ESTRUTURA DE DADOS

AULA 2

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

O objetivo desta aula é apresentar os conceitos que envolvem algoritmos de buscas e de ordenação de estruturas de dados. Ao longo de nossas discussões, serão apresentados diferentes algoritmos para realizar estas tarefas, cada um com as suas especificidades quanto à implementação e ao desempenho.

Para a realização da ordenação, os algoritmos tratados nesta aula serão:

- Ordenação por troca (*bubble sort*);
- Ordenação por intercalação (*merge sort*);
- Ordenação rápida (*quick sort*).

E os algoritmos de busca trabalhados serão:

- Busca sequencial;
- Busca binária.

Todos os todos os conceitos já trabalhados anteriormente, como a análise assintótica e recursividade, continuarão aparecendo de forma recorrente ao longo de toda aula.

TEMA 1 – ALGORITMOS DE ORDENAÇÃO

É usual dispormos de dados armazenados em um sistema que necessitam de algum tipo de ordenação. Certamente, em sua vida profissional, você necessitará ordenar dados com característica numérica, em ordem crescente ou decrescente, ou, então, nomes de pessoas cadastrados em uma agenda de contatos por ordem alfabética. Estas são tarefas corriqueiras no âmbito da computação.

É fato que raramente precisamos ordenar um único tipo de dado isolado, pois sempre mantemos um **conjunto de dados** dos mais diferentes para cada registro. Por exemplo, em um cadastro de dados de alunos matriculados em uma universidade, o registro de cada aluno pode conter um nome, um endereço, e, muitas vezes, outros dados pessoais. Ainda, é possível manter-se um registro único que está relacionado a este conjunto de dados. Neste caso, poderíamos, para validar o exemplo, utilizar um código de matrícula do aluno. Um código que seria único em todo o sistema e permitiria a caracterização do aluno.

Podemos extrapolar este conceito com a afirmação de que cada dado pertencente a uma coleção de dados deve conter um **número de registro (chave)**. Esta chave serve para identificar de forma única um dado dentro de toda a coleção. Todos os dados que estão vinculados a uma chave são chamados de **dado satélite** (Cormen, 2012).

Quando um algoritmo de ordenação permuta os dados, para colocá-los na ordem desejada, ele deve não só trocar os dados chave de posição, como também trazer junto, nesta troca de posições, todos os dados satélites de cada registro. Esta movimentação de dados, em memória ou não, consome recursos computacionais.

Uma maneira de otimizarmos um algoritmo é, em vez de permutarmos toda a coleção de dados satélite de cada entrada de dados, vincular uma variável do tipo **ponteiro** a cada registro. Assim, permutamos e organizamos somente estes ponteiros, não precisando realocar todo o conjunto de dados satélite em memória, reduzindo o consumo de recursos, aumentando a eficiência o desempenho do processo.

Um algoritmo de ordenação é um método que descreve como é possível colocar, em uma ordem específica, um conjunto de dados qualquer (Ascencio, 2012). Estes algoritmos devem ser independentes dos tipos de dados que serão ordenados, do volume de dados e da linguagem de programação utilizada para ordenação.

Para um melhor entendimento dos algoritmos de ordenação, adotaremos conjuntos de dados compostos de estruturas de dados homogêneas, numéricas e unidimensionais (vetores). Com os mesmos algoritmos que estudaremos, com este conjunto simples, será possível ordenar qualquer outro conjunto de dados, como caracteres alfanuméricos, por exemplo, ou mesmo conjuntos de dados de estruturas heterogêneas e/ou multidimensionais (matrizes ou tensores).

A busca de métodos de ordenação mais eficientes constitui um dos problemas fundamentais no desenvolvimento de algoritmos. Com impacto no custo de execução de determinado *software*, ainda não existe uma solução definitiva para este problema. Contudo, existem algoritmos muito eficientes para resolver problemas específicos.

Os algoritmos de ordenação são utilizados para tarefas cotidianas e/ou para resolver problemas de importância fundamental para o desenvolvimento da sociedade:

- Alguma aplicação pode necessitar visualizar dados ordenados, como listar todos os nomes de pessoas em uma agenda de e-mails por ordem alfabética;
- Programas de renderização gráfica utilizam algoritmos de ordenação para desenhar objetos gráficos em uma ordem pré-definida pelo desenvolvedor, de baixo para cima, por exemplo;
- Algoritmos de inteligência artificial utilizam a ordenação para localizar setores específicos da molécula de dna em busca de novas curas;
- Sistemas de tráfego aéreo utilizam critérios de prioridade para determinação da ordem de pouso em momentos de emergência.

TEMA 2 – ALGORITMO DE ORDENAÇÃO POR TROCA (*BUBBLE SORT*)

O algoritmo para o entendimento inicial da ordenação de dados é o de ordenação por troca, também chamado de ordenação bolha ou *bubble sort*. Apesar de ser o algoritmo de ordenação mais simples, e frequentemente o mais utilizado para o entendimento do processo de ordenação, este é um algoritmo computacionalmente muito caro e muito ineficiente. Notadamente porque emprega dois laços de repetição aninhados.

Dentro do laço interno, existe um teste condicional que compara o valor de uma posição do vetor com o próximo valor, efetuando uma troca entre eles, caso necessário. Com esta única comparação, o algoritmo é capaz de tratar um conjunto de dados de entrada desordenado e produzir um conjunto de dados de saída ordenado. Por exemplo, podemos ordenar valores numéricos de maneira crescente.

2.1 Pseudocódigo

A Figura 1 ilustra o pseudocódigo deste método. Nas linhas 6 a 8, há' a leitura dos dados de um vetor de tamanho declarado e chamado de **TAMANHOVETOR** em nosso código**.** Este tamanho pode ser qualquer valor definido pelo desenvolvedor. Das linhas 24 a 27, temos a impressão na tela do vetor já ordenado, de forma crescente.

Figura 1 – Pseudocódigo da ordenação por troca (*bubble sort*) crescente

```
1   algoritmo "BubbleSort"
2   var
3      X[TAMANHOVETOR], i, j, aux: inteiro
4   inicio
5         //PREENCHE O VETOR
6         para i de 0 até (TAMANHOVETOR - 1) faça
7            leia(X[i])
8         fimpara
9
10        //BUBBLE SORT CRESCENTE
11        //LAÇO EXTERNO DO TAMANHO VO VETOR
12        para i de 1 até TAMANHOVETOR faça
13           //LAÇO INTERNO DA 1ª ATÉ PENÚLTIMA POSIÇÃO
14           para j de 0 até (TAMANHOVETOR - 2) faça
15              //TROCA (SWAP)
16              se (X[j] > X[j + 1]) então
17                 aux <- X[j]
18                 X[j] <- X[j + 1]
19                 X[j + 1] <- aux
20              fimse
21           fimpara
22        fimpara
23
24        //IMPRIME O VETOR ORDENADO CRESCENTE
25        para i de 0 até (TAMANHOVETOR - 1) faça
26           leia(X[i])
27        fimpara
28  fimalgoritmo
```

O método de ordenação está explicitado entre as linhas 10 e 22, de forma crescente. Ou seja, os valores numéricos colocados no vetor são ordenados do menor para o maior valor. Vejamos em detalhe cada linha do algoritmo de ordenação:

- Linha 12: laço de repetição externo. Ele deve, obrigatoriamente, ter o tamanho do vetor;
- Linha 14: Laço de repetição interno, encadeado. Deve passar por todas as posições do vetor, iniciando no zero, e indo até a penúltima posição;
- Linha 16: condição que testa se o valor atual da varredura do vetor é **maior** que o valor subsequente a ele (**ordenação crescente**);
- Linhas 17, 18, 19: condicional simples. Caso a linha 16 resulte em **verdadeiro**, estas linhas serão executadas. Elas fazem a troca dos dois elementos do vetor testados na linha 16, utilizando uma variável auxiliar para armazenar temporariamente um destes valores, assim, ele não é apagado da memória do programa.

Por exemplo, vamos assumir um vetor numérico, aleatoriamente preenchido de comprimento 5, apresentado na Equação 1. Este vetor será ordenado de forma crescente, seguindo o pseudocódigo da Figura 1.

$$Vetor[5] = [6, 1, 8, 2, 7] \quad (1)$$

Para cada iteração do laço externo (linha 12), o laço interno (linha 14) passa por todo o vetor, fazendo comparações e trocas. Como a variável *i* é inicializada com 1 ($i = 1$), uma varredura completa do vetor para a primeira iteração é ilustrada na Figura 2. Os asteriscos duplos ** representam os 2 elementos comparados naquela iteração, e trocados, caso necessário. Cada linha da Figura 2 é uma iteração do laço interno.

Figura 2 – Funcionamento da ordenação por troca (*bubble sort*) para $i = 1$ e $j = 0 \ldots 4$

```
1  [**1, 6**, 8, 2, 7] -- 1 < 6 -> TROCA
2  [1, **6, 8**, 2, 7] -- 8 > 6 -> NÃO TROCA
3  [1, 6, **8, 2**, 7] -- 2 < 8 -> TROCA
4  [1, 6, 2, **7, 8**] -- 7 < 8 -> TROCA
```

Na Figura 2, linha 1, comparamos a primeira posição ($j = 0$) com a segunda ($j = 1$). Como estamos ordenando do menor para o maior, e $6 > 1$, trocamos. Já na linha 2, comparamos a segunda posição ($j = 1$) com a terceira ($j = 2$), e $8 > 6$. Portanto, o valor já está na posição correta, não havendo troca. Analogamente, fazemos as avaliações linha a linha da Figura 1.

Observe que o vetor final obtido na linha 4 da Figura 2 ainda não está totalmente ordenado de forma crescente. Isso ocorre porque este resultado é para $i = 1$. A variável *i* ainda precisa ser incrementada unitariamente até atingir o tamanho do vetor. E para cada incremento de *i*, teremos o laço interno executando novamente as 4 vezes, de forma análoga à Figura 2. Considerando o tamanho do vetor sendo 5, teremos, então, 20 repetições de testes de troca ($5 * 4 = 20$) sendo efetuadas até que o método bolha seja encerrado.

A Figura 3 ilustra a segunda iteração no laço externo ($i = 2$). Observe que, ao final deste ciclo de teste, tivemos somente uma troca efetuada, por sua vez, na anterior, forem três. Ainda, perceba que, na iteração 8, o vetor resultante já é um vetor ordenado crescentemente. Apesar disso, não significa que nosso algoritmo está encerrado. O modo como o algoritmo está construído na Figura 1

não sessa as comparações mesmo que o vetor já esteja ordenado. As iterações com $i = 3$ e $i = 4$ ocorrerão mesmo com o vetor já totalmente ordenado.

Figura 3 – Funcionamento da ordenação por troca (*bubble sort*) para $i = 2$ e $j = 0 \ldots 4$

```
5   [**1, 6**, 2, 7, 8] -- 6 > 1 -> NÃO TROCA
6   [1, **2, 6**, 7, 8] -- 2 < 6 -> TROCA
7   [1, 2, **6, 7**, 8] -- 7 > 6 -> NÃO TROCA
8   [1, 2, 6, **7, 8**] -- 8 > 7 -> NÃO TROCA
```

Agora que você entendeu como funciona o *bubble sort*¸ vamos entender alguns pontos interessantes do algoritmo. O laço interno (linha 14 da Figura 4) é sempre executado somente até a penúltima posição do vetor. Mas por que não até a última? O motivo é porque este laço, ao chegar à penúltima posição, fará o teste condicional da linha 16, já comparando a penúltima com a última posição. Se este laço fosse até a última posição, não existiria um valor subsequente para ele comparar.

Figura 4 – Recorte do pseudocódigo da ordenação por troca apresentada na Figura 1

```
10          //BUBBLE SORT CRESCENTE
11          //LAÇO EXTERNO DO TAMANHO VO VETOR
12          para i de 1 até TAMANHOVETOR faça
13              //LAÇO INTERNO DA 1ª ATÉ PENÚLTIMA POSIÇÃO
14              para j de 0 até (TAMANHOVETOR - 2) faça
15                  //TROCA (SWAP)
16                  se (X[j] > X[j + 1]) então
```

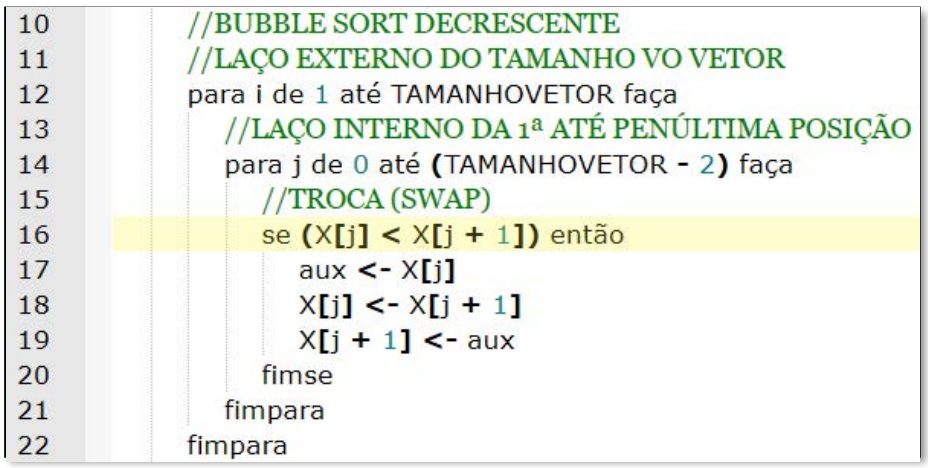
Outra característica interessante é o uso da variável para auxiliar na troca dos valores no vetor. Caso ela não fosse empregada, um valor do vetor poderia se sobrepor a outro ao realizar a troca, fazendo com que o valor sobrescrito desaparecesse. Assim, uma variável é utilizada para armazenar temporariamente o valor de um dos valores da troca do vetor. Este recurso da variável auxiliar é muito empregado em algoritmos de ordenação de modo geral, e continuará aparecendo em nossos próximos algoritmos.

Agora, e se desejarmos ordenar os dados de maneira decrescente? Ou seja, do maior para o menor, como poderíamos alterar nosso algoritmo para funcionar desta maneira?

A solução é bastante simples e envolve alterar unicamente uma linha de código. A linha 16 (Figura 4 e Figura 5) realiza a comparação de um valor com outro, portanto, se invertermos a comparação que está sendo feita nesta linha, obteremos uma ordenação inversa, conforme a Figura 5. A linha destacada em amarelo representa a alteração.

É interessante ressaltar que esta linha servirá para comparar qualquer tipo de variável. Portanto, se estivéssemos comparando caracteres alfanuméricos, poderíamos realizar esta comparação também alterando a linha 16 na Figura 5.

Figura 5 – Pseudocódigo da ordenação por troca (*bubble sort*) decrescente



2.2 Aprimoramento do *bubble sort*

Conforme vimos no pseudocódigo da Figura 1 e da Figura 4, o algoritmo proposto continua testando se os valores estão posicionados corretamente, mesmo quando o vetor já está ordenado. O que constitui desperdício de recursos computacionais.

Podemos solucionar este problema e otimizar o desempenho de nosso código, bem como trocar nosso laço externo do tipo *para* por um laço *enquanto* (ou mesmo um *repita*). Neste laço de *enquanto*, continuaremos fazendo nossa contagem que vai até o tamanho do vetor, exatamente como antes, mas também acrescentamos outra condição, uma variável do tipo lógico com dois estados, inicializada com nível lógico baixo e que se mantém neste valor ao menos que exista uma troca. Portanto, enquanto houver trocas, seu valor será de nível alto. Quando o nível lógico volta a ser baixo, o laço encerrará precocemente, economizando nas iterações desnecessárias. Confira o pseudocódigo na Figura 6, que ordena crescentemente.

Figura 6 – Pseudocódigo da ordenação por troca (*bubble sort*) crescente aprimorado com uma variável de *flag*

```
1  algoritmo "BubbleSortAprimorado"
2  var
3     X[TAMANHOVETOR], i, j, aux: inteiro
4     troca: lógico
5  inicio
6        //PREENCHE O VETOR
7        para i de 0 até (TAMANHOVETOR - 1) faça
8           leia(X[i])
9        fimpara
10
11       //BUBBLE SORT CRESCENTE
12       i = 1, troca = 1
13       //LAÇO EXTERNO DO TAMANHO VO VETOR
14       enquanto ((i <= TAMANHOVETOR) E (troca == 1))
15          troca = 0
16          //LAÇO INTERNO DA 1ª ATÉ PENÚLTIMA POSIÇÃO
17          para j de 0 até (TAMANHOVETOR - 2) faça
18             //TROCA (SWAP)
19             se (X[j] > X[j + 1]) então
20             troca = 1
21                aux <- X[j]
22                X[j] <- X[j + 1]
23                X[j + 1] <- aux
24             fimse
25          fimpara
26          i = i + 1
27       fimpara
28
29       //IMPRIME O VETOR ORDENADO CRESCENTE
30       para i de 0 até (TAMANHOVETOR - 1) faça
31          leia(X[i])
32       fimpara
33 fimalgoritmo
```

2.3 Complexidade Assintótica

A complexidade assintótica para o pior caso do método de ordenação por troca, tanto para a versão original (Figura 1) quanto para a versão aprimorada (Figura 6) será igual.

Lembrando que um laço de repetição só tem impacto no desempenho assintótico do algoritmo caso estejam aninhados, ou seja, um inserido no outro. Na Figura 1, percebemos a existência de dois laços de repetição aninhados (linha 12 e 14). Portanto, temos para *Big-O*:

$$0_{BubbleSort}(n^2) \quad (2)$$

Analogamente, a versão aprimorada também tem dois laços aninhados, resultando na mesma complexidade para o pior caso. Se a complexidade é a mesma, por que então utilizamos a versão aprimorada? Porque para as situações em que o pior caso não ocorrerá, a nova versão (Figura 5) se sairá melhor.

Na primeira versão, independentemente da situação, as iterações serão executadas até o final, então a complexidade para o melhor caso (BigΩ) será igual à do pior caso $\Omega(n^2)$. Já na versão aprimorada, em situações de melhor caso, apresentará um comportamento $\Omega(n)$.

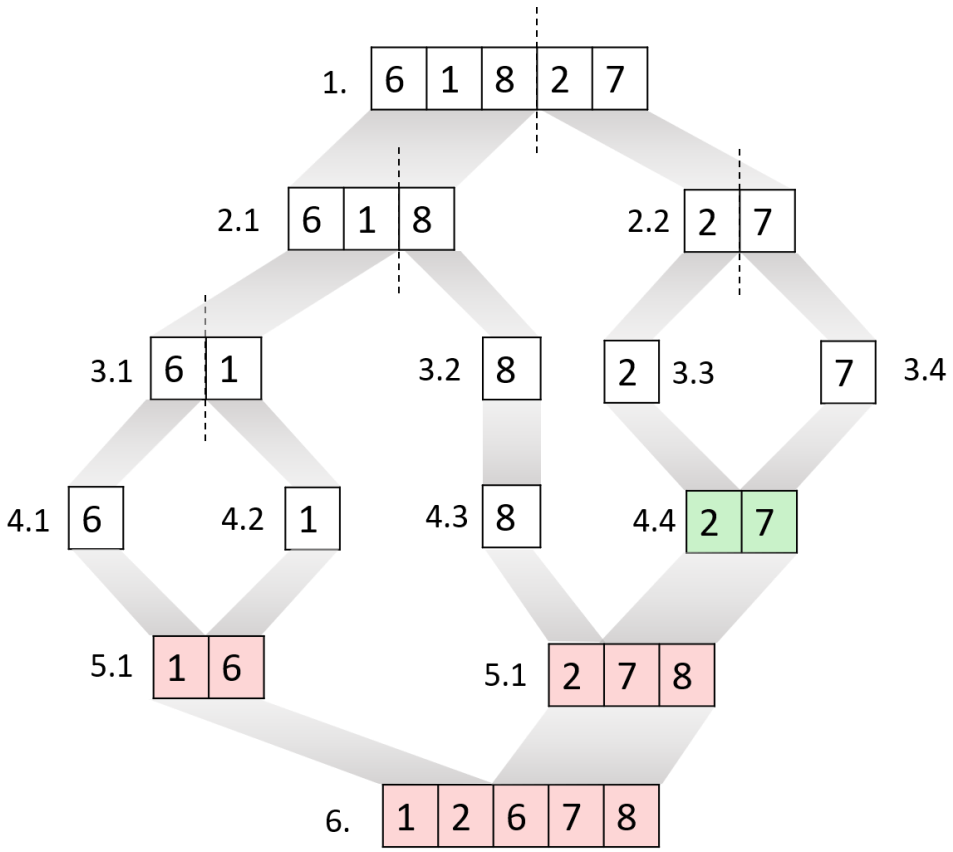
TEMA 3 – ALGORITMO DE ORDENAÇÃO POR INTERCALAÇÃO (*MERGE SORT*)

Este algoritmo de ordenação trabalha com a divisão da estrutura de dados em partes menores, empregando uma técnica chamada de dividir para conquistar. Para tal, empregam-se funções recursivas para a realização desta tarefa.

O processo do *merge sort* consiste em dividir uma estrutura de dados de tamanho *n* (um vetor por exemplo) em *n* partes de tamanho unitário. Ou seja, o conjunto de dados inicial vai sendo dividido ao meio até que restem somente partes indivisíveis. Quando duas partes subsequentes atingirem tamanho um, elas realizam o processo chamado de intercalação. De forma simples, podemos entender a intercalação como a ordenação (crescente ou decrescente) seguida da agregação das partes para formarem um vetor maior.

Vamos observar um exemplo de *merge sort*¸ apresentado na Figura 7. Temos um vetor aleatório, o mesmo utilizado no *bubble sort* $Vetor[5] = [6, 1, 8, 2, 7]$, que precisa ser ordenado de forma crescente. O *merge sort* vai, então, seguindo o princípio de “dividir para conquistar”, quebrar ao meio o vetor inicial de tamanho 5, e continuará quebrando até que restem somente partes unitárias para serem ordenadas.

Figura 7 – Exemplo de funcionamento da ordenação por intercalação (*merge sort*) crescente



A Figura 7 está dividida em 6 etapas. Na etapa 1, o vetor de tamanho 5 será dividido ao meio. Como o *merge sort* trabalha com recursividade, na primeira etapa, a função de ordenação é chamada pela primeira vez. O ponto do meio será na terceira posição ($Vetor[2] = 8$). Portanto, ficaremos com 2 vetores menores (Figura 5 – 2.1 e 2.2), um de tamanho 3, outro de tamanho 2. Esta primeira divisão ocorreu por meio da chamada de uma segunda instância da função (recursividade). Neste momento, temos duas instâncias de *merge sort* abertas na memória de nosso programa.

Como ambos vetores ainda não têm tamanho unitário, precisamos chamar a função de divisão novamente, abrindo uma terceira instância de *merge sort*. Nossos 2 vetores virarão 4 (3.1, 3.2, 3.3 e 3.4). Observe que, agora, temos alguns vetores unitários (3.2, 3.3 e 3.4). Portanto, nesta etapa, todos os vetores de tamanho um começam a ser intercalados (comparados entre si, ordenados crescentemente e agregados para um tamanho maior).

Quando os vetores unitários 3.3 e 3.4 são ordenados, eles voltam a ser agregados, tendo novamente tamanho 2 e encerrando a chamada de uma função recursiva. Na etapa 4, temos 4.1 e 4.2 sendo ordenados e agregados

novamente, bem como 4.3 e 4.4. Por fim, o vetor original vai sendo retomado e ordenado à medida que as chamadas recursivas vão se encerrando.

3.1 Pseudocódigo

As Figura 8, 9 e 10 ilustram o pseudocódigo da ordenação por intercalação (*merge sort*). Na Figura 8, temos o algoritmo principal, onde há a leitura do vetor com dados aleatórios (linhas 5 a 8), a impressão na tela dos dados (linhas 13 a 16) e a primeira chamada da função *merge()* (linha 11). A função *merge()* é ativada com a passagem de três parâmetros. O primeiro é o vetor de dados a ser ordenado, o segundo é a posição inicial do vetor que será ordenado e o terceiro é a última posição a ser ordenada. Como queremos ordenar todo o vetor, passamos como posição inicial o valor zero e como posição final $TAMANHODOVETOR - 1$.

Figura 8 – Pseudocódigo da ordenação por intercalação (*merge sort*) crescente. Algoritmo principal com a chamada da função de ordenação *merge*

```
1  algoritmo "MergeSort"
2  var
3     X[TAMANHOVETOR], i: inteiro
4  inicio
5        //PREENCHE O VETOR
6        para i de 0 até (TAMANHOVETOR - 1) faça
7           leia(X[i])
8        fimpara
9
10       //MERGE SORT CRESCENTE
11       merge(X, 0, TAMANHOVETOR) //CHAMA A FUNÇÃO DE ORDENAÇÃO
12
13       //IMPRIME O VETOR ORDENADO CRESCENTE
14       para i de 0 até (TAMANHOVETOR - 1) faça
15          leia(X[i])
16       fimpara
17 fimalgoritmo
```

A função de ordenação chamada *merge* é apresenta na Figura 9. Ela é responsável por alguns pontos na ordenação. Vamos observar a análise de cada linha deste pseudocódigo:

- Linha 22: condição que testa se a dimensão do vetor é unitária ou é maior que um. Caso início seja igual a fim, significa que é unitário;
- Linha 23: encontra o ponto do meio do vetor;

- Linha 24: após a descoberta do ponto central, chama recursivamente a função merge para a primeira metade do vetor recebido como parâmetro, abrindo uma nova instância desta função;
- Linha 25: após a descoberta do ponto central, chama recursivamente a função merge para a segunda metade do vetor recebido como parâmetro, abrindo uma nova instância desta função;
- Linha 26: agrega os vetores menores em um de tamanho maior e ordenado (crescentemente neste exemplo) por meio da chamada de outra função chamada *intercala*;

Figura 9 – Pseudocódigo da ordenação por intercalação (*merge sort*) crescente. Código da função *merge*, responsável por subdividir recursivamente a estrutura de dados.

```
18  função merge(X, inicio, fim)
19  var
20     meio: inteiro
21  inicio
22        se (inicio < fim) então
23           meio <- parteinteira((inicio + fim) / 2) //DIVIDE AO MEIO
24           merge(X, inicio, meio) //CONTINUA DIVIDINDO A PARTIR DA 1ª METADE
25           merge(X, meio + 1, fim) //CONTINUA DIVIDINDO A PARTIR DA 2ª METADE
26           intercala(X, inicio, fim, meio) //AGREGA DE VOLTA 2 VETORES ORDENADOS
27        fimse
28  fimfunção
```

Na Figura 10, temos o algoritmo que realiza a ordenação com uma função chamada *intercala*. Perceba, na declaração das variáveis, que dispomos de um vetor auxiliar, responsável por receber os dados ordenados ao longo da função e, ao final dela (linhas 59, 60 e 61), transpor para o vetor original e ordenado.

Nas linhas 33 a 57, o algoritmo salva no vetor auxiliar os valores de ambos vetores que estão sendo agregados aos já ordenados. Ambos vetores vão sendo comparados entre si no momento da ordenação. Lembrando que tais vetores não necessitam ter o mesmo tamanho (vide Figura 7 – 4.3 e 4.4).

Nas linhas 59 a 61, todo o vetor auxiliar ordenado é passado diretamente para o nosso vetor principal, e até então desordenado (vetor *X* em nosso algoritmo). Observe que somente os valores alterados passarão para o vetor *X*. todos os valores não ordenados não são sobrescritos.

Figura 10 – Pseudocódigo da ordenação por intercalação (*merge sort*) crescente. Código da função *intercala*, responsável por ordenar duas partes da estrutura de dados

```
29  função intercala(X, inicio, fim, meio)
30  var
31     poslivre, inicio_vetor1, inicio_vetor2, i, aux[TAMANHOVETOR]: inteiro
32  inicio
33     inicio_vetor1 <- inicio
34     inicio_vetor2 <- meio + 1
35     poslivre <-meio + 1
36     enquanto (inicio_vetor1 <= meio E inicio_vetor2 <= fim)
37        se (X[inicio_vetor1] <= x[inicio_vetor2]) então
38           aux[poslivre] <- X[inicio_vetor1]
39           inicio_vetor1 <- inicio_vetor1 + 1
40        senão
41           aux[poslivre] <- X[inicio_vetor2]
42           inicio_vetor2 <- inicio_vetor2 + 1
43        fimse
44        poslivre <- poslivre +1
45     fimenquanto
46     //SE AINDA EXISTIR NUMEROS NO PRIMEIRO VETOR
47     //QUE NÃO FORAM INTERCALADOS
48     para i de inicio_vetor1 até meio faça
49        aux[poslivre] <- X[i]
50        poslivre <- poslivre + 1
51     fimpara
52     //SE AINDA EXISTIR NUMEROS NO SEGUNDO VETOR
53     //QUE NÃO FORAM INTERCALADOS
54     para i de inicio_vetor2 até fim faça
55        aux[poslivre] <- X[i]
56        poslivre <- poslivre + 1
57     fimpara
58     //RETORNA OS VALORES DO VETOR AUXILIAR PARA O VETOR X
59     para i de inicio até fim faça
60        X[i] <- aux[i]
61     fimpara
62  fimfunção
```

3.2 Complexidade Assintótica

Para encontrarmos a complexidade assintótica para o pior caso do *merge sort*, precisamos, primeiro, analisar a complexidade individual de ambas funções propostas. Na Figura 9, temos a função *merge* sendo chamada recursivamente. A complexidade assintótica para o pior caso de uma função recursiva será $O(log(n))$.

A função *merge* também faz a chamada de outra função, a *intercala* (Figura 10). Esta função opera de forma iterativa. Não temos laços encadeados, portanto a complexidade para um único laço será $O(n)$.

Como a função *intercala* está inserida na função *merge*, para encontrar a complexidade assintótica *Big-Oh* resultante, multiplicamos a complexidade de ambas funções (Equação 3). A complexidade para o pior do *merge sort* é superior em tempo de execução se comparado com o *bubble sort*.

$$0_{MergeSort}(n.\log(n)) \quad (3)$$

TEMA 4 – ALGORITMO DE ORDENAÇÃO RÁPIDA (*QUICK SORT*)

O *quick sort* é bastante empregado em virtude de sua eficiência e facilidade de implementação. A ordenação rápida trabalha com o conceito de pivô, em que um elemento do conjunto de dados é selecionado para servir como referência de comparação para os outros valores.

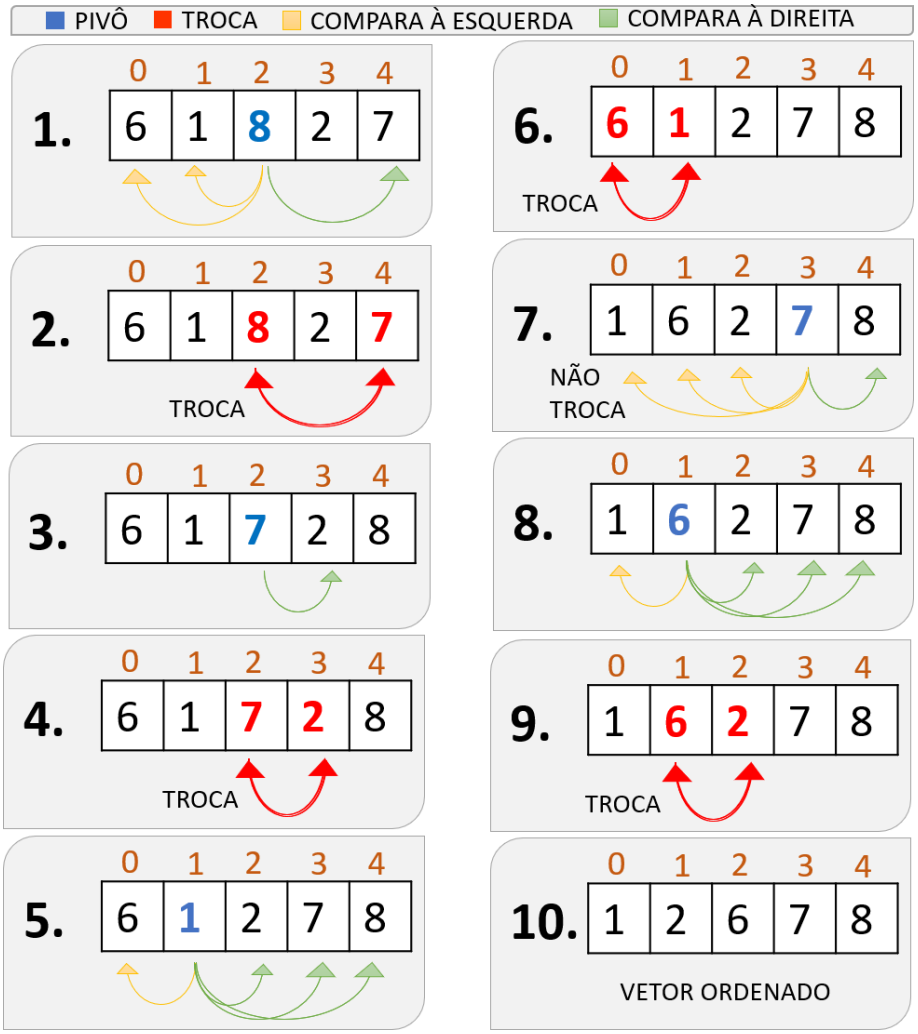
A ordenação ocorre quando o método divide a estrutura de dados em duas partes, semelhante ao *merge sort*, porém esta divisão não necessariamente ocorre no ponto central do vetor.

Diferentemente do *merge sort*, o objetivo deste algoritmo não é quebrar o vetor no menor tamanho possível, mas sim dividi-lo para encontrar o elemento pivô. Para uma ordenação crescente, os elementos à esquerda do pivô devem sempre ser menores do que ele, e os à direita, maiores que o pivô. O algoritmo é aplicado de forma recursiva nas partes divididas até que a estrutura esteja ordenada.

Vamos analisar um exemplo de *quick sort*, com o mesmo vetor $Vetor[5] = [6, 1, 8, 2, 7]$ empregado no *bubble sort* e no *merge sort*. O método de ordenação levará 10 passos para ordenar completamente, de forma crescente, o vetor de tamanho 5.

Observe na Figura 11 que, diferentemente do *merge sort*, o vetor não vai sendo dividido em partes menores, ou seja, não opera com o princípio de dividir para conquistar. Uma nova instância da função de ordenação é aberta sempre que um novo pivô precisa ser encontrado.

Figura 11 – Exemplo de funcionamento da ordenação rápida (*quick sort*) crescente



Vamos analisar todos os passos do algoritmo de *quick sort,* conforme apresentado na Figura 11. O algoritmo inicia encontrando o pivô. Este pivô será inicialmente o valor do meio do vetor, assim como ocorre no *merge sort*. Portanto, a posição do meio será $Vetor[2] = 8$.

Com base no pivô e considerando uma ordenação crescente, o algoritmo compara todos os valores que estão à esquerda dele, verificado se eles são menores que o pivô. Em seguida, são comparados todos os valores que estão à direita do pivô, verificando se eles são maiores que o valor de referência do pivô. Sempre que um valor maior que o pivô é encontrado do lado esquerdo e/ou um valor menor é achado do lado direito, ambos trocam de posição (*swap*). Assim, na Figura 11, etapa 1, o pivô é selecionado e comparado com os elementos esquerdos e direitos.

Na etapa 2, perceba que o elemento $Vetor[4] = 7$, ao lado direito, é encontrado, pois ele deveria estar antes do pivô (menor que o pivô). Como ao lado esquerdo não foi encontrado qualquer divergência, trocamos de lugar este elemento destoante com o próprio pivô.

As varreduras em ambos os lados continuam até que as duas atinjam a posição do pivô (etapa 3). Portanto, na etapa 4, temos mais uma troca com o pivô, pois ao lado direito existe mais um valor menor que o pivô. Quando as varreduras se encontram na posição do pivô, esta instância da função *quick sort* é encerrada e o ponto de parada da verdura direita, que será o valor do pivô, é retornado para a função que a chamou.

Na etapa 5, temos um novo pivô, obtido om base no ponto de parada da etapa 4. Nesta etapa, o algoritmo das varreduras novamente é efetuado em ambos os lados do pivô. Uma divergência é encontrada do lado esquerdo desta vez (6 > 1), pois o valor é maior que o pivô. Na etapa 6, temos esta troca acontecendo entre o elemento esquerdo e o pivô.

Na etapa 7, temos um novo pivô. Nesta situação, o elemento 7 já está no seu local correto, não existindo qualquer troca para ser feita, então as comparações são executadas sem trocas.

Na etapa 8, o pivô volta a ser a segunda posição, que contém outro valor agora. Temos uma troca para ser efetuada entre ele e o elemento à direita subsequente (etapa 9). Na etapa 10, temos o vetor final ordenado crescentemente.

4.1 Pseudocódigo

As Figura 12, 13, 14 e 15 ilustram o pseudocódigo da ordenação rápida (*quick sort*). Na Figura 11 está o algoritmo principal, em que temos a leitura do vetor com dados aleatórios (linhas 6 a 9), a impressão na tela dos dados (linhas 14 a 16) e a primeira chamada da função *quicksort()* (linha 11). A função *quicksort()* é chamada passando três parâmetros para a função. O primeiro é o vetor de dados a ser ordenado, o segundo é a posição inicial da ordenação e o terceiro é a última posição a ser ordenada. Como queremos ordenar todo o vetor, passamos como posição inicial o valor zero e como posição final $TAMANHODOVETOR - 1$.

Figura 12 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Algoritmo principal com a chamada da função de ordenação *quicksort*

```
1   algoritmo "QuickSort"
2   var
3      X[TAMANHOVETOR], i: inteiro
4   inicio
5         //PREENCHE O VETOR
6         para i de 0 até (TAMANHOVETOR - 1) faça
7            leia(X[i])
8         fimpara
9
10        //QUICK SORT CRESCENTE
11        quicksort(X, 0, TAMANHOVETOR) //CHAMA A FUNÇÃO DE ORDENAÇÃO
12
13        //IMPRIME O VETOR ORDENADO CRESCENTE
14        para i de 0 até (TAMANHOVETOR - 1) faça
15           leia(X[i])
16        fimpara
17  fimalgoritmo
```

A função de ordenação chamada *quicksort()* é apresenta na Figura 13. Ela é responsável por algumas etapas na ordenação. Vamos observar a análise de cada linha deste pseudocódigo:

- Linha 23: condição que testa se ainda temos alguma parte para ser ordenada;
- Linha 23: encontra o novo pivô do vetor. O cálculo é feito em uma nova função chamada *partição*, que também realiza a ordenação e retorna o valor ao final;
- Linha 24: após a descoberta do novo pivô, chama recursivamente a função quicksort() para a primeira metade do vetor recebido como parâmetro, abrindo uma nova instância desta função;
- Linha 25: após a descoberta do novo pivô, chama recursivamente a função *quicksort()* para a segunda metade do vetor recebido como parâmetro, abrindo uma nova instância desta função.

Figura 13 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Código da função *quicksort*, responsável por subdividir recursivamente a estrutura de dados

```
19  função quicksort(X, inicio, fim)
20  var
21     div: inteiro
22  inicio
23     se (inicio < fim) entao
24        div = particao(X, inicio, fim) //RETORNA O PONTO DE PARADA
25        quicksort(X, inicio, div) //CONTINUA DIVIDINDO A PARTIR DA 1ª PARTE
26        quicksort(X, div+1, fim) //CONTINUA DIVIDINDO A PARTIR DA 2ª PARTE
27     fimse
28  fimfunção
```

A função que realmente realiza a ordenação é chamada *partição* e está apresentada na Figura 14. Vejamos o que ela faz:

- Linha 34: encontra a posição do pivô;
- Linha 35: com a posição do pivô calculada, resgata o pivô nos dados;
- Linha 38: verifica se ainda resta algo para ser ordenado;
- Linhas 31 a 41: busca elementos destoantes a direita do pivô. Iniciando na penúltima posição do vetor e seguindo em direção ao pivô;
- Linhas 42 a 44: busca elementos destoantes a esquerda do pivô. Iniciando na primeira posição do vetor e seguindo em direção ao pivô;
- Linhas 47 a 49: Caso sejam localizados elementos destoantes nas linhas acima, uma função de troca é chamada;
- Linha 53: retorna o valor à direita do pivô para a função *quicksort()* que a chamou. Este valor retornado será usado nas próximas chamadas da função.

E se desejássemos realizar a ordenação decrescente de nosso vetor, onde deveríamos modificar nosso pseudocódigo? No *quick sort*, a alteração deve ser feita em duas linhas de código. Na função de *partição()* (Figura 14), bastaria invertermos o sinal da comparação das linhas 41 e 44, realizando uma ordenação decrescente.

Figura 14 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Código da função *particao*, responsável por encontrar o pivô e varrer os lados esquerdos e direito dele buscando valores incoerentes

```
30  função particao(X, inicio, fim)
31  var
32      posicao_pivo, pivo, i, j: inteiro
33  inicio
34      posicao_pivo = parteinteira((inicio + fim)/2);
35      pivo = X[posicao_pivo] //ENCONTRA O PIVÔ
36      i = inicio - 1
37      j = fim + 1
38      enquanto (i < j) faça
39          repita //PROCURA POR VALOR INCOERENTE À DIREITA DO PIVÔ
40              j = j - 1
41          até (X[j] <= pivo)
42          repita //PROCURA POR VALOR INCOERENTE À ESQUERDA DO PIVÔ
43              i = i + 1
44          até (X[i] >= pivo)
45          //TROCA UM VALOR INCOERENTE À ESQUERDA
46          //COM UM VALOR À DIREITA DO PIVÔ
47          se (i < j) então
48              troca(X, i, j);
49          fimse
50      fimenquanto
51      //RETORNA ONDE O LADO DIREITO PAROU
52      //ESTE VAOR SERÁ USADO COMO NOVO MEIO
53      retorne j
54  fimfunção
```

Na Figura 15, temos somente uma função de troca com variável auxiliar. A função recebe como parâmetro o vetor e as posições que precisam ser trocadas e executa a troca.

Figura 15 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Código da função *troca,* responsável por trocar dois valores incoerentes, um do lado esquerdo, outro do lado direito

```
56  função troca(X, i, j)
57  var
58      aux: inteiro
59  inicio
60      //TROCA COM VARIÁVEL AUXILIAR
61      aux = X[i]
62      X[i] = X[j]
63      X[j] = aux
64  fimfunção
```

4.2 Complexidade Assintótica

Para encontrarmos a complexidade assintótica para o pior caso do *quick*, precisamos analisar a complexidade individual de ambas funções propostas. Na Figura 13, temos a função *quicksort()* sendo chamada recursivamente. A complexidade assintótica para o pior caso de uma função recursiva será $O(log(n))$.

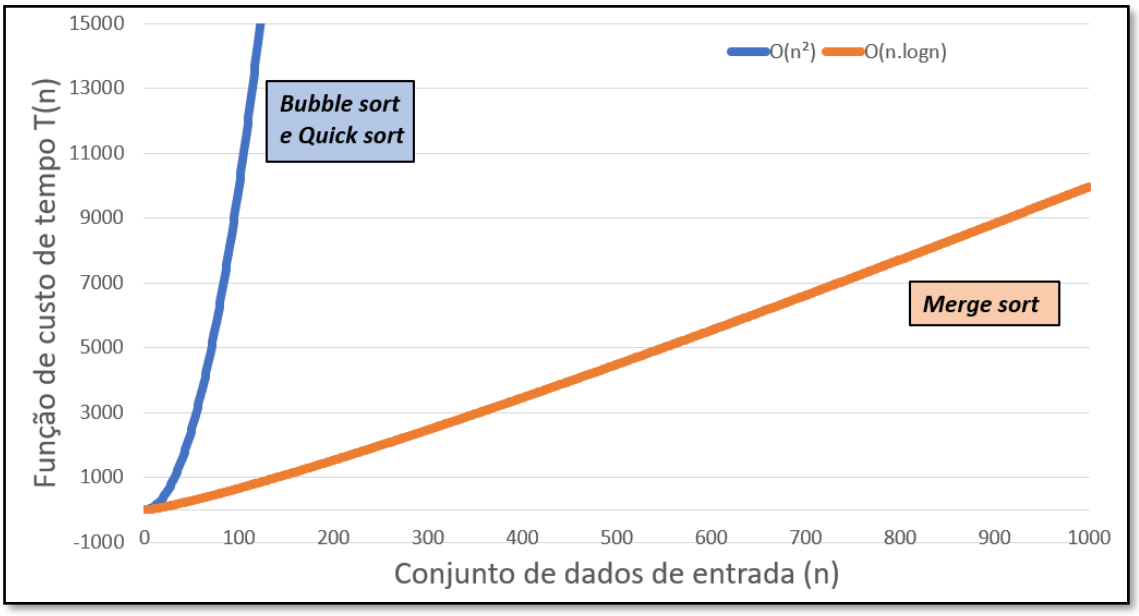
A função *quicksort()* também faz a chamada de outra função, a *partição()*. Esta função opera de forma iterativa. Neste caso, temos dois laços aninhados, portanto, a complexidade será $O(n^2)$. Note que temos um laço *enquanto* e, dentro dele, dois comandos *repita*. Porém, um comando repita só se inicia após o término do outro, não gerando três níveis de alinhamento.

Como a função *partição()* está inserida na função *quicksort()*, para encontrarmos a complexidade assintótica *Big-Oh* resultante, multiplicamos a complexidade de ambas funções $O(n^2.\log(n))$. Neste caso, a taxa de crescimento de n^2 é muito superior a $\log(n)$, em qualquer ponto que analisamos. Assim, podemos negligenciar o termo de menor crescimento. A complexidade final para o *quick sort* está na Equação 4.

$$O_{QuickSort}(n^2) \tag{4}$$

A Figura 16 apresenta um gráfico final comparativo de desempenho assintótico *Big-Oh* para os três algoritmos de ordenação apresentados nesta aula. Lembrando que a análise *Big-Oh* se refere ao pior cenário.

Figura 16 – Comparativo gráfico dos três métodos de ordenação, comparando o tempo de execução em função do conjunto de dados de entrada



Ao analisar o gráfico de desempenho da Figura 16, notamos que a complexidade para o pior caso do *quick sort* é inferior em tempo de execução para qualquer ponto, quando comparamos com o *merge sort* e é igual à do *bubble sort*.

Note que isso não significa que o *quick sort* apresenta um desempenho pior que o *merge sort.* O desempenho só será pior quando o algoritmo estiver operando no pior cenário possível $O(n^2)$. Observe a Tabela I, em que há a complexidade em tempo de execução para o pior cenário, o melhor e o cenário médio. Caso consideremos um cenário diferente, o *quick sort* pode se comportar de forma superior aos outros dois.

Por exemplo, para o melhor cenário, o Grande-Omega, vemos que o *bubble sort* passa a ser complexidade *n*. O *quick sort* será *n.log*, igual ao *merge sort*.

Tabela I – Comparativo de complexidade entre os três algoritmos estudados

Algoritmo	Grande-Omega	Grande-Teta	Grande-O
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n.logn)$	$\Theta(n.logn)$	$O(n.logn)$
Quick Sort	$\Omega(n.logn)$	$\Theta(n.logn)$	$O(n^2)$

TEMA 5 – ALGORITMOS DE BUSCA

Algoritmos de busca servem para localizar um valor específico em um conjunto de dados. Assim como apresentado nos algoritmos de ordenação, a quantidade de algoritmos de busca disponível é diversa, e cada um destes algoritmos apresentará vantagens e desvantagens, além de desempenhos distintos e, consequentemente, aplicações diferentes. Neste tema, estudaremos dois algoritmos de busca fundamentais para as funções de manipulação de grandes quantidades de dados: busca sequencial e busca binária.

5.1 Busca sequencial

A busca sequencial tem como fim introduzir o assunto de algoritmos de busca. A proposta do algoritmo de busca sequencial consiste em realizar uma busca em um conjunto de dados (ordenado ou não), empregando ao menos um laço de repetição que fará a varredura do conjunto de dados (lógica iterativa). Para cada iteração, temos uma comparação simples entre o elemento a ser localizado e cada elemento deste conjunto de dados. A Figura 17 ilustra o algoritmo para um vetor não ordenado.

Figura 17 – Pseudocódigo da busca sequencial para vetores não ordenados

```
1   algoritmo "BuscaSequencial_NaoOrdenado"
2   var
3      X[TAMANHOVETOR], i, achou, buscado: inteiro
4   inicio
5         //PREENCHE O VETOR COM DADOS ALEATORIOS
6         para i de 0 até (TAMANHOVETOR - 1) faça
7            leia(X[i])
8         fimpara
9         //LÊ VALOR A SER BUSCADO
10        leia(buscado)
11
12        //BUSCA SEQUENCIAL
13        achou = 0
14        i = 0
15        enquanto ((i <= TAMANHOVETOR) E (achou == 0))
16           se (X[i] == buscado) então
17              achou = 1
18           senão
19              i = i + 1
20           fimse
21        fimenquanto
22        se (achou == 0) então
23           escreva("Valor não encontrado.")
24        senão
25           escreva("Valor encontrado na posição.", i + 1)
26        fimse
27  fimalgoritmo
```

Vamos analisar algumas partes importantes deste código:

- Linhas 6 a 8: lê dados aleatórios de um vetor;
- Linha 10: lê o valor que será buscado no vetor utilizando a busca sequencial;
- Linha 13: variável achou é inicializada com zero. Esta variável irá para nível lógico alto quando o número buscado for localizado no vetor;
- Linha 15: laço de repetição que executará enquanto duas condições forem satisfeitas. O vetor não chegar no seu final e o valor não foi localizado;
- Linha 16: compara o valor desejado com cada posição do vetor. A varredura ocorre da posição zero do vetor até a última posição se assim for necessário;
- Linha 17: coloca em nível alto a variável achou, informando que o valor foi localizado no vetor;
- Linha 19: quando o valor não é localizado em uma posição, incrementa-se o índice de varredura;
- Linhas 22 a 26: imprime na tela se o valor foi encontrar e qual a posição ele foi localizado.

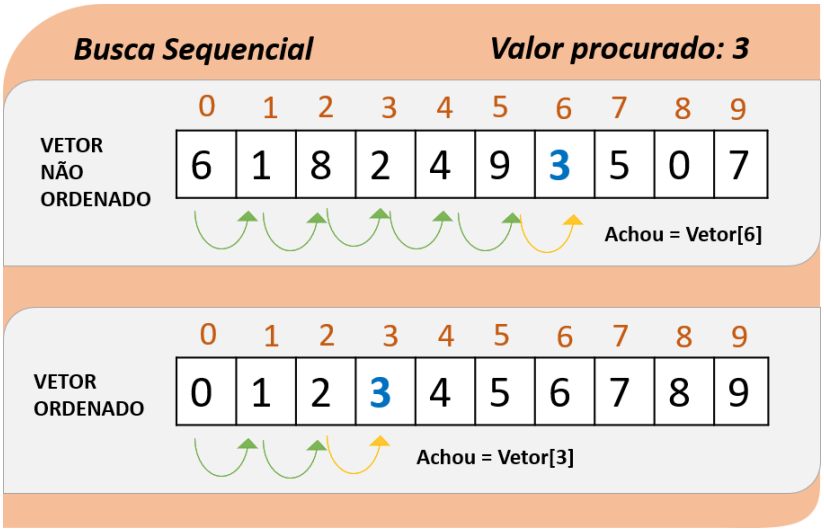
Na Figura 18, temos o mesmo algoritmo, mas agora para um vetor de dados já ordenados. Porém, como o vetor está ordenado, podemos definir mais uma condição de parada (linha 16). Caso os valores testados ultrapassem o valor desejado (ordenação crescente), significa que o valor buscado não existe naquele vetor, e, portanto, a busca pode ser encerra precocemente.

Figura 18 – Pseudocódigo da busca sequencial para vetores ordenados

```
13      //BUSCA SEQUENCIAL PARA VETOR ORDENADO
14      achou = 0
15      i = 0
16      enquanto ((i <= TAMANHOVETOR) E (achou == 0) E (buscado >= X[i]))
17          se (X[i] == buscado) então
18              achou = 1
19          senão
20              i = i + 1
21          fimse
22      fimenquanto
23      se (achou == 0) então
24          escreva("Valor não encontrado.")
25      senão
26          escreva("Valor encontrado na posição.", i + 1)
27      fimse
```

A Figura 19 mostra um comparativo no número de iterações para um vetor não ordenado comparado com um já ordenado. Em ambos os casos estamos buscando o valor 3 dentro de um vetor de tamanho 10. No vetor não ordenado, o valor 3 está localizado mais ao fim do vetor. Como a varredura se inicia na posição zero, são necessárias 7 comparações até atingir o valor 3 e encerrar a busca. Para a situação de um vetor já ordenado crescentemente, o valor está localizado na quarta posição do vetor, economizando algumas iterações na execução do programa.

Figura 19 – Comparação de iterações para um vetor ordenado e um não ordenado usando busca sequencial



É válido ressaltar que, no exemplo da Figura 19, o modo como os dados estão distribuídos em ambos vetores (ordenado e não ordenado) fizeram com que o ordenado tivesse seu valor encontrado primeiro. Mas isso nem sempre é verdade. Se o valor buscado estivesse colocado, randomicamente, na primeira posição do vetor desordenado, ele já seria encontrado na primeira busca, enquanto, no ordenado, somente na iteração 4. A complexidade assintótica para o tempo de execução da busca sequencial, com vetor ordenado ou não, será sempre $O(n)$.

5.2 Busca binária

A busca binária que trabalha com o conceito de dividir um conjunto de dados sempre ao meio e comparar o ponto central com o valor buscado é o conhecido processo de **dividir para conquistar**. Se este ponto central for menor ou maior que o valor buscado, a região de busca do vetor é limitada para o ramo

esquerdo ou direto do conjunto de dados. Com esta proposta, a busca binária é capaz de atingir seu objetivo mais rapidamente e, ainda, trabalhar com recursividade, tornando o método mais eficiente se comparado ao sequencial.

Uma característica importante é que a busca binária só funciona com conjuntos de dados já ordenados, caso contrário, a verificação feita com o ponto central não seria possível. Portanto, se o conjunto de dados não estiver ordenado, ele primeiro precisa ser ordenado para depois ser buscado.

Na Figura 20, temos um exemplo de busca binária em um vetor de tamanho 10, ordenado crescentemente. Queremos buscar o valor 3, vejamos o que ocorre em cada etapa:

- Etapa 1: Ponto central é localizado. Valor central é 4. Compara-se o 4 com 3 (4 > 3), portanto o valor 3 deve estar do lado esquerdo do 4, ou seja, antes dele.
- Etapa 2: agora nosso vetor tem como última posição de busca a posição 4. Portanto, o ponto central entre 0 e 4 será o 2. Na posição 2, temos o valor 2. Este valor é menor do que 3 (3 > 2), portanto o nosso valor desejado está à direita do 2 e a esquerda do 4.
- Etapa 3: agora, nosso vetor tem como primeira posição a 2, e última posição de busca a posição 4. O ponto central será a posição 3, em que o valor 3 está localizado. Sendo assim, na terceira iteração está localizado o valor desejado.

Comparando a busca binária com a busca sequencial em um vetor ordenado, para o exemplo apresentado nas Figuras 19 e 20, a busca binária obteve o valor desejado uma iteração antes da sua concorrente. A busca binária apresenta como complexidade assintótica $O(logn)$.

Figura 20 – Iterações para a busca binária

O pseudocódigo da busca binária é apresentado na Figura 21.

Figura 21 – Pseudocódigo da busca binária

```
1   algoritmo "BuscaBinaria"
2   var
3      X[TAMANHOVETOR], i, achou, buscado: inteiro
4      inicio, fim, meio: inteiro
5   inicio
6         //PREENCHE O VETOR COM DADOS ORDENADOS
7         //OU PREENCHE ALEATORIAMENTE E ORDENA COM ALGUM ALGORITMO
8         para i de 0 até (TAMANHOVETOR - 1) faça
9            leia(X[i])
10        fimpara
11        //LÊ VALOR A SER BUSCADO
12        leia(buscado)
13
14        //BUSCA BINARIA
15        achou = 0
16        inicio = 0
17        fim = TAMANHOVETOR
18        meio = parteinteira((inicio+fim) / 2)
19        enquanto ((inicio <= fim) E (achou == 0))
20           se (X[meio] == buscado) então
21              achou = 1
22           senão
23              se (meio < X[meio]) então
24                 fim = meio  - 1
25              senão
26                 inicio = meio + 1
27                 meio = parteinteira((inicio+fim) / 2)
28              fimse
29           fimse
30        fimenquanto
31        se (achou == 0) então
32           escreva("Valor não encontrado.")
33        senão
34           escreva("Valor encontrado na posição.", meio)
35        fimse
36  fimalgoritmo
```

Vejamos alguns pontos importantes estão listados abaixo:

- Linhas 8 a 10: lê dados já ordenados de um vetor. Caso não estejam ordenados, é necessário ordenar empregando algum dos métodos apresentados nesta Aula 2;
- Linha 12: lê o valor que será buscado no vetor utilizando a busca binária;
- Linha 15: variável achou é inicializada com zero. Esta variável irá para nível lógico alto quando o número buscado for localizado no vetor;
- Linha 16 e 17: inicializa a região de busca. Neste caso, o vetor inteiro será buscado;
- Linha 18: encontra o ponto central do vetor;
- Linha 19: laço de repetição que executará enquanto duas condições forem satisfeitas. O vetor não chegar no seu final e o valor não foi localizado;
- Linha 20: testa se o valor do meio é o valor desejado;
- Linha 21: se o valor do meio é o buscado. Marca-o;
- Linha 23 a 28: se o valor do meio para aquela iteração não é o buscado, delimita-se a região de busca para o lado esquerdo ou direito. Calcula-se um novo ponto central para este vetor restringido;
- Linhas 31 a 35: imprime na tela se o valor foi encontrar e qual a posição ele foi localizado.

FINALIZANDO

Nesta aula, aprendemos alguns dos algoritmos de ordenação de dados. Verificamos o *bubble sort*, algoritmo que funciona como porta de entrada para análise de algoritmos de ordenação com complexidade $O(n^2)$.

Vimos também o *merge sort* e o *quick sort*. Ambos são algoritmos muito empregados em aplicações reais e trabalham com conceitos de recursividade. O *merge sort* tem complexidade $O(n.logn)$ enquanto o *quick sort* contém $O(n^2)$.

Vimos também dois algoritmos de busca. O sequencial, que é um algoritmo iterativo e de baixa eficácia, com complexidade $O(n)$, e o binário, que trabalha com o conceito de dividir para conquistar e, portanto, complexidade para tempo de execução $O(logn)$. Embora com um desempenho superior ao sequencial, este segundo só é capaz de operar em conjuntos de dados obrigatoriamente já ordenados.

REFERÊNCIAS

ASCENCIO, A. F. G. **Estrutura de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson, 2011.

ASCENCIO, A. F. G. **Fundamentos da programação de computadores**: algoritmos, Pascal, C/C++ (padrão ANSI) JAVA. 3. ed. São Paulo: Pearson, 2012.

CORMEN, T. H. **Algoritmos**: teoria e prática. 3. ed. São Paulo: Elsevier, 2012.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson, 2008.

LAUREANO, M. **Estrutura de dados com algoritmos E C**. Rio de Janeiro: Brasport, 2008.