Entendendo Ordenação

Processos de ordenação de números são muito importantes em computação. Existem vários tipos de algoritmos de ordenação, como, por exemplo: insertionsort, selectionsort, bubblesort, shellsort, heapsort, quicksort, mergesort, radixsort, entre outros.

Usamos uma notação matemática especial para representar o desempenho de algoritmos. A diferença está no tempo de execução. O pior deles leva tempo $O(n^2)$ e o melhor deles leva tempo O(n). Quanto mais rápido cresce a função (n^2. cresce mais rápido do que n) pior ela é, porque requer um maior número de operações para uma entrada de n elemento. Entretanto, os recursos necessários para o desenvolvimento desses algoritmos também mudam. Os algoritmos de tempo O(n) utilizam menos recursos computacionais em relação aos que levam tempo quadrático $O(n^2)$.

Em muitas aplicações, os elementos precisam ser ordenados. E, para ordenar, você precisa rearranjar um conjunto de informações numa disposição que pode ser crescente ou decrescente. O principal objetivo de se ter os dados ordenados é a facilidade posterior de se realizar buscas.

Imagine procurar uma palavra em um dicionário físico, porém de forma que as palavras foram colocadas aleatoriamente. Ficaria muito difícil de encontrar a palavra, concorda? Por isso, no dicionário, as palavras estão ordenadas alfabeticamente. As comparações são poucas e a busca muito rápida, mesmo havendo milhares de palavras de entrada, representadas por n.

BUBBLESORT

O bubblesort ou ordenação por bolhas é assim chamado porque tem a ideia de que os números maiores vão flutuando como bolhas até chegar na última posição, ou em suas posições corretas. Na computação, o bubblesort é um dos algoritmos de ordenação mais utilizados, por ser fácil de entender, desenvolver e de se usar.

Você vai comparando elementos dois a dois e quando o primeiro número é maior do que o segundo, acontece a troca dos elementos. Dessa forma, a comparação acontece do primeiro com o segundo elemento, do segundo com o terceiro elemento, do terceiro com o quarto, e assim sucessivamente até o penúltimo com o último elemento. Isso garante que o último elemento será o maior.

O processo continua, porém, agora considera a lista do primeiro até o penúltimo elemento, pois o último já está ordenado. O processo termina quando sobrar apenas um elemento para ordenar. Nesse caso, a lista toda já estará ordenada.

Para exemplificar, vamos considerar os valores inteiros relacionados a seguir:

26 47 38 11 95

26 47 38 11 95 compara 26 e 47

26 47 38 11 95 compara 47 e 38

26 38 47 11 95 compara 47 e 11

26 38 11 47 95 compara 47 e 95

26 38 11 47 95 final do primeiro processo

26 38 11 47 95 compara 26 e 38

26 38 11 47 95 compara 38 e 11

26 11 38 47 95 compara 38 e 47

26 11 38 47 95 final do segundo processo

26 11 38 47 95 compara 26 e 11

11 26 38 47 95 compara 26 e 38

11 26 38 47 95 final do terceiro processo

11 26 38 47 95 compara 11 e 26

11 26 38 47 95 final do último processo

11 26 38 47 95 final do último processo

O algoritmo do BubbleSort está a seguir:

```
Bolha (numeros [] inteiro)
   início_módulo
    Declarar
     constante n ← numeros.tamanho inteiro;
     aux, i, j inteiro;
```

```
    para i de 0 até n-2 passo +1 faça
      para j de 0 até n-2-i passo +1 faça
        se (numeros[j] > numeros[j+1])
          então
            aux ← numeros[j];
            numeros[j] ← numeros[j+1];
            numeros[j+1] ← aux;
        fimse;
      fimpara;
    fimpara;
fim_módulo;
```

```
Bolha (numeros [] inteiro)
    início_módulo
     Declarar
      constante n ← numeros.tamanho inteiro;
      aux, i, j inteiro;

      para i de 0 até n-2 passo +1 faça
         para j de 0 até n-2-i passo +1 faça
            se (numeros[j] > numeros[j+1])
              então
                aux ← numeros[j];
                numeros[j] ← numeros[j+1];
                numeros[j+1] ← aux;
            fimse;
          fimpara;
       fimpara;
fim_módulo;
```

QUICKSORT

O QuickSort é o algoritmo de ordenação por troca de partição. O Quicksort usa a ideia de uma ordenação dos elementos por meio de partições.

No Quicksort, a lista é dividida em sublistas. Essas sublistas são as partições. Cada uma das partições é dividida novamente em outras partições até que a divisão não tem mais como acontecer. Nesse caso, a partição vai ter apenas um elemento.

A ideia do Quicksort é dividir o problema em problemas menores e depois conquistar para que possam ser solucionados de forma mais fácil e rápida.

O Pivô é o elemento escolhido de dentro da lista, normalmente, é o primeiro elemento. Ele ajuda a determinar as partições, de forma que os elementos da primeira partição são menores ou iguais ao pivô, depois vem o pivô, e os elementos da segunda lista são maiores do que o pivô.

Para exemplificar, vamos considerar os valores inteiros relacionados a seguir:

25 57 48 37 12 92 86 33

(12) 25 (57 48 37 92 86 33) pivô 25

12 25 (57 48 37 92 86 33) pivô 12

12 25 (48 37 33) 57 (92 86) pivô 57

12 25 (37 33) 48 57 (92 86) pivô 48

12 25 (33) 37 48 57 (92 86) pivô 37

12 25 33 37 48 57 (92 86) pivô 33

12 25 33 37 47 57 (86) 92 pivô 92

12 25 33 37 47 57 86 92 pivô 86

12 25 33 37 47 57 86 92 lista ordenada

O algoritmo do quicksort está a seguir:

```
quicksort(p inteiro, q inteiro, vetor[] inteiro)
  início_módulo
```

```
  Declarar

   x inteiro;



   se (p < q)

     então

       x ← particao(p, q, vetor);

       quicksort(p, x - 1, vetor);

       quicksort(x + 1, q, vetor);

    fimse;

fimMódulo;



inteiro particao(p inteiro, q inteiro, vetor[] inteiro)

 início_módulo

  Declarar

   j ← p – 1inteiro;

   temp, i, aux ← vetor[q] inteiro;



 para i de p até q passo +1 faça

   se (vetor[i] <= aux)
```

```
    então
      j ← j +1;
      temp ← vetor[i];
      vetor[i] ← vetor[j];
      vetor[j] ← temp;
   fimse;
 fimpara;
 retornar j;
fimmódulo;
```

MERGESORT

O MergeSort é um algoritmo de ordenação por intercalação, porque utiliza a ideia de uma ordenação dos elementos por meio da união de elementos já ordenados.

No MergeSort, a ideia é unir duas metades que já estão ordenadas de duas sublistas de informações, e a partir disso gerar uma lista ordenada de dados. As sublistas de informações precisam estar ordenadas primeiro. Neste caso, essas sublistas foram geradas de outras sublistas de informações que já estavam ordenadas. Essa subdivisão acontece até que a sublista tenha menos do que dois elementos.

Quando você tem uma lista de informações, a primeira sublista estará com a primeira metade da lista de informações e a segunda sublista com a segunda metade da lista de informações.

A ideia do MergeSort é dividir o problema em problemas menores e depois conquistar para que possam ser solucionados de forma mais fácil e rápida.

Para exemplificar, vamos considerar os valores inteiros relacionados a seguir:

25 57 48 37 12 92 86 33

Divide cada sublista em duas metades

(25 57 48 37 12 92 86 33)

(25 57 48 37) (12 92 86 33)

(25 57) (48 37) (12 92 86 33)

(25) (57) (48 37) (12 92 86 33)

(25) (57) (48) (37) (12 92 86 33)

(25) (57) (48) (37) (12 92) (86 33)

(25) (57) (48) (37) (12) (92) (86 33)

(25) (57) (48) (37) (12) (92) (86) (33)

Conquista intercalando as listas, duas a duas, ordenando os elementos parciais das sublistas

(25) (57) (48) (37) (12) (92) (86) (33)

(25) (57) (48) (37) (12) (92) (33 86)

(25) (57) (48) (37) (12 92) (33 86)

(25) (57) (48) (37) (12 33 86 92)

(25) (57) (37 48) (12 33 86 92)

(25 57) (37 48) (12 33 86 92)

(25 37 48 57) (12 33 86 92)

(12 25 33 37 48 57 86 92)

12 25 33 37 48 57 86 92 lista ordenada

O algoritmo do MergeSort está a seguir:

```
 merge(a[] numérico_inteiro, inicio numérico_inteiro, meio numérico_inteiro, fim
numérico_inteiro)

  início_módulo

    Declarar

      n, b[n], i1, i2, j ← 0 númerico_inteiro;



    n ← fim - inicio + 1;

    i1 ← inicio;
```

```
i2 ← meio + 1;

enquanto (i1 <= meio e i2 <= fim) faça
  se (a[i1] < a[i2])
   então
     b[j] ← a[i1];
     i1 ← i1 + 1;
   senão
     b[j] ← a[i2];
     i2 ← i2 + 1;
  fimse;
  j ← j + 1;
fimenquanto;

enquanto (i1 <= meio) faça
  b[j] ← a[i1];
  i1 ← i1 + 1;
  j ← j + 1;
fimenquanto;
```

```
  enquanto (i2 <= fim) faça
    b[j] ← a[i2];
    i2 ← i2 + 1;
    j ← j + 1;
  fimenquanto;

  para j de 0 até n-1 passo +1 faça
    a[inicio + j] ← b[j];
  fimpara;
fimmódulo;
```

IMPLEMENTANDO ORDENAÇÕES NO JAVA

A linguagem de programação que estamos utilizando é a linguagem Java.

Lembre-se de que você deve deixar seu programa bem alinhado e identado para que, posteriormente, você consiga entender e dar suporte para o programa.

A documentação é uma parte importante quando você desenvolve um programa. Faça sempre comentários no seu código.

Segue as ordenações na linguagem Java:

BUBBLE SORT

```
public static void Bolha (int numeros [])
  {
    final int n = numeros.length;
    int aux;

    for (int i = 0 ; i < n-1 ; i++)
     {
       for (int j = 0 ; j < n-1-i ; j++)
        {
          if (numeros[j] > numeros[j+1])
          {
            aux = numeros[j];
            numeros[j] = numeros[j+1];
            numeros[j+1] = aux;
          }
```

```
    }
  }
}
```

QUICKSORT

```
public static void quicksort(int p, int q, int vetor[])
{
  int x;

  if (p < q)
  {
    x = particao(p, q, vetor);
    quicksort(p, x - 1, vetor);
    quicksort(x + 1, q, vetor);
  }
}

public static int particao(int p, int q, int vetor[])
```

```
{
  int j = p - 1;
  int temp, aux = vetor[q];

  for (int i = p; i <= q; i++)
  {
    if (vetor[i] <= aux)
    {
      j++;
      temp = vetor[i];
      vetor[i] = vetor[j];
      vetor[j] = temp;
    }
  }
  return j;
}
```

MERGESORT

```
public static void merge(int[] a, int inicio, int meio, int fim)
 {
  int n = fim - inicio + 1;
  int[] b = new int[n];

  int i1 = inicio;
  int i2 = meio + 1;
  int j = 0;

  while (i1 <= meio && i2 <= fim)
   {
    if (a[i1] < a[i2])
     {
      b[j] = a[i1];
      i1++;
     }
    else
     {
      b[j] = a[i2];
```

```
    i2++;
  }
  j = j + 1;
}

while (i1 <= meio)
{
  b[j] = a[i1];
  i1 = i1 + 1;
  j = j + 1;
}

while (i2 <= fim)
{
  b[j] = a[i2];
  i2 = i2 + 1;
  j = j + 1;
}
```

```
  for (j = 0; j < n; j++)
  {
    a[inicio + j] = b[j];
  }
}

public static void mergeSort(int[] a, int inicio, int fim)
{
  if (inicio == fim)
  {
    return;
  }
  int meio = (inicio + fim) / 2;
  mergeSort(a, inicio, meio);
  mergeSort(a, meio + 1, fim);
  merge(a, inicio, meio, fim);
}
```

Atividade extra

Indicação de leitura: **Estrutura de Dados: algoritmos, análise da complexidade e implementações em Java e C/C++**, da Ana Fernanda Gomes Ascencio e Graziela Santos de Araújo, capítulo 2.

Referência Bibliográfica

PUGA, S.; RISSETTI, G. **Lógica de Programação e Estruturas de Dados, com aplicações em Java.** São Paulo. Editora Pearson. 3ª. Edição. 2016.

FORBELLONE, A.L.V.; EBERSPACHER, H.F. **Lógica de Programação: a construção de algoritmos e estruturas de dados.** 3ª Edição. São Paulo. Prentice Hall. 2005.

Atividade Prática 03 – Entendendo ordenação

Título da Prática: Ordenação BubbleSort

Objetivos: Entender como utilizar o netbeans para desenvolver programas em Java para ordenação BubbleSort

Materiais, Métodos e Ferramentas: Computador, netbeans, Java.

Atividade Prática

O bubblesort ou ordenação bolha é assim chamado porque tem a ideia de que os números maiores vão flutuando como bolhas até chegar na última posição, ou em suas posições corretas.

Você vai comparando elementos dois a dois e quando o primeiro número é maior do que o segundo, acontece a troca dos elementos. Dessa forma, a comparação acontece do primeiro com o segundo elementos, do segundo com o terceiro elementos, do terceiro com o quarto e assim sucessivamente até o penúltimo com o último elementos. Isso garante que o último elemento será o maior.

O processo vai continuando, porém, agora considerando a lista do primeiro até o penúltimo elemento. Pois o último já está ordenado. E o processo termina quando sobrar apenas um elemento para ordenar. Nesse caso, a lista toda já estará ordenada.

O Algoritmo de ordenação BubbleSort pode ser escrito como segue.

```
Bolha (numeros [] inteiro)
    início_módulo
```

```
Declarar

 constante n ← numeros.tamanho inteiro;

 aux, i, j inteiro;



 para i de 0 até n-2 passo +1 faça

   para j de 0 até n-2-i passo +1 faça

     se (numeros[j] > numeros[j+1])

       então

         aux ← numeros[j];

         numeros[j] ← numeros[j+1];

         numeros[j+1] ← aux;

     fimse;

   fimpara;

 fimpara;

fim_módulo;
```

Desenvolva o programa em Java deste algoritmo no NetBeans.

Gabarito Atividade Prática

```
import javax.swing.*;


class OrdenacaoBolha
{


  public static void Bolha (int numeros [])
  {
    final int n = numeros.length;
    int aux;


    for (int i = 0 ; i < n-1 ; i++)
    {
      for (int j = 0 ; j < n-1-i ; j++)
      {
        if (numeros[j] > numeros[j+1])
        {
          aux = numeros[j];
```

```
            numeros[j] = numeros[j+1];

            numeros[j+1] = aux;

          }

        }

    }

}


public static void main (String arg [])

{

  int num [] = new int [10];


  for (int i = 0 ; i < 10 ; i++)

   {

     String s;

     s = JOptionPane.showInputDialog("Digite número inteiro");

     num[i] = Integer.parseInt(s);

   }


  Bolha(num);
```

```
    String s = “”;

    for (int i = 0 ; i < num.length ; i++)

     {

       s = s + num[i] + " ";

     }

    JOptionPane.showMessageDialog(null,s);



    System.exit(0);

  }

}
```

Ir para exercício