Trabalho - Aplicação de divide-and-conquer

Docente: Marcia Aparecida Fernandes

- Alunos:
  Arthur José dos Santos
  Jean Carlo Alves Ferreira
  Maria Gabriela Cuenca Oliva

Problema: Remoção de Duplicatas.

O problema consiste em remover eficientemente itens duplicados de um array ordenado "in-place" (ou seja, sem usar estruturas de dados adicionais) e retornar o comprimento do novo array que contém apenas elementos únicos.

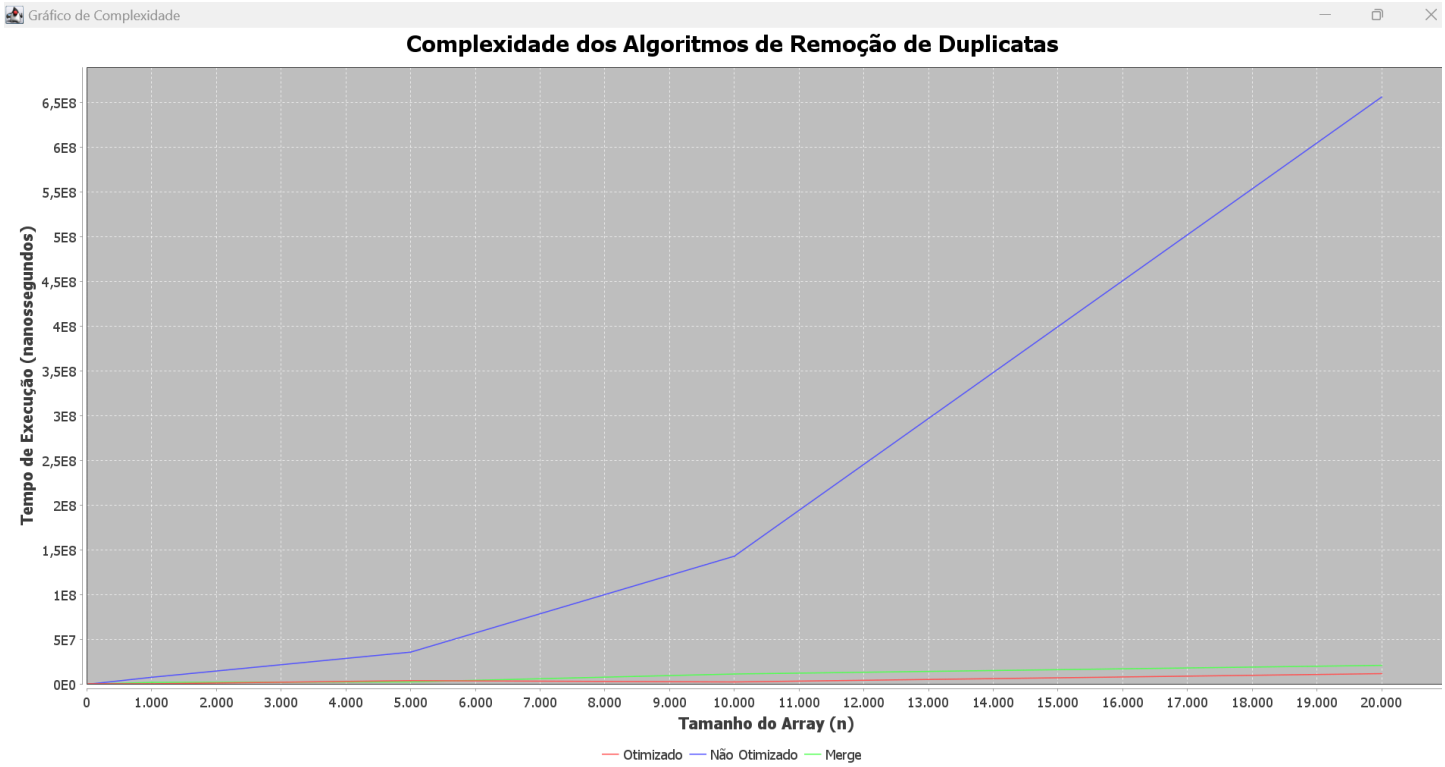
Especificação dos Algoritmos utilizados.

Para abordar deste problema, consideramos diferentes algoritmos:

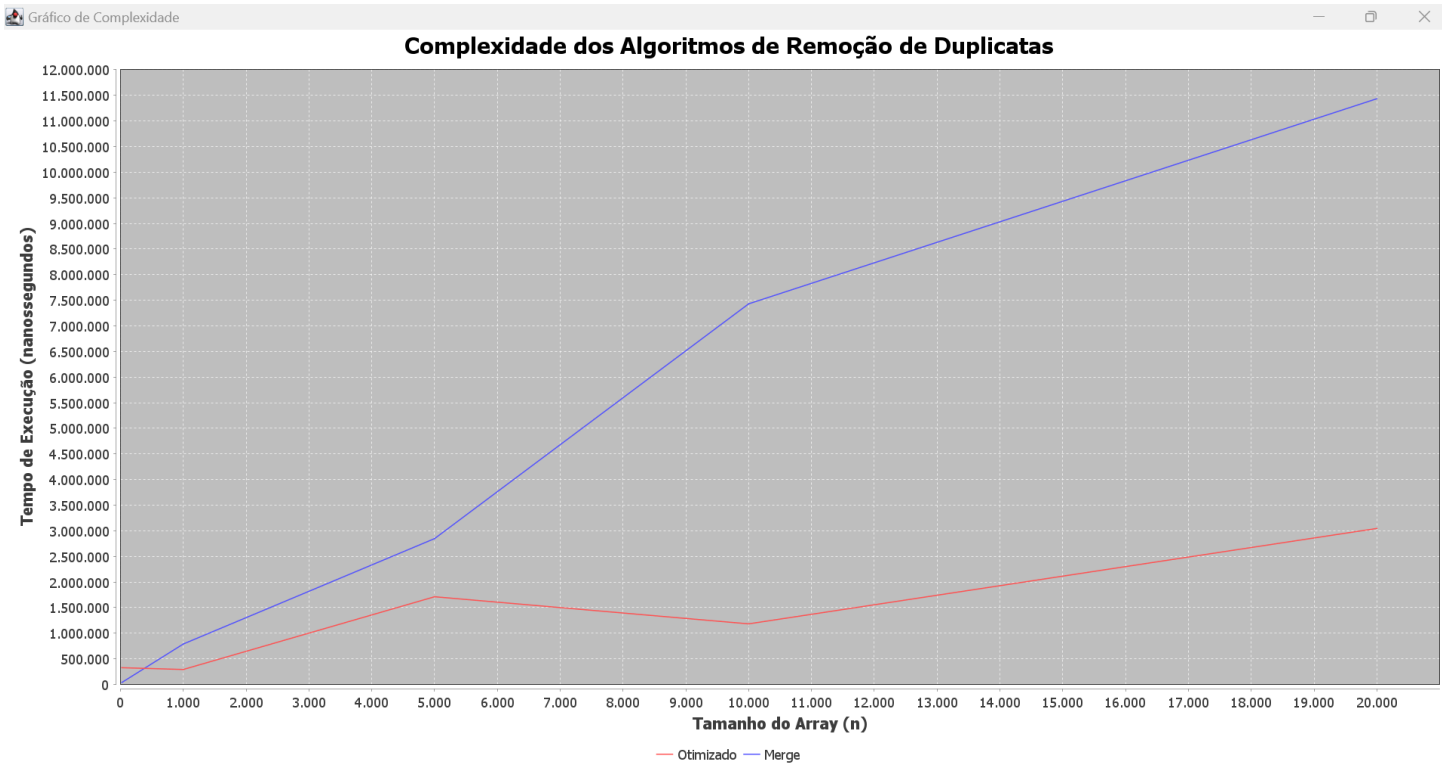
- Algoritmo Otimizado: Utiliza a ordenação do array para agrupar elementos duplicados consecutivos, e depois percorre o array para identificar e remover duplicatas. A complexidade principal é dominada pela ordenação, que é O(n log n).
- Algoritmo Não Otimizado: Usa loops aninhados para verificar duplicatas, resultando em uma complexidade quadrática O(n^2).
- Algoritmo com Merge: Aplica uma abordagem de divisão e conquista, dividindo o array recursivamente e mesclando os resultados para remover duplicatas. A complexidade é O(n log n).

Análise dos Algoritmos

Comparação Gráfica dos Três Algorítimos



Comparação dos Algorítimos Otimizado x Merge



Complexidade Temporal:

- Algoritmo Otimizado:
  A complexidade principal é dominada pela ordenação do array usando Arrays.sort(duplicatas), que é O(n log n), onde n é o tamanho do array.
  Após a ordenação, percorremos o array uma vez (O(n)) para identificar e remover duplicatas.
  Em geral, a complexidade temporal tende a ser eficiente para arrays maiores devido à eficácia da ordenação e à iteração única pelo array.

```
private static int[] removerDuplicatasOtimizado(int[] duplicatas) {
        // Verifica se o array está vazio
        if (duplicatas.length == 0) {
            return new int[0]; // Retorna um array vazio se não há elementos
        }

        // Ordena o array para agrupar elementos duplicados consecutivos
        Arrays.sort(duplicatas);

        // Inicializa o tamanho do array resultante com o primeiro elemento
        int novoTamanho = 1;

        // Percorre o array para identificar e copiar elementos únicos
        for (int i = 1; i < duplicatas.length; i++) {
            if (duplicatas[i] != duplicatas[i - 1]) {
                duplicatas[novoTamanho] = duplicatas[i];
                novoTamanho++;
            }
        }

        // Retorna um novo array com elementos únicos de tamanho novoTamanho
        return Arrays.copyOf(duplicatas, novoTamanho);
    }
```

- Algoritmo Não Otimizado:
  Utiliza dois loops aninhados para verificar duplicatas, resultando em uma complexidade quadrática O(n^2), onde n é o tamanho do array.
  Conforme o tamanho do array aumenta, o tempo de execução cresce de forma significativa, tornando-se rapidamente ineficiente para arrays maiores.

```
    private static int[] removerDuplicatasNaive(int[] duplicatas) {
        int count = 0;
        for (int i = 0; i < duplicatas.length; i++) {
            for (int j = i + 1; j < duplicatas.length; j++) {
                if (duplicatas[i] == duplicatas[j]) {
                    count++;
                    break; // Uma vez que encontramos uma duplicata, podemos sair do loop inter
                }
            }
        }

        int[] arrayUnico = new int[duplicatas.length - count];
        int index = 0;

        for (int i = 0; i < duplicatas.length; i++) {
            boolean isDuplicate = false;
            for (int j = 0; j < index; j++) {
                if (duplicatas[i] == arrayUnico[j]) {
                    isDuplicate = true;
                    break;
                }
            }
            if (!isDuplicate) {
                arrayUnico[index] = duplicatas[i];
                index++;
            }
        }

        return arrayUnico;
    }
```

- Algoritmo com Merge:
  Aplica uma abordagem de divisão e conquista, dividindo o array recursivamente e mesclando os resultados para remover duplicatas.
  A complexidade é O(n log n) devido à divisão do array em partes menores e à mesclagem subsequente.
  Este algoritmo é mais eficiente que o não otimizado, mas pode ser um pouco mais lento que o otimizado devido à natureza recursiva da divisão.

```
private static int[] removerDuplicatasPorMerge(int[] array) {
        if (array.length <= 1) {
            return array;
        }

        int meio = array.length / 2;
        int[] esquerda = removerDuplicatasPorMerge(Arrays.copyOfRange(array, 0, meio));
        int[] direita = removerDuplicatasPorMerge(Arrays.copyOfRange(array, meio, array.length)

        return mesclarArrayEsquerdaDireita(esquerda, direita);
    }

    // Método para mesclar dois arrays ordenados
    private static int[] mesclarArrayEsquerdaDireita(int[] esquerda, int[] direita) {
        int[] resultado = new int[esquerda.length + direita.length];
        int i = 0, j = 0, k = 0;

        while (i < esquerda.length && j < direita.length) {
            if (esquerda[i] < direita[j]) {
                resultado[k++] = esquerda[i++];
            } else if (esquerda[i] > direita[j]) {
                resultado[k++] = direita[j++];
            } else {
                resultado[k++] = esquerda[i++]; // Adiciona apenas um dos elementos se forem ig
                j++;
            }
        }

        // Adiciona os elementos restantes da parte não processada dos arrays
        while (i < esquerda.length) {
            resultado[k++] = esquerda[i++];
        }
        while (j < direita.length) {
            resultado[k++] = direita[j++];
        }

        // Retorna apenas a parte preenchida do array resultante
        return Arrays.copyOf(resultado, k);
    }
```

Eficiência em Relação ao Tamanho do Array:

- Algoritmo Otimizado:
  O tempo de execução aumenta de forma mais controlada com o aumento do tamanho do array,

devido à eficiência da ordenação inicial.
Para arrays muito grandes, a complexidade O(n log n) é mais sustentável do que abordagens quadráticas.

- Algoritmo Não Otimizado:
  Torna-se rapidamente ineficiente à medida que o tamanho do array cresce, devido à sua complexidade quadrática.
  O tempo de execução aumenta exponencialmente com o tamanho do array, tornando-o inadequado para arrays grandes.
- Algoritmo com Merge:
  Demonstra um desempenho moderado em relação ao tamanho do array.
  A complexidade O(n log n) permite que o algoritmo lide com arrays maiores de forma mais eficiente do que o não otimizado, embora possa ser superado pelo algoritmo estritamente otimizado em termos de tempo de execução.

Tabela Comparativa de Tempos de Execução

Tamanho do Array	Tempo (otimizado) (nanos)	Tempo (não otimizado) (nanos)
20	523800	21700
1000	474300	7743800
5000	4006600	35835900
10000	2547100	142970200
20000	11809600	655761700

Conclusão

- O Algoritmo Otimizado é geralmente o mais eficiente em termos de complexidade temporal para a remoção de duplicatas em arrays ordenados.
- O Algoritmo com Merge representa uma boa alternativa, especialmente para tamanhos de array moderados, oferecendo um equilíbrio entre eficiência e implementação simplificada.
- O Algoritmo Não Otimizado é adequado apenas para tamanhos muito pequenos de array devido à sua complexidade quadrática, tornando-se rapidamente impraticável à medida que o tamanho do array aumenta.

Em resumo, a escolha do algoritmo mais adequado depende das necessidades específicas de eficiência e do tamanho esperado do array de entrada. Para tamanhos grandes de array, o algoritmo otimizado ou o algoritmo com merge são escolhas mais sustentáveis em termos de desempenho.

Referências

- *Algoritmo Otimizado*:
  Removendo Itens Duplicados em um Array Ordenado - A Arte de Programar
  Weberton Faria, Autor em A Arte de Programar
  https://aartedeprogramar.tech/removendo-itens-duplicados-em-arrays/
- *Teorema Meste*:
  Computer science is no more about computers
  than astronomy is about telescopes.
  — Edsger W. Dijkstra
  https://www.ime.usp.br/~pf/analise_de_algoritmos/