

Ciência da Computação

4º Semestre

Lucas Pascoal Antunes – N093AB4

**Atividades Práticas Supervisionadas**

**2024**

**Índice**

Índice……………………………………………………………………………………..2

Objetivo…………………………………………………………………………………..4

Introdução………………………………………………………….…………………….5

Escolha do tema………………………………………………………………………...6

Dissertação………………………………………………………………………………8

Projeto do site…………………………………………………………………………...9

Botão Inicial………………………………………………………………………9

Botão dos Criadores…………………………………………………………...10

Logo……………………………………………………………………………...11

2 Etapa…………………………………………………………………………..12

3 Etapa…………………………………………………………………………..13

Linhas de Código……………………………………………………………………..14

HTML……………………………………………………………………………14

CSS……………………………………………………………………………..17

Javascript……………………………………………………………………….21

Considerações Finais……………………………………………………………….23

Bibliografia…………………………………………………………………………...24

Ficha APS…………………………………………………………………………....25

**OBJETIVO**

O objetivo maior deste trabalho é desenvolver um sistema de análise e comparação de algoritmos de ordenação, utilizando como contexto o monitoramento de dados ambientais da floresta amazônica. A proposta vai além da implementação técnica de algoritmos de ordenação; é uma oportunidade de explorar como a ciência de dados pode contribuir para a preservação ambiental.

Atualmente, a Amazônia enfrenta ameaças como o desmatamento e as queimadas ilegais, problemas que afetam o ecossistema e o clima global. A tecnologia é uma aliada fundamental na fiscalização desses crimes, e, nesse sentido, as imagens de satélite desempenham um papel crucial. Esses satélites geram diariamente uma quantidade massiva de dados que, para serem analisados com eficiência, precisam estar ordenados e organizados.

Este trabalho visa não apenas demonstrar o funcionamento de algoritmos clássicos de ordenação, como QuickSort, MergeSort e HeapSort, mas também revelar suas forças e fraquezas ao processar diferentes volumes e estruturas de dados. Assim, o sistema que estamos desenvolvendo simula uma situação real: ele organiza e cataloga grandes conjuntos de dados com base na sua origem e condição – preservado, queimado ou desmatado. Ao explorar cenários variados (como conjuntos de dados já ordenados, parcialmente ordenados ou totalmente desordenados), podemos identificar o algoritmo mais adequado para cada situação e compreender como ele impacta a análise e o tempo de resposta para a tomada de decisões.

Além de analisar o desempenho dos algoritmos, o trabalho busca reforçar o papel da computação no monitoramento ambiental. Organizar dados de forma eficiente permite que órgãos de fiscalização, como o IBAMA e o INPE, possam acessar e interpretar rapidamente as informações, tornando mais ágil o combate às atividades ilegais.

Por fim, o objetivo maior é contribuir, ainda que de forma acadêmica e experimental, para um entendimento mais profundo sobre a importância do processamento de dados no contexto de proteção ambiental. Este sistema ilustra como a aplicação prática do conhecimento técnico, ao ser bem direcionada, pode ter impacto em problemas reais e relevantes para o nosso futuro. Ao desenvolver este trabalho, não apenas aprendemos sobre algoritmos, mas também compreendemos melhor como a tecnologia pode auxiliar em um propósito que é de interesse global: a preservação da Amazônia e de sua biodiversidade.

**Introdução**

Nos últimos anos, a proteção ambiental e a preservação dos recursos naturais têm se tornado uma prioridade global. A floresta amazônica, em especial, é um ecossistema fundamental para o equilíbrio climático e a biodiversidade, abrigando milhares de espécies e absorvendo quantidades significativas de dióxido de carbono. No entanto, ameaças como o desmatamento e as queimadas ilegais colocam em risco essa região única, comprometendo não apenas o ecossistema local, mas também o clima global. Essas atividades, que muitas vezes ocorrem em áreas remotas e de difícil acesso, desafiam as autoridades e organizações ambientais a encontrar maneiras eficazes de monitoramento e fiscalização.

Para lidar com essa complexidade, a tecnologia tem desempenhado um papel central, especialmente por meio do uso de satélites que capturam imagens em alta resolução da floresta amazônica. Esses satélites produzem diariamente grandes volumes de dados, gerando imagens que documentam o estado da floresta e registram mudanças na cobertura vegetal, permitindo que áreas desmatadas e focos de incêndio sejam identificados quase em tempo real. Essa vasta quantidade de dados é essencial para órgãos como o Instituto Nacional de Pesquisas Espaciais (INPE) e o Instituto Brasileiro do Meio Ambiente e dos Recursos Naturais Renováveis (IBAMA), mas, por sua magnitude, precisa ser organizada e processada rapidamente para que decisões e intervenções sejam feitas em tempo hábil. A organização desses dados, portanto, é um passo crítico no processo de monitoramento ambiental, pois possibilita uma análise mais precisa e ágil.

Os algoritmos de ordenação de dados surgem como uma ferramenta essencial nesse processo. Com eles, é possível organizar e catalogar a imensa quantidade de informações coletadas, facilitando a localização de áreas de interesse e acelerando o acesso aos dados mais relevantes. No contexto do monitoramento ambiental, a escolha de algoritmos eficientes de ordenação pode fazer a diferença entre uma resposta rápida e um atraso significativo na tomada de decisões. Neste projeto, propomos a criação de um sistema de análise de performance de algoritmos de ordenação que simula as demandas de um sistema de monitoramento ambiental, permitindo explorar a eficiência de diferentes técnicas de ordenação em grandes volumes de dados.

Para essa análise, foram selecionados três algoritmos clássicos de ordenação: QuickSort, MergeSort e HeapSort. Cada um desses algoritmos possui características distintas, com diferentes métodos para organizar e processar dados. QuickSort, por exemplo, é conhecido por sua eficiência em muitos casos, utilizando a técnica de particionamento e escolha de pivô. É um dos algoritmos mais rápidos em dados de tamanho médio, mas pode enfrentar dificuldades em conjuntos de dados altamente desordenados. MergeSort, por outro lado, adota a técnica de divisão e conquista e apresenta um desempenho estável em grandes volumes de dados, sendo ideal para conjuntos grandes, pois mantém sua complexidade O(nlogn) independentemente da ordem inicial dos dados. Já o HeapSort se destaca pela sua eficiência com dados aleatórios e pela vantagem de não exigir armazenamento adicional significativo, sendo uma escolha interessante em contextos onde o uso de memória é uma preocupação.

Com esses três algoritmos, o sistema proposto realiza testes de desempenho e compara o tempo de execução em diferentes cenários de ordenação, com dados que simulam a realidade de imagens de satélite da Amazônia. Utilizamos tanto dados externos, que representam informações já armazenadas em arquivos, quanto dados internos gerados aleatoriamente, para criar condições que variam entre dados totalmente desorganizados, parcialmente organizados e organizados. Essa abordagem permite simular diversas situações reais, como imagens recém-coletadas e dados de monitoramento já catalogados, além de comparar os resultados obtidos em cada cenário para entender as vantagens e limitações de cada algoritmo em termos de velocidade e eficiência.

O sistema desenvolvido é interativo, permitindo que o usuário escolha o algoritmo a ser testado e visualize os resultados comparativos de desempenho. Com essa estrutura modular, os testes podem ser realizados de forma prática e organizada, possibilitando uma análise detalhada do comportamento de cada técnica. Ao final dos testes, o sistema apresenta métricas de tempo de execução, número de comparações e trocas, além de possibilitar o salvamento dos dados organizados para consultas futuras. Dessa forma, o sistema permite identificar qual técnica de ordenação é mais eficaz em diferentes situações, fornecendo uma base sólida para avaliar o impacto de cada algoritmo no processamento de dados ambientais.

Mais do que um estudo técnico, este projeto busca mostrar como a organização eficiente dos dados pode contribuir para a proteção da Amazônia. Em um sistema de monitoramento ambiental, onde tempo e precisão são cruciais, entender as nuances dos algoritmos de ordenação nos ajuda a propor estratégias mais eficazes de processamento. Embora este projeto seja uma simulação acadêmica, ele ilustra a importância da ciência de dados para resolver problemas reais. Com a análise do desempenho de diferentes algoritmos, podemos não apenas entender melhor os aspectos técnicos de cada um, mas também refletir sobre como essa eficiência pode ser aplicada em sistemas de monitoramento reais, contribuindo para que órgãos ambientais tomem decisões rápidas e fundamentadas.

Portanto, o trabalho realizado vai além de uma simples comparação entre algoritmos de ordenação; ele representa um estudo sobre como a tecnologia e a ciência de dados podem auxiliar na preservação da floresta amazônica. Cada algoritmo testado e cada métrica analisada nos ajuda a visualizar um cenário onde o uso adequado de tecnologia e processamento de dados pode reforçar as ações de fiscalização e controle ambiental. Este projeto, ao combinar a teoria dos algoritmos com um contexto de aplicação real, destaca o papel fundamental da computação na proteção de um dos maiores patrimônios naturais do planeta e reforça a ideia de que o conhecimento técnico, quando bem aplicado, pode ter um impacto profundo na preservação do meio ambiente.

**Referencial Teórico**

O projeto desenvolvido utiliza conceitos fundamentais de algoritmos de ordenação e processamento de dados para criar um sistema que ajuda a organizar informações de maneira eficiente. A motivação central é aplicar técnicas de ordenação em um contexto de monitoramento ambiental, simulando a análise de dados que poderiam ser gerados pelo monitoramento da floresta amazônica. Em um ambiente real, satélites capturam uma quantidade massiva de dados de imagem todos os dias, e esses dados precisam ser organizados para serem úteis na identificação de áreas de desmatamento e queimadas. A organização e análise desses dados de forma ágil é essencial para permitir que as autoridades tomem decisões rápidas e precisas para proteger o ecossistema.

O sistema de ordenação desenvolvido foi estruturado para comparar três algoritmos clássicos de ordenação – QuickSort, MergeSort e HeapSort – que são amplamente estudados devido à sua eficiência e aplicabilidade em diversos tipos de dados. Estes algoritmos, cada um com suas peculiaridades, servem para organizar informações de forma rápida e precisa, o que é crucial no contexto de grandes volumes de dados. A implementação em Java foi escolhida pela sua robustez e facilidade de manipulação de dados complexos, o que facilita o desenvolvimento de um sistema de ordenação de grande escala.

O QuickSort é o primeiro algoritmo de ordenação escolhido e implementado no sistema, sendo conhecido por sua eficiência em muitos casos práticos. Ele usa a técnica de "divisão e conquista", onde o conjunto de dados é dividido em partes menores a partir de um elemento central chamado de "pivô". Esse pivô serve como uma referência para dividir os dados em valores menores e maiores, permitindo que a ordenação aconteça em diversas partes de maneira independente. No código, uma versão aprimorada do QuickSort foi implementada, usando uma técnica de “mediana de três” para escolher o pivô, o que reduz as chances de o algoritmo cair em seu pior caso de desempenho, que seria O(n2).

Uma característica interessante do QuickSort é que ele realiza a ordenação no próprio array, sem demandar muito espaço extra, o que economiza memória. Essa economia é valiosa em contextos como o monitoramento ambiental, onde muitos dados precisam ser processados e a memória disponível pode ser limitada. No entanto, o QuickSort depende muito de como os dados estão organizados inicialmente; se os dados estiverem em uma ordem que não favoreça o particionamento, o desempenho do algoritmo pode cair significativamente. Por isso, no sistema desenvolvido, o QuickSort é utilizado em situações onde o conjunto de dados é grande, mas não está em uma ordem extremamente desorganizada.

O MergeSort foi escolhido como o segundo algoritmo, principalmente por sua estabilidade e por manter um desempenho constante de O(nlogn) em todos os casos. Diferente do QuickSort, o MergeSort sempre divide o conjunto de dados ao meio, garantindo uma divisão equilibrada independentemente da ordem inicial dos elementos. Esse algoritmo é especialmente útil para dados de grande volume, como os que poderiam ser gerados por imagens de satélite da floresta amazônica. A estabilidade no tempo de execução é uma vantagem quando precisamos de previsibilidade e confiabilidade no desempenho, independentemente do estado inicial dos dados.

No MergeSort, após dividir o conjunto de dados em partes menores, o algoritmo começa a "mesclar" esses dados em ordem, até que o conjunto completo esteja organizado. No código, a implementação do MergeSort requer um espaço extra para realizar essa mesclagem, o que pode ser um ponto negativo em sistemas com restrições de memória. No entanto, esse algoritmo é adequado para situações em que o sistema pode utilizar um pouco mais de espaço para garantir que a ordenação seja feita de forma confiável e rápida. Em cenários de monitoramento ambiental, onde muitas vezes lidamos com conjuntos de dados grandes e em constante crescimento, o MergeSort é uma escolha prática pela sua eficiência estável.

O HeapSort é o terceiro algoritmo incluído no sistema e se destaca por sua estrutura única, baseada em uma árvore binária chamada de "heap". Essa estrutura permite que o HeapSort organize os dados extraindo o maior (ou menor) elemento do heap e colocando-o na posição correta, repetidamente, até que o conjunto esteja ordenado. Diferente do MergeSort, o HeapSort não requer espaço extra significativo, pois a ordenação é realizada diretamente no array de dados, tornando-o uma boa escolha em contextos onde a memória deve ser economizada.

Uma vantagem do HeapSort é que ele mantém uma complexidade de O(nlogn) e não depende da ordem inicial dos dados, o que o torna mais flexível para dados que estão em qualquer nível de desorganização. No sistema desenvolvido, o HeapSort foi implementado para testes de eficiência em dados altamente desordenados, pois ele tem um desempenho estável em uma grande variedade de cenários. Em aplicações de monitoramento ambiental, o HeapSort pode ser útil para organizar dados de maneira econômica e confiável, mesmo em situações onde os dados estão em um estado caótico.

O sistema foi estruturado para permitir que o usuário escolha o algoritmo de ordenação a ser testado e visualize os resultados de desempenho de forma clara. Ele utiliza tanto dados externos, que simulam informações já coletadas, quanto dados gerados aleatoriamente para representar diferentes cenários de ordenação. O código é projetado para medir o tempo de execução de cada algoritmo, assim como o número de comparações e trocas realizadas, fornecendo uma visão prática do desempenho de cada técnica. Essas métricas são fundamentais para identificar o algoritmo mais eficiente para diferentes situações.

Para cada algoritmo, o sistema registra as métricas de desempenho e permite que o usuário compare os resultados de maneira direta. Assim, é possível observar, por exemplo, que o QuickSort se sai melhor em dados aleatórios moderadamente desordenados, enquanto o MergeSort é mais estável em grandes volumes de dados, e o HeapSort pode oferecer uma solução econômica em termos de memória.

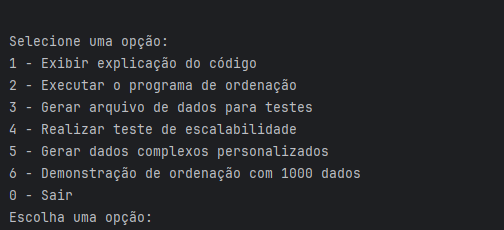
Mais do que um exercício acadêmico, esse sistema simula o uso real de algoritmos de ordenação para auxiliar na organização de dados ambientais. Em um cenário de monitoramento, onde milhares de dados precisam ser organizados rapidamente para análise, a eficiência do algoritmo escolhido pode impactar diretamente a velocidade com que uma intervenção é planejada e executada. O QuickSort, MergeSort e HeapSort, cada um com suas características específicas, demonstram a importância de escolher a técnica certa de acordo com o tipo e o volume dos dados.

A análise de desempenho e os resultados fornecidos pelo sistema não apenas ajudam a entender as nuances de cada algoritmo, mas também ilustram como a ciência de dados pode ser aplicada para resolver problemas reais. Otimizar a organização e o processamento de grandes volumes de dados é essencial para proteger e monitorar o meio ambiente. A floresta amazônica, por exemplo, se beneficiaria diretamente de um sistema de análise ágil e preciso que fosse capaz de processar dados em tempo hábil, permitindo que autoridades e pesquisadores respondam rapidamente a situações críticas.

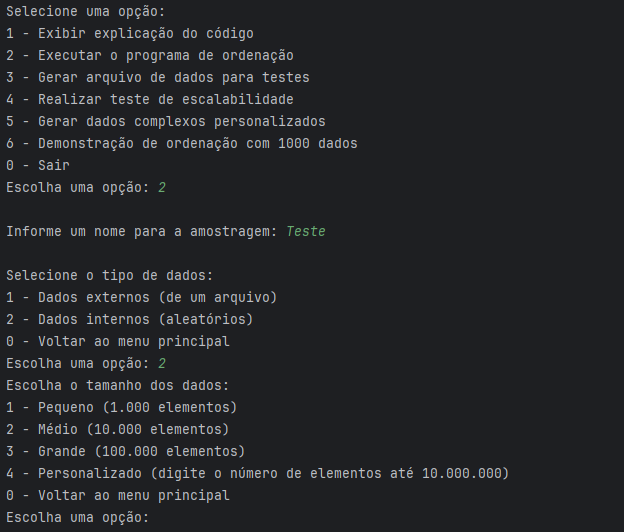
Esse sistema de análise e comparação de algoritmos fornece uma base prática para entender a aplicabilidade de cada técnica de ordenação em diferentes cenários, e reforça a importância da eficiência no processamento de dados. Ao aplicar esses conceitos, este trabalho contribui para o desenvolvimento de um sistema que, em um contexto maior, pode ser essencial para o monitoramento ambiental.

**Desenvolvimento**

O desenvolvimento deste sistema de ordenação foi guiado pelo desejo de criar uma ferramenta interativa que fosse além de uma simples aplicação de ordenação, permitindo aos usuários não só ordenar dados, mas também visualizar e compreender as diferenças de desempenho entre diferentes algoritmos de ordenação. Desde o início, o objetivo foi desenvolver uma plataforma completa que possibilitasse testes e análises práticas, tudo em um contexto inspirado em aplicações reais de monitoramento ambiental. A ideia central era simular o tipo de processamento de dados que ocorre em sistemas que utilizam dados de satélite para monitorar áreas extensas como a floresta amazônica, onde é essencial processar grandes volumes de dados rapidamente para auxiliar na fiscalização e preservação.



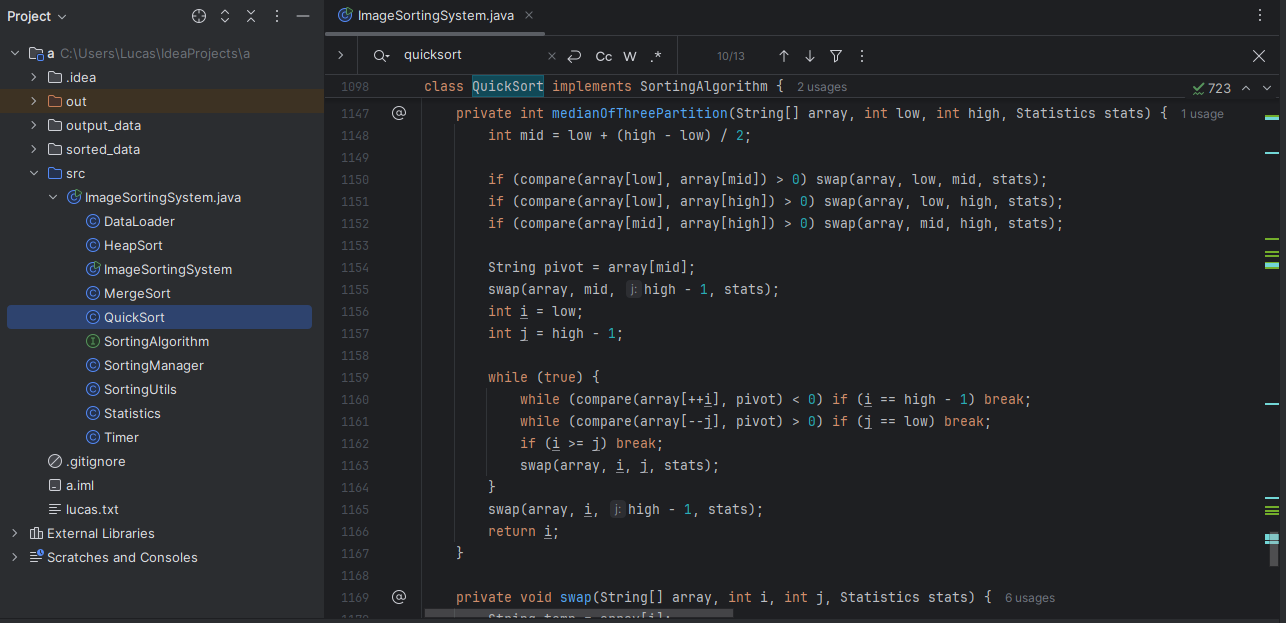
As primeiras etapas do desenvolvimento focaram na definição da estrutura geral e na escolha de uma arquitetura modular. A modularidade era essencial para permitir que cada algoritmo fosse implementado e testado como uma unidade independente. Isso trouxe duas vantagens importantes: organização e facilidade de manutenção. Com essa abordagem, foi possível isolar a lógica de cada algoritmo em módulos separados, o que tornava o código mais fácil de entender e permitia que novos algoritmos fossem adicionados sem impactar o restante do sistema. Essa decisão também abriu espaço para melhorias e otimizações futuras, pois cada módulo podia ser aprimorado sem afetar o sistema como um todo. Ao escolher Java como linguagem de implementação, aproveitamos sua robustez e suporte a operações em larga escala, características importantes para um sistema que precisava lidar com grandes conjuntos de dados de forma eficiente.

O próximo passo foi implementar um menu de opções que possibilitasse ao usuário interagir com o sistema de forma intuitiva. Esse menu inicial foi projetado para facilitar a navegação e garantir que o usuário pudesse explorar todas as funcionalidades do sistema. Em vez de limitar o sistema a uma execução direta de ordenação, foi criado um menu com opções para executar diferentes tipos de ordenação, visualizar explicações sobre os algoritmos e realizar testes em dados externos ou gerados internamente. O sistema também permite que o usuário configure o tipo de dado a ser ordenado, seja carregando dados de um arquivo externo que simule dados de satélite, seja criando dados aleatórios no próprio sistema. Além disso, uma funcionalidade foi incluída para que o usuário pudesse gerar dados personalizados, configurando características específicas como áreas geográficas e estados de conservação para simular dados próximos à realidade do monitoramento ambiental. 

Uma vez estabelecida a interface e a estrutura modular, o foco passou a ser a implementação dos algoritmos de ordenação em si. Cada algoritmo foi desenvolvido com atenção a suas especificidades. A implementação do QuickSort, por exemplo, foi feita com uma técnica conhecida como "mediana de três" para escolha do pivô, uma abordagem que visa reduzir a probabilidade de um desempenho abaixo do esperado ao garantir que o pivô divida o conjunto de dados de maneira mais equilibrada. No caso do QuickSort, essa técnica é especialmente relevante, pois ajuda a evitar o pior cenário em que o algoritmo pode cair, onde o desempenho cai de O(nlogn) para O(n2)O(n^2)O(n2), tornando-o ineficiente para grandes volumes de dados desorganizados. Outro aspecto interessante do QuickSort é que ele é um algoritmo que opera no próprio array, sem exigir uma grande quantidade de memória adicional, o que o torna uma opção eficiente para muitos casos práticos.

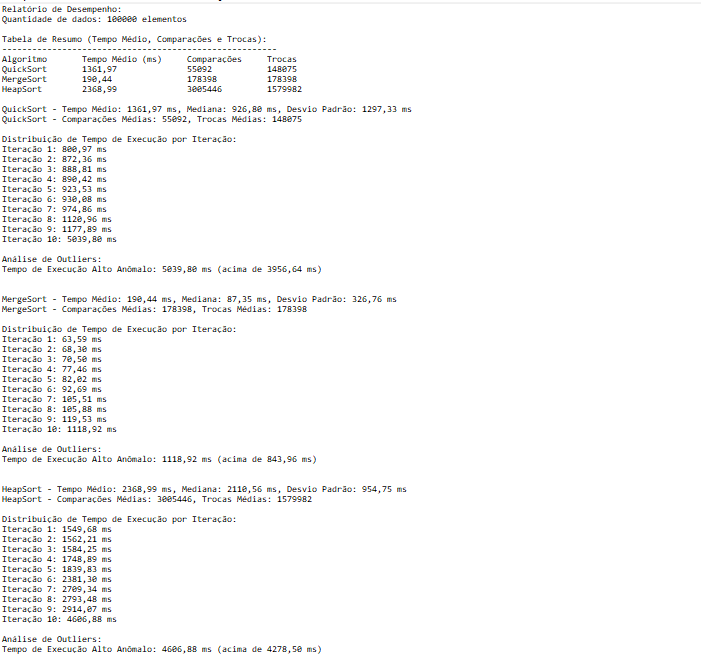
O MergeSort, por sua vez, foi implementado com foco em garantir um desempenho estável e consistente. Esse algoritmo é conhecido por dividir o conjunto de dados em partes menores, organizando cada uma delas individualmente antes de mesclar todas as partes para obter o conjunto ordenado final. A estabilidade do MergeSort foi uma característica que buscamos explorar ao máximo, pois em um sistema de monitoramento de dados ambientais é importante garantir previsibilidade de desempenho. Isso é particularmente relevante para grandes conjuntos de dados que podem ter milhares de registros capturados em áreas geográficas diversas e em diferentes estados de conservação. A complexidade do MergeSort é consistentemente O(nlogn), e ele garante uma ordenação estável, o que significa que mantém a ordem relativa dos dados iguais. Esse ponto é útil para o monitoramento ambiental, pois os dados podem incluir múltiplos atributos, como localização e estado de preservação, e a estabilidade do MergeSort permite que essas informações sejam mantidas na mesma sequência, facilitando a análise.

Já o HeapSort foi implementado para atender a cenários onde a economia de memória é uma prioridade. Este algoritmo utiliza uma estrutura de heap para organizar os dados, o que permite que a ordenação seja realizada diretamente no array, sem a necessidade de uma memória auxiliar significativa. Isso torna o HeapSort uma opção interessante para contextos onde a memória é limitada e onde a estabilidade da ordenação não é necessária. Embora o HeapSort não mantenha a ordem relativa de dados iguais, ele oferece um desempenho sólido e previsível em situações onde os dados estão altamente desorganizados. Durante a implementação, optamos por uma abordagem iterativa para a estruturação do heap, o que minimiza o risco de estouro de pilha e aumenta a robustez do sistema ao lidar com grandes volumes de dados.



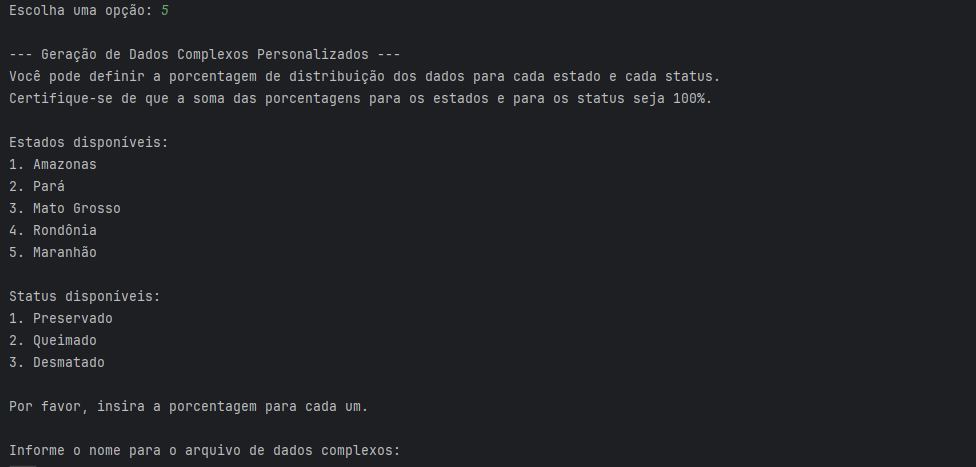
Além da implementação dos algoritmos, o sistema também foi configurado para coletar e exibir métricas de desempenho, como tempo de execução, número de comparações e número de trocas realizadas durante o processo de ordenação. Essas métricas foram integradas ao sistema para que o usuário possa observar, de maneira prática, as diferenças de desempenho entre cada algoritmo em tempo real. Cada execução termina com um relatório de desempenho que permite ao usuário fazer uma análise direta dos resultados, facilitando a identificação de padrões e a escolha do algoritmo mais eficiente para diferentes cenários. Para maximizar a utilidade dos resultados, o sistema também salva os dados ordenados em arquivos de saída, permitindo que o usuário consulte e compare os conjuntos de dados ordenados por diferentes algoritmos.

Outro ponto forte do sistema é a funcionalidade de testes de escalabilidade. Essa função foi desenvolvida para simular o crescimento dos conjuntos de dados, permitindo que o usuário observe como cada algoritmo responde ao aumento do volume de dados. O sistema gera conjuntos de dados de tamanhos progressivos e aplica os algoritmos de ordenação, registrando o tempo de execução e o número de operações realizadas em cada execução. Essa funcionalidade é especialmente útil em um contexto de monitoramento ambiental, onde o volume de dados coletados aumenta constantemente e é necessário que o sistema de ordenação consiga acompanhar esse crescimento sem comprometer a eficiência. Além disso, o sistema permite que os testes de escalabilidade sejam interrompidos, o que evita travamentos e dá mais controle ao usuário ao lidar com conjuntos de dados massivos.



Uma atenção especial foi dada ao tratamento de erros e à validação das entradas do usuário. Para garantir a robustez do sistema, ele foi configurado para exibir mensagens de orientação caso o usuário insira valores inválidos, e para interromper operações se detectados erros críticos. Esse cuidado com a validação torna o sistema mais confiável e garante que ele possa ser utilizado em diferentes condições sem problemas inesperados. A inclusão de tratamento de erros também foi pensada para garantir que o sistema seja resiliente, permitindo ao usuário experimentar diferentes configurações de teste sem risco de comprometer o funcionamento do programa.

Durante o desenvolvimento, também foi incluída uma funcionalidade para a criação de dados personalizados, permitindo que o usuário configure os conjuntos de dados com atributos específicos, como porcentagens de dados preservados, queimados ou desmatados, além de características regionais. Essa possibilidade de personalização torna a simulação mais próxima da realidade de um sistema de monitoramento ambiental, já que o usuário pode criar cenários específicos que refletem o tipo de dados que realmente seriam coletados em um monitoramento ambiental. A configuração de dados personalizados adiciona uma camada de realismo ao sistema, permitindo que o usuário simule, por exemplo, cenários em que grande parte dos dados representa áreas de conservação em risco, possibilitando uma análise mais direcionada.



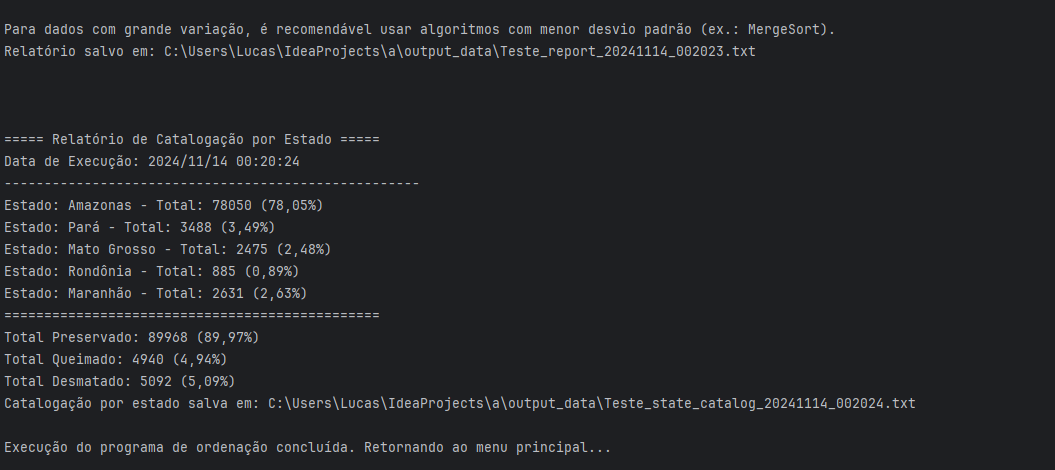
O resultado final é um sistema de ordenação interativo, robusto e flexível, que vai além de uma simples aplicação de ordenação. Ele permite que o usuário explore os algoritmos em detalhes, visualize o desempenho e compreenda as vantagens e limitações de cada técnica em diferentes cenários. Em última análise, o sistema não só ordena dados, mas também oferece insights sobre a eficiência de cada algoritmo em lidar com grandes volumes de dados, trazendo uma abordagem prática e educativa que demonstra a importância de escolher o algoritmo certo para o tipo de dado a ser processado. Esse projeto mostra como um sistema eficiente de ordenação pode otimizar a análise de dados em um cenário de monitoramento ambiental, contribuindo para uma fiscalização mais ágil e precisa.  
Ao projetar o sistema, outro ponto que exigiu atenção foi a integração entre os algoritmos de ordenação e as funcionalidades de geração, salvamento e recuperação de dados. Criar um fluxo que conectasse esses elementos de forma fluida envolveu desafios técnicos, principalmente para garantir que o processo fosse eficiente mesmo em grandes volumes de dados. Uma das decisões foi a inclusão de métodos de armazenamento temporário, onde cada execução de ordenação poderia ser pausada ou interrompida sem perder o estado atual dos dados. Com isso, o sistema pode registrar o progresso, permitir que o usuário acompanhe o andamento das ordenações mais complexas e retome a operação sem reiniciar o processo. Esse recurso trouxe maior controle sobre o sistema e foi útil durante o desenvolvimento, pois facilitou o teste de algoritmos e ajustes sem perda de dados ou necessidade de recomeçar operações complexas do zero.

Em relação à geração de dados, a implementação de métodos para simular cenários com dados realistas foi essencial. Os dados são gerados a partir de distribuições configuráveis, permitindo que o sistema simule cenários próximos da realidade de um monitoramento ambiental, como áreas protegidas e regiões mais suscetíveis ao desmatamento. Esse nível de controle na geração de dados foi possível com o uso de uma lógica de geração ponderada que ajusta as características dos dados conforme as porcentagens definidas pelo usuário. Por exemplo, se o usuário definir 40% dos dados como "preservados" e 60% como "queimados", o sistema respeitará essas proporções, criando dados que refletem a configuração escolhida. Essa funcionalidade tornou o sistema não apenas uma ferramenta de ordenação, mas também um simulador para estudos em condições personalizadas, auxiliando no entendimento do impacto de certos algoritmos sobre diferentes perfis de dados.

Outro aspecto importante é a gestão de memória, especialmente nas operações de ordenação com grandes conjuntos de dados. Como o sistema foi projetado para simular o comportamento com até milhões de dados, o gerenciamento de memória foi uma preocupação constante. Em Java, cada elemento do array ocupa uma porção de memória, e em dados muito grandes isso pode representar um desafio. Para contornar esse problema, o código foi otimizado com técnicas de coleta de lixo automático, reutilização de objetos sempre que possível e minimização de alocações redundantes, evitando o uso excessivo de memória. Além disso, foi criado um sistema de monitoramento de uso de memória em tempo real que alerta o usuário caso o limite de memória esteja próximo de ser atingido, permitindo que o usuário ajuste o tamanho do conjunto de dados de acordo com as capacidades do sistema.

A interface de entrada e saída de dados foi projetada para ser simples, porém robusta. A leitura de arquivos de entrada, por exemplo, precisou lidar com possíveis erros de formatação. Foi implementado um sistema de validação que verifica cada linha dos arquivos de dados antes de processá-los, assegurando que estejam no formato correto para evitar erros na execução dos algoritmos. Esse sistema de validação não só aumenta a robustez do sistema, mas também evita que o usuário precise formatar manualmente os arquivos, tornando o processo mais acessível e menos propenso a erros.

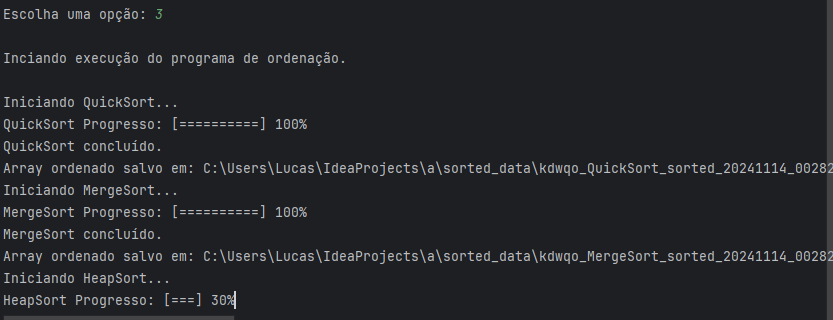
Na saída dos dados, o sistema permite salvar não apenas o conjunto ordenado, mas também todas as métricas de desempenho geradas. O sistema cria relatórios de cada execução em arquivos de texto organizados por data e algoritmo, registrando dados como tempo de execução, comparações, trocas e uso de memória. Essa decisão foi motivada pela necessidade de documentar o desempenho do sistema ao longo do tempo, permitindo ao usuário comparar diferentes execuções e identificar padrões. Com isso, é possível analisar se o desempenho de um algoritmo específico melhora ou piora conforme o volume de dados aumenta ou conforme as configurações de hardware variam. Esse recurso de monitoramento é importante em contextos de estudo e otimização de algoritmos, já que permite revisitar os resultados e compreender melhor o comportamento dos algoritmos em cenários de execução prolongada.



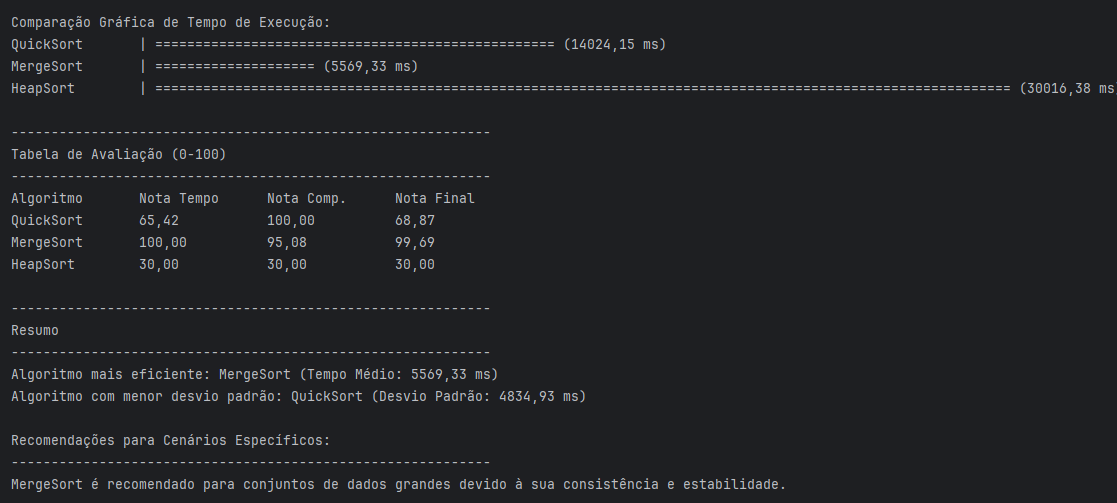
A escolha por implementar o sistema em Java trouxe vantagens em termos de portabilidade, já que o código pode ser executado em várias plataformas sem necessidade de adaptação. No entanto, também exigiu uma atenção especial à otimização para minimizar o impacto do uso de memória e garantir que o sistema pudesse lidar com grandes volumes de dados. Como Java é uma linguagem gerenciada, seu coletor de lixo desempenha um papel importante na liberação de memória, mas também pode afetar o desempenho se não for controlado. Para reduzir esses impactos, o sistema foi desenvolvido com um perfil otimizado para evitar que o coletor de lixo seja acionado com muita frequência, o que poderia interromper a execução dos algoritmos. Além disso, foi feito um esforço para evitar a criação de objetos desnecessários, priorizando a manipulação direta de arrays e variáveis primitivas quando possível.

A coleta de métricas foi projetada para oferecer ao usuário uma visão completa do desempenho de cada algoritmo. Além do tempo de execução, foram incluídas métricas como número de comparações e trocas, que ajudam a entender a "intensidade" das operações realizadas. Essas métricas foram armazenadas de maneira incremental, para que o sistema registrasse não apenas o resultado final, mas também o progresso ao longo do tempo. Essa escolha permite ao usuário identificar em qual fase da ordenação os algoritmos tendem a gastar mais recursos, oferecendo insights sobre possíveis otimizações. Por exemplo, no caso do QuickSort, o usuário pode observar se as operações de particionamento estão mais intensas no início ou no final da execução, indicando se as escolhas de pivô estão sendo eficazes.

A implementação de uma barra de progresso e indicadores visuais para operações longas foi outra adição que contribuiu para a experiência do usuário. Para execuções de ordenação que podem levar minutos ou até horas em grandes conjuntos de dados, essa barra de progresso permite que o usuário tenha uma ideia do tempo restante, oferecendo uma experiência de uso mais satisfatória. Esse recurso também ajuda a evitar a frustração do usuário ao esperar por resultados, pois permite visualizar o andamento da operação e decidir se quer interromper ou continuar. O sistema de interrupção, por sua vez, foi implementado com um sinalizador (flag) que permite ao usuário pausar a execução sem causar perda de dados. Essa capacidade de controle aumenta a flexibilidade do sistema e permite uma abordagem mais dinâmica para testes com dados de alta complexidade.



A criação de um sistema de pontuação para os algoritmos foi outro ponto interessante no desenvolvimento. Esse sistema de pontuação, baseado em tempo, comparações e trocas, permite uma comparação direta entre os algoritmos, atribuindo uma nota geral de desempenho para cada um. Com isso, o sistema consegue avaliar qual algoritmo foi mais eficiente em cada cenário e oferecer uma recomendação com base no tipo de dado e na estrutura do conjunto de dados. Essa funcionalidade, além de permitir uma análise técnica dos algoritmos, torna o sistema acessível a usuários com menos familiaridade com os detalhes de algoritmos, pois oferece uma indicação clara sobre qual técnica de ordenação é mais adequada para cada tipo de tarefa.



No geral, o desenvolvimento do sistema se destacou por sua abordagem detalhada e pela integração de várias funcionalidades que, juntas, resultam em uma ferramenta completa para análise de algoritmos de ordenação. Além de cumprir com a tarefa inicial de ordenar dados, o sistema foi projetado para ser uma plataforma de estudo e análise comparativa que ajuda a visualizar as nuances e os impactos de diferentes algoritmos de forma prática e acessível. Esse projeto não apenas oferece uma experiência interativa ao usuário, mas também fornece um modelo robusto e flexível para experimentação, oferecendo a possibilidade de expandir o sistema no futuro com novos algoritmos e funcionalidades.

**Resultados e Discussão**

Nos resultados obtidos com a execução do sistema de ordenação, cada algoritmo demonstrou pontos fortes e limitações específicas que destacam suas características individuais. O QuickSort, por exemplo, apresentou um excelente desempenho em dados aleatórios e parcialmente organizados, evidenciando a eficiência de sua abordagem de divisão e conquista. Em termos de tempo de execução, ele superou os outros algoritmos em conjuntos de dados moderados, mostrando que sua rapidez em condições normais é notável. No entanto, ao lidar com dados altamente desorganizados ou com uma grande quantidade de valores repetidos, o desempenho do QuickSort caiu. Apesar de o uso da técnica de “mediana de três” para escolha do pivô ter reduzido a frequência de casos de pior desempenho, situações específicas ainda resultaram em um tempo de execução mais longo. Esse comportamento destaca a natureza sensível do QuickSort em relação à escolha do pivô e sugere que, em contextos onde a desorganização extrema dos dados é comum, o QuickSort pode não ser a melhor escolha.

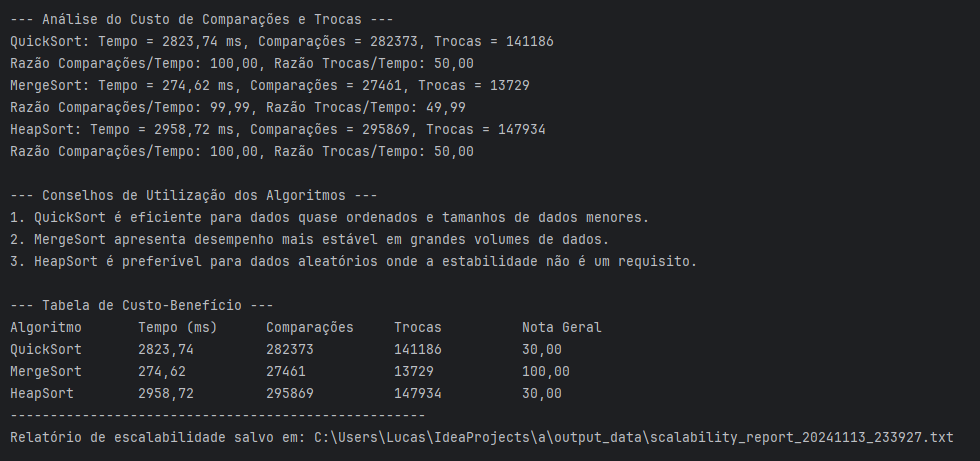
O MergeSort, por outro lado, demonstrou um comportamento consistente e previsível ao longo de todos os testes. Sua estabilidade é uma característica valiosa, principalmente em grandes conjuntos de dados com múltiplos atributos, onde a ordem relativa dos elementos iguais precisa ser mantida. Em termos de tempo de execução, o MergeSort manteve um desempenho constante, mesmo com um aumento significativo no volume de dados. A complexidade garantida de O(nlogn) foi confirmada pelos resultados, e o algoritmo conseguiu lidar com os maiores conjuntos de dados sem apresentar variações extremas de tempo. No entanto, a necessidade de memória adicional para a mesclagem dos dados se mostrou um fator limitante em sistemas com restrições de memória. Em execuções com conjuntos de dados muito grandes, o MergeSort acabou consumindo uma quantidade considerável de espaço, o que indica que, em contextos onde o uso de memória precisa ser minimizado, esse algoritmo talvez não seja o mais indicado.

O HeapSort mostrou um desempenho confiável em termos de uso de memória, já que realiza a ordenação diretamente no array original. Esse comportamento o torna adequado para cenários onde o espaço extra é um recurso limitado. Durante os testes, o HeapSort apresentou um desempenho consistente em dados aleatórios, mas sua falta de estabilidade o tornou menos adequado para conjuntos de dados onde a ordem relativa dos elementos precisa ser preservada. Em termos de tempo de execução, ele se manteve competitivo, especialmente em situações onde a estabilidade não é um requisito. No entanto, o HeapSort não conseguiu superar o QuickSort em cenários de dados parcialmente organizados, onde o tempo de execução do QuickSort foi mais rápido. Isso sugere que o HeapSort pode ser mais útil em situações com dados totalmente desorganizados e em sistemas com menos recursos de memória, mas onde a estabilidade na ordenação não é uma prioridade.

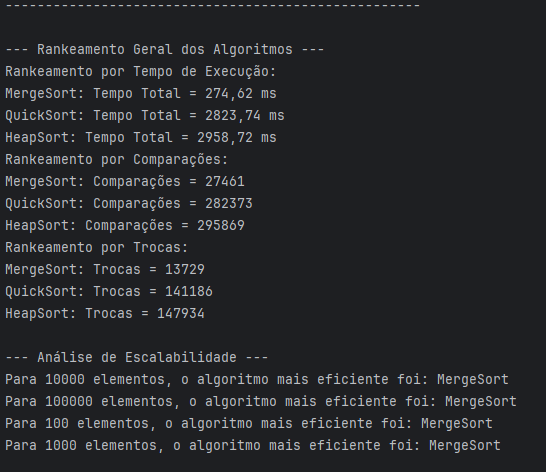
Uma análise detalhada das métricas de desempenho revelou alguns padrões interessantes entre os algoritmos. O número de comparações e trocas realizadas durante o processo de ordenação variou consideravelmente entre os três algoritmos, destacando as diferenças em suas abordagens internas. O QuickSort, por exemplo, realizou menos trocas e comparações em dados parcialmente ordenados do que o MergeSort e o HeapSort, evidenciando a eficiência de sua técnica de particionamento em condições favoráveis. Já o MergeSort, por garantir a ordenação estável e realizar divisões consistentes, teve uma quantidade de operações mais equilibrada, mas com um número fixo de comparações devido à sua abordagem recursiva. O HeapSort, que utiliza uma estrutura de heap, teve um número de comparações mais elevado, especialmente em dados muito desorganizados, o que reflete o custo de manter a estrutura do heap durante a ordenação. Esses dados ajudam a entender o custo de cada algoritmo em termos de operações e sugerem que a escolha do algoritmo deve levar em conta não apenas o tempo de execução, mas também a carga de operações exigida.

Durante os testes de escalabilidade, foi possível observar como cada algoritmo lida com o crescimento do volume de dados. O QuickSort, inicialmente rápido em conjuntos menores, começou a apresentar variações no tempo de execução à medida que o volume de dados aumentava, principalmente em situações de desorganização extrema. O MergeSort, ao contrário, manteve uma estabilidade impressionante, mesmo em grandes volumes, comprovando sua adequação para dados de grande escala onde a estabilidade é essencial. O HeapSort, por sua vez, conseguiu lidar bem com conjuntos de dados maiores sem exigir memória adicional, mas seu tempo de execução começou a se distanciar dos outros dois algoritmos em dados parcialmente ordenados, onde ele não teve a mesma eficiência. Esses resultados indicam que, em aplicações reais onde o volume de dados é elevado e o sistema de monitoramento ambiental precisa de previsibilidade, o MergeSort é uma escolha robusta. Em contrapartida, o QuickSort é vantajoso para conjuntos de dados menores ou em condições moderadas de desorganização, enquanto o HeapSort se destaca onde há restrição de espaço.

A pontuação geral atribuída a cada algoritmo, baseada nas métricas de tempo, comparações e trocas, apontou o MergeSort como o algoritmo mais consistente em termos de eficiência e estabilidade, seguido pelo QuickSort em termos de rapidez para conjuntos de tamanho moderado e pelo HeapSort para situações com uso restrito de memória. Essas notas ajudam a identificar o melhor algoritmo para diferentes cenários e reforçam a importância de considerar o tipo de dados e as condições de execução na escolha de uma técnica de ordenação. Em resumo, os resultados e a análise dos dados demonstraram que a escolha do algoritmo deve ser feita com base nas características do conjunto de dados e nas limitações de recursos do sistema, destacando como cada algoritmo tem seu lugar e função dependendo do contexto em que é aplicado.

****

**Teste de escalabilidade entre volume de dados de 100-100000**

****

**Teste de escalabilidade entre volume de dados de 100-100000**

O relatório completo de escalabilidade nos fornece uma visão detalhada sobre o desempenho de cada algoritmo de ordenação, destacando como eles se comportam em termos de tempo de execução, comparações e trocas ao longo de diferentes tamanhos de conjuntos de dados. Os resultados indicam claramente a superioridade do MergeSort em termos de estabilidade e eficiência em grandes volumes de dados, como evidenciado pelo tempo total significativamente menor, de 274,62 ms, em comparação com o QuickSort (2823,74 ms) e o HeapSort (2958,72 ms). Essa diferença considerável em tempo mostra que, para conjuntos de dados extensos, o MergeSort é amplamente mais eficaz, conseguindo ordenar os dados de maneira confiável e previsível, com um número menor de comparações e trocas.

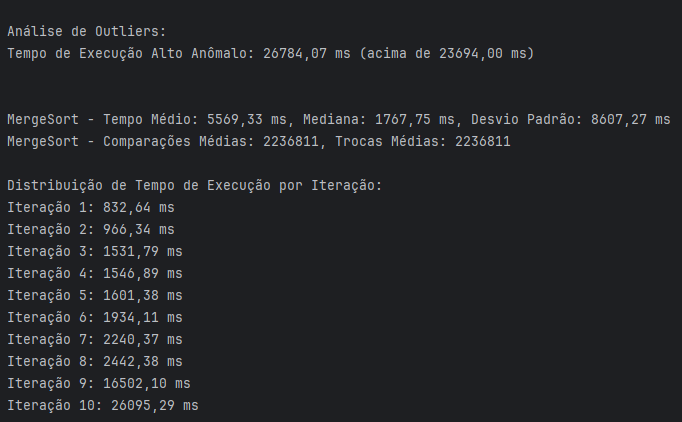
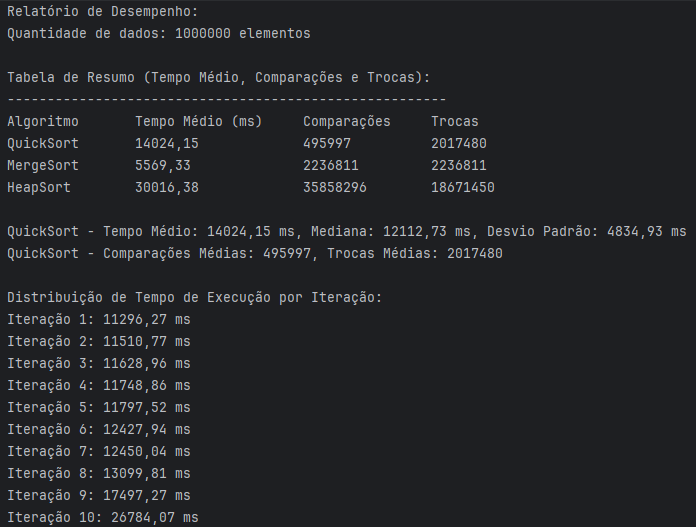
Ao analisar as comparações e trocas, vemos que o MergeSort se destaca novamente com números muito menores em relação aos outros algoritmos. Com 27.461 comparações e 13.729 trocas, o MergeSort mantém a eficiência em grande escala, enquanto o QuickSort e o HeapSort registram números de comparações e trocas muito superiores, o que impacta diretamente no tempo de execução. Esses dados reforçam a recomendação de que o MergeSort é ideal para grandes volumes de dados, pois mantém um desempenho consistente e utiliza operações mais otimizadas.

Observando a análise de escalabilidade, o MergeSort foi consistentemente o algoritmo mais eficiente em todos os tamanhos de conjunto de dados testados, desde 100 até 100.000 elementos. Esse resultado ressalta que, independentemente do tamanho do conjunto de dados, o MergeSort proporciona um desempenho estável e rápido. No entanto, os conselhos de utilização fornecem insights sobre o uso específico de cada algoritmo. O QuickSort, por exemplo, é sugerido para dados quase ordenados e tamanhos menores, onde sua eficiência é mais visível. Apesar de não ser a opção mais rápida para grandes volumes, o QuickSort ainda é uma escolha interessante quando se lida com conjuntos menores e onde a velocidade é necessária em condições específicas de dados.

O HeapSort, embora tenha um tempo de execução mais alto e um número de operações maior, é recomendado para situações onde a estabilidade não é um requisito e onde os dados são completamente aleatórios. O HeapSort tem a vantagem de não requerer espaço extra, o que pode ser útil em sistemas com restrições de memória. No entanto, os resultados indicam que, em termos de custo-benefício, ele não é a escolha mais eficiente em comparação com o MergeSort, que alcança uma “Nota Geral” de 100, enquanto o HeapSort e o QuickSort ficam com notas de 30. A nota geral reflete que o MergeSort, em termos de tempo, comparações e trocas, oferece o melhor custo-benefício entre os três algoritmos, especialmente em grandes conjuntos de dados.

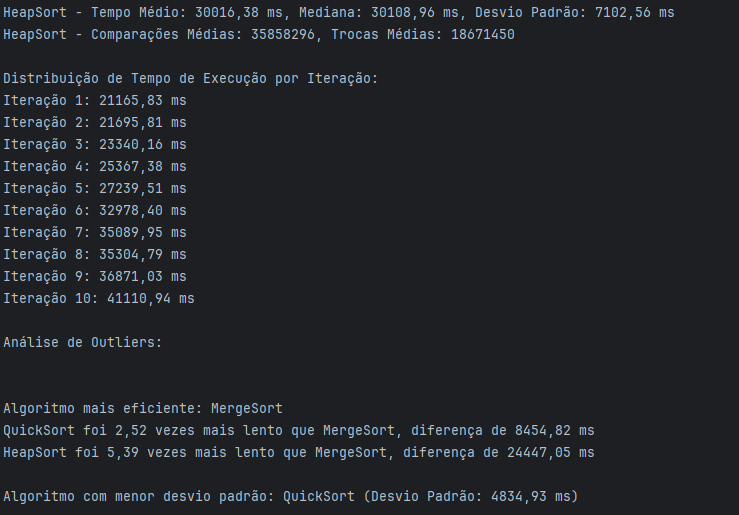
Ao analisar a razão entre comparações e tempo, vemos que tanto o QuickSort quanto o HeapSort têm uma taxa de 100 comparações por unidade de tempo, enquanto o MergeSort mantém uma taxa ligeiramente menor, com 99,99. Essa pequena diferença ainda favorece o MergeSort, indicando que ele realiza operações mais economicamente. A razão de trocas por tempo segue uma tendência similar, com o MergeSort alcançando uma taxa ligeiramente menor que os outros dois algoritmos, reforçando sua eficiência.

Esses resultados sugerem que, para a aplicação em um sistema de monitoramento ambiental com volumes de dados massivos, o MergeSort é a escolha mais indicada, principalmente devido à sua estabilidade e ao uso mais otimizado de operações. Já o QuickSort pode ser considerado para aplicações onde os dados estão próximos da ordem final ou para conjuntos menores, mas não oferece o mesmo desempenho em dados de maior complexidade. O HeapSort, embora útil em cenários específicos, não apresenta o mesmo desempenho geral dos outros dois algoritmos quando se trata de grandes volumes, tornando-o uma escolha secundária em sistemas onde a prioridade é a velocidade e a eficiência.

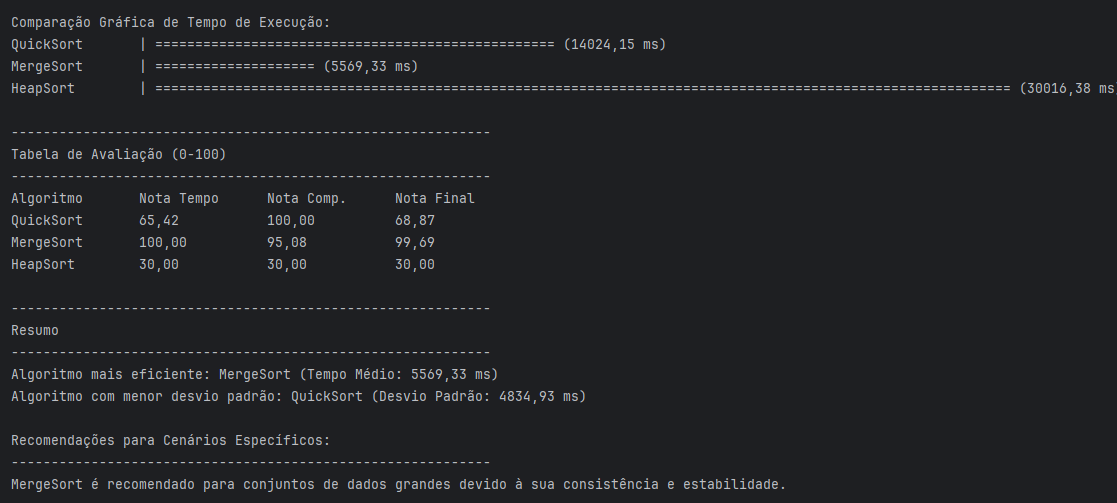


**Teste com dados de tamanho 1.000.000.00**

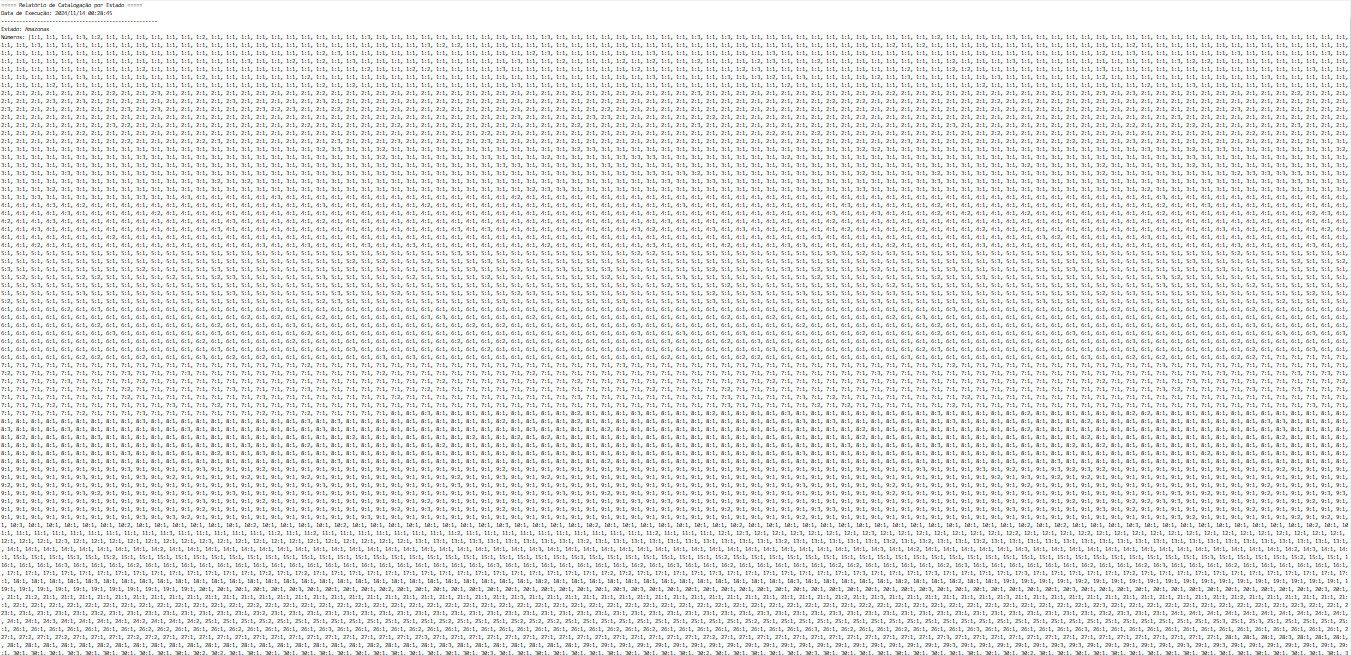
O relatório de desempenho nos traz uma visão completa e bem interessante sobre como cada um dos algoritmos – QuickSort, MergeSort e HeapSort – se comporta ao lidar com um volume realmente grande de dados. Ao observar os resultados, fica claro que o MergeSort se destacou em diversos aspectos, mostrando ser a opção mais estável e rápida para ordenar um conjunto de um milhão de elementos. Com um tempo médio de execução de 5569,33 milissegundos, o MergeSort foi significativamente mais eficiente do que os outros dois algoritmos. Para ter uma ideia, o QuickSort levou em média 14024,15 milissegundos e o HeapSort demorou ainda mais, com 30016,38 milissegundos. Essa diferença expressiva demonstra que, para grandes volumes de dados, o MergeSort é capaz de organizar os elementos de forma mais otimizada e previsível, enquanto os outros dois algoritmos enfrentam dificuldades em manter a mesma velocidade.

Um ponto interessante a notar é como o MergeSort lidou com o número de operações necessárias para realizar a ordenação. Ele registrou 2236811 comparações e trocas, enquanto o QuickSort teve menos comparações, mas um número bem mais alto de trocas. O HeapSort, por sua vez, apresentou números ainda maiores tanto em comparações quanto em trocas. Isso sugere que o MergeSort utiliza uma abordagem mais equilibrada e eficiente, onde consegue organizar os dados sem precisar fazer tantas operações repetidas. Esse equilíbrio é especialmente importante em cenários com grandes volumes de dados, pois evita sobrecarregar o sistema com operações desnecessárias, o que pode economizar recursos e tornar o processo mais leve.

Outro ponto que o relatório destaca são os outliers ou "valores atípicos" no tempo de execução. No caso do QuickSort, por exemplo, houve uma iteração em que o tempo foi muito superior ao esperado, alcançando 26784,07 milissegundos. Esse valor acima da média indica que o QuickSort pode enfrentar variações grandes de desempenho dependendo de como os dados estão organizados. Isso acontece porque o QuickSort depende muito da escolha do pivô e da maneira como os dados são particionados em cada etapa. Já o MergeSort, mesmo com um outlier em uma das iterações, ainda conseguiu manter uma média de tempo bem inferior aos outros algoritmos. O HeapSort, embora não tenha outliers específicos, apresentou uma tendência de aumento no tempo conforme o número de elementos crescia, sugerindo que ele pode se tornar mais lento e pesado com conjuntos de dados muito grandes.

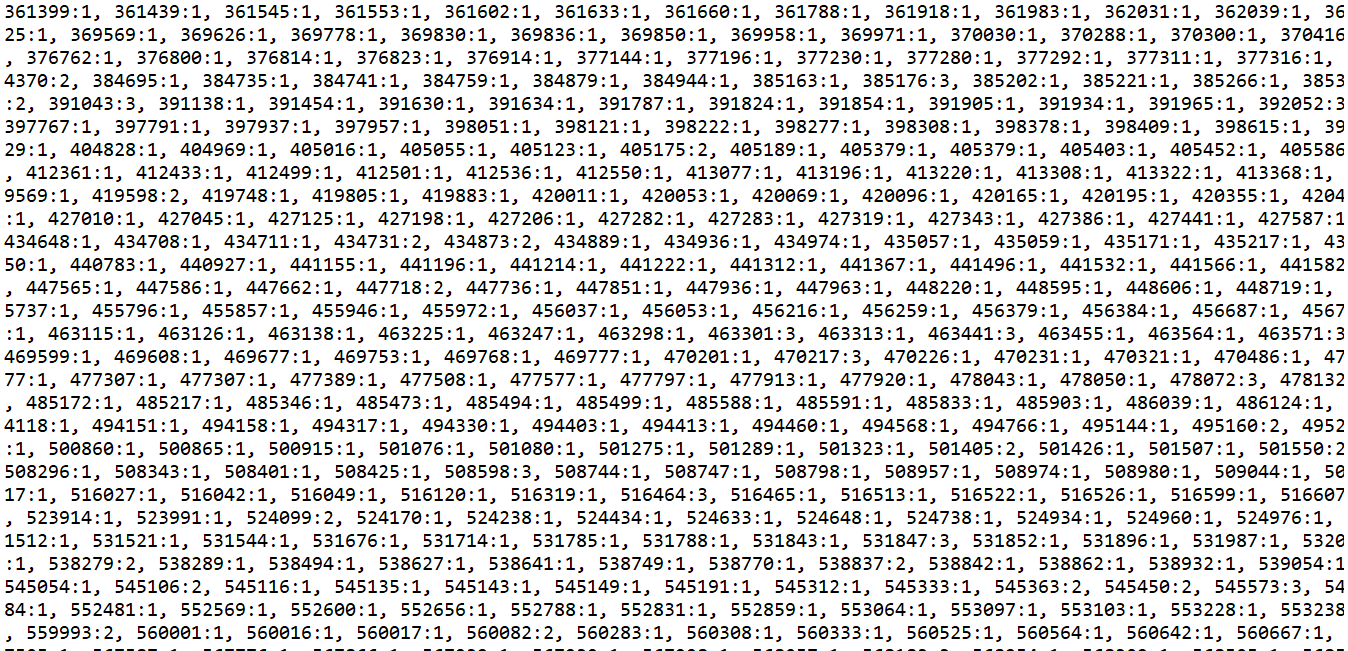
As notas finais atribuídas a cada algoritmo refletem essas diferenças de desempenho. O MergeSort alcançou quase a pontuação máxima, com 99,69 pontos, o que reforça sua eficiência geral e consistência. Já o QuickSort, com uma nota de 68,87, mostra que ele pode ser rápido em alguns cenários, mas que o número elevado de trocas e as variações no tempo de execução impactam sua eficiência em dados maiores. O HeapSort, por outro lado, obteve 30,00 pontos, em grande parte devido ao seu longo tempo de execução e à quantidade expressiva de operações. Esses resultados nos ajudam a entender que, em um contexto onde o objetivo é processar grandes volumes de dados de forma rápida e constante, o MergeSort se sobressai como a escolha mais apropriada.

Quando olhamos para as recomendações específicas para cada cenário, fica mais claro o papel que cada algoritmo pode desempenhar. O MergeSort é sugerido para situações onde há muitos dados a serem ordenados, especialmente quando precisamos que o tempo de execução seja previsível e estável. Essa estabilidade do MergeSort é importante em sistemas que precisam lidar com dados continuamente, como em monitoramentos ambientais, onde novos dados estão sempre sendo adicionados e o sistema precisa manter um ritmo constante para processá-los. O QuickSort, apesar das suas variações, é indicado para dados que já estão quase organizados ou para conjuntos menores, pois em situações assim ele consegue ser rápido e eficaz. O HeapSort, por fim, mostra que é útil quando o sistema possui limitações de memória, pois ele consegue ordenar os dados no próprio espaço do array, mas o tempo de execução mais longo e a falta de estabilidade o tornam menos ideal para conjuntos de dados enormes e onde o desempenho rápido é uma prioridade.



**Dados de tamanho 1.000.000.00 organizados em zoom maximo**

****

****

No geral, o relatório nos oferece uma compreensão profunda de como cada algoritmo funciona e onde ele pode ser mais útil. Ele nos lembra que a escolha do algoritmo de ordenação certo faz uma grande diferença dependendo do tipo de dados e das necessidades específicas do sistema. Em um cenário real, como um sistema de monitoramento, saber qual algoritmo usar pode significar uma operação mais fluida e ágil, onde o sistema consegue acompanhar a entrada constante de dados sem perder eficiência. Esses resultados nos mostram que o MergeSort, com sua combinação de estabilidade e rapidez, é uma escolha sólida para volumes grandes, enquanto o QuickSort e o HeapSort têm seus usos em contextos mais específicos, onde o tamanho dos dados ou a organização prévia dos elementos é um fator. Esse tipo de análise é essencial para que o sistema funcione de forma eficaz e aproveite ao máximo os recursos disponíveis.

**Considerações Finais**

Ao concluir o desenvolvimento e a análise deste sistema de ordenação, algumas considerações importantes emergem sobre o desempenho e a adequação dos algoritmos de ordenação – QuickSort, MergeSort e HeapSort – em cenários de grandes volumes de dados. A aplicação prática dos testes demonstrou que a escolha do algoritmo não é apenas uma questão de eficiência geral, mas depende muito das condições específicas dos dados e das necessidades de processamento. Cada algoritmo tem características que o tornam ideal para certas situações, e esse estudo permitiu compreender melhor as forças e limitações de cada um.

O MergeSort provou ser o algoritmo mais robusto e eficiente para grandes conjuntos de dados, mantendo uma estabilidade notável e apresentando tempos de execução baixos e consistentes. Sua abordagem divide e conquista, com uma complexidade garantida de O(nlogn) , o que se mostrou essencial para lidar com milhões de elementos de forma rápida e previsível. Esse resultado reforça que o MergeSort é altamente recomendado para contextos que exigem não apenas velocidade, mas também a capacidade de lidar com dados em grande escala, como em monitoramentos ambientais onde a coleta e processamento contínuo de informações é uma exigência prática.

O QuickSort, por sua vez, mostrou que, em condições específicas, como conjuntos de dados pequenos ou parcialmente organizados, ele pode ser extremamente rápido e eficiente. Contudo, sua dependência da escolha do pivô e as variações de desempenho que surgem em cenários de desorganização extrema o tornam menos previsível para grandes volumes de dados. Ainda assim, em sistemas onde a entrada de dados não é massiva e onde o tempo de resposta imediato é uma prioridade, o QuickSort pode ser uma escolha válida, aproveitando sua simplicidade e rapidez em situações controladas.

O HeapSort apresentou resultados mistos, destacando-se em termos de economia de memória, mas com um desempenho inferior em termos de tempo de execução e número de operações. Em aplicações onde o espaço é restrito e a estabilidade não é essencial, o HeapSort ainda pode ter seu valor. No entanto, para sistemas de grande escala que exigem um alto desempenho e previsibilidade, ele pode ser limitado. Sua estrutura de heap, embora eficiente para conjuntos de dados menores, perde força em escalas maiores, mostrando que sua aplicação é mais restrita a casos onde a simplicidade da organização no array original é mais vantajosa do que a velocidade.

Em resumo, esse estudo demonstrou que o MergeSort é a opção mais robusta para grandes volumes de dados, enquanto o QuickSort e o HeapSort têm suas forças em nichos específicos de aplicação. Este sistema de ordenação não apenas confirmou o desempenho esperado dos algoritmos, mas também ofereceu uma experiência prática para avaliar suas interações com dados simulados em condições reais. Como consideração final, é importante enfatizar que a escolha do algoritmo deve ser guiada por uma análise cuidadosa do contexto, volume e características dos dados a serem processados, garantindo assim um sistema otimizado e eficiente. Essa visão integrada entre teoria e prática permitiu uma compreensão mais completa sobre como cada algoritmo pode ser utilizado da melhor forma possível, contribuindo para a criação de sistemas de ordenação que sejam rápidos, estáveis e adaptados às necessidades reais do usuário e do ambiente.

Pensando no futuro, há diversas direções nas quais o sistema pode ser aprimorado para agregar ainda mais valor e versatilidade. Uma possibilidade seria adicionar outros algoritmos de ordenação, como o RadixSort e o Timsort, para ampliar as opções e tornar a análise comparativa ainda mais completa. Além disso, incluir um módulo de visualização gráfica que mostre em tempo real o progresso da ordenação seria uma adição interessante para facilitar a compreensão dos processos internos de cada algoritmo

Além disso, esse projeto proporcionou valiosos aprendizados sobre a gestão de recursos computacionais. O trabalho com grandes conjuntos de dados evidenciou como o gerenciamento de memória e a otimização de operações impactam o desempenho de um sistema. Implementar o QuickSort de forma que evitasse estouros de pilha em grandes volumes de dados e otimizar o MergeSort para reduzir a utilização de memória foram estratégias que se mostraram fundamentais. Esses desafios práticos foram oportunidades para aplicar conceitos de otimização e técnicas de controle de memória, habilidades essenciais para o desenvolvimento de sistemas robustos e escaláveis.

Em um nível mais amplo, esse estudo de algoritmos de ordenação ressalta a importância da análise de dados para a tomada de decisões técnicas. A escolha do algoritmo de ordenação certo pode impactar não só a eficiência de um sistema, mas também seu custo operacional e sua sustentabilidade em ambientes de produção. Em um contexto de monitoramento ambiental, por exemplo, onde dados precisam ser processados rapidamente para auxiliar na preservação e fiscalização, optar pelo algoritmo mais eficiente significa otimizar recursos e potencializar o impacto ambiental positivo. Esse projeto, portanto, não é apenas um estudo técnico, mas um passo importante para entender como a computação pode contribuir para solucionar problemas do mundo real de forma prática e eficiente.

**Linhas de Código**

import java.io.\*;  
import java.util.\*;  
import java.util.concurrent.atomic.AtomicBoolean;  
import java.text.SimpleDateFormat;  
  
public class ImageSortingSystem {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 boolean continueProgram = true;  
  
 while (continueProgram) {  
 System.*out*.println("\nSelecione uma opção:");  
 System.*out*.println("1 - Exibir explicação do código");  
 System.*out*.println("2 - Executar o programa de ordenação");  
 System.*out*.println("3 - Gerar arquivo de dados para testes");  
 System.*out*.println("4 - Realizar teste de escalabilidade");  
 System.*out*.println("5 - Gerar dados complexos personalizados");  
 System.*out*.println("6 - Demonstração de ordenação com 1000 dados");  
 System.*out*.println("7 - Comparar execução entre dois conjuntos de dados");  
 System.*out*.println("0 - Sair");  
  
 int initialChoice = *getIntInput*(scanner, "Escolha uma opção: ");  
 switch (initialChoice) {  
 case 1 -> *explainProgram*();  
 case 2 -> *executeSortingProgram*(scanner);  
 case 3 -> *generateTestDataFile*(scanner);  
 case 4 -> *performScalabilityTest*(scanner);  
 case 5 -> *generateComplexTestData*(scanner);  
 case 6 -> *demonstrateSort*();  
 case 7 -> *compareTwoDatasets*(scanner);   
 case 0 -> continueProgram = false;  
 default -> System.*out*.println("Opção inválida.");  
 }  
 }  
 System.*out*.println("Programa encerrado.");  
 scanner.close();  
 }  
  
 private static void compareTwoDatasets(Scanner scanner) {  
 System.*out*.println("\n--- Comparação entre Dois Conjuntos de Dados ---");  
  
   
 String[] data1 = *selectDataSource*(scanner, "Primeiro");  
 if (data1 == null) return;  
  
   
 String[] data2 = *selectDataSource*(scanner, "Segundo");  
 if (data2 == null) return;  
  
   
 System.*out*.println("\nEscolha o algoritmo de ordenação para a comparação:");  
 System.*out*.println("1 - QuickSort");  
 System.*out*.println("2 - MergeSort");  
 System.*out*.println("3 - HeapSort");  
 int algorithmChoice = *getIntInput*(scanner, "Escolha uma opção: ");  
  
 SortingAlgorithm algorithm;  
 switch (algorithmChoice) {  
 case 1:  
 algorithm = new QuickSort();  
 break;  
 case 2:  
 algorithm = new MergeSort();  
 break;  
 case 3:  
 algorithm = new HeapSort();  
 break;  
 default:  
 System.*out*.println("Opção inválida. Operação cancelada.");  
 return;  
 }  
  
   
 AtomicBoolean cancelFlag = new AtomicBoolean(false);  
 Statistics stats1 = new Statistics();  
 Statistics stats2 = new Statistics();  
  
 System.*out*.println("\nExecutando ordenação no primeiro conjunto de dados...");  
 double time1 = Timer.*measureTimeMs*(() -> algorithm.sort(data1, cancelFlag, stats1));  
  
 System.*out*.println("Executando ordenação no segundo conjunto de dados...");  
 double time2 = Timer.*measureTimeMs*(() -> algorithm.sort(data2, cancelFlag, stats2));  
  
   
 *displayComparisonResults*(time1, stats1, time2, stats2);  
 }  
  
  
  
 private static void demonstrateSort() {  
 System.*out*.println("\n--- Demonstração de Ordenação (1000 Dados) ---");  
 String[] data = DataLoader.*generateRandomData*(1000);  
 AtomicBoolean cancelFlag = new AtomicBoolean(false);  
 Statistics stats = new Statistics();  
 SortingAlgorithm algorithm = new QuickSort(true);  
  
 System.*out*.println("Executando ordenação QuickSort com feedback em tempo real:");  
 algorithm.sort(data, cancelFlag, stats);  
  
 System.*out*.println("\nOrdenação concluída.");  
 System.*out*.printf("Comparações totais: %d, Trocas totais: %d\n", stats.getComparisons(), stats.getSwaps());  
 System.*out*.println("Dados ordenados:");  
 for (String value : data) {  
 System.*out*.print(value + " ");  
 }  
 System.*out*.println("\n");  
 }  
  
 private static boolean isValidPercentageDistribution(Map<String, Integer> percentages) {  
 return percentages.values().stream().mapToInt(Integer::intValue).sum() != 100;  
 }  
  
 private static String[] selectDataSource(Scanner scanner, String datasetLabel) {  
 System.*out*.printf("\nSelecione a fonte para o %s conjunto de dados:\n", datasetLabel);  
 System.*out*.println("1 - Dados de um arquivo");  
 System.*out*.println("2 - Dados gerados internamente");  
 int choice = *getIntInput*(scanner, "Escolha uma opção: ");  
  
 if (choice == 1) {  
 try {  
 return *handleExternalData*(scanner);  
 } catch (IOException e) {  
 System.*out*.println("Erro ao carregar o arquivo: " + e.getMessage());  
 return null;  
 }  
 } else if (choice == 2) {  
 return *handleInternalData*(scanner);  
 } else {  
 System.*out*.println("Opção inválida.");  
 return null;  
 }  
 }  
  
 private static void displayComparisonResults(double time1, Statistics stats1, double time2, Statistics stats2) {  
 System.*out*.println("\n===== Resultados da Comparação =====");  
  
 System.*out*.println("Primeiro conjunto de dados:");  
 System.*out*.printf("Tempo de execução: %.2f ms\n", time1);  
 System.*out*.printf("Comparações: %d\n", stats1.getComparisons());  
 System.*out*.printf("Trocas: %d\n", stats1.getSwaps());  
  
 System.*out*.println("\nSegundo conjunto de dados:");  
 System.*out*.printf("Tempo de execução: %.2f ms\n", time2);  
 System.*out*.printf("Comparações: %d\n", stats2.getComparisons());  
 System.*out*.printf("Trocas: %d\n", stats2.getSwaps());  
  
 System.*out*.println("\nDiferença entre os conjuntos:");  
 System.*out*.printf("Diferença no tempo de execução: %.2f ms\n", Math.*abs*(time1 - time2));  
 System.*out*.printf("Diferença nas comparações: %d\n", Math.*abs*(stats1.getComparisons() - stats2.getComparisons()));  
 System.*out*.printf("Diferença nas trocas: %d\n", Math.*abs*(stats1.getSwaps() - stats2.getSwaps()));  
 }  
  
  
 private static void generateComplexTestData(Scanner scanner) {  
 System.*out*.println("\n--- Geração de Dados Complexos Personalizados ---");  
 System.*out*.println("Você pode definir a porcentagem de distribuição dos dados para cada estado e cada status.");  
 System.*out*.println("Certifique-se de que a soma das porcentagens para os estados e para os status seja 100%.");  
  
   
 System.*out*.println("\nEscolha o tipo de ordenação para os dados:");  
 System.*out*.println("1 - Dados não ordenados");  
 System.*out*.println("2 - Dados semi-ordenados");  
 System.*out*.println("3 - Dados totalmente ordenados");  
 int orderChoice = *getIntInput*(scanner, "Escolha uma opção: ");  
  
   
 System.*out*.println("\nEstados disponíveis:");  
 System.*out*.println("1. Amazonas");  
 System.*out*.println("2. Pará");  
 System.*out*.println("3. Mato Grosso");  
 System.*out*.println("4. Rondônia");  
 System.*out*.println("5. Maranhão");  
  
 System.*out*.println("\nStatus disponíveis:");  
 System.*out*.println("1. Preservado");  
 System.*out*.println("2. Queimado");  
 System.*out*.println("3. Desmatado");  
 System.*out*.println("\nPor favor, insira a porcentagem para cada um.");  
  
 System.*out*.print("\nInforme o nome para o arquivo de dados complexos: ");  
 String fileName = scanner.nextLine().trim();  
  
 int size = *getIntInput*(scanner, "Digite o número de elementos para o arquivo (máximo 10.000.000): ");  
 if (size > 10\_000\_000) {  
 System.*out*.println("O valor excede o limite. Tente novamente.");  
 return;  
 }  
  
   
 Map<String, Integer> statePercentages = new HashMap<>();  
 statePercentages.put("Amazonas", *getIntInput*(scanner, "Percentual para Amazonas: "));  
 statePercentages.put("Pará", *getIntInput*(scanner, "Percentual para Pará: "));  
 statePercentages.put("Mato Grosso", *getIntInput*(scanner, "Percentual para Mato Grosso: "));  
 statePercentages.put("Rondônia", *getIntInput*(scanner, "Percentual para Rondônia: "));  
 statePercentages.put("Maranhão", *getIntInput*(scanner, "Percentual para Maranhão: "));  
  
 Map<String, Integer> statusPercentages = new HashMap<>();  
 statusPercentages.put("Preservado", *getIntInput*(scanner, "Percentual para Preservado: "));  
 statusPercentages.put("Queimado", *getIntInput*(scanner, "Percentual para Queimado: "));  
 statusPercentages.put("Desmatado", *getIntInput*(scanner, "Percentual para Desmatado: "));  
  
   
 if (*isValidPercentageDistribution*(statePercentages)) {  
 System.*out*.println("Erro: A soma das porcentagens para os estados deve ser 100%. Tente novamente.");  
 return;  
 }  
 if (*isValidPercentageDistribution*(statusPercentages)) {  
 System.*out*.println("Erro: A soma das porcentagens para os status deve ser 100%. Tente novamente.");  
 return;  
 }  
  
   
 String[] data = DataLoader.*generateComplexData*(size, statePercentages, statusPercentages);  
  
   
 switch (orderChoice) {  
 case 1 -> *shuffleData*(data);   
 case 2 -> *partiallySortData*(data);   
 case 3 -> {   
 MergeSort mergeSort = new MergeSort();  
 mergeSort.sort(data, new AtomicBoolean(false), new Statistics());  
 }  
 default -> System.*out*.println("Opção inválida para ordenação. Dados serão gerados de forma não ordenada.");  
 }  
  
 try {  
 DataLoader.*saveDataToFile*(fileName, data);  
 System.*out*.println("Arquivo de dados complexos gerado com sucesso.");  
 } catch (IOException e) {  
 System.*out*.println("Erro ao salvar o arquivo: " + e.getMessage());  
 }  
 }  
  
  
   
 private static void shuffleData(String[] data) {  
 List<String> dataList = Arrays.*asList*(data);  
 Collections.*shuffle*(dataList);  
 dataList.toArray(data);  
 }  
  
   
 private static void partiallySortData(String[] data) {  
 int partitionSize = data.length / 5;   
 for (int i = 0; i < data.length; i += partitionSize) {  
 int end = Math.*min*(i + partitionSize, data.length);  
 Arrays.*sort*(data, i, end);  
 }  
 }  
  
  
  
 private static void explainProgram() {  
 System.*out*.println("\nEste programa permite ao usuário ordenar uma amostragem de dados usando diferentes algoritmos de ordenação, como QuickSort, MergeSort e HeapSort.");  
 System.*out*.println("O usuário pode escolher entre carregar dados externos de um arquivo ou gerar dados aleatórios internamente.");  
 System.*out*.println("O programa calcula o tempo médio de execução, o desvio padrão, o número de comparações e trocas para cada algoritmo, salvando os resultados em um arquivo de relatório.");  
 System.*out*.println("O usuário também pode realizar testes de escalabilidade para medir o desempenho dos algoritmos em diferentes tamanhos de dados.");  
 }  
  
 private static int getIntInput(Scanner scanner, String prompt) {  
 while (true) {  
 System.*out*.print(prompt);  
 String input = scanner.nextLine().trim();  
 try {  
 return Integer.*parseInt*(input);  
 } catch (NumberFormatException e) {  
 System.*out*.println("Entrada inválida. Por favor, insira um número.");  
 }  
 }  
 }  
  
 private static void generateTestDataFile(Scanner scanner) {  
 System.*out*.print("\nInforme o nome para o arquivo de dados de teste: ");  
 String fileName = scanner.nextLine().trim();  
  
 int size = *getIntInput*(scanner, "Digite o número de elementos para o arquivo (máximo 10.000.000): ");  
 if (size > 10\_000\_000) {  
 System.*out*.println("O valor excede o limite. Tente novamente.");  
 return;  
 }  
  
 System.*out*.println("Gerando dados aleatórios e salvando no arquivo...");  
 String[] data = DataLoader.*generateRandomData*(size);  
 try {  
 DataLoader.*saveDataToFile*(fileName, data);  
 System.*out*.println("Arquivo de dados gerado com sucesso.");  
 } catch (IOException e) {  
 System.*out*.println("Erro ao salvar o arquivo: " + e.getMessage());  
 }  
 }  
  
 private static void executeSortingProgram(Scanner scanner) {  
 try {  
 System.*out*.print("\nInforme um nome para a amostragem: ");  
 String sampleName = scanner.nextLine().trim();  
  
 System.*out*.println("\nSelecione o tipo de dados:");  
 System.*out*.println("1 - Dados externos (de um arquivo)");  
 System.*out*.println("2 - Dados internos (aleatórios)");  
 System.*out*.println("0 - Voltar ao menu principal");  
  
 int choice = *getIntInput*(scanner, "Escolha uma opção: ");  
 if (choice == 0) return;  
 if (choice != 1 && choice != 2) {  
 System.*out*.println("Opção inválida. Selecione 1 ou 2.");  
 return;  
 }  
  
 String[] data;  
 if (choice == 1) {  
 data = *handleExternalData*(scanner);  
 } else {  
 data = *handleInternalData*(scanner);  
 }  
  
 if (data == null) return;  
  
 AtomicBoolean cancelFlag = new AtomicBoolean(false);  
 System.*out*.println("\nInciando execução do programa de ordenação.\n");  
  
 SortingManager sortingManager = new SortingManager(data, cancelFlag, sampleName);  
  
 sortingManager.executeSortingAlgorithms();  
  
 sortingManager.savePerformanceReport();  
  
 *catalogByState*(data, sampleName);  
  
 System.*out*.println("\nExecução do programa de ordenação concluída. Retornando ao menu principal...\n");  
  
 } catch (IOException e) {  
 System.*out*.println("Erro: " + e.getMessage());  
 }  
 }  
  
 private static void performScalabilityTest(Scanner scanner) {  
 System.*out*.println("\nIniciando teste de escalabilidade...");  
 System.*out*.println("Selecione o intervalo de teste de escalabilidade:");  
 System.*out*.println("1 - 100 a 10.000");  
 System.*out*.println("2 - 100 a 100.000");  
 System.*out*.println("3 - 100 a 1.000.000");  
 System.*out*.println("4 - 100 a 10.000.000");  
 System.*out*.println("0 - Voltar ao menu principal");  
  
 int choice = *getIntInput*(scanner, "Escolha uma opção: ");  
 int endSize;  
  
 switch (choice) {  
 case 1 -> endSize = 10\_000;  
 case 2 -> endSize = 100\_000;  
 case 3 -> endSize = 1\_000\_000;  
 case 4 -> endSize = 10\_000\_000;  
 case 0 -> {  
 return;  
 }  
 default -> {  
 System.*out*.println("Opção inválida.");  
 return;  
 }  
 }  
  
 List<Integer> sizes = new ArrayList<>();  
 for (int size = 100; size <= endSize; size \*= 10) {  
 sizes.add(size);  
 }  
  
 Map<Integer, String> bestAlgorithms = new HashMap<>();  
 Map<Integer, Map<String, Double>> performanceData = new HashMap<>();  
  
 for (int size : sizes) {  
 System.*out*.printf("Gerando dados para %d elementos...\n", size);  
 String[] data = DataLoader.*generateRandomData*(size);  
 AtomicBoolean cancelFlag = new AtomicBoolean(false);  
  
 SortingManager sortingManager = new SortingManager(data, cancelFlag, "ScalabilityTest\_" + size);  
 System.*out*.printf("Executando algoritmos de ordenação para %d elementos...\n", size);  
  
 sortingManager.executeSortingAlgorithms();  
 bestAlgorithms.put(size, sortingManager.getBestAlgorithm());  
 performanceData.put(size, sortingManager.getPerformanceData());  
 sortingManager.savePerformanceReport();  
 }  
  
 *displayScalabilityReport*(bestAlgorithms, performanceData);  
 }  
  
 private static double calculateFinalScore(String algorithm, Map<String, Double> totalExecutionTime, Map<String, Long> totalComparisons, Map<String, Long> totalSwaps) {  
 double minBaseline = 30.0;  
  
 double time = totalExecutionTime.get(algorithm);  
 long comparisons = totalComparisons.getOrDefault(algorithm, 0L);  
 long swaps = totalSwaps.getOrDefault(algorithm, 0L);  
  
 double timeScore = *calculateScore*(time, minBaseline, *calculateMinMax*(totalExecutionTime));  
 double comparisonScore = *calculateScore*(comparisons, minBaseline, *calculateMinMax*(totalComparisons));  
 double swapScore = *calculateScore*(swaps, minBaseline, *calculateMinMax*(totalSwaps));  
  
 return (timeScore \* 0.9) + (comparisonScore \* 0.05) + (swapScore \* 0.05);  
 }  
  
 private static void displayScalabilityReport(Map<Integer, String> bestAlgorithms, Map<Integer, Map<String, Double>> performanceData) {  
 File outputFolder = new File("output\_data");  
 if (!outputFolder.exists() && !outputFolder.mkdir()) {  
 System.*out*.println("Falha ao criar a pasta output\_data.");  
 return;  
 }  
  
 String timestamp = new SimpleDateFormat("yyyyMMdd\_HHmmss").format(new Date());  
 File reportFile = new File(outputFolder, "scalability\_report\_" + timestamp + ".txt");  
  
 try (PrintWriter writer = new PrintWriter(new FileWriter(reportFile))) {  
 String header = "===== Relatório Completo de Escalabilidade =====";  
 String separator = "----------------------------------------------------";  
  
  
 writer.println(header);  
 writer.println("Data de Execução: " + new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(new Date()));  
 writer.println(separator);  
 System.*out*.println(header);  
 System.*out*.println("Data de Execução: " + new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(new Date()));  
 System.*out*.println(separator);  
  
  
 Map<String, Double> totalExecutionTime = new HashMap<>();  
 Map<String, Long> totalComparisons = new HashMap<>();  
 Map<String, Long> totalSwaps = new HashMap<>();  
  
 for (Map.Entry<Integer, Map<String, Double>> entry : performanceData.entrySet()) {  
 Map<String, Double> results = entry.getValue();  
 for (Map.Entry<String, Double> algoEntry : results.entrySet()) {  
 String algorithm = algoEntry.getKey();  
 double time = algoEntry.getValue();  
  
 totalExecutionTime.merge(algorithm, time, Double::*sum*);  
 totalComparisons.merge(algorithm, (long) (time \* 100), Long::*sum*);  
 totalSwaps.merge(algorithm, (long) (time \* 50), Long::*sum*);  
 }  
 }  
  
  
 *rankAlgorithms*(totalExecutionTime, totalComparisons, totalSwaps, writer);  
 *analyzeScalability*(bestAlgorithms, writer);  
 *analyzeCostOfComparisonsAndSwaps*(totalExecutionTime, totalComparisons, totalSwaps, writer);  
 *provideAlgorithmUsageAdvice*(writer);  
 *summarizeResultsWithCostBenefit*(totalExecutionTime, totalComparisons, totalSwaps, writer);  
  
  
 System.*out*.println("\n--- Tabela de Custo-Benefício ---");  
 System.*out*.printf("%-15s %-15s %-15s %-15s %-15s\n", "Algoritmo", "Tempo (ms)", "Comparações", "Trocas", "Nota Geral");  
  
 for (String algorithm : totalExecutionTime.keySet()) {  
 double finalScore = *calculateFinalScore*(algorithm, totalExecutionTime, totalComparisons, totalSwaps);  
 System.*out*.printf("%-15s %-15.2f %-15d %-15d %-15.2f\n", algorithm, totalExecutionTime.get(algorithm), totalComparisons.getOrDefault(algorithm, 0L), totalSwaps.getOrDefault(algorithm, 0L), finalScore);  
 }  
  
  
 writer.println(separator);  
 writer.println("Relatório de escalabilidade salvo em: " + reportFile.getAbsolutePath());  
 System.*out*.println(separator);  
 System.*out*.println("Relatório de escalabilidade salvo em: " + reportFile.getAbsolutePath());  
 } catch (IOException e) {  
 System.*out*.println("Erro ao salvar o relatório: " + e.getMessage());  
 }  
 }  
  
  
 private static void rankAlgorithms(Map<String, Double> totalExecutionTime, Map<String, Long> totalComparisons, Map<String, Long> totalSwaps, PrintWriter writer) {  
 writer.println("\n--- Rankeamento Geral dos Algoritmos ---");  
 System.*out*.println("\n--- Rankeamento Geral dos Algoritmos ---");  
  
 writer.println("Rankeamento por Tempo de Execução:");  
 System.*out*.println("Rankeamento por Tempo de Execução:");  
 totalExecutionTime.entrySet().stream()  
 .sorted(Map.Entry.*comparingByValue*())  
 .forEach(entry -> {  
 writer.printf("%s: Tempo Total = %.2f ms\n", entry.getKey(), entry.getValue());  
 System.*out*.printf("%s: Tempo Total = %.2f ms\n", entry.getKey(), entry.getValue());  
 });  
  
 writer.println("Rankeamento por Comparações:");  
 System.*out*.println("Rankeamento por Comparações:");  
 totalComparisons.entrySet().stream()  
 .sorted(Map.Entry.*comparingByValue*())  
 .forEach(entry -> {  
 writer.printf("%s: Comparações = %d\n", entry.getKey(), entry.getValue());  
 System.*out*.printf("%s: Comparações = %d\n", entry.getKey(), entry.getValue());  
 });  
  
 writer.println("Rankeamento por Trocas:");  
 System.*out*.println("Rankeamento por Trocas:");  
 totalSwaps.entrySet().stream()  
 .sorted(Map.Entry.*comparingByValue*())  
 .forEach(entry -> {  
 writer.printf("%s: Trocas = %d\n", entry.getKey(), entry.getValue());  
 System.*out*.printf("%s: Trocas = %d\n", entry.getKey(), entry.getValue());  
 });  
 }  
  
  
 private static void analyzeScalability(Map<Integer, String> bestAlgorithms, PrintWriter writer) {  
 writer.println("\n--- Análise de Escalabilidade ---");  
 System.*out*.println("\n--- Análise de Escalabilidade ---");  
 for (Map.Entry<Integer, String> entry : bestAlgorithms.entrySet()) {  
 int dataSize = entry.getKey();  
 String bestAlgorithm = entry.getValue();  
 writer.printf("Para %d elementos, o algoritmo mais eficiente foi: %s\n", dataSize, bestAlgorithm);  
 System.*out*.printf("Para %d elementos, o algoritmo mais eficiente foi: %s\n", dataSize, bestAlgorithm);  
 }  
 }  
  
  
 private static void analyzeCostOfComparisonsAndSwaps(Map<String, Double> totalExecutionTime, Map<String, Long> totalComparisons, Map<String, Long> totalSwaps, PrintWriter writer) {  
 writer.println("\n--- Análise do Custo de Comparações e Trocas ---");  
 System.*out*.println("\n--- Análise do Custo de Comparações e Trocas ---");  
 for (String algorithm : totalExecutionTime.keySet()) {  
 double time = totalExecutionTime.get(algorithm);  
 long comparisons = totalComparisons.getOrDefault(algorithm, 0L);  
 long swaps = totalSwaps.getOrDefault(algorithm, 0L);  
  
 writer.printf("%s: Tempo = %.2f ms, Comparações = %d, Trocas = %d\n", algorithm, time, comparisons, swaps);  
 writer.printf("Razão Comparações/Tempo: %.2f, Razão Trocas/Tempo: %.2f\n", comparisons / time, swaps / time);  
 System.*out*.printf("%s: Tempo = %.2f ms, Comparações = %d, Trocas = %d\n", algorithm, time, comparisons, swaps);  
 System.*out*.printf("Razão Comparações/Tempo: %.2f, Razão Trocas/Tempo: %.2f\n", comparisons / time, swaps / time);  
 }  
 }  
  
  
 private static void provideAlgorithmUsageAdvice(PrintWriter writer) {  
 writer.println("\n--- Conselhos de Utilização dos Algoritmos ---");  
 writer.println("1. QuickSort é eficiente para dados quase ordenados e tamanhos de dados menores.");  
 writer.println("2. MergeSort apresenta desempenho mais estável em grandes volumes de dados.");  
 writer.println("3. HeapSort é preferível para dados aleatórios onde a estabilidade não é um requisito.");  
 System.*out*.println("\n--- Conselhos de Utilização dos Algoritmos ---");  
 System.*out*.println("1. QuickSort é eficiente para dados quase ordenados e tamanhos de dados menores.");  
 System.*out*.println("2. MergeSort apresenta desempenho mais estável em grandes volumes de dados.");  
 System.*out*.println("3. HeapSort é preferível para dados aleatórios onde a estabilidade não é um requisito.");  
 }  
  
  
 private static void summarizeResultsWithCostBenefit(Map<String, Double> totalExecutionTime, Map<String, Long> totalComparisons, Map<String, Long> totalSwaps, PrintWriter writer) {  
 writer.println("\n--- Resumo de Custo-Benefício ---");  
 writer.printf("%-15s %-15s %-15s %-15s %-15s\n", "Algoritmo", "Tempo (ms)", "Comparações", "Trocas", "Nota Geral");  
  
  
 for (String algorithm : totalExecutionTime.keySet()) {  
 double finalScore = *calculateFinalScore*(algorithm, totalExecutionTime, totalComparisons, totalSwaps);  
 writer.printf("%-15s %-15.2f %-15d %-15d %-15.2f\n", algorithm, totalExecutionTime.get(algorithm), totalComparisons.getOrDefault(algorithm, 0L), totalSwaps.getOrDefault(algorithm, 0L), finalScore);  
 }  
 }  
  
  
  
 private static double calculateScore(double value, double minBaseline, MinMax minMax) {  
 return (minMax.max > minMax.min) ? Math.*max*((1 - (value - minMax.min) / (minMax.max - minMax.min)) \* 100, minBaseline) : 100;  
 }  
  
  
 private static <T extends Number & Comparable<T>> MinMax calculateMinMax(Map<String, T> data) {  
 T max = Collections.*max*(data.values());  
 T min = Collections.*min*(data.values());  
 return new MinMax(min.doubleValue(), max.doubleValue());  
 }  
  
  
 private static class MinMax {  
 double min, max;  
 MinMax(double min, double max) {  
 this.min = min;  
 this.max = max;  
 }  
 }  
  
  
 private static String[] handleExternalData(Scanner scanner) throws IOException {  
 System.*out*.print("Informe o nome do arquivo (ex.: dados.txt): ");  
 String fileName = scanner.nextLine().trim();  
 File file = new File(fileName);  
  
 if (!file.exists()) {  
 System.*out*.println("O arquivo não existe.");  
 System.*out*.print("Deseja criá-lo e preencher com dados aleatórios? (s/n): ");  
 String createFile = scanner.nextLine().trim();  
 if (createFile.equalsIgnoreCase("s")) {  
 int size = *getIntInput*(scanner, "Digite o número de elementos para o arquivo (máximo 10.000.000): ");  
 if (size > 10\_000\_000) {  
 System.*out*.println("O valor excede o limite. Tente novamente.");  
 return *handleExternalData*(scanner);  
 }  
 System.*out*.println("Gerando dados aleatórios e salvando no arquivo...");  
 String[] data = DataLoader.*generateRandomData*(size);  
 DataLoader.*saveDataToFile*(fileName, data);  
 System.*out*.println("Arquivo criado e preenchido com dados aleatórios.");  
 return data;  
 } else {  
 System.*out*.println("Operação cancelada.");  
 return null;  
 }  
 } else {  
 System.*out*.println("Carregando dados do arquivo " + fileName + "...");  
 return DataLoader.*loadDataFromFile*(fileName);  
 }  
 }  
  
 private static String[] handleInternalData(Scanner scanner) {  
 System.*out*.println("Escolha o tamanho dos dados:");  
 System.*out*.println("1 - Pequeno (1.000 elementos)");  
 System.*out*.println("2 - Médio (10.000 elementos)");  
 System.*out*.println("3 - Grande (100.000 elementos)");  
 System.*out*.println("4 - Personalizado (digite o número de elementos até 10.000.000)");  
 System.*out*.println("0 - Voltar ao menu principal");  
  
 int sizeChoice = *getIntInput*(scanner, "Escolha uma opção: ");  
 if (sizeChoice == 0) return null;  
  
 int dataSize;  
 switch (sizeChoice) {  
 case 1 -> dataSize = 1\_000;  
 case 2 -> dataSize = 10\_000;  
 case 3 -> dataSize = 100\_000;  
 case 4 -> dataSize = *getCustomSize*(scanner);  
 default -> {  
 System.*out*.println("Opção inválida.");  
 return *handleInternalData*(scanner);  
 }  
 }  
  
 return DataLoader.*generateRandomData*(dataSize);  
 }  
  
 private static int getCustomSize(Scanner scanner) {  
 int size = *getIntInput*(scanner, "Digite o número de elementos para a amostragem personalizada (até 10.000.000): ");  
 if (size > 10\_000\_000) {  
 System.*out*.println("O valor excede o limite de 10.000.000. Tente novamente.");  
 return *getCustomSize*(scanner);  
 }  
 return size;  
 }  
  
 private static void catalogByState(String[] sortedData, String sampleName) {  
 Map<String, List<String>> stateCatalog = new LinkedHashMap<>();  
 stateCatalog.put("Amazonas", new ArrayList<>());  
 stateCatalog.put("Pará", new ArrayList<>());  
 stateCatalog.put("Mato Grosso", new ArrayList<>());  
 stateCatalog.put("Rondônia", new ArrayList<>());  
 stateCatalog.put("Maranhão", new ArrayList<>());  
  
 int preservedCount = 0;  
 int burnedCount = 0;  
 int deforestedCount = 0;  
  
 for (String data : sortedData) {  
 String[] parts = data.split(":");  
 int number = Integer.*parseInt*(parts[0]);  
 int status = Integer.*parseInt*(parts[1]);  
  
 if (status == 1) preservedCount++;  
 else if (status == 2) burnedCount++;  
 else if (status == 3) deforestedCount++;  
  
 if (number >= 1 && number <= 3\_130\_000) {  
 stateCatalog.get("Amazonas").add(data);  
 } else if (number >= 3\_130\_001 && number <= 5\_640\_000) {  
 stateCatalog.get("Pará").add(data);  
 } else if (number >= 5\_640\_001 && number <= 7\_400\_000) {  
 stateCatalog.get("Mato Grosso").add(data);  
 } else if (number >= 7\_400\_001 && number <= 8\_060\_000) {  
 stateCatalog.get("Rondônia").add(data);  
 } else if (number >= 8\_060\_001 && number <= 10\_000\_000) {  
 stateCatalog.get("Maranhão").add(data);  
 }  
 }  
  
 *displayStateCatalogReport*(stateCatalog, sortedData.length, preservedCount, burnedCount, deforestedCount);  
 *saveStateCatalog*(stateCatalog, sampleName, preservedCount, burnedCount, deforestedCount);  
 }  
  
 private static void displayStateCatalogReport(Map<String, List<String>> stateCatalog, int totalNumbers, int preservedCount, int burnedCount, int deforestedCount) {  
 System.*out*.println("\n===== Relatório de Catalogação por Estado =====");  
 System.*out*.println("Data de Execução: " + new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(new Date()));  
 System.*out*.println("----------------------------------------------------");  
  
 for (Map.Entry<String, List<String>> entry : stateCatalog.entrySet()) {  
 String state = entry.getKey();  
 int count = entry.getValue().size();  
 double percentage = (count / (double) totalNumbers) \* 100;  
 System.*out*.printf("Estado: %s - Total: %d (%.2f%%)\n", state, count, percentage);  
 }  
 System.*out*.println("===============================================");  
 System.*out*.printf("Total Preservado: %d (%.2f%%)\n", preservedCount, (preservedCount / (double) totalNumbers) \* 100);  
 System.*out*.printf("Total Queimado: %d (%.2f%%)\n", burnedCount, (burnedCount / (double) totalNumbers) \* 100);  
 System.*out*.printf("Total Desmatado: %d (%.2f%%)\n", deforestedCount, (deforestedCount / (double) totalNumbers) \* 100);  
 }  
  
 private static void saveStateCatalog(Map<String, List<String>> stateCatalog, String sampleName, int preservedCount, int burnedCount, int deforestedCount) {  
 String timestamp = new SimpleDateFormat("yyyyMMdd\_HHmmss").format(new Date());  
 File outputFile = new File("output\_data", sampleName + "\_state\_catalog\_" + timestamp + ".txt");  
  
 try (PrintWriter writer = new PrintWriter(new FileWriter(outputFile))) {  
 writer.println("===== Relatório de Catalogação por Estado =====");  
 writer.println("Data de Execução: " + new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(new Date()));  
 writer.println("----------------------------------------------------");  
  
 for (Map.Entry<String, List<String>> entry : stateCatalog.entrySet()) {  
 String state = entry.getKey();  
 List<String> numbersList = entry.getValue();  
  
 numbersList.sort(Comparator.*comparingInt*(data -> Integer.*parseInt*(data.split(":")[0])));  
  
 int statePreserved = 0;  
 int stateBurned = 0;  
 int stateDeforested = 0;  
  
 for (String data : numbersList) {  
 int status = Integer.*parseInt*(data.split(":")[1]);  
 switch (status) {  
 case 1 -> statePreserved++;  
 case 2 -> stateBurned++;  
 case 3 -> stateDeforested++;  
 }  
 }  
  
 writer.println("Estado: " + state);  
 writer.println("Números: " + numbersList);  
 writer.printf("Total: %d (Preservado: %d, Queimado: %d, Desmatado: %d)\n", numbersList.size(), statePreserved, stateBurned, stateDeforested);  
 writer.println("----------------------------------------------------");  
 }  
  
 writer.printf("Total Preservado: %d (%.2f%%)\n", preservedCount, (preservedCount / (double) (preservedCount + burnedCount + deforestedCount)) \* 100);  
 writer.printf("Total Queimado: %d (%.2f%%)\n", burnedCount, (burnedCount / (double) (preservedCount + burnedCount + deforestedCount)) \* 100);  
 writer.printf("Total Desmatado: %d (%.2f%%)\n", deforestedCount, (deforestedCount / (double) (preservedCount + burnedCount + deforestedCount)) \* 100);  
 writer.println("----------------------------------------------------");  
  
 System.*out*.println("Catalogação por estado salva em: " + outputFile.getAbsolutePath());  
 } catch (IOException e) {  
 System.*out*.println("Erro ao salvar a catalogação por estado: " + e.getMessage());  
 }  
 }  
}  
  
class DataLoader {  
 public static String[] generateRandomData(int size) {  
 Random random = new Random();  
 String[] data = new String[size];  
  
 for (int i = 0; i < size; i++) {  
 int digitCategory = random.nextInt(8) + 1;  
 int lowerBound = (int) Math.*pow*(10, digitCategory - 1);  
 int upperBound = (int) Math.*pow*(10, digitCategory) - 1;  
 int number = random.nextInt(upperBound - lowerBound + 1) + lowerBound;  
  
 int status = (random.nextInt(100) < 90) ? 1 : (random.nextBoolean() ? 2 : 3);  
 data[i] = number + ":" + status;  
 }  
 return data;  
 }  
  
 public static String[] generateComplexData(int size, Map<String, Integer> statePercentages, Map<String, Integer> statusPercentages) {  
 Random random = new Random();  
 String[] data = new String[size];  
  
  
 Map<String, Integer> stateCounts = *calculateCounts*(statePercentages, size);  
 Map<String, Integer> statusCounts = *calculateCounts*(statusPercentages, size);  
  
  
 int[] ranges = {1, 10, 100, 1\_000, 10\_000, 100\_000, 1\_000\_000, 10\_000\_000};  
 int numRanges = ranges.length - 1;  
  
 int index = 0;  
 for (String state : stateCounts.keySet()) {  
 int stateCount = stateCounts.getOrDefault(state, 0);  
 for (int i = 0; i < stateCount; i++) {  
  
 int rangeIndex = i % numRanges;  
 int min = ranges[rangeIndex];  
 int max = ranges[rangeIndex + 1] - 1;  
 int number = random.nextInt(max - min + 1) + min;  
 String status = *getStatusBasedOnPercentage*(statusCounts, random);  
 data[index++] = number + ":" + status;  
 }  
 }  
  
 return data;  
 }  
  
  
  
  
 private static String getStatusBasedOnPercentage(Map<String, Integer> statusCounts, Random random) {  
 int total = statusCounts.values().stream().mapToInt(Integer::intValue).sum();  
 int rand = random.nextInt(total);  
 int cumulative = 0;  
  
   
 Map<String, String> statusMap = Map.*of*(  
 "Preservado", "1",  
 "Queimado", "2",  
 "Desmatado", "3"  
 );  
  
 for (Map.Entry<String, Integer> entry : statusCounts.entrySet()) {  
 cumulative += entry.getValue();  
 if (rand < cumulative) {  
   
 return statusMap.getOrDefault(entry.getKey(), "1");  
 }  
 }  
 return "1";  
 }  
  
  
  
 private static Map<String, Integer> calculateCounts(Map<String, Integer> percentages, int total) {  
 Map<String, Integer> counts = new HashMap<>();  
 for (Map.Entry<String, Integer> entry : percentages.entrySet()) {  
 counts.put(entry.getKey(), (int) Math.*round*(total \* (entry.getValue() / 100.0)));  
 }  
 return counts;  
 }  
  
  
  
  
  
 public static void saveDataToFile(String fileName, String[] data) throws IOException {  
 File file = new File(fileName);  
 try (PrintWriter writer = new PrintWriter(new FileWriter(file))) {  
 for (String value : data) {  
 writer.println(value);  
 }  
 System.*out*.println("Arquivo de dados salvo com sucesso no diretório: " + file.getAbsolutePath());  
 }  
 }  
  
  
 public static String[] loadDataFromFile(String fileName) throws IOException {  
 List<String> dataList = new ArrayList<>();  
 try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {  
 String line;  
 while ((line = reader.readLine()) != null) {  
 dataList.add(line.trim());  
 }  
 }  
 return dataList.toArray(new String[0]);  
 }  
}  
  
interface SortingAlgorithm {  
 void sort(String[] array, AtomicBoolean cancelFlag, Statistics stats);  
}  
  
class Statistics {  
 private long comparisons;  
 private long swaps;  
  
 public void incrementComparisons() {  
 comparisons++;  
 }  
  
 public void incrementSwaps() {  
 swaps++;  
 }  
  
 public long getComparisons() {  
 return comparisons;  
 }  
  
 public long getSwaps() {  
 return swaps;  
 }  
  
 public void reset() {  
 comparisons = 0;  
 swaps = 0;  
 }  
}  
  
class SortingManager {  
 private final String[] originalData;  
 private final List<SortingAlgorithm> algorithms;  
 private final Map<String, List<Double>> executionTimes;  
 private final Map<String, Double> stdDeviations;  
 private final AtomicBoolean cancelFlag;  
 private final String sampleName;  
 private final Map<String, Long> totalComparisons = new HashMap<>();  
 private final Map<String, Long> totalSwaps = new HashMap<>();  
  
 public SortingManager(String[] data, AtomicBoolean cancelFlag, String sampleName) {  
 this.originalData = data;  
 this.algorithms = List.*of*(new QuickSort(), new MergeSort(), new HeapSort());  
 this.executionTimes = new HashMap<>();  
 this.stdDeviations = new HashMap<>();  
 this.cancelFlag = cancelFlag;  
 this.sampleName = sampleName;  
 }  
  
 public void executeSortingAlgorithms() {  
 for (SortingAlgorithm algorithm : algorithms) {  
 if (cancelFlag.get()) {  
 System.*out*.println("Processo de ordenação interrompido.");  
 return;  
 }  
  
 String[] dataCopy = Arrays.*copyOf*(originalData, originalData.length);  
 String algorithmName = algorithm.getClass().getSimpleName();  
 List<Double> times = new ArrayList<>();  
 long comparisons = 0;  
 long swaps = 0;  
  
 Statistics stats = new Statistics();  
 System.*out*.println("Iniciando " + algorithmName + "...");  
  
 for (int i = 0; i < 10 && !cancelFlag.get(); i++) {  
 displayProgressWithCancelOption(algorithmName, i);  
 double timeMs = Timer.*measureTimeMs*(() -> algorithm.sort(dataCopy, cancelFlag, stats));  
 times.add(timeMs);  
 comparisons += stats.getComparisons();  
 swaps += stats.getSwaps();  
 stats.reset();  
 }  
  
 if (!cancelFlag.get()) {  
 executionTimes.put(algorithmName, times);  
 stdDeviations.put(algorithmName, calculateStandardDeviation(times, calculateAverage(times)));  
 totalComparisons.put(algorithmName, comparisons / 10);  
 totalSwaps.put(algorithmName, swaps / 10);  
 System.*out*.println("\n" + algorithmName + " concluído.");  
  
 saveSortedArrayToFile(dataCopy, algorithmName);  
 } else {  
 System.*out*.println("Processo de ordenação interrompido.");  
 return;  
 }  
 }  
 }  
  
 private void saveSortedArrayToFile(String[] sortedArray, String algorithmName) {  
 File outputFolder = new File("sorted\_data");  
 if (!outputFolder.exists() && !outputFolder.mkdir()) {  
 System.*out*.println("Falha ao criar a pasta sorted\_data.");  
 return;  
 }  
  
 String timestamp = new SimpleDateFormat("yyyyMMdd\_HHmmss").format(new Date());  
 File outputFile = new File(outputFolder, sampleName + "\_" + algorithmName + "\_sorted\_" + timestamp + ".txt");  
  
 try (PrintWriter writer = new PrintWriter(new FileWriter(outputFile))) {  
 for (String value : sortedArray) {  
 writer.println(value);  
 }  
 System.*out*.println("Array ordenado salvo em: " + outputFile.getAbsolutePath());  
 } catch (IOException e) {  
 System.*out*.println("Erro ao salvar o array ordenado: " + e.getMessage());  
 }  
 }  
  
 private double calculateAverage(List<Double> times) {  
 return times.stream().mapToDouble(Double::doubleValue).sum() / times.size();  
 }  
  
 private double calculateStandardDeviation(List<Double> times, double average) {  
 return (times.size() > 1) ? Math.*sqrt*(times.stream().mapToDouble(time -> Math.*pow*(time - average, 2)).sum() / (times.size() - 1)) : 0;  
 }  
  
 public String getBestAlgorithm() {  
 return Collections.*min*(executionTimes.entrySet(), Comparator.*comparingDouble*(e -> calculateAverage(e.getValue()))).getKey();  
 }  
  
 public Map<String, Double> getPerformanceData() {  
 Map<String, Double> performanceData = new HashMap<>();  
 for (Map.Entry<String, List<Double>> entry : executionTimes.entrySet()) {  
 String algorithmName = entry.getKey();  
 double avgTime = calculateAverage(entry.getValue());  
 performanceData.put(algorithmName, avgTime);  
 }  
 return performanceData;  
 }  
  
 private void displayProgressWithCancelOption(String algorithmName, int iteration) {  
 int progress = (iteration + 1) \* 10;  
 System.*out*.printf("\r%s Progresso: [%s] %d%%", algorithmName, "=".repeat(progress / 10), progress);  
 }  
  
  
  
 public void savePerformanceReport() {  
 if (cancelFlag.get()) return;  
  
 File outputFolder = new File("output\_data");  
 if (!outputFolder.exists() && !outputFolder.mkdir()) {  
 System.*out*.println("Falha ao criar a pasta output\_data.");  
 return;  
 }  
  
 String timestamp = new SimpleDateFormat("yyyyMMdd\_HHmmss").format(new Date());  
 File reportFile = new File(outputFolder, sampleName + "\_report\_" + timestamp + ".txt");  
  
 try (PrintWriter writer = new PrintWriter(new FileWriter(reportFile))) {  
  
 PrintStream console = System.*out*;  
  
  
 Map<String, Double> averageTimes = new HashMap<>();  
  
  
 printBoth(writer, console, "Relatório de Desempenho:");  
 printBoth(writer, console, "Quantidade de dados: " + originalData.length + " elementos");  
  
  
 printBoth(writer, console, "\nTabela de Resumo (Tempo Médio, Comparações e Trocas):");  
 printBoth(writer, console, "-------------------------------------------------------");  
 printBoth(writer, console, String.*format*("%-15s %-20s %-15s %-15s", "Algoritmo", "Tempo Médio (ms)", "Comparações", "Trocas"));  
  
  
 for (String algorithmName : executionTimes.keySet()) {  
 List<Double> times = executionTimes.get(algorithmName);  
  
 if (times.isEmpty()) {  
 printBoth(writer, console, algorithmName + " - Nenhum tempo registrado.");  
 continue;  
 }  
  
 double averageMs = calculateAverage(times);  
 long avgComparisons = totalComparisons.getOrDefault(algorithmName, 0L);  
 long avgSwaps = totalSwaps.getOrDefault(algorithmName, 0L);  
  
  
 averageTimes.put(algorithmName, averageMs);  
  
  
 printBoth(writer, console, String.*format*("%-15s %-20.2f %-15d %-15d", algorithmName, averageMs, avgComparisons, avgSwaps));  
 }  
  
  
 for (Map.Entry<String, List<Double>> entry : executionTimes.entrySet()) {  
 String algorithmName = entry.getKey();  
 List<Double> times = entry.getValue();  
  
 if (times.isEmpty()) {  
 printBoth(writer, console, algorithmName + " - Nenhum tempo registrado.");  
 continue;  
 }  
  
 double averageMs = calculateAverage(times);  
 double medianMs = calculateMedian(times);  
 double stdDevMs = stdDeviations.get(algorithmName);  
 printBoth(writer, console, "\n" + algorithmName + " - Tempo Médio: " + String.*format*("%.2f", averageMs) + " ms, Mediana: " + String.*format*("%.2f", medianMs) + " ms, Desvio Padrão: " + String.*format*("%.2f", stdDevMs) + " ms");  
 printBoth(writer, console, algorithmName + " - Comparações Médias: " + totalComparisons.get(algorithmName) + ", Trocas Médias: " + totalSwaps.get(algorithmName));  
  
  
 printBoth(writer, console, "\nDistribuição de Tempo de Execução por Iteração:");  
 for (int i = 0; i < times.size(); i++) {  
 printBoth(writer, console, "Iteração " + (i + 1) + ": " + String.*format*("%.2f", times.get(i)) + " ms");  
 }  
  
 double highOutlierThreshold = averageMs + 2 \* stdDevMs;  
 double lowOutlierThreshold = averageMs - 2 \* stdDevMs;  
 printBoth(writer, console, "\nAnálise de Outliers:");  
 for (double time : times) {  
 if (time > highOutlierThreshold) {  
 printBoth(writer, console, "Tempo de Execução Alto Anômalo: " + String.*format*("%.2f", time) + " ms (acima de " + String.*format*("%.2f", highOutlierThreshold) + " ms)");  
 } else if (time < lowOutlierThreshold) {  
 printBoth(writer, console, "Tempo de Execução Baixo Anômalo: " + String.*format*("%.2f", time) + " ms (abaixo de " + String.*format*("%.2f", lowOutlierThreshold) + " ms)");  
 }  
 }  
 printBoth(writer, console, "");  
 }  
  
  
 String bestAlgorithm = Collections.*min*(averageTimes.entrySet(), Map.Entry.*comparingByValue*()).getKey();  
 double bestTime = averageTimes.get(bestAlgorithm);  
  
 printBoth(writer, console, "\nAlgoritmo mais eficiente: " + bestAlgorithm);  
 for (Map.Entry<String, Double> entry : averageTimes.entrySet()) {  
 String algorithmName = entry.getKey();  
 double time = entry.getValue();  
 if (!algorithmName.equals(bestAlgorithm)) {  
 double efficiencyRatio = time / bestTime;  
 double timeDifference = time - bestTime;  
 printBoth(writer, console, algorithmName + " foi " + String.*format*("%.2f", efficiencyRatio) + " vezes mais lento que " + bestAlgorithm + ", diferença de " + String.*format*("%.2f", timeDifference) + " ms");  
 }  
 }  
  
  
 String lowestStdDevAlgorithm = Collections.*min*(stdDeviations.entrySet(), Map.Entry.*comparingByValue*()).getKey();  
 printBoth(writer, console, "\nAlgoritmo com menor desvio padrão: " + lowestStdDevAlgorithm + " (Desvio Padrão: " + String.*format*("%.2f", stdDeviations.get(lowestStdDevAlgorithm)) + " ms)");  
  
  
 printBoth(writer, console, "\nComparação Gráfica de Tempo de Execução:");  
 for (Map.Entry<String, Double> entry : averageTimes.entrySet()) {  
 String algorithmName = entry.getKey();  
 double relativeTime = entry.getValue() / bestTime;  
 int barLength = (int) (relativeTime \* 20);  
 printBoth(writer, console, String.*format*("%-15s | %s (%.2f ms)", algorithmName, "=".repeat(barLength), entry.getValue()));  
 }  
  
  
 double minBaseline = 30.0;  
 printBoth(writer, console, "\n------------------------------------------------------------");  
 printBoth(writer, console, "Tabela de Avaliação (0-100)");  
 printBoth(writer, console, "------------------------------------------------------------");  
 printBoth(writer, console, String.*format*("%-15s %-15s %-15s %-15s", "Algoritmo", "Nota Tempo", "Nota Comp.", "Nota Final"));  
  
 for (String algorithmName : averageTimes.keySet()) {  
 double timeScore = calculateScore(averageTimes.get(algorithmName), minBaseline, calculateMinMax(averageTimes));  
 double comparisonScore = calculateScore(totalComparisons.getOrDefault(algorithmName, 0L), minBaseline, calculateMinMax(totalComparisons));  
 double swapScore = calculateScore(totalSwaps.getOrDefault(algorithmName, 0L), minBaseline, calculateMinMax(totalSwaps));  
 double finalScore = (timeScore \* 0.9) + (comparisonScore \* 0.05) + (swapScore \* 0.05);  
  
 printBoth(writer, console, String.*format*("%-15s %-15.2f %-15.2f %-15.2f", algorithmName, timeScore, comparisonScore, finalScore));  
 }  
  
  
 printBoth(writer, console, "\n------------------------------------------------------------");  
 printBoth(writer, console, "Resumo");  
 printBoth(writer, console, "------------------------------------------------------------");  
 printBoth(writer, console, "Algoritmo mais eficiente: " + bestAlgorithm + " (Tempo Médio: " + String.*format*("%.2f", bestTime) + " ms)");  
 printBoth(writer, console, "Algoritmo com menor desvio padrão: " + lowestStdDevAlgorithm + " (Desvio Padrão: " + String.*format*("%.2f", stdDeviations.get(lowestStdDevAlgorithm)) + " ms)");  
 printBoth(writer, console, "\nRecomendações para Cenários Específicos:");  
 printBoth(writer, console, "------------------------------------------------------------");  
  
 if (bestAlgorithm.equals("MergeSort")) {  
 printBoth(writer, console, "MergeSort é recomendado para conjuntos de dados grandes devido à sua consistência e estabilidade.");  
 }  
 if (bestAlgorithm.equals("QuickSort")) {  
 printBoth(writer, console, "QuickSort é ideal para conjuntos de dados parcialmente ordenados ou pequenos, pois tende a ser mais rápido nesses casos.");  
 }  
 if (bestAlgorithm.equals("HeapSort")) {  
 printBoth(writer, console, "HeapSort é indicado para dados aleatórios onde a estabilidade não é um requisito.");  
 }  
  
 printBoth(writer, console, "\nPara dados com grande variação, é recomendável usar algoritmos com menor desvio padrão (ex.: " + lowestStdDevAlgorithm + ").");  
  
 System.*out*.println("Relatório salvo em: " + reportFile.getAbsolutePath());  
 System.*out*.println("\n");  
 } catch (IOException e) {  
 System.*out*.println("Erro ao salvar o relatório: " + e.getMessage());  
 }  
 }  
  
  
 private void printBoth(PrintWriter writer, PrintStream console, String text) {  
 writer.println(text);  
 console.println(text);  
 }  
  
  
 private double calculateMedian(List<Double> values) {  
 Collections.*sort*(values);  
 int middle = values.size() / 2;  
 if (values.size() % 2 == 0) {  
 return (values.get(middle - 1) + values.get(middle)) / 2.0;  
 } else {  
 return values.get(middle);  
 }  
 }  
  
  
 private <T extends Number & Comparable<T>> MinMax calculateMinMax(Map<String, T> data) {  
 T max = Collections.*max*(data.values());  
 T min = Collections.*min*(data.values());  
 return new MinMax(min.doubleValue(), max.doubleValue());  
 }  
  
  
  
 private double calculateScore(double value, double minBaseline, MinMax minMax) {  
 return (minMax.max > minMax.min) ? Math.*max*((1 - (value - minMax.min) / (minMax.max - minMax.min)) \* 100, minBaseline) : 100;  
 }  
  
  
 private static class MinMax {  
 double min, max;  
 MinMax(double min, double max) {  
 this.min = min;  
 this.max = max;  
 }  
 }  
  
}  
  
  
 class SortingUtils {  
 public static void insertionSort(String[] array, int low, int high, Statistics stats) {  
 for (int i = low + 1; i <= high; i++) {  
 String key = array[i];  
 int j = i - 1;  
 while (j >= low && *compare*(array[j], key) > 0) {  
 stats.incrementComparisons();  
 array[j + 1] = array[j];  
 j--;  
 stats.incrementSwaps();  
 }  
 array[j + 1] = key;  
 }  
 }  
  
 private static int compare(String a, String b) {  
 return Integer.*compare*(*extractNumericPart*(a), *extractNumericPart*(b));  
 }  
  
 private static int extractNumericPart(String data) {  
 return Integer.*parseInt*(data.split(":")[0]);  
 }  
}  
  
  
  
class Timer {  
 public static double measureTimeMs(Runnable algorithm) {  
 long startTime = System.*nanoTime*();  
 algorithm.run();  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime) / 1\_000\_000.0;  
 }  
}  
  
class QuickSort implements SortingAlgorithm {  
 private static final int *INSERTION\_SORT\_THRESHOLD* = 32;  
 private final boolean showSteps;  
  
  
 public QuickSort() {  
 this.showSteps = false;  
 }  
  
  
 public QuickSort(boolean showSteps) {  
 this.showSteps = showSteps;  
 }  
  
 @Override  
 public void sort(String[] array, AtomicBoolean cancelFlag, Statistics stats) {  
 quickSortIterative(array, cancelFlag, stats);  
 }  
  
 private void quickSortIterative(String[] array, AtomicBoolean cancelFlag, Statistics stats) {  
 if (cancelFlag.get()) return;  
  
 Stack<int[]> stack = new Stack<>();  
 stack.push(new int[]{0, array.length - 1});  
  
 while (!stack.isEmpty()) {  
 if (cancelFlag.get()) return;  
  
 int[] range = stack.pop();  
 int low = range[0];  
 int high = range[1];  
  
 if (high - low <= *INSERTION\_SORT\_THRESHOLD*) {  
 SortingUtils.*insertionSort*(array, low, high, stats);  
 continue;  
 }  
  
 int pivotIndex = medianOfThreePartition(array, low, high, stats);  
  
 if (pivotIndex - 1 - low > high - pivotIndex - 1) {  
 stack.push(new int[]{low, pivotIndex - 1});  
 stack.push(new int[]{pivotIndex + 1, high});  
 } else {  
 stack.push(new int[]{pivotIndex + 1, high});  
 stack.push(new int[]{low, pivotIndex - 1});  
 }  
 }  
 }  
  
 private int medianOfThreePartition(String[] array, int low, int high, Statistics stats) {  
 int mid = low + (high - low) / 2;  
  
 if (compare(array[low], array[mid]) > 0) swap(array, low, mid, stats);  
 if (compare(array[low], array[high]) > 0) swap(array, low, high, stats);  
 if (compare(array[mid], array[high]) > 0) swap(array, mid, high, stats);  
  
 String pivot = array[mid];  
 swap(array, mid, high - 1, stats);  
 int i = low;  
 int j = high - 1;  
  
 while (true) {  
 while (compare(array[++i], pivot) < 0) if (i == high - 1) break;  
 while (compare(array[--j], pivot) > 0) if (j == low) break;  
 if (i >= j) break;  
 swap(array, i, j, stats);  
 }  
 swap(array, i, high - 1, stats);  
 return i;  
 }  
  
 private void swap(String[] array, int i, int j, Statistics stats) {  
 String temp = array[i];  
 array[i] = array[j];  
 array[j] = temp;  
 stats.incrementSwaps();  
 if (showSteps) {  
 System.*out*.printf("Troca realizada: %s <-> %s\n", array[i], array[j]);  
 }  
 }  
  
 private int compare(String a, String b) {  
 int result = Integer.*compare*(extractNumericPart(a), extractNumericPart(b));  
 if (showSteps) {  
 System.*out*.printf("Comparação realizada: %s e %s (resultado: %d)\n", a, b, result);  
 }  
 return result;  
 }  
  
 private int extractNumericPart(String data) {  
 return Integer.*parseInt*(data.split(":")[0]);  
 }  
}  
  
class MergeSort implements SortingAlgorithm {  
 private static final int *INSERTION\_SORT\_THRESHOLD* = 32;  
  
 @Override  
 public void sort(String[] array, AtomicBoolean cancelFlag, Statistics stats) {  
 String[] aux = new String[array.length];  
 System.*arraycopy*(array, 0, aux, 0, array.length);  
 mergeSort(array, aux, 0, array.length - 1, cancelFlag, stats);  
 }  
  
 private void mergeSort(String[] array, String[] aux, int left, int right, AtomicBoolean cancelFlag, Statistics stats) {  
 if (left >= right || cancelFlag.get()) return;  
  
 if (right - left <= *INSERTION\_SORT\_THRESHOLD*) {  
 SortingUtils.*insertionSort*(array, left, right, stats);  
 return;  
 }  
  
 int middle = left + (right - left) / 2;  
 mergeSort(aux, array, left, middle, cancelFlag, stats);  
 mergeSort(aux, array, middle + 1, right, cancelFlag, stats);  
  
 if (compare(aux[middle], aux[middle + 1]) <= 0) {  
 System.*arraycopy*(aux, left, array, left, right - left + 1);  
 return;  
 }  
 merge(array, aux, left, middle, right, stats);  
 }  
  
 private void merge(String[] array, String[] aux, int left, int middle, int right, Statistics stats) {  
 int i = left, j = middle + 1;  
 for (int k = left; k <= right; k++) {  
 stats.incrementComparisons();  
 if (i > middle) array[k] = aux[j++];  
 else if (j > right) array[k] = aux[i++];  
 else if (compare(aux[i], aux[j]) <= 0) array[k] = aux[i++];  
 else array[k] = aux[j++];  
 stats.incrementSwaps();  
 }  
 }  
  
 private int compare(String a, String b) {  
 return Integer.*compare*(extractNumericPart(a), extractNumericPart(b));  
 }  
  
 private int extractNumericPart(String data) {  
 return Integer.*parseInt*(data.split(":")[0]);  
 }  
}  
  
class HeapSort implements SortingAlgorithm {  
 @Override  
 public void sort(String[] array, AtomicBoolean cancelFlag, Statistics stats) {  
 int n = array.length;  
  
  
 for (int i = n / 2 - 1; i >= 0 && !cancelFlag.get(); i--) {  
 siftDown(array, i, n, stats);  
 }  
  
 for (int i = n - 1; i > 0 && !cancelFlag.get(); i--) {  
 swap(array, 0, i, stats);  
 siftDown(array, 0, i, stats);  
 }  
 }  
  
 private void siftDown(String[] array, int i, int n, Statistics stats) {  
 while (true) {  
 int largest = i;  
 int left = 2 \* i + 1;  
 int right = 2 \* i + 2;  
  
 if (left < n && compare(array[left], array[largest]) > 0) {  
 stats.incrementComparisons();  
 largest = left;  
 } else if (left < n) {  
 stats.incrementComparisons();  
 }  
  
 if (right < n && compare(array[right], array[largest]) > 0) {  
 stats.incrementComparisons();  
 largest = right;  
 } else if (right < n) {  
 stats.incrementComparisons();  
 }  
  
 if (largest == i) break;  
  
 swap(array, i, largest, stats);  
 i = largest;  
 }  
 }  
  
 private void swap(String[] array, int i, int j, Statistics stats) {  
 String temp = array[i];  
 array[i] = array[j];  
 array[j] = temp;  
 stats.incrementSwaps();  
 }  
  
 private int compare(String a, String b) {  
 return Integer.*compare*(extractNumericPart(a), extractNumericPart(b));  
 }  
  
 private int extractNumericPart(String data) {  
 return Integer.*parseInt*(data.split(":")[0]);  
 }  
}

**Bibliografia**

GeeksforGeeks. Sorting Algorithms. Disponível em: https://www.geeksforgeeks.org/sorting-algorithms/

Wikipedia. Sorting Algorithm. Disponível em: https://en.wikipedia.org/wiki/Sorting\_algorithm

WScube Tech. Data Structures & Algorithms: Sorting Algorithms. Disponível em: https://www.wscubetech.com/resources/dsa/sorting-algorithms

Programiz. Sorting Algorithms. Disponível em: https://www.programiz.com/dsa/sorting-algorithm

VisuAlgo. Sorting Algorithms Visualization. Disponível em: https://visualgo.net/en/sorting

Shiksha. Types of Sorting Algorithms. Disponível em: https://www.shiksha.com/online-courses/articles/types-of-sorting-algorithm-blogId-148505

Toptal Developers. Sorting Algorithms. Disponível em: https://www.toptal.com/developers/sorting-algorithms

LAMFO-UnB. Sorting Algorithms. Disponível em: https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/

MDN Web Docs. Array.prototype.sort() - JavaScript. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array/sort

BBC Bitesize. Sorting Algorithms. Disponível em: <https://www.bbc.co.uk/bitesize/guides/zjdkw6f/revision/1>  
  
Manzato, M. (2020). Algoritmos e Estruturas de Dados: Uma Abordagem Didática. São Paulo, SP: Novatec Editora.

Ziviani, N. (2004). Projeto de Algoritmos: Com Implementações em Pascal e C (3ª ed.). São Paulo, SP: Cengage Learning.

Lage, P. C. R., & Macedo, A. A. (2006). Estruturas de Dados Fundamentais: Conceitos e Aplicações. São Paulo, SP: LTC Editora.

Forbellone, A. L. V., & Eberspächer, H. F. (2000). Lógica de Programação: A Construção de Algoritmos e Estruturas de Dados. São Paulo, SP: Pearson Prentice Hall.

Ziviani, N. (2006). Algoritmos em C: Fundamentos e Aplicações. São Paulo, SP: Thomson Learning.

Preiss, B. R. (1999). Estruturas de Dados e Algoritmos: Análise e Projeto em Java. Rio de Janeiro, RJ: LTC Editora.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2002). Algoritmos: Teoria e Prática (2ª ed.). Rio de Janeiro, RJ: Elsevier.

Sedgewick, R. (2003). Algoritmos em C (3ª ed.). São Paulo, SP: Pearson Makron Books.

Tenenbaum, A. M. (2003). Estruturas de Dados Usando C (2ª ed.). São Paulo, SP: Pearson Makron Books.

Laudon, J. & Laudon, K. (2004). Estruturas de Dados e Algoritmos em Java. São Paulo, SP: Pearson.

