Pesquisa e Ordenação

Onde são mostrados algoritmos para busca e ordenação de elementos em vetores.

Pré-requisito(s):	Objetivos (ao final você deverá ser capaz de):
• Saber declarar e utilizar vetores em Java • Saber definir e utilizar métodos em Java com passagem de parâmetros	• Conhecer algoritmos de pesquisa de elementos em vetores • Escrever programas Java que realizem busca de elementos em vetores • Conhecer algoritmos clássicos para ordenação de vetores • Utilizar métodos de ordenação em programas Java

Realizar uma busca ou pesquisa de dados em uma grande coleção faz cada vez mais parte de nosso dia a dia. Fazemos uso desse mecanismo quando utilizamos um site de busca para procurar por algo na Internet, por exemplo. A definição de uma palavra no dicionário ou ainda pesquisar um número de telefone ou endereço em uma lista telefônica são outros exemplos de atividades do cotidiano. Todos os exemplos citados envolvem pesquisa de valores em uma grande coleção de dados. Essa pesquisa pode ser facilitada ou não a depender da forma como os dados estão previamente organizados. Imagine, por exemplo, uma lista telefônica que não estivesse ordenada pelo sobrenome das pessoas; na verdade, que não houvesse critério algum de ordenação. Como procurar pelo número de telefone de alguém? Este capítulo apresenta dois algoritmos diferentes de pesquisa: pesquisa seqüencial e pesquisa binária. O capítulo também apresenta três algoritmos clássicos de ordenação de dados em uma seqüência. Tanto uma lista telefônica quanto um dicionário de palavras são exemplos de seqüência ordenada de dados que facilita a busca de algum valor.

7.1 Pesquisa de dados

Vimos no capítulo 5, como definir e utilizar estruturas de dados compostas para o armazenamento de coleções de dados e informações. Em particular, vimos como uma estrutura do tipo vetor contribui para uma melhor organização dos dados na memória e facilita a elaboração de algoritmos reduzindo sobremaneira a quantidade de variáveis diferentes necessárias.

Uma vez armazenados, entretanto, é natural que esses dados sejam recuperados em momentos oportunos dentro da solução algorítmica de um determinado problema. Inúmeros são os exemplos de operações que fazem uso de um método de recuperação desse tipo: recuperação de dados de transações bancárias de um cliente através de um número de conta, recuperação de dados de alunos através de seu número de matrícula, recuperação de informações sobre um produto em estoque através de seu código de barras, entre outros.

A recuperação desses dados se dá através de um processo genérico que independe do problema em questão. Esse processo genérico é comumente chamado de **algoritmo de pesquisa** e deve ser projetado de forma a garantir a confiabilidade e eficiência exigidas pela importância das aplicações existentes.

Existem dois tipos conhecidos de algoritmos de pesquisa: o algoritmo de **pesquisa seqüencial** e o algoritmo de **pesquisa binária**.

As considerações que podem ser feitas são do tipo: o vetor está ou não ordenado; o elemento ocorre uma única vez (pesquisa única) ou repetidas vezes no vetor (pesquisa múltipla).

7.1.1 Pesquisa seqüencial

O método mais simples de se verificar a presença de um determinado elemento numa seqüência, é percorrê-la a partir do início, efetuando comparações, até que o elemento seja encontrado ou o fim da seqüência seja alcançado.

Considere um vetor de tamanho `n` e um elemento com valor `X` que se deseja pesquisar nesse vetor. O fluxograma da figura 7.1 ilustra os passos do algoritmo de pesquisa seqüencial para verificar se o elemento se encontra na estrutura ou não. Note que a pesquisa começa com o índice `i` igual a `0` (zero) e o valor `X` é comparado com o valor armazenado nessa posição do vetor (`vetor[i] = X?`). Caso esse valor seja igual, o índice `i` do vetor é retornado; caso contrário, o índice é incrementado e uma nova comparação é realizada. O processo se repete enquanto o valor de `i` é menor que o tamanho do vetor (`i<n?`). Caso o valor de `X` nunca seja encontrado, uma mensagem de falha é retornada.

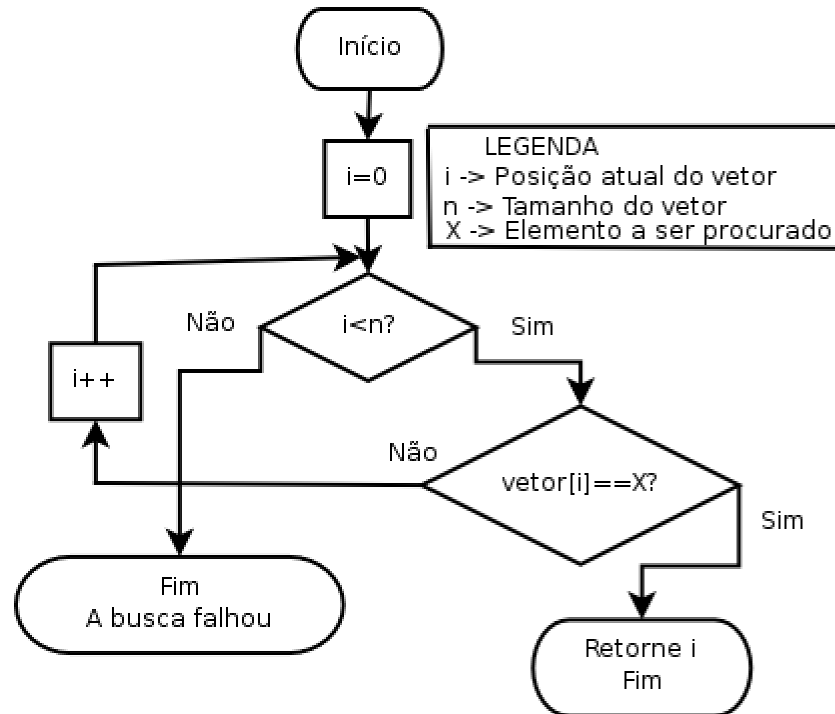


Figura 7.1 – Fluxograma para pesquisa seqüencial de dados em vetor.

O problema ilustrado na figura 5.12 do capítulo 5 é um exemplo de busca seqüencial. O problema consiste em recuperar de um vetor de mercadorias, os dados da mercadoria com um determinado código. Esse código é utilizado para realizar a pesquisa. O algoritmo está reproduzido na figura 7.2 a seguir.

```
1.  Algoritmo Mercadorias;
2.  Tipos
3.     Tmercadoria = registro
4.                    COD : string[6];
5.                   NOME : string[15];
6.                   PRECO: real;
7.                    EST : integer;
8.                 fimregistro;
9.     ConjuntoMercadoria = vetor [50] de registro;
10. Variáveis
11.    TAB : ConjuntoMercadoria;
12.      I : integer;
13.      K : literal;
14.   RESP : literal;
15. Begin
16.   //Leitura dos dados das 50 mercadorias
17.   para I <- 0 até 49 faça
18.      leia(TAB[I].COD);
19.      leia(TAB[I].NOME);
20.      leia(TAB[I].PRECO);
21.      leia(TAB[I].EST);
22.   fimpara;
23.   repita
```

```
24.     //leitura da chave de pesquisa
25.     escreva("Entre com o código desejado: ");
26.     leia(K);
27.     //testa em cada registro se o código é igual a chave
28. pesquisada
29.     para I <- 0 até 49 faça
30.        se (K = TAB[I].COD) então
31.          escreva(TAB[I].NOME, TAB[I].PRECO, TAB[I].EST);
32.        //verifica se o usuário deseja pesquisar outro código}
33.        escreva("Outra mercadoria(S/N)?");
34.        leia(RESP);
35.     fimpara;
36.   até (RESP = "N");
37. Fim.
```

Figura 7.2 – Pesquisa seqüencial em vetores.

No exemplo, o código da mercadoria a ser pesquisada é fornecido pelo usuário na linha 26. Esse código armazenado na variável `K` é comparado com o código de todas as mercadorias existentes no vetor, a começar pela primeira mercadoria (índice `I = 0`, linha 29). Caso o código seja encontrado (linha 30), as informações presentes no registro da mercadoria em questão são impressas (linha 31).

7.1.2 Pesquisa Binária

O algoritmo de pesquisa seqüencial é bastante simples e é razoavelmente eficiente para seqüências com poucos elementos. Entretanto, para seqüências de tamanho considerável, que ocorrem na maioria das aplicações existentes, a utilização do algoritmo se torna inviável. Uma estratégia interessante e eficiente é utilizada no método de **pesquisa binária**. É em média um algoritmo mais rápido que o anterior, porém exige que o vetor esteja previamente ordenado.

O algoritmo divide a seqüência ao meio e testa se o elemento a ser procurado está acima ou abaixo da linha de divisão. Se estiver acima, por exemplo, toda a metade abaixo é desprezada. Em seguida, se o elemento não foi encontrado, a seqüência é novamente dividida ao meio e o procedimento de pesquisa se repete até que o elemento seja encontrado ou não haja mais divisões possíveis.

Considere um vetor ordenado `k` de tamanho `N` e um elemento cuja `chave` se deseja pesquisar nesse vetor [n*ota: normalmente, chamamos de "chave" um valor que identifica unicamente um elemento em um conjunto de dados, por exemplo, o CPF de uma pessoa, ou o número de matrícula de um aluno. Através dessa chave, podemos recuperar as informações daquela pessoa em questão*]. O fluxograma da figura 7.3 ilustra os passos do algoritmo de pesquisa binária para verificar se a chave do elemento se encontra na estrutura ou não.

O algoritmo de pesquisa considera duas variáveis auxiliares, `inf` e `sup`, para

armazenar o menor e o maior índices, respectivamente, a se considerar na pesquisa. Enquanto o valor de `inf` for menor ou igual ao de `sup`, a pesquisa é realizada capturando-se o índice do meio (`meio = (inf+sup)/2`) e testando se o valor armazenado nesse índice é igual à chave (`chave = k[meio]?`). Caso os valores sejam iguais, a busca é finalizada com sucesso. Caso contrário, um teste é realizado para decidir que metade do vetor deve ser considerada a partir de então: se o valor da `chave` for menor que o valor armazenado no meio, então o limite de pesquisa no vetor passa a ser o índice anterior ao do meio (`sup = meio-1`); se o valor da chave for maior que o o valor armazenado no meio, então a pesquisa passa a ser iniciada a partir do índice posterior ao do meio (`inf = meio+1`).

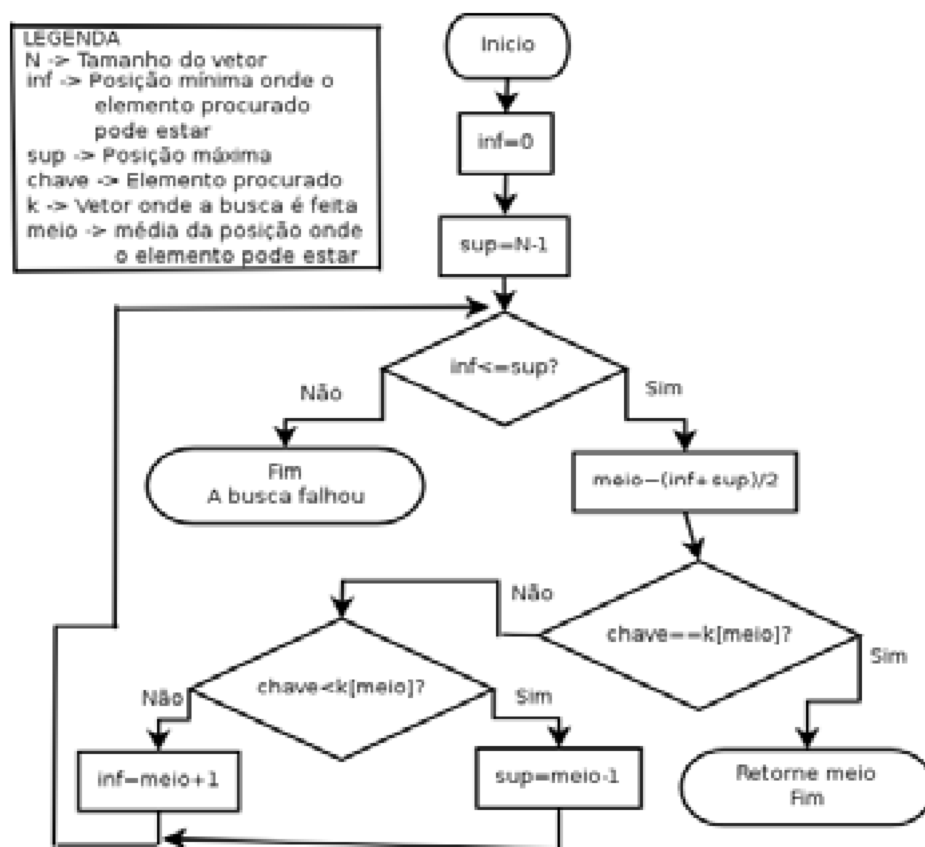


Figura 7.3 – Fluxograma para pesquisa binária de dados em um vetor.

A figura 7.4 ilustra a aplicação do algoritmo de pesquisa binária para pesquisa de uma chave de valor 15 no vetor ordenado considerado. As posições sombreadas representam o índice correspondente à variável `meio` em cada um dos passos do algoritmo. Para esse exemplo, inicialmente o valor de `inf` seria 0 e `sup` seria 12. No passo 2, o valor de `inf` seria 0 e o de `sup` seria 5. No passo 3, `inf = 3` e `sup = 5`. E, finalmente, no passo 4, `inf = sup = 3`. Nesse exemplo, o valor da chave é encontrado no vetor em quatro passos.

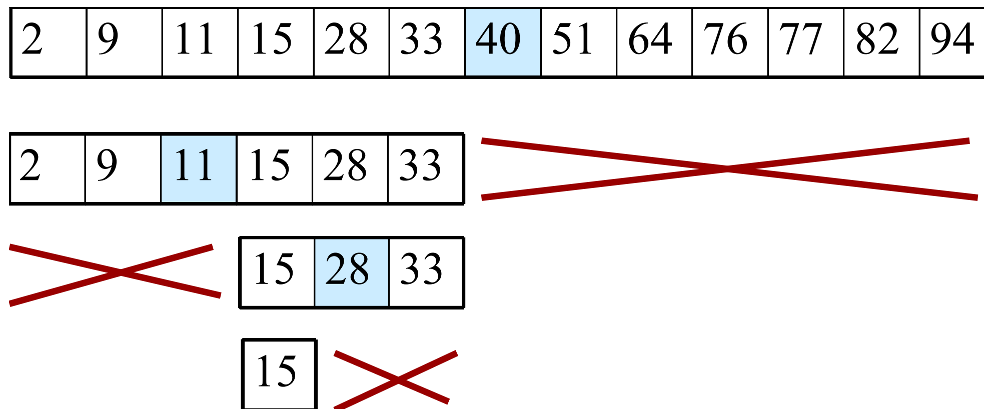


Figura 7.4 – Pesquisa binária da chave de valor 15 em um vetor ordenado de 13 elementos.

Considere como exemplo, o problema da busca de alunos de um curso (figura 7.5). Cada aluno é representado por um registro contendo nome, número de matrícula, sua média na disciplina e sua freqüência. Considere ainda que os dados dos alunos, ao serem lidos, são automaticamente armazenados em registros numa lista ordenada pelo número de matrícula do aluno. O problema consiste em, dado um número de matrícula, imprimir as informações referentes.

```
1.  Algoritmo PesquisaAlunos;
2.  Tipos
3.    TipoAluno = registro
4.                   Nome: literal;
5.                   Matricula: literal;
6.                   Media: real;
7.                   Frequencia: inteiro;
8.                fimregistro;
9.  Variaveis
10.    Aluno: TipoAluno;
11.    valunos: vetor[50] de Aluno;
12.    nalunos, inf, sup, meio: inteiro;
13. Inicio
14.    para I <- 0 até 49 faça
15.       leia (valunos[I].Nome);
16.       leia (valunos[I].Matricula);
17.       leia (valunos[I].Media);
18.       leia (valunos[I].Frequencia);
19.    fimpara;
20.    escreva(“Matricula do aluno que deseja consultar: “);
21.    leia(chave);
22.    inf <- 0;
23.    sup <- 49;
24.    enquanto inf <= sup faça
25.       meio <- (inf + sup)/2;
26.       se (chave = valunos[meio].Matricula) entao
27.         inicio
28.           escreva(“Nome: “, valunos[meio].Nome);
29.           escreva(“Nome: “, valunos[meio].Media);
30.           escreva(“Nome: “, valunos[meio].Frequencia);
```

```
31.             sair;
32.           fim
33.         senão
34.           se (chave < valunos[meio].Matricula) então
35.              sup <- meio-1;
36.           senão
37.              inf <- meio+1;
38.           fimse
39.        fimse
40.    fimenquanto;
41.    escreva("Total de Reprovados: ", Reprovados);
42. Fim.
43.
44.
```

Figura 7.5 – Pesquisa binária da chave de valor 15 em um vetor ordenado de 13 elementos.

A chave de busca, que é o número de matrícula do aluno, é lido na linha 21. As variáveis `inf` e `sup` assumem o valor do menor índice do vetor e do maior índice, respectivamente (linhas 22 e 23). A partir daí o processo repetitivo descrito pelo fluxograma é representado pelo laço da da linha 24: enquanto `inf` e `sup` não se cruzarem, a chave lida é comparada com o elemento central do pedaço de vetor limitado por `inf` e `sup` (linha 26). Caso o valor da chave corresponda ao do elemento em questão, o restante dos dados do aluno são escritos, caso contrário, o valor das variáveis `inf` e `sup` serão atualizados de acordo com o valor da chave procurada em relação a esse elemento central (linhas 36 a 39).

7.2 Ordenação de dados

Ordenar uma determinada seqüência significa rearranjar os elementos dessa seqüência em ordem crescente ou decrescente. O objetivo principal da ordenação é facilitar a recuperação posterior de dados da seqüência ordenada. Imagine, por exemplo, como seria complicado utilizar uma lista telefônica se os nomes das pessoas não estivessem listados em ordem alfabética.

Os problemas de ordenação são comuns tanto em aplicações comerciais quanto científicas. Entretanto, raros são os problemas que se resumem à pura ordenação de seqüências de elementos. Normalmente, os problemas de ordenação são inseridos em problemas de pesquisa, intercalação e atualização. Isto torna ainda mais importante o projeto e a construção de algoritmos eficientes e confiáveis para tratar o problema.

Assim como acontece nos problemas de pesquisa de dados, a ordenação de dados numa seqüência também considera a presença de uma chave, que é utilizada para controlar o processo de ordenação. Os demais dados de um eventual registro

não influenciam no processo.

A ordenação de dados em um vetor também é um processo genérico que independe do problema em particular. Existem vários algoritmos de ordenação de dados armazenados em memória interna. Esse algoritmos se diferenciam principalmente em relação ao tempo de processamento que gastam para realizar a ordenação. A quantidade de memória extra utilizada pelo algoritmo também é um aspecto muito importante; quantos menos memória utilizada, melhor a qualidade do algoritmo. Algumas características do vetor a ser ordenado também influenciam muito na eficiência de um ou outro algoritmo. Por exemplo, alguns algoritmos são melhores que outros caso o vetor já esteja parcialmente ordenado. Alguns algoritmos são mais eficientes e/ou econômicos para vetores com grandes quantidades de dados, enquanto outros são mais apropriados caso o tamanho do vetor não seja tão grande. De qualquer forma, qualquer tipo de avaliação e/ou comparação da qualidade dos algoritmos de ordenação aqui apresentados, seja ela empírica ou medidas em termos formais de complexidade, está fora do escopo deste curso.

Nas seções seguintes, iremos utilizar diretamente a sintaxe de Java para estudar três algoritmos de ordenação básicos: Ordenação por Seleção, Método da Bolha, e Ordenação por Inserção. No capítulo 8, veremos outros métodos de ordenação mais elaborados.

7.2.1 Ordenação por Seleção

Este algoritmo de ordenação é um dos mais simples e intuitivos dentre os existentes. A idéia básica é buscar o menor (ou o maior, dependendo da ordem desejada) elemento da seqüência considerada e colocá-lo no início da seqüência. Uma vez que o menor elemento se encontra na primeira posição, o algoritmo repete o processo a partir da segunda posição, e assim sucessivamente.

A lógica do algoritmo consiste em se varrer a seqüência comparando todos os elementos com o primeiro. Caso o primeiro elemento esteja desordenado em relação ao elemento que está sendo comparado com ele no momento, é feita a troca. Ao se chegar ao final da seqüência, teremos o menor valor (ou o maior) na primeira posição da seqüência.

O algoritmo é ilustrado para a seqüência de seis chaves presentes em um vetor apresentada na figura 7.6. No exemplo, o objetivo é ordenar o vetor em ordem crescente dos valores das chaves. As chaves em negrito representam chaves que sofreram troca de posição entre si. As posições sombreadas já estão ordenadas.

Chaves iniciais:	*28*	*33*	*9*	*11*	*15*	*2*

i = 0	**2**	*33*	*9*	*11*	*15*	**28**
i = 1	*2*	**9**	**33**	*11*	*15*	*28*
i = 2	*2*	*9*	**11**	**33**	*15*	*28*
i = 3	*2*	*9*	*11*	**15**	**33**	*28*
i = 4	*2*	*9*	*11*	*15*	**28**	**33**
	0	1	2	3	4	5

Figura 7.6 – Exemplo de aplicação da ordenação por seleção.

O pseudocódigo do algoritmo de ordenação por seleção está ilustrado na figura 7.7. Ele está implementado como um método da classe `Ordenacao`, que agrupa todos os métodos de ordenação que trabalharemos (figura 7.8). Os dados armazenados no vetor a ser ordenado são do tipo `Elemento`, que representa um registro de dados genérico (figura 7.9). Esse registro contem apenas um campo do tipo inteiro para representar a chave utilizada para ordenação. Nada impede entretanto que outros campos sejam adicionados, de acordo com a necessidade do problema em particular.

```
1.  public static void selecao (Elemento v[], int n) {
2.      for (int i = 0; i < n; i++) {
3.        int min = i;
4.        for (int j = i + 1; j < n; j++)
5.          if (v[j].chave < v[min].chave)
6.            min = j;
7.        Elemento x = v[min];
8.        v[min] = v[i];
9.        v[i] = x;
10.     }
11.   }
```

Figura 7.7 – Algoritmo de ordenação por seleção em Java. Como parâmetro são passados o vetor a ser ordenado e sua dimensão.

```
1.  public class Ordenacao {
2.    public static void selecao (Elemento v[], int n)
3.    public static void insercao (Elemento v[], int n)
4.    public static void bolha (Elemento v[], int n)
5.  }
```

Figura 7.8 – Classe `Ordenacao` agrupa os três métodos de ordenação a serem trabalhados.

```
1.  class Elemento {
2.     int chave;
3.     //outros campos
4.  }
```

Figura 7.9 – Classe `Elemento` representa o registro do dado que será procurado através de sua chave de valor inteiro.

7.2.2 Ordenação pelo Método da Bolha

O método da bolha realiza comparações e trocas entre elementos consecutivos da seqüência, a fim de "empurrar" os maiores (em caso de ordenação ascendente) elementos para o final da mesma (borbulhamento). Caso o elemento da posição seguinte seja maior que o da posição atual, os elementos são trocados de posição. Várias iterações são efetuadas e, para cada seqüência considerada, a aplicação da estratégia gera uma nova seqüência pela eliminação do maior elemento da seqüência original.

Uma variável lógica de controle é utilizada para registrar a ocorrência ou não de troca entre elementos da seqüência. Quando nenhuma troca é efetuada, significa que a seqüência já estava ordenada e o algoritmo encerra a execução.

O algoritmo é ilustrado para a seqüência de seis chaves presentes em um vetor apresentada na figura 7.10. As chaves em negrito representam chaves que sofreram troca de posição entre si. As posições sombreadas já estão ordenadas.

limites:	*28*	*33*	*9*	*11*	*15*	*2*
`limite = 6`	*28*	**9**	**33**	*11*	*15*	*2*
`limite = 6`	*28*	*9*	**11**	**33**	*15*	*2*
`limite = 6`	*28*	*9*	*11*	**15**	**33**	*2*
`limite = 6`	*28*	*9*	*11*	*15*	**2**	**33**
`limite = 5`	**9**	**28**	*11*	*15*	*2*	*33*
`limite = 5`	*9*	**11**	**28**	*15*	*2*	*33*
`limite = 5`	*9*	*11*	**15**	**28**	*2*	*33*
`limite = 5`	*9*	*11*	*15*	**2**	**28**	*33*
`limite = 4`	*9*	*11*	**2**	**15**	*28*	*33*
`limite = 3`	*9*	**2**	**11**	*15*	*28*	*33*
`limite = 2`	**2**	**9**	*11*	*15*	*28*	*33*
	0	1	2	3	4	5

Figura 7.10 – Exemplo de aplicação da ordenação pelo método da bolha.

O pseudocódigo do algoritmo de ordenação pelo método da bolha está ilustrado na figura 7.11.

```
1.  public static void bolha(Elemento v[], int n) {
2.      boolean trocou;
3.      int limite = n;
4.      do {
5.           trocou = false;
6.           for (int j = 0; j < limite-1; j++)
7.           if (v[j].chave > v[j+1].chave) {
8.               Elemento temp = v[j];
9.               v[j] = v[j+1];
10.              v[j+1] = temp;
11.              trocou = true;
12.          }
13.          limite = limite - 1;
14.      }
15.      while (!trocou);
16. }
```

Figura 7.11 – Algoritmo de ordenação pelo método da bolha em Java.

7.2.3 Ordenação por Inserção

O método de ordenação por inserção é o mais rápido entre os outros dois métodos considerados básicos e apresentados anteriormente. O algoritmo consiste em ordenar a seqüência desejada utilizando uma subseqüência ordenada localizada no início da mesma. A cada passo, o i-ésimo elemento da seqüência fonte é resgatado e inserido na subseqüência de destino, que se encontra inicialmente vazia. Por curiosidade, este é exatamente o método que a maioria das pessoas utiliza para ordenar as cartas de sua mão em um jogo de baralho como o pôquer, por exemplo.

O algoritmo é ilustrado para a mesma seqüência de seis chaves utilizadas anteriormente, conforme apresentado na figura 7.12. As chaves em negrito representam chaves que acabaram de ser inseridas na subseqüência de destino. A subseqüência de destino está sombreada.

Chaves iniciais:	*28*	*33*	*9*	*11*	*15*	*2*
i = 1	*28*	**33**	*9*	*11*	*15*	*2*
i = 2	**9**	*28*	*33*	*11*	*15*	*2*
i = 3	*9*	**11**	*28*	*33*	*15*	*2*
i = 4	*9*	*11*	**15**	*28*	*33*	*2*
i = 5	**2**	*9*	*11*	*15*	*28*	*33*
	0	1	2	3	4	5

Figura 7.12 – Exemplo de aplicação da ordenação por inserção em Java.

O pseudocódigo do algoritmo de ordenação por inserção está ilustrado na figura 7.13.

```
1.  public static void insercao (Elemento v[], int n) {
2.      for (int i = 1; i < n; i++) {
3.        Elemento temp = v[i];
4.        int j = i - 1;
5.        while ((j >= 0) && (v[j].chave > temp.chave)) {
6.          v[j + 1] = v[j];
7.          j--;
8.        }
9.        v[j + 1] = temp;
10.     }
11. }
```

Figura 7.13 – Algoritmo de ordenação por inserção em Java.

7.3 Estudo de caso: Mercadorias

Considere mais uma vez o problema do estoque de mercadorias ilustrado anteriormente. Suponha que uma grande empresa possui um estoque de mercadorias armazenadas e precisa de um programa que realize a leitura dos dados dessas mercadorias e armazene em uma estrutura de tal forma que seja possível realizar a busca dos dados referentes a uma determinada mercadoria de forma eficiente sempre que requisitado.

Poderíamos utilizar qualquer um dos dois algoritmos para pesquisa de dados, vistos anteriormente na seção 7.1: o algoritmo de pesquisa seqüencial e o algoritmo de pesquisa binária. Entretanto, eficiência é um requisito primordial para a empresa e, dessa forma, devemos optar pelo algoritmo de pesquisa binária, notadamente mais eficiente, sobretudo se considerarmos que uma grande empresa normalmente possui um igualmente grande estoque de mercadorias.

O problema é que, como visto, a pesquisa binária somente se aplica a vetores já ordenados. Oras, mas porque não utilizar então a pesquisa seqüencial, que não precisa de ordenação?

Esse é um questionamento relevante, mas observe que a ordenação das mercadorias na estrutura ocorrerá uma única vez, enquanto que a pesquisa por uma determinada mercadoria ocorrerá com freqüência, talvez várias vezes por dia, ou por hora.

Por esse motivo, logo após a leitura e armazenamento dos dados de todas as mercadorias em estoque, devemos proceder à ordenação dessas mercadorias. Para isso, podemos utilizar qualquer método de ordenação visto na seção 7.2.

A figura 7.14 traz um programa Java que realiza a leitura e cadastro das

mercadorias de uma empresa, ordena a estrutura pelo código da mercadoria e realiza a busca das informações de uma determinada mercadoria sempre que desejado. Por motivos didáticos, a solução aqui proposta oferece um menu para escolha do método de ordenação desejado, bem como a opção pelo método de pesquisa preferido.

Os códigos Java para pesquisa seqüencial e binária estão ilustrados na figura 7.15.

A classe `Elemento` apresentada na figura 7.9, foi expandida para considerar os demais dados das mercadorias (figura 7.16).

```
1.  public class Mercadorias {
2.
3.    public static void main(String[] args) {
4.
5.       //Trecho para cadastro das mercadorias em estoque
6.       System.out.print("Qtde de mercadorias em estoque: ");
7.       int total = System_in.readInt();
8.       Elemento[] estoque = new Elemento[total];
9.       for (int i=0; i<total; i++) {
10.           estoque[i] = new Elemento();
11.           System.out.print("CODIGO: ");
12.           estoque[i].chave = System_in.readInt();
13.           System.out.print("NOME: ");
14.           estoque[i].nome = System_in.readString();
15.           System.out.print("PRECO: ");
16.           estoque[i].preco = System_in.readFloat();
17.           System.out.print("QTDE: ");
18.           estoque[i].qtde = System_in.readInt();
19.           System.out.println();
20.       }
21.
22.       //Trecho para escolha do algoritmo de ordenação
23.       System.out.print("Ordenar as mercadorias (s/n)? ");
24.       String resp = System_in.readString();
25.       System.out.println();
26.       if (resp.equals("s")) {
27.           System.out.print("Algoritmo de ordenação desejado "+
28.                          "(1)seleção (2)bolha (3)inserção: ");
29.           int ord = System_in.readInt();
30.           switch (ord) {
31.           case 1: Ordenacao.selecao(estoque, total); break;
32.           case 2: Ordenacao.bolha(estoque, total); break;
33.           case 3: Ordenacao.insercao(estoque, total); break;
34.           }
35.       }
36.
37.       //Trecho para escolha do algoritmo de pesquisa
38.       System.out.println();
39.       System.out.print("Algoritmo de pesquisa desejado:
40.                       (1)Sequencial (2)Binária: ");
41.       int pesq = System_in.readInt();
42.       Elemento mercadoria;
43.       int indice = -1;
```

```
44.       System.out.print("Informe o código da mercadoria (0 para
45.                        finalizar): ");
46.       int chave = System_in.readInt();
47.       System.out.println();
48.       while (chave != 0) {
49.         switch (pesq) {
50.         case 1: indice = Pesquisa.sequencial(chave, estoque,
51                                                 total); break;
52.         case 2: indice = Pesquisa.binaria(chave, estoque,
53.                                                 total); break;
54.         }
55.         if (indice == -1)
56.           System.out.println("Mercadoria inexistente!");
57.         else {
58.           mercadoria = estoque[indice];
59.           System.out.println("Informações sobre a mercadoria
60.                               "+chave+":");
61.           System.out.println("NOME: "+mercadoria.nome);
62.           System.out.println("PRECO: "+mercadoria.preco);
63.           System.out.println("QTDE: "+mercadoria.qtde);
64.         }
65.         System.out.println();
66.         System.out.print("Informe o código da mercadoria (0
67.                          para finalizar): ");
68.         chave = System_in.readInt();
69.     }
70.   }
71. }
```

Figura 7.14 – Programa Java para leitura e cadastro de mercadorias, com opção de pesquisa e ordenação.

Note que a depender da escolha do usuário (linha 30), um dos três métodos de ordenação vistos será invocado com os parâmetros adequados (linhas 31, 32 e 33). Uma vez ordenado o vetor de mercadorias, o usuário é solicitado a escolher o método de pesquisa (linha 41) e a chave a ser pesquisada (linha 46).

```
1.  public class Pesquisa {
2.
3.    public static int sequencial(int chave,Elemento v[],int n){
4.      for (int i = 0; i < n; i++)
5.        if (v[i].chave == chave)
6.          return i;
7.      return -1;
8.    }
9.
10.   public static int binaria(int chave, Elemento v[], int n)
11.   {
12.     int esq = 0;
13.     int dir = n - 1;
14.     while ( esq <= dir ) {
15.          int meio = (dir + esq) / 2;
16.          if (chave == v[meio].chave)
17.             return meio;
```

```
18.            if ( chave < v[meio].chave )
19.               dir = meio - 1;
20.            else
21.               esq = meio + 1;
22.       }
23.       return -1;
24.    }
25.
26. }
```

Figura 7.15 – Implementação dos métodos Java para pesquisa seqüencial e binária.

Note que caso o valor da chave de pesquisa seja encontrado no vetor passado como argumento, o método retorna como resultado o valor do índice do vetor onde o elemento procurado se encontra (linha 6 e linha 17). Caso, entretanto, o elemento desejado não seja encontrado, o método retorna o valor -1 (linha 7 e linha 23).

```
1.  class Elemento {
2.     int chave;
3.     String nome;
4.     float preco;
5.     int qtde;
6.  }
```

Figura 7.16 – Classe `Elemento` expandida para conter outros dados das mercadorias.

Resumo

- Comumente fazemos uso de alguma técnica para procurar um dado em uma lista de dados semelhantes. Exemplos são busca de telefones pelo sobrenome numa lista telefônica ou busca pelo significado de uma palavra em um dicionário.
- Existem duas formas clássicas de se realizar a pesquisa de um elemento em um conjunto de dados: pesquisa seqüencial de valores ou pesquisa binária.
- A pesquisa seqüencial procura item por item, começando pelo primeiro da lista em questão até que o item desejado seja encontrado ou a lista se encerre.
- A pesquisa binária divide a lista inicial em duas de acordo com o valor do item procurado: caso seja maior que o elemento central, repete-se o processo considerando a segunda metade da lista apenas, caso seja menor, considera-se a primeira metade apenas.
- A pesquisa binária necessita que a lista de elementos esteja ordenada de acordo com algum critério. A pesquisa seqüencial não necessita de ordenação, entretanto é mais ineficiente.

- Para se ordenar uma lista (vetor) de elementos, existem pelo menos três algoritmos clássicos: por seleção, método da bolha, e por inserção.
- O algoritmo de ordenação por seleção procura pelo menor elemento da lista e o troca de posição com o primeiro. O processo é repetido considerando gora o segundo elemento até que todos estejam ordenados.
- O algoritmo de ordenação pelo método da bolha compara dois elementos adjacentes no vetor e o de maior valor é sempre empurrado para a direita através da troca até que atinja a última posição do vetor. O processo recomeça e as trocas acorrem até a penúltima posição. O processo continua até que não haja mais trocas.
- O algoritmo de ordenação por inserção é o mais eficiente dos três. Ele vasculha elemento por elemento da lista inserindo-o já na posição adequada considerando uma lista inicialmente vazia e que se encontra no início do prório vetor a ser ordenado.

Na próxima aula...

… veremos como uma função ou procedimento pode realizar uma chamada a ela mesma para resolver problemas de forma mais intuitiva. Veremos também um outro algoritmo de ordenação que faz uso dessa técnica e é o algoritmo de ordenação mais eficiente existente e o mais utilizado.

Referências e Sugestões de Leitura

Uma avaliação mais profunda sobre algoritmos de ordenação, com análise de desempenho e comparações, é realizada em

ZIVIANI, Nivio. Projeto de Algoritmos com implementações em java e C++, Thomson, 2007.

Implementação de algoritmos de pesquisa e de ordenação em Java pode ser vista no capítulo 16 de

DEITEL, H., DEITEL, P. Java como Programar, Prentice-Hall, 2005.

Exercícios Propostos

7.1 – Escreva um programa Java que gere um vetor de 1000 elementos inteiros aleatoriamente e ordene-o de forma decrescente. Utilize o método de ordenação

que desejar.

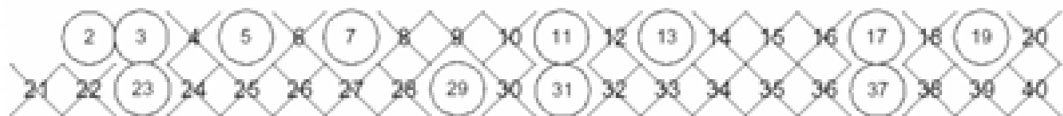
7.2 - Escreva um programa que determine se um array de inteiros está ou não ordenado. No primeiro caso deve indicar se o ordenamento é de forma crescente ou decrescente.

7.3 - Escreva um programa que preencha um array de inteiros com 20 valores aleatórios entre 0 e 99. Mostre o conteúdo deste array, listando os vinte elementos por ordem inversa. A seguir, pedir um número ao utilizador e procurar este número no array. Se encontrar, informe a posição em que foi encontrado, caso contrário diga que não foi encontrado. O programa termina quando o utilizador introduzir um número negativo.

7.4 - Faça um programa de consulta de telefones a partir de um nome informado por uma chave de dados: leia nomes de pessoas com seus respectivos telefones, sendo a quantidade determinada pelo usuário. Em seguida pergunte ao usuário qual o nome que ele deseja consultar o telefone. Após sua resposta, exiba o telefone da pessoa procurada. Informe também se o nome é inexistente no banco de dados.

7.5 – Faça um programa que leia dois conjuntos de números (podem ser de tamanhos diferentes) e ordene-os de forma crescente. Crie um outro vetor para armazenar os dois conjuntos unidos, sendo que os números devem permanecer ordenados. Finalmente, exiba este vetor resultante.

7.6 – Uma forma simples e eficiente de calcular todos os números primos até um certo valor n é o método da *Peneira de Eratosthenes*. O processo é simples: escrevem-se todos os valores entre 2 e n (limite máximo). Em seguida, faz-se um círculo em volta do 2, marcando como primo e riscam-se todos os seus múltiplos. Continua-se a fazer círculos em volta do menor inteiro que se encontra, eliminando todos os seus múltiplos. Quando não restarem números sem terem círculos à volta ou traços por cima, os números com círculos à volta representam todos os primos até n. A figura seguinte apresenta o método para n=40.



Escreva um programa em Java que implemente a *Peneira de Eratosthenes*. Deverá ser pedido ao utilizador um número inteiro correspondente ao limite máximo. Crie um vetor com todos os inteiros de 2 a n e elimine os valores não-primos.

No final, o usuário deve ter a possibilidade de entrar com um valor e o programa deve dizer se é um valor primo ou não efetuando uma busca do valor no vetor final. Se não encontrar, valor não é primo, caso contrário, sim.

7.7 – Um outro algoritmo de ordenação se chama *Bucket Sort.* O método de

ordenação inicia com um vetor unidimensional de inteiros positivos a ser ordenado e um vetor bidimensional (matriz) de inteiros com linhas indexadas de 0-9 e colunas indexadas de 0 a n-1, onde n é o número dos valores a ser ordenado. Cada linha do vetor bidimensional é chamada de bucket. Escreva um método chamado *bucketsort* que opera da seguinte maneira:

a) coloque cada valor do vetor unidimenional em uma linha do bucket, com base nas "unidades" (o dígito mais à direita) do valor. Por exemplo, o valor 97 seria colocado na linha 7, 3 na linha 3 e 100 na linha 0. Esse procedimento é chamado "passagem de distribuição"

b) realize um loop pelo bucket linha por linha e copie os valores de volta para o vetor original. Esse procedimento é chamado "passagem de coleta". A nova ordem dos valores precedentes no vetor unidimensional é 100, 3 e 97.

c) repita esse processo para a posição de cada dígito subseqüente (dezenas, centenas, milhares, etc.).

Na segunda passagem (dezenas), 100 é colocado na linha 0, 3 é colocado na linha e 97 é colocado na linha 9. Depois da passagem de coleta, a ordem dos valores no vetor unidimensional é 100, 3 e 97. Na terceira passagem (centenas), 100 é colocado na linha 1, 3 é colocado na linha 0 e 97 é colocado na linha 0, depois do 3. Depois desta última passagem de coleta, o vetor original estará ordenado. Observe que o vetor bucket tem dez vezes o comprimento do vetor sendo classificado. Essa técnica de ordenação tem desempenho melhor que um *bubblesort*, mas exige muito mais memória – o *bubblesort* exige espaço para somente um elemento adicional de dados.