Ordenação *(Sorting)*

Sistemas de Informação – Univás

Roberto Ribeiro Rocha

rrocha.roberto@gmail.com

Junho de 2022

Sumário

1 Introdução

Em muitas aplicações, os dados devem ser armazenados em uma determinada ordem. Vários algoritmos usam dados ordenados para operar de maneira mais eficiente. Assim, temos basicamente duas opções para ordenar as informações:

- Inserção dos elementos respeitando a ordem;
- Aplicar um algoritmo de ordenação a partir de um conjunto de dados já criado (desordenado).

Um ponto importante é usarmos **Algoritmos de Ordenação** que sejam eficientes, em dois aspectos:

- tempo (rápidos e espertos)
- espaço (ocupam pouca memória na execução)

Uma forma de ter os dados ordenados é utilizando uma Árvore Binária de Busca, e fazendo o percurso em-ordem. Além desta, existem outras formas de fazermos a ordenação dos dados.

O cenário que usaremos para discutir os algoritmos de ordenação é o seguinte:

- A entrada é um vetor de elementos (desordenados) que precisam ser ordenados;
- A saída é o mesmo vetor com seus elementos na ordem especificada;
- O espaço que pode ser utilizado é apenas o espaço do próprio vetor.

Um ponto essencial na ordenação de elementos é a **comparação**. A comparação é usada para tomar a decisão se um elemento irá ou não mudar de lugar dentro do vetor.

Os algoritmos de ordenação podem ser aplicados a qualquer informação, desde que exista uma **ordem** bem definida entre os elementos. Assim deve existir uma função de comparação entre estes elementos, na qual seja retornado um valor indicando a ordemm relativa dos dois objetos.

Nos casos em que a informação é complexa, cada elemento do vetor contém apenas uma referência (ponteiro) para a informação. Assim a ordenação pode ser feita sem necessidade de mover os valores em si, mas movendo somente as referências dentro do vetor.

Este material mostra os métodos de ordenação mais comuns utilizados.

2 *Bubble Sort* – Ordenação bolha

Foi dado este nome, pela imagem usada em sua descrição: *os elementos maiores são mais "leves", e sobem como bolhas até suas posições corretas* ou *fazer os elementos mais "leves" borbulharem para cima.*

A ideia fundamental é fazer uma série de comparações entre os elementos do vetor. Quando dois elementos estão fora de ordem, há uma troca de posição entre eles.

Veja estas animações que ilustram o que ocorre:

`https://www.youtube.com/watch?v=Cq7SMsQBEUw`

`https://www.youtube.com/watch?v=lyZQPjUT5B4`

Note como os valores maiores caminham para o final do vetor. Quando o processo termina, o vetor estará ordenado.

2.1 Detalhes do funcionamento

Os elementos são comparados da seguinte maneira:

O 1^o. elemento é comparado com o 2^o. Se eles estiverem na ordem incorreta, a troca é feita.

Em seguida, o 2^o elemento é comparado com o 3^o e a verificação é feita novamente. O processo continua até o penúltimo elemento seja comparado com o último. Assim, garante-se que o elemento de maior valor do vetor será levado para a última posição.

A ordenação continua, levando o segundo maior elemento o penúltimo lugar do vetor, o terceiro, etc., até que o vetor esteja ordenado.

Exemplo: Considerar o vetor de inteiros: 25 48 37 12 57 86 33 92

Seguido os passos indicados:

$$
\begin{array}{llll}
25\ 48\ 37\ 12\ 57\ 86\ 33\ 92 & \rightarrow & 25 \times 48 & \\
25\ 48\ 37\ 12\ 57\ 86\ 33\ 92 & \rightarrow & 48 \times 37 & \rightarrow\ troca \\
25\ 37\ 48\ 12\ 57\ 86\ 33\ 92 & \rightarrow & 48 \times 12 & \rightarrow\ troca \\
25\ 37\ 12\ 48\ 57\ 86\ 33\ 92 & \rightarrow & 48 \times 57 & \\
25\ 37\ 12\ 48\ 57\ 86\ 33\ 92 & \rightarrow & 57 \times 86 & \\
25\ 37\ 12\ 48\ 57\ 86\ 33\ 92 & \rightarrow & 86 \times 33 & \rightarrow\ troca \\
25\ 37\ 12\ 48\ 57\ 33\ 86\ 92 & \rightarrow & 86 \times 92 & \\
25\ 37\ 12\ 48\ 57\ 33\ 86\ \mathbf{92} & \rightarrow & final\ da\ 1^a\ passada &
\end{array}
$$

Neste ponto, o maior elemento 92, já está na sua posição correta.

$$
\begin{array}{llll}
25\ 37\ 12\ 48\ 57\ 33\ 86\ \mathbf{92} & \rightarrow & 25 \times 37 & \\
25\ 37\ 12\ 48\ 57\ 33\ 86\ \mathbf{92} & \rightarrow & 37 \times 12 & \rightarrow\ troca \\
25\ 12\ 37\ 48\ 57\ 33\ 86\ \mathbf{92} & \rightarrow & 37 \times 48 & \\
25\ 12\ 37\ 48\ 57\ 33\ 86\ \mathbf{92} & \rightarrow & 48 \times 57 & \\
25\ 12\ 37\ 48\ 57\ 33\ 86\ \mathbf{92} & \rightarrow & 57 \times 33 & \rightarrow\ troca \\
25\ 12\ 37\ 48\ 33\ 57\ 86\ \mathbf{92} & \rightarrow & 57 \times 86 & \\
25\ 12\ 37\ 48\ 33\ 57\ \mathbf{86}\ \mathbf{92} & \rightarrow & final\ da\ 2^a\ passada &
\end{array}
$$

Neste ponto, o 2^o maior elemento 86, já está na sua posição correta.

25 12 37 48 33 57 **86 92** → 25×12 → *troca*
12 25 37 48 33 57 **86 92** → 25×37
12 25 37 48 33 57 **86 92** → 37×48
12 25 37 48 33 57 **86 92** → 48×33 → *troca*
12 25 37 33 48 57 **86 92** → 48×57
12 25 37 33 48 **57 86 92** → *final da* 3^a *passada*

O 57 ficou na sua posição final.

12 25 37 33 48 **57 86 92** → 12×25
12 25 37 33 48 **57 86 92** → 25×37
12 25 37 33 48 **57 86 92** → 37×33 → *troca*
12 25 33 37 48 **57 86 92** → 37×48
12 25 33 37 **48 57 86 92** → *final da* 4^a*. Passada*

O 48 ficou na sua posição final.

12 25 33 37 **48 57 86 92** → 12×25
12 25 33 37 **48 57 86 92** → 25×33
12 25 33 37 **48 57 86 92** → 33×37
12 25 33 **37 48 57 86 92** → *final da* 5^a*. passada*

O 37 ficou na sua posição final.

12 25 33 **37 48 57 86 92** → 12×25
12 25 33 **37 48 57 86 92** → 25×33
12 25 **33 37 48 57 86 92** → *final da* 6^a*. Passada*

O 33 ficou na sua posição final.

12 25 **33 37 48 57 86 92** → 12×25
12 **25 33 37 48 57 86 92** → *final da* 7^a*. Passada*

O 25 ficou na sua posição final.

12 25 33 37 48 57 86 92 → *final da ordenação*

O 12 ficou na sua posição final.

Percebam que após a troca 37×33 (4^a passada), o vetor já se encontra totalmente ordenado (mas a versão inicial do algoritmo ainda não trata este caso) e ele continua comparando (sem trocar) até chegar ao final do processo.

Assim, a seguinte classe possui um método que implementa esse algoritmo:

```
public class OrdenacaoBolha {
```

```
  public void ordenarBolhaV1(int [] vetor) {
    int n = vetor.length;
    int i, j;
    for (i=n-1; i >= 1; i--) {
      for (j=0; j<i; j++) {
        if (vetor[j] > vetor[j+1]) { //caso estiver fora de ordem

          //faz a troca dos elementos
          int temp = vetor[j];
          vetor[j] = vetor[j+1];
          vetor[j+1] = temp;
        }
      }
    }
  }
}
```

Exercício 1 – Testar o método de ordenação V1 utilizando o seguinte código de *main*:

```
/* executa o algoritmo de ordenacao bolha */
int vetor[] = new int [] {25, 48, 37, 12, 57, 86, 33, 92};
new OrdenacaoBolha().ordenarBolhaV1(vetor);
System.out.print("Vetor ordenado: ");
for (int valor: vetor) {
  System.out.print(" " + valor);
}
```

Observação: Para evitar que processo continue mesmo depois do vetor estar ordenado, podemos interromper o processo quando houver uma passagem inteira sem trocas, usando uma variante do algoritmo anterior:

```
/* ordenacao bolha versao 2 */
public void ordenarBolhaV2(int [] vetor) {
  int n = vetor.length;
  int i, j;
  for (i=n-1; i >= 1; i--) {
    boolean troca = false; //indica se houve ou não troca em uma passada
    for (j=0; j<i; j++) {
      if (vetor[j] > vetor[j+1]) { //caso estiver fora de ordem

        //troca os elementos
        int temp = vetor[j];
        vetor[j] = vetor[j+1];
        vetor[j+1] = temp;
        troca = true;
      }
    }
    if (!troca) { /* nao houve troca */
      return;
    }
  }
}
```

2.2 Usando `Comparators`

Esse mesmo algoritmo pode ser reaproveitado para ordenar vetores que guardam outras informações, inclusive com objetos. Para isso é necessário alguns ajustes:

- Criar uma classe com um método para fazer a comparação da informação desejada.
- Alterar a assinatura do método para receber um objeto *comparador*.
- Alterar o código que faz a comparação para usar este novo objeto *comparador*.

Assim, vamos re-escrever o algoritmo de ordenação tornando-o independente da informação armazenada no vetor, aumentando seu potencial de reuso.

A classe com o método de comparação ficaria assim:

```
public class InteiroComparator implements Comparator<Integer> {
  @Override
  public int compare(Integer a, Integer b) {

    if (a < b) {
      return -1;
    } else if (a > b) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Veja a documentação oficial para mais informações: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html`

Nossa função de ordenação agora usa um objeto de comparação:

```
  /* ordenacao bolha versão 3 */
  public void ordenarBolhaV3(int[] vetor, InteiroComparator cmp) {
    int n = vetor.length;
    int i, j;
    for (i = n - 1; i >= 1; i--) {
      int troca = 0;
      for (j = 0; j < i; j++)
        if (cmp.compare(vetor[j], vetor[j + 1]) > 0) { // caso estiver fora de ordem
          int temp = vetor[j];
          vetor[j] = vetor[j + 1];
          vetor[j + 1] = temp;
          troca = 1;
        }
      if (troca == 0) { // nao houve troca
        return;
      }
    }
  }
```

O uso de *comparators* permite separar a lógica de comparação da lógica de ordenação.

Agora, caso quisermos ordenar inversamente, basta inverter a lógica de comparação dentro do *comparator*. A ideia é ter uma função de comparação que recebe dois elementos e verifica a relação de ordem entre eles, mas agora de forma invertida.

Exercício 2 – Fazer o teste utilizando um `Comparator` para ordenar o vetor de duas maneiras diferentes:

- De maneira crescente e
- De maneira decrescente.

2.3 Ordenando vetores de Objetos

Utilizando a técnica do `Comparator`, podemos ordenar um vetor de qualquer tipo de objeto.

Vejamos como poderíamos ordenar um vetor de alunos, utilizando a seguinte definição da classe Aluno:

```
class Aluno {
  String nome;
  Integer matricula;
  String email;
}
```

Uma função de comparação, neste caso, receberia como parâmetros duas referências para `Aluno` e, considerando uma ordenação que usa a matrícula do aluno como chave de comparação, poderia ser a seguinte implementação:

```
class AlunoComparator implements Comparator<Aluno> {

  public int compare(Aluno a, Aluno b) {
    if (a.matricula < b.matricula) {
      return -1;
    } else if (a.matricula > b.matricula) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Esta solução permite, novamente, separar a lógica de comparação da lógica de ordenação.

Exercício 3 – (Para casa) Implementar um método para ordenar um vetor de alunos com a seguinte assinatura:

`ordenarBolhaV4(Aluno[] vetor, Comparator<Aluno> cmp)`

Implementar um `main` para testar o método.

Observação: caso o objeto seja mais complexo, basta fazer a implementação de comparação adequada dentro do *comparator*, por exemplo, comparando dois ou mais atributos.

2.4 A versão genérica do algoritmo

Agora faremos uma última alteração para deixar o método genérico o suficiente para receber um vetor qualquer objeto e um comparador também genérico.

Então na assinatura da classe, definimos um tipo genérico:

```
public class OrdenacaoBolha<T> { //repare o tipo genérico T
...
}
```

E na assinatura do método utilizamos este tipo genérico:

```
  /* ordenacao bolha versão 5 */
  public void ordenarBolhaV5(T[] vetor, Comparator<T> cmp) { //repare o tipo genérico T
    int n = vetor.length;
    int i, j;
    for (i = n - 1; i >= 1; i--) {
      int troca = 0;
      for (j = 0; j < i; j++)
        if (cmp.compare(vetor[j], vetor[j + 1]) > 0) { // caso estiver fora de ordem
          T temp = vetor[j]; //repare o tipo genérico T
          vetor[j] = vetor[j + 1];
          vetor[j + 1] = temp;
          troca = 1;
        }
      if (troca == 0) { // nao houve troca
        return;
      }
    }
  }
```

Assim, o algoritmo de ordenação será o mesmo, porém usando comparadores diferentes para diferentes tipos de dados.

Logo o método `main` para esta última versão poderia ser da seguinte forma (repare a *inner class*):

```
String vetor[] = new String[] { "25", "48", "37", "12", "57", "86", "33", "92" };
Comparator<String> stringComparator = new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return s1.compareTo(s2);
  }
};
OrdenacaoBolha<String> ord = new OrdenacaoBolha<>();
ord.ordenarBolhaV5(vetor, stringComparator);
System.out.print("Vetor ordenado: ");
for (String valor : vetor) {
  System.out.print(" " + valor);
}
```

Uma otimização que podemos fazer no código é substituir a *inner class* por uma *lambda expression*. Assim a declaração e inicialização do comparator ficaria assim:

```
Comparator<String> stringComparator = (s1, s2) -> s1.compareTo(s2);
```

E o restante do código do `main` continuaria da mesma forma.

Essa solução é bastante robusta e a ordenação é genérica o suficiente para podermos utilizar o mesmo método (V5) para quaisquer objetos, bastando implementar a interface `Comparator` para comparar os diferentes objetos.

Exercício 4 – Implementar um comparador de objetos `String`, e fazer o teste para ordenar um vetor de `String`.

Exercício 5 – (Extra) Fazer o teste de mesa do algoritmo *Bubble Sort*.

3 *Quick Sort* – Ordenação rápida

O algoritmo *quick sort* procura resolver o problema por partes. Ele usa a seguinte ideia: **dividir para conquistar** → problemas menores são mais fáceis de resolver.

O *quick sort* utiliza um elemento (do vetor) arbitrário, chamado **pivô**, onde todos elementos menores que ele devem ficar à sua esquerda e todos os elementos maiores devem ficar à sua direita. Para facilitar a implementação, o pivô será o primeiro elemento do vetor (mas pode ser qualquer elemento).

O algoritmo utiliza dois índices i e j. O i inicia uma posição após o pivô e caminha da esquerda para a direita. O j inicia na última posição e caminha em sentido contrário. Em cada passo, é feira a **comparação** destes dois valores em relação ao pivô. A **troca** dos elementos somente é feita caso aconteça as duas condições abaixo:

- O valor da esquerda é maior que o pivô.
- O valor da direita é menor que o pivô.

Caso o valor da esquerda é menor que o pivô, nós incrementamos o índice i.

Caso o valor da direita é maior que o pivô, nós decrementamos o índice j.

Esses passos são executados até o i ultrapassar o j.

Por último trocamos o pivô com o elemento da posição j.

Assim, na primeira passada, nós colocaremos o pivô em sua posição correta no vetor, com os menores à sua esquerda e os maiores à sua direita, realizando **trocas à distância**.

O próximo passo é ordenar **recursivamente** o sub-vetor da esquerda e o sub-vetor da direita, selecionando um pivô para cada sub-vetor e colocando-o em sua posição correta novamente.

Assim, recursivamente, chegaremos a um sub-vetor com uma posição apenas, que por si só, já estará ordenado, ou seja, todas as posições do vetor já estarão em seus respectivos lugares, obtendo assim o vetor ordenado.

3.1 Exemplo genérico

Suponhamos que o elemento x (pivô) deva ocupar a posição i do vetor, quando este já estiver ordenado.

Com a primeira execução do algoritmo, este fato ocorre quando todos os elementos $v[0], \ldots, v[i-1]$ são menores que x, e todos os elementos $v[i+1], \ldots, v[n-1]$ são maiores que x.

Com x na posição correta, com índice i, temos que ordenar os sub-vetores $v[0], \ldots, v[i-1]$ e $v[i+1], \ldots, v[n-1]$.

$v[0], \ldots, v[i-1]$							vet[i]	$v[i+1], \ldots, v[n-1]$										
a	b	c	d	e	f	g	x	i	j	k	l	m	n	o	p	q	r	s
menores que pivô ⟵							pivô	⟶ maiores que pivô										

Esses sub-problemas são resolvidos (recursivamente) de forma semelhante, cada vez com vetores menores até cada "sub-vetor" ficar com apenas 1 elemento.

Da mesma forma que fazemos as trocas com o vetor original, iremos fazer trocas usando os "sub-vetores"

3.2 Exemplo prático

Veja esta animação que ilustra o que ocorre:

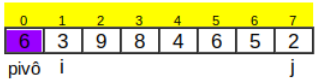
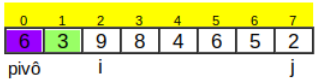
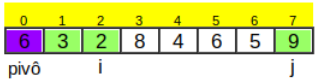
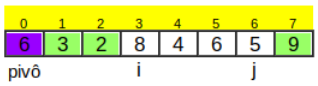
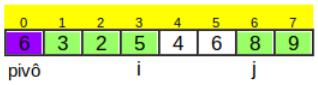
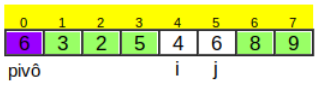
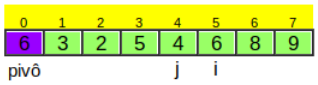
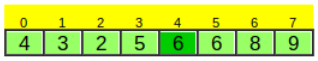
`https://www.youtube.com/watch?v=ywWBy6J5gz8`

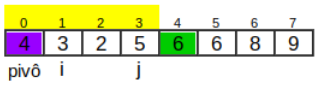
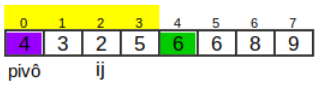
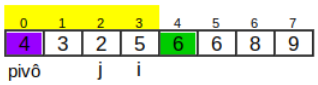
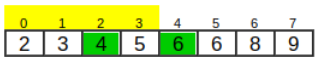
Veja uma outra forma de caminhar através dos índices, que também funciona:

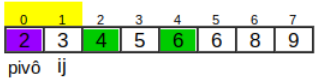
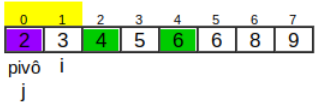
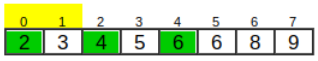
`https://www.youtube.com/watch?v=cnzIChso3cc`

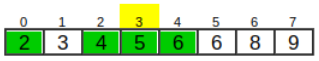
A estratégia ainda continua em colocar os elementos menores que o pivô à sua esquerda e os maiores à sua direita, lembrando... de maneira recursiva até o vetor conter apenas 1 elemento.

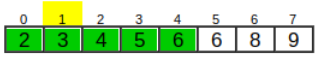
Veja um exemplo da execução do algoritmo em um vetor de números inteiros:

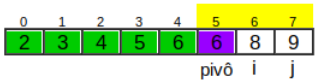
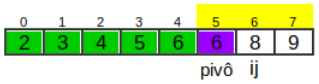
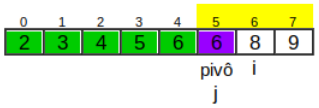
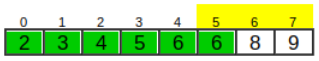
Contexto recursivo	Iteração	vetor	Ação
ctx1 **ini=0** **fim=7**	0		
ctx1	1		
ctx1	2		→ trocou o 2 com o 9
ctx1	3		
ctx1	4		→ trocou o 8 com o 5
ctx1	5		
ctx1	6		
ctx1	7		→ trocou o 6 com o 4

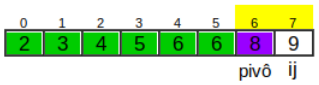
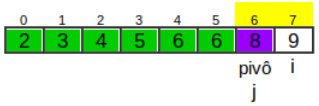
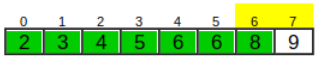
Contexto recursivo	Iteração	vetor	Ação
ctx2 **ini=0** **fim=3**	0		
ctx2	1		
ctx2	2		
ctx2	3		→ trocou o 4 com o 2

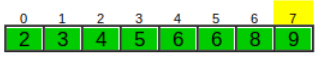
Contexto recursivo	Iteração	vetor	Ação
ctx3 **ini=0** **fim=1**	0		
ctx3	1		
ctx3	2		→ trocou o 2 (pivô) com o 2 (ele mesmo)

Contexto recursivo	Iteração	vetor	Ação
ctx4 **ini=3** **fim=3**	0		→ o vetor só tem 1 elemento, já está ordenado

Contexto recursivo	Iteração	vetor	Ação
ctx5 **ini=1** **fim=1**	0		→ o vetor só tem 1 elemento, já está ordenado

Contexto recursivo	Iteração	vetor	Ação
ctx6 **ini=5** **fim=7**	0		
ctx6	1		
ctx6	2		
ctx6	3		→ trocou o 6 (pivô) com o 6 (ele mesmo)

Contexto recursivo	Iteração	vetor	Ação
ctx7 **ini=6** **fim=7**	0		
ctx7	1		
ctx7	2		→ trocou o 8 (pivô) com o 8 (ele mesmo)

Contexto recursivo	Iteração	vetor	Ação
ctx8 **ini=7** **fim=7**	0		→ o vetor só tem 1 elemento, já está ordenado

3.3 O algoritmo

Dado um vetor de n elementos, o algoritmo *quick sort* pode ser implementado utilizando as seguintes instruções:

O *Quick Sort* é o algoritmo de ordenação mais utilizado no desenvolvimento de aplicações, devido sua eficiência.

Algoritmo 1: Quick Sort

```
Entrada: Vetor vet[], int ini, int fim
se ini < fim então // se o sub-vetor possui mais de 1 elemento
    pivo ← vet[ini]
    i ← ini + 1 // incrementa, pois o primeiro é o pivô
    j ← fim
    enquanto i ≤ j faça
        enquanto i ≤ fim E vet[i] < pivo faça i++

        enquanto j > ini E vet[j] > pivo faça j--

        se i < j então // faz a troca de vet[i] com vet[j]
            troca elemento o elemento i com o j
            i++
            j--
    troca elemento j com o elemento pivô no vetor
    quickSort(vet, ini, j-1)// ordena recursivamente o sub-vetor da esquerda
    quickSort(vet, j+1, fim)// ordena recursivamente o sub-vetor da direita
```

Porém, um item importante no algoritmo é a **escolha do pivô**, pois ele afeta diretamente no desempenho do algoritmo. O melhor caso ocorre quando o pivô representa o valor mediano do conjunto dos elementos do vetor.

Cuidado: o valor mediano não significa que o valor está no meio do vetor original, pois ele está desordenado.

Uma estratégia de escolher um bom valor para o pivô, é obter o valor mediano de 3 (ou mais) valores arbitrários do vetor.

Exercício 6 – Implementar o código do algoritmo *Quick Sort* e criar um `main` para testar ele.

Exercício 7 – (Extra) Fazer o teste de mesa para entender bem o funcionamento do algoritmo.