

## 2. Métodos de Ordenação

A análise dos algoritmos de ordenação considerou os seguintes métodos: *BinaryInsertionSort*, *BubbleSort*, *BucketSort*, *CountingSort*, *HeapSort*, *InsertionSort*, *MergeSort*, *QuickSort*, *RadixSort* e *SelectionSort*.

### Binary Insertion Sort

*BinaryInsertionSort* é uma variação do *InsertionSort* que utiliza uma busca binária para determinar a correta localização dos elementos no processo de ordenação. Assim como o *InsertionSort*, constrói a saída ordenada através de um processo individual de cada item da entrada. Essa técnica reduz o número de comparações feitas em uma ordenação realizada no *InsertionSort* e, portanto, consegue reduzir o tempo no pior caso. Com relação a complexidade de tempo, temos o melhor caso  $O(n)$ , caso médio  $O(n^2)$  e pior caso  $O(n\log n)$ .

- **Exemplo Ilustrativo:**

(Fonte: Wikitechy.com)



Exemplo de como a ordenação ocorre no *InsertionSort*, onde o elemento em questão marcado em **verde** é inserido na sua posição correta após as comparações feitas com os elementos marcados em **vermelho**. No algoritmo *BinaryInsertionSort*, como a procura da posição correta utiliza a busca binária, o pior caso torna-se mais rápido.

- **Implementação:**

```

protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);
    for (int i=0; i<this.arrayOrdenado.length; ++i){
        int temp=this.arrayOrdenado[i];
        int indiceEsq=0;
        int indiceDir=i;
        while (indiceEsq<indiceDir){
            int indiceMeio=(indiceEsq+indiceDir)/2;
            if (temp>=this.arrayOrdenado[indiceMeio]) {
                indiceEsq = indiceMeio + 1;
            }
            else {
                indiceDir=indiceMeio;
            }
        }
        for (int j=i;j>indiceEsq;--j) {
            troca(j-1,j);
        }
    }
}

private void troca(int i,int j){
    int k=this.arrayOrdenado[i];
    this.arrayOrdenado[i]=this.arrayOrdenado[j];
    this.arrayOrdenado[j]=k;
}

```

## Bubble Sort

BubbleSort é um algoritmo que executa repetidas trocas de elementos adjacentes para corrigir a ordenação. Considerado o jeito mais simples para a tarefa de ordenação, o BubbleSort percorre o vetor a ser ordenado diversas vezes, e em cada passagem faz os elementos flutuarem aproximando-se das suas devidas posições. O algoritmo executa as mudanças até percorrer o array de entrada sem nenhum troca, pois nesse momento teremos a certeza que o array está ordenado. O movimento de flutuação lembra como as bolhas em um tanque fazem, por isso o nome BubbleSort. Com relação a complexidade de tempo, temos o melhor caso  $O(n)$ , caso médio e pior caso  $O(n^2)$ .

- **Exemplo Ilustrativo:**

(Fonte: Wikipedia.org)

### Primeiro Passo

$(\underline{5} \ 1 \ 4 \ 2 \ 8) \rightarrow (\underline{1} \ 5 \ 4 \ 2 \ 8)$ , compara os 2 primeiros elementos e troca, visto que  $5 > 1$   
 $(\underline{1} \ 5 \ 4 \ 2 \ 8) \rightarrow (\underline{1} \ 4 \ 5 \ 2 \ 8)$ , realiza a troca, visto que  $5 > 4$   
 $(\underline{1} \ 4 \ 5 \ 2 \ 8) \rightarrow (\underline{1} \ 4 \ 2 \ 5 \ 8)$ , realiza a troca, visto que  $5 > 2$   
 $(\underline{1} \ 4 \ 2 \ 5 \ 8) \rightarrow (\underline{1} \ 4 \ 2 \ 5 \ 8)$ , não troca, pois elementos já estão em ordem ( $8 > 5$ )

### Segundo Passo

$(\underline{1} \ 4 \ 2 \ 5 \ 8) \rightarrow (\underline{1} \ 4 \ 2 \ 5 \ 8)$   
 $(\underline{1} \ 4 \ 2 \ 5 \ 8) \rightarrow (\underline{1} \ 2 \ 4 \ 5 \ 8)$ , realiza a troca, visto que  $4 > 2$   
 $(\underline{1} \ 2 \ 4 \ 5 \ 8) \rightarrow (\underline{1} \ 2 \ 4 \ 5 \ 8)$   
 $(\underline{1} \ 2 \ 4 \ 5 \ 8) \rightarrow (\underline{1} \ 2 \ 4 \ 5 \ 8)$

Array já está ordenado, contudo, o algoritmo ainda não sabe e precisa de uma rodada completa sem nenhuma troca.

### Terceiro Passo

$(\underline{1} \ 2 \ 4 \ 5 \ 8) \rightarrow (\underline{1} \ 2 \ 4 \ 5 \ 8)$   
 $(\underline{1} \ 2 \ 4 \ 5 \ 8) \rightarrow (\underline{1} \ 2 \ 4 \ 5 \ 8)$   
 $(\underline{1} \ 2 \ 4 \ 5 \ 8) \rightarrow (\underline{1} \ 2 \ 4 \ 5 \ 8)$   
 $(\underline{1} \ 2 \ 4 \ 5 \ 8) \rightarrow (\underline{1} \ 2 \ 4 \ 5 \ 8)$

- **Implementação:**

```

@Override
protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);

    int len = this.arrayOrdenado.length;
    for (int i = 0; i < len-1; i++) {
        for (int j = 0; j < len-i-1; j++) {
            if (this.arrayOrdenado[j] > this.arrayOrdenado[j+1]) {
                int aux = this.arrayOrdenado[j];
                this.arrayOrdenado[j] = this.arrayOrdenado[j+1];
                this.arrayOrdenado[j+1] = aux;
            }
        }
    }
}

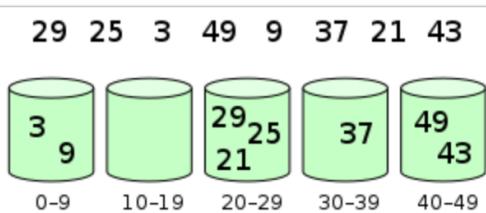
```

## Bucket Sort

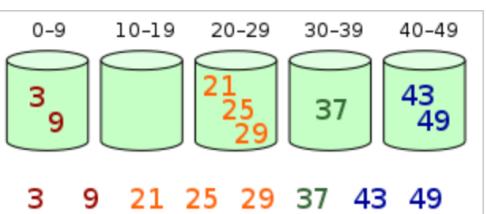
BucketSort é um algoritmo de ordenação que divide os elementos de entrada em cestos (buckets). Cada cesto é ordenado individualmente, e no final percorre-se os cestos em ordem listando os elementos e concatenando na ordem final. Essa técnica pressupõe que a entrada é uniformemente distribuída, e espera-se poucos elementos em cada intervalo. Com relação a complexidade de tempo, temos o melhor caso e caso médio  $O(n)$ , pior caso  $O(n^2)$ .

- **Exemplo Ilustrativo:**

(Fonte: Wikipedia.org)



Os elementos são distribuídos inicialmente em cestos (buckets).



Na sequência os elementos são ordenados em cada cesto e no final percorre-se os cestos em ordem listando os elementos e concatenando na ordem final.

- **Implementação:**

```

protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);

    // obtém o valor maximo para saber o numero de dígitos
    int valorMaximo = obterMaximo(this.arrayOrdenado);
    int [] balde=new int[valorMaximo+1];

    for (int i=0; i<balde.length; i++) {
        balde[i]=0;
    }

    for (int i=0; i<this.arrayOrdenado.length; i++) {
        balde[this.arrayOrdenado[i]]++;
    }

    int indiceSaida=0;
    for (int i=0; i<balde.length; i++) {
        for (int j=0; j<balde[i]; j++) {
            this.arrayOrdenado[indiceSaida++]=i;
        }
    }
}

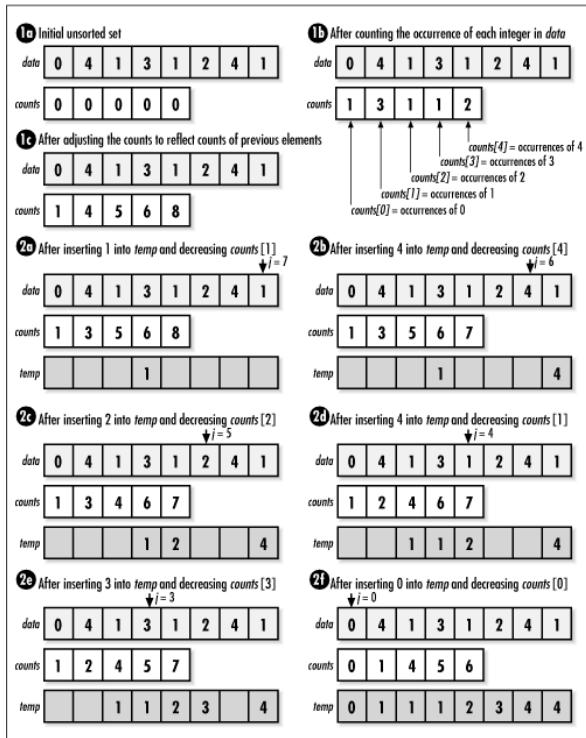
```

## Counting Sort

CountingSort é um algoritmo de ordenação por contagem, sem a necessidade de realizar comparações. Essa técnica utiliza a contagem do número de elementos, através de valores-chaves distintos, e utiliza cálculos para a posição adequada de cada elemento. Com relação a complexidade de tempo o CountingSort é um algoritmo estável, portanto, temos  $O(n + k)$  em todos os casos (melhor, médio e pior) onde  $k$  é o intervalo de chaves não negativas.

- **Exemplo Ilustrativo:**

(Fonte: oreilly.com)



Os primeiros passos do exemplo, descritos na sequencia 1a, 1b e 1c, mostram a entrada inicial utilizando um outro array auxiliar para contar a ocorrência de cada elemento na entrada, com as suas respectivas referencias. A partir da sequencia 2a, inicia-se então a “varredura” do array inicial do ultimo elemento até o primeiro inserindo os elementos nas respectivas posições do array final de saída já ordenado.

- **Implementação:**

```

protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);

    // cria um contador para cada numero possível e inicializa com zeros
    int valorMaximo = obterMaximo(this.arrayOrdenado);
    int contador[] = new int[valorMaximo+1];

    // preenche os buckets
    for (int i : this.arrayOrdenado) {
        contador[i]++;
    }

    // ordena
    int indice = 0;
    for (int i = 0; i < contador.length; i++) {
        while (0 < contador[i]) {
            this.arrayOrdenado[indice++] = i;
            contador[i]--;
        }
    }
}

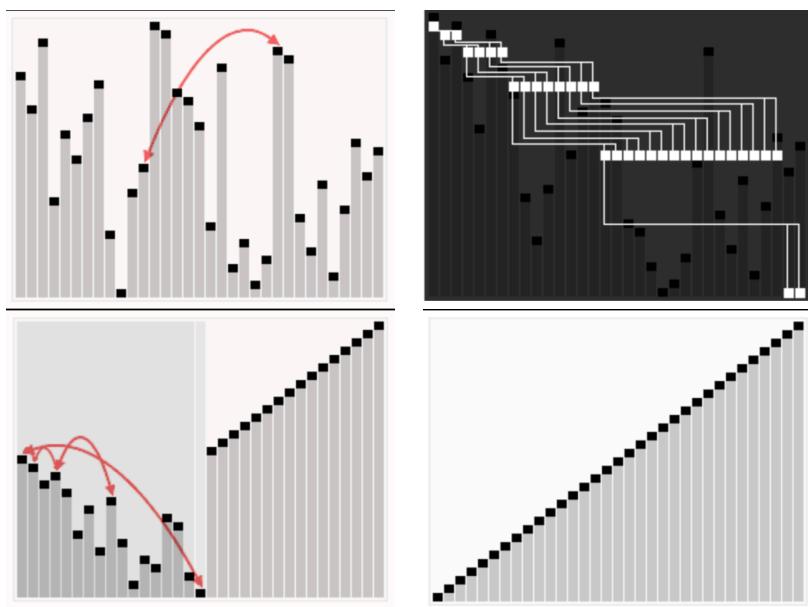
```

## Heap Sort

HeapSort é um algoritmo de ordenação por seleção. Considerado uma evolução do SelectionSort, esse algoritmo divide a entrada em regiões ordenadas e não-ordenadas, e interativamente diminui as regiões não-ordenadas movendo seus elementos para regiões ordenadas. Essa técnica utiliza a estrutura de dados Heap, representada por uma árvore binária hierárquica com algumas condições: folhas "encostadas" obrigatoriamente à esquerda; conteúdo de todos os nós são maiores ou iguais aos filhos; e os nodos são numerados de cima para baixo, da esquerda para a direita. Com relação a complexidade de tempo, temos O(nlogn) em todos os casos (melhor, médio e pior).

- **Exemplo Ilustrativo:**

(Fonte: Wikipedia.org)



A sequencia de 4 quadros simulando o HeapSort mostra a entrada sendo reordenada inicialmente para atender os requisitos de Heap, demonstrada na segunda imagem com fundo preto. A seguir temos as ordenações feitas interativamente diminuindo as regiões não-ordenadas e movendo os elementos para as regiões ordenadas.

- **Implementação:**

```

protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);
    constróiHeap();
    ordenaHeap();
}

private void constróiHeap() {
    for (int i = this.arrayOrdenado.length / 2 - 1; i >= 0; i--)
        restauraHeap(this.arrayOrdenado.length, i);
}

private void ordenaHeap() {
    for (int i = this.arrayOrdenado.length - 1; i >= 0; i--) {
        int temp = this.arrayOrdenado[0];
        this.arrayOrdenado[0] = this.arrayOrdenado[i];
        this.arrayOrdenado[i] = temp;

        restauraHeap(i, 0);
    }
}

private void restauraHeap(int tamanho, int i) {
    int leftChild = 2 * i + 1;
    int rightChild = 2 * i + 2;
    int largest = i;

    // se o filho da esquerda é maior que seu pai...
    if (leftChild < tamanho && this.arrayOrdenado[leftChild] > this.arrayOrdenado[largest]) {
        largest = leftChild;
    }

    // se o filho da direita é maior que seu pai...
    if (rightChild < tamanho && this.arrayOrdenado[rightChild] > this.arrayOrdenado[largest]) {
        largest = rightChild;
    }

    // se a troca deve ocorrer...
    if (largest != i) {
        int temp = this.arrayOrdenado[i];
        this.arrayOrdenado[i] = this.arrayOrdenado[largest];
        this.arrayOrdenado[largest] = temp;
        restauraHeap(tamanho, largest);
    }
}

```

## Insertion Sort

InsertionSort é um algoritmo de ordenação que constrói a saída ordenada através de um processo individual de cada item da entrada. É bem menos eficiente em grandes entradas comparado com algoritmos mais avançados como QuickSort, HeapSort ou MergeSort. Analogicamente essa técnica pode ser comparada com o processo de ordenação que normalmente fazemos com cartas de um baralho, ou seja, pegamos individualmente cada item e procuramos a posição correta desse item no conjunto. Com relação a complexidade de tempo, temos o melhor caso  $O(n)$ , caso médio e pior caso  $O(n^2)$ .

- **Exemplo Ilustrativo:**

(Fonte: Wikitechy.com)



Exemplo de como a ordenação ocorre no InsertionSort, onde o elemento em questão marcado em **verde** é inserido na sua posição correta após as comparações feitas com os elementos marcados em **vermelho**.

- **Implementação:**

```
protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);

    int temp;

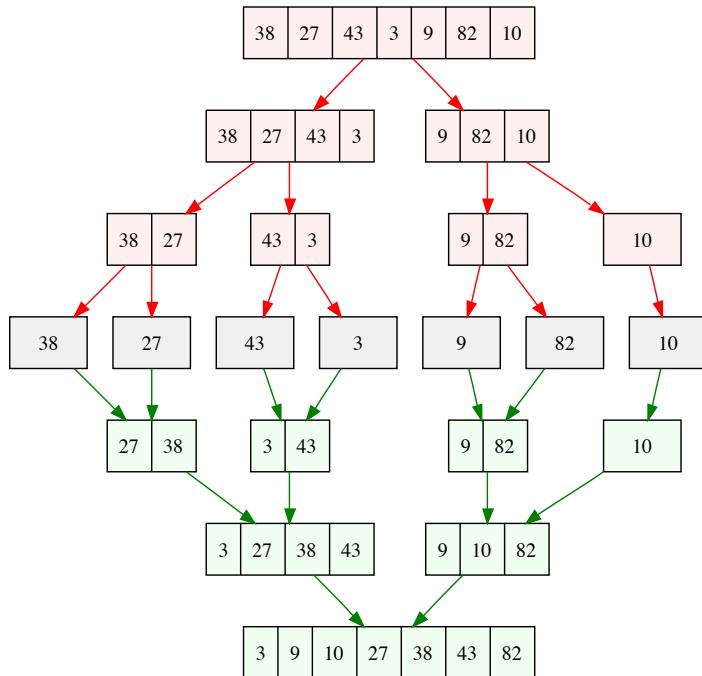
    for (int i = 1; i < this.arrayOrdenado.length; i++) {
        for(int j = i ; j > 0 ; j--){
            if(this.arrayOrdenado[j] < this.arrayOrdenado[j-1]){
                temp = this.arrayOrdenado[j];
                this.arrayOrdenado[j] = this.arrayOrdenado[j-1];
                this.arrayOrdenado[j-1] = temp;
            }
        }
    }
}
```

## Merge Sort

MergeSort é um algoritmo de ordenação que utiliza comparações em uma abordagem de divisão e conquista, ou seja, divide um problema em vários subproblemas, através da recursividade, e após os subproblemas serem resolvidos o problema conquista sua solução agregando todas as soluções dos subproblemas. Devido a sua característica de "misturar" ou agregar as soluções dos subproblemas no problema de origem, esse algoritmo recebeu o nome de Merge Sort. Esse método de ordenação possui alto nível de recursividade, e por isso o algoritmo requisita um alto consumo de memória. Portanto, essa técnica pode não ser muito eficiente em caso de escassez de recurso. Com relação a complexidade de tempo, temos  $\Theta(n \log(n))$  em todos os casos (melhor, médio e pior).

- **Exemplo Ilustrativo:**

(Fonte: Wikipedia.org)



O diagrama ao lado mostra um array desordenado marcado em vermelho que, através de várias divisões torna-se um array ordenado marcado em verde através do processo de divisão-conquista e merge.

- **Implementação:**

```

@Override
protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);
    doMergeSort(0, this.arrayInicial.length - 1);
}

private void doMergeSort(int menorIndice, int maiorIndice) {
    if (menorIndice < maiorIndice) {
        int meio = (menorIndice+maiorIndice)/2;

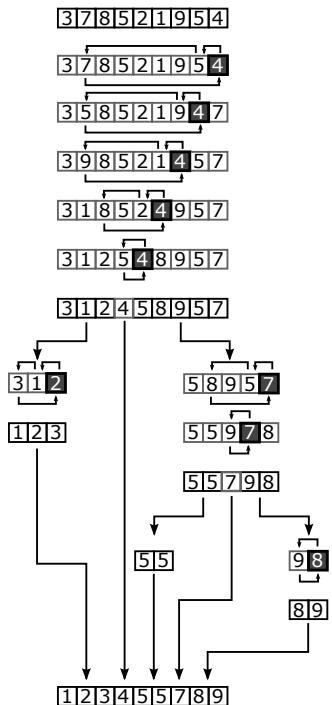
        doMergeSort(menorIndice, meio);
        doMergeSort(meio + 1, maiorIndice);
        mergePartes(menorIndice, meio, maiorIndice);
    }
}
  
```

## Quick Sort

QuickSort é um algoritmo de ordenação que utiliza comparações em uma abordagem de divisão e conquista, ou seja, divide um problema em vários subproblemas, através da recursividade, e após os subproblemas serem resolvidos o problema conquista sua solução agregando todas as soluções dos subproblemas. Essa técnica de comparação elege um elemento no array de entrada (denominado como pivô), particiona a entrada sucessivamente até que as sub-listas sejam unitárias (ou vazias) para facilitar a junção das partes e ordenação. Com relação a complexidade de tempo, temos  $O(n\log n)$  no melhor e caso médio. Entretanto, no pior caso a complexidade pode ser  $O(n^2)$ , ocorrendo geralmente quando o vetor de entrada já está ordenado.

- **Exemplo Ilustrativo:**

(Fonte: Wikipedia.org)



No exemplo ao lado temos o pivô indicado pela posição sombreada. O pivô é sempre escolhido como último elemento da partição, e através de um processo recursivo de divisão e conquista o array é finalmente ordenado.

- **Implementação:**

```

@Override
protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);
    doQuickSort(0, this.arrayInicial.length - 1);
}

private void doQuickSort(int menorIndice, int maiorIndice) {
    if (menorIndice < maiorIndice) {
        int pivot = particiona(menorIndice, maiorIndice);
        doQuickSort(menorIndice, pivot-1);
        doQuickSort(pivot+1, maiorIndice);
    }
}

private int particiona(int menorIndice, int maiorIndice) {
    int pivot = maiorIndice;

    int contador = menorIndice;
    for (int i = menorIndice; i < maiorIndice; i++) {
        if (this.arrayOrdenado[i] < this.arrayOrdenado[pivot]) {
            int temp = this.arrayOrdenado[contador];
            this.arrayOrdenado[contador] = this.arrayOrdenado[i];
            this.arrayOrdenado[i] = temp;
            contador++;
        }
    }
    int temp = this.arrayOrdenado[pivot];
    this.arrayOrdenado[pivot] = this.arrayOrdenado[contador];
    this.arrayOrdenado[contador] = temp;

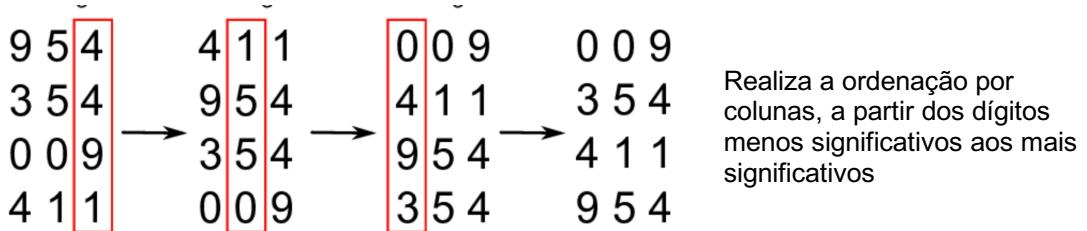
    return contador;
}
  
```

## Radix Sort

RadixSort é um algoritmo de ordenação de números inteiros que utiliza o dígito para a tarefa principal. Essa técnica realiza a ordenação por colunas, a partir dos dígitos menos significativos aos mais significativos utilizando um algoritmo estável. Com relação a complexidade de tempo, temos  $\Theta(dn)$  em todos os casos (melhor, médio e pior) onde temos  $d$  como o número de dígitos dos elementos de entrada.

- **Exemplo Ilustrativo:**

(Fonte: GitHub.com)



- **Implementação:**

```
protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);

    // obtém o valor maximo para saber o numero de dígitos
    int valorMaximo = obterMaximo(this.arrayOrdenado);

    // ordena para cada dígito
    for (int exp = 1; valorMaximo/exp > 0; exp *= 10)
        ordenaContagem(exp);

}

private void ordenaContagem(int exp)
{
    int saida[] = new int[this.arrayOrdenado.length];
    int i;
    int contador[] = new int[10];
    Arrays.fill(contador, 0);

    // armazena a contagem de ocorrências no array contador
    for (i = 0; i < this.arrayOrdenado.length; i++)
        contador[ (this.arrayOrdenado[i]/exp)%10 ]++;

    // muda o array contador para que contenha a posição atual de seu dígito na saída
    for (i = 1; i < 10; i++)
        contador[i] += contador[i - 1];

    // constrói o array saida
    for (i = this.arrayOrdenado.length - 1; i >= 0; i--)
    {
        saida[contador[ (this.arrayOrdenado[i]/exp)%10 ] - 1] = this.arrayOrdenado[i];
        contador[ (this.arrayOrdenado[i]/exp)%10 ]--;
    }

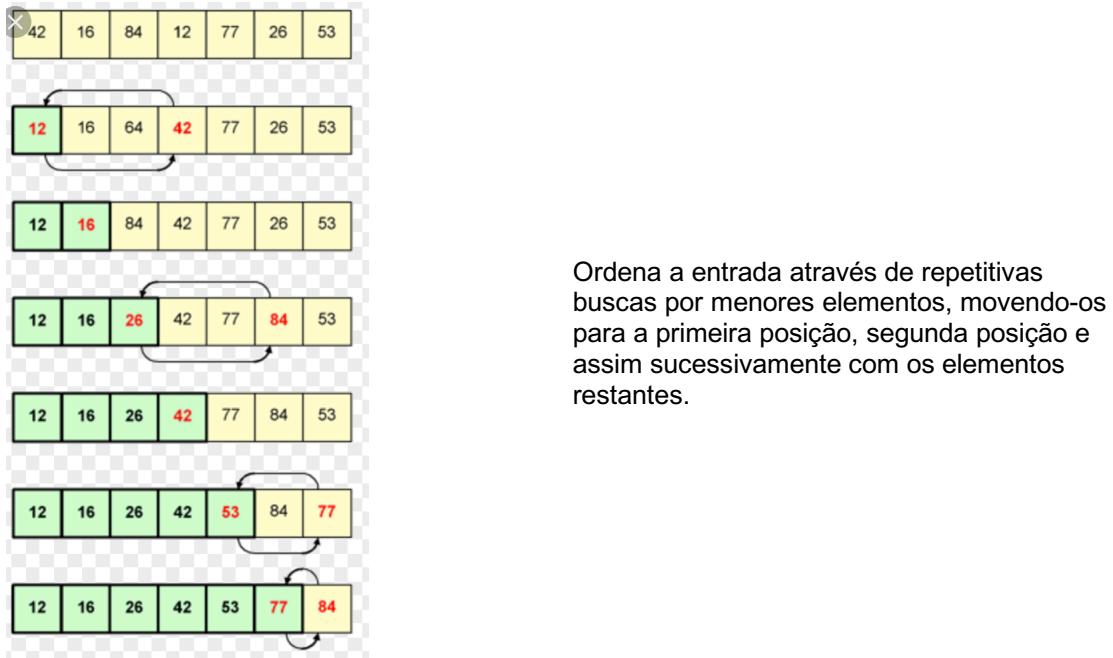
    // reorganiza o array ordenado de saída
    for (i = 0; i < this.arrayOrdenado.length; i++)
        this.arrayOrdenado[i] = saida[i];
}
```

## Selection Sort

SelectionSort é um algoritmo de ordenação que organiza a entrada através de repetitivas buscas por menores elementos, movendo-os para a primeira posição, segunda posição e assim sucessivamente com os elementos restantes. Essa técnica é conhecida por sua simplicidade, entretanto, torna-se ineficiente com grandes entradas. Com relação a complexidade de tempo, temos  $\Theta(n^2)$  em todos os casos (melhor, médio e pior).

- **Exemplo Ilustrativo:**

(Fonte: GitHub.com)



- **Implementação:**

```
protected void realizarOrdenacao() {
    System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0, this.arrayInicial.length);
    int tamanhoArray = this.arrayOrdenado.length;

    // varre o array movendo para a fronteira do sub-array não ordenado
    for (int i=0; i<tamanhoArray-1; i++) {
        // obtém o menor índice do array não ordenado
        int indiceMinimo = i;
        for (int j = i+1; j<tamanhoArray; j++) {
            if (this.arrayOrdenado[j] < this.arrayOrdenado[indiceMinimo]) {
                indiceMinimo = j;
            }
        }
        // realiza a troca do menor elemento com o primeiro
        int temp = this.arrayOrdenado[indiceMinimo];
        this.arrayOrdenado[indiceMinimo] = this.arrayOrdenado[i];
        this.arrayOrdenado[i] = temp;
    }
}
```

### 3. Experimento

O experimento e documentação desse projeto mostraram a complexidade, eficiência e comparação entre as técnicas no quesito tempo de execução. Para reproduzir o experimento novamente é necessário executar a classe *ExperimentosTeste.java* pertencente ao código-fonte do projeto publicado no GitHub, seguindo as indicações descritas em:

<https://github.com/ddangelorb/aa0/blob/master/README.md>

As seguintes configurações foram utilizadas no experimento aqui documentado:

- Máquina de Processamento
  - Tipo: **MacBook Pro**
  - Sistema Operacional: **macOS Mojave v10.14**
  - Processador: **2 GHz Intel Core i5**
  - Memória RAM: **8 GB 1867 MHz LPDDR3**
- Configurações de software
  - IDE (Integrated Development Environment): **Eclipse IDE for Java Developers, Version: 2018-12 (4.10.0), Build id: 20181214-0600**
  - Linguagem de Programação: **Java, jdk 1.8.0\_131**
  - JVM Parâmetros: **-ea -Xms15g -Xmx15g -Xss5g**
- Experimento
  - Tempo de Execução total: **25:00:32** (25 minutos e 32 milésimos)
  - Variações de entradas para todos os métodos:
    - n=25.000 com Array Aleatório, Crescente e Decrescente
    - n=50.000 com Array Aleatório, Crescente e Decrescente
    - n=100.000 com Array Aleatório, Crescente e Decrescente
    - n=200.000 com Array Aleatório, Crescente e Decrescente
    - n=400.000 com Array Aleatório, Crescente e Decrescente

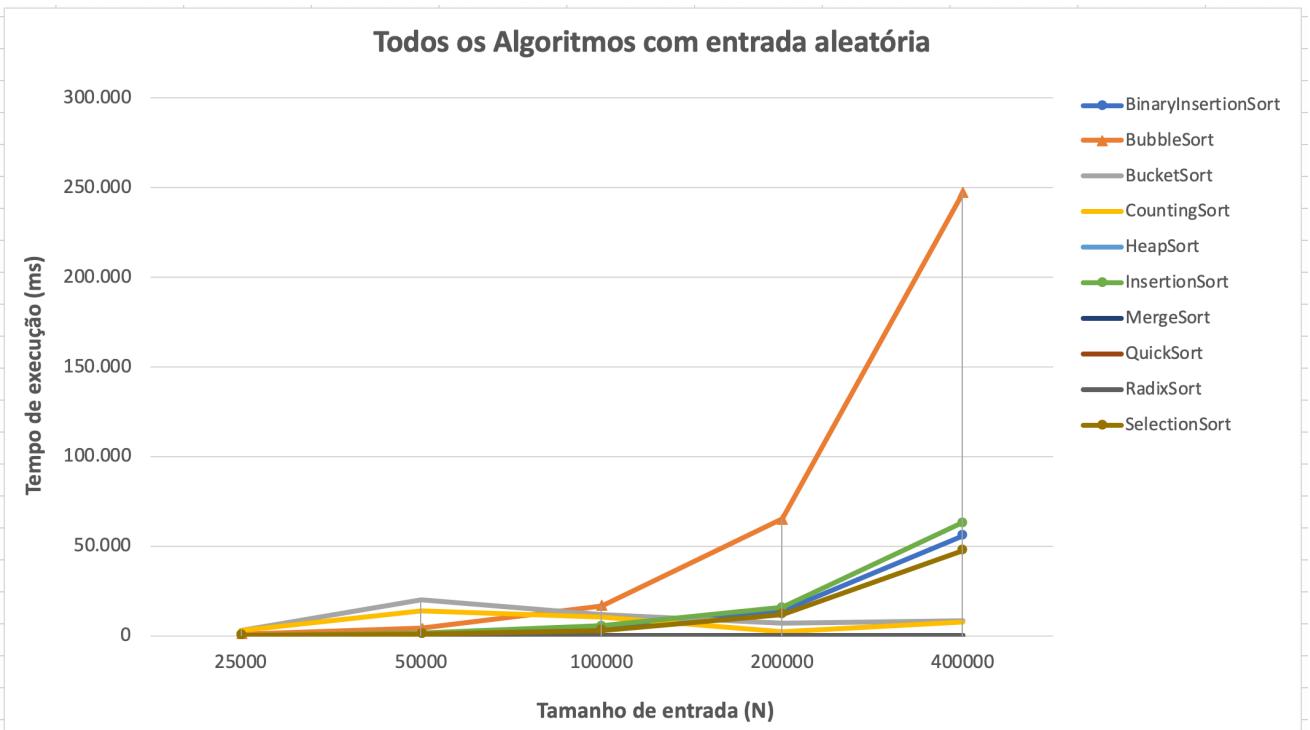
- Resultado Experimento

	Algoritmos de Ordenação									
	Binary Insertion Sort	Bubble Sort	Bucket Sort	Counting Sort	Heap Sort	Insertion Sort	Merge Sort	Quick Sort	Radix Sort	Selection Sort
Tempo (ms) para entrada N = 25.000										
Aleatório	238	956	2.973	2.872	3	267	7	2	5	195
Crescente	0	239	0	0	1	168	3	304	2	186
Decrescente	437	230	0	0	2	328	5	393	2	228
Tempo (ms) para entrada N = 50.000										
Aleatório	878	3.820	20.025	13.852	10	1.029	13	4	11	797
Crescente	2	972	0	0	4	650	8	1.210	4	750
Decrescente	1.746	928	0	0	4	1.325	8	1.581	4	922
Tempo (ms) para entrada N = 100.000										
Aleatório	3.524	16.524	11.375	10.068	24	5.143	43	26	38	2.974
Crescente	8	2.690	4	1	10	2.343	17	5.442	9	3.145
Decrescente	6.978	3.654	1	2	10	5.221	16	6.297	8	3.626
Tempo (ms) para entrada N = 200.000										
Aleatório	13.860	65.042	6.692	1.970	52	15.716	190	20	37	11.785
Crescente	7	15.068	2	1	19	10.568	32	18.943	21	11.957
Decrescente	28.529	14.736	1	1	19	21.004	86	25.043	36	14.539
Tempo (ms) para entrada N = 400.000										
Aleatório	55.883	247.019	7.944	7.708	77	62.834	93	41	68	47.503
Crescente	16	64.776	3	5	38	41.970	77	76.708	57	47.052
Decrescente	111.845	58.687	5	3	39	83.835	70	100.411	40	58.535
	223.951	495.341	49.025	36.483	312	252.401	668	236.425	342	204.194

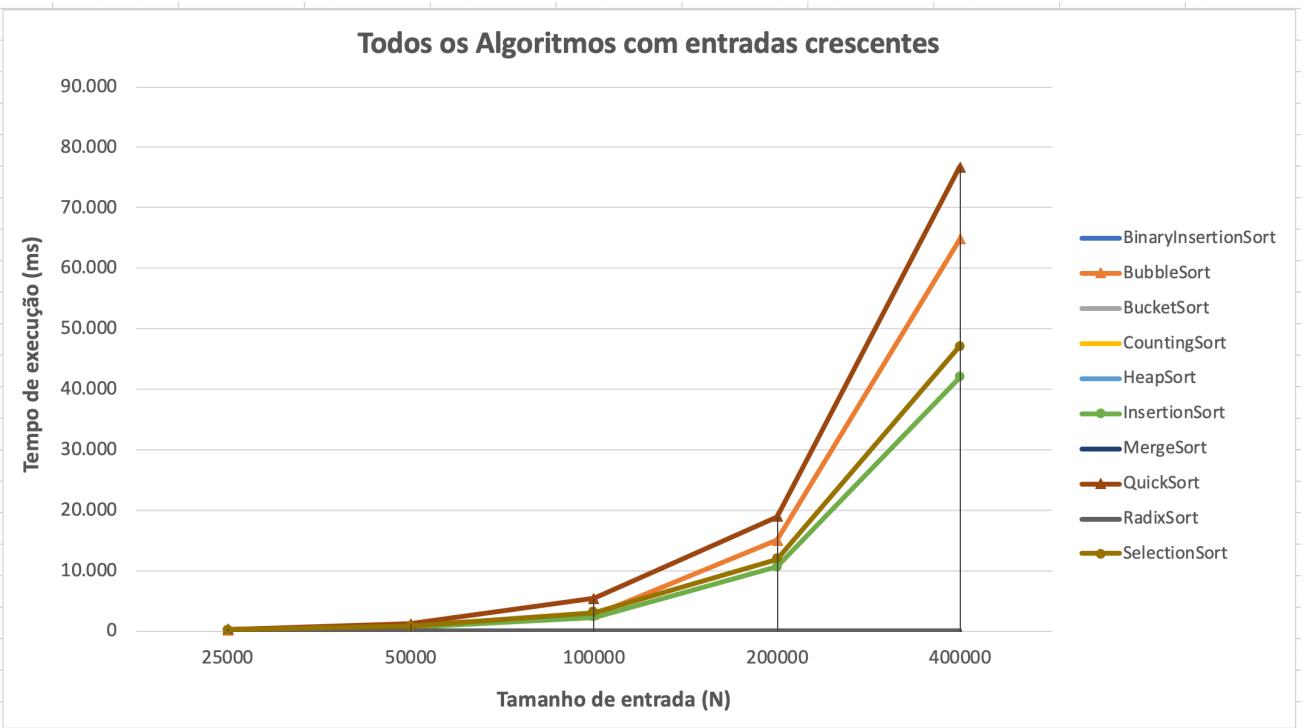
Tempos coletados em milissegundos

## 4. Análise dos dados

Os melhores resultados em termos de tempo de execução considerando todo o experimento foram nessa ordem: HeapSort, RadixSort e MergeSort. Ambos HeapSort e MergeSort possuem complexidade de tempo em  $\Theta(n \log n)$  em todos os casos (melhor, médio e pior), comprovando então a expectativa desses algoritmos serem de fato os que deveriam ter o melhor desempenho. O pior resultado em termos de tempo de execução foi o BubbleSort, confirmando as expectativas de que grandes entradas podem prejudicar o desempenho, especialmente nos casos médio e pior onde temos a complexidade quadrática. Podemos visualizar graficamente também, através de dados extraídos desse experimento, a complexidade dos algoritmos com o cenário de entradas aleatórias.

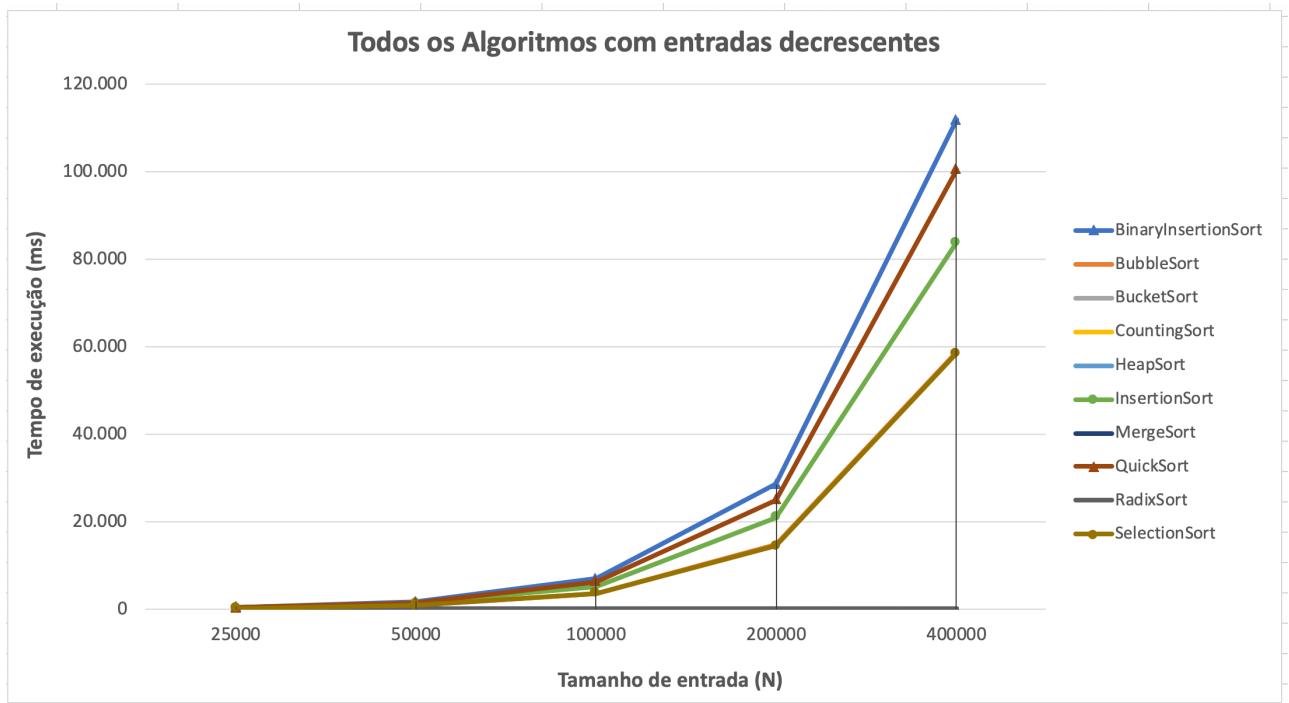


A análise do gráfico acima, com todos os algoritmos de ordenação e entradas aleatórias, mostra que os algoritmos de complexidade  $O(n^2)$  (BubbleSort, InsertionSort, BinaryInsertionSort e SelectionSort) cresceram muito mais rapidamente a partir de entradas maiores que 100.000 registros.



A análise do gráfico acima, com todos os algoritmos de ordenação e entradas crescentes, mostra que temos o mesmo comportamento dos algoritmos de

complexidade  $O(n^2)$ , com exceção do QuickSort que aparece nessa lista como o pior em tempo, pois aqui nesse cenário não temos mais a “vantagem” de termos o pivô com elementos aleatórios. Importante notar também que o BinaryInsertionSort já não aparece mais como um dos piores em desempenho, pois seu melhor caso utiliza-se da busca binária (para arrays já ordenados, nesse caso com entradas crescentes).



A análise do gráfico acima, com todos os algoritmos de ordenação e entradas decrescentes, mostra comportamento similar a entradas aleatórias com a volta do BinaryInsertionSort, pois não temos mais o benefício do melhor caso (array de entrada não está ordenado em forma crescente).

## 5. Documentação Técnica e Código Fonte

[All Classes](#)[Packages](#)[aao.algoritmos](#)  
[aao.testes](#)[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV](#) [NEXT](#) [FRAMES](#) [NO FRAMES](#)**Packages****Package**[aao.algoritmos](#)  
[aao.testes](#)**Description**

---

[All Classes](#)

BaseSort  
BaseTeste  
BinaryInsertionSort  
BinaryInsertionSortTeste  
BubbleSort  
BubbleSortTeste  
BucketSort  
BucketSortTeste  
CountingSort  
CountingSortTeste  
ExperimentosTeste  
HeapSort  
HeapSortTeste  
InsertionSort  
InsertionSortTeste  
*ISortable*  
MergeSort  
MergeSortTeste  
QuickSort  
QuickSortTeste  
RadixSort  
RadixSortTeste  
SelectionSort  
SelectionSortTeste

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV](#) [NEXT](#) [FRAMES](#) [NO FRAMES](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class BaseSort

java.lang.Object  
aao.algoritmos.BaseSort

All Implemented Interfaces:

ISortable

Direct Known Subclasses:

BinaryInsertionSort, BubbleSort, BucketSort, CountingSort, HeapSort,  
InsertionSort, MergeSort, QuickSort, RadixSort, SelectionSort

---

```
public abstract class BaseSort
extends java.lang.Object
implements ISortable
```

Classe base para todos os metodos de ordenacao. Implementa a interface ISortable para termos um padrao e polimorfismo na implementacao do metodo ordenar. Nesse estudo de caso estamos considerando 'apenas' ordenacoes atraves de arrays de inteiros, mas os metodos podem ser extensivo a outros tipos de dados.

Author:

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

**BaseSort**(java.lang.String metodo)

### Method Summary

All Methods Instance Methods Concrete Methods

##### Modifier and Type Method and Description

int[] **getArrayOrdenado()**

```
void ordenar(int[] args)
Metodo de ordenacao que deve ser implantado na classe de ordenacao
seguindo o respectivo metodo.

void ordenar(java.lang.String[] args)
```

## Methods inherited from class java.lang.Object

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

## Constructor Detail

### BaseSort

```
public BaseSort(java.lang.String metodo)
```

## Method Detail

### ordenar

```
public void ordenar(java.lang.String[] args)
```

### ordenar

```
public void ordenar(int[] args)
```

#### Description copied from interface: **ISortable**

Metodo de ordenacao que deve ser implantado na classe de ordenacao seguindo o respectivo metodo.

**Specified by:**

ordenar in interface ISortable

**Parameters:**

args – Um array de numeros inteiros a ser ordenado

### getArrayOrdenado

```
public int[] getArrayOrdenado()
```

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class BaseTeste

java.lang.Object  
aao.testes.BaseTeste

### Direct Known Subclasses:

BinaryInsertionSortTeste, BucketSortTeste, CountingSortTeste,  
ExperimentosTeste, HeapSortTeste, InsertionSortTeste, MergeSortTeste,  
QuickSortTeste, RadixSortTeste, SelectionSortTeste

---

```
public class BaseTeste
extends java.lang.Object
```

Classe base para rodas os testes e experimentos

### Author:

ddangelorb

## Constructor Summary

### Constructors

#### Constructor and Description

**BaseTeste()**

## Method Summary

### Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

**BaseTeste**

```
public BaseTeste()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [\*\*NEXT CLASS\*\*](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD [DETAIL: FIELD | CONSTR | METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class BinaryInsertionSort

java.lang.Object

aao.algoritmos.BaseSort

aao.algoritmos.BinaryInsertionSort

**All Implemented Interfaces:**[ISortable](#)

```
public class BinaryInsertionSort
extends BaseSort
```

BinaryInsertionSort eh uma variacao do [InsertionSort](#) que utiliza uma busca binaria para determinar a correta localizacao dos elementos no processo de ordenacao. Assim como o [InsertionSort](#), constroi a saida ordenada atraves de um processo individual de cada item da entrada. Essa tecnica reduz o numero de comparacoes feitas em uma ordenacao realizada no [InsertionSort](#) e, portanto, consegue reduzir o tempo no pior caso. Com relacao a complexidade de tempo, temos o melhor caso O(n), caso medio O(n<sup>2</sup>) e pior caso O(nlogn).

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[BinaryInsertionSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

#### Methods inherited from class java.lang.Object

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

## Constructor Detail

### BinaryInsertionSort

```
public BinaryInsertionSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class BinaryInsertionSortTeste

```
java.lang.Object
  aao.testes.BaseTeste
    aao.testes.BinaryInsertionSortTeste
```

```
public class BinaryInsertionSortTeste
extends BaseTeste
```

### Constructor Summary

#### Constructors

##### Constructor and Description

[BinaryInsertionSortTeste\(\)](#)

### Method Summary

#### Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### Constructor Detail

#### BinaryInsertionSortTeste

```
public BinaryInsertionSortTeste()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class BubbleSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.BubbleSort

**All Implemented Interfaces:**[ISortable](#)

```
public class BubbleSort
extends BaseSort
```

BubbleSort eh um algoritmo que executa repetidas trocas de elementos adjacentes para corrigir a ordenacao. Considerado o jeito mais simples para a tarefa de ordenacao, o BubbleSort percorre o vetor a ser ordenado diversas vezes, e em cada passagem faz os elementos flutarem aproximando-se das suas devidas posicoes. O movimento de flutuacao lembra como as bolhas em um tanque fazem, por isso o nome BubbleSort. O algoritmo executa as trocas ate percorrer o array de entrada sem nenhum troca, pois nesse momento teremos a certeza que o array esta ordenado. Com relacao a complexidade de tempo, temos o melhor caso  $O(n)$ , caso medio e pior caso  $O(n^2)$ .

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[BubbleSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

#### Methods inherited from class java.lang.Object

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

## ***Constructor Detail***

### **BubbleSort**

```
public BubbleSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#)

[NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class BubbleSortTeste

```
java.lang.Object
  aao.testes.BaseTeste
    aao.testes.BubbleSortTeste
```

```
public class BubbleSortTeste
extends BaseTeste
```

### **Constructor Summary**

#### **Constructors**

##### **Constructor and Description**

**BubbleSortTeste()**

### **Method Summary**

#### **Methods inherited from class java.lang.Object**

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### **Constructor Detail**

#### **BubbleSortTeste**

```
public BubbleSortTeste()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

[SUMMARY: NESTED | FIELD | CONSTR | METHOD](#)[DETAIL: FIELD | CONSTR | METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class BucketSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.BucketSort

**All Implemented Interfaces:**[ISortable](#)

---

```
public class BucketSort
extends BaseSort
```

BucketSort eh um algoritmo de ordenacao que divide os elementos de entrada em cestos (buckets). Cada cesto eh ordenado individualmente, e no final percorre-se os cestos em ordem listando os elementos e concatenando na ordem final. Essa tecnica pressupoe que a entrada eh uniformemente distribuida e espera-se poucos elementos em cada intervalo. Com relacao a complexidade de tempo, temos o melhor caso e caso medio  $O(n)$ , pior caso  $O(n^2)$ .

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[BucketSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

#### Methods inherited from class java.lang.Object

[equals](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

## Constructor Detail

### BucketSort

```
public BucketSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class BucketSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.BucketSortTeste

---

```
public class BucketSortTeste
extends BaseTeste
```

### **Constructor Summary**

#### Constructors

##### Constructor and Description

**BucketSortTeste()**

### **Method Summary**

#### Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### **Constructor Detail**

#### BucketSortTeste

public BucketSortTeste()

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class CountingSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.CountingSort

**All Implemented Interfaces:**[ISortable](#)

---

```
public class CountingSort
extends BaseSort
```

CountingSort eh um algoritmo de ordenacao por contagem, sem a necessidade de comparacao entre os elementos. Essa tecnica utiliza a contagem de numeros de elementos atraves de valores-chaves distintos, e utiliza calculos para a posicao adequada de cada elemento. Com relacao a complexidade de tempo o CountingSort eh um algoritmo estavel, portanto, temos O(n + k) em todos os casos (melhor, medio e pior) onde k eh o intervalo de chaves nao negativas.

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[CountingSort\( \)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

#### Methods inherited from class java.lang.Object

[equals](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

## Constructor Detail

### CountingSort

```
public CountingSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class CountingSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.CountingSortTeste

---

```
public class CountingSortTeste
extends BaseTeste
```

### Constructor Summary

#### Constructors

##### Constructor and Description

[CountingSortTeste\(\)](#)

### Method Summary

#### Methods inherited from class java.lang.Object

[equals](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

### Constructor Detail

#### CountingSortTeste

[public CountingSortTeste\(\)](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class ExperimentosTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.ExperimentosTeste

---

```
public class ExperimentosTeste
extends BaseTeste
```

### Constructor Summary

#### Constructors

##### Constructor and Description

[ExperimentosTeste\(\)](#)

### Method Summary

#### Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### Constructor Detail

#### ExperimentosTeste

```
public ExperimentosTeste()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class HeapSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.HeapSort

**All Implemented Interfaces:**[ISortable](#)

---

```
public class HeapSort
extends BaseSort
```

HeapSort eh um algoritmo de ordenacao por selecao. Considerado uma evolucao do [SelectionSort](#), esse algoritmo divide a entrada em regioes ordenadas e nao ordenadas, e interativamente diminui as regioes nao ordenadas movendo seus elementos para regioes ordenadas. Essa tecnica utiliza a estrutura de dados Heap, representada por uma arvore binaria hierarquica com algumas condicoes: folhas "encostadas" obrigatoriamente a esquerda; conteudo de todos os nohs sao maiores ou iguais aos filhos; e os nodos sao numerados de cima para baixo, da esquerda para a direita. Com relacao a complexidade de tempo, temos O(nlogn) em todos os casos (melhor, medio e pior).

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[HeapSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

#### Methods inherited from class java.lang.Object

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

## Constructor Detail

### HeapSort

```
public HeapSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class HeapSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.HeapSortTeste

---

```
public class HeapSortTeste
extends BaseTeste
```

### **Constructor Summary**

#### Constructors

##### Constructor and Description

[HeapSortTeste\(\)](#)

### **Method Summary**

#### Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### **Constructor Detail**

#### **HeapSortTeste**

`public HeapSortTeste()`

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

[SUMMARY: NESTED | FIELD | CONSTR | METHOD](#)[DETAIL: FIELD | CONSTR | METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class InsertionSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.InsertionSort

**All Implemented Interfaces:**[ISortable](#)

---

```
public class InsertionSort
extends BaseSort
```

InsertionSort eh um algoritmo de ordenacao que constroi a saida ordenada atraves de um processo individual de cada item da entrada. Eh bem menos eficiente em grandes entradas comparado com algoritmos mais avancados como [QuickSort](#), [HeapSort](#) ou [MergeSort](#). Analogicamente essa tecnica pode ser comparada com o processo de ordenacao que normalmente fazemos com cartas de um baralho, ou seja, pegamos individualmente cada item e procuramos a posicao correta desse item no conjunto. Com relacao a complexidade de tempo, temos o melhor caso O(n), caso medio e pior caso O(n<sup>2</sup>).

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[InsertionSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

#### Methods inherited from class java.lang.Object

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

## Constructor Detail

### InsertionSort

```
public InsertionSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#)

[NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class InsertionSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.InsertionSortTeste

---

```
public class InsertionSortTeste
extends BaseTeste
```

### Constructor Summary

#### Constructors

##### Constructor and Description

[InsertionSortTeste\(\)](#)

### Method Summary

#### Methods inherited from class java.lang.Object

[equals](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

### Constructor Detail

#### InsertionSortTeste

```
public InsertionSortTeste()
```

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

[SUMMARY: NESTED | FIELD | CONSTR | METHOD](#)[DETAIL: FIELD | CONSTR | METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Interface ISortable

### All Known Implementing Classes:

[BaseSort](#), [BinaryInsertionSort](#), [BubbleSort](#), [BucketSort](#), [CountingSort](#), [HeapSort](#), [InsertionSort](#), [MergeSort](#), [QuickSort](#), [RadixSort](#), [SelectionSort](#)

```
public interface ISortable
```

Interface a ser implementada em toda classe que propoe algum tipo de ordenacao

#### Author:

ddangelorb

### Method Summary

[All Methods](#)    [Instance Methods](#)    [Abstract Methods](#)

**Modifier and Type**

**Method and Description**

void

[ordenar\(int\[\] args\)](#)

Metodo de ordenacao que deve ser implantado na classe de ordenacao seguindo o respectivo metodo.

### Method Detail

**ordenar**

```
void ordenar(int[] args)
```

Metodo de ordenacao que deve ser implantado na classe de ordenacao seguindo o respectivo metodo.

**Parameters:**

args – Um array de numeros inteiros a ser ordenado

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class MergeSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.MergeSort

**All Implemented Interfaces:**[ISortable](#)

---

```
public class MergeSort
extends BaseSort
```

MergeSort eh um algoritmo de ordenacao que utiliza comparacoes em uma abordagem de divisao e conquista, ou seja, divide um problema em varios subproblemas, atraves da recursividade, e apos os subproblemas serem resolvidos o problema conquista sua solucao agregando todas as solucoes dos subproblemas. Devido a sua caracteristica de "misturar" ou agregar as solucoes dos subproblemas no problema de origem, esse algoritmo recebeu o nome de Merge Sort. Esse metodo de ordenacao possui alto nivel de recursividade, e por isso o algoritmo requisita um alto consumo de memoria. Portanto, essa tecnica pode nao ser muito eficiente em caso de escassez de recurso. Com relacao a complexidade de tempo, temos  $O(n \log n)$  em todos os casos (melhor, medio e pior).

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[MergeSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

## Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### MergeSort

```
public MergeSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)      DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class MergeSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.MergeSortTeste

---

```
public class MergeSortTeste
extends BaseTeste
```

### Constructor Summary

#### Constructors

#### Constructor and Description

[MergeSortTeste\(\)](#)

### Method Summary

#### Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### Constructor Detail

#### MergeSortTeste

`public MergeSortTeste()`

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class QuickSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.QuickSort

**All Implemented Interfaces:**[ISortable](#)

---

```
public class QuickSort
extends BaseSort
```

QuickSort eh um algoritmo de ordenacao que utiliza comparacoes em uma abordagem de divisao e conquista, ou seja, divide um problema em varios subproblemas, atraves da recursividade, e apos os subproblemas serem resolvidos o problema conquista sua solucao agregando todas as solucoes dos subproblemas. Essa tecnica de comparacao elege um elemento no array de entrada denominado-o como pivo, particionando a entrada sucessivamente ate que as sublistas sejam unitarias (ou vazias) para facilitar a juncao das partes e ordenacao. Com relacao a complexidade de tempo, temos  $O(n\log n)$  no melhor e caso medio. Entretanto, no pior caso a complexidade pode ser  $O(n^2)$ , ocorrendo geralmente quando o vetor de entrada ja esta ordenado.)

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

##### Constructor and Description

[QuickSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

## Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

## Constructor Detail

### QuickSort

`public QuickSort()`

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)      DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class QuickSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.QuickSortTeste

---

```
public class QuickSortTeste
extends BaseTeste
```

### **Constructor Summary**

#### Constructors

##### Constructor and Description

[QuickSortTeste\(\)](#)

### **Method Summary**

#### Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### **Constructor Detail**

#### QuickSortTeste

`public QuickSortTeste()`

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

[SUMMARY: NESTED | FIELD | CONSTR | METHOD](#)[DETAIL: FIELD | CONSTR | METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.algoritmos

## Class RadixSort

java.lang.Object

aao.algoritmos.BaseSort

aao.algoritmos.RadixSort

**All Implemented Interfaces:**

ISortable

```
public class RadixSort
extends BaseSort
```

RadixSort eh um algoritmo de ordenacao de numeros inteiros que utiliza o digito para a tarefa principal. Essa tecnica realiza a ordenacao por colunas, a partir dos digitos menos significativos ao mais significativo utilizando um algoritmo estavel. Com relacao a complexidade de tempo, temos O(dn) em todos os casos (melhor, medio e pior) onde temos d como o numero de digitos dos elementos de entrada.

**Author:**

ddangelorb

### Constructor Summary

#### Constructors

#### Constructor and Description

[RadixSort\(\)](#)

### Method Summary

#### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

#### Methods inherited from class java.lang.Object

[equals](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

## Constructor Detail

### RadixSort

```
public RadixSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class RadixSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.RadixSortTeste

---

```
public class RadixSortTeste
extends BaseTeste
```

### Constructor Summary

#### Constructors

##### Constructor and Description

[RadixSortTeste\(\)](#)

### Method Summary

#### Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### Constructor Detail

#### RadixSortTeste

```
public RadixSortTeste()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)SUMMARY: NESTED | FIELD | CONSTR | METHOD [DETAIL: FIELD](#) | [CONSTR](#) | [METHOD](#)

aao.algoritmos

## Class SelectionSort

java.lang.Object  
  aao.algoritmos.BaseSort  
    aao.algoritmos.SelectionSort

### All Implemented Interfaces:

[ISortable](#)

---

```
public class SelectionSort
extends BaseSort
```

SelectionSort eh um algoritmo de ordenacao que organiza a entrada atraves de repetitivas buscas por menores elementos, movendo-os para a primeira posicao, segunda posicao e assim sucessivamente com os elementos restantes. Essa tecnica eh conhecida por sua simplicidade, entretanto, torna-se ineficiente em entradas grandes. Com relacao a complexidade de tempo, temos  $O(n^2)$  em todos os casos (melhor, medio e pior).

### Author:

ddangelorb

## Constructor Summary

### Constructors

#### Constructor and Description

[SelectionSort\(\)](#)

## Method Summary

### Methods inherited from class aao.algoritmos.BaseSort

[getArrayOrdenado](#), [ordenar](#), [ordenar](#)

### Methods inherited from class java.lang.Object

[equals](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

## Constructor Detail

### SelectionSort

```
public SelectionSort()
```

[OVERVIEW](#) [PACKAGE](#) [CLASS](#) [USE](#) [TREE](#) [DEPRECATED](#) [INDEX](#) [HELP](#)

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

aao.testes

## Class SelectionSortTeste

java.lang.Object  
  aao.testes.BaseTeste  
    aao.testes.SelectionSortTeste

---

```
public class SelectionSortTeste
extends BaseTeste
```

### Constructor Summary

#### Constructors

##### Constructor and Description

[SelectionSortTeste\(\)](#)

### Method Summary

#### Methods inherited from class java.lang.Object

[equals](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

### Constructor Detail

#### SelectionSortTeste

```
public SelectionSortTeste()
```

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

```
1 package aaoo.algoritmos;
2
3 /**
4  * Classe base para todos os metodos de ordenacao. Implementa a interface ISortable
5  * para termos um padrao e polimorfismo na implementacao do metodo ordenar.
6  * Nesse estudo de caso estamos considerando 'apenas' ordenacoes atraves de arrays de
7  * inteiros, mas os metodos podem ser extensivo a outros tipos de dados.
8  * @author ddangelorb
9  */
10 public abstract class BaseSort implements ISortable {
11     protected String metodo;
12     protected int[] arrayInicial;
13     protected int[] arrayOrdenado;
14
15     public BaseSort(String metodo) {
16         this.metodo = metodo;
17     }
18
19     public void ordenar(String[] args) {
20         inicializarArrayInt(args);
21         ordenar(arrayInicial);
22     }
23
24     public void ordenar(int[] args) {
25         arrayInicial = args;
26         arrayOrdenado = new int[args.length];
27         realizarOrdenacao();
28     }
29
30     public int[] getArrayOrdenado() {
31         return this.arrayOrdenado;
32     }
33
34     protected void inicializarArrayInt(String[] args) {
35         if (args == null || args.length == 0) {
36             System.out.println("Por favor, informe um array de numeros inteiros no
37             formato: 'n1,n2,n3,...n'");
38             return;
39         }
40
41         boolean err = false;
42         arrayInicial = new int[args.length];
43
44         for (int i : arrayInicial) {
45             try {
46                 arrayInicial[i] = Integer.parseInt(args[i]);
47             }
48             catch (Exception e) {
49                 err = true;
50                 System.err.println("Erro na conversao do parametro numerico" +
51                     args[i] + " : " + e.getStackTrace());
52             }
53         }
54     }
55 }
```

```
51     if (err) {
52         System.out.println(".....");
53         System.out.println("Por favor, informe um array de numeros
54 inteiros no formato: 'n1,n2,n3,...n'");
55     }
56 }
57 }
58
59 protected int obterMaximo(int arrayComparacao[])
60 {
61     int valorMaximo = arrayComparacao[0];
62     for (int i = 1; i < arrayComparacao.length; i++) {
63         if (arrayComparacao[i] > valorMaximo) {
64             valorMaximo = arrayComparacao[i];
65         }
66     }
67     return valorMaximo;
68 }
69
70 /**
71 * Método abstrato que realizará a ordenação em todas as classes filhas
72 *(BubbleSort, BucketSort, etc)
73 */
74 protected abstract void realizarOrdenacao();
75 }
```

```
1 package aa0.testes;
2
3 import java.util.Random;
4
5 /**
6  * Classe base para rodas os testes e experimentos
7  * @author ddangelorb
8  *
9  */
10 public class BaseTeste {
11     /**
12      * Obtém um array aleatorio para os testes
13      * @param n Tamanho do array
14      * @return Retorna um array aleatorio de tamanho n
15      */
16     protected int[] getArrayAleatorio(int n) {
17         if (n==0)
18             return null;
19
20         Random rand = new Random();
21         int[] arr = new int[n];
22         for (int i=0; i<n; i++) {
23             arr[i] = rand.nextInt(Math.abs((n*n) - 1) + 1)) + 1;
24         }
25         return arr;
26     }
27
28     /**
29      *
30      * @param n
31      * @return
32      */
33     protected int[] getArrayCrescente(int n) {
34         if (n==0)
35             return null;
36
37         int[] arr = new int[n];
38         for (int i=0; i<n; i++) {
39             arr[i] = i;
40         }
41         return arr;
42     }
43
44     protected int[] getArrayDecrescente(int n) {
45         if (n==0)
46             return null;
47
48         int[] arr = new int[n];
49         int j = 0;
50         for (int i=n-1; i>=0; i--) {
51             arr[j] = i;
52             j++;
53         }
54         return arr;
55 }
```

```
55 }
56
57 protected void toPrintArrays(int [ ] arrayOriginal, int [ ] arrayOrdenado) {
58     System.out.print("Array original: ");
59     toStringArray(arrayOriginal);
60     System.out.print("Array ordenado: ");
61     toStringArray(arrayOrdenado);
62 }
63
64 protected void toStringArray(int arr[ ]) {
65     int len = arr.length;
66     for (int i=0; i<len; ++i) {
67         System.out.print(arr[i] + " ");
68     }
69     System.out.println();
70 }
71 }
72 }
```

```
1 package aa0.algoritmos;
2
3 /**
4  * BinaryInsertionSort eh uma variacao do {@link aa0.algoritmos.InsertionSort} que
5  * utiliza uma busca binaria para determinar a correta localizacao dos elementos no
6  * processo de ordenacao. Assim como o {@link aa0.algoritmos.InsertionSort}, constroi a
7  * saida ordenada atraves de um processo individual de cada item da entrada.
8  * Essa tecnica reduz o numero de comparacoes feitas em uma ordenacao realizada no
9  * {@link aa0.algoritmos.InsertionSort} e, portanto, consegue reduzir o tempo no pior
10 * caso.
11 * Com relacao a complexidade de tempo, temos o melhor caso O(n), caso medio
12 * O(n178) e pior caso O(nlogn).
13 * @author ddangelorb
14 */
15 public class BinaryInsertionSort extends BaseSort {
16     public BinaryInsertionSort() {
17         super("BinaryInsertionSort");
18     }
19
20     @Override
21     protected void realizarOrdenacao() {
22         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
23         this.arrayInicial.length);
24
25         for (int i=0; i<this.arrayOrdenado.length; ++i){
26             int temp=this.arrayOrdenado[i];
27             int indiceEsq=0;
28             int indiceDir=i;
29             while (indiceEsq<indiceDir){
30                 int indiceMeio=(indiceEsq+indiceDir)/2;
31                 if (temp>=this.arrayOrdenado[indiceMeio]) {
32                     indiceEsq = indiceMeio + 1;
33                 }
34                 else {
35                     indiceDir=indiceMeio;
36                 }
37             }
38             for (int j=i;j>indiceEsq;--j) {
39                 troca(j-1,j);
40             }
41         }
42     }
43
44     private void troca(int i,int j){
45         int k=this.arrayOrdenado[i];
46         this.arrayOrdenado[i]=this.arrayOrdenado[j];
47         this.arrayOrdenado[j]=k;
48     }
49 }
```

```
1 package aao.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aao.algoritmos.BinaryInsertionSort;
12
13 public class BinaryInsertionSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("BinaryInsertionSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         BinaryInsertionSort classeOrd = new BinaryInsertionSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         BinaryInsertionSort classeOrd = new BinaryInsertionSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         BinaryInsertionSort classeOrd = new BinaryInsertionSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         BinaryInsertionSort classeOrd = new BinaryInsertionSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
}
```

55 }  
56 |

```
1 package aaoo.algoritmos;
2
3 /**
4  * BubbleSort eh um algoritmo que executa repetidas trocas de elementos adjacentes
5  * para corrigir a ordenacao.
6  * Considerado o jeito mais simples para a tarefa de ordenacao, o BubbleSort percorre
7  * o vetor a ser ordenado diversas vezes, e em cada passagem faz os elementos flutarem
8  * aproximando-se das suas devidas posicoes. O movimento de flutuacao lembra como as
9  * bolhas em um tanque fazem, por isso o nome BubbleSort. O algoritmo executa as trocas
10 * ate percorrer o array de entrada sem nenhuma troca, pois nesse momento teremos a
11 * certeza que o array esta ordenado.
12 * Com relacao a complexidade de tempo, temos o melhor caso O(n), caso medio e pior
13 * caso O(n2).
14 * @author ddangelorb
15 *
16 */
17 public class BubbleSort extends BaseSort {
18
19     public BubbleSort() {
20         super("BubbleSort");
21     }
22
23     @Override
24     protected void realizarOrdenacao() {
25         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
26         this.arrayInicial.length);
27
28         int len = this.arrayOrdenado.length;
29         for (int i = 0; i < len-1; i++) {
30             for (int j = 0; j < len-i-1; j++) {
31                 if (this.arrayOrdenado[j] > this.arrayOrdenado[j+1]) {
32                     int aux = this.arrayOrdenado[j];
33                     this.arrayOrdenado[j] = this.arrayOrdenado[j+1];
34                     this.arrayOrdenado[j+1] = aux;
35                 }
36             }
37         }
38     }
39 }
40 }
```

```
1 package aaoo.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.AfterEach;
8 import org.junit.jupiter.api.BeforeEach;
9 import org.junit.jupiter.api.Test;
10 import org.junit.jupiter.api.TestInfo;
11
12 import aaoo.algoritmos.BubbleSort;
13
14 class BubbleSortTeste extends BaseTeste {
15
16     @BeforeEach
17     void setUp(TestInfo testInfo) throws Exception {
18         System.out.println("BubbleSortTeste." + testInfo.getDisplayName());
19     }
20
21     @AfterEach
22     void tearDown() throws Exception {
23     }
24
25     @Test
26     void testeAmostra() {
27         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
28         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
29
30         BubbleSort classeOrd = new BubbleSort();
31         classeOrd.ordenar(arrayEntrada);
32         int arraySaida[] = classeOrd.getArrayOrdenado();
33
34         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
35         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
36     }
37
38     @Test
39     void testeArrayAleatorio() {
40         int[] arrayEntrada = getArrayAleatorio(8);
41         BubbleSort classeOrd = new BubbleSort();
42         classeOrd.ordenar(arrayEntrada);
43         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
44     }
45
46     @Test
47     void testeArrayCrescente() {
48         int[] arrayEntrada = getArrayCrescente(8);
49         BubbleSort classeOrd = new BubbleSort();
50         classeOrd.ordenar(arrayEntrada);
51         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
52     }
53
54     @Test
```

```
55     void testeArrayDecrescente() {
56         int[] arrayEntrada = getArrayDecrescente(8);
57         BubbleSort classeOrd = new BubbleSort();
58         classeOrd.ordenar(arrayEntrada);
59         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado()));
60     }
61 }
62 }
```

```
1 package aaoo.algoritmos;
2
3 /**
4  * BucketSort eh um algoritmo de ordenacao que divide os elementos de entrada em
5  * cestos (buckets). Cada cesto eh ordenado individualmente, e no final percorre-se os
6  * cestos em ordem listando os elementos e concatenando na ordem final.
7  * Essa tecnica pressupoe que a entrada eh uniformemente distribuida e espera-se
8  * poucos elementos em cada intervalo.
9  * Com relacao a complexidade de tempo, temos o melhor caso e caso medio O(n), pior
10 * caso O(n178).
11 * @author ddangelorb
12 *
13 */
14
15 public class BucketSort extends BaseSort {
16     public BucketSort() {
17         super("BucketSort");
18     }
19
20     @Override
21     protected void realizarOrdenacao() {
22         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
23         this.arrayInicial.length);
24
25         // obtem o valor maximo para saber o numero de digitos
26         int valorMaximo = obterMaximo(this.arrayOrdenado);
27         int [] balde=new int[valorMaximo+1];
28
29         for (int i=0; i<balde.length; i++) {
30             balde[i]=0;
31         }
32
33         for (int i=0; i<this.arrayOrdenado.length; i++) {
34             balde[this.arrayOrdenado[i]]++;
35         }
36
37         int indiceSaida=0;
38         for (int i=0; i<balde.length; i++) {
39             for (int j=0; j<balde[i]; j++) {
40                 this.arrayOrdenado[indiceSaida++]=i;
41             }
42         }
43     }
44 }
```

```
1 package aaoo.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aaoo.algoritmos.BucketSort;
12
13 public class BucketSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("BucketSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         BucketSort classeOrd = new BucketSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         BucketSort classeOrd = new BucketSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         BucketSort classeOrd = new BucketSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         BucketSort classeOrd = new BucketSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
}
```

55 }  
56 |

```
1 package aaoo.algoritmos;
2
3 /**
4  * CountingSort eh um algoritmo de ordenacao por contagem, sem a necessidade de
5  * comparacao entre os elementos.
6  * Essa tecnica utiliza a contagem de numeros de elementos atraves de valores-chaves
7  * distintos, e utiliza calculos para a posicao adequada de cada elemento.
8  * Com relacao a complexidade de tempo o CountingSort eh um algoritmo estavel,
9  * portanto, temos O(n + k) em todos os casos (melhor, medio e pior) onde k eh o
10 * intervalo de chaves nao negativas.
11 * @author ddangelorb
12 *
13 */
14
15 public class CountingSort extends BaseSort {
16     public CountingSort() {
17         super("CountingSort");
18     }
19
20     @Override
21     protected void realizarOrdenacao() {
22         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
23         this.arrayInicial.length);
24
25         // cria um contador para cada numero possivel e inicializa com zeros
26         int valorMaximo = obterMaximo(this.arrayOrdenado);
27         int contador[] = new int[valorMaximo+1];
28
29         // preenche os buckets
30         for (int i : this.arrayOrdenado) {
31             contador[i]++;
32         }
33
34         // ordena
35         int indice = 0;
36         for (int i = 0; i < contador.length; i++) {
37             while (0 < contador[i]) {
38                 this.arrayOrdenado[indice++] = i;
39                 contador[i]--;
40             }
41         }
42     }
43 }
44 }
```

```
1 package aa0.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aa0.algoritmos.CountingSort;
12
13 public class CountingSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("CountingSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         CountingSort classeOrd = new CountingSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         CountingSort classeOrd = new CountingSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         CountingSort classeOrd = new CountingSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         CountingSort classeOrd = new CountingSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
```

```
55 |
56 }
57 |
```

```
1 package aaoo.testes;
2
3 import org.junit.jupiter.api.AfterEach;
4 import org.junit.jupiter.api.BeforeEach;
5 import org.junit.jupiter.api.Test;
6
7 import aaoo.algoritmos.*;
8
9 public class ExperimentosTeste extends BaseTeste {
10     private BinaryInsertionSort binaryInsertionSort;
11     private BubbleSort bubbleSort;
12     private BucketSort bucketSort;
13     private CountingSort countingSort;
14     private HeapSort heapSort;
15     private InsertionSort insertionSort;
16     private MergeSort mergeSort;
17     private QuickSort quickSort;
18     private RadixSort radixSort;
19     private SelectionSort selectionSort;
20
21     @BeforeEach
22     void setUp() throws Exception {
23         binaryInsertionSort = new BinaryInsertionSort();
24         bubbleSort = new BubbleSort();
25         bucketSort = new BucketSort();
26         countingSort = new CountingSort();
27         heapSort = new HeapSort();
28         insertionSort = new InsertionSort();
29         mergeSort = new MergeSort();
30         quickSort = new QuickSort();
31         radixSort = new RadixSort();
32         selectionSort = new SelectionSort();
33     }
34
35     @AfterEach
36     void tearDown() throws Exception {
37     }
38
39     @Test
40     void testeN25000() {
41         int n = 25000;
42         rodarExperimento(n);
43     }
44
45     @Test
46     void testeN50000() {
47         int n = 50000;
48         rodarExperimento(n);
49     }
50
51     @Test
52     void testeN100000() {
53         int n = 100000;
54         rodarExperimento(n);
```

```
55     }
56
57     @Test
58     void testeN200000() {
59         int n = 200000;
60         rodarExperimento(n);
61     }
62
63     @Test
64     void testeN400000() {
65         int n = 400000;
66         rodarExperimento(n);
67     }
68
69     private void doSortComTraceTempo(ISortable ordenador, int[] array, String descricao) {
70         System.out.println(descricao);
71         long tempoInicial = System.nanoTime();
72         ordenador.ordenar(array);
73         long tempoFinal = System.nanoTime();
74         double tempoDiff = (tempoFinal - tempoInicial) / 1000000;
75         System.out.println("        tempo total: " + tempoDiff + " milisegundos");
76     }
77
78     private void traceArrayExperimento(int[] array, String mensagem) {
79         System.out.println("    ");
80         System.out.println("        #####");
81         System.out.println("        " + mensagem + "        ");
82         doSortComTraceTempo(binaryInsertionSort, array, "BinaryInsertionSort com " + mensagem);
83         doSortComTraceTempo(bubbleSort, array, "BubbleSort com " + mensagem);
84         doSortComTraceTempo(bucketSort, array, "BucketSort com " + mensagem);
85         doSortComTraceTempo(countingSort, array, "CountingSort com " + mensagem);
86         doSortComTraceTempo(heapSort, array, "HeapSort com " + mensagem);
87         doSortComTraceTempo(insertionSort, array, "InsertionSort com " + mensagem);
88         doSortComTraceTempo(mergeSort, array, "MergeSort com " + mensagem);
89         doSortComTraceTempo(quickSort, array, "QuickSort com " + mensagem);
90         doSortComTraceTempo(radixSort, array, "RadixSort com " + mensagem);
91         doSortComTraceTempo(selectionSort, array, "SelectionSort com " + mensagem);
92         System.out.println("        #####");
93         System.out.println("    ");
94     }
95
96     private void rodarExperimento(int n) {
97         int[] arrayAleatorio = getArrayAleatorio(n);
98         int[] arrayCrescente = getArrayCrescente(n);
99         int[] arrayDecrescente = getArrayDecrescente(n);
100
101        System.out.println("    ");
102        System.out.println("    ");
103        System.out.println("        *****    ");
104        System.out.println("Experimento Teste n=" + n);
105
106        traceArrayExperimento(arrayAleatorio, "Array Aleatório");
```

```
107     traceArrayExperimento(arrayCrescente, "Array Crescente");
108     traceArrayExperimento(arrayDecrescente, "Array Decrescente");
109
110     System.out.println("  ");
111     System.out.println("  *****      ");
112 }
113
114 }
115 }
```

```
1 package aaoo.algoritmos;
2
3 /**
4  * HeapSort eh um algoritmo de ordenacao por selecao. Considerado uma evolucao do
5  * {@link aaoo.algoritmos.SelectionSort}, esse algoritmo divide a entrada em regioes
6  * ordenadas e nao ordenadas, e interativamente diminui as regioes nao ordenadas movendo
7  * seus elementos para regioes ordenadas.
8  * Essa tecnica utiliza a estrutura de dados Heap, representada por uma arvore binaria
9  * hierarquica com algumas condicoes: folhas "encostadas" obrigatoriamente a esquerda;
10 * conteudo de todos os nohs sao maiores ou iguais aos filhos; e os nodos sao numerados
11 * de cima para baixo, da esquerda para a direita.
12 * Com relacao a complexidade de tempo, temos O(nlogn) em todos os casos (melhor,
13 * medio e pior).
14 * @author ddangelorb
15 *
16 */
17 public class HeapSort extends BaseSort {
18     public HeapSort() {
19         super("HeapSort");
20     }
21
22     @Override
23     protected void realizarOrdenacao() {
24         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
25         this.arrayInicial.length);
26         constroiHeap();
27         ordenaHeap();
28     }
29
30     private void constroiHeap() {
31         for (int i = this.arrayOrdenado.length / 2-1; i >= 0; i--)
32             restauraHeap(this.arrayOrdenado.length, i);
33     }
34
35     private void ordenaHeap() {
36         for (int i = this.arrayOrdenado.length-1; i >= 0; i--) {
37             int temp = this.arrayOrdenado[0];
38             this.arrayOrdenado[0] = this.arrayOrdenado[i];
39             this.arrayOrdenado[i] = temp;
40
41             restauraHeap(i, 0);
42         }
43     }
44
45     private void restauraHeap(int tamanho, int i) {
46         int leftChild = 2*i+1;
47         int rightChild = 2*i+2;
48         int largest = i;
49
50         // se o filho da esquerda é maior que seu pai...
51         if (leftChild < tamanho && this.arrayOrdenado[leftChild] >
52             this.arrayOrdenado[largest]) {
53             largest = leftChild;
54         }
55     }
56 }
```

```
47     // se o filho da direita é maior que seu pai...
48     if (rightChild < tamanho && this.arrayOrdenado[rightChild] >
this.arrayOrdenado[largest]) {
49         largest = rightChild;
50     }
51
52     // se a troca deve ocorrer...
53     if (largest != i) {
54         int temp = this.arrayOrdenado[i];
55         this.arrayOrdenado[i] = this.arrayOrdenado[largest];
56         this.arrayOrdenado[largest] = temp;
57         restauraHeap(tamanho, largest);
58     }
59 }
60 }
```

```
1 package aa0.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aa0.algoritmos.HeapSort;
12
13 public class HeapSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("HeapSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         HeapSort classeOrd = new HeapSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         HeapSort classeOrd = new HeapSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         HeapSort classeOrd = new HeapSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         HeapSort classeOrd = new HeapSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
}
```

55 }  
56 |

```
1 package aaoo.algoritmos;
2
3 /**
4  * InsertionSort eh um algoritmo de ordenacao que constroi a saida ordenada atraves de
5  * um processo individual de cada item da entrada. Eh bem menos eficiente em grandes
6  * entradas comparado com algoritmos mais avancados como {@link
7  * aaoo.algoritmos.QuickSort}, {@link aaoo.algoritmos.HeapSort} ou {@link
8  * aaoo.algoritmos.MergeSort}.
9  * Analogicamente essa tecnica pode ser comparada com o processo de ordenacao que
10 * normalmente fazemos com cartas de um baralho, ou seja, pegamos individualmente cada
11 * item e procuramos a posicao correta desse item no conjunto.
12 * Com relacao a complexidade de tempo, temos o melhor caso O(n), caso medio e pior
13 * caso O(n2).
14 * @author ddangelorb
15 */
16 public class InsertionSort extends BaseSort {
17     public InsertionSort() {
18         super("InsertionSort");
19     }
20
21     @Override
22     protected void realizarOrdenacao() {
23         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
24         this.arrayInicial.length);
25
26         int temp;
27
28         for (int i = 1; i < this.arrayOrdenado.length; i++) {
29             for(int j = i ; j > 0 ; j--){
30                 if(this.arrayOrdenado[j] < this.arrayOrdenado[j-1]){
31                     temp = this.arrayOrdenado[j];
32                     this.arrayOrdenado[j] = this.arrayOrdenado[j-1];
33                     this.arrayOrdenado[j-1] = temp;
34                 }
35             }
36         }
37     }
38 }
```

```
1 package aa0.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aa0.algoritmos.InsertionSort;
12
13 public class InsertionSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("InsertionSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         InsertionSort classeOrd = new InsertionSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         InsertionSort classeOrd = new InsertionSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         InsertionSort classeOrd = new InsertionSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         InsertionSort classeOrd = new InsertionSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
```

55 }  
56 |

```
1 package aa0.algoritmos;
2
3 /**
4 * Interface a ser implementada em toda classe que propoe algum tipo de ordenacao
5 * @author ddangelorb
6 *
7 */
8 public interface ISortable {
9     /**
10      * Metodo de ordenacao que deve ser implantado na classe de ordenacao seguindo o
11      * respectivo metodo.
12      * @param args Um array de numeros inteiros a ser ordenado
13      */
14     void ordenar(int[] args);
15 }
```

```
1 package aaoo.algoritmos;
2
3 /**
4  * MergeSort eh um algoritmo de ordenacao que utiliza comparacoes em uma abordagem de
5  * divisao e conquista, ou seja, divide um problema em varios subproblemas, atraves da
6  * recursividade, e apos os subproblemas serem resolvidos o problema conquista sua
7  * solucao agregando todas as solucoes dos subproblemas.
8  * Devido a sua caracteristica de "misturar" ou agregar as solucoes dos subproblemas
9  * no problema de origem, esse algoritmo recebeu o nome de Merge Sort. Esse metodo de
10 * ordenacao possui alto nivel de recursividade, e por isso o algoritmo requisita um alto
11 * consumo de memoria. Portanto, essa tecnica pode nao ser muito eficiente em caso de
12 * escassez de recurso.
13 * Com relacao a complexidade de tempo, temos O(nlog(n)) em todos os casos (melhor,
14 * medio e pior).
15 * @author ddangelorb
16 *
17 */
18
19 public class MergeSort extends BaseSort {
20     public MergeSort() {
21         super("MergeSort");
22     }
23
24     @Override
25     protected void realizarOrdenacao() {
26         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
27         this.arrayInicial.length);
28
29         doMergeSort(0, this.arrayInicial.length - 1);
30     }
31
32     private void doMergeSort(int menorIndice, int maiorIndice) {
33         if (menorIndice < maiorIndice) {
34             int meio = (menorIndice+maiorIndice)/2;
35
36             doMergeSort(menorIndice, meio);
37             doMergeSort(meio + 1, maiorIndice);
38             mergePartes(menorIndice, meio, maiorIndice);
39         }
40     }
41
42     private void mergePartes(int menorIndice, int meio, int maiorIndice) {
43         // obtém o tamanho dos 2 sub-arrays de trabalho
44         int tamanhoArrayEsq = meio - menorIndice + 1;
45         int tamanhoArrayDir = maiorIndice - meio;
46
47         // cria os 2 sub-arrays que serão mergeados
48         int arrayEsq[] = new int [tamanhoArrayEsq];
49         int arrayDir[] = new int [tamanhoArrayDir];
50
51         // copia dos 2 sub-arrays de trabalho
52         for (int i=0; i<tamanhoArrayEsq; ++i) {
53             arrayEsq[i] = this.arrayOrdenado[menorIndice + i];
54         }
55         for (int j=0; j<tamanhoArrayDir; ++j) {
56             arrayDir[j] = this.arrayOrdenado[meio + 1+ j];
57         }
58
59         // mergea os 2 sub-arrays
60         int k = 0;
61         for (int i=menorIndice; i<maiorIndice; ++i) {
62             if (arrayEsq[k] < arrayDir[k]) {
63                 this.arrayOrdenado[i] = arrayEsq[k];
64             } else {
65                 this.arrayOrdenado[i] = arrayDir[k];
66             }
67             k++;
68         }
69     }
70 }
```

```
47     }
48
49     int indiceEsq = 0;
50     int indiceDir = 0;
51     int k = menorIndice;
52
53     while (indiceEsq < tamanhoArrayEsq && indiceDir < tamanhoArrayDir) {
54         if (arrayEsq[indiceEsq] <= arrayDir[indiceDir]) {
55             this.arrayOrdenado[k] = arrayEsq[indiceEsq];
56             indiceEsq++;
57         }
58         else {
59             this.arrayOrdenado[k] = arrayDir[indiceDir];
60             indiceDir++;
61         }
62         k++;
63     }
64
65     // copia se houver valores remanescentes do sub-array esq
66     while (indiceEsq < tamanhoArrayEsq) {
67         this.arrayOrdenado[k] = arrayEsq[indiceEsq];
68         indiceEsq++;
69         k++;
70     }
71
72     // copia se houver valores remanescentes do sub-array dir
73     while (indiceDir < tamanhoArrayDir) {
74         this.arrayOrdenado[k] = arrayDir[indiceDir];
75         indiceDir++;
76         k++;
77     }
78 }
79 }
80 }
```

```
1 package aa0.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aa0.algoritmos.MergeSort;
12
13 public class MergeSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("MergeSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         MergeSort classeOrd = new MergeSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         MergeSort classeOrd = new MergeSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         MergeSort classeOrd = new MergeSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         MergeSort classeOrd = new MergeSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
}
```

55 }  
56 |

```
1 package aaoo.algoritmos;
2
3 /**
4  * QuickSort eh um algoritmo de ordenacao que utiliza comparacoes em uma abordagem de
5  * divisao e conquista, ou seja, divide um problema em varios subproblemas, atraves da
6  * recursividade, e apos os subproblemas serem resolvidos o problema conquista sua
7  * solucao agregando todas as solucoes dos subproblemas.
8  * Essa tecnica de comparacao elege um elemento no array de entrada denominado-o como
9  * pivo, particionado a entrada sucessivamente ate que as sublistas sejam unitarias (ou
10 * vazias) para facilitar a juncao das partes e ordenacao.
11 */
12 * Com relacao a complexidade de tempo, temos O(nlogn) no melhor e caso medio.
13 Entretanto, no pior caso a complexidade pode ser O(n178), ocorrendo geralmente
14 quando o vetor de entrada ja esta ordenado.
15 )
16 * @author ddangelorb
17 *
18 */
19
20 public class QuickSort extends BaseSort {
21     public QuickSort() {
22         super("QuickSort");
23     }
24
25     @Override
26     protected void realizarOrdenacao() {
27         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
28         this.arrayInicial.length);
29         doQuickSort(0, this.arrayInicial.length - 1);
30     }
31
32     private void doQuickSort(int menorIndice, int maiorIndice) {
33         if (menorIndice < maiorIndice) {
34             int pivot = particiona(menorIndice, maiorIndice);
35             doQuickSort(menorIndice, pivot-1);
36             doQuickSort(pivot+1, maiorIndice);
37         }
38     }
39
40     private int particiona(int menorIndice, int maiorIndice) {
41         int pivot = maiorIndice;
42
43         int contador = menorIndice;
44         for (int i = menorIndice; i<maiorIndice; i++) {
45             if (this.arrayOrdenado[i] < this.arrayOrdenado[pivot]) {
46                 int temp = this.arrayOrdenado[contador];
47                 this.arrayOrdenado[contador] = this.arrayOrdenado[i];
48                 this.arrayOrdenado[i] = temp;
49                 contador++;
50             }
51         }
52         int temp = this.arrayOrdenado[pivot];
53         this.arrayOrdenado[pivot] = this.arrayOrdenado[contador];
54         this.arrayOrdenado[contador] = temp;
55
56         return contador;
57     }
58 }
```

48 }  
49 |

```
1 package aa0.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aa0.algoritmos.QuickSort;
12
13 public class QuickSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("QuickSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         QuickSort classeOrd = new QuickSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         QuickSort classeOrd = new QuickSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         QuickSort classeOrd = new QuickSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         QuickSort classeOrd = new QuickSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
}
```

55 }  
56 |

```
1 package aaoo.algoritmos;
2
3 import java.util.Arrays;
4
5 /**
6  * RadixSort eh um algoritmo de ordenacao de numeros inteiros que utiliza o digito
7  * para a tarefa principal.
8  * Essa tecnica realiza a ordenacao por colunas, a partir dos digitos menos
9  * significativos ao mais significativo utilizando um algoritmo estavel.
10 * Com relacao a complexidade de tempo, temos O(dn) em todos os casos (melhor, medio e
11 * pior) onde temos d como o numero de digitos dos elementos de entrada.
12 * @author ddangelorb
13 *
14 */
15
16
17 public class RadixSort extends BaseSort {
18     public RadixSort() {
19         super("RadixSort");
20     }
21
22     @Override
23     protected void realizarOrdenacao() {
24         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
25         this.arrayInicial.length);
26
27         // obtem o valor maximo para saber o numero de digitos
28         int valorMaximo = obterMaximo(this.arrayOrdenado);
29
30         // ordena para cada digito
31         for (int exp = 1; valorMaximo/exp > 0; exp *= 10)
32             ordenaContagem(exp);
33
34     }
35
36     private void ordenaContagem(int exp)
37     {
38         int saida[] = new int[this.arrayOrdenado.length];
39         int i;
40         int contador[] = new int[10];
41         Arrays.fill(contador,0);
42
43         // armazena a contagem de ocorrencias no array contador
44         for (i = 0; i < this.arrayOrdenado.length; i++)
45             contador[ (this.arrayOrdenado[i]/exp)%10 ]++;
46
47         // muda o array contador para que contenha a posicao atual de seu digito na
48         // saida
49         for (i = 1; i < 10; i++)
50             contador[i] += contador[i - 1];
51
52         // constroi o array saida
53         for (i = this.arrayOrdenado.length - 1; i >= 0; i--)
54         {
55             saida[contador[ (this.arrayOrdenado[i]/exp)%10 ] - 1] =
56             this.arrayOrdenado[i];
57             contador[ (this.arrayOrdenado[i]/exp)%10 ]--;
58         }
59     }
60 }
```

```
50     }
51
52     // reorganiza o array ordenado de saida
53     for (i = 0; i < this.arrayOrdenado.length; i++)
54         this.arrayOrdenado[i] = saida[i];
55     }
56 }
57 }
```

```
1 package aao.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aao.algoritmos.RadixSort;
12
13 public class RadixSortTeste extends BaseTeste {
14     @BeforeEach
15     void setUp(TestInfo testInfo) throws Exception {
16         System.out.println("RadixSortTeste." + testInfo.getDisplayName());
17     }
18
19     @Test
20     void testeAmostra() {
21         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
22         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
23
24         RadixSort classeOrd = new RadixSort();
25         classeOrd.ordenar(arrayEntrada);
26         int arraySaida[] = classeOrd.getArrayOrdenado();
27
28         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
29         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
30     }
31
32     @Test
33     void testeArrayAleatorio() {
34         int[] arrayEntrada = getArrayAleatorio(8);
35         RadixSort classeOrd = new RadixSort();
36         classeOrd.ordenar(arrayEntrada);
37         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
38     }
39
40     @Test
41     void testeArrayCrescente() {
42         int[] arrayEntrada = getArrayCrescente(8);
43         RadixSort classeOrd = new RadixSort();
44         classeOrd.ordenar(arrayEntrada);
45         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
46     }
47
48     @Test
49     void testeArrayDecrescente() {
50         int[] arrayEntrada = getArrayDecrescente(8);
51         RadixSort classeOrd = new RadixSort();
52         classeOrd.ordenar(arrayEntrada);
53         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
54     }
}
```

55 }  
56 |

```
1 package aaoo.algoritmos;
2
3 /**
4  * SelectionSort eh um algoritmo de ordenacao que organiza a entrada atraves de
5  * repetitivas buscas por menores elementos, movendo-os para a primeira posicao, segunda
6  * posicao e assim sucessivamente com os elementos restantes.
7  * Essa tecnica eh conhecida por sua simplicidade, entretanto, torna-se ineficiente em
8  * entradas grandes.
9  * Com relacao a complexidade de tempo, temos O(n2) em todos os casos (melhor,
10 * medio e pior).
11 * @author ddangelorb
12 *
13 */
14
15 public class SelectionSort extends BaseSort {
16     public SelectionSort() {
17         super("SelectionSort");
18     }
19
20     @Override
21     protected void realizarOrdenacao() {
22         System.arraycopy(this.arrayInicial, 0, this.arrayOrdenado, 0,
23         this.arrayInicial.length);
24         int tamanhoArray = this.arrayOrdenado.length;
25
26         // varre o array movendo para a fronteira do sub-array não ordenado
27         for (int i=0; i<tamanhoArray-1; i++) {
28             {
29                 // obtém o minimo indice do array não ordenado
30                 int indiceMinimo = i;
31                 for (int j = i+1; j<tamanhoArray; j++) {
32                     if (this.arrayOrdenado[j] < this.arrayOrdenado[indiceMinimo]) {
33                         indiceMinimo = j;
34                     }
35                 }
36             }
37         }
38     }
39 }
```

```
1 package aao.testes;
2
3 import static org.junit.Assert.assertTrue;
4
5 import java.util.Arrays;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9 import org.junit.jupiter.api.TestInfo;
10
11 import aao.algoritmos.SelectionSort;
12
13 public class SelectionSortTeste extends BaseTeste {
14
15     @BeforeEach
16     void setUp(TestInfo testInfo) throws Exception {
17         System.out.println("SelectionSortTeste." + testInfo.getDisplayName());
18     }
19
20     @Test
21     void testeAmostra() {
22         int arrayEntrada[] = {64, 34, 25, 12, 22, 11, 90};
23         int arrayEntradaOrdenado[] = {11, 12, 22, 25, 34, 64, 90};
24
25         SelectionSort classeOrd = new SelectionSort();
26         classeOrd.ordenar(arrayEntrada);
27         int arraySaida[] = classeOrd.getArrayOrdenado();
28
29         assertTrue(Arrays.equals(arrayEntradaOrdenado, arraySaida));
30         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
31     }
32
33     @Test
34     void testeArrayAleatorio() {
35         int[] arrayEntrada = getArrayAleatorio(8);
36         SelectionSort classeOrd = new SelectionSort();
37         classeOrd.ordenar(arrayEntrada);
38         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
39     }
40
41     @Test
42     void testeArrayCrescente() {
43         int[] arrayEntrada = getArrayCrescente(8);
44         SelectionSort classeOrd = new SelectionSort();
45         classeOrd.ordenar(arrayEntrada);
46         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
47     }
48
49     @Test
50     void testeArrayDecrescente() {
51         int[] arrayEntrada = getArrayDecrescente(8);
52         SelectionSort classeOrd = new SelectionSort();
53         classeOrd.ordenar(arrayEntrada);
54         toPrintArrays(arrayEntrada, classeOrd.getArrayOrdenado());
```

```
55 }  
56 }  
57 }
```