

O relatório abaixo foi elaborado de acordo com a atividade, seguindo a orientação dos padrões *generics* em java e utilizando os fundamentos apresentados pelo CORMEN. Neste relatório, farei uma breve discussão, apresentando principais detalhes da logica utilizado nos códigos, incluindo comentários das principais funções a análise da complexidade de tempo (Big O).

CÓDIGO 1 – MERGESORT OTIMIZADO

DESCRIÇÃO GERAL DO CÓDIGO

O programa MergeSortOtimizado.java define um método de ordenação que utiliza MergeSort com duas otimizações principais:

- 1.InsertionSort para Subvetores Pequenos: Subvetores com tamanho 15 ou menos são ordenados usando InsertionSort, que é mais eficiente para conjuntos pequenos de dados.
- 2.Verificação de Ordem antes da Fusão: Antes de fundir duas metades de um vetor, o algoritmo verifica se elas já estão ordenadas. Se $\text{vetor}[\text{meio}] \leq \text{vetor}[\text{meio} + 1]$, a fusão é evitada, o que pode reduzir significativamente o número de operações necessárias em vetores parcialmente ordenados.

FUNÇÕES PRINCIPAIS E COMPLEXIDADE DE TEMPO

`ordenarMergeSort(T[] vetor)`

- Complexidade de Tempo: $O(\log n)$ no caso médio e no pior caso.
- Este é o método inicial que chama o processo recursivo de divisão do vetor para o MergeSort. Ele também verifica se o vetor já está ordenado, o que é uma operação $O(n)$, mas que pode economizar significativamente tempo se o vetor não necessitar de ordenação.

`ordenarMergeSortRecursivo(T[] vetor, int inicio, int fim)`

- Complexidade de Tempo: $O(n \log n)$ no caso médio e no pior caso. A complexidade permanece a mesma de um MergeSort tradicional porque ainda divide o vetor e realiza a fusão.
- Executa a lógica de divisão recursiva do vetor e decide se deve proceder com a fusão ou usar InsertionSort para subvetores pequenos. As otimizações adicionadas não mudam a complexidade de tempo teórica do MergeSort, mas melhoram o desempenho prático.

Rondineli Seba Salomão

Matrícula: 20200022063

mesclar(T[] vetor, int inicio, int meio, int fim)

-Complexidade de Tempo: $O(n)$ para cada chamada, onde n é o número de elementos a serem fundidos.

-Combina dois subvetores ordenados em um único vetor ordenado. Esta função é chamada após a recursão e é uma parte essencial do MergeSort, responsável por unir as partes divididas.

ordenarPorInsercao(T[] vetor, int inicio, int fim)

-Complexidade de Tempo: $O(k^2)$, onde k é o número de elementos no subvetor.

-Implementa o algoritmo de ordenação por inserção, ideal para pequenas quantidades de dados devido à sua simplicidade. É usada aqui para subvetores de tamanho 15 ou menos, aproveitando sua eficiência nesses casos.

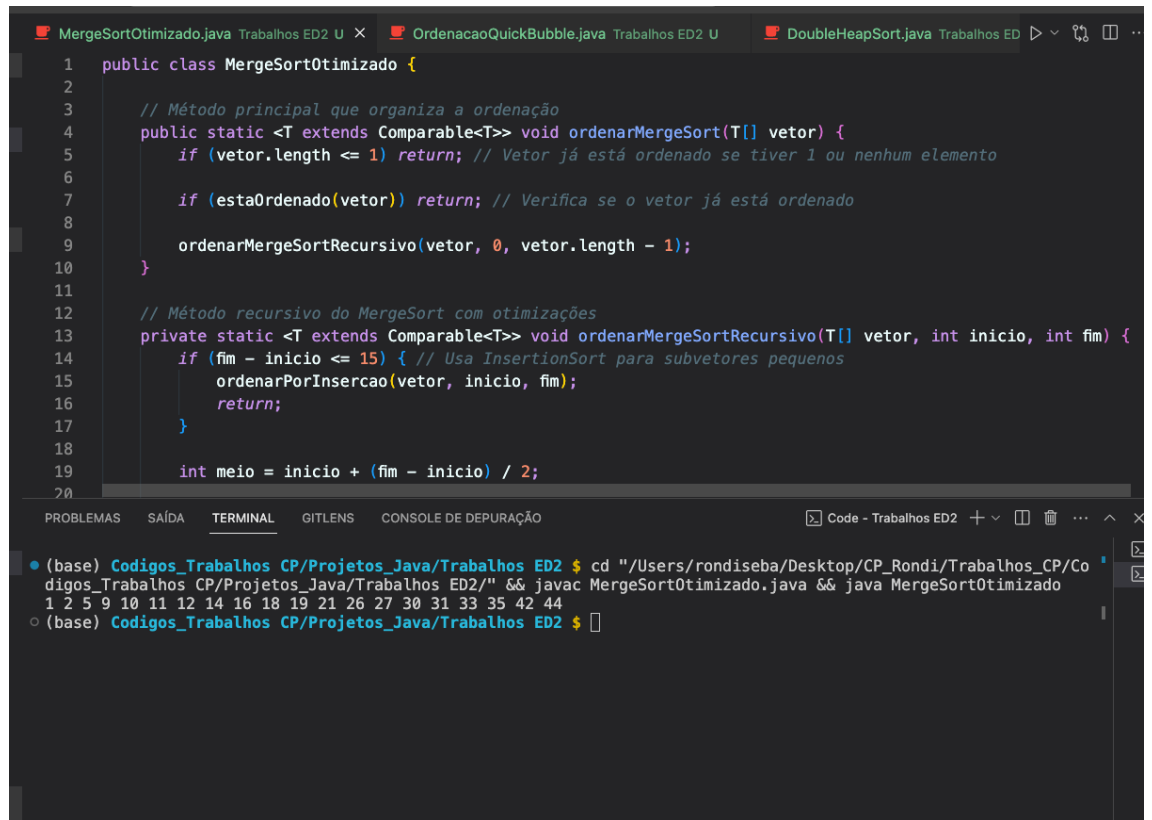
estaOrdenado(T[] vetor)

-Complexidade de Tempo: $O(n)$.

-Verifica se o vetor está ordenado, percorrendo o vetor uma vez. É uma verificação preliminar que pode evitar a necessidade de ordenação se o vetor já estiver ordenado.

Conclusão

O MergeSortOtimizado apresenta melhorias significativas sobre o MergeSort tradicional para casos práticos, especialmente quando muitos subvetores já estão ordenados ou são pequenos, reduzindo a complexidade. As otimizações não alteram a complexidade de tempo teórica principal do MergeSort, mas podem oferecer ganhos de desempenho substanciais ao reduzir o número de operações necessárias em cenários comuns. Essas melhorias tornam o algoritmo particularmente adequado para aplicações que frequentemente lidam com dados parcialmente ordenados ou em pequenas quantidades.



```
1 public class MergeSortOtimizado {
2
3     // Método principal que organiza a ordenação
4     public static <T extends Comparable<T>> void ordenarMergeSort(T[] vetor) {
5         if (vetor.length <= 1) return; // Vetor já está ordenado se tiver 1 ou nenhum elemento
6
7         if (estaOrdenado(vetor)) return; // Verifica se o vetor já está ordenado
8
9         ordenarMergeSortRecursivo(vetor, 0, vetor.length - 1);
10    }
11
12    // Método recursivo do MergeSort com otimizações
13    private static <T extends Comparable<T>> void ordenarMergeSortRecursivo(T[] vetor, int inicio, int fim) {
14        if (fim - inicio <= 15) { // Usa InsertionSort para subvetores pequenos
15            ordenarPorInsercao(vetor, inicio, fim);
16            return;
17        }
18
19        int meio = inicio + (fim - inicio) / 2;
20    }
21}
```

PROBLEMAS SAÍDA **TERMINAL** GITLENS CONSOLE DE DEPURACÃO Code - Trabalhos ED2

```
• (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $ cd "/Users/rondiseba/Desktop/CP_Rondi/Trabalhos_CP/Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2/" && javac MergeSortOtimizado.java && java MergeSortOtimizado
1 2 5 9 10 11 12 14 16 18 19 21 26 27 30 31 33 35 42 44
○ (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $
```

CODIGO 2 - SELECTSORT MODIFICADO

DESCRIÇÃO GERAL DO CÓDIGO

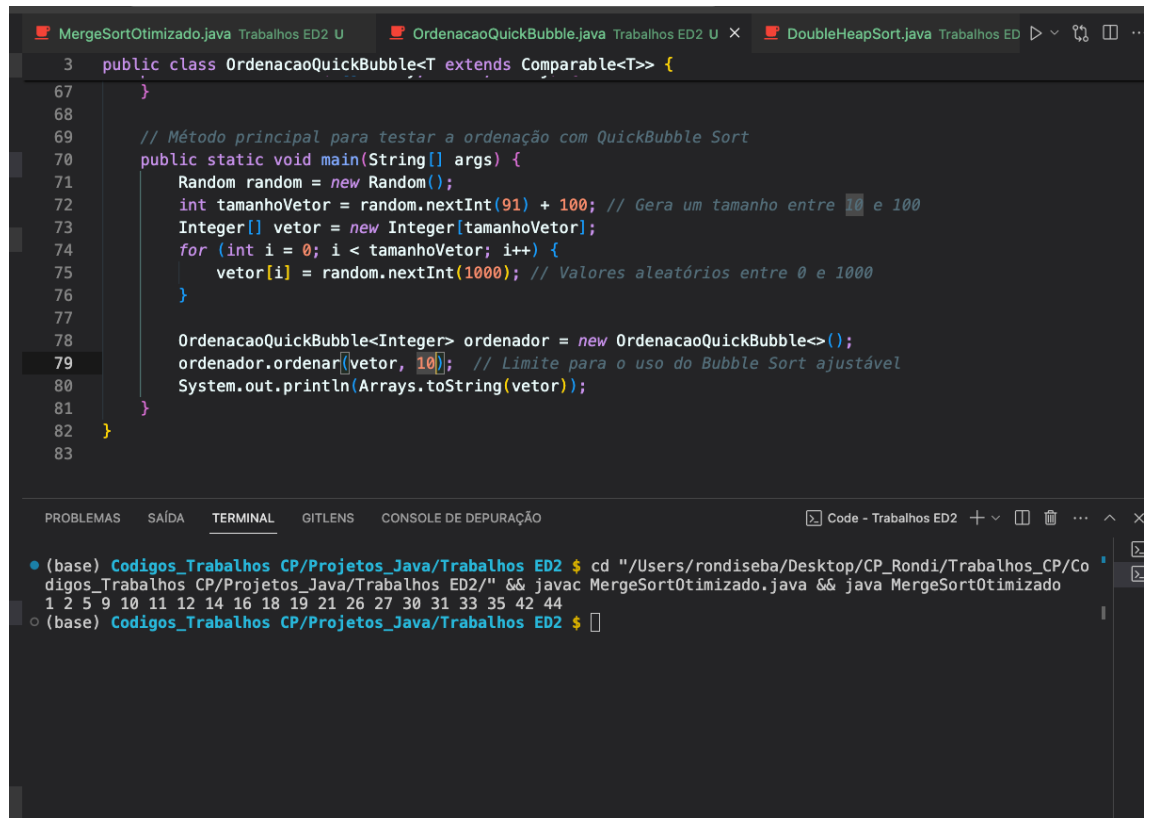
Implementei uma versão modificada do algoritmo de ordenação por seleção (SelectSort), onde a principal é que, em cada iteração, o algoritmo identifica tanto o menor quanto o maior elemento no subvetor não ordenado e os coloca em suas posições corretas, um no início e outro no final do subvetor. Este método é conhecido como MinMax SelectSort.

FUNÇÕES PRINCIPAIS E COMPLEXIDADE DE TEMPO

`selectSortMinMax(T[] vetor)`

- Função: Ordena o vetor passado como parâmetro.
- Complexidade de Tempo: A análise de complexidade de tempo para esta função é um pouco mais complexa do que o SelectSort tradicional devido à busca simultânea do menor e do maior valor. No entanto, em termos de comparações, este método pode ser mais eficiente. Cada iteração realiza duas comparações por elemento (uma para o mínimo e uma para o máximo), mas como estamos resolvendo duas posições por iteração, o número total de iterações é reduzido pela metade. Ainda assim, o comportamento geral permanece $O(n^2)$.

O algoritmo MinMax SelectSort oferece uma pequena otimização sobre o algoritmo de SelectSort tradicional ao resolver duas posições do vetor em cada passagem (o menor e o maior elemento). No entanto, sua complexidade de tempo geral permanece quadrática, $O(n^2)$, o que limita sua utilidade para conjuntos de dados maiores. A otimização reduz o número de passagens necessárias para ordenar o vetor pela metade, mas o número de comparações por passagem aumenta ligeiramente devido à necessidade de identificar tanto o mínimo quanto o máximo simultaneamente.



```
3 public class OrdenacaoQuickBubble<T extends Comparable<T>> {
67 }
68
69 // Método principal para testar a ordenação com QuickBubble Sort
70 public static void main(String[] args) {
71     Random random = new Random();
72     int tamanhoVetor = random.nextInt(91) + 100; // Gera um tamanho entre 10 e 100
73     Integer[] vetor = new Integer[tamanhoVetor];
74     for (int i = 0; i < tamanhoVetor; i++) {
75         vetor[i] = random.nextInt(1000); // Valores aleatórios entre 0 e 1000
76     }
77
78     OrdenacaoQuickBubble<Integer> ordenador = new OrdenacaoQuickBubble<>();
79     ordenador.ordenar(vetor, 10); // Limite para o uso do Bubble Sort ajustável
80     System.out.println(Arrays.toString(vetor));
81 }
82
83
```

PROBLEMAS SAÍDA **TERMINAL** GITLENS CONSOLE DE DEPURACÃO Code - Trabalhos ED2

```
• (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $ cd "/Users/rondiseba/Desktop/CP_Rondi/Trabalhos_CP/Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2/" && javac MergeSortOtimizado.java && java MergeSortOtimizado
1 2 5 9 10 11 12 14 16 18 19 21 26 27 30 31 33 35 42 44
○ (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $
```

CÓDIGO 3 - QUICKSORTMEDIANADETRES

DESCRIÇÃO GERAL DO CÓDIGO

Busquei mplementar o QuickSort com uma modificação na seleção do pivô, utilizando a mediana de três elementos (o primeiro, o meio e o último do segmento a ser ordenado, conforme atividade). Esta estratégia ajuda a evitar o pior caso do QuickSort, que ocorre quando o pivô escolhido é consistentemente o menor ou o maior elemento do segmento. Além disso, quando o segmento a ser ordenado é igual ou menor que um limite especificado, o código aplica o BubbleSort, que é mais eficiente para pequenas quantidades de dados devido à sua simplicidade e menor custo de sobrecarga.

FUNÇÕES PRINCIPAIS E COMPLEXIDADE DE TEMPO

`quickSort(int[] vetor, int esq, int dir, int limiteBubbleSort)`

- Função: Ordena o vetor usando uma abordagem híbrida de QuickSort com BubbleSort para pequenos segmentos.
- Complexidade de Tempo: No caso médio, a complexidade é $O(n \log n)$. Esta complexidade é mantida devido ao uso eficiente do QuickSort para a maioria das divisões. No entanto, o uso de BubbleSort para segmentos menores tem uma complexidade de $O(k^2)$, sendo k o tamanho do segmento. Portanto, a eficiência geral depende da escolha adequada do limite para aplicar o BubbleSort.

`particionar(int[] vetor, int esq, int dir)`

- Função: Particiona o vetor escolhendo um pivô baseado na mediana de três.
- Complexidade de Tempo: $O(n)$, onde n é o número de elementos no segmento sendo particionado. Esta função executa uma passagem pelo segmento para reorganizar os elementos em relação ao pivô.

`medianaDeTres(int[] vetor, int a, int b, int c)`

- Função: Determina o valor mediano entre três elementos do vetor para melhor escolha do pivô.
- Complexidade de Tempo: $O(1)$, pois apenas realiza um número fixo de comparações entre três elementos.

`trocar(int[] vetor, int i, int j)`

Rondineli Seba Salomão

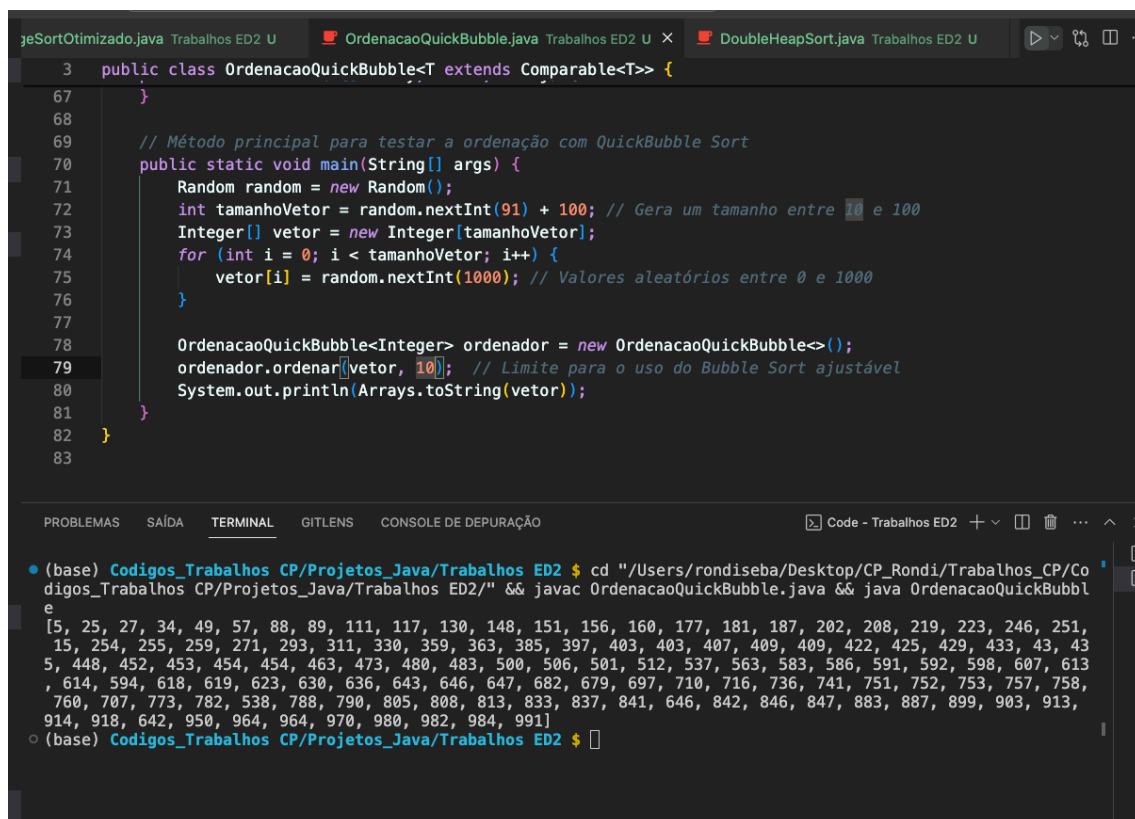
Matrícula: 20200022063

- Função: Troca dois elementos de posição no vetor.
- Complexidade de Tempo: $O(1)$, pois a troca é uma operação constante.

`bubbleSort(int[] vetor, int esq, int dir)`

- Função: Ordena um pequeno segmento do vetor usando o algoritmo BubbleSort.
- Complexidade de Tempo: $O(k^2)$, onde k é o número de elementos entre `esq` e `dir`. Esta função é eficiente para pequenos valores de k devido à simplicidade das comparações e trocas.

A combinação de QuickSort com BubbleSort em `QuickSortMedianaDeTres` apresenta uma solução eficiente para a ordenação de vetores. A seleção do pivô utilizando a mediana de três ajuda a evitar cenários de pior caso no QuickSort, enquanto a aplicação do BubbleSort para pequenos segmentos pode reduzir o custo das chamadas recursivas quando os segmentos já são pequenos. A eficácia dessa abordagem híbrida depende crucialmente da escolha apropriada do limite L , que determina quando usar BubbleSort em vez de continuar com a recursividade do QuickSort.



```
jeSortOtimizado.java Trabalhos ED2 U x OrdenacaoQuickBubble.java Trabalhos ED2 U x DoubleHeapSort.java Trabalhos ED2 U
3 public class OrdenacaoQuickBubble<T extends Comparable<T>> {
67 }
68
69 // Método principal para testar a ordenação com QuickBubble Sort
70 public static void main(String[] args) {
71     Random random = new Random();
72     int tamanhoVetor = random.nextInt(91) + 100; // Gera um tamanho entre 10 e 100
73     Integer[] vetor = new Integer[tamanhoVetor];
74     for (int i = 0; i < tamanhoVetor; i++) {
75         vetor[i] = random.nextInt(1000); // Valores aleatórios entre 0 e 1000
76     }
77
78     OrdenacaoQuickBubble<Integer> ordenador = new OrdenacaoQuickBubble<>();
79     ordenador.ordenar(vetor, 10); // Limite para o uso do Bubble Sort ajustável
80     System.out.println(Arrays.toString(vetor));
81 }
82 }
83

PROBLEMAS SAÍDA TERMINAL GITLENS CONSOLE DE DEPUAÇÃO Code - Trabalhos ED2
• (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $ cd "/Users/rondiseba/Desktop/CP_Rondi/Trabalhos_CP/Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2/" && javac OrdenacaoQuickBubble.java && java OrdenacaoQuickBubble
[5, 25, 27, 34, 49, 57, 88, 89, 111, 117, 130, 148, 151, 156, 160, 177, 181, 187, 202, 208, 219, 223, 246, 251, 15, 254, 255, 259, 271, 293, 311, 330, 359, 363, 385, 397, 403, 403, 407, 409, 409, 422, 425, 429, 433, 43, 43, 5, 448, 452, 453, 454, 454, 463, 473, 480, 483, 500, 506, 501, 512, 537, 563, 583, 586, 591, 592, 598, 607, 613, 614, 594, 618, 619, 623, 630, 636, 643, 646, 647, 682, 679, 697, 710, 716, 736, 741, 751, 752, 753, 757, 758, 760, 707, 773, 782, 538, 788, 790, 805, 808, 813, 833, 837, 841, 646, 842, 846, 847, 883, 887, 899, 903, 913, 914, 918, 642, 950, 964, 964, 970, 980, 982, 984, 991]
○ (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $
```

CÓDIGO 4

Implementei uma variação do algoritmo de ordenação HeapSort chamado DoubleHeapSort. Esta variação utiliza dois heaps: um heap mínimo e um heap máximo. A ideia é preencher um vetor de maneira que os menores elementos (retirados do heap mínimo) sejam colocados a partir do início do vetor, enquanto os maiores elementos (retirados do heap máximo) sejam colocados a partir do final do vetor, encontrando-se no meio.

ESTRUTURAS UTILIZADAS

1. Heap Mínimo (HeapMinimo)

Adiciona e remove elementos mantendo a menor chave no topo (raiz).

2. Heap Máximo (HeapMaximo)

Adiciona e remove elementos mantendo a maior chave no topo (raiz).

Funções Principais

1. Adicionar(T valor) - Para ambos os heaps (mínimo e máximo)

- Insere um elemento no heap e o reordena para manter as propriedades de heap mínimo ou máximo.

Complexidade de Tempo: $O(\log n)$ por elemento inserido, onde n é o número de elementos no heap. Isso ocorre porque, em cada inserção, pode ser necessário percorrer o caminho da posição inserida até a raiz para reajustar a estrutura do heap.

2. remover() - Para ambos os heaps (mínimo e máximo)

Remove o elemento da raiz (mínimo ou máximo, dependendo do heap), coloca o último elemento do heap na raiz e reordena para manter as propriedades do heap.

Complexidade de Tempo: $O(\log n)$, similar à inserção, pois o elemento que é colocado na raiz pode precisar ser "afundado" até a posição correta para manter as propriedades de heap.

3. ordenar(Integer[] vetor)

Utiliza os dois heaps para ordenar o vetor. Os elementos são primeiro inseridos nos dois heaps e depois removidos alternadamente (do heap mínimo para o início do vetor e do heap máximo para o final do vetor).

Rondineli Seba Salomão

Matrícula: 20200022063

Complexidade de Tempo: Inserção de todos os elementos nos heaps: $2n \cdot O(\log n)$, pois cada elemento é inserido em ambos os heaps.

Remoção de todos os elementos dos heaps e colocação no vetor: $n \cdot O(\log n)$, pois cada elemento é removido uma vez de um dos dois heaps.

A complexidade total para a função `ordenar()` é, portanto, $O(n \log n)$.

O DoubleHeapSort é um algoritmo de ordenação que usa a estrutura de dados heap para organizar os elementos. Apesar de requerer a manutenção de dois heaps, o algoritmo mantém uma complexidade de tempo $O(n \log n)$, o que é comparável a outros algoritmos de ordenação eficientes como o merge sort e o quicksort.

O DoubleHeapSort é útil em cenários onde uma distribuição balanceada dos elementos (maiores e menores alternadamente) é necessária imediatamente após a ordenação. No entanto, deve-se considerar o overhead associado ao uso de duas estruturas de heap em comparação com métodos mais simples que usam uma única estrutura.

```
jeSortOtimizado.java Trabalhos ED2 U | OrdenacaoQuickBubble.java Trabalhos ED2 U | DoubleHeapSort.java Trabalhos ED2 U x | > ~ 🔍 📄 ..
84 public class DoubleHeapSort {
85     public static void ordenar(Integer[] vetor) {
94         while (esquerda <= direita) {
97             } else {
100             }
101             esquerda++;
102             direita--;
103         }
104     }
105
106     public static void main(String[] args) {
107         Integer[] vetor = {10, 2, 8, 6, 7, 4, 3, 5, 9, 1, 0, 11, 15, 14, 13, 12, 17, 16, 19, 18};
108         ordenar(vetor);
109         for (int i : vetor) {
110             System.out.print(i + " ");
111         }
112     }
113 }
114
```

PROBLEMAS SAÍDA **TERMINAL** GITLENS CONSOLE DE DEPURAÇÃO | Code - Trabalhos ED2 + ▾ 🗑️ ... ^ x

```
• (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $ cd "/Users/rondiseba/Desktop/CP_Rondi/Trabalhos_CP/Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2/" && javac DoubleHeapSort.java && java DoubleHeapSort
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
○ (base) Codigos_Trabalhos CP/Projetos_Java/Trabalhos ED2 $
```