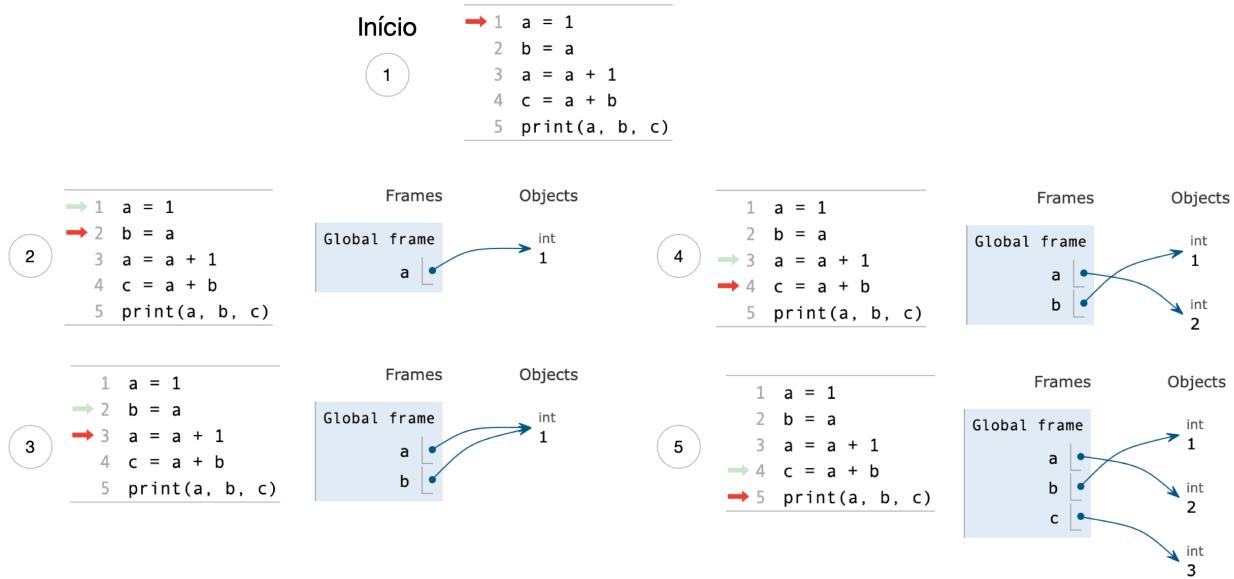


Figura 8.2: Visualização do frame global com Python Tutor

O ambiente em que o corpo da função é avaliado consiste em dois frames: primeiro o frame local que contém ligações de parâmetros formais; segundo o frame global que contém todo o resto. Cada instância de um aplicativo de função possui seu próprio frame local independente.

A figura 8.4, no final deste capítulo ilustra os frames global e local do exemplo de código abaixo:

```
def square(x):
    return x * x

def sum_squares(x, y):
    return square(x) + square(y)

result = sum_squares(5, 12)
```

Saída: 169

O parâmetro formal é vinculado ao valor do parâmetro real quando a função é chamada. Um novo frame é criado quando entramos em uma função. O escopo nada mais é que o mapeamento de nomes para objetos. Dentro de uma função, para poder acessar uma variável definida externamente, temos que marcá-la como global, mas isto é altamente desaconselhável pois a execução de suas funções passam a ter efeitos colaterais que não ficam mais restritos aos parâmetros e valores de retorno.

Saída:

Saída:

Saída:

8.9 Exercícios

- **Reverso do número.** Faça uma função que retorne o reverso de um número inteiro informado. Por exemplo: 127 -> 721.

8.10 Funções como argumentos

Os argumentos de uma função podem ser de qualquer tipo, inclusive funções. Para exemplificar, considere as três funções a seguir onde todas calculam somas. A primeira, `sum_naturals`, calcula a soma dos números naturais até n . A segunda, `sum_cubes`, calcula a soma dos cubos de números naturais até n . A terceira, `pi_sum`, calcula a soma dos termos da série: $\frac{8}{1*3} + \frac{8}{5*7} + \frac{8}{9*11} + \dots$, que converge para π muito devagar.

p.s.: Os exemplos a seguir usam um recurso do python de atribuições em tuplas para atribuir um par de valores a um par de variáveis. Este recurso será melhor explorado na aula sobre tuplas.

```
def sum_naturals(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total

print(sum_naturals(100))
```

Saída: 5050

```
def sum_cubes(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + k*k*k, k + 1
    return total

print(sum_cubes(100))
```

Saída: 25502500

```
def pi_sum(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + 8 / ((4*k-3) * (4*k-1)), k + 1
    return total

print(pi_sum(100))
```

Saída: 3.1365926848388144

Essas três funções compartilham claramente um padrão básico comum. Eles são na sua maioria idênticos, diferindo apenas no nome e na função de k usada para calcular o termo a ser adicionado. Poderíamos gerar cada uma das funções preenchendo os slots no mesmo modelo:

```
def <nome> (n):
    total, k = 0, 1
    while k <= n:
```

```
total, k = total + <termo> (k), k + 1
return total
```

A presença de um padrão tão comum é uma forte evidência de que há uma abstração útil à espera de ser trazida à superfície. Cada uma dessas funções é uma soma de termos. Como designers de programas, gostaríamos que nossa linguagem fosse suficientemente poderosa para que possamos escrever uma função que exprima o conceito de soma em si, ao invés de apenas funções que calculam somas particulares. Podemos fazê-lo facilmente em Python, tomando o modelo comum mostrado acima e transformando os “slots” em parâmetros formais:

No exemplo a seguir, a função `summation` leva como seus dois argumentos o limite superior n junto com a função `termo` que calcula o k -ésimo termo. Podemos usar a função `summation` exatamente como qualquer outra função. Observe, por exemplo como o “biding” da função `cube` com o argumento local de nome `term` garante que o resultado $1 * 1 * 1 + 2 * 2 * 2 + 3 * 3 * 3 = 36$ seja calculado corretamente.

```
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

def cube(x):
    return x*x*x

def sum_cubes(n):
    return summation(n, cube)

result = sum_cubes(3)
print(result)
```

Saída: 36

Usando uma função identidade (i.e., função que retorna seu argumento), também podemos somar números naturais usando exatamente a mesma função de soma `summation`.

```
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

def identity(x):
    return x

def sum_naturals(n):
    return summation(n, identity)

result = sum_naturals(10)
print(result)
```

Saída: 55

Podemos definir uma função `pi_sum` usando nossa abstração `summation` definindo uma função

`pi_term` para calcular cada termo. Passamos o argumento `1e6`, uma notação para $1 \times 10^6 = 1000000$, para gerar uma aproximação próxima de π .

```
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

def pi_term(x):
    return 8 / ((4*x-3) * (4*x-1))

def pi_sum(n):
    return summation(n, pi_term)

result = pi_sum(1e6)
print(result)
```

Saída: 3.141592153589902

8.11 Funções Aninhadas e Funções como valores de retorno

Os exemplos anteriores demonstraram como a capacidade de passar funções como argumentos aumenta significativamente o poder de expressão de nossa linguagem de programação. Cada conceito geral ou equação mapeia sua própria função. Uma consequência negativa dessa abordagem é que nosso frame global fica poluído com nomes de pequenas funções, que devem ser únicos. Outro problema é que somos limitados por assinaturas específicas de funções com um número fixo de argumentos. As definições de funções aninhadas endereçam esses dois problemas.

Um ambiente pode consistir em uma cadeia arbitrariamente longa de frames, que sempre termina com o frame global. Nos exemplos anteriores os ambientes tinham no máximo dois frames: um frame local e o frame global. Ao chamar funções definidas dentro de outras funções, através de declarações aninhadas, podemos criar cadeias mais longas, como veremos no exemplo a seguir.

Com isso perceberemos duas vantagens principais do escopo lexico em Python. Os nomes de uma função local não interferem com nomes externos à função em que está definido, porque o nome da função local será vinculado no ambiente local atual em que foi definido, em vez do ambiente global. Uma função local pode acessar o ambiente da função na qual ela está encerrada (*closure*), porque o corpo da função local é avaliado em um ambiente que estende o ambiente de avaliação em que foi definido.

Podemos alcançar um poder ainda mais expressivo em nossos programas, criando funções cujos valores retornados são eles próprios funções. Uma característica importante das linguagens de programação com alcance lexico é que as funções definidas localmente mantêm seu ambiente pai quando elas são retornadas. O exemplo que vamos estudar a seguir ilustra o poder deste recurso.

8.12 Estudo de caso: algoritmo de Aproximações de Newton

Este exemplo irá nos mostrar como os valores de retorno da função e as definições locais podem funcionar juntas para expressar idéias gerais de forma concisa. Vamos revisitar o algoritmo de aproximações de Newton, amplamente usado no aprendizado de máquinas (machine learning), computação científica, design de hardware e otimização.

O método de Newton é uma abordagem iterativa clássica para encontrar os argumentos de uma função matemática que produz um valor de retorno 0. Esses valores são chamados zeros da função. Encontrar um zero de uma função é muitas vezes equivalente a resolver algum outro problema de interesse, como por exemplo, calcular a raiz quadrada de um número.

Observe que parte da aprendizagem de ciência da computação é entender como quantidades como essas podem ser computadas e a abordagem geral aqui apresentada é aplicável para resolver uma grande classe de equações.

O método de Newton é um poderoso método computacional geral para resolver equações diferenciáveis. Algoritmos muito rápidos para logaritmos e divisão de inteiros grandes empregam variantes desta técnica. O método de Newton é um algoritmo de melhoria iterativo: melhora um palpite do zero para qualquer função que seja diferenciável, o que significa que pode ser aproximado por uma linha reta (tangente) em qualquer ponto. O método de Newton segue essas aproximações lineares para encontrar zeros de função.

A figura 8.3(a) mostra uma linha que atravessa o ponto $(x, f(x))$ e que tem a mesma inclinação que a curva para a função $f(x)$ nesse ponto. Essa linha é chamada de tangente, e sua inclinação é chamada de derivada de f em x .

A inclinação desta linha (slope) nos fornece a proporção da mudança no valor da função para a alteração no argumento da função. Por isso, traduzindo x por $f(x)$ dividido pela inclinação dará o valor do argumento no qual essa linha tangente toca o ponto 0.

Usando o método de Newton, podemos calcular raízes de grau arbitrário n . O grau n da raiz de a é x , tal que $x * x * x \dots x = a$ com x repetido n vezes. Por exemplo,

- A raiz quadrada de 64 é 8, porque $8 * 8 = 64$
- A raiz cubica de 64 é 4, porque $4 * 4 * 4 = 64$
- A raiz sexta de 64 é 2, porque $2 * 2 * 2 * 2 * 2 * 2 = 64$

Podemos calcular raízes usando o método de Newton com as seguintes observações: A raiz n -ésima de a (escrito $\sqrt[n]{a}$) é o valor x tal que $x^n - a = 0$

Se pudermos encontrar o zero dessa última equação, então podemos calcular as raízes de grau n para um dado número a . A figura 8.3(b) traça as curvas para n igual a 2, 3 e 6 e a igual a 64, onde podemos visualizar essa relação.

Na solução que se segue, a função `newton_update` expressa o processo computacional de seguir esta linha tangente para 0, para uma função f e sua derivada df .

Em seguida, definimos a função `find_root` em termos da função `newton_update`, do algoritmo de melhoria `improve` e de uma função de comparação para ver se $f(x)$ está perto de 0.

Iremos obter as raízes de grau arbitrário n , calculando $f(x) = x^n - a$ e sua derivada $df(x) = n * x^{n-1}$.

O código abaixo exemplifica a implementação do método de Newton.

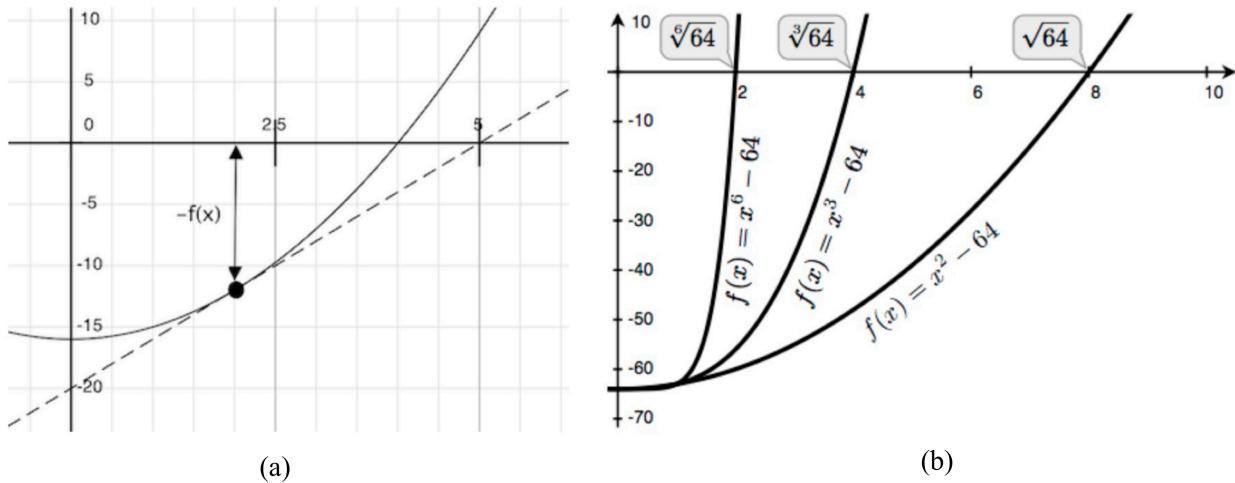


Figura 8.3: Método de Newton

```

def improve(update, close, guess=1):
    while not close(guess):
        guess = update(guess)
    return guess

def newton_update(f, df):
    def update(x):
        return x - f(x) / df(x)
    return update

def find_zero(f, df):
    def near_zero(x):
        return abs(f(x)) <= 1e-15
    return improve(newton_update(f, df), near_zero)

def nth_root_of_a(n, a):
    def f(x):
        return x**n - a
    def df(x):
        return n * x**(n-1)
    return find_zero(f, df)

print(nth_root_of_a(3, 64))

```

Saída: 4.0

8.13 Funções anônimas: lambda

Além da declaração `def`, o Python também fornece um construtor que gera objetos de função, chamada de `lambda`. Como `def`, essa expressão cria uma função a ser chamada posteriormente, mas retorna a função em vez de atribuí-la a um nome. É por isso que as funções lambdas às vezes são conhecidas como funções anônimas (ou seja, sem nome).

A forma geral da função lambda é a palavra-chave `lambda`, seguida por um ou mais argumentos (exatamente como a lista de argumentos que você coloca entre parênteses em um cabeçalho `def`), seguida por uma expressão após dois pontos:

```
lambda argument1, argument2, ... argumentN: expressão usando argumentos
```

Os objetos de função retornados pela execução de expressões lambda funcionam exatamente da mesma forma que os criados e atribuídos por `defs`, mas há algumas diferenças que tornam as funções lambdas úteis em funções especializadas:

lambda é uma expressão, não uma declaração

Por causa disso, uma função `lambda` pode aparecer em lugares em que um `def` não é permitido pela sintaxe do Python - por exemplo, dentro de um literal de lista ou de uma chamada de função. Com `def`, as funções podem ser referenciadas pelo nome, mas devem ser criadas em outro lugar. Como uma expressão, `lambda` retorna um valor (uma nova função) que pode ser atribuído opcionalmente a um nome. Em contraste, a instrução `def` sempre atribui a nova função ao nome no cabeçalho, em vez de retorná-la como resultado.

O corpo de lambda é uma expressão única, não um bloco de declarações

O corpo do `lambda` é semelhante ao que você colocaria na declaração de retorno de um `def`; você simplesmente digita o resultado como uma expressão nua, em vez de explicitamente retorná-lo. Por ser limitado a uma expressão, um `lambda` é menos genérico que um `def` - você só pode expressar a lógica em um corpo lambda sem usar instruções como `if`. Isso ocorre por construção, para limitar o aninhamento de programa: o `lambda` é projetado para codificar funções simples e o `def` manipula tarefas maiores.

```
def func(x, y, z):
    return x + y + z

func(2, 3, 4)
```

Saída: 9

```
f = lambda x, y, z: x + y + z

f(2, 3, 4)
```

Saída: 9

Lambda é comumente usado para codificar tabelas de “execução”, que são listas ou dicionários de ações a serem executadas sob demanda. Por exemplo:

```
L = [lambda x: x ** 2, lambda x: x ** 3, lambda x: x ** 4]

for f in L:
    print(f(2))

print(L[0](3))
```

Saída

```
4
8
16
9
```

A expressão `lambda` é mais útil como um atalho para `def`, quando você precisa inserir pequenos pedaços de código executável em locais onde as declarações são sintaticamente ilegais. O trecho de código anterior, por exemplo, constrói uma lista de três funções incorporando expressões `lambda` dentro de um literal de lista; um `def` não funcionará dentro de uma lista literal, porque é uma declaração, não uma expressão. A codificação `def` equivalente exigiria nomes de funções temporários (que poderiam colidir com outros nomes) e definições de funções fora do contexto de uso pretendido (que pode estar distante):

```
def f1(x):
    return x ** 2

def f2(x):
    return x ** 3

def f3(x):
    return x ** 4

L = [f1, f2, f3]
for f in L:
    print(f(2))

print(L[0](3))
```

Saída

```
4
8
16
9
```

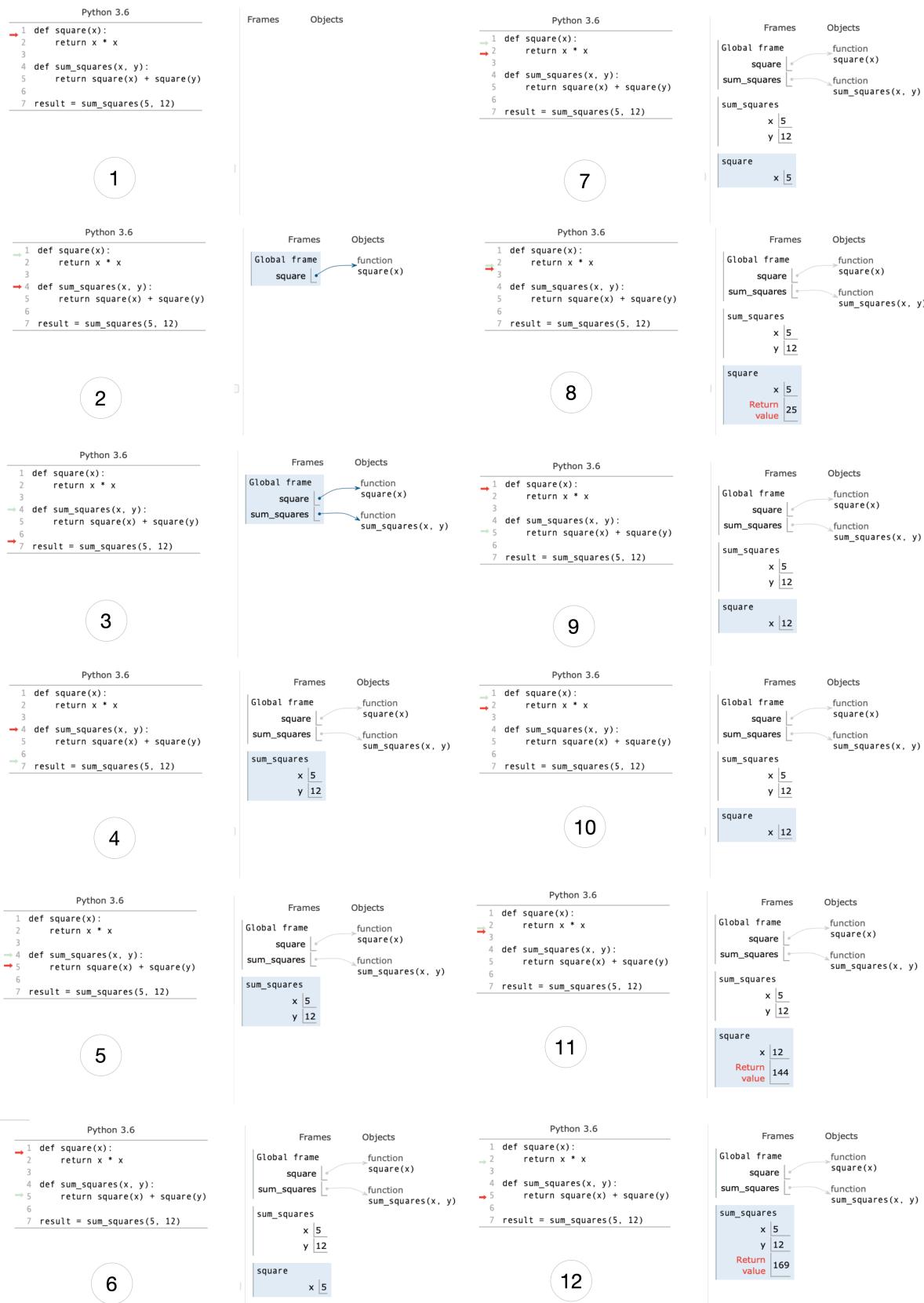


Figura 8.4: Visualização de frames independentes global e local com Python Tutor

Capítulo 9

Strings

9.1 Seqüências

Uma sequência é uma coleção ordenada de valores. A seqüência é uma abstração poderosa e fundamental em ciência da computação. As seqüências não são instâncias de uma built-int específica ou representação de dados abstratos, mas sim uma coleção de comportamentos que são compartilhados entre vários tipos diferentes de dados. Ou seja, existem muitos tipos de seqüências, mas todos compartilham um comportamento comum. Em particular:

Comprimento (len): Uma sequência tem um comprimento finito. Uma seqüência vazia tem o comprimento 0.

Seleção de elementos: Uma sequência tem um elemento correspondente a qualquer índice inteiro não negativo inferior ao seu comprimento, começando em 0 para o primeiro elemento.

O Python inclui vários tipos de dados nativos que são seqüências. Neste curso nós vamos abordar **strings, tuplas e listas**.

9.2 Strings em Python

Uma string é uma seqüência de caracteres. Um caracter é simplesmente um símbolo. Por exemplo, o idioma inglês tem 26 caracteres. Em Python, string é uma sequência de caracteres *Unicode*. O Unicode foi introduzido para incluir todos os caracteres em todos os idiomas e trazer uniformidade na codificação. Você pode aprender mais sobre o Unicode aqui. Note também que strings são objetos imutáveis, isto é, não podem ser modificados:

```
s = "hello"
s[0] = 'y'
s = 'y' + s[1:len(s)]
print(s)
```

Saída

```
Traceback (most recent call last):
  File "/Users/rodacki/ex_string.py", line 2, in <module>
    s[0] = 'y'
```

```
TypeError: 'str' object does not support item assignment
```

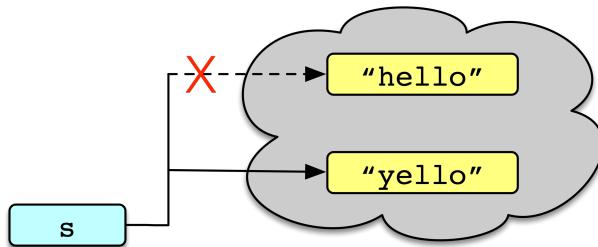


Figura 9.1: Imutabilidade de strings

9.2.1 Comprimento de Strings

- `len()` é a função usada para se obter o comprimento da cadeia de caracteres. Ex:

```
s = "abc"
len(s)
```

Saída: 3

9.2.2 Acessando elementos de uma String

Cada um dos caracteres de uma string corresponde a um número de índice, começando com o número de índice 0 e indice superior igual a `len() - 1`. Se tivermos uma string longa e quisermos identificar um item próximo ao fim, também podemos contar de trás para frente, isto é, a partir do final da string, começando pelo índice `-1` e terminando em `-len()`.

Ao referenciar números de índice, podemos isolar um dos caracteres em uma string. Fazemos isso colocando os índices entre colchetes (`"[]"`). Com isso, pode-se obter o valor/caracter em um determinado índice ou posição.

A figura 9.2 mostra os índices da string `s`, sendo que `s = 'ALGORITMOS'`.

	0	1	2	3	4	5	6	7	8	9
<code>s =</code>	A	L	G	O	R	I	T	M	O	S
	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Figura 9.2: Indexação de strings

- a indexação sempre começa em 0 na forma direta.
- último elemento sempre tem índice `-1` na forma reversa.

```
s = "abc"
print(s[0], end=" ")
print(s[1], end=" ")
print(s[2], end=" ")
print(s[-1], end=" ")
print(s[-2], end=" ")
print(s[-3], end=" ")
```

Saída: a b c c b a

9.2.3 Slices de strings

- pode-se fatiar (slice) strings usando [start: stop: step]
- pode-se fornecer dois números apenas, [start: stop], sendo que step = 1, por padrão
- pode-se também omitir os números e deixar apenas dois pontos (:). Neste caso, o valor padrão para start = 0 e para stop = len().

```
s = "abcdefghijklmnopqrstuvwxyz"
print(s[3:6])
print(s[3:6:2])
print(s[::-1])
print(s[::-1])
print(s[4:1:-2])
```

Saída:

```
def
df
abcdefghijklmnopqrstuvwxyz
hgfedcba
ec
```

9.2.4 Operador in

O operador **in** verifica se uma substring é parte de uma outra string. Exemplos:

```
'tho' in 'Python'
```

Saída: True

```
'thor' in 'Python'
```

Saída: False

9.2.5 Método strip

O método **strip** retorna uma string sem os brancos e caracteres de mudança de linha no início e final de uma string:

```
aa = '\n abcndef \n'
print(aa)
print("<", aa, ">")
print(aa.strip())
```

Saída:

```
abcndef

<
  abcndef
>
abcndef
```

9.2.6 Método find

O método **find** retorna onde a substring começa na string. Este retorna -1 quando a substring não ocorre na string:

```
p = 'python'
print(p.find('tho'), p.find('thor'))
```

Saída: 2 -1

9.2.7 Método startswith

O método **startswith()** verifica se a string inicia com str, opcionalmente restringindo a correspondência com os índices de início (beg) e fim (end).

Sintaxe

```
str.startswith (str, beg = 0, end = len (string));

str = "this is string example....wow!!!"
print (str.startswith( 'this' ))
print (str.startswith( 'string', 8 ))
```

Saída:

```
True
True
```

9.2.8 Método replace

O método **replace** serve para trocar todas as ocorrências de uma substring por outra em uma string. Note que como strings são imutáveis, uma nova string é retornada.

```
s = "abcaabcdgabc abc a b c"
s.replace("abc", "")
```

Saída: 'dfg a b c'

9.2.9 Strings e laços

Os dois trechos de código abaixo iteram sobre um string e produzem o mesmo resultado. No entanto, o segundo é considerado mais “pythonico”, isto é, mais elegante, mais “clean”:

```
s = "abcdefghijklm"

for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")

for char in s:
    if char == 'i' or char == 'u':
        print("There is an i or u")
```

9.2.10 Exercícios

1. Letras em comum: conte o número de letras em comum entre duas strings. Use a builtin `isalpha` para descobrir se um caracter é letra ou não. Ex.:

```
x = "A"  
x.isalpha() == true  
y = "3"  
y.isaplha() == false
```

2. Escreva uma função que faz a busca de todas as ocorrências de uma palavra em um texto.

Capítulo 10

Tuplas

Tuplas são seqüências ordenadas de elementos separados por vírgulas, representados ou não entre parênteses, isto é, os parênteses não são obrigatórios. Pode-se ainda misturar elementos de tipos diferentes. No entanto, tuplas tem a característica de imutabilidade, isto é, seus elementos são imutáveis. Em outras palavras, não é possível alterar os valores dos elementos.

Exemplos de tuplas:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )
tup3 = "a", "b", "c", "d"
tup4 = ("IFC", )
tup5 = ()
```

Observações:

- em `tup4 = ("IFC",)`, `tup4` representa uma tupla com um único elemento. Note a vírgula após o elemento. Ela é necessária para diferenciar de uma expressão entre parênteses. Veja o exemplo abaixo:

```
t1 = 'A',
t2 = ('A')
print(type(t1), type(t2))
```

Saída: <class 'tuple'> <class 'str'>

- em `tup5 = ()`, `tup5` representa uma tupla vazia. Outra forma de criar uma tupla é com a função integrada `tuple`. Sem argumentos, cria uma tupla vazia, se os argumentos forem uma sequência (string, lista ou tupla), o resultado é uma tupla com os elementos da sequência:

```
t = tuple('IFC')
print(t, type(t))
```

Saída: ('I', 'F', 'C') <class 'tuple'>

10.1 Acessando Valores em tuplas

Como Strings, os índices de tupla começam em 0 e podem ser cortados (slice), concatenados, e assim por diante. Ainda como Strings, para acessar os valores em uma tupla, use os colchetes para

cortar junto com o índice, ou índices, para obter o valor disponível nesse índice. Exemplos:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7)
print ("tup1[0]: ", tup1[0])
print ("tup2[1:5]: ", tup2[1:5])
# A sentença abaixo não é valida --> imutabilidade!
tup1[0] = "calculus"
```

Saída:

```
tup1[0]: physics
tup2[1:5]: (2, 3, 4, 5)
Traceback (most recent call last):
  File "/Users/rodacki/ex_tuplas.py", line 6, in <module>
    tup1[0] = "calculus"
TypeError: 'tuple' object does not support item assignment
```

10.2 Contatenação de tuplas

Assim como nas strings, a concatenação de tuplas pode ser feita com o operador +. Por exemplo:

```
tupX = (12, 34.56)
tupY = ('abc', 'xyz')

# Vamos criar uma nova tupla da seguinte maneira
tupZ = tupX + tupY
print (tupZ)
```

Saída: (12, 34.56, 'abc', 'xyz')

10.3 Eliminando elementos de uma tupla

Não é possível remover elementos de uma tupla individualmente. Porém, é possível juntar outra tupla com os elementos indesejados descartados.

Para remover explicitamente uma tupla inteira, basta usar a sentença `del`. Exemplos:

```
tup1 = ('physics', 'chemistry', 1997, 2000);
print(type(tup1[3]))
tup2 = tup1[0:1] + (tup1[3], )
print (tup2)
del tup1;
print ("Depois de remover a tup1 : ")
print (tup1)
```

Saída:

```
<class 'int'>
('physics', 2000)
Depois de remover a tup1 :
Traceback (most recent call last):
```

```
File "/Users/rodacki/ex_delete.py", line 7, in <module>
    print (tup1)
NameError: name 'tup1' is not defined
```

10.4 Operações básicas em tuplas

As tuplas respondem a todas as operações de seqüência geral que usamos em Strings anteriormente: comprimento (`len`), concatenação (+), repetição (*), membro (`in`) e iteração. Exemplos:

```
print(len((1, 2, 3)))
print((1, 2, 3) + (4, 5, 6))
print(( 'Hi!', ) * 4)
print(3 in (1, 2, 3))
for x in (1,2,3) : print (x, end = ' ')
```

Saída:

```
3
(1, 2, 3, 4, 5, 6)
('Hi!', 'Hi!', 'Hi!', 'Hi!')
True
1 2 3
```

10.5 Atribuições com tuplas

O Python possui um recurso de atribuição de tupla muito poderoso que permite que uma tupla de variáveis à esquerda de uma atribuição seja atribuída valores de uma tupla à direita da atribuição. (Já vimos isso usado antes, na seção “Funcões como Argumento”, para pares, mas pode ser generalizada para qualquer número de variáveis.)

Uma maneira de pensar sobre a atribuição de tupla é como empacotamento (packing) / descompactação (unpacking) de tupla.

- os valores à esquerda são “empacotados” (*packing*) em uma tupla;
- os valores em uma tupla à direita são “descompactados” (*unpacking*) nas variáveis / nomes à direita:

```
t = ("Paulo", 2020, "IFC")      # tuple packing
(name, year, studies) = t       # tuple unpacking
print(studies, year, name)
```

Saída: IFC 2020 Paulo

- Swap de valores de variáveis: de vez em quando, é necessário trocar os valores de duas variáveis. Com declarações de atribuição convencionais, temos que usar uma variável temporária. Por exemplo, para trocar x e y:

```
x = 1
y = 2
# maneira errada
x = y
```

```
y = x
print(x, y)
```

Saída: 2 2

```
x = 1
y = 2
# correto, mas precisa de uma variável temporária
temp = x
x = y
y = temp
print(x, y)
```

Saída: 2 1

```
x = 1
y = 2
# usando tuplas
(x, y) = (y, x)    # ou, simplesmente: x, y = y, x
print(x, y)
```

Saída: 2 1

- O número de variáveis à esquerda e o número de valores à direita precisam ser iguais:

```
a, b = 1, 2, 3
```

Saída:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

10.6 Tuplas no retorno de funções

Tuplas podem ser usadas para retornar mais de um valor de uma função. Por exemplo:

```
def quociente_e_resto(x, y):
    q = x // y
    r = x % y
    return (q, r)

(quo, res) = quociente_e_resto(7, 4)
print(quo, res)
```

Saída: 1 3

10.7 Tuplas como argumentos de comprimento variável de funções

As funções em Python podem receber um número variável de argumentos. Um nome de parâmetro que comece com * reúne vários argumentos em uma tupla. Esses argumentos são chamados de argumentos de comprimento variável e não são nomeados na definição da função, ao contrário dos argumentos necessários e padrão. Esta tupla permanece vazia se nenhum argumento adicional for especificado durante a chamada da função. Vejamos o exemplo a seguir:

```

def printinfo( arg1, *vartuple ):
    # Imprime as variáveis passadas como argumento
    print("A saída é: ")
    print(arg1)
    for var in vartuple:
        print(var)
    return

# Agora chamamos a função printinfo
printinfo( 10 )
printinfo( 70, 60, 50 )

```

Saída:

```

A saída é:
10
A saída é:
70
60
50

```

O contrário de reunir é espalhar. Se você tiver uma sequência de valores e quiser passá-la a uma função como argumentos múltiplos, pode usar o operador *. Por exemplo, a função `quociente_e_resto` definida anteriormente recebe exatamente dois argumentos; ele não funciona com uma tupla:

```

def quociente_e_resto(x, y):
    q = x // y
    r = x % y
    return (q, r)

t = 7, 4
quo, res = quociente_e_resto(t)
print(quo, res)

```

Saída:

```

Traceback (most recent call last):
  File "/Users/rodacki/ex_tupla.py", line 8, in <module>
    quo, res = quociente_e_resto(t)
TypeError: quociente_e_resto() missing 1 required positional argument: 'y'

```

No entanto, se você espalhar a tupla, aí funciona:

```

t = 7, 4
quo, res = quociente_e_resto(*t)
print(quo, res)

```

Saída: 1 3

Capítulo 11

Listas

Uma lista (list) em Python é uma sequência ou coleção ordenada de valores. Cada valor na lista é identificado por um índice. Os valores que formam uma lista são chamados elementos ou itens. Listas são similares a tuplas, no entanto, diferentemente de tuplas, lista são **mutáveis**, isto é, os itens de uma lista podem ser alterados.

A maneira mais simples de se criar uma lista é envolver os elementos da lista por colchetes ([e]):

```
frutas = ["banana", "maça", "cereja"]
print(frutas)

frutas[0] = "pera"
frutas[-1] = "laranja"
print(frutas)
```

Saída:

```
['banana', 'maça', 'cereja']
['pera', 'maça', 'laranja']
```

Também pode-se criar uma lista através da builtin `list`:

```
digitos = list(range(10))
print(digitos, type(digitos))
curso = list("IFC-BCC")
print(curso, type(curso))
```

Saída:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>
['I', 'F', 'C', 'B', 'C', 'C'] <class 'list'>
```

Como em tuplas, os elementos de uma lista não precisam ser do mesmo tipo. A lista a seguir contém um string, um float, um inteiro e uma outra lista:

```
x = ["oi", 2.0, 5, [10, 20]]
print(len(x))
print(len(x[3]))
print(x[3][0])
```

Saída:

```
4
2
10
```

Uma lista em uma outra lista é dita aninhada (nested) e a lista mais interna é chamada frequentemente de sublista (sublist). Finalmente, existe uma lista especial que não contém elemento algum. Ela é chamada de lista vazia e é denotada por [].

```
red = "vermelho"
cores = ['azul', "amarelo", 'verde', "branco", red]
numeros = [123, 5460]
vazia = []
mista = [1, 1, [2, 3], [5, 8, 13]]
print(len(cores), len(numeros), len(vazia), len(mista), len(mista[3]))
```

Saída: 5 2 0 4 3

A sintaxe para acessar um elemento de uma lista é a mesma usada para acessar um caractere de um string ou um elemento de uma tupla. Nós usamos o operador de indexação ([] – não confundir com a lista vazia). A expressão dentro dos colchetes especifica o índice. Lembre-se que o índice do primeiro elemento é 0. Qualquer expressão que tenha como resultado um número inteiro pode ser usada como índice. Índices negativos indicarão elementos da direita para a esquerda ao invés de da esquerda para a direita.

11.1 Operações básicas: Pertinência, Concatenação e Repetição

Os operadores `in` e `not in` são operadores booleanos ou lógicos que testam a pertinência (membership) em uma sequência. Novamente, como com strings, o operador `+` concatena listas. Analogamente, o operador `*` repete os itens em uma lista um dado número de vezes. Exemplos:

```
citricas = ["laranja", "abacaxi"]
frutas = [ "maça", "pêra"] + citricas
print(frutas)
print("pêra" in frutas)
zeros = [0] * 4
print(zeros)
```

Saída:

```
['maça', 'pêra', 'laranja', 'abacaxi']
True
[0, 0, 0, 0]
```

11.2 Métodos de Listas

O Python oferece métodos que operam em listas. Estes são alguns exemplos:

- `append` adiciona um novo elemento ao fim de uma lista;
- `extend` toma uma lista como argumento e adiciona todos os elementos;
- `sort` classifica os elementos da lista em ordem ascendente.

Há várias formas de excluir elementos de uma lista. Se souber o índice do elemento que procura, você pode usar `pop`. `pop` altera a lista e retorna o elemento que foi excluído. Se você não incluir um índice, ele exclui e retorna o último elemento. Se não precisar do valor removido, você pode usar a instrução `del`. `del` também manipula índices negativos e produz um erro de execução se o índice estiver fora do intervalo da lista. Além disso, você pode usar uma fatia como argumento para `del`. Como é usual, fatias selecionam todos os elementos até, mas não incluindo, o segundo índice. Por fim, se souber o elemento que quer excluir (mas não o índice), você pode usar `remove`.

```

11 = ['a', 'b', 'c']
11.append('d')
print(11)

12 = ['a', 'b', 'c']
13 = ['d', 'e']
12.extend(13)
print(12)

14 = ['d', 'c', 'e', 'b', 'a']
14.sort()
print(14)

15 = ['a', 'b', 'c']
x = 15.pop(1)
print(x)

16 = ['a', 'b', 'c']
del 16[1]
print(16)

17 = ['a', 'b', 'c']
17.remove('b')
print(17)

```

Saída:

```

['a', 'b', 'c', 'd']
['a', 'b', 'c', 'd', 'e']
['a', 'b', 'c', 'd', 'e']
b
['a', 'c']
['a', 'c']

```

11.3 Listas e Strings

Como vimos, a função `list` quebra uma string em letras individuais. Se você quiser quebrar uma string em palavras, você pode usar o método `split`:

```

s = "Bacharelado em Ciência da Computação"
t = s.split()
print(t)

"Bemvindo à disciplina de Algoritmos".split()

```

Saída:

```
[‘Bacharelado’, ‘em’, ‘Ciência’, ‘da’, ‘Computação’]
[‘Bemvindo’, ‘à’, ‘disciplina’, ‘de’, ‘Algoritmos’]
```

Um argumento opcional chamado `delimiter` especifica quais caracteres podem ser usados para demonstrar os limites das palavras. O valor padrão (default) para `delimiter` são: espaços, tabs e *newline*. O exemplo seguinte usa um hífen como delimitador:

```
s = ‘spam-spam-spam’
delimiter = ‘-’
t = s.split(delimiter)
print(t)
```

Saída: [‘spam’, ‘spam’, ‘spam’]

Podem haver substrings vazias no retorno do método `split`:

```
s = “1;2;;3”
t = s.split(‘;’)
print(t)
```

Saída: [‘1’, ‘2’, ‘’, ‘3’]

O método `join` é o contrário de `split`. Ele toma uma lista de strings e concatena os elementos. Por `join` ser um método de string (função que opera em strings), então é preciso invocá-lo no delimitador e passar a lista como parâmetro:

```
t = [‘The’, ‘measure’, ‘of’, ‘love’, ‘is’, ‘to’, ‘love’, ‘without’, ‘
     measure’]
delimiter = ‘ ’
s = delimiter.join(t)
print(s)
```

Saída: The measure of love is to love without measure

Nesse caso, o delimitador é um caractere de espaço, então `join` coloca um espaço entre as palavras. Para concatenar strings sem espaços, você pode usar a string vazia ‘’, como delimitador.

Exemplo de aplicação

A função abaixo devolve listas contendo as palavras e as linhas em um texto.

```
def wc(text):
    return text.split(), text.split(‘\n’)

text = “Eu estou testando o \n número de linhas \n e palavras”
w, c = wc(text)
print(w)
print(c)
```

Saída:

```
[‘Eu’, ‘estou’, ‘testando’, ‘o’, ‘número’, ‘de’, ‘linhas’, ‘e’, ‘palavras’]
[‘Eu estou testando o’, ‘número de linhas’, ‘e palavras’]
```

11.4 Referências de Listas

Como sabemos, objetos Python podem ser identificados usando seus identificadores (nomes, variáveis) que são únicos. Podemos também testar se dois nomes se referem ao mesmo objeto usando o operador `is`. O operador `is` retorna True se as duas referências são ao mesmo objeto. Em outras palavras, as referências são a mesma. Vejamos o exemplo abaixo com strings. Vamos usar novamente o Python Tutor nestes exemplos, cujo resultado é mostrado na figura 11.1.

```
a = "banana"
b = "banana"
print(a is b)  # True
```

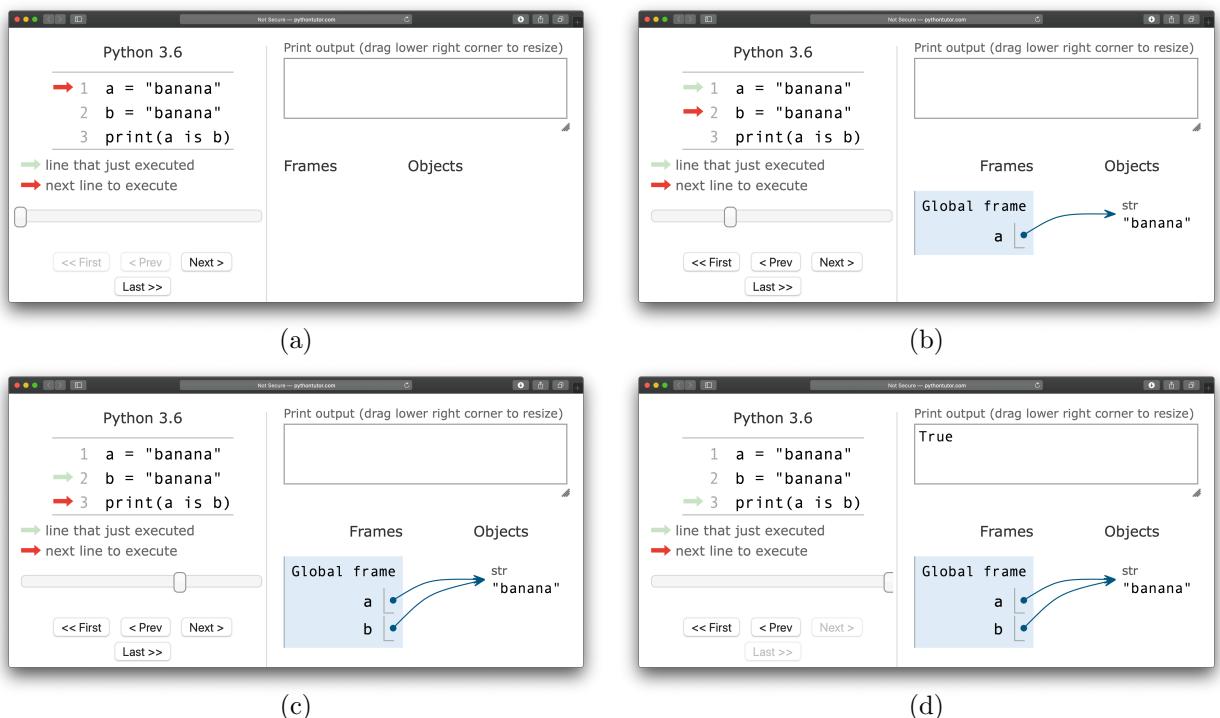


Figura 11.1: Comparaçāo de referências de strings

Como strings são imutáveis, Python optimiza recursos fazendo com que duas variáveis que se referem ao mesmo string se referirem ao mesmo objeto.

Este não é o caso com listas. Considere o exemplo seguir. Aqui `a` e `b` se referem a duas listas diferentes, cada uma por acaso tem os mesmos elementos como valores.

```
a = [81, 82, 83]
b = [81, 82, 83]

print(a is b)  # False
print(a == b)  # True
```

Existe um ponto importante a ser notado a respeito do diagrama de referências da figura 11.2. O valor `a` é uma referência a uma coleção de referências. Essas referências na realidade se referem a valores inteiros em uma lista. Em outras palavras, uma lista é uma coleção de referências para

objetos. É interessante que apesar de *a* e *b* serem duas listas distintas (duas coleções diferentes de referências), o objeto inteiro 81 é compartilhado por ambos. Como strings, inteiros também são imutáveis, portanto Python optimiza e permite que todos compartilhem o mesmo objeto.

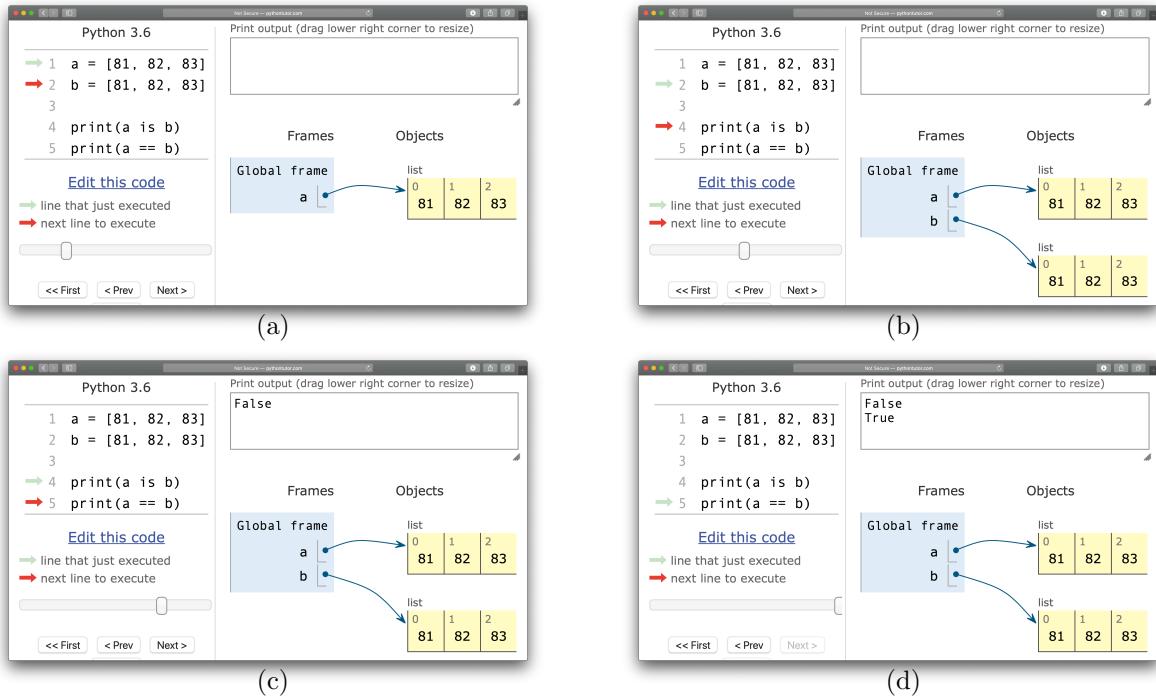


Figura 11.2: Comparação de referências de listas

11.4.1 Alias e clone de Listas

Como variáveis fazem referência a objetos, se atribuirmos uma variável a outra, ambas as variáveis passam a fazer referência ao mesmo objeto. No exemplo a seguir, a mesma lista tem dois nomes diferentes, *a* e *b*. Neste caso, dizemos que *a* e *b* são “alias” da lista. Mudanças feitas com um nome afeta o outro. No exemplo a seguir, os ids de *a* e *b* são os mesmos depois da execução do comando de atribuição *b = a*. A figura 11.3 mostra as referências *a* e *b* na memória ao longo da execução do código abaixo.

```

a = [81, 82, 83]
b = [81, 82, 83]

print(a == b)      # True
print(a is b)      # False

b = a
print(a == b)      # True
print(a is b)      # True

b[0] = 5
print(a)           # [81, 82, 83]

```

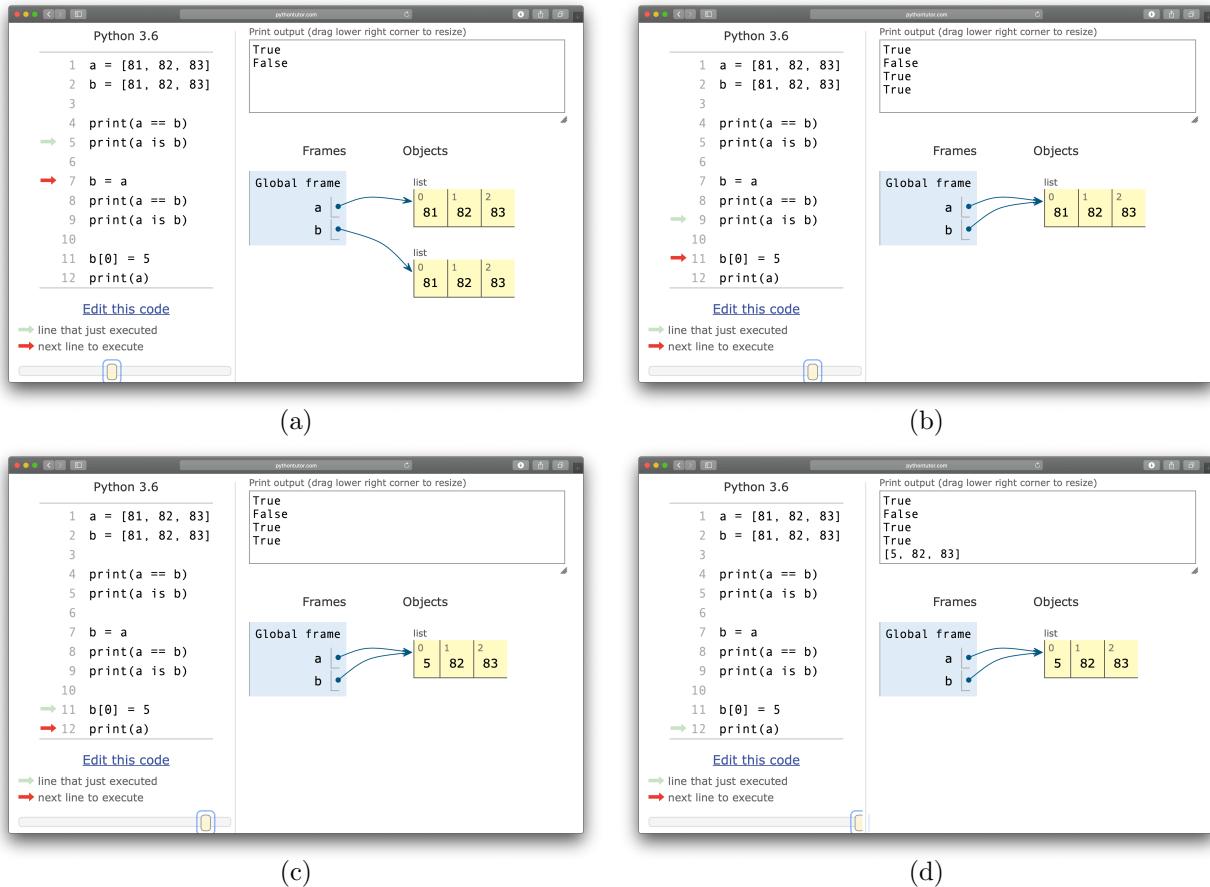


Figura 11.3: Exemplo de alias de listas

Apesar desse comportamento ser útil, ele é algumas vezes inesperado ou indesejável. Em geral, é mais seguro evitar *aliasing* quando você está trabalhando com objetos mutáveis. É evidente que com objetos imutáveis não há problema. Por isto, Python é livre para usar alias de strings e inteiros quando surge uma oportunidade para economizar espaço.

Se desejamos modificar uma lista e também manter uma cópia da lista original, temos que ser capazes de fazer uma cópia da lista, não apenas da referência. Este processo é algumas vezes chamado de clonar (*cloning*), para evitar a ambiguidade da palavra cópia.

A maneira mais fácil de clonarmos uma lista é usar o operador de fatiamento (*slice*). Tomar qualquer fatia de *a*, cria uma nova lista. Para clonar uma lista inteira basta tomarmos a fatia como sendo a lista toda:

```
a = [81, 82, 83, 84, 85]
b = a[:]

print(a == b)      # True
print(a is b)      # False

b[0] = 5
print(a)           # [81, 82, 83, 84, 85]
print(b)           # [5, 82, 83, 84, 85]
```

A figura 11.4 mostra passo a passo o exemplo de clonagem com fatiamento.

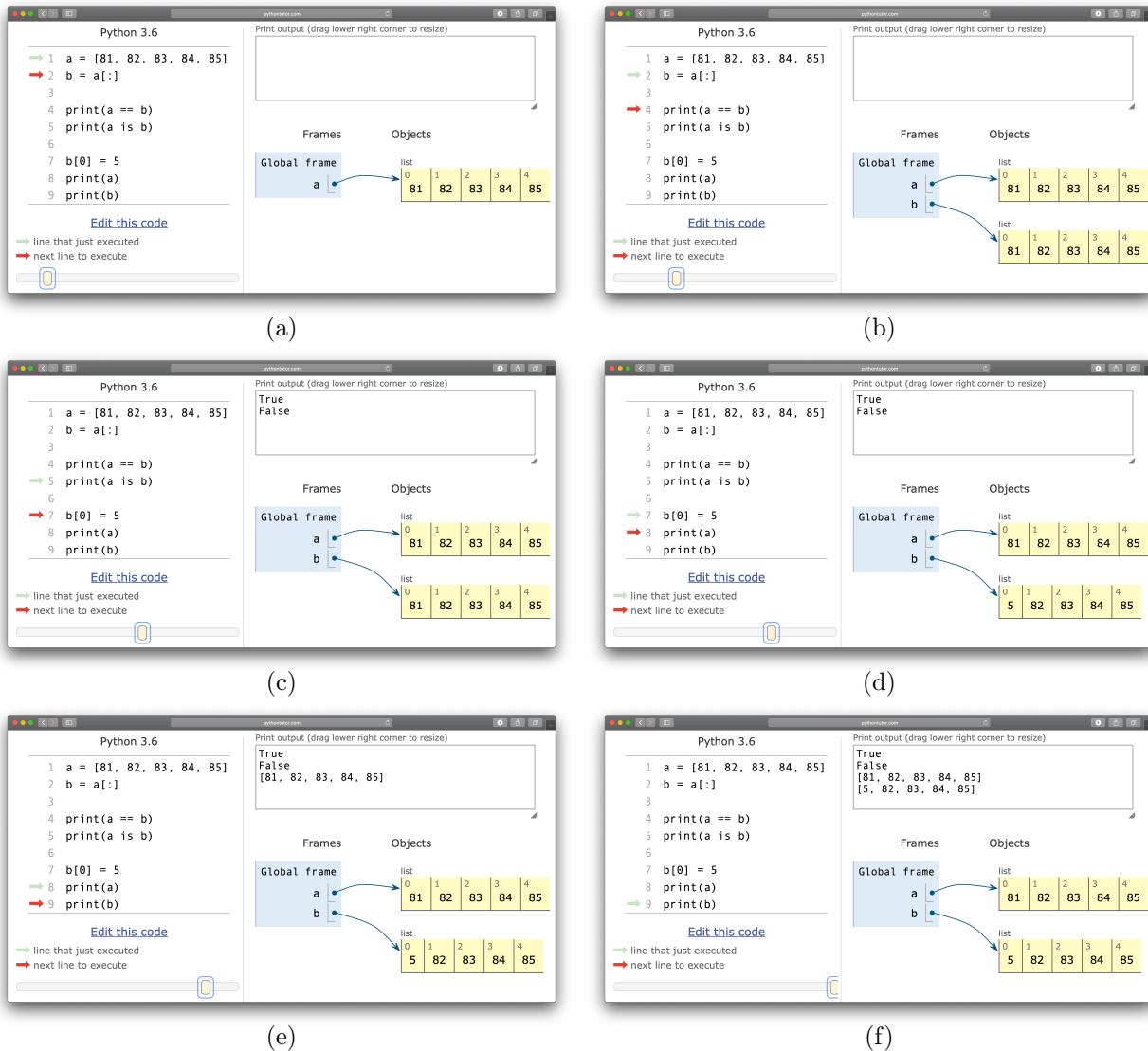


Figura 11.4: Exemplo de clonagem com fatiamento

11.4.2 Listas como Parâmetros

Funções que recebem listas como argumentos e as alteram durante a execução são chamadas de modificadoras (modifiers) e as mudanças que elas realizam são chamadas efeitos colaterais (side effects). Ao passar uma lista como argumento estamos realmente passando para a função uma referência para a lista e não uma cópia (clone) da lista. Como listas são mutáveis, as alterações feitas nos elementos referenciados pelos parâmetros mudarão a lista que o argumento está referenciando. Por exemplo, no exemplo a seguir, a função `double_stuff` recebe uma lista como argumento e multiplica cada elemento da lista por 2. A figura 11.5 mostra o estado das referências à memória ao longo da execução do código.

```

def double_stuff(lista):
    for i in range(len(lista)):
        lista[i] = 2 * lista[i]

things = [2, 5]
print(things)
double_stuff(things)
print(things)

```

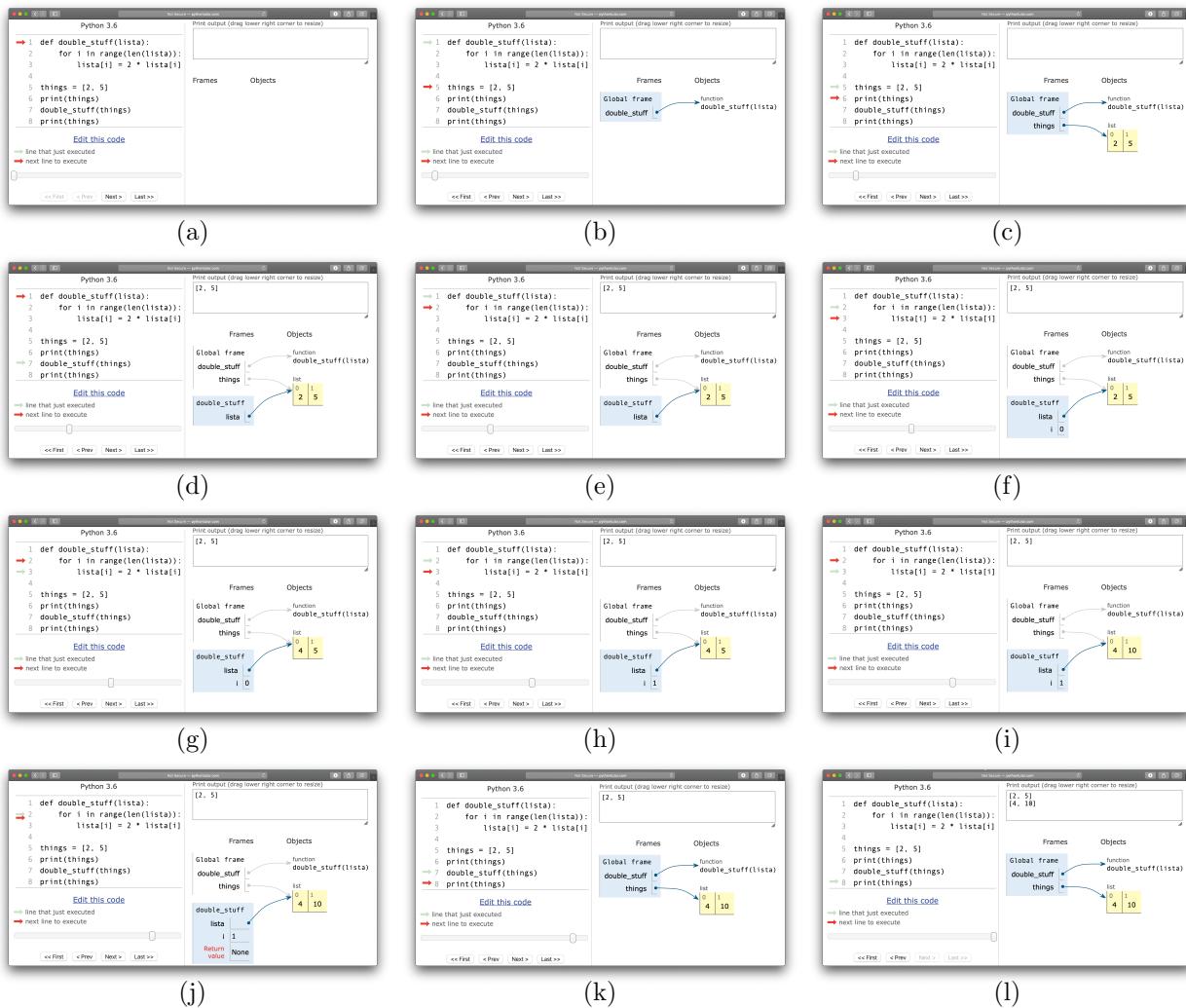


Figura 11.5: Exemplo passagem de lista como parâmetro de função

11.5 Processamento de Seqüências

As seqüências são uma forma tão comum de dados compostos que todos os programas são geralmente organizados em torno dessa abstração única. Os componentes modulares que possuem seqüências como entradas e saídas podem ser misturados e combinados para executar o processamento de dados. Os componentes complexos podem ser definidos encadeando um pipeline de operações de processamento de seqüência, cada um dos quais de forma simples e focado.

11.5.1 Compreensão de Listas

Muitas operações de processamento de seqüência podem ser expressas avaliando uma expressão fixa para cada elemento em uma seqüência e coletando os valores resultantes em uma seqüência de resultados. Em Python, uma compreensão de lista (**List Comprehensions**) é uma expressão que executa tal computação.

Suponha que você tenha escrito uma função que determina se um número inteiro é primo ou não e eu então peça a você para produzir uma lista com os números primos no intervalo de 0 à 50. Uma solução poderia ser a que segue:

```
def is_prime(x):
    for y in range(2, x):
        if x % y == 0:
            return False
    return True

primes = []
for x in range(2,50):
    if is_prime(x):
        primes.append(x)
print(primes)
```

Saída: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Porém Python tem uma solução mais elegante:

```
def is_prime(x):
    for y in range(2, x):
        if x % y == 0:
            return False
    return True

primes = [x for x in range(2, 50) if is_prime(x)]
print(primes)
```

Saída: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Uma compreensão de lista consiste nas seguintes partes:

- Uma seqüência de entrada.
- Uma variável representando membros da seqüência de entrada.
- Opcionalmente, uma expressão Predicado.
- Expansão de saída produzindo elementos da lista de saída dos membros da Seqüência de Entrada que satisfazem o predicado.

Outros exemplos:

```
Celsius = []
temp = True
while temp:
    temp = input("Enter a new temperature (Click Return, for finish):")
    if temp != "":
        Celsius.append(float(temp))

print("Celsius: ", Celsius)
Fahrenheit = [ ((9/5)*x + 32) for x in Celsius ]
print("Fahrenheit: ", Fahrenheit)
```

Saída:

```
Enter a new temperature (Click Return, for finish):34
Enter a new temperature (Click Return, for finish):32
Enter a new temperature (Click Return, for finish):
Celsius: [34.0, 32.0]
Fahrenheit: [93.2, 89.6]
```

No exemplo a seguir, a compreensão da lista verificará primeiro se o número x é divisível em 3 e, em seguida, verifique se x é divisível em 5. Se x satisfizer ambos os requisitos, ele será impresso.

```
number_list = [x for x in range(100) if (x % 3 == 0) if (x % 5 == 0)]
print(number_list)
```

Saída: [0, 15, 30, 45, 60, 75, 90]

Exemplo de aplicação

Uma tripla pitagórica consiste em três inteiros positivos a , b e c , tal que $a^2 + b^2 = c^2$. Essa tripla é comumente escrita (a, b, c) e o exemplo mais conhecido é a tripla $(3, 4, 5)$. A compreensão de lista a seguir cria as tripas de Pitágoras no intervalo $(1, 30)$:

```
[(a,b,c) for a in range(1, 30)
    for b in range(a, 30)
        for c in range(b, 30) if a**2 + b**2 == c**2]
```

Saída:

```
[(3, 4, 5),
(5, 12, 13),
(6, 8, 10),
(7, 24, 25),
(8, 15, 17),
(9, 12, 15),
(10, 24, 26),
(12, 16, 20),
(15, 20, 25),
(20, 21, 29)]
```

11.5.2 Compreensão de Listas Aninhadas

Uma matriz de identidade de tamanho n é uma matriz quadrada n por n com 1 na diagonal principal e 0 em outro lugar. Em Python podemos representar essa matriz por uma lista de listas, onde cada sub-lista representa uma linha. Uma matriz de 3 por 3 seria representada pela seguinte lista:

```
[[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]
```

A matriz no exemplo anterior pode ser gerada pela seguinte compreensão:

```
[[1 if col == row else 0 for col in range(0, 3)] for row in range(0, 3)]
```

Saída: [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

11.5.3 Exercícios:

- Removendo as vogais de uma frase:** pegue uma string como entrada e retorne uma string com as vogais removidas. Escreva os códigos de Python com implementações de laços FOR e Compreensão de Listas (LC);
- Matriz Transposta:** Dada uma matriz A de ordem $m \times n$, a matriz transposta dela será representada por A^t de ordem “invertida” $n \times m$. Essa ordem invertida significa que para transformarmos uma matriz em matriz transposta, basta trocar os elementos das linhas pelo das colunas e vice-versa.

Exemplo:

```
matrix = [[1, 2, 3, 4],
          [5, 6, 7, 8],
          [9, 10, 11, 12]]
rows = len(matrix)
cols = len(matrix[0])
transposta = [[0 for j in range(rows)] for i in range(cols)]
for i in range(cols):
    for j in range(rows):
        transposta[i][j] = matrix[j][i]
print(matrix)
print(transposta)
```

Saída:

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Sua tarefa é gerar a transposta da matriz definida acima, mas usando somente compreensão de listas, isto é, sem usar laços.

- Findall occurrences:** usando compreensão de Listas, construa uma função com a seguinte assinatura: `def findall(palavra, texto):`, que retorna uma lista com os índices de todas as ocorrências de uma palavra em um texto.

4. **Lançamento de dados:** faça um programa que simule um lançamento de dados. Lance o dado 100 vezes e armazene os resultados em uma lista. Depois, mostre quantas vezes cada valor foi conseguido. Dica: use uma lista para os contadores (1 – 6) e uma função para gerar números aleatórios, simulando os lançamentos dos dados.

5. **Números inteiros e seus quadrados:**

- (a) Crie uma lista com inteiros de 0 a 9 elevados ao quadrado;
- (b) Crie uma lista com inteiros de 0 a 9, cujo quadrado é maior que 5 e menor que 50.

6. **Combinações de comidas e bebidas:** crie uma lista com todas as combinações de comidas e bebidas das listas abaixo:

```
[‘agua’, ‘chá’, ‘suco’]  
[‘presunto’, ‘ovos’, ‘queijo’]
```

Capítulo 12

Arrays multidimensionais (Matrizes)

Muitas vezes na Ciéncia a informaçao é organizada em linhas e colunas, formando agrupamentos retangulares denominados matrizes. Com frequênciа, essas matrizes aparecem como tabelas de dados numéricos que surgem em observações físicas, mas também ocorrem em vários contextos matemáticos.

Por exemplo, toda informaçao necessária para resolver um sistema de equações tal como:

$$\begin{aligned} 5x + y &= 3 \\ 2x - y &= 4 \end{aligned}$$

está encorpada na matriz

$$\begin{pmatrix} 5 & 1 & 3 \\ 2 & -1 & 4 \end{pmatrix}$$

e que a solução do sistema pode ser obtida efetuando operações apropriadas nessa matriz. Isto é particularmente importante no desenvolvimento de programas de computador para resolver sistemas de equações lineares dentre outras várias aplicações científicas.

Nós vimos que podemos representar arrays multidimensionais em Python usando listas aninhadas. Para criar uma matriz de $n * m$ elementos podemos primeiro criar uma lista de n elementos (por exemplo, de n zeros) e, em seguida, fazer de cada um dos elementos um link para outra lista unidimensional de m elementos. Por exemplo:

```
n = 3
m = 4
a = [0] * n
for i in range(n):
    a[i] = [0] * m
print(a)
```

Saída: `[[0, 0, 0], [0, 0, 0], [0, 0, 0]]`

Uma outra alternativa é usarmos comprehensão de listas. Por exemplo, para criar um array de dimensões $d_1 \times d_2 \dots \times d_k$ inicialmente vazio, fazemos:

`[[[] for i_{k-1} in range(d_{k-1})...] for i_2 in range(d_2)] for i_1 in range(d_1)]`

Abaixo, um exemplo de um array de dimensões $3 \times 4 \times 5$, com todos os elementos iguais a zero:

```
mat = [[[0 for j in range(5)] for j in range(4)] for i in range(3)]
print(mat)
```

Saída:

```
[[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]],
 [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]],
 [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]]
```

Exercício:

Crie um conjunto de funções para realizar as operações básicas sobre matrizes quadradas:

- Soma de 2 matrizes de dimensões $n \times n$.
- Subtração de 2 matrizes de dimensões $n \times n$.
- Cálculo da transposta de uma matriz de dimensão $n \times n$.
- Multiplicação de 2 matrizes com dimensões $n \times n$.

Dicas:

- Na soma de matrizes quadradas, para cada posição (i,j) fazemos:

$$mat_3[i][j] = mat_1[i][j] + mat_2[i][j]$$

- e na multiplicação de matrizes fazemos:

$$mat_3[i][j] = \sum_{k=0}^{n-1} mat_1[i][k] * mat_2[k][j]$$

12.1 Biblioteca Numpy

Listas não é a estrutura de dados mais adequada para operações com matrizes. Python possui uma biblioteca, chamada NumPy (Numerical Python), que contém tipos para representar vetores e matrizes juntamente com diversas operações, dentre elas operações comuns de álgebra linear e transformadas de Fourier. NumPy é implementado para trazer maior eficiência do código em Python para aplicações científicas.

Você pode instalar o pacote Numpy via terminal (ou prompt de comando, no windows):

```
$ pip3 install --upgrade numpy
```

O principal objeto de NumPy é a matriz multidimensional homogênea. Trata-se de uma tabela de elementos (geralmente números), todos do mesmo tipo, indexados por uma tupla de números inteiros positivos. A tipo matriz de NumPy é chamada `ndarray`. Também é conhecido pelo alias `array`. Observe que `numpy.array` não é o mesmo que o tipo `array.array` da biblioteca padrão do Python, que apenas administra arrays unidimensionais e oferece menos funcionalidades.

12.2 Criação de Matrizes

Existem várias maneiras de criar arrays. Por exemplo, você pode criar uma matriz a partir de uma lista Python ou tupla regular usando a função `array`. O tipo da matriz resultante é deduzido do tipo de elementos nas seqüências.

```
import numpy as np
a = np.array([2,3,4])
print(a)

b = np.array([1.2, 3.5, 5.1])
print(b)

c = np.array([(1.5,2,3), (4,5,6)])
print(c)
```

Saída:

```
[2 3 4]
```

```
[1.2 3.5 5.1]
```

```
[[1.5 2. 3. ]
 [4. 5. 6. ]]
```

Muitas vezes, os elementos de uma matriz são originalmente desconhecidos, mas seu tamanho é conhecido. Portanto, NumPy oferece várias funções para criar arrays com conteúdo inicial, como a função `zeros` que cria uma matriz cheia de zeros, a função `ones` que cria uma matriz cheia de uns e a função `empty` cria uma matriz cujo conteúdo inicial é aleatório e depende do estado da memória. Veja os exemplos abaixo:

```
np.zeros( (3,4) )
```

Saída:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
np.ones( (2,3,4), dtype=np.int16 )    # dtype tambem pode ser especificado
```

Saída:

```
array([[[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]],

      [[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]]], dtype=int16)
```

```
np.empty( (2,3) )
```

Saída:

```
array([[-2.68156159e+154, -2.68156159e+154,  2.17231289e-314],
       [ 2.17009491e-314,  2.17232352e-314,  2.17009616e-314]])
```

Para criar sequências de números, NumPy fornece uma função análoga ao `range` que retorna arrays em vez de listas:

```
np.arange( 0, 2, 0.3 )      # estamos usando argumento do tipo float
```

Saída: array([0., 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])

Um array pode ser criado com mais do que uma dimensão utilizando as funções `arange` e `reshape`:

```
a = np.arange(10).reshape(2,5)
```

Saída:

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
a.reshape(5,2)
```

Saída:

```
array([[0, 1,
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

12.3 Operações Básicas

Operadores aritméticos (`*`, `-`, `+`, `/`, `**`) em matrizes aplicam-se de forma elementar. Uma nova matriz é criada e preenchida com o resultado:

```
a = np.array([20, 29, 38, 47])
a ** 2
```

Saída: array([400, 841, 1444, 2209])

Operadores diádicos, como `+=` e `*=`, atuam *in place* para modificar uma matriz existente em vez de criar uma nova:

```
a = np.ones((2,3), dtype=int)
a *= 3
```

Saída:

```
array([[3, 3, 3],
       [3, 3, 3]])
```

Como visto e, ao contrário de muitas linguagens que operam com matrizes, o operador do produto `*` também opera de forma elementar em matrizes numPy. A multiplicação de matrizes pode ser realizada usando a função ou método `dot`:

```
A = np.array( [[1,1],
              [0,1]] )
B = np.array( [[2,0],
              [3,4]] )
A*B                                # multiplicação elemento x elemento
```

Saída:

```
array([[2, 0],
       [0, 4]])
```

```
A = np.array( [[1,1],
              [0,1]] )
B = np.array( [[2,0],
              [3,4]] )
A.dot(B)                            # multiplicação de matrizes
np.dot(A, B)                        # outra forma de multiplicar
```

Saída:

```
array([[5, 4],
       [3, 4]])
```

NumPy oferece operações de álgebra linear no pacote `linalg`. Por exemplo, a função `inv` calcula a inversa de uma matriz:

```
from numpy import *
from numpy.linalg import inv
a = array([[2, 7, 2], [0, 1, 3], [1, 0, 2]])
b = inv(a)
```

Saída:

```
array([[ 0.08695652, -0.60869565,  0.82608696],
       [ 0.13043478,  0.08695652, -0.26086957],
       [-0.04347826,  0.30434783,  0.08695652]])
```

Na biblioteca existe uma variedade de outras funções, por exemplo, para calcular autovalores e autovetores, resolução de um sistema de equações lineares, etc. Para mais informações, consulte a página web: <http://www.numpy.org/>

12.4 Exercícios:

1. Considere o vetor $[1, 2, 3, 4, 5]$. Construa um novo vetor com 3 zeros consecutivos intercalados entre cada valor.
2. Dadas duas matrizes A e B de $n \times n$, obter a soma dos elementos que corresponde à diagonal de $A.dot(B)$ por dois métodos diferentes.

Capítulo 13

Conjuntos e Dicionários

Em tipos abstratos de dados, nós cobrimos até agora as seqüências que são coleções ordenadas de valores (e.g., strings, tuplas e listas). Em uma coleção ordenada, tanto o valor quanto a posição de cada item são significativos e cada item é acessado por sua posição. Nesta aula, vamos estudar as coleções não ordenadas. Do ponto de vista do usuário, apenas os valores dos itens são importantes; para o usuário, a posição de um item não é um problema. Assim, nenhuma das operações em uma coleção não ordenada é baseada em posição. Uma vez adicionado, um item é acessado pelo seu valor. Os usuários podem inserir, recuperar ou remover itens de coleções não ordenadas, mas eles não podem acessar o i -ésimo item, o próximo item ou o item anterior. Alguns exemplos de coleções não ordenadas são conjuntos e dicionários.

13.1 Conjuntos

Como vocês aprenderam com o estudo de matemática, um conjunto é uma coleção de itens sem nenhuma ordem particular. Do ponto de vista do usuário, os itens em um conjunto são únicos. Ou seja, não há itens duplicados em um conjunto. Em matemática, realizamos muitas operações em conjuntos. Algumas das operações mais típicas são as seguintes:

- Retorne o número de itens no conjunto.
- Teste o conjunto vazio (um conjunto que não contém itens).
- Adicione um item ao conjunto.
- Remova um item do conjunto.
- Teste a associação do conjunto (se um determinado item está ou não no conjunto).
- Obtenha a união de dois conjuntos. A união de dois conjuntos A e B é um conjunto que contém todos os itens em A e todos os itens em B.
- Obtenha a interseção de dois conjuntos. A interseção de dois conjuntos A e B é o conjunto de itens em A que também são itens em B.
- Obtenha a diferença de dois conjuntos. A diferença de dois conjuntos A e B é o conjunto de itens em A que não são também itens em B.
- Teste um conjunto para determinar se outro conjunto é ou não seu subconjunto. O conjunto B é um subconjunto do conjunto A se e somente se B for um conjunto vazio ou todos os itens em B também estão em A.

Observe que as operações de diferença e subconjunto não são simétricas. Por exemplo, a diferença dos conjuntos A e B nem sempre é a mesma diferença entre os conjuntos B e A.

Para descrever o conteúdo de um conjunto, usamos a notação { ... }, mas assumimos que os itens não estão em ordem particular. Exemplo:

```
s = {1, 3, 5, 7}
print(s, type(s))
```

Saída: {1, 3, 5, 7} <class 'set'>

No próximo exemplo, criamos dois conjuntos denominados A e B e realizamos algumas operações neles. Quando o construtor de conjunto recebe uma lista como um argumento, os elementos da lista são copiados para o conjunto, omitindo itens duplicados:

```
S = [0, 1, 1, 2]
A = set(S)
print(A)
B = set()
print(1 in A)
print(B)
C = {}
```

Saída:

{0, 1, 2}

True

set()

```
C = A.intersection(B)
print(C)
B.add(1)
B.add(1)
B.add(5)
print(B)
```

Saída:

set()

{1, 5}

```
A.intersection(B)
```

Saída: {1}

```
A.union(B)
```

Saída: {0, 1, 2, 5}

```
A.difference(B)
```

Saída: {0, 2}

```
B.remove(1)
```

Saída: {5}

```
B.issubset(A)
```

Saída: False

Ao contrário de uma lista, um conjunto não permite nenhum acesso baseado em índice. Vocês podem estar se perguntando como um programador pode visitar todos os itens em um conjunto depois de terem sido adicionados. Observe que a classe `set` inclui um iterador, que permite ao programador usar um laço `for` em um conjunto para visitar seus itens em uma ordem não especificada:

```
for item in A:  
    print(item, end=" ")
```

Saída: 0 1 2

13.2 Dicionários

Dicionários são estruturas de dados que implementam mapeamentos. Um mapeamento é uma coleção de associações entre pares de valores. O primeiro elemento do par é chamado de chave (`key`) e o outro de conteúdo (`value`). De certa forma, um mapeamento é uma generalização da idéia de acessar dados por índices, exceto que num mapeamento os índices (ou chaves) podem ser de qualquer tipo imutável.

13.2.1 Motivação

As listas organizam seus elementos por posição. Este modo de organização é útil quando você deseja localizar o primeiro elemento, o último elemento ou visitar cada elemento em uma sequência. Suponha, por exemplo, que precisamos armazenar as temperaturas de três cidades: Campinas, Londres e Paris. Para este propósito, podemos usar uma lista:

```
temps = [23, 15.4, 17.5]
```

Mas, então, precisamos lembrar a sequência de cidades, por exemplo, o índice 0 corresponde à cidade de Blumenau, o índice 1 a Londres e o índice 2 a Paris. Ou seja, a temperatura de Londres é obtida como `temps[1]`. Um **dicionário** com o nome da cidade como índice é mais conveniente, porque isso nos permite escrever `temps["Londres"]` para procurar a temperatura em Londres. Esse dicionário é criado por uma das duas instruções a seguir.

```
temps = {'Blumenau': 23, 'Londres': 15.4, 'Paris': 17.5}  
# ou  
temps = dict(Blumenau=23, Londres=15.4, Paris=17.5)  
  
print(temps['Blumenau'])
```

Saída: 23

Pares adicionais de valor de texto podem ser adicionados quando desejado. Nós podemos, por exemplo, escrever

```
temps['Madri'] = 26.0
```

O dicionário de temperaturas tem agora quatro pares de valores de texto e sua impressão resulta em:

```
print(temps)
```

Saída: {'Blumenau': 23, 'Londres': 15.4, 'Paris': 17.5, 'Madri': 26.0}

13.2.2 Exemplo

Em algumas situações, a posição de um dado em uma estrutura é irrelevante. Neste caso estariamos interessados em sua associação com algum outro elemento na estrutura. Por exemplo, suponha que queiramos recuperar o curso e o respectivo coeficiente de rendimento dos alunos do IFC. Você pode usar uma lista separada para cada item. Note que cada lista deve ter o mesmo comprimento. As informações são armazenadas em listas no mesmo índice, onde cada índice refere-se a informações para uma pessoa diferente. A partir disso, podemos então definir uma função para recuperar as informações do curso e média geral do aluno. O exemplo abaixo mostra esta abordagem:

```
# listas com os itens
nomes = ['Ana', 'Paulo', 'Denise', 'Katia']
notas = [7.5, 8.2, 6.9, 8.5]
cursos = ['computação', 'eng. mecânica', 'eng. civil', 'eng. elétrica']

# função para recuperar as informações
def get_curso_e_nota(aluno, lista_nomes, lista_notas, lista_cursos):
    i = lista_nomes.index(aluno)
    nota = lista_notas[i]
    curso = lista_cursos[i]
    return (curso, nota)

# exemplo de utilização
curso, nota = get_curso_e_nota('Denise', nomes, notas, cursos)
print("Denise estuda {} e tem média geral {}".format(curso, nota))
```

Saída: Denise estuda eng. civil e tem média geral 6.9

Esta solução tem problemas se tivermos muitas informações diferentes para gerenciar. Devemos manter muitas listas e passá-las como argumentos. Temos sempre que indexar as listas por números inteiros e, além disso, devemos nos lembrar de atualizar informações em várias listas. Seria bom indexar o item de interesse diretamente (nem sempre um inteiro) e seria bom usar uma estrutura de dados, sem listas separadas. Novamente, **dicionários** é a solução:

Considere que queiramos representar as informações de um estudante:

```
aluno = {'nome': 'Paulo', 'idade': 21, 'mat': ['MC102', 'MA111']}
```

Uma variável do tipo dicionário pode ser “indexada”

```
print(aluno['nome'])
```

Saída: Paulo

O conteúdo associado a uma chave pode ser alterado atribuindo-se um novo valor àquela posição do dicionário:

```
aluno['idade'] = 22
print(aluno)
```

Saída: {'nome': 'Paulo', 'idade': 22, 'mat': ['MC102', 'MA111']}

13.2.3 Método get

O método `get` serve para obter o conteúdo associado a determinada chave de um dicionário. Sua sintaxe é:

```
get (chave, valor)
```

Ao contrário do acesso utilizando diretamente a `chave`, a utilização do método `get` não causa erro caso a `chave` não exista. Ao invés disso, o método retorna o `valor`. Se o `valor` não for especificado, o método retorna `None`. Veja os exemplos abaixo:

```
aluno = {'nome': 'Paulo', 'idade': 21, 'mat': ['MC102', 'MA111']}
print(aluno['email'])
```

Saída:

```
Traceback (most recent call last):
  File "/Users/rodacki/Developer/exemplo.py", line 25, in <module>
    print(aluno['email'])
KeyError: 'email'
```

```
aluno = {'nome': 'Paulo', 'idade': 21, 'mat': ['MC102', 'MA111']}
print(aluno.get('email', 'Não existe.'))
print(aluno.get('email'))
```

Saída:

```
Não existe.
None
```

Novos valores podem ser acrescentados a um dicionário fazendo atribuição a uma chave ainda não definida:

```
aluno['fone'] = '9-9191-1919'
print(aluno)
```

Saída:

```
{'nome': 'Paulo', 'idade': 22, 'mat': ['MC102', 'MA111'], 'fone': '9-9191-1919'}
```

13.2.4 A Função dict

A função `dict` é usada para construir dicionários e requer como parâmetros:

- Uma lista de tuplas, cada uma com um par chave/conteúdo, ou
- Uma seqüência de itens no formato chave=valor. Nesse caso, as chaves têm que ser strings, mas são escritas sem aspas.

```
d = dict([(1, 2), ('chave', 'conteudo'), ('cursos', ['MC102', 'MC300'])])
print(d[1])
print(d['chave'])
print(d)
```

Saída:

2

conteudo

{1: 2, 'chave': 'conteudo', 'cursos': ['MC102', 'MC300']}

13.3 Chaves e Conteúdos (valores) de Dicionários

Valores:

- podem ser de qualquer tipo (imutável e mutável)
- podem ser duplicados
- os valores do dicionário podem ser listas, ou mesmo outros dicionários!

Chaves:

- deve ser único;
- deve ser de algum tipo imutável (int, float, string, tuple, bool)
 - cuidado com o a escolha do tipo de float como uma chave
 - as tuplas podem ser usados como chaves se elas contêm apenas listas, números ou tuplas; se uma tupla contiver qualquer objeto mutável, direta ou indiretamente, não pode ser usada como uma chave.

13.3.1 Métodos items, keys e values

- `items()` retorna um objeto iterável para os pares chave/conteúdo do dicionário
- `keys()` retorna um objeto iterável para as chaves do dicionário
- `values()` retorna um objeto iterável para os valores do dicionário

Para visualizar os pares, chaves e valores, vocês podem usar o construtor `list`. `list` retorna uma lista de todas as chaves usadas no dicionário, em ordem arbitrária. Se desejar que ele seja ordenado, basta usar `sorted`.

Para verificar se uma única chave está no dicionário, use o operador `in`. O código a seguir exemplifica o uso destes métodos:

```
dic = { "Nick": "pet", "Maria": "girl", "Joao": "boy" }
print(list(dic.items()))
print(list(dic.keys()))
print(sorted(dic.values()))
print('Maria' in dic)
```

Saída:

```
[('Nick', 'pet'), ('Maria', 'girl'), ('Joao', 'boy')]
['Nick', 'Maria', 'Joao']
['boy', 'girl', 'pet']
True
```

13.3.2 Método copy

O método `copy` retorna um outro dicionário com os mesmos pares chave/conteúdo. Observe que os conteúdos não são cópias, mas apenas referências para os mesmos valores.

```
dic = { "Nick": "pet", "Maria": "girl", "Joao": "boy" }
x = dic.copy()
```

```
y = dic
print(x)
print(y)
```

Saída:

```
{'Nick': 'pet', 'Maria': 'girl', 'Joao': 'boy'}
{'Nick': 'pet', 'Maria': 'girl', 'Joao': 'boy'}
```

@@@ tutor

13.3.3 Método clear

O método `clear` remove todos os elementos do dicionário:

```
dic = { "Nick": "pet", "Maria": "girl", "Joao": "boy" }
x = dic.copy()
y = dic
dic.clear()
print(dict)
print(x)
print(y)
```

Saída:

```
<class 'dict'>
{'Nick': 'pet', 'Maria': 'girl', 'Joao': 'boy'}
{}
```

13.4 Compreensão de dicionários

As comprehensões de dicionários podem ser usadas para criar dicionários a partir de expressões de chaves e valores arbitrárias. Por exemplo:

```
{x: x**2 for x in (2, 4, 6)}
```

Saída: {2: 4, 4: 16, 6: 36}

13.5 Iterando em dicionários

Ao fazer uma iteração sobre dicionários, a chave e o conteúdo (ou valor) correspondente podem ser recuperados ao mesmo tempo usando o método `items()`:

```
cavaleiros = { 'Galahad' : 'O Valente',
               'Gareth' : 'O Franco',
               'Lancelot': 'O Cavaleiro Branco',
               'Ivain'   : 'O Cavaleiro do Leão',
               'Gauvain' : 'O Falcão' }
for k, v in cavaleiros.items():
    print(k, v, sep=', ')
```

Saída:

```
Galahad, O Valente
Gareth, O Franco
Lancelot, O Cavaleiro Branco
Ivain, O Cavaleiro do Leão
Gauvain, O Falcão
```

13.5.1 Exercício: Contando letras em um String

1. Faça uma função chamada `contaletra` que, dada um string, retorna a letra mais comum nesse string (em caso de empate retorne qualquer uma das mais frequentes).
 - Idéia: usar um dicionário para contar cada letra.
 - A letra é a chave do dicionário, e o valor será quantas vezes a letra foi encontrada.
2. Modifique a função `contaletra` para que ela não conte brancos e pontuação como letras. Conte as letras maiúsculas e minúsculas como as mesmas letras. Dica: use a função `lower` para transformar maiúsculas em minúsculas.

13.6 Ordenando dicionários por chave e valor

Embora os pares de chave-valores estejam em uma certa ordem na instrução de instanciação, chamando o método de lista (que criará uma lista de suas chaves), podemos ver que eles não necessariamente são armazenados ordenados (no exemplo, em ordem alfabética). Neste caso, se quisermos exibir as chaves em ordem, podemos usar a função `sorted`. Para o caso de mostrarmos os valores em ordem, devemos chamar `sorter` associado ao método `values()`. Veja os exemplos a seguir:

```
numbers = {'first': 1, 'second': 2, 'third': 3, 'Fourth': 4}
print(list(numbers))
print(sorted(numbers))    # Equivalente a chamar sorted(numbers.keys())
print(sorted(numbers.values()))
```

Saída:

```
['first', 'second', 'third', 'Fourth']
['Fourth', 'first', 'second', 'third']
[1, 2, 3, 4]
```

13.7 Ordenação Customizada

Se simplesmente fornecermos à função `sorted` as chaves e valores do dicionário como argumentos, ele irá executar uma ordenação simples, mas utilizando seus outros argumentos (isto é, `key` e `reverse`), podemos fazer com que ele execute tipos mais complexos.

O argumento `key` (não confundir com as chaves do dicionário), quando usado para classificação, nos permite definir funções específicas para usar ao ordenar os itens, como um iterador (em nosso objeto dicionário). Nos dois exemplos acima, as chaves e os valores eram os itens a serem classificados e os itens usados para comparação, mas se quiséssemos classificar nossas chaves do dicionário usando nossos valores, então diríamos à função `sorted` para fazer isso através de seu argumento `key`. Como segue:

```
numbers = {'first': 1, 'second': 2, 'third': 3, 'Fourth': 4}
print(sorted(numbers, key=lambda x: (numbers[x], x)))
```

Saída: ['first', 'second', 'third', 'Fourth']

13.7.1 Exercícios

1. Escreva uma função que retorna a palavra mais comum de uma string:
 - Use o método `split()` para quebrar a string em uma lista de palavras.
 - Use as palavras como chaves do dicionário.

- Converta cada palavra para minúsculo com a função `lower`.
 - Remova os caracteres de pontuação das palavras (mais difícil).
2. Uma data no formato “8-Mar-15” inclui o nome do mês, que deve ser traduzido para um número. Crie um dicionário adequado para decodificar nomes de mês para números. Crie uma função que use operações de string para dividir a data em 3 itens usando o caractere “-”. Traduzir o mês, corrigir o ano para incluir todos os dígitos. A função deverá aceitar uma data no formato “dd-MMM-yy” e responderá com uma tupla de (y, m, d).

Capítulo 14

Arquivos

14.1 Arquivos Texto

Até agora, nós vimos no curso, exemplos de programas que obtiveram os dados de entrada de usuários via teclado. A maioria desses programas pode receber seus dados de entrada de arquivos texto também. Um arquivo texto é um objeto de software que armazena dados em um meio permanente, como um disco, CD ou memória flash. Quando comparado à entrada de dados via teclado, as principais vantagens de se obter dados de entrada de um arquivo são as seguintes:

- O conjunto de dados pode ser muito maior.
- Os dados podem ser inseridos muito mais rapidamente e com menos chance de erro.
- Os dados podem ser usados repetidamente com o mesmo programa ou com diferentes programas.

Um nome e caminho únicos são usados por usuários ou em programas ou scripts para acessar um arquivo texto para fins de leitura e modificação. As tarefas básicas envolvidas na manipulação de arquivos são ler dados de arquivos e escrever ou anexar dados em arquivos.

Leitura e escrita em arquivos em Python são muito fáceis de gerenciar. No nosso primeiro exemplo, vamos mostrar como ler dados de um arquivo texto. A maneira de dizer ao Python que queremos ler de um arquivo é usar a função `open`. O primeiro parâmetro é o nome do arquivo que queremos ler e com o segundo parâmetro, atribuído ao valor “r”, afirmamos que queremos ler do arquivo.

O “r” é opcional. Um comando `open` com apenas um nome de arquivo é aberto para leitura por padrão. A função `open` retorna um objeto de arquivo, que oferece métodos e atributos de arquivo. Exemplo:

```
fobj = open("/Users/rodacki/Developer/files/poema.txt", "r")
print(fobj)
```

Saída:

```
<_io.TextIOWrapper name='/Users/rodacki/Developer/files/poema.txt' mode='r' encoding='UTF-8'>
```

Depois de termos finalizado o trabalho com um arquivo, devemos fechá-lo novamente usando o método do objeto do arquivo `close`:

```
fobj.close()
```

Agora vamos finalmente abrir e ler os dados contidos em um arquivo. O método `rstrip` no exemplo a seguir é usado para retirar os espaços em branco (incluindo o caracter de nova linha ou *newline*) do lado direito da string “line”:

```
fobj = open("/Users/rodacki/Developer/files/poemafp.txt", "r")
for line in fobj:
    print(line.rstrip())
```

```
fobj.close()
Saída:
O poeta é um fingidor.
Finge tão completamente
Que chega a fingir que é dor
A dor que deveras sente.
```

Fernando Pessoa

14.1.1 Escrevendo em um arquivo texto

O seguinte código abre um objeto de arquivo de escrita em um arquivo chamado “files/myfile.txt”. Se o arquivo não existir, ele será criado com o nome do caminho fornecido. Se o arquivo já existe, o Python o abre. Quando os dados são gravados no arquivo e o arquivo é fechado (`close`), todos os dados anteriormente existentes no arquivo são apagados. Os dados de seqüência de caracteres (strings) são escritos (ou gravados) para um arquivo usando o método de gravação com o objeto de arquivo. O método de escrita espera um único argumento de string. Se você quiser que o texto de saída termine com uma nova linha, você deve incluir o caractere de escape `\n` na string. O exemplo a seguir escreve duas linhas de texto no arquivo:

```
f = open("/Users/rodacki/Developer/files/myfile.txt", 'w')
f.write("First line.\nSecond line.\n")
f.close()
```

Saída (conteúdo do arquivo):

First line.
Second line.

Assim como na leitura, quando todas as saídas forem concluídas, o arquivo deve ser fechado usando o método `close`. A falha ao fechar um arquivo de saída pode resultar na perda de dados.

14.1.2 Escrevendo números em um arquivo texto

Os dados em um arquivo texto podem ser vistos como caracteres, palavras, números ou linhas de texto, dependendo do formato do arquivo texto e dos propósitos para os quais os dados são usados. Quando os dados são tratados como números (inteiros ou pontos flutuantes), eles devem ser separados por espaços em brancos, tabulações ou caracteres de mudança de linha. Veja no exemplo abaixo como se parece um arquivo de texto contendo seis números de ponto flutuante quando examinado com um editor de texto. Observe que este formato inclui um espaço ou uma nova linha como um separador de itens no texto.

```
34.6 22.33 66.75
77.12 21.44 99.01
```

Todos os dados de saída ou entrada de um arquivo texto devem ser strings. Assim, os números devem ser convertidos em strings antes da saída, e essas strings devem ser convertidas de volta em números após a entrada. Em Python, os valores da maioria dos tipos de dados podem ser convertidos em strings usando a função `str`. As strings resultantes são então escritas em um arquivo com espaço ou uma nova linha como um caractere separador.

O próximo trecho de código ilustra a saída de inteiros para um arquivo texto. Cinco números inteiros aleatórios entre 1 e 500 são gerados e escritos em um arquivo texto chamado “files/integers.txt”. O caractere de *newline* é o separador.

```
import random
```

```
f = open("/Users/rodacki/Developer/files/integers.txt", 'w')
for count in range(5):
    number = random.randint(1, 500)
    f.write(str(number) + "\n")
f.close()
```

Saída (conteúdo do arquivo):

```
278
205
213
162
285
```

14.1.3 Lendo dados de um arquivo texto

Existem várias maneiras de ler dados de um arquivo de entrada. A maneira mais simples é usar o método `read` para inserir todo o conteúdo do arquivo como uma seqüência única em um objeto string. Se o arquivo contiver várias linhas de texto, os caracteres de nova linha (*newline*) serão incorporados nesta string. O exemplo a seguir mostra como usar o método `read`:

```
f = open("/Users/rodacki/Developer/files/integers.txt", 'r')
text = f.read()
print(text)
```

Saída:

```
278
205
213
162
285
```

Após a conclusão da entrada, outra chamada para ler retornará uma string vazia, para indicar que o final do arquivo foi atingido. Para repetir a leitura em uma entrada, o arquivo deve ser reaberto. Não é necessário fechar o arquivo.

Alternativamente, uma aplicação pode ler e processar o texto, uma linha de cada vez. Um laço `for` cumpre isso muito bem. O laço `for` exibe um objeto de arquivo como uma seqüência de linhas de texto. Em cada iteração do laço, a variável de iteração está vinculada à próxima linha de texto na seqüência. No exemplo a seguir, reabrimos o nosso arquivo de exemplo e visitamos as linhas de texto nele contidas:

```
f = open("/Users/rodacki/Developer/files/integers.txt", 'r')
l = 1
for line in f:
    print("Linha", l, line.rstrip())
    l = l + 1
f.close()
```

Saída:

```
Linha 1 278
Linha 2 205
Linha 3 213
Linha 4 162
Linha 5 285
```

Observe que a impressão exibe uma nova linha extra. Isso ocorre porque cada linha de entrada de texto do arquivo mantém seu caractere de *newline*.

Podemos usar o método `readline` nos casos em que desejamos ler um número específico de linhas de um arquivo (digamos, a primeira linha somente). O método `readline` consome uma linha de entrada e retorna essa string (incluindo o *newline*). Se `readline` encontrar o final do arquivo, ele retorna a string vazia. Revisitamos o exemplo anterior e usamos o laço `while` para inserir todas as linhas de texto com `readline`:

```
f = open("/Users/rodacki/Developer/files/integers.txt", 'r')
while True:
    line = f.readline()
    if line == "":
        break
    print(line)
f.close()
```

Saída:

278

205

213

162

285

14.1.4 Lendo números de um arquivo texto

Todas as operações de entrada de arquivo retornam dados para o programa como strings. Se essas strings representarem outros tipos de dados, como números inteiros ou números de ponto flutuante, o programador deve convertê-los para os tipos apropriados antes de manipulá-los. Em Python, as representações de seqüência de números inteiros e números de ponto flutuante podem ser convertidas para os próprios números usando as funções `int` e `float`, respectivamente.

Ao ler dados de um arquivo, outra consideração importante é o formato dos itens de dados no arquivo. Anteriormente, mostramos um exemplo de código que emitia números inteiros, escolhidos de forma aleatória, separados por caracteres de *newline* para um arquivo texto. Durante a entrada, esses dados podem ser lidos com um simples laço `for`. Este laço acessa uma linha de texto em cada iteração. Para converter esta linha para o inteiro contido nele, o programador executa o método `strip` para remover o *newline* e, em seguida, executa a função `int` para obter o valor inteiro. O exemplo abaixo ilustra essa técnica. Ele abre o arquivo de inteiros aleatórios escrito anteriormente, lê-os e imprime sua soma.

```
f = open("/Users/rodacki/Developer/files/integers.txt", 'r')
sum = 0
for line in f:
    line = line.strip()
    number = int(line)
    sum += number
print("A soma é", sum)
f.close()
```

Saída: A soma é 1143

Obter números de um arquivo texto no qual eles são separados por espaços é um pouco mais complicado. Podemos continuar lendo as linhas em um laço `for`, como antes. Mas cada linha agora pode conter vários números inteiros separados por espaços. Você pode usar o método de seqüência `split` para obter uma lista das cadeias de caracteres que representam esses números inteiros e, em seguida, processar cada seqüência desta lista com outro laço `for`.

Modificamos o exemplo anterior para lidar com números inteiros separados por espaços e/ou caracteres de *newline*.

```
f = open("/Users/rodacki/Developer/files/integers.txt", 'r')
sum = 0
for line in f:
    wordlist = line.split()
    for word in wordlist:
        number = int(word)
        sum += number
print("A soma é", sum)
f.close()
```

Saída: A soma é 1143

14.2 Tratamento de exceções

Se abrirmos um arquivo para leitura e ele não existir, ocorrerá um erro:

```
f = open("naoExiste", 'r')
```

Saída:

```
Traceback (most recent call last):
  File "/Users/rodacki/Developer/exemplo.py", line 1, in <module>
    f = open("naoExiste", 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'naoExiste'
```

Exceções são eventos inesperados que ocorrem durante a execução de um programa. Uma exceção pode resultar de um erro lógico ou de uma situação imprevista, como no exemplo anterior. Em Python, as exceções (também conhecidas como erros) são objetos que são criados (ou lançados) por código que encontra uma circunstância inesperada. Uma exceção pode ser capturada por um contexto que “trata” a exceção de forma apropriada. Se não for detectada, uma exceção faz com que o interpretador Python pare de executar o programa e emita uma mensagem apropriada para o console. Vamos examinar os tipos de erro mais comuns em Python e o mecanismo para capturar e manipular erros que foram gerados.

14.2.1 Tipos de Erros mais comuns

O Python inclui uma hierarquia rica de classes de exceções que designam várias categorias de erros. Vários desses erros podem ser levantados em casos excepcionais por comportamentos apresentados neste curso. Por exemplo, o uso de um identificador indefinido em uma expressão faz com que seja lançada uma exceção `NameError`. O envio do número, tipo ou valor errados para os parâmetros de uma função é outra causa comum para uma exceção. Por exemplo, uma chamada para `abs("hello")` causará um `TypeError` porque o parâmetro não é numérico e uma chamada para `abs(3, 5)` causará um `TypeError` porque somente um parâmetro é esperado. Normalmente, um `ValueError` é gerado quando o número e tipo de parâmetros corretos são enviados, mas um valor é ilegítimo para o contexto da função. Por exemplo, o construtor `int` aceita uma string,

como em `int("137")`, mas um `ValueError` é gerado se essa string não representa um inteiro, como acontece com `int("3.14")` ou `int("hello")`.

Os tipos de seqüência de Python (por exemplo, lista, tupla e str) lançam um `IndexError` quando a expressão como `data[k]` é usada com um inteiro `k` que não é um índice válido para a sequência dada. Os conjuntos e os dicionários lançam um `KeyError` quando uma tentativa é feita para acessar um elemento inexistente.

14.2.2 Tratando uma exceção

O objetivo do tratamento de exceções é evitar totalmente a possibilidade de uma exceção ser levantada através do uso de um teste condicional pró-ativo. Na possibilidade de divisão por zero, podemos evitar a situação ofensiva, escrevendo:

```
if y != 0:
    ratio = x / y
else:
    ... faça alguma coisa ...
```

A filosofia, muitas vezes abraçada pelos programadores de Python, é que “é mais fácil pedir perdão do que obter permissão”. Essa citação é atribuída a Grace Hopper, pioneira em informática. O sentimento é que não precisamos gastar tempo de execução extra salvaguardando contra todos os casos excepcionais, desde que haja um mecanismo para lidar com um problema depois que ele surgir.

Em Python, esta filosofia é implementada usando uma estrutura de controle de tentativa-e-erro. Revisando nosso exemplo, a operação de divisão pode ser salvaguardada da seguinte maneira:

```
try:
    ratio = x / y
except ZeroDivisionError:
    ... faça alguma coisa ...
```

Nesta estrutura, o bloco “try” é o código principal a ser executado. Embora seja um único comando neste exemplo, ele geralmente pode ser um bloco maior de código. O bloco “try” pode ser seguido de um ou mais casos “except”, cada um com um tipo de erro identificado e um bloco de código que deve ser executado se o erro designado for lançado no bloco “try”. A vantagem relativa de usar uma estrutura de tentativa-e-erro é que o caso não excepcional funciona de forma eficiente, sem verificações estranhas para a condição excepcional.

O tratamento de exceções é particularmente útil ao trabalhar com a entrada do usuário, ou ao ler ou escrever para arquivos, porque tais interações são inherentemente menos previsíveis. No nosso exemplo para arquivo inexistente podemos fazer:

```
try:
    arq = open("files/naoExiste.txt", "r")
    print("Abri o arquivo com sucesso.")
except FileNotFoundError:
    print("Não foi possível abrir o arquivo.")
print("Fim!")
```

Saída:

`Não foi possível abrir o arquivo.`

`Fim!`

O exemplo abaixo lê um arquivo e mostra o conteúdo dele na tela:

```
try:
    arq = open("/Users/rodacki/Developer/files/versos.txt", "r")
```

```

while True:
    s = arq.read(1)
    print(s, end="")
    if(s == ""):
        break
arq.close()
except:
    print("Arquivo versos.txt não pode ser aberto.")

```

Saída:

Por muito tempo achei que a ausência é falta.
 E lastimava, ignorante, a falta.
 Hoje não a lastimo.
 Não há falta na ausência.
 A ausência é um estar em mim.
 E sinto-a, branca, tão pegada, aconchegada nos meus braços,
 que rio e danço e invento exclamações alegres,
 porque a ausência, essa ausência assimilada,
 ninguém a rouba mais de mim.

Carlos Drummond de Andrade

14.3 Editando Arquivos texto

Com o modo de operação “r+” (ver tabela abaixo), podemos ler todo o texto de um arquivo e fazer qualquer alteração que julgarmos necessária. O texto alterado pode então ser sobreescrito sobre o texto anterior. Ao realizar a leitura de um caractere, ou uma linha, automaticamente o indicador de posição do arquivo se move para o próximo caractere (ou linha). Para voltar ao início do arquivo novamente você pode usar o método `seek`:

`seek(offset, from_what)`

onde o primeiro parâmetro indica quantos bytes se mover a partir do valor inicial `from_what`. Os valores de `from_what` podem ser: 0: indica início do arquivo. 1: indica a posição atual no arquivo. 2: indica a posição final do arquivo.

Os modos de abertura de arquivo texto e o indicador de posição são:

modo	operações	indicador de posição
r	leitura	início do arquivo
r+	leitura e escrita	início do arquivo
w	escrita	início do arquivo
a	(append) escrita	final do arquivo

14.3.1 Exemplo

No arquivo `poema2.txt` cujo conteúdo é exibido abaixo, existe um tratamento em terceira pessoa quando o autor pretendia que o mesmo fosse em segunda pessoa.

Nos momentos tristes busco pelo seu sorriso
 Pois sei que um dia sentiremos juntos então
 que valeu a pena suportar a dor da ausência,

somente pela alegria de nosso reencontro.

Para modificá-lo, o exemplo de código a seguir procura no texto a palavra “seu” e substitui por “teu”.

```
try:
    f = open("/Users/rodacki/Developer/files/poema2.txt", "r+")
    text = ""
    while True:
        line = f.readline()
        if line == "":
            break
        if "seu" in line:
            line = line.replace("seu", "teu")
        text += line
    f.seek(0, 0)
    f.write(text)
    f.close()
    print("Texto modificado!")
except IOError:
    print("Problemas no arquivo poema2.txt")
```

Resultado:

Nos momentos tristes busco pelo teu sorriso
 Pois sei que um dia sentiremos juntos então
 que valeu a pena suportar a dor da ausência,
 somente pela alegria de nosso reencontro.

14.4 Parâmetros do Programa

É possível um programa em Python receber parâmetros diretamente da linha de comando quando o programa é executado. Para isso devemos importar o módulo `sys` e ler os dados armazenados na lista `sys.argv`. O primeiro parâmetro na lista `sys.argv` é o nome do arquivo que contém o programa. Os demais parâmetros aparecem na mesma ordem em que foram digitados na linha de comando. O programa abaixo imprime os parâmetros da linha de comando, um por linha:

```
import sys
print("Voce executou o programa com ", len(sys.argv), " parametros!")
print("Os parâmetros foram")
for p in sys.argv:
    print(p)
```

Exemplo de execução em linha de comando:

```
Paulos-MacBook-Pro:files rodacki$ python3 exemplo.py teste1 teste2
Voce executou o programa com 3 parametros!
Os parametros foram
exemplo.py
teste1
teste2
```

Seu uso é útil em programas onde dados de entrada são passados via linha de comando. Por exemplo, quando os dados a serem processados estão em um arquivo, cujo nome é passado para o programa via linha de comando. O programa a seguir mostra o conteúdo de um arquivo cujo nome do mesmo é passado como parâmetro:

```
if len(sys.argv) != 2:
```

```

    print("Usage: python more.py nome_do_arquivo")
else:
    try:
        arq = open(sys.argv[1], "r")
        while True:
            t = arq.readline()
            print(t, end="")
            if t == "":
                break
        arq.close()
    except:
        print("Arquivo não existe!")

```

Exemplo de execução em linha de comando:

```

Paulos-MacBook-Pro:files rodacki$ python3 exemplo.py poemafp.txt
O poeta é um fingidor.
Finge tão completamente
Que chega a fingir que é dor
A dor que deveras sente.

```

Fernando Pessoa

14.4.1 Exemplos

1. Somando colunas de um arquivo

Suponha que você tenha um arquivo contendo colunas de números gerados por outro sistema e precise somar os números de cada coluna.

```

1 5 10 2 1.0
2 10 20 4 2.0
3 15 30 8 3
4 20 40 16 4.0

```

Como já vimos, a divisão de strings é a principal operação por trás da solução desse problema, mas como um bônus adicional, vamos usar a **builtin eval** para converter os strings das colunas em números. Ex:

```

x = 1
eval("x + 1")

```

Saída: 2

Solução: o código abaixo soluciona o problema, supondo que os números encontram-se em um arquivo chamado `tabela1.txt`

```

# solução:
import sys

def soma(nCols, fileName):
    sums = [0] * nCols
    for line in open(fileName):
        cols = line.split()
        for i in range(nCols):
            sums[i] += eval(cols[i])
    return sums

# Just to simulate command line call

```

```

sys.argv = ['soma1.py', '5', 'files/tabela1.txt']
# Comment the above line in a real script.

print(soma(eval(sys.argv[1]), sys.argv[2]))

```

Saída: [10, 50, 100, 30, 10.0]

2. Somando com zip e comprehensão de listas

Para este exemplo, os dados estão separados por vírgula ao invés de espaço, por exemplo:

```

1,5,10,2,1
2,10,20,4,2
3,15,30,8,3
4,20,40,16,4

```

Iremos utilizar comprehensão de listas e a função `zip` para evitar laços aninhados. A **builtin** `zip` usa uma ou mais sequências como argumentos e retorna uma série de tuplas que emparelham itens paralelos obtidos dessas sequências. Por exemplo:

```

x = [1, 2, 3]
y = [4, 5, 6]
zipped = zip(x, y)
print(list(zipped))

```

Saída: [(1, 4), (2, 5), (3, 6)]

Solução: o código abaixo soluciona o problema, supondo que os números encontram-se em um arquivo chamado `tabela2.txt`

```

# solução:
import sys

def soma(nCols, fileName):
    sums = [0] * nCols
    for line in open(fileName):
        cols = line.split(',')
        nums = [int(x) for x in cols]
        both = zip(sums, nums)
        sums = [x + y for (x, y) in both]
    return sums

# Just to simulate command line call
sys.argv = ['soma2.py', '5', 'files/tabela2.txt']
# Comment the above line in a real script.

print(soma(eval(sys.argv[1]), sys.argv[2]))

```

Saída: [10, 50, 100, 30, 10]

3. Armazenando Dados de Arquivo em Dicionários Aninhados

Recebemos um arquivo de dados com medições de algumas propriedades com nomes dados (aqui A, B, C ...). Cada propriedade é medida por um determinado número de vezes. Os dados são organizados como uma tabela onde as linhas contêm as medidas e as colunas representam as propriedades medidas:

A	B	C	D	
1	11.7	0.035	2017	99.1
2	9.2	0.037	2019	101.2
3	12.2	no	no	105.2
4	10.1	0.031	no	102.1

```
5 9.1 0.033 2009 103.3
6 8.7 0.036 2015 101.9
```

A palavra `no` significa nenhum dado, ou seja, falta-nos uma medida. Queremos ler esta tabela em um dicionário de dados para que possamos procurar a medida `i` da (digamos) propriedade C como `data['C'][i]`. Para cada propriedade p, queremos calcular a média de todas as medidas e armazená-las como `data[p]['mean']`.

Solução: o código abaixo soluciona o problema, supondo que os números encontram-se em um arquivo chamado `tabela3.txt`

```
#solução
infile = open('files/tabela3.txt', 'r')
lines = infile.readlines()
infile.close()
data = {} # data[property][measurement_no] =
           # propertyvalue
first_line = lines[0]
properties = first_line.split()

for p in properties:
    data[p] = {}

for line in lines[1:]:
    words = line.split()
    i = int(words[0]) # número da medição
    values = words[1:] # valores das propriedades
    for p, v in zip(properties, values):
        if v != 'no':
            data[p][i] = float(v)

# Computando a média das medidas
for p in data:
    values = data[p].values()
    data[p]['mean'] = sum(values) / len(values)

for p in sorted(data):
    print('Valor médio da propriedade %s = %g' % (p, data[p]['mean']))
```

Saída:

```
Valor médio da propriedade A = 10.1667
Valor médio da propriedade B = 0.0344
Valor médio da propriedade C = 2015
Valor médio da propriedade D = 102.133
```

14.5 Arquivos Binários

Arquivos podem ter o mais variado conteúdo (imagens, videos, audios, documentos, etc), mas do ponto de vista dos programas existem apenas dois tipos de arquivo:

Arquivo texto: Armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de textos simples. Exemplos: código fonte Python, documento texto simples, páginas HTML.

Arquivo binário: Sequência de bits sujeita às convenções dos programas que o gerou, não legíveis diretamente. Exemplos: arquivos executáveis, arquivos compactados, documentos do Office.

A motivação principal é que objetos (como inteiros, listas, dicionários) na sua representação em binário, ocupam pouco espaço na memória, quando comparado com sua representação em formato texto:

- Para representarmos o número 123456789.00 na forma textual, teríamos que convertê-lo para strings e gastaríamos 12 bytes para representar este número.
- Sua representação binária, no entanto ocupa sempre 64 bits (ou 8 bytes).

Armazenar objetos em arquivos de forma análoga a utilizada em memória permite:

- Reduzir o tamanho do arquivo.
- Guardar estruturas complexas tendo acesso simples.

Se você usar um editor de texto para abrir um arquivo binário, você verá quantidades abundantes de caracteres acentuados aparentemente aleatórios e caracteres incomuns, e linhas longas transbordando de texto – este exercício é seguro, mas inútil. No entanto, editar ou salvar um arquivo binário em um editor de texto irá corromper o arquivo, então nunca faça isso. A razão pela qual a corrupção ocorre é porque aplicar uma interpretação do modo de texto irá alterar certas sequências de bytes – como descartar bytes NUL, converter linhas novas, descartar sequências que são inválidas sob uma certa codificação de caracteres, etc. – o que significa que abrir e salvar um arquivo binário Provavelmente produzem um arquivo com bytes diferentes.

14.5.1 Abrindo Arquivos Binários

Assim como em arquivos texto, devemos abrir o arquivo com a função `open` e atribuir o objeto arquivo resultante para um nome. Desta forma o nome estará associado ao objeto arquivo, com métodos para leitura e escrita sobre este:

```
arq = open("nome_do_arquivo", modo)
```

onde:

modo	operações
rb	leitura
wb	escrita
r+b	leitura e escrita

14.5.2 Lendo e Escrevendo em Arquivos Binários

Python dispõe de vários métodos para ler e escrever em arquivos binários. Nós utilizaremos neste curso o **Pickle**. O módulo `pickle` implementa protocolos binários para serialização e deserialização de objetos Python. “*Pickling*” é o processo pelo qual uma hierarquia de objetos Python é convertida em um fluxo de bytes, e “*Unpickling*” é a operação inversa, pelo que um fluxo de bytes (de um arquivo binário ou objeto do tipo bytes) é convertido de novo em uma hierarquia de objetos. O *Pickling* (e o *Unpickling*) é alternativamente conhecido como “serialização”, “marshalling,” ou “flattening” no entanto, para evitar confusões, os termos aqui utilizados serão “*Pickling*” e “*Unpickling*”.

Para escrever um objeto em um arquivo binário usamos o método `pickle.dump`.

```
pickle.dump(objeto, var_arquivo)
```

onde:

- `objeto`: este é o objeto a ser salvo em arquivo.
- `var_arquivo`: este é o nome associado a um objeto arquivo previamente aberto em modo binário.

Por exemplo, o programa a seguir salva uma lista em arquivo:

```
import pickle

try:
    arq = open("teste.bin", "wb")
    lst = [x ** 2 for x in range(10)]
    pickle.dump(lst, arq)
    arq.close()
except IOError:
    print("Problemas com o arquivo teste.bin")
```

Para ler um objeto de um arquivo binário usamos o método `pickle.load`.

`var_objeto = pickle.load(var_arquivo)`

onde:

- var arquivo: este é o nome associado a um objeto arquivo previamente aberto em modo binário.
- O método automaticamente reconhece o tipo de objeto salvo em arquivo, carrega este para a memória e associa este objeto ao nome var_objeto.

O programa a seguir lê a lista previamente salva em arquivo:

```
import pickle

try:
    arq = open("teste.bin", "rb")
    lst = pickle.load(arq)
    print(lst)
    arq.close()
except IOError:
    print("Problemas com o arquivo teste.bin")
```

Saída: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

14.5.3 Exemplo

Neste exemplo, vamos criar um programa que simula um cadastro de alunos de uma turma de Algoritmos. Para representar o aluno vamos criar um dicionário com os campos `nome` e `notas` que irão armazenar, respectivamente, o nome do aluno e as notas dos labs de Algoritmos. Para representar o `cadastro` criaremos um objeto do tipo lista que contém a lista de alunos. Além disso, o programa deverá fornecer um menu de operações com as funções de inserção e remoção do aluno no cadastro, a inserção de notas e a visualização da lista de alunos. No término no programa ele deverá persistir o cadastro para futuras consultas e alterações.

As funções abaixo implementam as operações básicas do programa, recebendo como parâmetro o cadastro:

```
def CreateAluno(cadastro, nome):
    aluno = {}
    aluno["nome"] = nome
    aluno["notas"] = []
    cadastro.append(aluno)

def ExcluiAluno(cadastro, nome):
    for aluno in cadastro:
        if aluno["nome"] == nome:
```

```

        cadastro.remove(aluno)
        return
print(nome, " não encontrado!")

def InsertNotas(cadastro, nome, notas):
    for aluno in cadastro:
        if aluno["nome"] == nome:
            aluno["notas"] += notas.split()
            return
    print(nome, " não encontrado!")

def ListaAlunos(cadastro):
    for aluno in cadastro:
        for key in aluno:
            print(key, ":", aluno[key])

```

O menu de opções é implementado abaixo:

```

def Menu(cadastro):
    while True:
        print("\nEscolha uma opção:\n 1- Incluir Aluno\n 2- Excluir Aluno")
        print(" 3- Incluir Notas\n 4- Listar Turma\n 5- Sair\n")
        op = int(input())
        if op == 1:
            nome = input("Digite o nome do aluno: ")
            CreateAluno(cadastro, nome)
        elif op == 2:
            nome = input("Digite o nome do aluno: ")
            ExcluiAluno(cadastro, nome)
        elif op == 3:
            nome = input("Digite o nome do aluno: ")
            notas = input("Digite as notas do aluno separados por espaço:")
        elif op == 4:
            ListaAlunos(cadastro)
        elif op == 5:
            return
        else:
            print("Opção inválida!")

```

Finalmente, no programa principal, tratamos a persistencia dos dados em memória de disco para futuro uso do mesmo.

```

import pickle

try:
    cadastro = pickle.load(open("cadastro.bin", "rb"))
    Menu(cadastro)
    print("\nSalvando Cadastro...")
    pickle.dump(cadastro, open("cadastro.bin", "wb"))
except FileNotFoundError:
    print("Criando Cadastro...")

```

```

cadastro = []
Menu(cadastro)
pickle.dump(cadastro, open("cadastro.bin", "wb"))
except IOError:
    print("Problemas no arquivo de cadastro")

```

Exercício

Modifique o cadastro do exemplo anterior para conter a lista de alunos de cada turma de Algoritmos, a saber: turmas ABCD, EF, GHIJ, KLMN, OP, QRST, UZVX, WY, 4567.

14.6 Acessando e manipulando arquivos e diretórios em disco

Os arquivos são organizados em diretórios (também chamados de “pastas”). Cada programa em execução tem um “diretório atual”, que é o diretório-padrão da maior parte das operações. Nos exemplos anteriores, nós abrimos os arquivos tanto para leitura quanto para escrita na pasta “files” dentro do diretório atual, que no nosso caso é o diretório onde estão armazenados os cadernos de lições.

O módulo `os` de python fornece funções para trabalhar com arquivos e diretórios (“os” é a abreviação de “sistema operacional” em inglês). Por exemplo, `os.getcwd` devolve o nome do diretório atual:

```

import os
cwd = os.getcwd()
print(cwd)

```

Exemplo de execução:

```

Paulos-MacBook-Pro:files rodacki$ python3 exemplo.py
/Users/rodacki/Developer

```

A sigla `cwd` é a abreviação de “diretório de trabalho atual” em inglês (*current working directory*). O resultado neste exemplo é o diretório onde o usuário se encontrava ao solicitar a execução do arquivo fonte Python.

A tabela abaixo fornece algumas das principais funções do módulo `os`:

função	Descrição
<code>chdir (path)</code>	Altera o diretório de trabalho atual para o diretório indicado pela variável ‘path’
<code>getcwd ()</code>	Retorna o caminho (path) do diretório de trabalho atual
<code>listdir (path)</code>	Retorna uma lista dos nomes no diretório indicado por ‘path’
<code>mkdir (path)</code>	Cria um novo diretório e o coloca no diretório de trabalho atual
<code>remove (path)</code>	Remove o arquivo indicado por ‘path’ do diretório de trabalho atual
<code>rename (old, new)</code>	Renomeia o arquivo ou diretório de ‘antigo’ para ‘novo’
<code>rmdir (path)</code>	Remove o diretório de nome ‘path’ do diretório de trabalho atual

Por exemplo, se quisermos alterar o diretório atual para a pasta ‘files’ escrevemos:

```
import os
cwd = os.getcwd()
os.chdir(cwd + "/files")
cwd = os.getcwd()
print(cwd)
print(os.listdir(cwd))
```

Exemplo de execução:

```
/Users/rodacki/Developer/files
['soma1.py', 'vector.py', 'poema.txt', 'tabela3.txt', 'tabela2.txt',
 'tabela1.txt', 'teste.bin', 'poemafp.txt', 'sqrt', 'versos.txt',
 'poema2.txt', 'integers.txt', 'myfile.txt', 'soma2.py']
```

Uma string como ‘/Users/rodacki/Developer/files’, que identifica um arquivo ou diretório, é chamada de caminho (path).

Um nome de arquivo simples, como “poema.txt”, também é considerado um caminho, mas é um caminho relativo, porque se relaciona ao diretório atual. Se o diretório atual é ‘/Users/rodacki/Developer/files’, o nome de arquivo ‘poema.txt’ se refere ao arquivo ‘/Users/rodacki/Developer/files/poema.txt’.

Um caminho que começa com “/” não depende do diretório atual; isso é chamado de caminho absoluto. Para encontrar o caminho absoluto para um arquivo, você pode usar `os.path.abspath`:

```
import os
f = os.path.abspath("poema.txt")
print(os.path.isfile(f))
```

Saída: True

O módulo `os.path` fornece outras funções para trabalhar com nomes de arquivo e caminhos. Por exemplo:

Função	Descrição
<code>join (path, name)</code>	Une dois ou mais componentes do nome do caminho, inserindo ‘/’ se necessário.
<code>exists (path)</code>	Retorna True se o caminho existe e False caso contrário
<code>isdir (path)</code>	Retorna True se ‘path’ é um nome de diretório e False caso contrário
<code>isfile (path)</code>	Retorna True se ‘path’ é um nome de arquivo e False caso contrário

Para demonstrar essas funções, o exemplo seguinte “passeia” por um diretório, exibe os nomes de todos os arquivos e chama a si mesmo recursivamente em todos os diretórios:

```
import os

def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.abspath(name)
        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

```
cwd = os.getcwd()
print(os.listdir(cwd))
walk(cwd)
```

Exemplo de execução:

```
Paulos-MacBook-Pro:files rodacki$ python3 exemplo.py
['soma1.py', 'vector.py', 'poema.txt', 'tabela3.txt', 'tabela2.txt',
 'tabela1.txt', 'teste.bin', 'poemafp.txt', 'sqrt', 'versos.txt',
 'poema2.txt', 'exemplo.py', 'integers.txt', 'myfile.txt', 'soma2.py']
/Users/rodacki/Developer/files/soma1.py
/Users/rodacki/Developer/files/vector.py
/Users/rodacki/Developer/files/poema.txt
/Users/rodacki/Developer/files/tabela3.txt
/Users/rodacki/Developer/files/tabela2.txt
/Users/rodacki/Developer/files/tabela1.txt
/Users/rodacki/Developer/files/teste.bin
/Users/rodacki/Developer/files/poemafp.txt
/Users/rodacki/Developer/files/sqrt
/Users/rodacki/Developer/files/versos.txt
/Users/rodacki/Developer/files/poema2.txt
/Users/rodacki/Developer/files/exemplo.py
/Users/rodacki/Developer/files/integers.txt
/Users/rodacki/Developer/files/myfile.txt
/Users/rodacki/Developer/files/soma2.py
```


Capítulo 15

Pacotes e Módulos

15.1 Reuso de Código

Se tivéssemos que escrever todos os programas a partir do zero, não poderíamos fazer muito. Parte da diversão da programação é usar algo que outra pessoa escreveu para resolver um problema mais rapidamente. Outro aspecto divertido da programação é escrever o código que outros possam reutilizar em seus programas. Considerando que as funções nos permitem “compor” partes de código para que possam ser reutilizadas ao longo de um programa, os módulos fornecem um meio de coletar conjuntos de funções (e como veremos nesta aula, tipos de dados personalizados) juntos para que possam ser usado por qualquer número de programas.

15.1.1 Pacotes e Módulos

Um módulo Python, de maneira sucinta, é um arquivo .py. Um módulo pode conter qualquer código Python que nós gostamos. Todos os programas que escrevemos até agora foram contidos em um único arquivo .py e, portanto, são módulos e programas. A principal diferença é que os programas são projetados para serem executados, enquanto os módulos são projetados para serem importados e usados por programas. Um pacote é simplesmente um diretório que contém um conjunto de módulos que podem estar inter-relacionados ou não. Os pacotes funcionam como coleções para organizar módulos de forma hierárquica. Quando queremos usar uma função de um módulo ou pacote, precisamos importá-lo. Existem muitas formas diferentes de importar módulos e pacotes. A sintaxe mais básica é:

```
import numpy
```

após o qual qualquer função em `numpy` pode ser chamada como `numpy.function()`. Você pode mudar internamente o nome do pacote, por exemplo porque ele é longo ou porque este nome conflita com o nome de um objeto seu. No exemplo abaixo nos alteramos o nome de `numpy` para `np`, o que é bastante padrão na computação científica:

```
import numpy as np
```

Todas as funções em `numpy` podem ser chamadas internamente como `np.function()`. Os pacotes também podem ter subpacotes. Por exemplo, o pacote `numpy` tem um subpacote chamado `random`, que tem um conjunto de funções para lidar com variáveis aleatórias. Se o pacote `numpy` for importado com `import numpy as np`, as funções no sub-pacote `random` podem ser chamadas como `np.random.function()`.

15.1.2 Um pouco sobre pacotes

Python é uma ótima linguagem de programação de propósito geral por conta própria, mas com a ajuda de alguns pacotes populares (e.g., numpy, scipy, matplotlib). Veja mais em [https://pypi.org/]) torna-se um ambiente poderoso para a computação científica. Numpy, que significa Numerical Python, é a principal biblioteca usada na computação científica em Python. Ela fornece um objeto de vetores multidimensionais (matrizes) de alto desempenho e ferramentas para trabalhar com esses vetores.

Se você precisar apenas de uma função específica, não é necessário importar todo o pacote. Por exemplo, se você quiser usar apenas a função coseno do pacote `numpy`, você pode importá-lo a partir de `numpy` assim:

```
from numpy import cos
```

após o qual você pode simplesmente chamar a função coseno como `cos()`. Você pode mesmo renomear as funções quando as importa. Por exemplo:

```
from numpy import cos as coseno
```

após o qual você pode chamar a função `coseno()` para calcular o coseno (eu sei, muito bobo, mas isso pode se tornar útil).

Onde devemos inserir as declarações de importação? É prática comum colocar todas as instruções de importação no início dos arquivos .py, após a documentação do módulo. Além disso, recomendamos importar os módulos de biblioteca padrão do Python primeiro, depois módulos de biblioteca de terceiros e, finalmente, nossos próprios módulos.

15.1.3 Localizando Módulos

Quando importamos pela primeira vez um módulo, se ele não for uma *builtin*, o Python procura o módulo em cada diretório listado na variável de ambiente PYTHONPATH (`sys.path`) que normalmente inclui a pasta corrente em primeiro lugar. Uma consequência disso é que se criarmos um módulo ou programa com o mesmo nome de um dos módulos de biblioteca do Python, o nosso será encontrado primeiro, causando inevitavelmente problemas. Para evitar isso, nunca crie um programa ou módulo com o mesmo nome de um dos diretórios (pacotes) ou módulos de nível superior da biblioteca Python (i.e., não inclui os sub-módulos), a menos que você esteja fornecendo sua própria implementação desse módulo e que o substitua deliberadamente.

Nós podemos incluir o diretório ou diretórios de nossos módulos na variável de ambiente através da biblioteca `sys.path` assim:

```
import sys
sys.path.append('my_path')

import my_lib
```

15.2 Módulos personalizados

Nós abordamos até agora algumas ferramentas de software na resolução de problemas computacionais. A mais importante dessas ferramentas são os mecanismos de abstração para simplificar os projetos e controlar a complexidade das soluções. Os mecanismos de abstração incluem funções, módulos, objetos e classes. Em vários momentos começamos com uma visão externa de um recurso, mostrando o que ele faz e como ele pode ser usado. Por exemplo, para usar a função `sqrt` da *builtin* `math`, você o importa: `from math import sqrt`, executa: `help(sqrt)` para aprender a usar a

função corretamente e então inclua-o apropriadamente em seu código. Os mesmos procedimentos são seguidos para estruturas de dados *builtin*, como strings e listas. Do ponto de vista de um usuário, você não teve que se preocupar com a forma como um recurso é executado. A beleza e a utilidade de uma abstração é que ela liberta você da necessidade de se preocupar com tais detalhes. Infelizmente, nem todas as abstrações são encontradas em builtins. Às vezes, você precisa personalizar uma abstração para atender às necessidades de um aplicativo especializado ou conjunto de aplicativos que você está desenvolvendo. Ao projetar sua própria abstração, você deve ter uma visão diferente da dos usuários e se preocupar com o funcionamento interno de um recurso. O programador que define uma nova função ou constrói um novo módulo de recursos está usando os recursos fornecidos por outros para criar novos componentes de software. Nós vamos mostrar nesta aula como projetar, implementar e testar outro mecanismo de abstração - uma classe. As linguagens de programação que permitem ao programador definir novas classes de objetos são chamados de linguagem orientada a objetos. Essas linguagens também suportam um estilo de programação chamado programação orientada a objetos. Ao contrário da programação baseada em objetos, que simplesmente usa objetos e classes prontas dentro uma estrutura de funções e código algorítmico, a programação orientada a objetos oferece mecanismos para conceber e construir sistemas de software inteiros a partir de classes cooperantes.

Como funções, os objetos são abstrações. Uma função agrupa um algoritmo em uma única operação que pode ser chamada pelo nome. Um objeto envolve um conjunto de valores de dados – seu estado, e um conjunto de operações – seus métodos, em uma única entidade que pode ser referenciada com um nome. Isso torna um objeto uma abstração mais complexa do que uma função. Uma definição de classe é como um modelo para cada um dos objetos dessa classe. Este modelo contém:

- Definições de todos os métodos que seus objetos reconhecem;
- Descrições das estruturas de dados usadas para manter o estado de um objeto, ou seus atributos.

Para ilustrar estas idéias, analisaremos a construção de um tipo abstrato de dados personalizado.

15.2.1 O módulo `vector.py`

O que queremos construir é um módulo que representa um tipo abstrato de dados chamado de vetor euclidiano (também conhecido como vetor geométrico) e operações sobre este. Um vetor é definido como uma entidade que possui tanto a magnitude quanto a direção. O vetor é tipicamente desenhado como uma seta; a direção é indicada por onde a flecha está apontando, e a magnitude pelo comprimento da própria seta. Por questão de simplicidade, vamos trabalhar com vetores de duas dimensões (representados num plano) embora os exemplos podem ser facilmente extendidos para três dimensões.

Uma outra maneira de pensar em um vetor é a diferença entre dois pontos. Considere como você pode fazer instruções para andar de um ponto para outro. A figura 15.1 mostra alguns vetores e possíveis traduções:

Portanto, podemos representar os vetores assim:

```
vector(3, 4)
vector(2, -1)
vector(-15, 3)
```

Antes de continuar a olhar para o tipo `vector` e seus métodos de soma, subtração etc., vamos recordar as principais operações sobre o tipo `vector` usando a notação encontrada em livros de matemática e física.

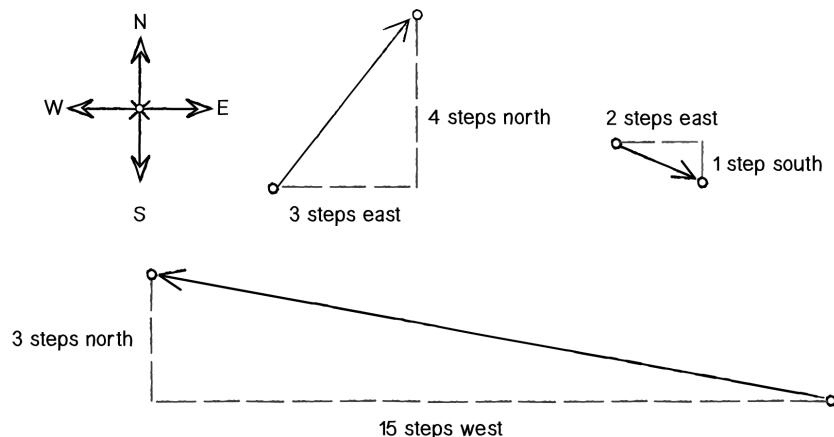


Figura 15.1: Exemplos de vetores em 2D

Vector Add - adição de vetores

Digamos que eu tenho os seguintes dois vetores (figura 15.2):

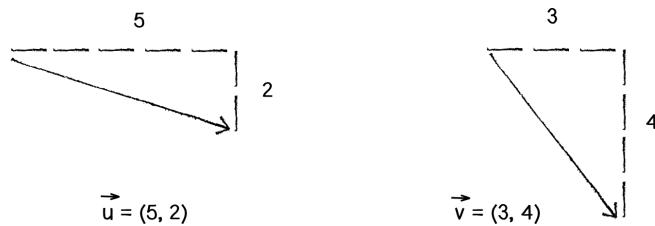


Figura 15.2: Vetores a serem adicionados

Cada vetor tem dois componentes, x e y . Para adicionar dois vetores juntos, simplesmente adicionamos ambos x e os dois y , conforme ilustrado na figura 15.3:

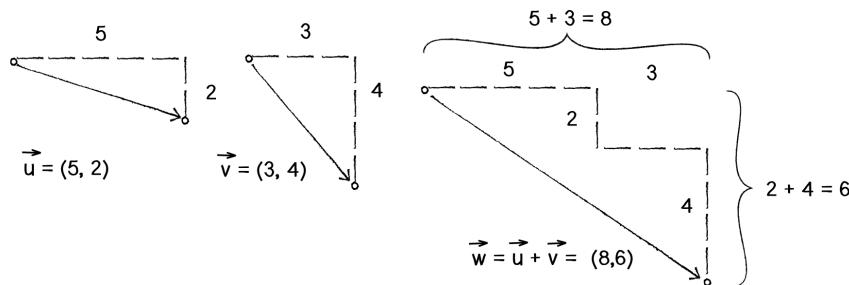


Figura 15.3: Adição de vetores

Vector Subtraction - subtração de vetores

Analogamente, a subtração de dois vetores se dá pela subtração de seus componentes conforme esquematizado na figura 15.4:

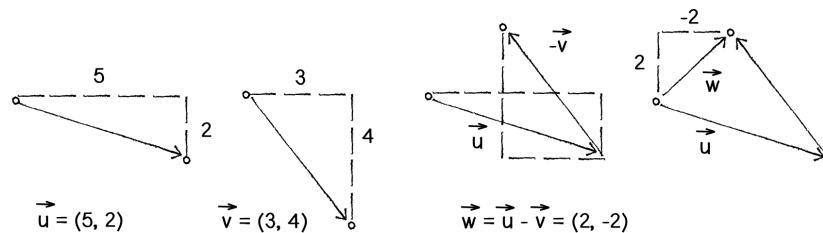


Figura 15.4: Subtração de vetores

Vector Multiplication - multiplicação por um escalar

Quando falamos de vetores, um *escalar* é simplesmente um número. Ou seja, diferente de vetores, os escalares possuem magnitude, mas não possuem uma direção. Uma multiplicação de um vetor por um escalar corresponde a multiplicar cada componente do vetor pelo escalar. Na prática, funciona como alterar o tamanho do vetor na proporção do escalar.

A figura 15.5 ilustra a multiplicação de um vetor \vec{u} pelo escalar 3. Note que o resultado é o vetor \vec{w} cuja magnitude é três vezes maior que do vetor original \vec{u} .

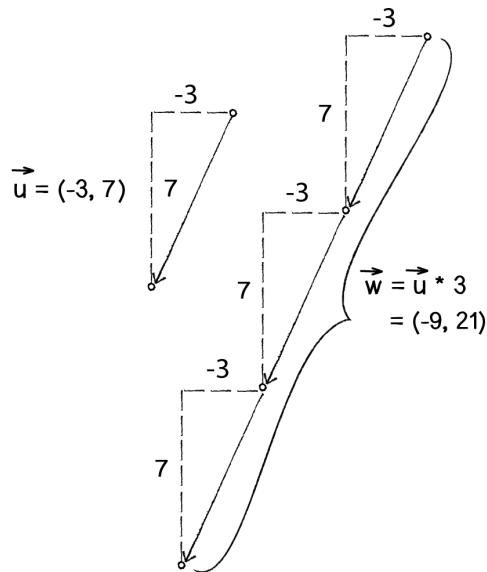


Figura 15.5: Multiplicação de vetor por escalar

Vector Division - divisão de vetor por escalar

A divisão de vetor por escalar funciona de forma análoga à multiplicação - simplesmente substituímos o sinal de multiplicação (*) pelo sinal de divisão (/). A figura 15.6 ilustra esta operação.

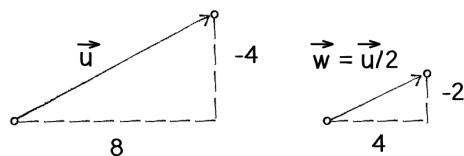


Figura 15.6: Divisão de vetor por escalar

Vector Magnitude - comprimento ou módulo do vetor

A multiplicação e a divisão, como acabamos de ver, são meios pelos quais o comprimento do vetor pode ser alterado sem afetar a direção. Talvez você esteja se perguntando: “OK, então, como eu sei qual é o comprimento de um vetor? Conheço os componentes (x e y), mas qual o comprimento (em pixels) é a seta real?” Compreender como calcular o comprimento (também conhecido como magnitude) de um vetor é incrivelmente útil e importante.

Observe nos diagramas anteriores como o vetor, desenhado como uma seta e dois componentes (x e y), cria um triângulo reto. Os lados são os componentes e a hipotenusa é a própria flecha. Armado com o teorema de Pitágoras podemos calcular a magnitude de v da seguinte maneira:

$$\| v \| = \sqrt{x^2 + y^2}$$

Vector Normalization - normalização de vetores

Para normalizar um vetor, simplesmente dividimos cada componente por sua magnitude. Isso é bastante intuitivo. Digamos que um vetor tem comprimento 5. Bem, 5 dividido por 5 é 1. Então, olhando para o nosso triângulo retângulo, então precisamos escalar a hipotenusa para baixo dividindo-se por 5. Nesse processo os lados diminuirão, divididos por 5 também, conforme exemplificado na figura 15.7:

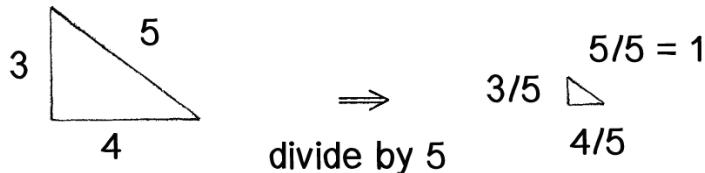


Figura 15.7: Normalização de vetor

A operação de normalização, portanto, cria um vetor *normalizado*, com a mesma direção do vetor original, porém com magnitude igual a 1, conforme esquematizado na figura 15.8:

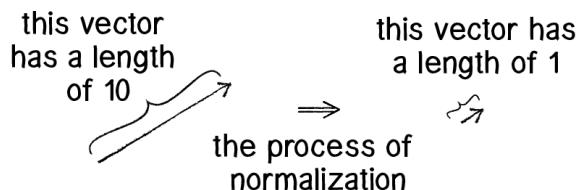


Figura 15.8: Criação de vetor de comprimento unitário

Para qualquer vetor \vec{u} , seu vetor normalizado (ou vetor unitário, denotado por \hat{u}) é calculado por:

$$\hat{u} = \frac{\vec{u}}{\| u \|}$$

Existem muitas operações matemáticas que são comumente usadas com vetores, mas para nosso exemplo, vamos parar por aqui. Eis nosso código:

```
#!/usr/bin/env python3

"""vector.py: A simple little Vector class. Enabling basic vector math.
"""

class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __truediv__(self, scalar):
        return Vector(self.x / scalar, self.y / scalar)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __ne__(self, other):
        return self.x != other.x or self.y != other.y

    def __abs__(self):
        return (self.x**2 + self.y**2)**0.5

    def __lt__(self, other):
        if self.x < other.x:
            return True
        if self.x == other.x:
            return self.y < other.y
        return False

    def __gt__(self, other):
        if self.x > other.x:
            return True
        if self.x == other.x:
            return self.y > other.y
        return False

    def __le__(self, other):
        if self < other:
            return True
        if self == other:
            return True
        return False

    def __ge__(self, other):
        if self > other:
            return True
        if self == other:
            return True
        return False

    def __neg__(self):
        return Vector(-self.x, -self.y)

    def __pos__(self):
        return Vector(self.x, self.y)

    def __radd__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __rmul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

```

--author__ = "Marcio Pereira"
--license__ = "GPL"
--version__ = "1.0"

class Vector:
    """Represents a Vector."""

    def __init__(self, x, y):
        """Constructor creates a Vector with x-axes and y-axes values."""
        self.x = x
        self.y = y

    def __str__(self):
        """ Return a string representation of self."""
        return "Vector(" + str(self.x) + ", " + str(self.y) + ")"

# for testing: create and use some Vector objects
if __name__ == '__main__':
    v1 = Vector(3, 4)
    v2 = Vector(2, -1)
    v3 = Vector(-15, 3)
    print(v1)
    print(v2.x)
    print(v3.y)

```

Saída:

```

Vector(3, 4)
2
3

```

A estrutura deste módulo (e a maioria dos outros) difere pouco do de um programa. A primeira linha é a linha **shebang** (a linha que começa com os caracteres `#!`) e, em seguida, é comum ter uma seqüência de citações entre aspas triplas que fornece uma visão geral do conteúdo do módulo, muitas vezes incluindo alguns exemplos de uso - esta é a **docstring** do módulo. Depois, temos alguns comentários (normalmente, os direitos autorais e as informações da licença), seguido do corpo do módulo e ao final, dentro de uma sentença `if`, uma parte reservada aos testes de suas funcionalidades.

Sempre que um módulo é importado, o Python cria uma variável para o módulo chamado `__name__` e armazena o nome do módulo nesta variável. O nome de um módulo é simplesmente o nome do seu arquivo `.py`, mas sem a extensão. Então, neste exemplo, quando o módulo for importado `__name__` terá o valor `"vector"`, e a condição `if` não será atendida, então as últimas linhas de teste do módulo não serão executadas. Isto significa que estas últimas linhas têm praticamente nenhum custo quando o módulo é importado. Sempre que um arquivo `.py` é executado, o Python cria uma variável para o programa chamado `__name__` e define-o para a string `"__main__"`. Então, se fosse executar o `vector.py` como se fosse um programa, o Python configurará `__name__` para `"__main__"` e a condição `if` irá avaliar para `True` e as últimas linhas de teste do módulo serão executadas.

15.3 A Classe Vector

A sintaxe de definição de classe tem duas partes: um cabeçalho de classe e um conjunto de definições de método que seguem o cabeçalho da classe. O cabeçalho da classe consiste no nome da classe e, no nome da classe “pai”, onde este objeto “herda” suas características. Quando não fornecido, esta classe é a classe `Object` de Python. Este mecanismo é conhecido como mecanismo de herança de classes, porém está fora do escopo deste curso. O nome da classe é um identificador em Python. Embora os nomes de tipos *builtins* não estejam capitalizados, os programadores Python tipicamente usam seus nomes de classes capitalizados para distingui-los de nomes de variáveis.

15.4 Definições de Métodos

Todas as definições de método são recuadas abaixo do cabeçalho da classe. Como os métodos são parecidos com funções, a sintaxe de suas definições é semelhante. Note, no entanto, que cada definição de método deve incluir um primeiro parâmetro chamado `self`, mesmo que esse método não receba nenhum argumento quando chamado. Quando um método é chamado com um objeto, o interpretador liga o parâmetro `self` a esse objeto para que o código do método possa se referir ao objeto pelo nome.

15.4.1 O Método `__init__`:

A maioria das classes inclui um método especial chamado `__init__`. Observe que `__init__` deve começar e terminar com dois sublinhados consecutivos. Esse método também é chamado de construtor da classe, porque é executado automaticamente quando um usuário instancia a classe. Assim, quando o segmento de código:

```
v1 = Vector(3, 4)
```

é executado, o Python executa automaticamente o construtor ou o método `__init__` da classe `Vector`. O objetivo do construtor é inicializar os atributos de um objeto individual. Além de si próprio, o construtor `Vector` espera dois argumentos que fornecem os valores iniciais para esses atributos. A partir deste momento, quando nos referimos ao construtor de uma classe, queremos dizer seu método `__init__`.

Os atributos de um objeto são representados como variáveis “de instância”. Cada objeto individual possui seu próprio conjunto de variáveis “de instância”. Essas variáveis servem de armazenamento para seu estado. O escopo de uma variável de instância (incluindo `self`) é a definição de classe inteira. Assim, todos os métodos da classe estão em posição de fazer referência às variáveis “da instância”. A vida útil de uma variável de instância é a vida útil do objeto em si.

Dentro da definição da classe, os nomes das variáveis “de instância” devem começar com `self`. Variáveis declaradas como da classe (i.e., sem `self`) são compartilhadas por todos os objetos daquele tipo.

15.4.2 O Método `__str__`:

As classes geralmente incluem um método `__str__`. Esse método cria e retorna uma representação em string do estado de um objeto. Quando a função `str` é chamada com um objeto, o método `__str__` desse objeto é automaticamente invocado para obter a seqüência de caracteres que descreve o objeto. Por exemplo, a chamada da função `str(v1)` é equivalente à chamada do método `v1.__str__()`, e é mais simples de escrever. Note também que a função `print` do Python chama o método `__str__` da classe do objeto a ser impresso. Exemplo:

```
print(v2)
```

Saída: Vector(2, -1)

15.4.3 Outros métodos especiais:

Podemos também definir as operações básicas de soma, subtração, multiplicação e divisão, usando métodos especiais (*dunder methods*) do Python [<https://docs.python.org/3/reference/datamodel.html>]:

```
class Vector:
    """Represents a Vector"""

    def __init__(self, x, y):
        """Constructor creates a Vector with x-axes and y-axes values."""
        self.x = x
        self.y = y

    def __str__(self):
        """Return a string representation of self."""
        return "Vector(" + str(self.x) + ", " + str(self.y) + ")"

    def __add__(self, other):
        """Return the sum of self and Vector object other."""
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        """Return the difference of self and Vector object other."""
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, alpha):
        """Return the product of self and numeric object alpha."""
        return Vector(self.x * alpha, self.y * alpha)

    def __rmul__(self, alpha):
        """Return the reverse product of self and numeric object alpha."""
        return Vector(self.x * alpha, self.y * alpha)

    def __truediv__(self, alpha):
        """Return the division of self and numeric object alpha."""
        return Vector(self.x / alpha, self.y / alpha)

# for testing: create and use some Vector objects
if __name__ == '__main__':
    v1 = Vector(3, 4)
    v2 = Vector(2, -1)
    v3 = Vector(-15, 3)
    print(v1)
    print(v2 + v1)
    print(v1 - v3)
    print(v3 * 2)
    print(4 * v1)
    print(v2 / 2)
```

Saída:

```
Vector(3, 4)
Vector(5, 3)
Vector(18, 1)
Vector(-30, 6)
Vector(12, 16)
Vector(1.0, -0.5)
```

Métodos normais de uma classe

Os métodos “normais” de uma classe são chamados pelos nomes. Vamos definir as operações de Normalização e Magnitude de vetores como métodos comuns, isto é, sem recorrer aos métodos especiais:

```
from math import sqrt

class Vector:
    def __init__(self, x, y):
        """Constructor creates a Vector with x-axes and y-axes values."""
        self.x = x
        self.y = y

    def __str__(self):
        """ Return a string representation of self."""
        return "Vector(" + str(self.x) + ", " + str(self.y) + ")"

    def __add__(self, other):
        """ Return the sum of self and Vector object other."""
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        """ Return the difference of self and Vector object other."""
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, alpha):
        """ Return the product of self and numeric object alpha."""
        return Vector(self.x * alpha, self.y * alpha)

    def __rmul__(self, alpha):
        """ Return the reverse product of self and numeric object alpha.
        """
        return Vector(self.x * alpha, self.y * alpha)

    def __truediv__(self, alpha):
        """ Return the division of self and numeric object alpha."""
        return Vector(self.x / alpha, self.y / alpha)

    def magnitude(self):
        """ Return the magnitude of self object."""
        return sqrt(self.x ** 2 + self.y ** 2)

    def norm(self):
```

```
""" Return the self object normalized."""
return self / self.magnitude()

# for testing: create and use some Vector objects
if __name__ == '__main__':
    v1 = Vector(3, 4)
    v2 = Vector(2, -1)
    v3 = Vector(-15, 3)
    print(v1.magnitude())
    print(v1.norm())
```

Saída:

```
5.0
Vector(0.6, 0.8)
```

15.4.4 Exercícios

1. **Classe Aluno:** Crie uma classe aluno com dois atributos, nome e lista de notas. Construa métodos para visualizar o aluno e a lista de notas do mesmo, bem como para inserir uma nota
2. **Classe Cadastro:** Para representar o cadastro de alunos, crie uma classe que contém um campo cadastro que é uma lista de Alunos. Crie métodos nesta classe para:
 - incluir um aluno no cadastro.
 - excluir um aluno do cadastro.
 - imprimir os alunos no cadastro.

Capítulo 16

Tópicos complementares de Python

16.1 Iteradores e Geradores

Quando introduzimos a sintaxe for-loop começando como:

```
for elemento in iterável:
```

notamos que existem muitos tipos de objetos no Python que se qualificam como sendo iteráveis. Os tipos básicos de contêiner, como list, tuple e set, são qualificados como iteráveis. Além disso, uma string pode produzir uma iteração de seus caracteres, um dicionário pode produzir uma iteração de suas chaves e um arquivo pode produzir uma iteração de suas linhas.

Tipos definidos pelo usuário também podem suportar iteração. No Python, o mecanismo de iteração é baseado nas seguintes convenções:

1. Um iterador é um objeto que gerencia uma iteração por meio de uma série de valores. Se uma variável `i`, identifica um objeto iterador, então cada chamada para a função interna, `next(i)`, produz um elemento subsequente da série subjacente, com uma exceção `StopIteration` levantada para indicar que não há elementos adicionais.

2. Um iterável é um objeto, `obj`, que produz um iterador através da sintaxe `iter(obj)`.

Por essas definições, uma instância de uma lista é iterável, mas não é um iterador. Com `data = [1, 2, 4, 8]`, não é legal chamar `next(data)`. No entanto, um objeto iterador pode ser produzido com a sintaxe, `i = iter(data)` e, em seguida, cada chamada subsequente para o `next(i)` retornará um elemento dessa lista. A sintaxe for-loop no Python simplesmente automatiza esse processo, criando um iterador para o iterável e, em seguida, chamando repetidamente o próximo elemento até capturar a exceção `StopIteration`.

Em geral, é possível criar vários iteradores com base no mesmo objeto iterável, com cada iterador mantendo seu próprio estado de progresso. No entanto, os iteradores normalmente mantêm seu estado com referência indireta à coleção original de elementos. Por exemplo, chamar `iter(data)` em uma instância da lista produz uma instância da classe iterator da lista. Esse iterador não armazena sua própria cópia da lista de elementos. Em vez disso, ele mantém um índice atual na lista original, representando o próximo elemento a ser relatado. Portanto, se o conteúdo da lista original for modificado depois que o iterador for construído, mas antes que a iteração seja concluída, o iterador relatará o conteúdo atualizado da lista.

O Python também suporta funções e classes que produzem uma série implícita de valores iteráveis, ou seja, sem construir uma estrutura de dados para armazenar todos os seus valores de uma só vez. Por exemplo, a chamada `range(1000000)` não retorna uma lista de números; ele retorna um objeto de intervalo que é iterável. Esse objeto gera um milhão de valores um de cada vez e somente conforme necessário. Esta técnica, chamada de avaliação preguiçosa tem uma grande vantagem. No

caso de range, esta técnica permite que um loop na forma `for j in range(1000000):`, execute sem necessitar de alocar memória para armazenar um milhão de valores. Além disso, se tal loop fosse interrompido de alguma forma, nenhum tempo teria sido gasto computando os valores não usados do intervalo.

Vemos a avaliação preguiçosa sendo usada em muitas das bibliotecas do Python. Por exemplo, a classe dicionário (`dic`) suporta os métodos `keys()`, `values()` e `items()`, que respectivamente produzem uma “visão” de todas as chaves, valores ou pares (chave, valor) dentro de um dicionário. Nenhum desses métodos produz uma lista explícita de resultados. Em vez disso, as visualizações que são produzidas são objetos iteráveis com base no conteúdo real do dicionário. Uma lista explícita de valores de tal iteração pode ser construída imediatamente chamando o construtor de classe de lista (`list`) com a iteração como um parâmetro. Por exemplo, a sintaxe `list(range(1000))` produz uma instância de lista com valores de 0 a 999, enquanto a sintaxe `list(d.values())` produz uma lista que possui elementos baseados nos valores atuais do dicionário `d`. Podemos, da mesma forma, construir uma tupla ou definir uma instância com base em um dado iterável.

16.1.1 Geradores

Na próxima seção, explicarei como definir uma classe cujas instâncias sirvam como iteradores. No entanto, a técnica mais conveniente para criar iteradores em Python é através do uso de geradores. Um gerador é implementado com uma sintaxe muito semelhante a uma função, mas em vez de retornar valores, uma instrução `yield` é executada para indicar cada elemento da série. Como exemplo, considere o objetivo de determinar todos os fatores de um inteiro positivo. Por exemplo, o número 100 possui os fatores 1, 2, 4, 5, 10, 20, 25, 50, 100. Uma função tradicional pode produzir e retornar uma lista contendo todos os fatores, implementados como:

```
def factors(n):
    results = []
    for k in range(1, n+1):
        if n % k == 0:
            results.append(k)
    return results

print(factors(100))
```

Saída: [1, 2, 4, 5, 10, 20, 25, 50, 100]

Em contraste, uma implementação de um gerador para computar esses fatores poderia ser implementada da seguinte forma:

```
def factors(n):
    for k in range(1, n+1):
        if n % k == 0:
            yield k
```

Se um programador escreve um loop como `for fator in factors(100):`, uma instância de nosso gerador é criada. Para cada iteração do loop, o Python executa nosso procedimento até que uma instrução `yield` indique o próximo valor. Nesse ponto, o procedimento é temporariamente interrompido, apenas para ser retomado quando outro valor é solicitado. Quando o fluxo de controle atinge naturalmente o final de nosso procedimento uma exceção **StopIteration** é automaticamente levantada.

Exemplo: Fibonacci

Neste exemplo vamos enfatizar os benefícios da avaliação lenta ao usar um gerador em vez de uma função tradicional. Os resultados só são computados se solicitados, e toda a série não precisa residir na memória ao mesmo tempo. Na verdade, um gerador pode efetivamente produzir uma série infinita de valores. Como sabemos, os números de Fibonacci formam uma sequência matemática clássica, começando com o valor 0, depois com o valor 1 e, em seguida, cada valor subsequente sendo a soma dos dois valores anteriores. Assim, a série de Fibonacci começa como: 0,1,1,2,3,5,8,13, O seguinte gerador produz esta série infinita:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

obj = fibonacci()
n_fib = next(obj)
while n_fib < 1000:
    print(n_fib, end = ", ")
    n_fib = next(obj)
print("...")
```

Saída: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

16.1.2 Iteradores

A iteração é um conceito importante no projeto de estruturas de dados. Por exemplo, um iterador para uma coleção fornece um comportamento-chave: ele suporta um método especial chamado `__next__` que retorna o próximo elemento da coleção, se houver, ou gera uma exceção `StopIteration` para indicar que não há elementos adicionais. Felizmente, é raro ter que implementar diretamente uma classe de iteradores. Nossa abordagem preferida é o uso da sintaxe de geradores, que produz automaticamente um iterador de valores produzidos.

O Python também ajuda fornecendo uma implementação do iterador automático para qualquer classe que defina `__len__` e `__getitem__`. Para fornecer um exemplo instrutivo de um iterador de baixo nível, o código abaixo demonstra apenas uma classe de iterador que funciona em qualquer coleção que suporte tanto o `__len__` quanto o `__getitem__`. Esta classe pode ser instanciada como `SequenceIterator(data)`. Ele opera mantendo uma referência interna à sequência de dados, bem como um índice atual na sequência. Cada vez que `__next__` é chamado, o índice é incrementado até atingir o final da sequência.

```
class SequenceIterator:
    """An iterator for any of Python's sequence types."""

    def __init__(self, sequence):
        """Create an iterator for the given sequence."""
        self.seq = sequence # keep a reference to the underlying data
        self.k = -1          # will increment to 0 on first call to next

    def __next__(self):
        """Return the next element, or else raise StopIteration error."""
        self.k += 1          # advance to next index
        if self.k < len(self.seq):
            return(self.seq[self.k]) # return the data element
```

```

        else:
            raise StopIteration()           # there are no more elements

    def __iter__(self):
        """By convention, an iterator must return itself as an iterator.
        """
        return self

my_list = [2, 4, 6, 8, 10]
my_iter = SequenceIterator(my_list)
for el in my_iter:
    print(el, end=" ")
print()
Saída: 2 4 6 8 10

```

Exemplo: A classe Range

Estamos prontos para desenvolver nossa própria classe que imita a classe `range` do Python. `range` é uma classe que pode representar efetivamente um intervalo desejado de elementos sem nunca armazená-los explicitamente na memória. A sintaxe `len(r)` informará o número de elementos que estão no intervalo determinado. Um intervalo também suporta o método `__getitem__`, de modo que a sintaxe `r[15]` relata o décimo sexto elemento no intervalo (como `r[0]` é o primeiro elemento). Como a classe suporta `__len__` e `__getitem__`, ela herda o suporte automático para iteração, e é por isso que é possível executar um loop `for` em um intervalo.

Neste ponto, estamos prontos para demonstrar nossa própria versão de tal classe. O fragmento de código abaixo fornece uma classe que chamamos de `my_range` (para diferenciá-lo claramente do intervalo interno). O maior desafio na implementação é calcular corretamente o número de elementos que pertencem ao intervalo, dados os parâmetros enviados pelo chamador ao construir um intervalo. Calculando esse valor no construtor e armazenando-o como `self.len`, torna-se trivial devolvê-lo a partir do método `__len__`. Para implementar adequadamente uma chamada para `__getitem__(k)`, simplesmente pegamos o valor inicial do intervalo mais `k` vezes o tamanho do passo (ou seja, para `k = 0`, retornamos o valor inicial). Existem algumas sutilezas que vale a pena examinar no código:

- Calculamos o número de elementos no intervalo como:

```
max(0, (stop - start + step - 1) // step)
```

(Vale a pena testar esta fórmula para tamanhos de degraus positivos e negativos.)

- O método `__getitem__` suporta corretamente índices negativos convertendo um índice `-k` em `len(self) - k` antes de calcular o resultado.

```

class my_range:
    """A class that mimics the built-in range class."""

    def __init__(self, start, stop=None, step=1):
        """Initialize a my_range instance. Semantics is similar to built-in
        range class."""
        if step == 0:
            raise ValueError('step cannot be 0')

        if stop is None:                 # special case of range(n)
            start, stop = 0, start      # should be treated as if range(0,n)

        # calculate the effective length once

```

```

        self._length = max(0, (stop - start + step - 1) // step)
        # need knowledge of start and step (but not stop) to support
    __getitem__
        self._start = start
        self._step = step

    def __len__(self):
        """Return number of entries in the range."""
        return self._length

    def __getitem__(self, k):
        """Return entry at index k (using standard interpretation if
negative)."""
        if k < 0:
            k += len(self) # attempt to convert negative index

        if not 0 <= k < self._length:
            raise IndexError('index out of range')

        return self._start + k * self._step

for i in my_range(0, 10, 2):
    print(i, end=" ")
print()
my_iter = iter(my_range(0, 10, 2))
el_0 = next(my_iter)
el_1 = next(my_iter)
el_2 = next(my_iter)
print(el_0, el_1, el_2)

```

Saída:

```
0 2 4 6 8
0 2 4
```

16.2 Polinômios como dicionários

Considere o polinômio: $p(x) = -1 + x^2 + 3x^7$

Os dados associados a este polinômio podem ser vistos como um conjunto de pares de coeficientes de potência, neste caso o coeficiente -1 pertence à potência 0, o coeficiente 1 pertence à potência 2 e o coeficiente 3 pertence à potência 7. Um dicionário pode ser usado para mapear uma potência para um coeficiente:

```
p1 = {0: -1, 2: 1, 7: 3}
```

É claro que uma lista também pode ser usada, mas, nesse caso, devemos preencher todos os coeficientes zero também, já que o índice deve corresponder à potência:

```
p2 = [-1, 0, 1, 0, 0, 0, 0, 3]
```

A vantagem de um dicionário é que precisamos armazenar apenas os coeficientes não nulos. Para o polinômio $1 + x^{100}$ o dicionário contém dois elementos enquanto a lista contém 101 elementos. Outra grande vantagem de usar um dicionário para representar um polinômio do que uma lista é que os potências negativas são facilmente permitidas, por exemplo:

```
p3 = {-3: 0.5, 4: 2}
```

pode representar $\frac{1}{2}x^{-3} + 2x^4$. Com uma representação em lista, as potências negativas exigiriam muito mais book-keeping.

16.2.1 Avaliação de Polinômios

A seguinte função pode ser usada para avaliar um polinômio representado como um dicionário:

```
def eval_poly_dict(poly, x):
    sum = 0.0
    for power in poly:
        sum += poly[power] * x ** power
    return sum

p1 = {0: -1, 2: 1, 7: 3}
print(eval_poly_dict(p1, 2))
```

Saída: 387.0

Uma implementação mais compacta pode usar a função sum do Python para somar os elementos de uma lista:

```
def eval_poly_dict2(poly, x):
    return sum([poly[power]* x ** power for power in poly])

p1 = {0: -1, 2: 1, 7: 3}
print(eval_poly_dict2(p1, 2))
```

Saída: 387

Podemos, de fato, descartar os colchetes e armazenar todos os números de potência $poly[power] * x ** power$

em uma lista, porque a soma pode adicionar diretamente elementos de um iterador, como `for power in poly:`

```
def eval_poly_dict2(poly, x):
    return sum(poly[power]* x ** power for power in poly)

p1 = {0: -1, 2: 1, 7: 3}
print(eval_poly_dict2(p1, 2))
```

Saída: 387

16.3 Plotando funções

A visualização de uma função $f(x)$ é feita desenhando a curva $y=f(x)$ em um sistema de coordenadas xy . Quando usamos um computador para fazer essa tarefa, dizemos que traçamos a curva. Tecnicamente, plotamos uma curva desenhando linhas retas entre n pontos na curva. Quanto mais pontos usamos, mais suave a curva aparece. Suponha que queremos plotar a função $f(x)$ para $a \leq x \leq b$. Primeiro nós selecionamos n coordenadas x no intervalo $[a, b]$, digamos que nomeamos estes x_0, x_1, \dots, x_{n-1} . Então nós calculamos $y_i = f(x_i)$ para $i = 0, 1, \dots, n-1$. Os pontos $(x_i, y_i), i = 0, 1, \dots, n-1$, agora formam a curva $y = f(x)$. Normalmente, escolhemos as coordenadas x_i para serem espaçadas igualmente, ou seja,

$$x_i = a + ih, \text{ onde } h = \frac{b-a}{n-1}$$

Se armazenarmos os valores x_i e y_i em duas matrizes x e y , podemos traçar a curva por um comando como `plot(x, y)`. Às vezes, os nomes da variável independente e da função diferem de x e f , mas o

procedimento de plotagem é o mesmo. Nossa exemplo de plotagem de curvas demonstra esse fato envolvendo uma função de t .

16.3.1 Matplotlib

O pacote padrão para plotagem de curvas em Python é o Matplotlib. Voce pode instalá-lo usando o comando pip:

```
pip install matplotlib
```

Vamos traçar a curva $y = t^2 \exp(-t^2)$ para valores de t entre 0 e 3 usando o matplotlib. Primeiro, geramos coordenadas igualmente espaçadas para t , digamos 51 valores (50 intervalos). Então calculamos os valores y correspondentes nesses pontos, antes de chamarmos o comando `plot(t, y)` para fazer o gráfico da curva. Aqui está o programa completo:

```
from numpy import *
from matplotlib.pyplot import *

def f(t):
    return t**2 * exp(-t**2)

t = linspace(0, 3, 51)
y = zeros(len(t))
for i in range(len(t)):
    y[i] = f(t[i])
plot(t, y)
show()
```

A figura 16.1 mostra o resultado da execução do programa:

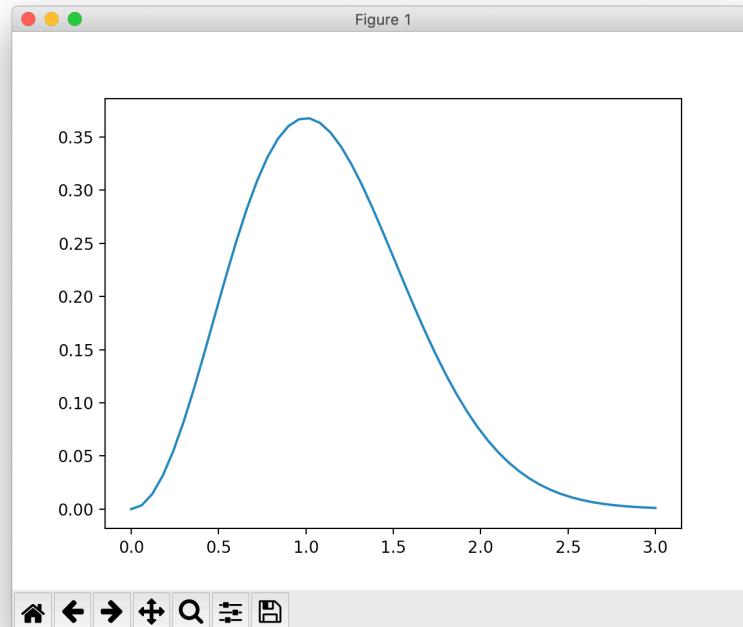


Figura 16.1: Exemplos de gráfico do matplotlib

Para incluir a plotagem em documentos eletrônicos, precisamos de uma cópia impressa da figura em PDF, PNG ou outro formato de imagem. A função `savefig` salva o gráfico em arquivos em vários formatos de imagem:

```
savefig('tmp1.pdf') # produz arquivo no formato PDF
savefig('tmp1.png') # produz arquivo no formato PNG
```

16.3.2 Decorando o gráfico

Os eixos x e y nos gráficos de curvas podem ter rótulos, no exemplo t e y, respectivamente. Além disso, a curva pode ser identificada com um rótulo ou legenda, como é frequentemente chamado. Um título acima do gráfico também é comum. Além disso, podemos querer controlar a extensão dos eixos (embora a maioria dos programas de plotagem ajuste automaticamente os eixos para o intervalo dos dados). Todas essas coisas são facilmente adicionadas após o comando de plotagem, conforme exemplo abaixo:

```
plot(t, y)
xlabel('t')
ylabel('y')
legend(['texp(-t)'])
axis([0, 3, -0.05, 0.6])           # [tmin, tmax, ymin, ymax]
title('Matplotlib Demo')
savefig('tmp2.pdf')
show()
```

O resultado é exibido na figura 16.2:

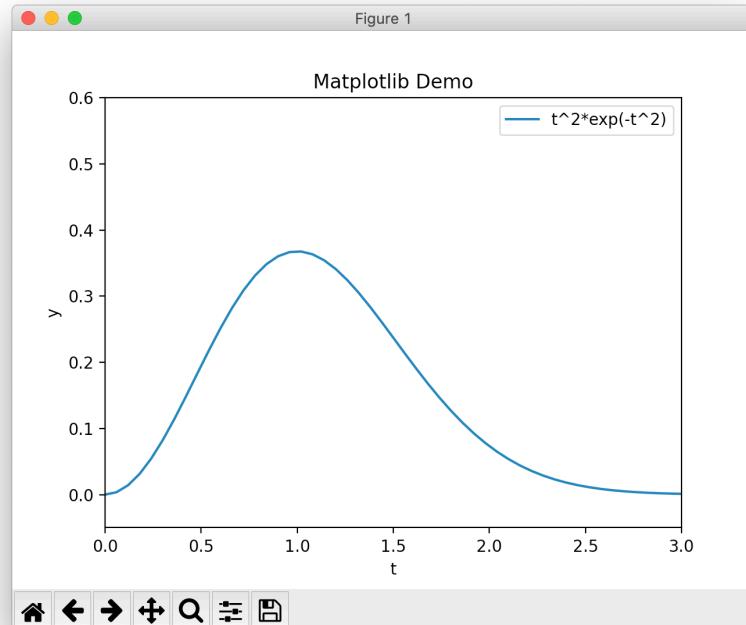


Figura 16.2: Exemplos de gráfico do matplotlib com legendas

16.3.3 Plotando múltiplas curvas

Uma tarefa comum de plotagem é comparar duas ou mais curvas, o que requer que várias curvas sejam desenhadas no mesmo gráfico. Suponha que queremos plotar as duas funções $f_1(t) = t^2 \exp(-t^2)$ e $f_2(t) = t^4 \exp(-t^2)$. Podemos, então, apenas emitir dois comandos de plotagem, um para cada função:

```
from numpy import *
from matplotlib.pyplot import *

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)
plot(t, y1, 'r-')
plot(t, y2, 'bo')
xlabel('t')
ylabel('y')
legend(['texp(-t)', 'texp(-t)'])
title('Plotando duas curvas no mesmo gráfico')
show()
```

Nesses comandos de plotagem, também especificamos o tipo de linha: r- significa linha vermelha (r) e tracejado (-), enquanto bo significa um círculo azul (b) e tracejado (o) em cada ponto de dados. As legendas de cada curva são especificadas em uma lista onde a seqüência de strings corresponde à seqüência de comandos de plotagem.

O resultado é exibido na figura 16.3:

Maiores informações sobre a biblioteca matplotlib podem ser encontradas no seguinte link:
<https://matplotlib.org/tutorials/index.html>.

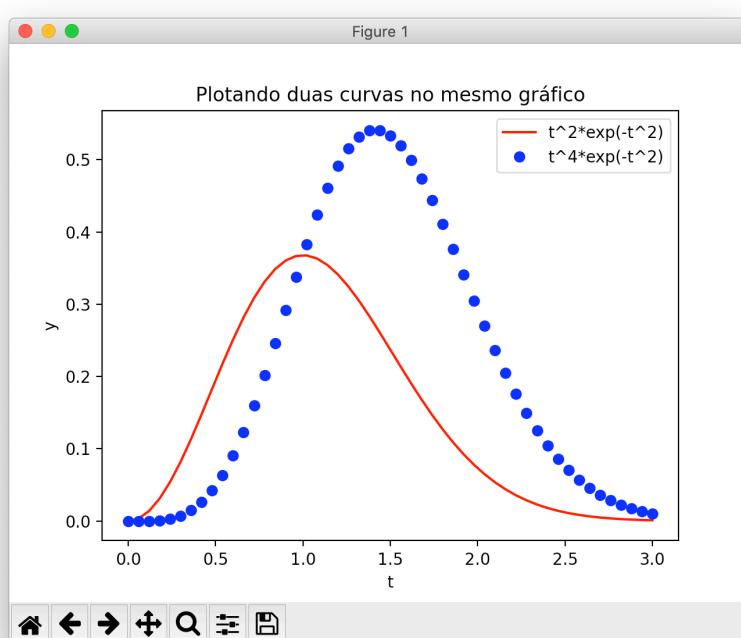


Figura 16.3: Exemplos de plotagem de múltiplas curvas

Capítulo 17

Expressões Regulares

17.1 Expressões Regulares

Expressões regulares são formas concisas de descrever um conjunto de strings que satisfazem um determinado padrão. Por exemplo, podemos criar uma expressão regular para descrever todas as strings na forma dd/dd/dddd, um padrão que representa datas, onde d é um dígito qualquer.

Dada uma expressão regular podemos resolver por exemplo este problema:

- Existe uma sequência de caracteres numa string de entrada que pode ser interpretada como um número de telefone? E qual é ele?

Note que números de telefones podem vir em vários “formatos”. Exemplos:

- 19-91234-5678
- (019) 91234 5678
- (19)912345678
- 91234-5678

Expressões regulares são uma mini-linguagem que permite especificar as regras de construção de um conjunto de strings. Essa mini-linguagem de especificação é muito parecida entre as diferentes linguagens de programação que contém o conceito de expressões regulares (também chamado de RE ou REGEX). Assim, aprender a escrever expressões regulares em Python será útil também para descrever REs em outras linguagens de programação.

Um exemplo de expressão regular é: '\d+'

Essa RE representa uma sequência de 1 ou mais dígitos. Vamos ver algumas regras de como escrever essas REs mais adiante. No momento vamos ver como usar uma RE.

É conveniente escrever a string da RE com um *r* na frente para especificar uma **raw string**, isto é, uma string onde sequências de caracteres como '\n' são tratados como 2 caracteres e não como uma quebra de linha.

Assim a RE é: *r'\d+'*

17.1.1 Método search

Expressões regulares e operações sobre eles são definidas na biblioteca `re` de Python. A principal função da biblioteca é a função `search`. Dada uma RE e uma string, a função busca a primeira ocorrência de uma substring especificada pela RE.

```
import re
a = re.search(r'\d+', 'Ouviram do Ipir723anga margens 45')
print(a)
```

Saída: <re.Match object; span=(15, 18), match='723'>

O resultado de `search` é do tipo `match` que permite extrair informação sobre qual é a substring que foi encontrada (o `match`) e onde na string ele foi encontrado (o `span`).

```
b = re.search(r'\d+', 'Ouviram do Ipiranga margens')
print(b, type(b))
```

Saída: None <class 'NoneType'>

Neste último exemplo, nenhum `match` é encontrado. Neste caso, `b` retorna o valor `None` (classe `NoneType`). Como vimos, o valor `None` se comporta como `False` em expressões condicionais. Assim, depois de usar o método `search` deve-se verificar se algo foi encontrado:

```
b = re.search(r'\d+', 'Ouviram do Ipiranga margens')
if b:
    pass
else:
    print("Não encontrado")
a = 0
```

Saída: Não encontrado

17.2 Objetos do tipo match

O método `span` de um objeto `match` retorna a posição inicial e `final+1` de onde a substring foi encontrada, enquanto o método `group` retorna a substring encontrada:

```
a = re.search(r'\d+', 'Ouviram do Ipir723anga margens 45')
print(a)
print(a.span())
print(a.group())
```

Saída:

```
<re.Match object; span=(15, 18), match='723'>
(15, 18)
'723'
```

Note que o método `search` acha apenas a primeira instância da RE na string (o número 45 também satisfaaz a RE).

17.3 Outras Funções da biblioteca RE

- A função `findall` retorna uma lista de todas as ocorrências da RE;
- A função `sub` substitui na string todas as substrings que conformam com a expressão regular fornecida por uma outra substring;
- A função `split` funciona como a função `split` para strings, mas permite usar uma expressão regular como separador.

Exemplos:

```
import re
a = re.findall(r'\d+', 'Ouviram do Ipir723anga margens 45')
b = re.findall(r'\d+', 'Ouviram do Ipiranga margens')
c = re.sub(r'\d+', '#', 'Ouviram do Ipir723anga margens 45')
d = re.split(r'\d+', 'ab 1 cd34efg h 56789 z')
print(a)
print(b)
print(c)
print(d)
```

Saída:

```
[‘723’, ‘45’]
[]
Ouviram do Ipir#anga margens #
[‘ab’, ‘cd’, ‘efg h’, ‘z’]
```

17.4 Compilando REs

Procurar uma RE numa string pode ser um processamento custoso e demorado. É possível “compilar” uma RE de forma que a procura seja executada de forma mais rápida:

```
rec = re.compile(r’\d+’)
xpto = rec.search(‘Ouviram do Ipir723anga margens 45’)
print(xpto.group())
```

Saída: 723

As funções vistas anteriormente funcionam também como métodos de REs compilados, e normalmente permitem mais alternativas. Por exemplo, o método `search` de uma RE compilada permite dizer a partir de que ponto da string deve-se começar a procurar a RE:

```
import re
rec = re.compile(r’\d+’)
index = 0
found = True
while found:
    xpto = rec.search(‘Ouviram do Ipir723anga margens 45’, index)
    if xpto:
        print(xpto.group(), ‘at position’, xpto.span()[0])
        index = xpto.span()[1]
    else:
        found = False
print(‘Finish’)
```

Saída:

```
723 at position 15
45 at position 31
Finish
```

17.5 Regras básicas para escrita de uma RE

As letras e números numa RE representam a si próprios. Assim, a RE `r’wb45p’` representa apenas a substring ‘wb45p’.

Os caracteres especiais (chamados de meta-caracteres) são:

```
. ^ $ * + ? { } [ ] \ | ( )
```

17.5.1 Backslash

As expressões regulares usam o caractere de barra invertida (\) para indicar formas especiais ou para permitir a utilização de caracteres especiais sem invocar o seu significado especial. Para combinar um literal barra invertida, é preciso escrever ‘\\\\\\’ como uma seqüência RE, porque a expressão regular deve ser ‘\\’, e cada barra invertida deve ser expressa como ‘\\’ dentro de um string literal do Python.

17.5.2 Repetições

- O meta-caractere . (ponto) representa qualquer caractere. Por exemplo, a RE `r'.ao'` representa todas as strings de 3 caracteres cujos 2 últimos são “ao”:

```
import re
r = re.compile(r'.ão')
a = r.findall("Adotar um cão é uma ótima opção, pois eles são grandes
               companheiros!")
print(a)
```

Saída: ['cão', 'ção', 'são']

- O meta-caractere + representa 1 (uma) ou mais repetições do caractere ou grupo de caracteres imediatamente anterior.

```
import re
r1 = re.compile(r'abc(de)+')
a = r1.search("abcde2de")
print(a)
```

Saída: <re.Match object; span=(0, 5), match='abcde'>

```
import re
r1 = re.compile(r'abc(de)+')
# não acha pois tem que ter pelo menos uma cadeia "de"
a = r1.search("Este é o nosso abcedário")
print(a)
```

Saída: None

- O meta-caractere * representa 0 ou mais repetições do caractere ou grupo de caracteres imediatamente anterior. Exemplos:

```
import re
r1 = re.compile(r'abc(de)*')
a = r1.search("abcide")
print(a)
```

Saída: <re.Match object; span=(0, 3), match='abc'>

```
import re
r1 = re.compile(r'abc(de)*')
# não acha pois tem que ter pelo menos uma cadeia "de"
a = r1.search("Este é o nosso abcedário")
print(a)
```

Saída: <re.Match object; span=(15, 18), match='abc'>

17.6 Classes de caracteres

- A notação [] representa uma classe de caracteres, de forma que deve-se ter um match com algum dos caracteres da classe. Por exemplo, `r'm[ei]nta'` significa todas as strings de 5 caracteres que começam com *m* seguido de um *e*, ou *i* e terminam com *nta*. No exemplo abaixo, a palavra manta não faz parte da expressão regular:

```
r = re.compile(r'm[ei]nta')
a = r.findall("Não minta para mim. Vc sujou a manta de menta ou não?
               Vc é uma pimenta!")
print(a)
```

Saída: ['minta', 'menta', 'menta']

- O caractere – dentro do [] representa um intervalo. Assim, verb+[1-7]+ representa um dos dígitos de 1 a 7. De forma parecida, [a-z] e [0-9] representam as letras minúsculas e os dígitos, respectivamente.
- O caractere ^ no início de [] representa a negação da classe. Assim, r'ab[^h-z]', representa qualquer string começando com ab e terminando com qualquer caractere exceto os de h até z.

```
import re
r = re.compile(r'ab[^h-z]')
a = r.search("Oi abm")
print(a)
```

Saída: None

```
import re
r = re.compile(r'ab[^h-z]')
# não acha pois ab é seguido de h
a = r.search("Oi abh")
print(a)
```

Saída: None

Python fornece algumas classes pré-definidas que são bastante úteis:

- \d - Qualquer dígito decimal, isto é, [0-9]
- \D - É o complemento de , equivalente a [^0-9], i.e, faz o match com um caractere que não seja dígito.
- \s - Faz match com caracteres *whitespace*, i.e, equivalente a [\t\n\r\f\v].
- \S - O complemento de \s.
- \w - Faz o match com um caractere alfanumérico, i.e, equivalente a [a-zA-Z0-9].
- \W - O complemento de \w.
- teste

17.7 Opcional

O meta-caractere ? significa que o caractere que o precede pode ou não aparecer. Nos dois exemplos abaixo há um match de r'ab?c' tanto com abc quanto com ac.

```
import re
r = re.compile(r'ab?c')
a = r.search("ac")
print(a)
```

Saída: <re.Match object; span=(0, 2), match='ac'>

Pode-se criar um grupo incluindo-se uma string entre parênteses. Por exemplo, se quisermos detectar ocorrências de “Maio/18”, “Maio/2018” ou “Maio de 2018”, etc, podemos usar a RE r'Maio(/)? ?(de)? ?(20)?18'.

```
import re
r = re.compile(r'Maio(/)? ?(de)? ?(20)?18')
a = r.search("Maio/de18")
print(a)
b = r.search("Maio/2018")
print(b)
c = r.search("Maio/18")
print(c)
```

Saída:

```
<re.Match object; span=(0, 9), match='Maio/de18'>
<re.Match object; span=(0, 9), match='Maio/2018'>
<re.Match object; span=(0, 7), match='Maio/18'>
```

17.8 Outros Meta-Caracteres

- o caractere | representa um OU de diferentes REs.
- \b indica o separador de palavras (pontuação, branco, fim da string). Por exemplo: a RE `r'\bcasa\b'` seria a forma correta de procurar a palavra “casa” numa string.

```
import re
a = re.search(r'\bcasa\b', 'a casa')
print(a)
b = re.search(r'\bcasa\b', ' casa ')
print(b)
c = re.search(r'\bcasa\b', 'o casamento')
print(c)
```

Saída:

```
<re.Match object; span=(2, 6), match='casa'>
<re.Match object; span=(1, 5), match='casa'>
None
```

17.9 Exemplo - Buscando um email

Vamos construir uma RE para buscar um email:

- O userid é uma sequência de caracteres alfanuméricos \w+ separado por @.
- O host é uma sequência de caracteres alfanuméricos \w+.

```
re.search(r'\w+@\w+', 'bla bla bla abc@gmail.com bla')
```

Saída: <re.Match object; span=(12, 21), match='abc@gmail'>

O host não foi casado corretamente. O ponto não é um caractere alfanumérico. Vamos tentar `r'\w+@\w+\.\w+'` (note que . serve para considerar o caractere . e não o meta-caractere).

```
re.search(r'\w+@\w+\.\w+', 'bla bla bla abc@gmail.com bla')
```

Saída: <re.Match object; span=(12, 25), match='abc@gmail.com'>

```
re.search(r'\w+@\w+\.\w+', 'bla bla bla abc@gmail.com.br bla')
```

Saída: <re.Match object; span=(12, 25), match='abc@gmail.com'>

Note que neste último exemplo não foi casado corretamente o “.br”.

Podemos tentar `r'\w+@\w+\.\w+(\.\w+)?'`. Criamos um grupo no final `(.\w+)?` que é um ponto seguido de caracteres alfanuméricos, porém opcional.

```
re.search(r'\w+@\w+\.\w+(\.\w+)?', 'bla bla bla abc@gmail.com.br bla')
```

Saída: <re.Match object; span=(12, 28), match='abc@gmail.com.br'>

Agora a RE casa com os dois tipos de endereços de email!

17.10 Exercícios

1. Escreva uma RE para encontrar números de telefone do tipo:
 - (019)91234 5678

- 19 91234 5678
 - 19-91234-5678
 - (19)91234-5678
2. Faça uma função que recebe um string e retorna o string com os números de telefones transformados para um único formato: (19) 91234 5678

17.11 Referência

Consulte a página web (<https://docs.python.org/3/howto/regex.html>) para uma visão mais completa sobre expressões regulares em Python.

Parte II

Projeto de algoritmos

Capítulo 18

Ordenação

Vamos estudar alguns algoritmos para o seguinte problema:

- Dada uma coleção de elementos com uma relação de ordem entre si, devemos gerar uma saída com os elementos ordenados.

Antes porém, vamos estabelecer (pelo menos um motivo) por que é útil ter dados que estão ordenados. Em suma, os dados ordenados tornam o processamento de dados muito mais rápido.

Mas como medimos o que é “rápido” vs. o que é “lento”? A chave para analisar o tempo que um programa leva para executar é contar o número de operações que ele executará para um determinado tamanho de entrada. Os cientistas raramente se preocupam com algoritmos rápidos ou lentos para entradas muito pequenas: quase sempre são rápidos se a entrada for pequena. No entanto, ao processar grandes entradas (por exemplo, milhões de usuários que avaliam centenas ou mesmo milhares de dados), a velocidade torna-se crítica.

Vamos analisar quanto tempo leva para calcular o número de objetos iguais entre duas listas usando a função abaixo:

```
def numMatches(list1, list2):
    ''' return the number of elements that match between
        list1 and list2 '''
    count = 0
    for item1 in list1:
        if item2 in list2:
            if item1 == item2:
                count += 1
    return count
```

Para efeito de análise, vamos supor que as listas em questão tem comprimento 4, isto é, `len(list1) == len(list2) == 4`. Inicialmente, pegue o primeiro elemento da primeira lista e pergunte se este elemento está na segunda lista. O comando `in` é um pouco enganador porque esconde um número significativo de comparações. Como o Python verifica se um item está em uma lista? É necessário comparar esse item com cada item da lista! Então, neste caso, o primeiro item da primeira lista é comparado a cada item na segunda lista para determinar se está nessa lista ou não.

“Espere!”, Você diz. Se o item estiver realmente na lista, ele não precisa verificar os quatro itens, e pode parar de verificar quando ele encontra o item em questão. Isso é exatamente correto, mas na verdade não importa em nossa análise. Para uma análise como essa, os cientistas da computação são bastante pessimistas. Eles raramente se importam com o que acontece quando as coisas funcionam bem – o que eles importam é o que pode acontecer no pior dos casos. Nesse caso, o pior caso é quando o item não está na lista e o Python tem que compará-lo com cada item da lista para determinar que o elemento não está lá. Uma vez que o que nos interessa é o comportamento do

pior caso, realizaremos nossa análise como se estivéssemos lidando com o pior caso.

Então, de volta à análise: Para o primeiro item na lista, a Python fez quatro comparações aos itens na segunda lista – comandos que foram escondidos no comando `in`. Agora, nosso programa passa para o segundo item na primeira lista, onde novamente faz quatro comparações com a segunda lista. Da mesma forma, faz quatro comparações para cada um dos terceiro e quarto elementos na primeira lista, o que nos leva a um total de $4 + 4 + 4 + 4 = 4 * 4 = 16$ comparações. Isso provavelmente não soa como um número tão ruim. Afinal, seu computador pode fazer 16 comparações em menos de um segundo. Mas e se nossas listas fossem mais longas? E se a lista 1 tivesse 100 elementos e a lista 2, 1000 elementos (alto, mas não muito)? Isto posto, o sistema teria que fazer 1000 comparações (para os itens na segunda lista) para cada um dos 100 itens na primeira lista para um total de $100 * 1000 = 10^5$ comparações. Ainda não é enorme, mas espero que você possa ver onde isso está nos levando. Em geral, o algoritmo de *match* que escrevemos acima leva $N * M$ comparações, onde N é o tamanho da primeira lista e M é o tamanho da segunda lista. Por simplicidade, podemos assumir que as duas listas sempre terão o mesmo comprimento, N , caso em que ele faz as N^2 comparações.

A boa notícia é que podemos fazer significativamente melhor, mas temos que fazer uma suposição sobre as próprias listas. E se nossas listas fossem ordenadas alfabeticamente? Como isso pode tornar nosso algoritmo de *match* mais rápido? A resposta é que podemos manter as listas “sincronizadas”, por assim dizer, e “caminhar” através de ambas as listas ao mesmo tempo, em vez de selecionar um único elemento da primeira lista e compará-lo com todos os elementos na segunda. Por exemplo, se as listas são compostas de strings (por exemplo, nomes de bandas) ordenados lexicograficamente e, se você está olhando para o primeiro elemento da primeira lista e este string é igual a “Black Eyed Peas” e o primeiro elemento da segunda lista é “Counting Crows”, você sabe que o string “Black Eyed Peas” não aparecem na segunda lista, porque C é lexicograficamente maior que B. Então, você pode simplesmente descartar o “Black Eyed Peas” e passar para a próxima banda na primeira lista.

Veja como seria o algoritmo. Lembre-se que assumimos que as listas estão ordenadas.

- Inicialize um contador em 0
- Defina o item atual em cada lista para ser o primeiro item em cada lista
- Repita o procedimento a seguir até chegar ao final de uma das duas listas:
 - Compare os itens atuais em cada lista.
 - Se forem iguais, incremente o contador e avance o item atual de ambas as listas para os próximos itens nas listas.
 - Caso contrário, se o item atual na primeira lista for alfabeticamente menor do que o item atual na segunda lista, avance o item atual na primeira lista.
 - Caso contrário, avance o item atual na segunda lista.
- O contador detém o número de *matches*.

Antes de olhar para o código a seguir, pergunte a si mesmo: ”Que tipo de laço devo usar aqui? Um laço `for` ou `while`?

Aqui está o código Python correspondente:

```
def numMatches(list1, list2):
    '''return the number of elements that match between two sorted lists
    '''

    matches = 0
    i = 0
    j = 0
    while i < len(list1) and j < len(list2):
        if list1[i] == list2[j]:
```

```

        matches += 1
        i += 1
        j += 1
    elif list1[i] < list2[j]:
        i += 1
    else:
        j += 1
return matches

print(numMatches(['a', 'i', 'l', 'n', 'o', 's'], ['a', 'h', 'i', 'k', 'l'])
)

```

Saída: 3

Agora, a questão permanece: esta abordagem é realmente mais rápida do que a abordagem anterior para comparar os elementos em duas listas? A resposta é definitivamente sim. Voltemos a olhar o número de comparações que deveriam ser feitas para comparar duas listas com 4 elementos cada. Imagine que nenhum dos elementos coincida e que as listas são exatamente entrelaçadas alfabeticamente. Ou seja, primeiro o elemento da lista 1 é menor, então o elemento na lista 2 é menor, e assim por diante, como nas listas [“Amy Winehouse”, “Coldplay”, “Madonna”, “Red Hot Chili Peppers”] e [“Black Eyed Peas”, “Dave Matthews Band”, “Maroon 5”, “Stevie Nicks”]. Com estas duas listas, o código acima nunca irá disparar na primeira condição if – sempre aumentará i ou j, mas não ambos. Além disso, ele ficará sem elementos em uma lista, logo antes de ficar sem elementos na segunda. Essencialmente, o algoritmo irá analisar todos os elementos em ambas as listas.

No começo, pode parecer que não fizemos nenhuma melhoria. Afinal, ainda não estamos olhando todos os elementos de ambas as listas? Aha, mas agora há uma diferença importante! Enquanto antes de ver todos os elementos da segunda lista para cada elemento na primeira lista, aqui estamos olhando apenas todos os elementos da segunda lista uma vez. Em outras palavras, cada vez que “caminhamos” no laço, i ou j é incrementado mas nunca são diminuídos. Então, i ou j irão chegar ao final da lista depois que todos os elementos dessa lista terem sido analisados e os elementos menos 1 da outra lista serem analisados. Então, neste exemplo, isso significa que faremos exatamente 7 comparações. Em geral, se as listas são de comprimento N , o número de comparações que este algoritmo fará é $N + N - 1$ ou $2N - 1$. Então, mesmo para o caso em que uma lista tem 100 elementos e a segunda tem 1000, isso é apenas cerca de 1100 comparações, uma melhoria significativa em relação ao 10^5 da abordagem anterior! Os cientistas da computação chamam este segundo algoritmo de um algoritmo de tempo linear (em notação matemática, $O(n)$) porque a equação que descreve o número de comparações é linear. O primeiro algoritmo é chamado de algoritmo de tempo quadrático ($O(n^2)$) porque sua equação é quadrática.

18.1 Selection sort

Agora que temos pelo menos um motivo para classificar os dados (há muitos outros – você provavelmente pode pensar em vários deles), vejamos um primeiro algoritmo, chamado “Selection sort”, para realizar a classificação e vamos analisar sua complexidade em termos de tempo.

Nos nossos exemplos a seguir usaremos um lista de números como a coleção. É claro que quaisquer números possuem uma relação de ordem entre si. Apesar de usarmos inteiros, os algoritmos servem para ordenar qualquer coleção de elementos que possam ser comparados.

A idéia do algoritmo “selection sort” é a seguinte:

- Ache o menor elemento a partir da posição 0. Troque então este elemento com o elemento da

- posição 0
- Ache o menor elemento a partir da posição 1. Troque então este elemento com o elemento da posição 1
 - Ache o menor elemento a partir da posição 2. Troque então este elemento com o elemento da posição 2

E assim sucessivamente...

Este algoritmo é chamado de “selection sort” porque ele prossegue selecionando repetidamente o elemento mínimo restante e movendo-o para a próxima posição na lista.

```
def selectionSort(vet):
    for i in range(len(vet) - 1):
        min = i

        for j in range(i, len(vet)):
            if vet[min] > vet[j]:
                min = j
        if min != i:
            vet[i], vet[min] = vet[min], vet[i]

lista = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
selectionSort(lista)
print(lista)
```

Saída: [2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]

O algoritmo “selection sort” conforme descrito a seguir, tem duas propriedades que geralmente são desejáveis em algoritmos de classificação.

A primeira propriedade é que o algoritmo é realizado “in-place”. Isso significa que ele usa essencialmente nenhum armazenamento extra além do necessário para a entrada (a lista não ordenada nesse caso). Um pouco de armazenamento extra pode ser usado (por exemplo, uma variável temporária para manter o índice para o menor elemento). A propriedade importante é que o armazenamento extra necessário não deve aumentar à medida que o tamanho da entrada aumenta.

A segunda é que o algoritmo de classificação é estável. Isso significa que dois elementos que são iguais mantêm sua ordem relativa inicial. Isso se torna importante se houver informações adicionais anexadas aos valores que estão sendo classificados (por exemplo, se estamos ordenando uma lista de pessoas usando uma função de comparação que compara suas datas de nascimento). Os algoritmos de classificação estáveis garantem que a classificação de uma lista já ordenada deixa a ordem da lista inalterada, mesmo na presença de elementos que são tratados como iguais pela comparação. “Selection sort” é apenas um dos muitos algoritmos de classificação, alguns dos quais são mais rápidos do que outros. O algoritmo realiza uma troca por cada passagem na lista. Para fazer isso, ele procura o menor valor realizando uma passagem pela lista e, depois de completar a passagem, o coloca no local apropriado. Após a segunda passagem, o próximo menor elemento está no lugar. Este processo continua e requer $n - 1$ passagens para classificar n itens, uma vez que o item final deve estar no lugar após $n - 1$ passos. Sua complexidade portanto é expressa como $O(n^2)$.

18.1.1 Exercício

Faça uma execução passo-a-passo do Selection Sort com o vetor [30, 45, 21, 9, 15] preenchendo uma tabela com os resultados de cada iteração.

18.2 Bubble sort

O algoritmo “Bubble sort” faz várias passagens através de uma lista. Ele compara itens adjacentes e troca aqueles que estão fora de ordem. Cada passagem pela lista coloca o próximo maior valor no seu devido lugar. Em essência, cada item “borbulha” até o local onde pertence. Se houver n itens na lista, existem $n - 1$ pares de itens que precisam ser comparados na primeira passagem. É importante notar que, uma vez que o maior valor da lista faz parte de um par, ele será continuamente movido até a passagem estar completa.

No início da segunda passagem, o maior valor está agora no lugar. Existem $n - 1$ itens para ordenar, o que significa que haverá $n - 2$ pares. Uma vez que cada passagem coloca o próximo maior valor no lugar, o número total de passes necessários será $n - 1$. Depois de completar as $n - 1$ passagens, o menor item deve estar na posição correta, sem necessidade de processamento adicional.

```
def bubbleSort (vet):
    for i in range(len(vet) - 1):
        for j in range(0, len(vet) - i - 1):
            if vet[j] > vet[j + 1]:
                vet[j], vet[j + 1] = vet[j + 1], vet[j]
lista = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
bubbleSort(lista)
print(lista)
```

Saída: [2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]

Para analisar o “Bubble sort”, devemos notar que, independentemente de como os itens estão dispostos na lista inicial, as $n - 1$ passagens serão feitas para classificar uma lista de tamanho n . O número total de comparações é a soma dos primeiros $n - 1$ inteiros. Lembre-se de que a soma dos n primeiros inteiros é $\frac{1}{2}n^2 + \frac{1}{2}n$. A soma dos $n - 1$ primeiros inteiros é $\frac{1}{2}n^2 + \frac{1}{2}n - n$, que é $\frac{1}{2}n^2 - \frac{1}{2}n$. Isso ainda é uma comparação de $O(n^2)$. No melhor dos casos, se a lista já estiver ordenada, nenhuma troca será feita. No entanto, no pior caso, cada comparação causará uma troca. Em média, trocamos metade do tempo.

O algoritmo “Bubble sort” é muitas vezes considerado o método de classificação mais ineficiente, pois deve trocar itens antes que a localização final seja conhecida. Essas operações de câmbio “desperdiçadas” são muito onerosas. No entanto, como o “bubble sort” faz passar toda a parte não ordenada da lista, ele tem a capacidade de fazer algo que a maioria dos algoritmos de classificação não pode. Em particular, se durante uma passagem não houver trocas, então sabemos que a lista deve estar ordenada. O algoritmo pode ser modificado para parar cedo se achar que a lista foi ordenada. Isso significa que, para listas que exigem apenas algumas passagens, o “bubble sort” pode ter uma vantagem na medida em que reconhecerá a lista ordenada e irá parar. O código a seguir mostra esta modificação, que é muitas vezes referida como a “short bubble sort”.

18.2.1 Exercício

Faça uma execução passo-a-passo do Bubble Sort com o vetor [30, 45, 21, 9, 15] preenchendo uma tabela com os resultados de cada iteração.

18.3 Insertion sort

Nosso “Bubble sort” modificado funciona melhor do que o “Selection sort” para listas que já estão classificadas. Mas nosso “Bubble sort” modificado ainda pode funcionar mal se muitos itens estiverem fora de ordem na lista. Outro algoritmo, denominado “Insertion sort” tenta explorar a ordem parcial da lista de maneira diferente.

A estratégia é a seguinte:

- No i -ésimo passo pela lista, onde i varia de 1 a $n - 1$, o i -ésimo item deve ser inserido em seu lugar apropriado entre os primeiros i itens na lista.
- Após o i -ésimo passo, os primeiros “ i ” itens devem estar ordenados.

Este processo é análogo ao modo como muitas pessoas organizam cartas em suas mãos. Ou seja, se você mantiver as primeiras $i - 1$ cartas em ordem, você escolhe a i -ésima carta e compara com esses cartas até que seu local apropriado seja encontrado.

Tal como acontece com os outros algoritmos de ordenação, o “Insertion sort” consiste em dois laços. O laço externo atravessa as posições de 1 a $n - 1$. Para cada posição i neste laço, você salva o item e inicia o laço interno na posição $i - 1$. Para cada posição j neste laço, você move o item para $j + 1$ até encontrar o ponto de inserção para o item salvo (item i).

```
def insertionSort(vet):
    for i in range (1, len(vet)):
        aux = vet[i]
        j = i - 1
        while j >= 0 and vet[j] > aux: # Põe elementos v[j] > v[i]
            vet[j + 1] = vet[j]           # para frente e
            j -= 1                      # decrementa j
        vet[j + 1] = aux                # põe v[i] na pos. correta

lista = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
insertionSort(lista)
print(lista)
```

Saída: [2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]

Mais uma vez, a análise se concentra no laço interno. O laço externo executa $n - 1$ vezes. No pior caso, quando todos os dados estão fora de ordem, o laço interno é iniciado uma vez na primeira passagem pelo laço externo, duas vezes na segunda passagem, e assim por diante, para um total de $\frac{1}{2}n^2 - \frac{1}{2}n$ vezes. Assim, o comportamento do pior caso do tipo de inserção é $O(n^2)$.

Quanto mais itens na lista estiverem em ordem, melhor fica o “Insertion sort” até, no melhor caso de uma lista ordenada, ter o comportamento linear. No caso médio, no entanto, o “Insertion sort” ainda é quadrático.

18.4 Exercícios

1. **Ordenação por Inserção:** faça uma execução passo-a-passo do Insertion Sort com o vetor [30, 45, 21, 9, 15] preenchendo uma tabela com os resultados de cada iteração.
2. **Ordem Crescente:** Escreva uma função que verifique se um vetor $v[0..n-1]$ está em ordem crescente. Esse exercício põe em prática a estratégia de escrever os testes antes de inventar algoritmos para um dado problema.
3. **Insertion Sort Recursivo:** escreva uma versão recursiva do algoritmo de ordenação por inserção.
4. **Ordenação por inserção:** critique a seguinte implementação do algoritmo de ordenação por inserção

```
def insertion(v):
    n = len(v)
    for j in range(1, n):
        i = j - 1
        while i >= 0 and v[i] > v[i+1]:
            v[i], v[i+1] = v[i+1], v[i]
```

```
    return v

print(insertion([6, 2, 3, 8, 1, 7, 9, 4]))
```

5. **Mediana:** Seja $v[0..n-1]$ um vetor de números inteiros, todos diferentes entre si. A mediana do vetor é um elemento do vetor que seja maior que metade dos elementos do vetor e menor que (a outra) metade dos elementos. Mais precisamente, a mediana de $v[0..n-1]$ é um número m dotado de duas propriedades:

- (1) m é estritamente maior que exatamente $\lfloor (n - 1)/2 \rfloor$ elementos do vetor;
- (2) se n (número de elementos do vetor) é ímpar então m é igual a algum elemento do vetor, caso contrário, a mediana é definida como a média dos dois valores do meio.

Não confunda mediana com média. Por exemplo, a média de 1 2 99 é 51, enquanto a mediana é 2.

Escreva um algoritmo que encontre a mediana de um vetor $v[0..n-1]$ de números diferentes entre si.

Capítulo 19

Algoritmos de Busca

Um algoritmo de busca é um método para encontrar um item ou grupo de itens com propriedades específicas dentro de uma coleção de itens. Nós nos referimos à coleção de itens como um espaço de pesquisa. O espaço de busca pode ser algo concreto, como um conjunto de registros médicos, ou algo abstrato, como o conjunto de números inteiros. Um grande número de problemas que ocorrem na prática podem ser formulados como problemas de busca.

Vamos estudar dois algoritmos comumente usados em buscas. Em primeiro lugar, discutiremos o projeto do algoritmo, mostramos sua implementação como uma função de Python e, finalmente, fornecemos uma análise da complexidade computacional do mesmo. Para manter as coisas simples, as funções de busca irão processar uma lista de números inteiros. Listas de diferentes tamanhos podem ser passadas como parâmetros para as mesmas.

19.1 Busca pelo Mínimo

A função `min` de Python retorna o item mínimo ou menor em uma lista. Para estudar a complexidade desse algoritmo, vamos desenvolver uma versão alternativa que retorna a posição do item mínimo. O algoritmo assume que a lista não está vazia e que os itens estão em ordem arbitrária. O algoritmo começa tratando a primeira posição como a do item mínimo. Em seguida, procura à direita por um item menor e, se for encontrado, restabelece a posição do item mínimo para a posição atual. Quando o algoritmo atinge o final da lista, ele retorna a posição do item mínimo. Eis o código do algoritmo:

```
def ourMin(lst):
    """Returns the position of the minimum item."""
    minpos = 0
    current = 1
    while current < len(lst):
        if lst[current] < lst[minpos]:
            minpos = current
        current += 1
    return minpos

result = ourMin([3, 4, 8, 1, 9, 2, 7, 6])
print(result)
```

Saída: 3

Como você pode ver, há três instruções fora do laço que executam o mesmo número de vezes, independentemente do tamanho da lista. Assim, podemos descontá-los. Dentro do laço, encontramos

mais três instruções. Destes, a comparação na instrução **if** e o incremento de execução atual em cada passagem pelo laço. Não há laços aninhados nestas instruções. Este algoritmo deve visitar todos os itens da lista para garantir que localizou a posição do item mínimo. Assim, o algoritmo deve fazer $n - 1$ comparações para uma lista de tamanho n . Portanto, a complexidade do algoritmo é $O(n)$.

19.2 Busca Linear ou Sequencial

O operador **in** de Python procura por um item específico (vamos chamá-lo de item chave ou apenas chave) dentro de uma lista de itens organizados de forma arbitrária. Nessa lista, a única maneira de procurar um item de destino é começar com o item na primeira posição e compará-lo com a chave. Se os itens forem iguais, o método retornará **True**. Caso contrário, o método passa para a próxima posição e compara os itens novamente. Se o método chegar na última posição e ainda não encontrar a chave, ele retorna **False**. Esse tipo de pesquisa é chamado de busca seqüencial ou de busca linear. Uma função de pesquisa linear mais útil retornaria o índice de uma chave se for encontrado, ou **-1** caso contrário. Aqui está o código Python para uma função de busca linear:

```
def linearSearch(key, lst):
    """Returns the position of the key item if found,
       or -1 otherwise."""
    position = 0
    while position < len(lst):
        if key == lst[position]:
            return position
        position += 1
    return -1

result = linearSearch(3, [1, 5, 8, 2, 9, 3, 6, 4, 0])
print(result)
```

Saída: 5

19.2.1 Exercícios

1. Discuta a seguinte versão da função **buscaSequencial**

```
def buscaSequencial(key, vet):
    n = len(vet)
    for i in range(n):
        if key == vet[i]:
            break
    if i < n and vet[i] == key:
        return i
    else:
        return -1

print(buscaSequencial(3, [2, 4, 1, 9, 5, 3, 6]))
print(buscaSequencial(3, [2, 4, 1, 9, 5, 8, 8]))
```

2. Critique a seguinte versão da função **buscaSequencial**

```
def busca(key, vet):
    n = len(vet) - 1
    i = 0
```

```

while i < n and vet[i] != key:
    i += 1
if vet[i] == key:
    return i
else:
    return -1

print(busca(3, [2, 4, 1, 9, 5, 3, 6]))
print(busca(3, [2, 4, 1, 9, 5, 8, 3]))

```

3. Discuta a seguinte versão recursiva da função buscaSequencial

```

def buscaR(key, idx, vet):
    if idx < 0:
        return -1
    if key == vet[idx]:
        return idx
    return buscaR(key, idx-1, vet)

lista = [2, 4, 1, 9, 5, 3, 6]
print(buscaR(3, len(lista) - 1, lista))

```

19.3 Busca Binária em uma Lista

A busca linear é necessária para dados que não estão organizados em uma ordem específica. Ao pesquisar dados ordenados, podemos usar uma pesquisa binária.

Para entender como uma pesquisa binária funciona, pense no que acontece quando você procura uma palavra em um dicionário. Os dados em um dicionário estão organizados em ordem alfabética, então você não faz uma pesquisa sequencial. Em vez disso, você estima a posição alfabética da palavra que você deseja encontrar no dicionário e abre o livro o mais próximo possível dessa posição. Depois de abrir o livro, você determina se a palavra em questão reside, em termos alfabéticos, em uma página anterior ou na página posterior, e volta a pesquisar as páginas conforme necessário. Você repete esse processo até encontrar a palavra.

Agora vamos considerar um exemplo de uma pesquisa binária em Python. Para começar, vamos assumir que os itens da nossa lista de números estão classificados em ordem crescente. O algoritmo de busca vai diretamente para a posição intermediária na lista e compara o item nessa posição com a chave. Se houver uma correspondência, o algoritmo retorna a posição. Caso contrário, se a chave for menor do que o item atual, o algoritmo busca a parte da lista antes da posição intermediária. Se a chave for maior que o item atual, o algoritmo busca a parte da lista após a posição do meio. O processo de busca pára quando a chave é encontrada ou a posição de início atual é maior que a posição final atual.

Aqui está o código para a função de busca binária:

```

def binarySearch(key, lst):
    left, right = 0, len(lst) - 1
    while left <= right:
        midpoint = (left + right) // 2
        if key == lst[midpoint]:
            return midpoint
        elif key < lst[midpoint]:
            right = midpoint - 1
        else:

```

```

        left = midpoint + 1
    return -1

from random import randint

lst = [x * 2 for x in range(50)]
key = randint(0, 100)
print(key)
print(binarySearch(key, lst))

```

Há apenas um laço sem laços aninhados. Mais uma vez, o pior caso ocorre quando a chave não está na lista. Quantas vezes o laço é executado neste caso? Iste será igual ao número de vezes que o tamanho da lista pode ser dividido por 2 até o quociente ser 1. Para uma lista de tamanho n , você essencialmente realiza a redução $n/2 / 2 \dots / 2$ até que o resultado seja 1. Seja k o número de vezes que dividimos n por 2. Para resolver k , você tem $n/2^k = 1$, então $n = 2^k$ e $k = \log_2 n$. Assim, a complexidade do pior caso da pesquisa binária é $O(\log_2 n)$.

No exemplo da figura 19.1, são necessárias 3 comparações para encontrar o número 57, enquanto na busca sequencial são necessárias 5 comparações. Esse algoritmo funciona melhor à medida que o tamanho do problema aumenta. Nossa lista de 15 itens requer no máximo 4 comparações, enquanto que uma lista de 1.000.000 de itens requer, no máximo, apenas 20 comparações!

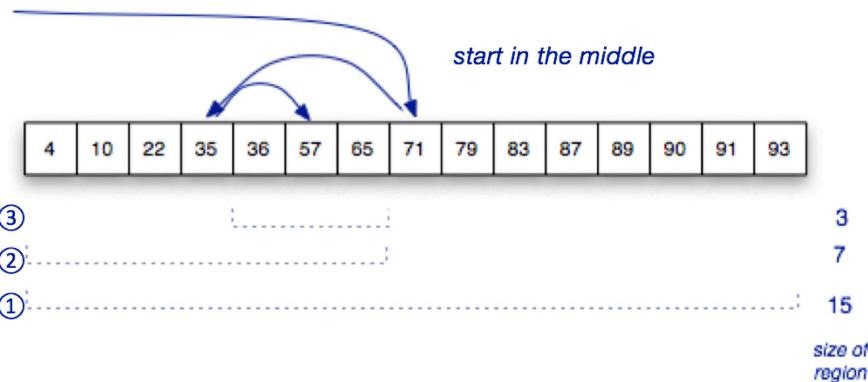


Figura 19.1: Exemplo de busca binária

19.4 Versão recursiva da busca binária

Para formular uma versão recursiva da busca binária é preciso generalizar ligeiramente o problema, trocando `vet[0..n-1]` por `vet[e..d]`. Para estabelecer uma ponte entre a formulação original e a generalizada, vamos usar uma função-embalagem (*wrapper-function*) `buscaBinaria`, que repassa o serviço para a função recursiva `bb`.

```

def BuscaBinaria(key, vet):
    # Esta função recebe uma chave key e um vetor
    # crescente vet[0..n-1] e devolve um índice m
    # em 0..n-1 tal que v[m] == key. Se tal m
    # não existe, devolve -1.
    return bb(key, 0, len(vet)-1, vet)

```

```

def bb(key, e, d, vet):
    # Esta função recebe uma chave key e um vetor
    # crescente vet[e..d] e devolve um índice m
    # em e..d tal que v[m] == key. Se tal m
    # não existe, devolve -1.
    if e > d:
        return -1
    else:
        m = (e + d) // 2
        if vet[m] == key:
            return m
        elif vet[m] < key:
            return bb(key, m+1, d, vet)
        else:
            return bb(key, e, m-1, vet)

print(BuscaBinaria(3, [3, 3]))

```

Saída: 0

19.4.1 Exercícios de Busca Binária

Implementações de busca binária exigem cuidado e atenção aos detalhes. É muito fácil escrever uma implementação que dá respostas erradas ou entra em loop. Os exercícios abaixo discutem versões alternativas da função `buscaBinaria`, diferentes da que examinamos acima. Todas procuram encontrar `key` em `vet[0..n-1]`. Todas produzem (ou deveriam produzir) um índice `m` em `0..n` tal que $vet[j-1] < key \leq vet[j]$

1. A seguinte implementação da busca binária funciona corretamente?

```

def buscaB(key, vet):
    e, d = -1, len(vet)
    while e < (d - 1):
        m = (e + d) // 2
        if key == vet[m]:
            return m
        if key > vet[m]:
            e = m
        else:
            d = m
    return -1

lista = [1, 2, 3, 4, 5, 6, 7, 8]
print(buscaB(9, lista))

```

2. O que acontece se trocarmos `while e < d-1:` por `while e < d:?`
3. A seguinte versão de `buscaBinaria` está correta? Justifique sua resposta.

```

def BinarySearch(key, vet):
    e, d = 0, len(vet)
    while e < d:                  # vet[e-1] < key <= vet[d]
        m = (e + d) // 2
        if vet[m] < key:

```

```

        e = m + 1
    else:
        d = m
    return d;                      # e == d

print(BinarySearch(3,[1,2,3,4]))

```

4. Critique a seguinte versão da função `bb`: a função recebe uma chave `key` e um vetor crescente `vet[e..d]` tal que $e \leq d$ e promete devolver um índice `m` em `e..d` tal que `vet[m] == key` (ou -1, se tal `m` não existe).

```

def bb (key, e, d, vet):
    if e <= d:
        if e == d and vet[d] == key:
            return d
        else:
            return -1
    m = (e + d) // 2
    if vet[m] == key:
        return m;
    elif vet[m] < key:
        return bb(key, m+1, d, vet)
    else:
        return bb(key, e, m-1, vet)

print(bb(3, 0, 4, [1, 2, 3, 4, 5]))

```

5. **Consumo de tempo:** Suponha que o vetor `vet` tem 511 elementos e que `key` não está no vetor. Quantas vezes, exatamente, a função `buscaBinaria` comparará `key` com um elemento do vetor? Suponha agora que o vetor `vet` tem 50000 elementos e que `key` não está no vetor. Quantas vezes, aproximadamente, a função `buscaBinaria` comparará `key` com um elemento do vetor?
6. **Exercício com chaves repetidas:** faça as funções de busca sequencial e busca binária assumindo que a lista possui chaves que podem aparecer repetidas. Neste caso, você deve retornar uma lista com todas as posições onde a chave foi encontrada. Se a chave só aparece uma vez, a lista conterá apenas um índice. E se a chave não aparece, as funções devem retornar a lista vazia.
7. **Exercício Two Sum:** Escreva um algoritmo eficiente para o seguinte problema: dados um vetor crescente `v[0..n-1]` de números inteiros e um número inteiro `t`, encontrar dois índices distintos `i` e `j` tais que `v[i] + v[j] = t`.

Capítulo 20

Recursividade

Uma função é chamada de recursiva se o corpo da função chama a própria função, direta ou indiretamente. Ou seja, o processo de execução do corpo de uma função recursiva pode, por sua vez, exigir a aplicação dessa função novamente. As funções recursivas não usam nenhuma sintaxe especial em Python, mas requerem algum esforço para entender e criar.

Um padrão comum pode ser encontrado no corpo de muitas funções recursivas. O corpo começa com um caso base, uma declaração condicional que define o comportamento da função para as entradas que são mais fáceis de processar. No caso do fatorial de um número inteiro positivo, o caso base é o fatorial de 1, que, por definição tem o valor 1. Note que algumas funções recursivas poderão ter múltiplos casos base.

Os casos base são seguidos por uma ou mais chamadas recursivas. As chamadas recursivas sempre têm um certo caráter: eles simplificam o problema original. As funções recursivas expressam a computação, simplificando os problemas de forma incremental.

20.1 Função Fatorial

As funções recursivas muitas vezes resolvem os problemas de maneira diferente das abordagens iterativas. Considere a função para calcular o fatorial de 4, isto é, $4! = 4 * 3 * 2 * 1 = 24$. Uma implementação natural usando uma declaração while acumula o total multiplicando juntos cada inteiro positivo até n .

```
def fact_iter (n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total

print(fact_iter (4))
```

Saída: 24

Por outro lado, uma implementação recursiva de fatorial pode expressar o fatorial de n (ou $n!$) em termos do fatorial de $n - 1$, isto é, $(n - 1)!$, e o caso base da recursão é a forma mais simples do problema: $1! = 1$.

@@@ todo tutor

Essas duas funções fatoriais diferem conceitualmente. A função iterativa constrói o resultado a partir do caso base 1, para o total final, multiplicando-se sucessivamente cada termo. A função recursiva, por outro lado, constrói o resultado diretamente do termo final, n , e o resultado do problema mais simples, fatorial $(n - 1)$.

À medida que a recursão “desenrola” através de sucessivas aplicações da função fatorial para instâncias de problemas mais simples e simples, o resultado é eventualmente construído a partir do caso base. A recursão termina passando o argumento 1 para a função fatorial; o resultado de cada chamada depende do próximo até o caso base ser atingido.

20.2 Torres de Hanoi

Uma aplicação muito elegante de resolução de problemas recursivos é a solução para um quebra-cabeças de matemática geralmente chamado de Torre de Hanói. Existem várias lendas a respeito da origem do jogo, a mais conhecida diz respeito a um templo Hindu, situado no centro do universo. Diz-se que Brahma supostamente havia criado uma torre com 64 discos de ouro e mais duas estacas equilibradas sobre uma plataforma. Brahma ordenou aos monges do templo que movessem todos os discos de uma estaca para outra seguindo as suas instruções. As regras eram simples: apenas um disco poderia ser movido por vez e nunca um disco maior deveria ficar por cima de um disco menor. Segundo a lenda, quando todos os discos fossem transferidos de uma estaca para a outra, o templo iria desmoronar.

A figura 20.1 mostra uma versão contendo apenas três discos. A tarefa é mover a torre da primeira estaca para a terceira estaca, usando a estaca central como uma espécie de lugar de repouso temporário durante o processo. É interessante observar que o número mínimo de “movimentos” para conseguir transferir todos os discos da primeira estaca à terceira é $2^n - 1$, sendo n o número de discos. Logo, para solucionar um Hanói de 6 discos, como no video foram necessários 63 movimentos. Para solucionar um Hanói de 7 discos, são necessários 127 movimentos; Para solucionar um Hanói de 15 discos, são necessários 32.767 movimentos; Para solucionar um Hanói de 64 discos, como diz a lenda, são necessários 18.446.744.073.709.551.615 movimentos.

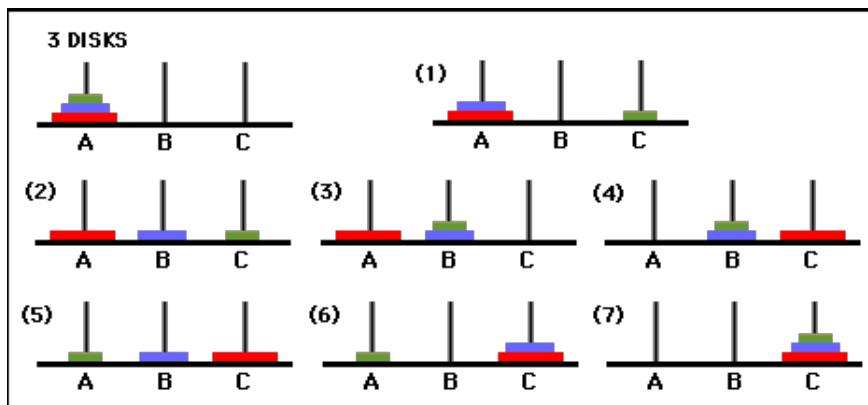


Figura 20.1: Torres de Hanoi com três discos

Queremos desenvolver um algoritmo para esse quebra-cabeças. Você pode pensar em nosso algoritmo como um conjunto de etapas que precisam se realizadas ou como um programa que gera um conjunto de instruções. Por exemplo, se rotularmos as três estacas A, B e C. As instruções podem começar assim:

- Mover o disco de A para C.
- Mover o disco de A para B.
- Mover o disco de C para B.
- ...

Este é um enigma difícil para a maioria das pessoas resolver. Claro, isso não é surpreendente, já que

a maioria das pessoas não é treinada no projeto de algoritmos. O processo de solução é bastante simples, se você sabe sobre a recursão. Comecemos por considerar alguns casos realmente fáceis. Suponha que tenhamos uma versão do quebra-cabeça com apenas um disco. Mover uma torre que consiste em um único disco é bastante simples; basta removê-lo de A e colocá-lo em C. Problema resolvido. OK, e se houver dois discos? Preciso obter o maior dos dois discos para postar C, mas o menor está sentado em cima dele. Eu preciso mover o disco menor para fora do caminho. Eu posso fazer isso, movendo-o para a estaca B. Agora, o disco grande em A, é claro, eu posso movê-lo para C e, em seguida, mover o disco menor da estaca B para a estaca C. Pronto.

Agora vamos pensar sobre uma torre de tamanho três. Para mover o disco maior para a estaca C, primeiro preciso mover os dois discos menores para fora do caminho. Os dois discos menores formam uma torre de tamanho 2. Usando o processo que descrevi acima, eu poderia mover essa torre (de dois discos) para a estaca B, e isso liberaria o maior disco para que eu possa movê-lo para a estaca C. Então eu só tenho que mover a torre de dois discos da estaca B para a estaca C. Resolver o caso do disco três resume-se em três etapas:

1. Mova a torre de dois discos de A para B.
2. Mova um disco de A para C.
3. Mova a torre de dois discos de B para C.

A primeira e terceira etapas envolvem mover uma torre de tamanho dois. Felizmente, já descobrimos como fazer isso. É como resolver o quebra-cabeça com dois discos, exceto que movemos a torre de A para B usando C como o lugar de repouso temporário e, em seguida, de B para C usando A como temporário. Acabamos de desenvolver o esboço de um algoritmo recursivo simples para o processo geral de mover uma torre de qualquer tamanho de uma estaca para outra.

Algoritmo: mover a torre de n-discos da fonte para o destino através do local de repouso:

- **passo 1:** mova a torre de (n-1) discos da fonte para o lugar de repouso;
- **passo 2:** mova a torre de 1 disco da origem para o destino;
- **passo 3:** mova a torre de (n-1) discos do lugar de repouso para o destino.

Qual é o argumento básico para este processo recursivo? Observe como um movimento de n discos resulta em dois movimentos recursivos de n - 1 discos. Uma vez que estamos reduzindo n de 1 a cada vez, o tamanho da torre eventualmente será 1. Uma torre de tamanho 1 pode ser movida diretamente, simplesmente movendo um único disco; não precisamos de chamadas recursivas para remover discos que estão acima. Corrigindo nosso algoritmo geral para incluir o caso base nos dá um algoritmo de movimento completo.

Exemplo: O exemplo abaixo mostra a codificação das Torres de Hanoi em Python. Observe que a função `moveTower` precisa de parâmetros para representar o tamanho da torre, *n*; a estaca fonte; a estaca de destino; e a estaca de repouso temporário. Pode-se usar um int para *n* e as strings “A”, “B” e “C” para representar as estacas. A função `hanoi` invoca a função `moveTower` para mover uma torre de tamanho *n* da estaca A para a estaca C. A título de exercício, implemente e teste o algoritmo abaixo para os valores 3 e 4.

```
# Solução:
def moveTower(n, source, dest, temp):
    if n == 1:
        print("Move disk from", source, "to", dest + ".")
    else:
        moveTower(n-1, source, temp, dest)
        moveTower(1, source, dest, temp)
        moveTower(n-1, temp, dest, source)

def hanoi(n):
    moveTower(n, "A", "C", "B")
```

Note como o código acima é “compacto”. Às vezes, usar recursão pode tornar quase trivial a escrita de código para resolver problemas difíceis. Então, nossa solução para a Torre de Hanói é um algoritmo “trivial” que requer apenas poucas linhas de código. O problema deste algoritmo está na eficiência do mesmo, ou seja, quantas etapas ele requer para resolver um problema de tamanho determinado. Neste caso, a dificuldade é determinada pelo número de discos na torre. A questão que queremos responder é quantas etapas são necessárias para mover uma torre de tamanho n ? Apenas olhando a estrutura do nosso algoritmo, você pode ver que mover uma torre de tamanho n exige que movamos uma torre do tamanho $n-1$ duas vezes, uma vez para deslocá-la para o maior disco e novamente para colocá-la novamente. Se adicionarmos outro disco à torre, essencialmente duplicamos o número de etapas necessárias para resolvê-lo. Em geral, resolver um quebra-cabeça de tamanho n exigirá $2^n - 1$ etapas.

Os cientistas da computação chamam isso de um algoritmo de tempo exponencial, uma vez que a medida do tamanho do problema, n , aparece no expoente desta fórmula. Os algoritmos exponenciais explodem muito rapidamente e só podem ser resolvidos praticamente em tamanhos relativamente pequenos, mesmo nos computadores mais rápidos. Apenas para ilustrar o ponto, se nossos monges realmente começaram com uma torre de apenas 64 discos e movido um disco a cada segundo, 24 horas por dia, todos os dias, sem cometer um erro, ainda assim eles levariam mais de 580 bilhões de anos para completar sua tarefa. Considerando que o universo tem cerca de 15 bilhões de anos agora, não estou muito preocupado em virarmos poeira ainda. Mesmo que o algoritmo para Torres de Hanoi seja fácil de expressar, pertence a uma classe conhecida como problemas intratáveis. Estes são problemas que requerem muito poder de computação (tempo ou memória) para serem resolvidos na prática, exceto para os casos mais simples.

20.3 Palíndromo

Um palíndromo é uma palavra, frase ou qualquer outra sequência de unidades (como uma cadeia de DNA) que tenha a propriedade de poder ser lida tanto da direita para a esquerda como da esquerda para a direita. Num palíndromo, normalmente são desconsiderados os sinais ortográficos (diacríticos ou de pontuação), assim como o espaços entre palavras.

Exemplos:

- radar
- Socoram-me, subi no ônibus em Marrocos.
- Was it a car or a cat I saw?
- Olé! Maracujá, caju, caramelô.

As frases formando um palíndromo também são chamadas de anacíclicas. O exemplo abaixo implementa uma função que verifica se um string é palíndromo.

```
# solução
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) < 2:
            return True
        else:
            if s[0] != s[-1]:
                return False
            else:
                return isPal(s[1:-1])
```

```

        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))

# testes:
print(isPalindrome("radar"))
print(isPalindrome("Socorram-me, subi no em Marrocos."))
print(isPalindrome("Qualquer texto"))

```

Saída:

```

True
True
False

```

20.4 Números de Fibonacci

Os números de Fibonacci são os números na seguinte sequência de números inteiros, chamados de seqüência de Fibonacci, e caracterizados pelo fato de que cada número após os dois primeiros é a soma dos dois precedentes:

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$

Por definição, os dois primeiros números na sequência Fibonacci são 0 e 1, e cada número subsequente é a soma dos dois anteriores.

A sequência F_n dos números de Fibonacci é definida pela relação de recorrência:

$F_n = F_{n-1} + F_{n-2}$, com $F_0 = 0$ e $F_1 = 1$.

Os números de Fibonacci aparecem muitas vezes em matemática, tanto que existe uma revista inteira dedicada ao estudo, o *Fibonacci Quarterly*. As aplicações dos números Fibonacci incluem algoritmos computacionais, como a técnica de busca Fibonacci e a estrutura de dados de heap Fibonacci, e gráficos chamados cubos Fibonacci utilizados para interligar sistemas paralelos e distribuídos. Eles também aparecem em configurações biológicas, como ramificação em árvores, filotaxis (o arranjo de folhas em um caule), os brotos de frutas de um abacaxi, a floração de uma alcachofra, uma samambaia desencadeada e o arranjo de brácteas de um cone de pinho (https://en.wikipedia.org/wiki/Fibonacci_number).

A figura 20.2 mostra uma parede de azulejos cujos comprimentos laterais são números sucessivos de Fibonacci:

20.4.1 Algoritmo Recursivo

O algoritmo a seguir usa a relação de recorrência dos números de Fibonacci para gerar recursivamente o n -ésimo número. Neste caso, estamos chamando a função `fib()` para retornar o quarto número da série de Fibonacci.

```

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

```

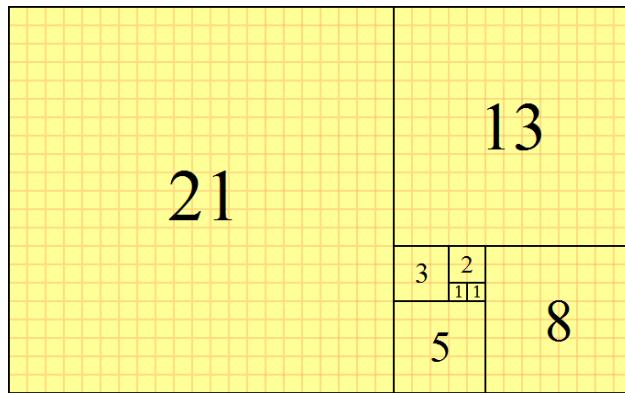


Figura 20.2: Números de fibonacci

```
print(fib(4))
```

Saída: 3

Vocês podem ter notado que quanto maior o argumento dado mais tempo a função leva para ser executada. Além disso, o tempo de execução aumenta rapidamente. Para entender por que, considere a Figura 20.3. Ela mostra o gráfico de chamada de fibonacci com n=4:

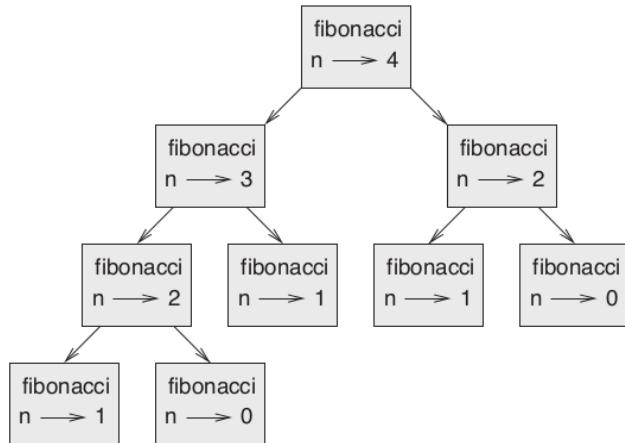


Figura 20.3: Árvore de recursão para fib(4)

O gráfico de chamada mostra um conjunto de frames com linhas que unem cada frame aos frames das funções chamadas. Na parte de cima do gráfico, fibonacci com n=4 chama fibonacci com n=3 e n=2. Por sua vez, fibonacci com n=3 chama fibonacci com n=2 e n=1. E assim por diante.

Observe quantas vezes fibonacci(0) e fibonacci(1) são chamadas. Essa é uma solução ineficiente para o problema, e piora conforme o argumento se torna maior.

20.4.2 Recursão com memorização

Uma solução é acompanhar os valores que já foram calculados, guardando-os em um dicionário. Um valor calculado anteriormente que é guardado para uso posterior é chamado de memo. Aqui está uma versão com memorização de fibonacci:

```

known = {0:0, 1:1}
def fibonacci(n):
    if n in known:
        return known[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res

print(fibonacci(300))

```

Saída: 222232244629420445529739893461909967206666939096499764990979600

A estrutura `known` é um dicionário que monitora os números de Fibonacci que já conhecemos. Começando com dois itens: 0 mapeia a 0 e 1 mapeia a 1.

Sempre que `fibonacci` é chamada, ela verifica `known`. Se o resultado já estiver lá, pode voltar imediatamente. Se não for o caso, é preciso calcular o novo valor, acrescentá-lo ao dicionário e devolvê-lo.

Ao executarmos essa versão de `fibonacci` podemos comparar com a original e descobrir que ela é muito mais rápida.

Exercício

Considerando os termos da seqüência de Fibonacci cujos valores não excedam quatro milhões, encontre a soma dos termos com valor par.

20.5 Recursão com Backtracking (Retrocesso)

Muitos problemas podem ser resolvidos enumerando-se de forma sistemática todas as possibilidades de arranjos que formam uma solução para um problema. Vimos anteriormente o seguinte exemplo:

- Determinar todas as soluções inteiras de um sistema linear como:

$$x_1 + x_2 + x_3 = C$$

com

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, C \geq 0$$

e todos inteiros.

Algoritmo:

- Para cada possível valor de x_1 entre 0 e C
- Para cada possível valor de x_2 entre 0 e $C - x_1$
- Faça:

$$x_3 = C - (x_1 + x_2)$$

- Imprima a solução $x_1 + x_2 + x_3 = C$

Abaixo temos o código de uma solução para o problema com $n = 3$ variáveis e constante C passada como parâmetro:

```

def solution(C):
    for x1 in range (0, C + 1):
        for x2 in range (0, C - x1 + 1):
            x3 = C - x1 - x2
            print("%d +%d +%d =%d" %(x1, x2, x3, C))

```

Nossa questão é como resolver este problema para o caso geral, onde n e C são parâmetros.

$$x_1 + x_2 + \cdots + x_{n-1} + x_n = C$$

A princípio deveríamos ter $n - 1$ laços encaixados. Mas não sabemos o valor de n . Só saberemos durante a execução do programa. A técnica de recursão pode nos ajudar a lidar com este problema:

- Construir uma função com um único laço e que recebe uma variável k como parâmetro.
- A variável k indica que estamos setando os possíveis valores de x_k .
- Para cada valor de x_k devemos setar o valor de x_{k+1} de forma recursiva!
- Se $k == n$ basta setar o valor da última variável.

Em Python teremos uma função com o seguinte protótipo:

```
def solution(n, C, k, R, x):
```

A variável R terá o valor da constante C menos os valores já setados para variáveis em chamadas recursivas anteriores, i.e.,

$$R = C - x_1 - \cdots - x_{k-1}$$

A lista x corresponde aos valores das variáveis. Lembre-se que em Python a lista começa na posição 0, por isso as variáveis serão

$$x[0], \dots, x[n-1]$$

Primeiramente temos o caso de parada (quando $k == n - 1$):

```
def solution(n, C, k, R, x):
    if (k == n - 1):
        #imprimindo a solução
        for i in range (0 , n-1):
            print(, %d + , %x[i], end=, )
        print(, %d = %d, %(R, C))
    # ...
```

A função completa é:

```
def solution(n, C, k, R, x):
    if k == n - 1:
        #imprimindo a solução
        for i in range (0, n-1):
            print(" %d + " %x[i], end="")
        print(" %d = %d" %(R, C))
    else:
        #seta cada possível valor de x[k] e faz recursão
        for x[k] in range(0, R + 1):
            solution(n, C, k + 1, R - x[k], x)
```

A chamada inicial da função deve ter $k = 0$.

```
import sys

# Just to simulate command line call
sys.argv = ['backtrack.py', 4, 7]
# Comment this line in a real script.

if len(sys.argv) != 3:
    print("Usage: python backtrack.py n [num vars] C [cte int]")
```

```

else:
    n = int(sys.argv[1])
    C = int(sys.argv[2])
    x = [0 for i in range(n)]
    solution(n, C, 0, C, x)

```

Um algoritmo de backtracking começa em um estado de inicialização predefinido e, em seguida, muda de estado em estado em busca de um estado final desejado. Em cada ponto do caminho, quando há uma escolha entre vários estados alternativos, o algoritmo escolhe um, possivelmente ao acaso, e continua. Se o algoritmo chegar a um estado que representa um resultado indesejável, ele retrocede (faz backtrack) até o último ponto em que houve uma alternativa inexplorada e tenta esta. Desta forma, o algoritmo busca de forma exaustiva em todos os estados ou atinge o estado final desejado.

A recursão é aplicada para retrocesso chamando uma função recursiva cada vez que um estado alternativo é considerado. A função recursiva testa o estado atual, e se é um estado final, o sucesso é relatado por todo o caminho de volta na cadeia de chamadas recorrentes. Caso contrário, existem duas possibilidades: (1) A função recursiva chama a si mesma num estado adjacente não testado; ou (2) Todos os estados adjacentes foram testados e a função recursiva relata falha na função que o chamou. Neste esquema, os registros de ativação na pilha de chamadas servem como a memória do sistema, de modo que, quando o controle retorna a uma função recursiva, pode retomar o lugar que deixou.

20.6 O Problema das oito Damas do Xadrez

No problema das oito Damas no tabuleiro de Xadrez, oito damas são colocadas em um tabuleiro de 8×8 casas de tal forma que as damas não se ameaçam. Uma dama pode atacar qualquer outra peça na mesma linha, coluna ou diagonal, então pode haver no máximo uma dama em cada linha, coluna e diagonal do tabuleiro. Não é óbvio que existe uma solução, mas a figura 20.4 mostra uma.

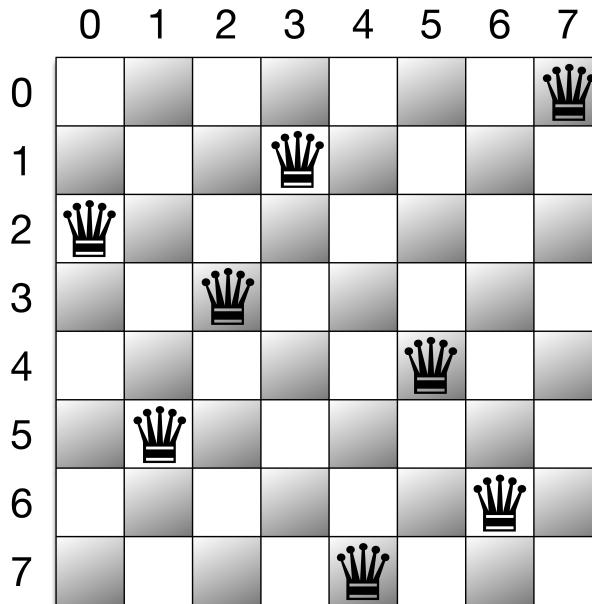


Figura 20.4: Uma solução do problema das 8 rainhas

Backtracking é a melhor abordagem para resolver este problema:

- colocamos a primeira dama no quadrado $(0, 0)$ da coluna 0. Colocamos a segunda dama na coluna 1 no primeiro quadrado não atacado, ou seja $(2, 1)$. Aplicando a mesma estratégia às colunas 2, 3 e 4, colocamos damas em quadrados $(4, 2)$, $(1, 3)$ e $(3, 4)$.
- Quando tentamos colocar uma dama na coluna 5, descobrimos que todos os quadrados estão sendo atacados, então voltamos para a coluna 4 e colocamos a dama no próximo quadrado, não sob ataque, o que é $(7, 4)$.
- No entanto, todos os quadrados da coluna 5 ainda estão sob ataque, e devemos voltar atrás para a coluna 4 novamente. Não há quadrados não testados na coluna 4, e voltamos para a coluna 3, onde tentamos o próximo quadrado não atacado em $(6, 3)$. Agora podemos avançar novamente para a coluna 5 e assim por diante. Desta forma, encontraremos uma solução se houver uma. Aqui está um algoritmo recursivo baseado na estratégia descrita. Inicialmente, o algoritmo é chamado com o valor de col igual a 0.

O código a seguir mostra a implementação desta abordagem em Python:

```
def canPlaceQueen(col, board)
    for each row in the board
        if board[row][col] is not under attack
            if col is the rightmost one
                place a queen at board[row][col]
                return True
            else:
                place a queen at board[row][col]
                if canPlaceQueen(col + 1, board)
                    return True
                else
                    remove the queen at board[row][col] (backtrack to
previous column)
    return False
```

20.6.1 Exercício

Escreva um programa em Python que tenta colocar n damas com segurança em um tabuleiro $n \times n$. O programa deve imprimir uma solução ou uma mensagem dizendo que não há solução.

Capítulo 21

Divisão e Conquista

Os algoritmos de classificação que estudamos têm tempos de execução $O(n^2)$. Existem várias variações nesses algoritmos de classificação, alguns dos quais são marginalmente mais rápidos, mas eles também são $O(n^2)$ nos casos médios e piores. Hoje iremos estudar alguns algoritmos melhores que são $O(n \log_2 n)$. O segredo para esses melhores algoritmos é o uso da estratégia de divisão e conquista que empregamos, por exemplo, na busca binária. Ou seja, cada algoritmo encontra uma maneira de quebrar a lista em sublistas menores. Essas sublistas são então ordenadas recursivamente. Idealmente, se o número dessas subdivisões é $\log_2(n)$ e a quantidade de trabalho necessário para reorganizar os dados em cada subdivisão é n , então a complexidade total de tal algoritmo de ordenação é $O(n \log_2 n)$. No quadro 21.1, você pode ver que a taxa de crescimento do trabalho de um algoritmo $O(n \log_2 n)$ é muito mais lenta que a de um algoritmo $O(n^2)$.

Quadro 21.1: Taxa de crescimento do trabalho de um algoritmo

n	$n \log_2 n$	n^2
512	4,608	262,144
1,024	10,240	1,048,576
2,048	22,458	4,194,304
8,192	106,496	67,108,864
16,384	229,376	268,435,456
32,768	491,520	1,073,741,824

Recapitulando, a estratégia de Divisão e Conquista pode ser feito em três amplos passos:

- **Dividir:** O passo de divisão quebra o problema original em subproblemas que são instâncias menores do problema original.
- **Conquistar:** O passo de conquistar resolve os subproblemas de forma recursiva. No passo recursivo, os subproblemas são resolvidos, até o caso base, que é o ponto em que o problema não pode ser mais dividido.
- **Combinar:** Neste passo, combina-se os subproblemas resolvidos para resolver o problema original.

21.1 Quick Sort

Vamos começar estudando um esboço da estratégia de Divisão e Conquista utilizada no algoritmo conhecido como quicksort:

- Comece selecionando o item no ponto médio da lista. Chamamos esse item de pivô. (Mais tarde, discutiremos maneiras alternativas de escolher o pivô).
- particione os itens na lista para que todos os itens inferiores ao pivô fiquem à esquerda do pivô, e o resto, à sua direita. A posição final do próprio pivô varia, dependendo dos itens reais envolvidos. Mas onde quer que o pivô termine, essa é a sua posição final na lista totalmente ordenada.
- Dividir e conquistar. Reaplicar o processo de forma recursiva às sublistas formadas pela divisão da lista no pivô. Uma sublista consiste em todos os itens à esquerda do pivô (agora os menores itens que o pivô) e a outra sublista tem todos os itens à direita (agora os maiores).
- O processo termina toda vez que encontramos uma sublista com menos de dois itens.

A parte mais complicada do algoritmo é a operação de partição dos itens em uma sublista. Existem duas maneiras principais de fazer isso. Informalmente, o que se segue é uma descrição do método mais fácil, pois se aplica a qualquer sublista:

1. Troque o pivô com o último item na sublista.
2. Estabeleça um limite entre os itens que são conhecidos como inferiores ao pivô e ao resto dos itens. Inicialmente, esse limite é posicionado imediatamente antes do primeiro item.
3. Iniciando com o primeiro item na sublista, explore a sublista. Toda vez que um item menor que o pivô é encontrado, troque-o com o primeiro item após o limite e avance o limite.
4. Termine, trocando o pivô com o primeiro item após o limite.

Depois que uma sublista foi particionada, reaplicamos o processo às suas sublistas à esquerda e à direita e assim por diante, até que as sublistas tenham um comprimento de um.

21.1.1 Análise de Complexidade

Apresentamos agora uma análise informal da complexidade do Quick Sort. Durante a primeira operação de partição, examinamos todos os itens desde o início da lista até o fim. Assim, a quantidade de trabalho durante esta operação é proporcional a n , o comprimento da lista. A quantidade de trabalho após esta partição é proporcional ao comprimento da sublista à esquerda mais o comprimento da sublista à direita, que em conjunto produzem $n - 1$. E quando essas sublistas são divididas, existem quatro partes cujo comprimento combinado é aproximadamente n , então o trabalho combinado é proporcional a n novamente. Como a lista é dividida em mais partes, o trabalho total permanece proporcional a n . Para completar a análise, precisamos determinar quantas vezes as listas são particionadas. Vamos fazer a suposição otimista de que, cada vez, a linha divisória entre as novas sublistas se torna tão próxima do centro da sublista atual quanto possível. Na prática, este não é geralmente o caso. Você já sabe, a partir da discussão do algoritmo de busca binária, que quando você divide uma lista pela metade repetidamente, você chega a um único elemento nas etapas em $\log_2 n$. Assim, o algoritmo é $O(n \log_2 n)$ no melhor caso.

Para o desempenho do pior caso, considere o caso de uma lista que já está ordenada. Se o elemento de pivô escolhido for o primeiro elemento, então, há $n - 1$ elementos à direita na primeira partição, $n - 2$ elementos à direita na segunda partição, e assim por diante. Embora nenhum elemento seja trocado, o número total de partições é $n - 1$ e o número total de comparações realizadas é $1/2n^2 - 1/2n$, o mesmo número no Selection Sort e no Bubble Sort. Assim, no pior caso, o algoritmo Quick Sort é $O(n^2)$.

Se Quick Sort for implementado como um algoritmo recursivo, a análise também deve considerar o uso de memória para a pilha de chamadas. Cada chamada recursiva requer uma quantidade constante de memória para um frame da pilha, e há duas chamadas recursivas após cada partição. Assim, o uso de memória é $O(\log_2 n)$ no melhor caso e $O(n)$ no pior dos casos. Embora o desempenho do pior caso do Quick Sort seja raro, os programadores certamente preferem evitá-

lo. Escolher o pivô na primeira ou última posição não é uma estratégia sábia. Outros métodos de escolha do pivô, como selecionar uma posição aleatória ou escolher a mediana dos primeiros, médios e últimos elementos, podem ajudar a aproximar o desempenho para $O(n \log_2 n)$.

21.1.2 Algoritmo Recursivo

O código a seguir mostra um exemplo de implementação recursiva do quicksort:

```
def quicksort(lst):
    quicksortHelper(lst, 0, len(lst) - 1)

def quicksortHelper(lst, left, right):
    if left < right:
        pivotLocation = partition(lst, left, right)
        quicksortHelper(lst, left, pivotLocation - 1)
        quicksortHelper(lst, pivotLocation + 1, right)

def partition(lst, left, right):
    # Find the pivot and exchange it with the last item
    middle = (left + right) // 2
    pivot = lst[middle]
    lst[middle] = lst[right]
    lst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in range(left, right):
        if lst[index] < pivot:
            lst[index], lst[boundary] = lst[boundary], lst[index]
            boundary += 1
    # Exchange the pivot item and the boundary item
    lst[boundary], lst[right] = lst[right], lst[boundary]
    return boundary
```

Aqui, temos um exemplo de execução:

```
lst = [12, 90, 47, -9, 78, 45, 78, 3323, 1, 2, 34, 20]
quicksort(lst)
print(lst)
```

Saída: [-9, 1, 2, 12, 20, 34, 45, 47, 78, 78, 90, 3323]

21.1.3 Algoritmo usando compreensão de listas

Podemos abreviar o nosso código do algoritmo quicksort utilizando comreensão de listas conforme ilustrado abaixo:

```
from random import randint

def qsort(v):
    if v == []:
        return []
    p = v[0]
    return qsort([x for x in v[1:] if x < p]) + [p] + qsort([x for x in v[1:] if x >= p])
```

```

lst = []
for count in range(20):
    lst.append(randint(1, 20))
print(lst)
res = qsort(lst)
print(res)

```

Saída:

[15, 14, 9, 13, 16, 1, 4, 20, 20, 16, 15, 19, 3, 16, 1, 6, 13, 3, 3, 18]

[1, 1, 3, 3, 3, 4, 6, 9, 13, 13, 14, 15, 15, 16, 16, 16, 18, 19, 20, 20]

Ou ainda:

```

def qsort2(L):
    return (qsort2([y for y in L[1:] if y < L[0]]) + L[:1] +
            qsort2([y for y in L[1:] if y >= L[0]])) if len(L) > 1 else L

```

21.1.4 Exercícios

1. Aplique o algoritmo de particionamento sobre o vetor [13, 19, 9, 5, 12, 21, 7, 4, 11, 2, 6, 6] com pivô igual a 6.
2. Qual o valor retornado pelo algoritmo de particionamento se todos os elementos do vetor tiverem valores iguais?
3. Faça uma execução passo-a-passo do Quick Sort com o vetor [4, 3, 6, 7, 9, 10, 5, 8].
4. Modifique o algoritmo Quick Sort para ordenar vetores em ordem decrescente.

21.2 Merge Sort

Outro algoritmo, denominado Merge Sort, emprega também uma estratégia recursiva, de divisão e conquista, para quebrar a barreira $O(n^2)$. Vejamos um resumo informal do algoritmo:

- Calcule a posição do meio de uma lista e ordene recursivamente suas sublistas da esquerda e direita (dividir e conquistar).
- Junte as duas sublistas ordenadas novamente em uma única lista ordenada (mesclagem ou merge).
- Pare o processo quando as sublistas não puderem mais ser subdivididas.

O processo de mesclagem, também conhecido por ordenação por intercalação usa uma lista auxiliar do mesmo tamanho da lista. A ideia é executar um laço onde em cada iteração testamos quem é o menor elemento dentre as duas sublistas e copiamos este elemento para a nova lista. Durante a execução deste laço podemos chegar em uma situação onde todos os elementos de uma das sublistas foram todos avaliados. Neste caso terminamos o laço e copiamos os elementos restantes da outra sublista. Vamos chamar essa lista auxiliar de `copyBuffer`. Para evitar a sobrecarga de alocação e desalocação do `copyBuffer` sempre que a função `Merge` for chamada, o buffer é alocado uma única vez no `mergeSort` e passado como argumento para as funções auxiliares.

Na figura 21.1 temos um exemplo com a ordem de execução das chamadas recursivas.

A figura 21.2 mostra o retorno do exemplo da figura 21.1.

21.2.1 Algoritmo

Três funções do Python colaboram no design do algoritmo:

- `MergeSort`: A função chamada pelos usuários.

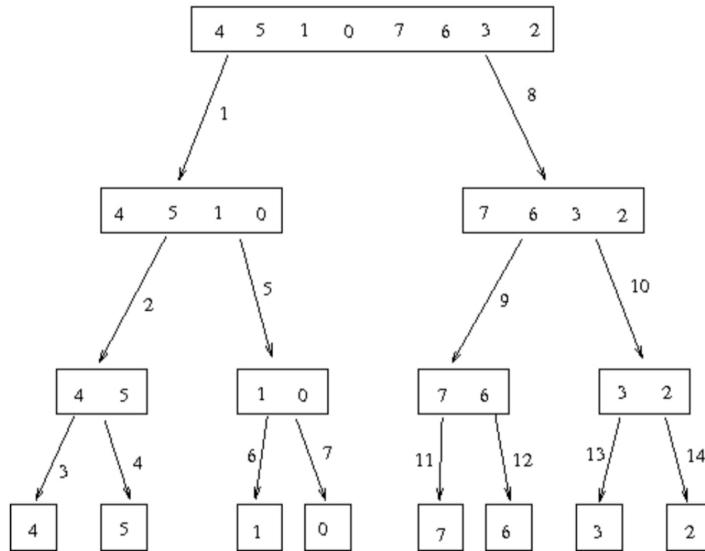


Figura 21.1: Ordem de execução das chamadas recursivas do Mergesort

- MergeSortHelper: A função auxiliar que esconde os parâmetros extras requeridos nas chamadas recursivas.
- Merge: A função que implementa o processo de mesclagem.

O código a seguir mostra um exemplo de implementação recursiva do quicksort:

```

def mergeSort(lst):
    # lst          list being sorted
    # copyBuffer  temporary space needed during merge
    copyBuffer = [0 for i in range(len(lst))]
    mergeSortHelper(lst, copyBuffer, 0, len(lst) - 1)

def mergeSortHelper(lst, copyBuffer, low, high):
    # lst          list being sorted
    # copyBuffer  temp space needed during merge
    # low, high   bounds of sublist
    # middle     midpoint of sublist
    if low < high:
        middle = (low + high) // 2
        mergeSortHelper(lst, copyBuffer, low, middle)
        mergeSortHelper(lst, copyBuffer, middle + 1, high)
        merge(lst, copyBuffer, low, middle, high)

def merge(lst, copyBuffer, low, middle, high):
    # lst          list that is being sorted
    # copyBuffer  temp space needed during the merge process
    # low         beginning of first sorted sublist
    # middle      end of first sorted sublist
    # middle + 1  beginning of second sorted sublist
    # high        end of second sorted sublist

    # Initialize i1 and i2 to the first items in each sublist
    i1 = low

```

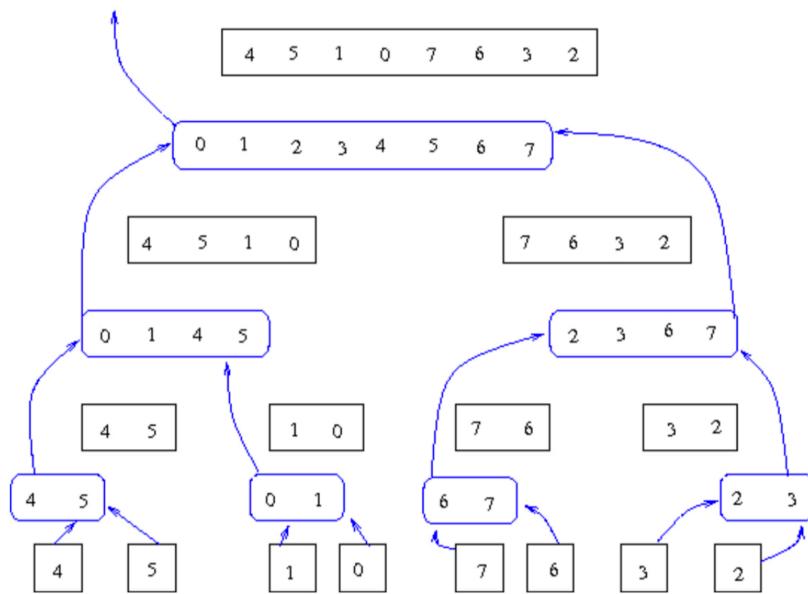


Figura 21.2: Retorno do exemplo anterior

```

i2 = middle + 1
# Interleave items from the sublists into the
# copyBuffer in such a way that order is maintained.
for i in range(low, high + 1):
    if i1 > middle:
        copyBuffer[i] = lst[i2] # First sublist exhausted
        i2 += 1
    elif i2 > high:
        copyBuffer[i] = lst[i1] # Second sublist exhausted
        i1 += 1
    elif lst[i1] < lst[i2]:
        copyBuffer[i] = lst[i1] # Item in first sublist is "<"
        i1 += 1
    else:
        copyBuffer[i] = lst[i2] # Item in second sublist is "<"
        i2 += 1
for i in range(low, high + 1): # Copy sorted items back to
    lst[i] = copyBuffer[i]      # proper position in lst

```

Aqui, temos um exemplo de execução:

```

lst = [12, 90, 47, -9, 78, 45, 78, 3323, 1, 2, 34, 20]
mergeSort(lst)
print(lst)

```

Saída: [-9, 1, 2, 12, 20, 34, 45, 47, 78, 78, 90, 3323]

A função de mesclagem combina duas sublistas ordenadas em uma sub-lista ordenada maior. A primeira sublista situa-se entre *low* e *middle* e a segunda entre o *middle + 1* e *high*. O processo consiste em três etapas:

1. Configure os ponteiros de índice para os primeiros itens em cada sublista. Estes estão em posições *low* e *middle + 1*.
2. Iniciando com o primeiro item em cada sublista, compare os itens repetidamente. Copie o

item menor de sua sublist para o buffer de cópia e avance para o próximo item na sublist. Repita até que todos os itens tenham sido copiados de ambas as sublistas. Se o final de uma sublist for atingido antes do outro, termine copiando os itens restantes da outra sublist.

3. Copie a parte do copyBuffer entre as partes *low* e *high* para as posições correspondentes em *lst*.

21.2.2 Análise de Complexidade

O tempo de execução da função de mesclagem é dominado pelas duas instruções, cada uma das quais $(high - low + 1)$ vezes. Conseqüentemente, o tempo de execução da função é $O (high - low)$, e todas as merges em um único nível levam o tempo $O (n)$. Como mergeSortHelper divide as sublistas o mais uniformemente possível em cada nível, o número de níveis é $O (\log_2 n)$ e o tempo máximo de execução dessa função é $O (n \log_2 n)$ em todos os casos. O merge possui dois requisitos de espaço que dependem do tamanho da lista. Primeiro, o espaço de $O (\log_2 n)$ é necessário na pilha de chamadas para suportar chamadas recursivas. Segundo, o espaço $O (n)$ é usado pelo buffer de cópia.

Exercícios

1. Mostre passo a passo a execução da função Merge considerando duas sub-listas:
[3, 5, 7, 10, 11, 12] e [4, 6, 8, 9, 11, 13, 14].
2. Faça uma execução Passo-a-Passo do MergeSort para a lista:
[30, 45, 21, 20, 6, 715, 100, 65, 33].
3. Reescreva o algoritmo MergeSort para que este passe a ordenar um vetor em ordem decrescente.
4. Considere o seguinte problema: Temos como entrada uma lista de inteiros *lst* (não necessariamente ordenada), e um inteiro *x*. Desenvolva um algoritmo que determina se há dois números em *lst* cuja soma seja *x*. Tente fazer o algoritmo o mais eficiente possível. Utilize um dos algoritmos de ordenação na sua solução.

21.2.3 Solução alternativa para o Merge sort

Esta solução alternativa não cria explicitamente uma lista auxiliar para o processo de mesclagem, porém ele retorna a lista ordenada. De qualquer forma você acaba tendo uma cópia da lista.

```
def merge(left, right):
    """Merge sort merging function."""

    left_index, right_index = 0, 0
    result = []
    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1

    result += left[left_index:]
    result += right[right_index:]
    return result
```

```
def merge_sort(array):
    if len(array) <= 1:  # base case
        return array

    # divide array in half and merge sort recursively
    half = len(array) // 2
    left = merge_sort(array[:half])
    right = merge_sort(array[half:])

    return merge(left, right)
```

21.2.4 Exercícios

1. Dada a seguinte lista de números:

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

Qual resposta abaixo ilustra a lista a ser ordenada depois de 3 chamadas recursivas ao merge sort?

- (a) [16, 49, 39, 27, 43, 34, 46, 40]
- (b) [21, 1]
- (c) [21, 1, 26, 45]
- (d) [21]

2. Dada a seguinte lista de números:

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

Qual resposta abaixo ilustra as primeiras duas listas que serão mescladas no processo do merge sort?

- (a) [21, 1] e [26, 45]
- (b) [[1, 2, 9, 21, 26, 28, 29, 45] e [16, 27, 34, 39, 40, 43, 46, 49]
- (c) [21] e [1]
- (d) [9] e [16]

Parte III

Tópicos avançados

Capítulo 22

Análise de algoritmos

A avaliação da eficiência de um algoritmo é chamada de **análise de um algoritmo**, e consiste fundamentalmente em estimar a quantidade de recursos requeridos pelo algoritmo, em especial o **tempo** necessário para solucionar um problema. Em geral, ao se analizar vários algoritmos que solucionam um determinado problema, o mais eficiente é identificado. Além disso, esta análise pode indicar mais de um candidato e apontar vários algoritmos inferiores que devem ser descartados.

Para se fazer a análise de algoritmos é necessário definir um modelo da tecnologia de implementação que será usada, incluindo um modelo para os recursos dessa tecnologia e os custos envolvidos. No presente documento, vamos utilizar o modelo computacional RAM onde operações simples tais como $+$, $-$, etc. consomem exatamente uma unidade de tempo para serem executadas. Laços e chamadas a sub-rotinas não são considerados operações simples, mas são compostos por agrupamentos de operações simples. Cada operação de acesso a memória também consome exatamente uma unidade de tempo[25]. Além disso, as instruções são executadas seqüencialmente, sem operações concorrentes.

Além da análise de algoritmos propriamente dita, o tempo de execução de um algoritmo pode ser avaliado através do processo de *benchmarking*. Quando temos dois ou mais algoritmos que resolvem o mesmo problema, podemos gerar várias instâncias típicas do problema, aceitando-as como um conjunto representativo, e executar os algoritmos medindo na prática o tempo que cada um leva para resolver o conjunto de instâncias. No caso do problema de ordenação, por exemplo, pode-se testar vários algoritmos quanto sua eficiência (e corretude) incluindo seqüências de números já ordenados, seqüências vazias e assim por diante.

A avaliação de eficiência de algoritmos através de *benchmarking* aparentemente é uma boa opção, porém ela traz algumas desvantagens. Em primeiro lugar, todos os algoritmos a serem avaliados precisam ser implementados a priori, isso pode ser uma dificuldade, visto que pode-se gastar muito tempo para implementar todos os algoritmos para um determinado problema. Outra desvantagem é o fato de que, mesmo com algoritmos corretos, certos problemas são intratáveis computacionalmente, fazendo com que o tempo para o algoritmo produzir uma resposta seja impraticável. Por isso, veremos nas próximas seções conceitos e métodos para fazer a análise “teórica” de algoritmos.

O comportamento de um algoritmo para resolver determinado problema pode ser diferente para cada possível entrada. Tomando como exemplo o problema de ordenação, um algoritmo para este problema leva tempos diferentes para ordenar duas seqüências com a mesma quantidade de números, dependendo de quão próximas elas já estão de estarem ordenadas. Entretanto, para analisar um

algoritmo, em geral é mais importante verificar que o seu tempo de execução cresce à medida que cresce o tamanho da entrada. Portanto, a seguir definiremos com maiores detalhes o “tamanho da entrada” e o “tempo de execução”.

22.0.1 Tamanho de entrada

O tamanho de entrada de um problema depende da natureza do problema. Para o problema de ordenação, uma boa medida do tamanho da entrada é a quantidade de números da sequência a ser ordenada. Para um algoritmo que resolve sistemas de n equações lineares com n incógnitas, o tamanho da entrada pode ser expresso por n . Algumas vezes é conveniente descrever o tamanho da entrada com mais de um parâmetro. Por exemplo, em algoritmos pra resolver problemas utilizando grafos, normalmente o tamanho da entrada é dado pela quantidade de vértices somada à quantidade de arestas do grafo.

22.0.2 Tempo de execução

O tempo de execução de um algoritmo para uma determinada entrada é função do número de operações primitivas ou “passos” executados. É conveniente definirmos a noção de “passo de execução” de forma mais independente possível de máquina. Podemos inicialmente dizer que cada linha de código (ou pseudo-código) leva um tempo constante para ser executada. Uma linha pode levar um tempo diferente de outra linha, então assumimos que i -ésima linha de código leva um tempo c_i constante. Isto reflete de forma razoavelmente fiel o modelo computacional que adotamos. Ao analisar algoritmos, podemos iniciar com expressões confusas para o tempo de execução, mas nosso objetivo é chegar a uma expressão simples, que facilite a comparação entre diferentes algoritmos para resolver um problema.

Definimos a função $T(n)$ para representar o tempo requerido para um algoritmo solucionar um problema de tamanho n . Podemos chamar $T(n)$ de tempo de execução do algoritmo. Por exemplo, um determinado algoritmo pode ter tempo de execução expresso por $T(n) = cn$, onde c é uma constante. Mesmo sem sabermos o tamanho de determinada entrada ou o tempo exato de execução do algoritmo, a expressão nos indica que o tempo de execução cresce linearmente à medida que cresce o tamanho da entrada do problema. Pode-se dizer que este algoritmo executa em *tempo linear* ou que é um *algoritmo linear* para resolver o problema.

Como já foi mencionado anteriormente, diferentes instâncias de um problema do mesmo tamanho podem acarretar em tempos de execução diferentes. Freqüentemente a análise de algoritmos é feita para o pior caso de entrada, ou seja, o tempo máximo de execução de uma entrada, dentre todas as possíveis entradas de tamanho n . A seguir, veremos dois exemplos de análise de algoritmos de ordenação.

22.1 Análise do algoritmo Selection Sort

Como um primeiro exemplo de análise de algoritmo, vamos considerar o algoritmo de ordenação por seleção, ou *SelectionSort* [19]. A idéia geral do algoritmo é bastante simples: dado um vetor A (ou array) com n números, em cada iteração o menor elemento da parte embaralhada vetor ocupa a primeira posição da parte ordenada, no início do vetor. Assim, na primeira iteração, o menor elemento vai ocupar a primeira posição do vetor. Na próxima iteração o segundo menor valor ocupa a segunda posição do vetor, e assim por diante. O índice i comanda o laço principal, e o índice j

varre a parte não-ordenada do vetor procurando pelo menor elemento. Quando ele é encontrado, seu índice é armazenado na variável *menor*, e ao final da varredura o menor valor é trocado com o *i*-ésimo valor.

A figura 22.1 ilustra este procedimento tomando como exemplo de entrada um vetor $A = \langle 25, 37, 12, 48, 57, 92, 33, 86 \rangle$.

Os quadrados representam os elementos do vetor A , com os seus índices logo acima. Os quadrados cinza representam a parte do vetor já ordenada. Cada figura (de a até h) mostra a iteração do algoritmo para cada valor do índice i .

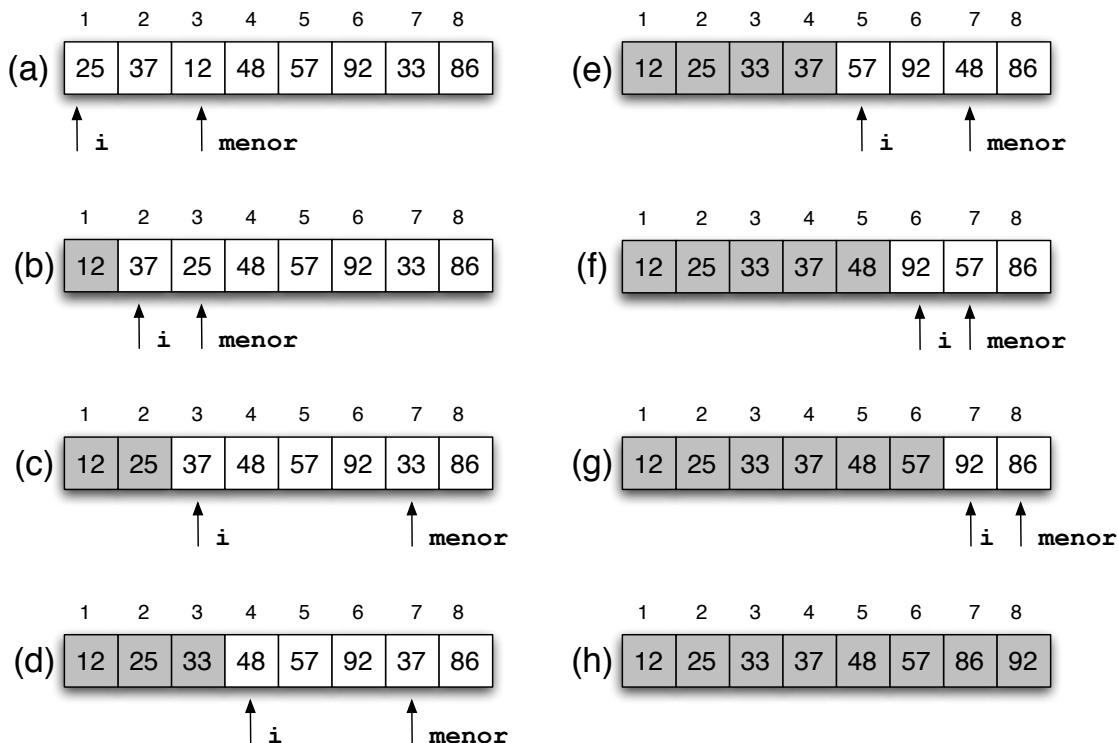


Figura 22.1: Exemplo de execução do algoritmo SelectionSort

Para analisar o *SelectionSort* precisamos olhar cuidadosamente o pseudo-código do algoritmo no quadro 22.1. Cada linha k de código consome um determinado tempo constante, chamado de “custo” e representado por c_k .

O cálculo do tempo de execução do algoritmo precisa verificar quantas vezes a k -ésima linha de custo c_k é executada para ordenar um vetor com n elementos. Para isso, definimos o termo t_i , que representa a quantidade de vezes que o laço da linha 4 é executado para cada $i = 1, 2, \dots, (n - 1)$. Assim, montamos a tabela 22.1 relacionando o custo de cada linha de pseudo-código com a quantidade de vezes que a linha é executada. Quando o laço “para–até–faça” encerra normalmente, o teste é executado uma vez a mais do que o corpo do laço.

O laço das linhas 4 a 6, é sempre executado j vezes, com i variando de 1 a $(n - 1)$ e j variando de $i + 1$ até n . Portanto, a linha 4 é executada $(n - i)$ vezes para cada valor de i , e as linhas 5 e 6 são executadas $(n - i) - 1$ vezes.

1 **Algoritmo:** SelectionSort(A)

```

2 para  $i \leftarrow 1$  até  $n - 1$  faça
3   menor  $\leftarrow i$ ;
4   para  $j \leftarrow i + 1$  até  $n$  faça
5     se  $A[j] < A[menor]$  então
6       menor  $\leftarrow j$ ;
7   temp  $\leftarrow A[menor]$ ;
8    $A[menor] \leftarrow A[i]$ ;
9    $A[i] \leftarrow temp$ ;

```

Algoritmo 22.1: SelectionSort

Quadro 22.1: Custo do SelectionSort

linha	custo	n. ^o de execuções
2	c_2	n
3	c_3	n-1
4	c_4	$\sum_{i=1}^{n-1} t_i$
5	c_5	$\sum_{i=1}^{n-1} (t_i - 1)$
6	c_6	$\sum_{i=1}^{n-1} (t_i - 1)$
7	c_7	n-1
8	c_8	n-1
9	c_9	n-1

O tempo de execução do algoritmo é igual à soma dos custos (tempos) de execução de cada comando. Um comando que leva c_k passos para executar e é executado n vezes vai contribuir com um tempo igual a $c_k n$ ao tempo total de execução. Para calcular o tempo de execução do *SelectionSort*, somamos os produtos dos custos com as quantidades de vezes que são executados, obtendo a seguinte expressão:

$$\begin{aligned}
T(n) = & c_2 n + c_3(n - 1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) \\
& + c_7(n - 1) + c_8(n - 1) + c_9(n - 1).
\end{aligned} \tag{22.1}$$

Como o laço da linha 4 é executado $(n - i)$ vezes para cada valor de i , com i variando de 1 até $(n - 1)$, temos:

$$\begin{aligned}
\sum_{i=1}^{n-1} t_i &= \sum_{i=1}^{n-1} (n - i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
\sum_{i=1}^{n-1} t_i &= n(n - 1) - \frac{n(n - 1)}{2}
\end{aligned}$$

$$\sum_{i=1}^{n-1} t_i = \frac{n(n-1)}{2}.$$

Para as linhas 5 e 6 temos:

$$\begin{aligned}\sum_{i=1}^{n-1} (t_i - 1) &= \sum_{i=1}^{n-1} t_i - \sum_{i=1}^{n-1} 1 \\ \sum_{i=1}^{n-1} (t_i - 1) &= \frac{n(n-1)}{2} - (n-1) \\ \sum_{i=1}^{n-1} (t_i - 1) &= \frac{n(n-1) - 2(n-1)}{2} \\ \sum_{i=1}^{n-1} (t_i - 1) &= \frac{(n-2)(n-1)}{2}.\end{aligned}$$

Substituindo estes termos na equação 22.1, encontramos:

$$\begin{aligned}T(n) &= c_2 n + c_3(n-1) + c_4 \left[\frac{n(n-1)}{2} \right] + c_5 \left[\frac{(n-2)(n-1)}{2} \right] + c_6 \left[\frac{(n-2)(n-1)}{2} \right] \\ &\quad + c_7(n-1) + c_8(n-1) + c_9(n-1) \therefore \\ T(n) &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right) n^2 + \left(c_2 + c_3 - \frac{c_4}{2} - \frac{3c_5}{2} - \frac{3c_6}{2} + c_7 + c_8 + c_9 \right) n \\ &\quad - (c_3 - c_5 - c_6 + c_7 + c_8 + c_9). \tag{22.2}\end{aligned}$$

Este tempo de execução pode ser expresso como $T(n) = an^2 + bn + c$, para constantes a , b e c que dependem dos custos (ou tempos) c_i de execução de cada comando do algoritmo. Portanto, verificamos que o tempo de execução do algoritmo *SelectionSort* é uma função quadrática do valor n .

Algoritmos *online*

Existem classes de algoritmos que não se enquadram nos termos descritos neste capítulo. Um exemplo disso são os algoritmos *online*. Na análise e projeto de algoritmos tradicionais, que podemos chamar de algoritmos *offline*, assume-se que o algoritmo possui total conhecimento dos dados de entrada. Entratanto em muitas aplicações práticas isso não acontece. Nos problemas *online* a entrada é parcialmente conhecida e dados de entrada relevantes podem surgir durante a execução do algoritmo.

Problemas *online* surgem em várias áreas de aplicação, tais como alocação de recursos em sistemas operacionais [18] [24], compressão de dados [27] [15], computação distribuída [5], mineração de dados [30], planejamento e controle de produção [6] [8], robótica [10] [26] [14] [11] e trabalho colaborativo [13] [12].

22.2 Comparação de tempos de execução

Na seção anterior, verificamos que o tempo de execução do algoritmo *SelectionSort* é uma função quadrática da quantidade de elementos a serem ordenados. Mas o que realmente significa um algoritmo ter tempo de execução *quadrático*?

Para ter noção do que significa isso, imaginemos que exista um determinado algoritmo A para resolver o problema de ordenação, e que o tempo de execução deste algoritmo é dado por $T_A(n) = 100n$ (portanto, uma função *linear* de n) e um outro algoritmo B, com tempo de execução dado por $T_B(n) = 2n^2$. Supondo que ambos os tempos de execução referem-se à quantidade de tempo em milissegundos que um determinado computador leva para executar os algoritmos para uma instância de problema de tamanho n , poderíamos montar o gráfico da figura 22.2 correlacionando o tempo de execução com o tamanho do problema para os dois algoritmos.

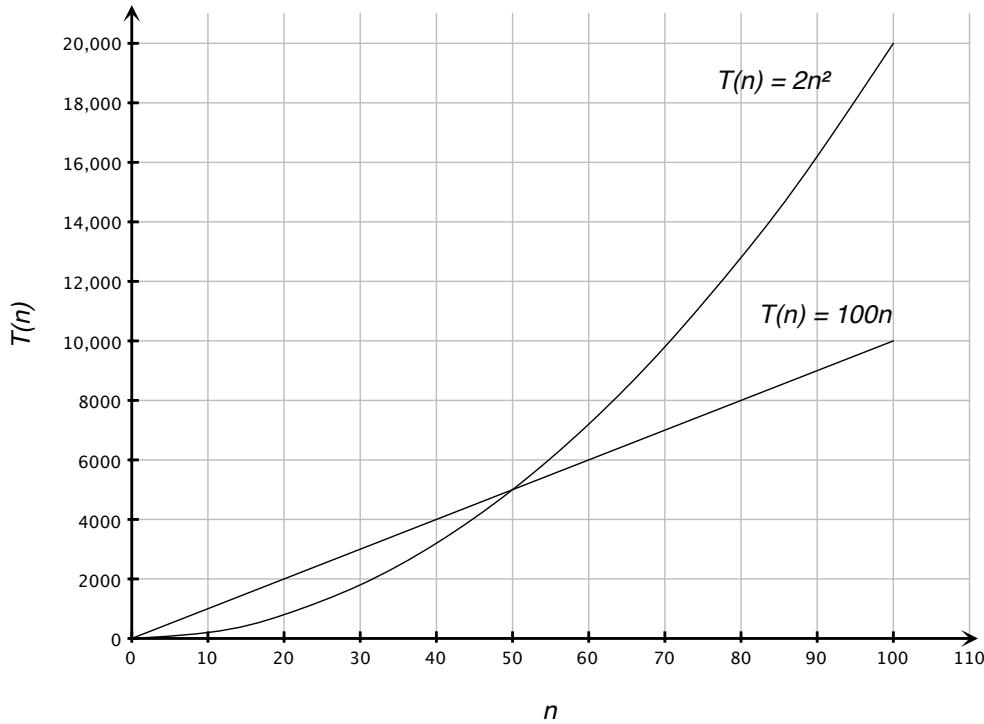


Figura 22.2: Tempo de execução de algoritmo quadrático x algoritmo linear

Neste gráfico podemos ver que, para instâncias do problema menores que 50, o algoritmo B produz uma resposta em menos tempo que o algoritmo A. Com uma entrada de tamanho 50, os dois algoritmos apresentam praticamente o mesmo desempenho, porém, quando a entrada começa a ficar maior, o algoritmo B tende a ser mais lento que o algoritmo A. Além disso, a partir de $n = 50$, quanto maior o valor de n , maior a diferença de desempenho dos dois algoritmos e portanto maior a vantagem de se usar o algoritmo A ao invés de B para resolver o mesmo problema. Para entradas de tamanho 100, o algoritmo A é duas vezes mais rápido que o algoritmo B. Para entradas de tamanho 1000, é 20 vezes mais rápido.

A função do tempo de execução de um algoritmo pode também nos indicar o tamanho máximo

de problema que podemos resolver com determinado algoritmo e determinado computador. À medida que a velocidade e o poder de processamento dos computadores cresce, obtemos uma melhora maior com algoritmos cuja função de tempo de execução cresce mais lentamente do que com aqueles cuja função cresce mais rapidamente. Por exemplo, com os algoritmos da figura 22.2, suponha que dispomos de 100 segundos de tempo de processamento. Se se o computador utilizado ficasse 10 vezes mais rápido, poderíamos resolver problemas que antes levariam 1000 segundos. Com o algoritmo A, essa melhoria permite que se resolva problemas 10 vezes maiores, mas com o algoritmo B, seria possível resolver problemas apenas 3 vezes maiores. A tabela 22.2 correlaciona os tamanhos máximos de problemas que poderiam ser resolvidos com os algoritmos A e B (denotados por $\max\langle n_A \rangle$ e $\max\langle n_B \rangle$, respectivamente), em função do tempo disponível em segundos.

Quadro 22.2: Tamanho máximo do problema em função do tempo

Tempo (seg.)	$\max\langle n_A \rangle$	$\max\langle n_B \rangle$
1	10	22
10	100	70
100	1000	223
1000	10000	707

É importante ressaltar que os valores na tabela 22.2 são meramente ilustrativos, pois na prática é difícil saber o tempo exato de execução de um algoritmo sem implementá-lo e executá-lo. Além disso, conforme já foi comentado, para alguns algoritmos o tempo de execução pode ser diferente para diferentes entradas do mesmo tamanho.

Na análise que fizemos do *SelectionSort*, assumimos algumas simplificações importantes. Em primeiro lugar, ignoramos o custo real de execução de cada comando, usando constantes arbitrárias c_i para representar estes custos. Depois disso, observamos que essas constantes representavam um detalhamento desnecessário, pois se não sabemos o valor real das constantes, poderíamos reduzir a função 22.2 a $T(n) = an^2 + bn + c$, para constantes a , b e c que dependem dos custos c_i .

Na verdade, ignoramos tanto os custos reais quanto os custos abstratos c_i . Porém, como foi visto acima, mesmo ignorando estes custos, podemos tirar conclusões importantes acerca do tempo de execução de algoritmos e comparar o desempenho de diferentes algoritmos para um mesmo problema através a análise da ordem ou taxa de crescimento da função do tempo de execução. Este assunto será abordado em maior profundidade no próximo capítulo.

22.3 Exercícios propostos

1. Dada a tabela abaixo, para cada função $f(n)$ e tempo t , determine o maior tamanho n do problema que pode ser resolvido no tempo t , assumindo que o problema leva $f(n)$ microsegundos para ser resolvido.

$f(n)$	$t = 1\text{seg}$	1 min	1 hora	1 dia	1 mês	1 ano	1 século
$\log n$							
\sqrt{n}							
n							
$n \log n$							
n^2							
n^3							
2^n							
$n!$							

2. Utilizando a figura 22.1 como exemplo, mostre a execução do algoritmo *SelectionSort* para a seqüência $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.
3. Seja uma matriz A de elementos inteiros de dimensões $n \times n$. Os elementos de cada linha e de cada coluna estão ordenados em ordem não-decrescente. Dado um valor x , escreva um algoritmo para determinar se existem $i, j \in 1, 2, \dots, n$ tais que $x = a_{i,j}$. Faça uma estimativa de seu tempo de execução $T(n)$ em função de n .

Capítulo 23

Ordens Assintóticas de Complexidade

No capítulo anterior, vimos conceitos relativos à eficiência de algoritmos e calculamos o tempo de execução do algoritmo *SelectionSort*. Dada uma entrada de valor n , verificamos que o algoritmo produz uma solução em um tempo que é uma função quadrática do tamanho do problema. Vimos também a comparação do tempo de execução entre um algoritmo com tempo quadrático e outro com tempo linear, e concluímos que, mesmo sem saber o tempo exato (real) de execução de determinado algoritmo, podemos obter informações importantes sobre o comportamento do algoritmo quando o tamanho do problema tende a crescer para um número suficientemente grande. Neste caso estamos avaliando a eficiência assintótica de algoritmos.

No presente capítulo, vamos estudar conceitos fundamentais necessários para a análise **assintótica** de algoritmos. Ou seja, conceitos que nos permitam avaliar a tentência de crescimento de tempo de execução de algoritmos, à medida que o tamanho da entrada do problema tende a um limite. Tal tendência reflete a quantidade de trabalho requerido pelo algoritmo para resolver um problema, essa quantidade é chamada de **complexidade do algoritmo** [29], a análise desta tendência também é chamada **análise de complexidade do algoritmo** [28].

As ordens assintóticas de complexidade seguem notações chamadas assintóticas, que são definidas em termos de funções cujos domínios estão contidos no conjunto dos números naturais $\mathbb{N} = \{0, 1, 2, \dots\}$

23.1 Notação O

A notação O define um limite assintótico superior para determinada função. Para definir a notação O , postulamos as seguintes afirmações: seja $T(n)$ o tempo de execução de algum algoritmo, medido em função do tamanho de entrada n , assim, $T(n)$ pode ser expresso por uma função $f(n)$, e assumimos que:

1. O argumento n é um inteiro não negativo (ou seja, está contido em \mathbb{N}), e
2. $f(n)$ é não-negativa para todos os argumentos n .

Agora, seja $g(n)$ uma função definida no domínio dos números naturais \mathbb{N} . Dizemos que “ $f(n)$ é $O(g(n))$ ” se o valor de $f(n)$ é no máximo uma constante vezes $g(n)$, exceto possivelmente para alguns valores pequenos de n . Mais formalmente, dizemos que “ $f(n)$ é $O(g(n))$ ” se existe um inteiro n_0 e uma constante $c > 0$ tal que para todos os inteiros $n \geq n_0$, temos $f(n) \leq cg(n)$. Ou então:

Isto implica no fato de $O(g(n))$ representar um *conjunto* de funções com as propriedade definidas

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todos } n \geq n_0\}.$$

acima. A figura 23.1 mostra graficamente esta situação: com valores de n maiores que n_0 , a função $f(n)$ sempre é menor que $g(n)$ multiplicado por uma determinada constante c . O valor n_0 representa um valor mínimo para que $g(n)$ seja um limite superior assintótico para $f(n)$.

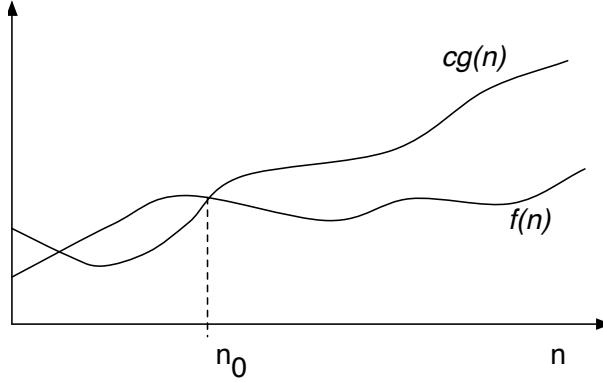


Figura 23.1: Notação O, onde $f(n) = O(g(n))$

Exemplo: suponha que um determinado algoritmo possui os seguintes tempos de execução $T(0) = 1$, $T(1) = 4$, $T(2) = 9$ e, no caso geral, $T(n) = (n+1)^2$. Podemos afirmar que $T(n)$ é $O(n^2)$, ou que $T(n)$ é *quadrático*, porque se $c = 4$ e $n_0 = 1$, então $T(n) = (n+1)^2 \leq 4n^2$ para $n \geq 1$. Para provar essa afirmação, expandimos $(n+1)^2$, resultando em $n^2 + 2n + 1$. Se $n \geq 1$, então $n \leq n^2$ e $1 \leq n^2$. Portanto: $n^2 + 2n + 1 \leq n^2 + 2n^2 + 2 = 4n^2$, provando que $T(n) \leq cf(n)$ para $c = 4$ e $f(n) = n^2$.

Em princípio, pode parecer estranho que, embora $(n+1)^2$ seja maior que n^2 , nós afirmamos que $(n+1)^2$ é $O(n^2)$. De fato, podemos dizer ainda que $(n+1)^2$ possui limite assintótico superior definido por qualquer fração de n^2 , por exemplo, $O(n^2/200)$. Para provar, basta fazer $n_0 = 1$ e $c = 800$. Então, se $n \geq 1$, temos que $(n+1)^2 \leq 800(n^2/200) = 4n^2$.

Estas observações são regidas pelos seguintes princípios gerais:

1. **Fatores constantes não importam:** para qualquer constante $d > 0$ e qualquer função $f(n)$, $f(n)$ é $O(df(n))$. Prova: seja $n_0 = 0$ e $c = 1/d$, então $f(n) \leq c(df(n))$, pois $cd = 1$. Da mesma forma, se sabemos que $f(n)$ é $O(g(n))$, então sabemos que $f(n)$ é $O(dg(n))$ para qualquer $d > 0$, mesmo se d possuir um valor muito pequeno. Isto se justifica porque sabemos que $f(n) \leq c_1 g(n)$ para uma constante c_1 e para todos $n \geq n_0$. Se escolhermos $c = c_1/d$, podemos verificar que $f(n) \leq c(dg(n))$ para $n \geq n_0$.
2. **Termos de menor ordem podem ser desprezados:** suponha que $f(n)$ é um polinômio na forma $a_k n^k + a_{k-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$ onde o coeficiente do termo de maior ordem, a_k , é positivo. Então, podemos desprezar todos os termos exceto o de maior grau (o de maior expoente k) e, pela regra anterior, ignorar a constante a_k , substituindo-a por 1. Portanto, podemos concluir que $f(n)$ é $O(n^k)$. Para provar este princípio, seja $n_0 = 1$ e c igual à soma de todos os coeficientes positivos a_i com $0 \leq i \leq k$. Se um coeficiente a_j é negativo, então certamente $a_j n^j \leq 0$ (pois $n \geq 0$). Se a_j é positivo, então $a_j n^j \leq a_j n^k$ para todo $j < k$, desde que $n \geq 1$. Portanto, $f(n)$ não é maior que n^k multiplicado pela soma dos coeficientes

positivos, ou seja cn^k .

Exemplo: considerando o polinômio $f(n) = 4n^5 + 3n^4 - 10n^3 + n - 15$, como o termo de maior grau é n^5 , podemos afirmar que $f(n)$ é $O(n^5)$. Para provar esta afirmação, seja $n_0 = 1$ e c a soma dos coeficientes positivos. Estes coeficientes são os dos termos de grau 5, 4 e 1, então $c = 4 + 3 + 1 = 8$. Assim, para $n \geq 1$, temos $4n^5 + 3n^4 - 10n^3 + n - 15 \leq 4n^5 + 3n^5 + n^5 = 8n^5$. Portanto $f(n) \leq 8n^5$, ou $f(n) \leq cn^5$ provando que $f(n)$ é $O(n^5)$.

A definição de $O(g(n))$ requer que todas as funções contidas no conjunto $O(g(n))$ sejam assintoticamente não-negativas, pois $0 \leq f(n) \leq cg(n)$ para valores de n suficientemente grandes. Conseqüentemente a própria função $g(n)$ deve ser assintoticamente não-negativa, caso contrário $O(g(n))$ é um conjunto vazio [20].

A notação O é usada para indicar limites superiores para o tempo de execução de algoritmos, porém, muitas vezes somente o limite superior não é suficiente. Por exemplo, apesar eventualmente verificarmos que o tempo de execução de algum algoritmo é quadrático (ou seja $O(n^2)$), podemos dizer também que ele é $O(2^n)$. Ou seja, estamos afirmando que ele não requer mais do que tempo exponencial para resolver o problema. Esta afirmação claramente é “crua” demais para se avaliar o tempo de execução, pois apesar de correta, ela é tecnicamente fraca porque na realidade o algoritmo executa em tempos muito menores que este [22]. Se quisermos obter uma estimativa mais realista do tempo de execução de um algoritmo, precisamos limites mais justos para estas medidas de tempo conforme será visto a seguir.

23.2 Notação Θ

No capítulo anterior, vimos que o algoritmo *SelectionSort* para o problema de ordenação é quadrático, sendo seu tempo de execução assintoticamente proporcional ao quadrado do tamanho de uma instância do problema. Com isso, estamos afirmando que o tempo de execução de *SelectionSort* é $T(n) = \Theta(n^2)$.

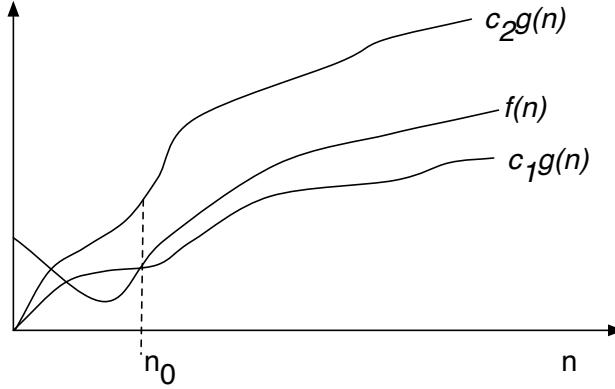
Vamos então definir o significado desta notação. Para uma dada função $g(n)$, denotamos por $\Theta(g(n))$ o *conjunto de funções*

$$\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que} \\ 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ para todos } n \geq n_0\}.$$

Uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existirem constantes positivas c_1 e c_2 tais que $f(n)$ possa ficar compreendida entre $c_1g(n)$ e $c_2g(n)$ para valores suficientemente grandes de n . É importante salientar $\Theta(g(n))$ é um conjunto, portanto quando escrevemos “ $f(n) = \Theta(g(n))$ ”, estamos indicando que $f(n)$ é membro de $\Theta(g(n))$, ou “ $f(n) \in \Theta(g(n))$ ”.

A figura 23.2 ilustra intuitivamente as funções $f(n)$ e $g(n)$, com $f(n) = \Theta(g(n))$. Para todos os valores de $n \geq n_0$, $f(n)$ é maior ou igual a $c_1g(n)$ e menor ou igual a $c_2g(n)$. Assim, podemos dizer que $g(n)$ é limite assintoticamente *justo*, ou assintoticamente *exato* para $f(n)$.

Exemplo: considerando o polinômio $f(n) = \frac{1}{2}n^2 - 3n$, afirmamos que ele é $\Theta(n^2)$. Para provar esta afirmação, pela definição de Θ , devemos determinar constantes positivas c_1, c_2 e n_0 tais que $c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2$ para todo $n \geq n_0$. Dividindo esta expressão por n^2 , obtemos $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq$

Figura 23.2: Notação Θ , onde $f(n) = \Theta(g(n))$

c_2 . Podemos satisfazer a inequação da direita ($\frac{1}{2} - \frac{3}{n} \leq c_2$) para qualquer valor $n \geq 1$ escolhendo $c_2 = 1/2$. Da mesma forma, podemos satisfazer a inequação da esquerda ($c_1 \leq \frac{1}{2} - \frac{3}{n}$) para qualquer valor $n \geq 7$ escolhendo $c_1 = 1/14$. Portanto, escolhendo $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, provamos que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Exemplo: analogamente ao exemplo anterior, podemos usar a definição formal de Θ para verificar que $15n^3 \neq \Theta(n^2)$. Para efeito de contradição, suponha que existam c_2 e n_0 tais que $15n^3 \leq c_2 n^2$ para todo $n \geq n_0$. Então temos $n \leq c_2/15$, que não pode ser satisfeito para valores de n arbitrariamente grandes, visto que c_2 é uma constante.

23.3 Notação Ω

Na seção 23.1, vimos que a notação O define limites assintóticos superiores para uma dada função. Da mesma forma, podemos estabelecer limites assintóticos inferiores pela definição da notação Ω a seguir:

$$\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq cg(n) \leq f(n) \text{ para todos } n \geq n_0\}.$$

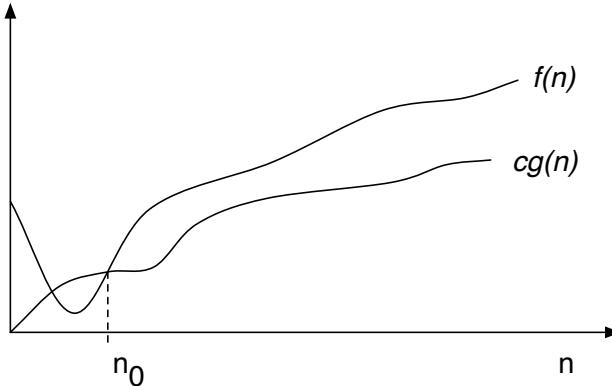
Esta definição atesta que se o tempo de execução de um determinado algoritmo é $\Omega(g(n))$, então, não importando qual instância particular de problema de tamanho n for escolhida como entrada dentre um conjunto de instâncias deste tamanho, o tempo de execução para este conjunto de instâncias é no mínimo uma constante multiplicada por $g(n)$, para n suficientemente grande.

A figura 23.3 ilustra intuitivamente o comportamento de uma função $f(n)$ com $f(n) = \Omega(g(n))$. Pode-se ver que para todos os valores de $n \geq n_0$, $f(n)$ é maior ou igual a $cg(n)$.

As definições assintóticas apresentadas até agora permitem que se prove o seguinte teorema:

Teorema: para qualquer par de funções $f(n)$ e $g(n)$, $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Para demonstrar deste teorema, vamos verificar que $f(n) = an^2 + bn + c = \Theta(n^2)$. Dadas quaisquer constantes a , b e c com $a > 0$, implica em podermos provar que $an^2 + bn + c = O(n^2)$ e $an^2 + bn + c =$

Figura 23.3: Notação Ω , onde $f(n) = \Omega(g(n))$

$\Omega(n^2)$. Este teorema freqüentemente é usado para provar limites assintoticamente exatos a partir de limites superiores e inferiores [9].

23.4 Análise do algoritmo Bubble Sort

Para exemplificar a notação Ω , vamos examinar o algoritmo *BubbleSort*. Este algoritmo é um dos mais simples para resolver o problema de ordenação, e funciona da seguinte forma: tomando como entrada um vetor com n elementos, o algoritmo inicia comparando o primeiro e o segundo elementos do vetor. Se o primeiro elemento for maior que o segundo, eles são trocados. A seguir, o algoritmo repete este procedimento com o próximo par de valores no vetor (segundo e terceiro elementos), até chegar ao último par. Ao final desse processo, o maior elemento estará colocado na última posição do vetor. Todo o procedimento é repetido até a penúltima posição do vetor, e assim por diante, até que todo o vetor esteja ordenado.

A figura 23.4 ilustra este procedimento tomando como exemplo de entrada uma vetor $A = \langle 25, 37, 12, 48, 57, 92, 33, 86 \rangle$. Os quadrados cinza escuro mostram a parte do vetor que já se encontra ordenada. Na primeira “passada”, o algoritmo move o maior valor para a última posição do vetor. Na segunda passada o segundo maior valor vai para a penúltima posição, e assim por diante. Neste exemplo é importante notar que, depois da quarta passada do algoritmo, o vetor encontra-se ordenado e a partir daí nenhuma troca de valores será feita.

O pseudo código do *BubbleSort* para ordenar um vetor A com n elementos é mostrado no quadro 23.1. A variável booleana “troca” sinaliza quando houve uma varredura no vetor sem ocorrer nenhuma troca de valores. Neste caso o vetor já está ordenado e o algoritmo pode encerrar sua execução.

Se analisarmos o pseudo-código do *BubbleSort* da mesma forma que fizemos com o algoritmo *SelectionSort* na seção 22.1, montamos o quadro 23.1 com custo e a quantidade de vezes que cada linha é executada.

	25 x 37		25 x 12 troca
	37 x 12 troca		57 x 37
	37 x 48		37 x 48
	48 x 57		48 x 57
	57 x 92		57 x 33 troca
	92 x 33 troca		57 x 86 troca
	92 x 86 troca		final da 2 ^a passada
	final da 1 ^a passada		

	12 x 25		12 x 25
	25 x 37		25 x 37
	37 x 48		37 x 33 troca
	48 x 33 troca		37 x 48 troca
	48 x 57 troca		final da 4 ^a passada
	final da 3 ^a passada		

Figura 23.4: Exemplo de execução do algoritmo BubbleSort

Quadro 23.1: Custo do BubbleSort

linha	custo	n. ^o de execuções
2	c_2	1
3	c_3	$n+1$
4	c_4	n
5	c_5	$\sum_{i=1}^n t_i$
6, 7, 8, 9, 10	$c_6, c_7, c_8, c_9, c_{10}$	$\sum_{i=1}^n (t_i - 1)$
11	c_{11}	n

Multiplicando o custo de cada linha com a quantidade de vezes que ela é executada, chegaremos à seguinte função para o tempo de execução do algoritmo:

```

1 Algoritmo: BubbleSort(A)
2 int i, j;
3 para i ← n até 1 faça
4   boolean troca ← falso;
5   para j ← 1 até (i-1) faça
6     se A[j] > A[j + 1] então
7       int temp ← A[j];
8       A[j] ← A[j + 1];
9       A[j + 1] ← temp;
10      troca ← verdadeiro;
11    se troca ≠ verdadeiro então retorna;
```

Algoritmo 23.1: BubbleSort

$$T(n) = c_2 + c_3(n + 1) + c_4n + c_5 \sum_{i=1}^n t_i + (c_6 + c_7 + c_8 + c_9 + c_{10}) \sum_{i=i}^n (t_i - 1) + c_{11}n.$$

Ao contrário do que acontece com o *SelectionSort*, entradas do mesmo tamanho podem produzir tempos de execução diferentes para o algoritmo *BubbleSort*. O pior caso para este algoritmo ocorre quando os valores da entrada estão em ordem decrescente, e o algoritmo executa em tempo quadrático (deixamos essa verificação como exercício para o leitor). Porém, quando os valores da entrada já estão ordenados, o algoritmo encerra sua execução logo na primeira passada completa pelo vetor. As linhas 3, 4 e 11 são executadas somente uma vez, a linha 5 é executada n vezes, a linha 6 é executada $n - 1$ vezes e as linhas 7 a 10 não são executadas, resultando em:

$$T(n) = c_2 + c_3 + c_4 + c_5n + c_6(n - 1) + c_{11} \therefore$$

$$T(n) = (c_5 + c_6)n + (c_2 + c_3 + c_4 + -c_6 + c_{11}).$$

Como os termos c_k são constantes, este resultado pode ser simplificado em $T(n) = an + b$, onde a e b são constantes dependentes de c_k . Isto nos mostra que, apesar do algoritmo rodar em tempo quadrático no pior caso, ele também pode rodar em tempo linear dependendo dos dados de entrada. Se o tempo de execução do *BubbleSort* é linear no melhor caso e quadrático no pior, podemos dizer que para este algoritmo, o tempo de execução fica entre $\Omega(n)$ e $O(n^2)$.

O que significa a taxa de crescimento de uma função?

No presente capítulo fizemos a análise do algoritmo *BubbleSort*, concluíndo que sua ordem de complexidade no pior caso é quadrática. Vimos também que fatores constantes podem ser desprezados na análise assintótica das funções que descrevem ordens de complexidade de algoritmos. Para comparação de desempenho de diferentes algoritmos que resolvem um mesmo problema, é importante analisar a taxa de crescimento das funções quando o tamanho da entrada tende a crescer. Mas, na prática, o que isso tudo significa?

O quadro 23.2 exemplifica o crescimento de funções comuns na análise de algoritmos em função do crescimento do tamanho do problema dado por n [25]. É mostrado o tempo necessário para cada algoritmo executar ξ operações, sendo que cada operação demora hipoteticamente um nanosegundo (10^{-9} segundos).

Quadro 23.2: Crescimento de funções

n	$\xi = \log n$	n	$n \log n$	n^2	2^n	$n!$
10	0.003 μ s	0.01 μ s	0.033 μ s	0.1 μ s	1 μ s	3.63 ms
20	0.004 μ s	0.02 μ s	0.086 μ s	0.4 μ s	1 ms	77.1 anos
30	0.005 μ s	0.03 μ s	0.147 μ s	0.9 μ s	1 s	10^{15} anos
40	0.005 μ s	0.04 μ s	0.213 μ s	1.6 μ s	18.3 min	
50	0.006 μ s	0.05 μ s	0.282 μ s	2.5 μ s	13 dias	
10^2	0.007 μ s	0.10 μ s	0.644 μ s	10 μ s	8×10^{13} anos	
10^3	0.010 μ s	1 μ s	9.966 μ s	1 ms		
10^4	0.013 μ s	10 μ s	130 μ s	100 ms		
10^5	0.017 μ s	0.1 ms	1.67 ms	10 s		
10^6	0.020 μ s	1 ms	19.93 ms	16.7 min		
10^7	0.023 μ s	0.01 s	0.23 s	1.16 dias		
10^8	0.027 μ s	0.10 s	2.66 s	115.7 dias		
10^9	0.030 μ s	1 s	29.90 s	31.7 anos		

Observando este quadro, podemos chegar às seguintes conclusões:

- (i) todos os algoritmos levam praticamente o mesmo tempo com $n = 10$;
- (ii) o uso do algoritmo com tempo de execução $n!$ se torna impraticável antes de $n = 20$;
- (iii) o algoritmo com tempo de execução igual a 2^n possui uma abrangência maior de utilização, mas também se torna impraticável para $n > 40$;
- (iv) o algoritmo quadrático (com tempo igual a n^2) é rápido até $n = 100$, mas seu desempenho logo se deteriora daí pra frente, tornando-se impraticável com $n > 10^6$;
- (v) os algoritmos com tempo n e $n \log n$ continuam com bom desempenho em problemas com entrada de até um bilhão de ítems; e
- (vi) na prática provavelmente não encontraremos um problema real onde um algoritmo $\Theta(\log n)$ seja lento demais.

Portanto, mesmo ignorando fatores constantes, podemos ter uma boa idéia se determinado algoritmo vai executar em um tempo plausível sendo dado o tamanho de entrada de uma instância do problema.

23.5 Combinando notações assintóticas

A natureza da notação assintótica nos permite escrever “ $n = O(n^2)$ ”, por exemplo. Essa expressão pode parecer estranha a princípio, mas se nos lembarmos que este tipo de notação define um *conjunto de funções*, a expressão acima indica que a função linear $f(n) = n$ pertence a um conjunto de funções que possuem cn^2 como um limite assintótico superior para determinada constante $c > 0$ e para valores de n suficientemente grandes. De fato, o sinal de igualdade em $n = O(n^2)$ na verdade significa *pertinência*, ou seja $n \in O(n^2)$ [20].

Estas observações mostram que existem técnicas gerais para combinar a notação assintótica em expressões que remetem a expressões relacionais da álgebra (como a igualdade mostrada acima) e levam a propriedades para a combinação de notação assintótica. Assim, uma série de regras podem ser inferidas. A primeira delas é a regra da soma.

23.5.1 Soma

Devido ao fato exposto acima, também é possível escrever $3n^2 + n - 2 = 3n^2 + \Theta(n)$. Esta equação pode ser interpretada da seguinte maneira: quando a notação assintótica aparece em alguma equação (ou inequação), ela denota alguma função desconhecida. Por exemplo: a equação $3n^2 + n - 2 = 3n^2 + \Theta(n)$ significa que $3n^2 + n - 2 = 3n^2 + f(n)$, e que $f(n)$ é alguma função pertencente a $\Theta(n)$. Neste caso em particular, $f(n) = n - 2$, que realmente é $\Theta(n)$.

Em alguns casos, a notação assintótica pode aparecer no lado esquerdo da equação. Por exemplo: $4n^3 + \Theta(n^2) = \Theta(n^3)$. Esta equação significa que, não importando qual função desconhecida seja escolhida para o lado esquerdo da igualdade, sempre é possível escolher uma outra função para o lado direito de forma que a igualdade seja válida. Ou seja, nesta equação, para qualquer função $f(n) \in \Theta(n^2)$, existe uma função $g(n) \in \Theta(n^3)$ tal que $4n^3 + f(n) = g(n)$ para qualquer n .

As expressões acima também podem ser interpretadas da seguinte maneira: suponha que estamos analisando um algoritmo composto por duas partes. O tempo de execução de uma delas é $O(n^2)$ e da outra é $O(n^3)$. Esses tempos poderiam ser somados para se obter o tempo de execução do algoritmo todo.

Formalmente, supondo que se saiba que determinado tempo de execução $T_1(n)$ é $O(g(n))$, e outro tempo de execução T_2 é $O(f(n))$. Além disso, suponha que $f(n)$ não cresce mais rapidamente que $g(n)$, ou seja, $f(n) \leq g(n)$. Então podemos concluir que $T_1(n) + T_2(n)$ é $O(g(n))$.

Para provar esta afirmação, sabemos que existem constantes n_1, n_2, n_3, c_1, c_2 e c_3 tais que:

- a) Se $n \geq n_1$, então $T_1(n) \leq c_1g(n)$;
- b) Se $n \geq n_2$, então $T_2(n) \leq c_2f(n)$;
- c) Se $n \geq n_3$, então $f(n) \leq c_3g(n)$.

Seja n_0 o maior valor entre n_1, n_2 e n_3 , tal que as inequações (a), (b) e (c) sejam satisfeitas para todo $n \geq n_0$. Então temos $T_1(n) + T_2(n) \leq c_1g(n) + c_2f(n)$. Utilizando a inequação (c), podemos escrever $T_1(n) + T_2(n) \leq c_1g(n) + c_2c_3g(n)$.

Portanto, temos que $T_1(n) + T_2(n) \leq cg(n)$, onde $c = c_1 + c_2c_3$, e assim concluímos que $T_1(n) + T_2(n)$ é $O(g(n))$.

23.5.2 Transitividade

A propriedade de transitividade para notação assintótica é semelhante à dos números inteiros e reais. Uma relação é dita transitiva se obedecer à seguinte lei: “se $A \leq B$ e $B \leq C$, então $A \leq C$ ”. Por exemplo, se $4 \leq 7$ e $7 \leq 11$, então sabemos que $4 \leq 11$.

A notação assintótica O , por exemplo, também possui transitividade: se $f(n)$ é $O(g(n))$ e $g(n)$ é $O(h(n))$, então implica em $f(n)$ ser $O(h(n))$.

Para provar esta propriedade, suponha que $f(n)$ é $O(g(n))$. Então, pela definição de O , existem constantes n_1 e c_1 tais que $f(n) \leq c_1 g(n)$ para todo $n \geq n_1$. Analogamente, se $g(n)$ é $O(h(n))$, então existem constantes n_2 e c_2 tais que $g(n) \leq c_2 h(n)$ para todo $n \geq n_2$. Seja n_0 o maior valor dentre n_1 e n_2 , e seja $c = c_1 c_2$. Então, para todo $n \geq n_0$, sabemos que $f(n) \leq c_1 g(n)$ e $g(n) \leq c_2 h(n)$. Portanto $f(n) \leq c_1 c_2 h(n)$, provando que $f(n)$ é $O(h(n))$.

Esta propriedade também se aplica às outras notações assintóticas. Resumidamente podemos listar as seguintes regras quanto à transitividade:

- se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, então $f(n) = \Theta(h(n))$;
- se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$;
- se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$, então $f(n) = \Omega(h(n))$.

23.5.3 Reflexividade

A propriedade de reflexividade afirma que determinada função $f(n)$ é $O(f(n))$. Como prova esta propriedade, basta usar a definição da notação O . Se $f(n)$ é $O(f(n))$, então existem constantes c e n tais que $f(n) \leq cf(n)$ para todo $n \geq n_0$. Seja $c = 1$ e n_0 , então $f(n) \leq f(n)$ para todo valor $n \geq 0$, atendendo a definição de O e provando a propriedade de reflexividade. Generalizando, temos que:

- $f(n) = \Theta(f(n))$;
- $f(n) = O(f(n))$;
- $f(n) = \Omega(f(n))$.

23.5.4 Simetria

Por fim, ainda temos as relações de simetria e simetria transposta, que intuitivamente podem ser observadas nas figuras 23.1, 23.2 e 23.3 e são definidas a seguir:

- Simetria: $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$;
- Simetria transposta: $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$.

Ordens de complexidade polinomiais e não-polinomiais (NP)

O grau de um polinômio é o maior expoente encontrado entre seus termos. Por exemplo, o grau do polinômio $f(n) = 4n^5 + 3n^4 - 10n^3 + n - 15$ é 5 porque $4n^5$ é o termo de maior ordem. Funções *exponenciais* possuem a forma a^n para $a > 1$. Qualquer função exponencial cresce mais rápido que qualquer função polinomial. Ou seja, é possível demonstrar que para qualquer polinômio $p(n)$ que $p(n)$ é $O(a^n)$. Da mesma forma, nenhuma função exponencial a^n com $a > 1$ é $O(p(n))$ para qualquer polinômio $p(n)$ [16].

Os algoritmos analisados neste documento são algoritmos da classe P, com tempo polinomial. Isto é, para entradas de tamanho n , seu tempo de execução no pior caso é $O(n^k)$ para alguma constante

k. Existe uma classe de problemas, chamados NP, para os quais não se conhecem soluções polinomiais. Um problema é dito não-deterministicamente polinomial (NP) se uma solução hipotética do problema puder ser verificada em tempo polinomial. Caso contrário, um problema não polinomial é chamado de NP completo. Caso consigamos verificar que determinado algoritmo é NP-completo, então ele é intratável. Neste caso, é melhor procurar desenvolver um algoritmo aproximativo do que tentar buscar a solução exata [25].

No próximo capítulo, estudaremos a análise de uma importante classe de algoritmos, os algoritmos *recursivos*.

23.6 Exercícios propostos

1. Considere as 2 funções $f_1: n^2$ e $f_2: n^3$

Para cada i e j iguais a 1 e 2, determine se $f_i(n)$ é $O(f_j(n))$. Encontre valores para n_0 e c que provem essa relação, ou estabeleça valores para derivar uma contradição que prove que $f_i(n)$ não é $O(f_j(n))$.

2. Encontre valores para n_0 e c que provem que as relações abaixo são verdadeiras.

- a) n^2 é $O(0.001n^3)$
- b) $25n^4 - 19n^3 + 13n^2 - 106n + 77$ é $O(n^4)$
- c) 2^{n+10} é $O(2^n)$
- d) n^{10} é $O(3^n)$

3. Considere o problema de **busca**:

Entrada: seqüência de n números $A = \langle a_1, a_2, \dots, a_n \rangle$ e um valor v .

Saída: um valor i tal que $v = A[i]$ ou o valor especial *NULO* caso v não apareça em A .

Escreva o pseudo código de um algoritmo para resolver este problema. Determine O e Ω do seu algoritmo.

4. Considere novamente o problema de **busca**. Observe que se a seqüência A estiver ordenada, podemos comparar v com o elemento no meio da seqüência e eliminar metade da seqüência para fazer uma nova comparação. A **busca binária** é um algoritmo que executa esse tipo de procedimento. Escreva o peseudo código de um algoritmo iterativo que execute busca binária. Prove que o algoritmo é $\Theta(n)$.

Capítulo 24

Análise de algoritmos recursivos

Até agora vimos exemplos de análise de dois algoritmos iterativos (*SelectionSort* e *BubbleSort*). Porém, muitos problemas em Ciência da Computação são modelados e resolvidos através de definições e algoritmos recursivos. O presente capítulo discute recorrência no contexto de algoritmos recursivos e apresenta as bases para análise de complexidade deste tipo de algoritmo.

24.1 Definições recursivas

Numa definição recursiva, classes de objetos ou fatos são definidos em termos de objetos ou fatos da mesma classe. A definição deve ter um significado. Por exemplo, a definição “um cavalo é um cavalo da mesma cor” não faz sentido e portanto não é uma definição válida. Além disso, a definição recursiva também não pode ser paradoxal. Por exemplo, “uma chave é uma chave se e somente se não for uma chave”. Este é outro exemplo de definição recursiva inválida. Ao invés disso, a definição recursiva deve conter as seguintes regras:

- a) Uma ou mais regras-base (ou regras básicas) nas quais objetos simples ou elementares são definidos, e;
- b) Uma ou mais regras indutivas, pelas quais objetos maiores são definidos em termos de objetos menores pertencentes à coleção de objetos definidos.

Exemplo: o fatorial de um número inteiro positivo n é denotado por $n!$ e seu valor é igual a $n \times (n - 1) \times (n - 2) \times \dots \times 3 \times 2 \times 1$. Por exemplo, $4! = 4 \times 3 \times 2 \times 1 = 24$. O caso especial $0!$ é definido com o valor 1. O fatorial pode ser definido recursivamente da seguinte forma:

$$n! = \begin{cases} 1, & \text{se } n \in \{0, 1\} \text{ (regra-base);} \\ n(n-1)!, & \text{se } n > 1 \text{ (regra indutiva).} \end{cases}$$

Exemplo: em estruturas de dados temos muitos exemplos de definições recursivas para estruturas comuns. A figura 24.1 mostra esquematicamente a definição recursiva de árvores binárias, onde uma árvore binária é: (i) uma árvore vazia (regra-base) ou (ii) um nó raiz seguido de duas sub-árvores, a sub-árvore da esquerda – *sae* – e a sub-árvore da direita – *sad* (regra indutiva).

A seguir, vamos discutir a noção de recursividade no projeto e análise de complexidade de algoritmos.

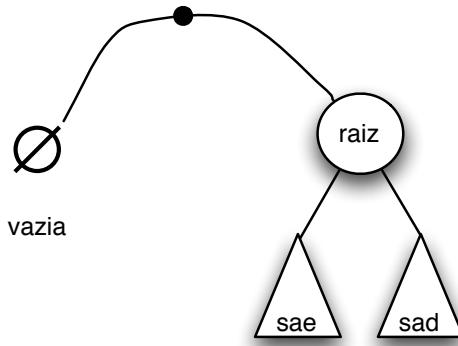


Figura 24.1: Definição recursiva de árvores binárias

24.2 Algoritmos recursivos

A idéia de recursividade é também comum em algoritmos, muitos algoritmos úteis na computação são recursivos. Em termos de codificação deste tipo de algoritmo, significa que uma instância de execução de um procedimento (ou função, ou método) recursivo chama outra instância do próprio procedimento. Por exemplo, a definição recursiva da função factorial pode levar à implementação recursiva¹ mostrada no quadro 24.1. A recursão acontece na linha 5, quando o procedimento chama uma outra instância de si próprio para calcular o fatorial de um número menor que o da instância atual.

```

1 Algoritmo: Fatorial(n)
2 se ( $n=0$ ) ou ( $n=1$ ) então
3   | retorna 1;
4 senão
5   | retorna  $n \times$  Fatorial( $n-1$ );

```

Algoritmo 24.1: Implementação recursiva do fatorial

Em geral, este tipo de algoritmo resolve o problema através de uma abordagem de “divisão e conquista”, composta por em três passos básicos: **divisão**, **conquista** e **combinação** [23]. Neste paradigma, o problema é *dividido* sucessivamente em sub-problemas menores até chegar em sub-problemas elementares de fácil solução, neste ponto o problema é resolvido, ou *conquistado*. A partir daí, os problemas menores solucionados são *combinados* gerando soluções de problemas maiores até que se chegue à solução do problema original. No caso do fatorial, a divisão acontece na chamada recursiva do cálculo do fatorial de um número menor, os casos elementares são para $n = 0$ e $n = 1$, e a conquista vem na multiplicação do resultado da chamada recursiva com o “ n ” atual.

Outro exemplo de algoritmo baseado em estratégia de divisão e conquista é o método de ordenação *MergeSort*. Neste método, seqüências de n números são divididas recursivamente em seqüências de tamanho $n/2$. O problema é “conquistado”, ou resolvido, quando se chega a seqüências de apenas

¹É possível também implementar a função factorial de forma iterativa, porém a implementação recursiva resulta em um código mais aderente à definição recursiva da função.

um número que, por definição, já estão ordenadas, constituindo portanto soluções de sub-problemas elementares.

A partir daí as soluções elementares são combinadas no procedimento chamado *merge*. Este procedimento combina dois vetores ordenados. Seu funcionamento pode ser exemplificado pela combinação de cartas de baralho. Suponha que tenhamos duas pilhas de cartas ordenadas e queremos combiná-las para ter apenas uma pilha. Para isso, basta olhar as primeiras cartas das duas pilhas e escolher a menor delas. Esta carta menor deve ser retirada e colocada na pilha resultante com a face que contém o número virada para baixo. Depois disso, o procedimento deve ser repetido com as duas cartas que estão no topo das duas pilhas originais. A figura 24.2 demonstra o funcionamento do *MergeSort* para uma seqüência de oito números.

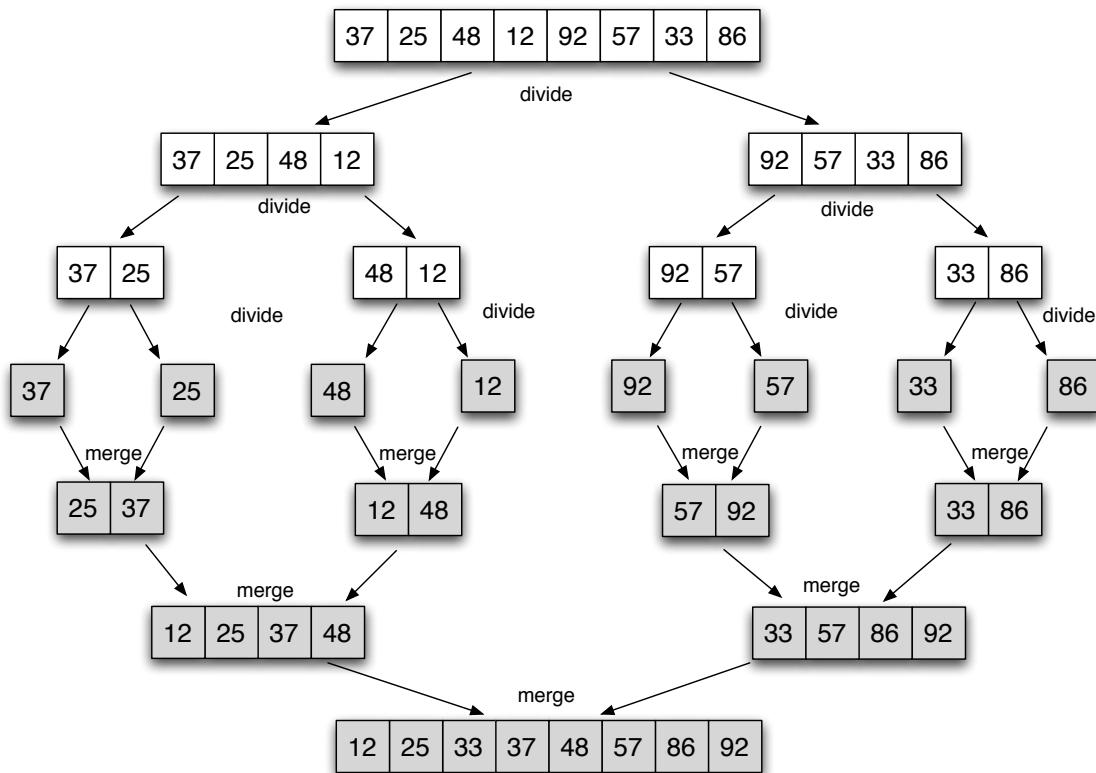


Figura 24.2: Exemplo de execução do algoritmo *MergeSort*

Conforme pode ser visto no pseudo-código do quadro 24.2, o algoritmo *MergeSort* recebe o vetor a ser ordenado (A) e os índices do primeiro e do último elemento. Se $ini < fim$, então calculase o índice do meio² do vetor (ou do sub-vetor) e chama-se recursivamente *MergeSort* para as duas metades do vetor. Caso contrário, tem-se um caso elementar, com vetor contendo zero ou um elemento apenas. Após isso, chama-se o procedimento *Merge* para combinar dois sub-vetores ordenados.

O procedimento *Merge*, cujo pseudocódigo é mostrado no quadro 24.3, combina duas regiões ordenadas do vetor A . Podemos imaginar as duas regiões como as duas pilhas de cartas ordenadas, sendo combinadas em uma pilha única. A chamada $Merge(A, ini, meio, fim)$ assume que os subvetores $A[ini \dots meio]$ e $A[(meio + 1) \dots fim]$ estão ordenados e $ini \leq meio < fim$. O

²A expressão $\lfloor x \rfloor$ denota o maior inteiro menor ou igual a x .

```

1 Algoritmo: Mergesort(A, ini, fim)
2 se  $ini < fim$  então
3    $meio \leftarrow \lfloor (ini + fim)/2 \rfloor;$ 
4   MergeSort(A, ini, meio);
5   MergeSort(A, meio+1, fim);
6   Merge(A, ini, meio, fim);

```

Algoritmo 24.2: MergeSort

procedimento Merge copia os valores dos dois subvetores para vetores auxiliares ESQ e DIR e os devolve combinados em ordem para o vetor A .

Para evitar que se verifique em cada passo se chegamos ao final de um dos dois vetores auxiliares, é usado um valor “sentinela” igual a ∞ . Dessa forma, quando o valor ∞ for alcançado, todos os valores do outro vetor auxiliar possuem valores menores, e são transferidos para o vetor A .

```

1 Algoritmo: Merge(A, ini, meio, fim)
2  $n_1 \leftarrow meio - ini + 1;$ 
3  $n_2 \leftarrow fim - meio;$ 
4 crie vetores auxiliares  $ESQ[1 \dots (n_1 + 1)]$  e  $DIR[1 \dots (n_2 + 1)]$ ;
5 para  $i \leftarrow 1$  até  $n_1$  faça
6    $| ESQ[i] \leftarrow A[ini + i - 1];$ 
7 para  $j \leftarrow 1$  até  $n_2$  faça
8    $| DIR[j] \leftarrow A[meio + j];$ 
9  $ESQ[n_1 + 1] \leftarrow \infty;$ 
10  $DIR[n_2 + 1] \leftarrow \infty;$ 
11  $i \leftarrow 1;$ 
12  $j \leftarrow 1;$ 
13 para  $k \leftarrow ini$  até  $fim$  faça
14   se  $ESQ[i] \leq DIR[j]$  então
15      $| A[k] \leftarrow ESQ[i];$ 
16      $| i \leftarrow i + 1;$ 
17   senão
18      $| A[k] \leftarrow DIR[j];$ 
19      $| j \leftarrow j + 1;$ 

```

Algoritmo 24.3: Procedimento Merge

A figura 24.3 ilustra a última execução do procedimento Merge no exemplo da figura 24.2. Esta execução corresponde à chamada $Merge(A, 1, 4, 8)$. Os quadrados escuros representam os valores finais em suas posições corretas nos vetores.

Observando o pseudocódigo do quadro 24.3, podemos verificar as linhas 2 a 4 e 9 a 12 executam em tempo constante. Os laços “Para” das linhas 5 a 8 levam tempo $\Theta(n_1+n_2) = \Theta(n)$ e o laço “Para” das linhas 13 a 19 executa n passos cada um com tempo constante, levando então tempo $\Theta(n)$. Portanto, concluimos que o procedimento Merge roda em tempo $\Theta(n)$, onde $n = fim - ini + 1$.

24.3 Análise de algoritmos recursivos

Para analisar um algoritmo recursivo, é necessário descrever seu tempo de execução através de uma equação de recorrência, que descreve o tempo de execução de uma problema de tamanho n em função de instâncias de tamanho menor do mesmo problema. A equação de recorrência pode então ser resolvida com auxílio de conceitos matemáticos [21].

A construção da equação de recorrência é feita a partir dos três passos básicos dos algoritmos recursivos (dividir, conquistar e combinar). Portanto, seja $T(n)$ o tempo de execução de um problema de tamanho n . Se o problema é suficientemente pequeno, por exemplo, com $n \leq c$, para determinada constante c , a solução elementar leva tempo constante, que é $\Theta(1)$. Supondo que a divisão do problema leva a a sub-problemas, cada qual de tamanho $1/b$ do tamanho original (por exemplo, para o *MergeSort*, a e b são iguais a 2). Seja $D(n)$ o tempo necessário para dividir o problema e $C(n)$ o tempo para combinar as soluções de sub-problemas na solução do problema original, temos a seguinte equação de recorrência:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n \leq c; \\ aT(n/b) + D(n) + C(n), & \text{caso contrário.} \end{cases} \quad (24.1)$$

Para demonstrar a utilização da equação de recorrência, vamos aplicá-la ao algoritmo *MergeSort* apresentado na seção anterior. O algoritmo funciona para problemas de tamanho ímpar, mas para simplificação da presente análise, vamos considerar que o tamanho do problema é potência de 2.

A **divisão** do problema acontece na linha 3 do algoritmo do quadro 24.2 e consiste apenas no cálculo do índice do meio do sub-vetor, levando portanto tempo constante, ou seja, $D(n) = \Theta(1)$. A **conquista** do problema é feita pela resolução de dois sub-problemas, cada um com tamanho $n/2$, o que acrescenta $2T(n/2)$ ao tempo de execução. A **combinação** de dois sub-problemas em um problema de tamanho n é feita pelo procedimento Merge e leva tempo $C(n) = \Theta(n)$ conforme foi visto na seção anterior. De acordo com as propriedades das combinações de notações assintóticas (seção 23.5), temos que $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$. Substituindo estes termos na equação geral de recorrência, temos a seguinte equação de recorrência para o tempo de execução do algoritmo *MergeSort* no pior caso:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1; \\ 2T(n/2) + \Theta(n), & \text{se } n > 1. \end{cases} \quad (24.2)$$

Após a montagem da equação de recorrência, é preciso resolvê-la para se obter a complexidade do algoritmo recursivo correspondente. Para isso existem três métodos: o método da árvore de recorrência, o método de substituição e o método baseado no teorema geral. A seguir, veremos em maiores detalhes cada um destes métodos.

24.3.1 Método da árvore de recorrência

O método da árvore de recorrência também é chamado de método iterativo pois promove-se a iteração da recorrência para construção da árvore. A idéia central deste método é expandir a árvore e expressar a soma de seus termos em função de n e de certas condições iniciais. Este método é particularmente adequado para análise de complexidade de algoritmos de divisão e conquista.

Para demonstrar o método, vamos tentar resolver a equação de recorrência 24.2, do algoritmo *mergeSort*. Seja a constante c o tempo necessário para resolver problemas de tamanho 1. Então esta equação pode ser reescrita da seguinte forma:

$$T(n) = \begin{cases} c, & \text{se } n = 1; \\ 2T(n/2) + cn, & \text{se } n > 1. \end{cases} \quad (24.3)$$

O desenvolvimento desta equação leva à árvore de recorrência exibida na figura 24.4. Numa árvore de recorrência, cada nó representa o custo de um único sub-problema dentro do conjunto de chamadas recursivas de um determinado procedimento. O custo de cada nível é obtido a partir da soma dos custos de todos os nós do nível, e o custo total é calculado através da soma dos custos de todos os níveis da árvore.

Inicialmente, a figura 24.4a mostra o tempo total $T(n)$. Em b, temos o início da expansão da árvore, onde a raiz cn representa o tempo da solução no nível mais alto da árvore, e as duas sub-árvore representam as duas recursões, ou seja, os tempos de resolução dos dois sub-problemas. O custo de cada um deles é $cn/2$. Continuando a expansão da árvore, chega-se a n problemas de tamanho 1, com tempo de execução constante igual a c , resultando na árvore mostrada na figura 24.4d.

Adicionando os custos em cada nível da árvore temos um custo cn no primeiro nível. No próximo nível temos $c(n/2) + c(n/2) = cn$. A seguir, temos $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, e assim por diante. No último nível, temos n nós com custo c cada, resultando em cn . O número total de níveis na árvore é $\log_2 n + 1$. Para comprovar isso, basta verificar que quando $n = 1$ a ávore só possui um nível. Como $\log_2 1 = 0$, então o número correto de níveis é dado por $\log_2 n + 1$.

Para computar o custo total representado pela equação 24.3, basta somar os custos de cada nível da árvore. Temos $\log_2 n + 1$ níveis com custo cn cada um. Portanto o custo total é $cn(\log_2 n + 1) = cn \log_2 n + cn$. Desprezando o termo de menor ordem e promovendo a mudança de base do logaritmo, temos que a ordem de complexidade do *MergeSort* é $\Theta(n \log n)$.

Um exemplo menos simples seria o da resolução da equação de recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$, onde vamos tentar encontrar o limite superior para o tempo de execução do algoritmo. Sem perda de generalidade, podemos iniciar criando a árvore de recorrência para a equação $T(n) = 3T(n/4) + cn^2$, sendo o coeficiente constante $c > 0$.

A árvore de recorrência é mostrada na figura 24.5. Para facilitar a análise, é conveniente assumirmos que n é potência de 4. Em 24.5a é mostrado $T(n)$, que sofre uma primeira expansão em 24.5b. O termo cn^2 na raiz da árvore representa o custo da primeira recursão, e suas três sub-árvore representam os custos dos três sub-problemas de tamanho $n/4$. Na figura 24.5c cada uma destas sub-árvore é expandida. Os nós de custo $T(n/4)$ são expandidos em subárvore com custo $c(n/4)^2$ na raiz e três sub-árvore filhas com custos $T(n/16)$. A expansão da árvore continua conforme descrito na equação de recorrência até que todas as suas partes constituintes sejam estruturadas conforme a figura 24.5d.

A cada nova recorrência, o tamanho do problema diminui. Portanto eventualmente chega-se a uma condição de contorno. Como assumimos que n é potência de 4, as folhas da árvore representam sub-problemas de tamanho $n = 1$. Para cálculo do custo total da árvore é necessário saber a sua profundidade. O tamanho do sub-problema para um nó na profundidade i é igual a $n/4^i$. Sub-

problemas de tamanho $n = 1$ estão na profundidade máxima da árvore i_{max} . Então, neste caso $n/4^{i_{max}} = 1$, ou equivalentemente $i_{max} = \log_4 n$. Assim, a árvore possui $\log_4 n + 1$ níveis ($0, 1, 2, \dots, \log_4 n$).

Em seguida, é necessário calcular o custo de cada nível da árvore. Um determinado nível possui o triplo do número de nós do nível acima. Então, o i -ésimo nível possui 3^i nós. O custo de cada nó do primeiro ao penúltimo nível é igual a $c(n/4^i)^2$, com $i = 0, 1, 2, \dots, (\log_4 n - 1)$. Multiplicando o custo de cada nó pela quantidade de nós de cada nível, temos o custo total de cada nível da árvore dado por $3^i c(n/4^i)^2 = (3/16)^i cn^2$.

No nível das folhas, com $i_{max} = \log_4 n$, temos um total de $3^{i_{max}}$ nós. Substituindo o valor de i_{max} , temos $3^{i_{max}} = 3^{\log_4 n} = n^{\log_4 3}$ nós no último nível da árvore³. Se cada nó contribui com um custo $T(1) = \Theta(1^2) = \Theta(1)$, então temos um custo total $n^{\log_4 3}T(1) = \Theta(n^{\log_4 3})$ neste nível.

Finalmente, para determinar o custo total da árvore, basta somar os custos de todos os níveis:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \therefore \\ T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}). \end{aligned} \quad (24.4)$$

A equação acima contém uma série geométrica definida como:

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1},$$

o que nos leva a reescrever a equação 24.4 da seguinte forma:

$$T(n) = \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}).$$

Esta equação é um tanto difícil de resolvê-la. Porém, podemos tirar proveito do fato de estarmos fazendo uma análise assintótica da complexidade da equação de recorrência. Assim, podemos considerar o tamanho do problema tendendo a infinito e a série geométrica infinita decrescente (com $|x| < 1$) como um limite superior, podendo ser expressa como:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x},$$

então, temos:

$$T(n) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \therefore$$

³Lembramos que $a^{\log_b c} = c^{\log_b a}$.

$$T(n) < \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \therefore$$

$$T(n) < \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \therefore$$

$$T(n) = O(n^2).$$

O método de resolução pela expansão da árvore de recorrência constitui uma forma intuitiva de avaliação de complexidade de algoritmos recursivos do tipo divisão e conquista. Entretanto, em alguns casos a resolução por este método é demasiadamente intrincada ou até mesmo impossível. A seguir veremos o método por substituição, que pode ser uma alternativa nesses casos.

24.3.2 Método de resolução por substituição

O método de resolução por substituição consiste em inicialmente estimar uma solução hipotética para o problema, e depois usar indução matemática para determinar as constantes envolvidas e provar a corretude da solução.

Como exemplo, vamos utilizar este método para provar que a solução da equação de recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ encontrada na seção anterior é correta. Sendo $T(n) = O(n^2)$ um limite superior para complexidade desta recorrência, queremos provar que $T(n) \leq dn^2$ para alguma constante $d > 0$. Usando a mesma constante $c > 0$ da seção anterior, temos:

$$T(n) \leq 3T(\lfloor n/4 \rfloor) + cn^2 \therefore T(n) \leq 3d\lfloor n/4 \rfloor^2 + cn^2$$

$$T(n) \leq 3d(n/4)^2 + cn^2 \therefore T(n) = \frac{3}{16}dn^2 + cn^2 \therefore T(n) \leq dn^2.$$

Onde $T(n) \leq dn^2$ é satisfeito com $d \geq (16/13)c$.

O método por substituição é relativamente mais simples e direto do que o método por expansão da árvore de recorrência, resultando em provas bastante suscintas das equações de recorrência. Entretanto, esbarra na dificuldade de se fazer uma boa estimativa inicial da solução. A árvore de recorrência pode ser uma ferramenta para se fazer estas estimativas, provando-se a corretude da solução por indução.

24.3.3 Método de resolução pelo teorema geral

O método de resolução pelo teorema geral é uma espécie de “receita de bolo” para resolução de equações de recorrência do tipo $T(n) = aT(n/b) + f(n)$, onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva. O problema deve ser enquadrado em um dentre três possíveis casos, feito isso, o teorema geral provê uma solução trivial. Vejamos então o teorema:

Teorema Geral

Sejam as constantes a e b onde $a \geq 1$ e $b > 1$. Seja $f(n)$ uma função e seja $T(n)$ definida no domínio dos inteiros não negativos pela equação de recorrência $T(n) = aT(n/b) + f(n)$, onde n/b pode ser

interpretado tanto por $\lfloor n/b \rfloor$ quanto por $\lceil n/b \rceil$. Então $T(n)$ pode ser limitada assintoticamente da seguinte forma:

1. se $f(n) = O(n^{\log_b a - \varepsilon})$ para uma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;
2. se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;
3. se $f(n) = \Omega(n^{\log_b a + \varepsilon})$, para uma constante $\varepsilon > 0$ e se $af(n/b) \leq cf(n)$ para uma constante $c < 1$ e para todos n suficientemente grandes, então $T(n) = \Theta(f(n))$.

No três casos enumerados no teorema acima, $f(n)$ é comparada com a função $n^{\log_b a}$. A solução da recorrência é dada pela maior das duas funções. No caso 1 a função $n^{\log_b a}$ é maior que $f(n)$. No caso 2 são iguais, e no caso 3 a função $f(n)$ é maior. No caso 2, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$. Outro aspecto importante é que no primeiro caso não basta $f(n)$ ser menor que $n^{\log_b a}$, ela tem que ser polinomialmente menor. Ou seja, $f(n)$ precisa ser assintoticamente menor que $n^{\log_b a}$ em um fator n^ε . Da mesma forma, no caso 3, $f(n)$ precisa ser polinomialmente maior que $n^{\log_b a}$ para satisfazer a condição $af(n/b) \leq cf(n)$.

O teorema geral não cobre todos os possíveis casos para $f(n)$. Por exemplo, quando $f(n)$ é menor que $n^{\log_b a}$, mas não é polinomialmente menor, existe uma lacuna no caso 1 e o teorema não pode ser empregado. Situações similares podem ocorrer nos casos 2 e 3.

Para exemplificar o emprego do teorema, tomemos a equação de recorrência $T(n) = 9T(n/3) + n$. Para esta equação, temos $a = 9$, $b = 3$ e $f(n) = n$. Então $n^{\log_b a} = n^{\log_3 9} = n^2 = \Theta(n^2)$. Visto que, para a constante $\varepsilon = 1$, temos $f(n) = O(n^{\log_3 9 - \varepsilon}) = O(n^{2-\varepsilon}) = O(n^{2-1}) = O(n)$. Portanto, aplicando o caso 1 do teorema geral, concluímos que a solução da equação de recorrência é $T(n) = \Theta(n^2)$.

Análise de competitividade

Devido às suas características, o modelo computacional RAM não é adequado para algoritmos *online*, e a análise de complexidade vista no presente documento não pode ser aplicada neste caso. O desempenho de tais algoritmos deve ser analisado através da **análise de competitividade** [2]. A idéia de competitividade consiste em comparar a saída gerada por algoritmo *online* co a saída gerada por um algoritmo *offline*. O algoritmo é determinístico e onisciente no sentido de que possui o conhecimento completo de entrada podendo assim computar uma solução ótima. Quanto mais próximo da solução ótima o algoritmo *online* chegar, maior seu grau de competitividade [13] .

Formalmente, um algoritmo *online* A é apresentado como um conjunto de requisições $s = s_1, s_2, \dots, s_m$. As requisições s_t , com $t \in [1 \dots m]$, devem ser atendidas em sua ordem de ocorrência. Mais especificamente, ao atender a requisição s_t , o algoritmo A desconhece qualquer requisição t' , onde $t' > t$. Atender uma requisição sempre implica em custo. O objetivo do algoritmo é minimizar o custo total decorrente da seqüência completa de requisições [3].

Na análise de competitividade, o algoritmo A é comparado com um algoritmo *offline* ótimo. Dada uma seqüência de requisições s , seja $C_A(s)$ o custo relativo a A e $C_{OPT}(s)$ o custo relativo a um algoritmo *offline* ótimo OPT . O algoritmo A é chamado **c -competitivo** se existe uma constante c tal que $C_A(s) \leq c.C_{OPT}(s) + a$, para todas as seqüências de requisições s . O fator c é chamado de **taxa de competitividade** de A [4]. A análise de competitividade tem recebido forte interesse de pesquisa nos últimos 15 anos, e constitui uma importante técnica para avaliação de desempenho no pior caso para algoritmos *online* [7].

Este capítulo apresentou recursividade no contexto projeto e análise de algoritmos, mostrando a montagem de equações de recorrência utilizadas para análise de complexidade deste tipo de algoritmo através de três métodos diferentes. A seguir são propostos exercícios relativos a estes tópicos.

24.4 Exercícios propostos

1. Utilizando a figura 24.2 como exemplo, mostre a execução do algoritmo *MergeSort* para a seqüência $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.
2. Verifique se as afirmações a seguir são verdadeiras ou falsas. Justifique sua resposta.
 - a) Nenhum algoritmo baseado no princípio de divisão e conquista tem tempo de execução de ordem exponencial. Isto ocorre porque estes algoritmos são recursivos e recursividade evita a explosão combinatória.
 - b) Suponha que em determinado algoritmo do tipo divisão e conquista cada etapa de divisão substitui o problema por 16 sub-problemas, cada um com 25% do tamanho do problema corrente. Além disso, o esforço para obter a solução do problema corrente, baseado nas soluções dos 16 sub-problemas menores é constante, independente do tamanho do problema. Neste caso, podemos dizer que a complexidade do algoritmo é $O(n^3)$.
 - c) Nenhum algoritmo baseado no princípio de divisão e conquista tem tempo de execução de ordem logarítmica. Isto ocorre porque estes algoritmos são recursivos e recursividade implica sempre num esforço pelo menos linear para a construção das chamadas recursivas.
3. Utilize a árvore de recorrência para determinar um limite superior assintótico para a recorrência $T(n) = 3T(n/2) + n$. Utilize o método de substituição para verificar a resposta.
4. Idem para a recorrência $T(n) = 4T(n/2) + cn$, onde c é uma constante.
5. Utilize o método de substituição para provar que $T(n) = \Theta(n \log n)$ é uma solução correta para a equação de recorrência $T(n) = 2T(n/2) + cn$, do algoritmo *MergeSort*.
6. Utilize o teorema geral para encontrar limites assintóticos Θ para as seguintes equações de recorrência:
 - a) $T(n) = T(2n/3) + 1$;
 - b) $T(n) = 3T(n/4) + n \log n$;
 - c) $T(n) = 4T(n/2) + n$;
 - d) $T(n) = 4T(n/2) + n^2$;
 - e) $T(n) = 4T(n/2) + n^3$.
7. O algoritmo de busca binária em vetores ordenados possui a seguinte equação de recorrência: $T(n) = T(n/2) + \Theta(1)$. Aplique o teorema geral para provar que a solução desta recorrência é $T(n) = \Theta(\log n)$.
8. Escreva um algoritmo recursivo para o problema de **busca binária**. Construa sua equação de recorrência e prove a complexidade do algoritmo.

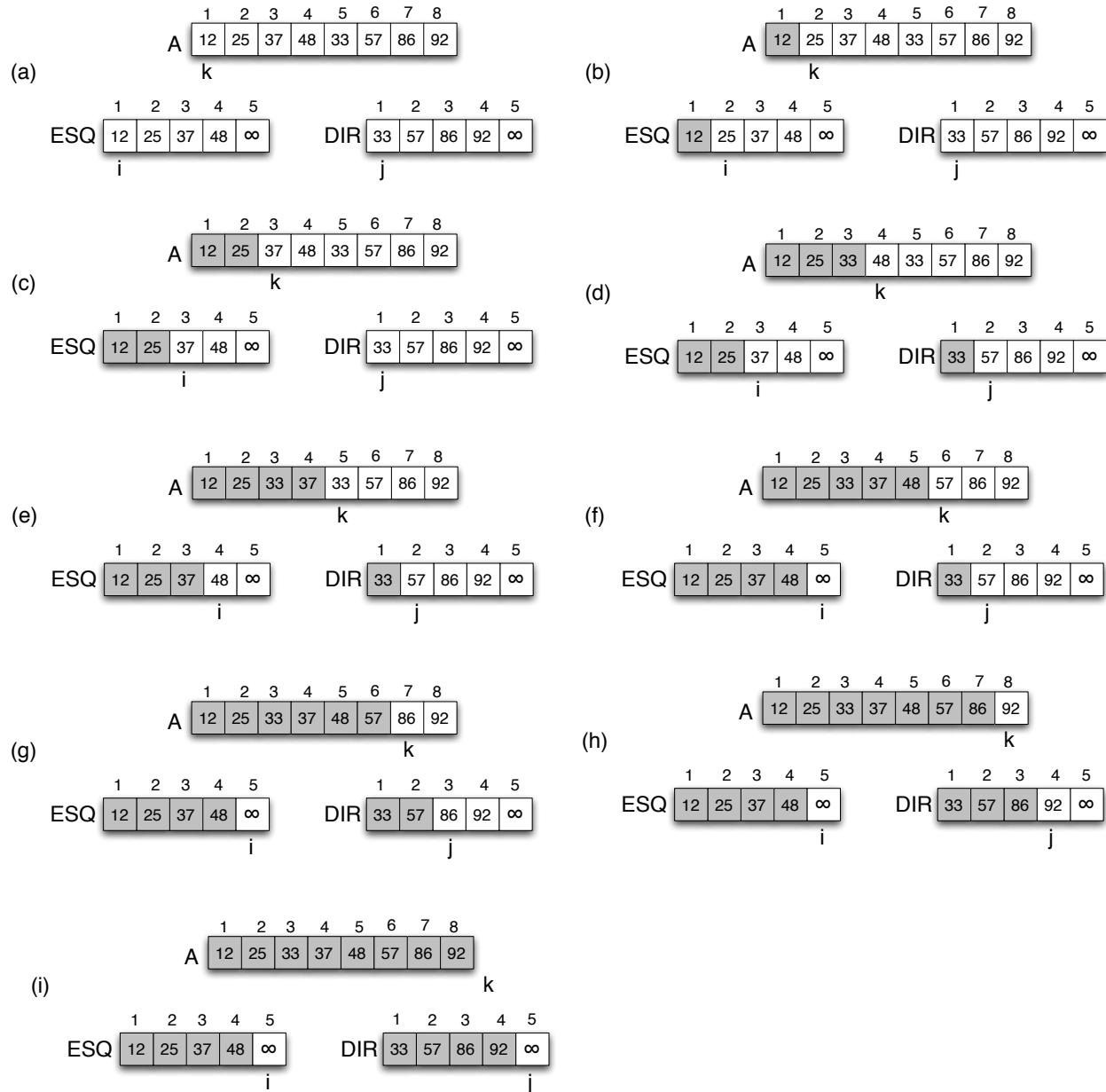


Figura 24.3: Exemplo de execução do procedimento Merge

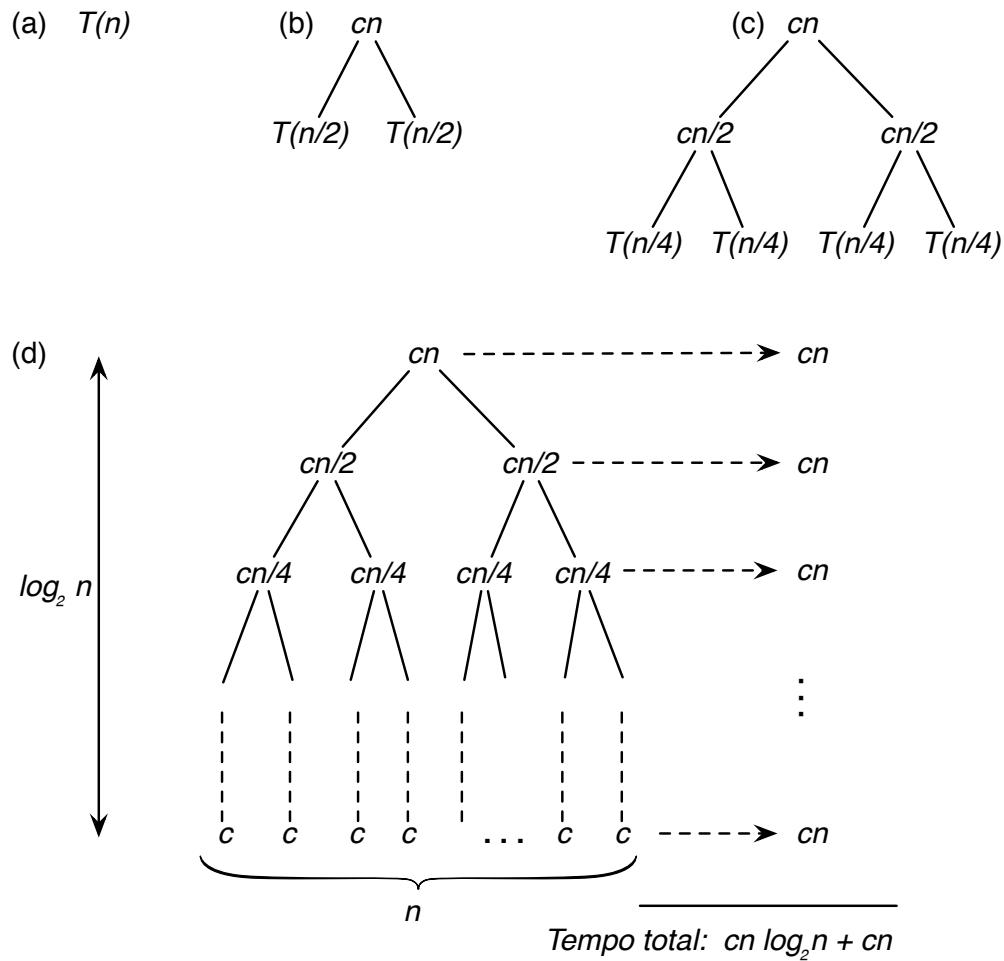
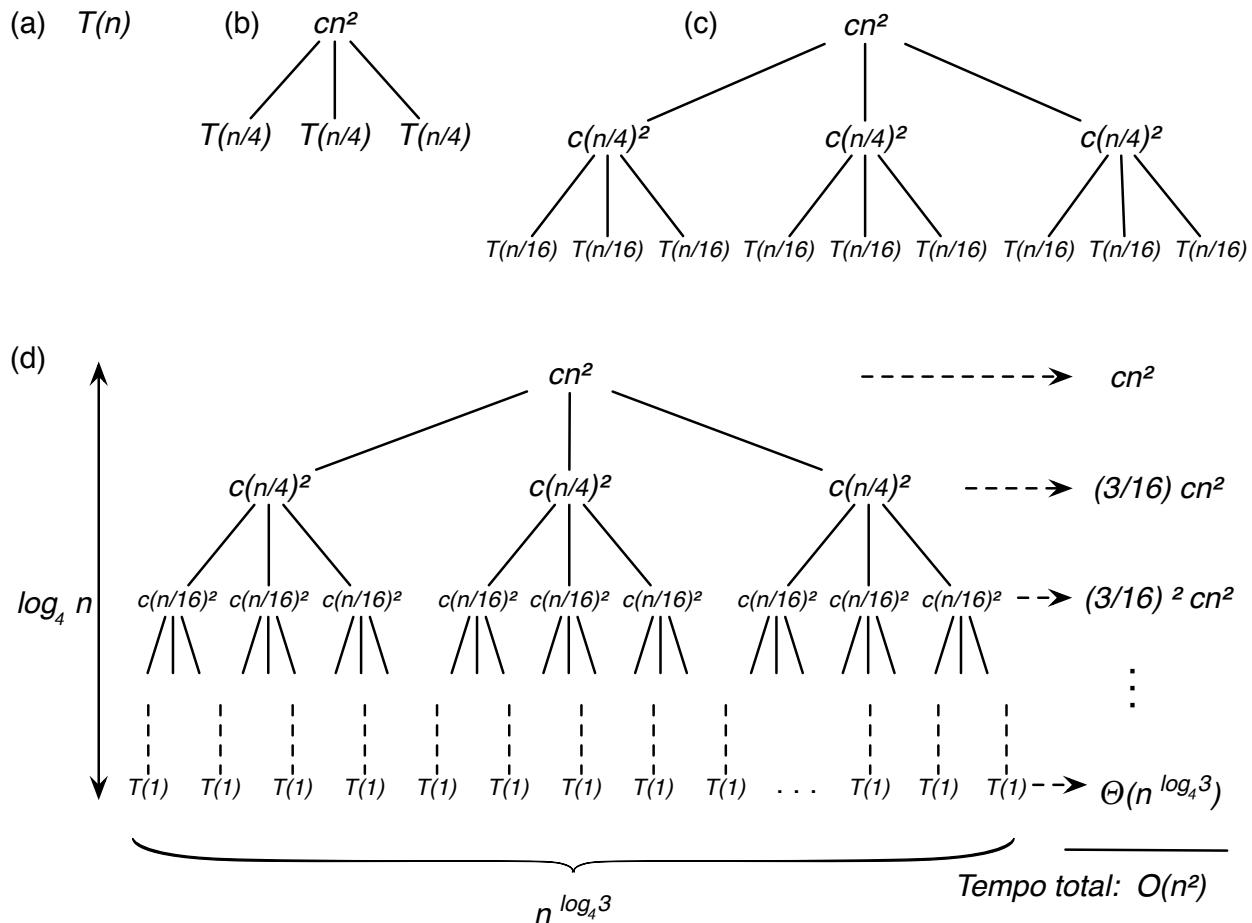


Figura 24.4: Árvore de recorrência para $T(n) = 2T(n/2) + cn$

Figura 24.5: Árvore de recorrência para $T(n) = 3T(n/4) + cn^2$

Índice Remissivo

computadores, 3

Referências Bibliográficas

- [1] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of computer science*. Computer Science Press, Inc., New York, NY, USA, 1992.
- [2] Miklos Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 35th Annual Symp. on Foundations of Computer Science*, pages 401–411, 2003.
- [3] Susanne Albers. Competitive online algorithms. *Optima: Mathematical Programming Society Newsletter*, 54:1–7, 1996.
- [4] Suzanne Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1–2):3–26, July 2003.
- [5] Yossi Azar. On-line load balancing. In *Theoretical Computer Science*, pages 218–225. Springer, 1992.
- [6] Sanjoy K. Baruah and Mary Ellen Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Transactions on Computers*, 47:1027–1033, 1996.
- [7] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge: Cambridge University Press. xviii, 414 p., 1998.
- [8] Facultad Ciencias, Exactas Naturales, Informe T, Leen Stougie, Esteban Feuerstein, Esteban Feuerstein, Marcelo Mydlarsz, and Marcelo Mydlarsz. On-line multi-threaded scheduling, 1999.
- [9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
- [10] Xiaotie Deng, Tiko Kameda, and Christos Papadimitriou. How to learn an unknown environment. i: the rectilinear case. *J. ACM*, 45(2):215–245, 1998.
- [11] Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. *ACM Trans. Algorithms*, 2(3):380–402, 2006.
- [12] Bruno Feijó, Paulo César Rodacki Gomes, Joao Bento, Sérgio Scheer, and Renato Cerqueira. Distributed agents supporting event-driven design processes. In John S. Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design 98*, chapter 10. Kluwer Academic Publishers, 1998.
- [13] Bruno Feijó, Paulo César Rodacki Gomes, Sérgio Scheer, and João Bento. Online algorithms supporting emergence in distributed cad systems. *Advances in Engineering Software*, 32(10), 2001.
- [14] Sandor Fekete, Rolf Klein, and Andreas Nuechter. Searching with an autonomous robot. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 449–450, New York, NY, USA, 2004. ACM.
- [15] Peter Fenwick. Burrows-wheeler compression: Principles and reflections. *Theor. Comput. Sci.*, 387(3):200–219, 2007.
- [16] Juris Hartmanis. On computational complexity and the nature of computer science. *ACM Comput. Surv.*, 27(1):7–16, 1995.
- [17] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. W. H. Freeman &

- Co., New York, NY, USA, 1978.
- [18] Tracy Kimbrel. Online paging and file caching with expiration times. *Theor. Comput. Sci.*, 268(1):119–131, 2001.
 - [19] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, 1998.
 - [20] Michael C. Loui. Computational complexity theory. *ACM Comput. Surv.*, 28(1):47–49, 1996.
 - [21] George S. Lueker. Some techniques for solving recurrences. *ACM Comput. Surv.*, 12(4):419–436, 1980.
 - [22] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.
 - [23] Edward M. Reingold. Basic techniques for design and analysis of algorithms. *ACM Comput. Surv.*, 28(1):19–21, 1996.
 - [24] Steven S. Seiden. Randomized online multi-threaded paging. *Nordic J. of Computing*, 6(2):148–161, 1999.
 - [25] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2008.
 - [26] Cyrill Stachniss, Óscar Martínez Mozos, and Wolfram Burgard. Efficient exploration of unknown indoor environments using a team of mobile robots. *Annals of Mathematics and Artificial Intelligence*, 52(2-4):205–227, 2008.
 - [27] Hsien-Wen Tseng and Chin-Chen Chang. Error resilient locally adaptive data compression. *J. Syst. Softw.*, 79(8):1156–1160, 2006.
 - [28] Toscani L. V. and Veloso P. *Complexidade de Algoritmos*. Série Livros Didáticos - SBC. Editora Sagra Luzzato, 2002.
 - [29] Bruce Weide. A survey of analysis techniques for discrete algorithms. *ACM Comput. Surv.*, 9(4):291–313, 1977.
 - [30] Yunfei Yin. A proximate dynamics model for data mining. *Expert Syst. Appl.*, 36(6):9819–9833, 2009.