Relatório do Software Anti-plágio CopySpider
Para mais detalhes sobre o CopySpider, acesse:https://copyspider.com.br

Instruções
Este relatório apresenta na próxima página uma tabela com o resumo da análise do CopySpider. Cada linha associa o conteúdo do arquivo de entrada com um documento encontrado na internet (para "Busca em arquivos da internet") ou do arquivo de entrada com outros arquivos em seu computador (para "Pesquisa em arquivos locais").

A quantidade de termos comuns representa um fator utilizado no cálculo de similaridade dos arquivos. Quanto maior a quantidade de termos comuns, combinada com o agrupamento desses termos, maior a similaridade entre os arquivos.

No início de cada comparação entre arquivos, encontram-se um resumo numérico dos resultados:
- Arquivo 1: <nome do arquivo> (<Ni> termos)
- Arquivo 2: <nome do arquivo> (<Nc> termos)
- Termos comuns: <N>
- Similaridade:
  * Índice antigo (S): <x> %
  * Índice novo (Si): <y> %
  * Agrupamento (Sg): <Alto|Moderado|Baixo>

No texto do documento, os termos em comum são marcados em cores diferentes:
- Amarelo: quando são considerados no cálculo do Novo Índice de Semelhança (Si) e;
- Vermelho: quando estão agrupados e fazem parte do Índice de Agrupamento (Sg).

Os termos marcados em amarelo são comuns entre os documentos, mas, por não estarem agrupados, tendem a não caracterizar cópia. Os termos marcados em vermelho também são comuns e têm maior chance de serem interpretados como cópia.

É importante destacar que a classificação da semelhança como Alta, Moderada e Baixa não representa um "índice de plágio". Por exemplo, documentos que citam de forma direta (transcrição) outros documentos, podem ter uma similaridade Alta e ainda assim não podem ser caracterizados como plágio. Há sempre a necessidade do avaliador fazer uma análise para decidir se as semelhanças encontradas caracterizam ou não o problema de plágio ou mesmo de erro de formatação ou adequação às normas de referências bibliográficas.

Veja também:
Analisando o resultado do CopySpider
Qual o percentual aceitável para ser considerado plágio?
Como interpretar os índices de semelhança?

Versão do CopySpider: 3.5
Relatório gerado por: luizsouza1411@outlook.com
Análise no modo: Web/Normal (disponibilidade de 99.17%) em 17:40 s
Idioma da busca: Português

Arquivos	Termos comuns	Semelhança	Agrupamento
APS 2o. Relatório ABNT.docx X www.dio.me/articles/merge-sort-78e447956565	174	Baixa	Moderado
APS 2o. Relatório ABNT.docx X medium.com/@deutnerg/dominando-algoritmos-de-ordena%C3%A7%C3%A3o-exemplos-pr%C3%A1ticos-em-python-para-desenvolvedores-c69659e612f1	101	Baixa	Moderado
APS 2o. Relatório ABNT.docx X dev.to/mrpunkdasilva/entendendo-bubble-sort-304e	81	Baixa	Moderado
APS 2o. Relatório ABNT.docx X guievbs.github.io/sorting-algorithms/quick_sort	72	Baixa	Moderado
APS 2o. Relatório ABNT.docx X pt.wikipedia.org/wiki/Merge_sort	47	Baixa	Moderado
APS 2o. Relatório ABNT.docx X pt.wikipedia.org/wiki/Melhor_caso%2C_pior_caso_e_caso_m%C3%A9dio	35	Baixa	Moderado
APS 2o. Relatório ABNT.docx X www.dio.me/articles/trabalho-academico-algoritmo-de-ordenacao-selection-sort-92aef5fc1119	261	Baixa	Baixo
APS 2o. Relatório ABNT.docx X www.kufunda.net/publicdocs/Estruturas de dados e seus algoritmos (3a. ed.). (Jayme Luiz Szwarcfiter).pdf	236	Baixa	Baixo
APS 2o. Relatório ABNT.docx X www.dio.me/articles/quick-sort-origem-funcionamento-e-aplicacoes-praticas-artigo-b0757c97fc62	207	Baixa	Baixo
APS 2o. Relatório ABNT.docx X www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c	196	Baixa	Baixo

Arquivos com problema de download

https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/OMergeSort.html - Não foi possível baixar o arquivo. É recomendável baixar o arquivo manualmente e realizar a análise em conluio (Um contra todos). - (22) The requested URL returned error: 404; [csu] timeout

Arquivos com problema de conversão

https://www.lucasdorioverde.mt.gov.br/arquivos/userfiles/educacao/MATERIAL_DIDATICO/LIVRO_5_ANO_EF_Lucas_do_Rio_Verde.pdf - Não foi possível converter o arquivo. É recomendável converter o arquivo para texto manualmente e realizar a análise em conluio (Um contra todos).: Erro ao tentar converter: Page tree root must be a dictionary

https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort - Não foi possível converter o arquivo. É recomendável converter o arquivo para texto manualmente e realizar a análise em conluio (Um contra todos).: msg.the_file_is_empty

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)
Arquivo 2: www.dio.me/articles/merge-sort-78e447956565 (4768 termos)
Termos comuns: 174
Similaridade
Índice antigo (S): 2,05%
Índice novo (Si): 4,49%
Agrupamento (Sg): Moderado
O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 76f36445o40b25t25

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.

Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.

Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.

A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados

esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal $O(n^2)$ tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos para produzir o array completamente ordenado.
Funcionamento do Algoritmo:

O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.

Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.

3.5 ANÁLISE DE COMPLEXIDADE

3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL

A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE

Bubble Sort: Apresenta complexidade quadrática $O(n^2)$ tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso $O(n)$ ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço $O(1)$.

Selection Sort: Mantém complexidade $O(n^2)$ em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.

Merge Sort: Garante complexidade $O(n \log n)$ em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar $O(n)$ para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.

Quick Sort: Apresenta complexidade $O(n \log n)$ no caso médio, mas pode degradar para $O(n^2)$ no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar $O(\log n)$ para a pilha de recursão.

3.6 RESULTADOS EXPERIMENTAIS

3.6.1 METODOLOGIA DE TESTES

Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:

Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.

Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.

Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.

3.6.2 AMBIENTE DE EXECUÇÃO

Configuração do Sistema:

Linguagem: Python 3.x

Tamanho do conjunto de dados: 1.000 registros

Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas

3.6.3 RESULTADOS OBTIDOS

Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS

Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :

Observações sobre os dados coletados:

Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.

Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).

Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade $O(n^2)$.

Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).

3.7 ANÁLISE COMPARATIVA

3.7.1 USO DE MEMÓRIA

A análise do consumo de memória revela características distintas entre os algoritmos:

Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.

Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA

Escalabilidade:

Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários. No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando

45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos O(n^2)
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso O(n^2), os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo O(n log n) em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA
A solução foi organizada em uma arquitetura modular composta por três componentes principais:
4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS
main.py - Módulo Principal
Responsável pela orquestração do fluxo de execução completo
Realiza o carregamento dos dados via pandas
Executa sequencialmente os quatro algoritmos de ordenação
Coleta métricas de desempenho (tempo, memória, comparações, trocas)
Apresenta resultados formatados e análise comparativa
sorts.py - Módulo de Algoritmos
Contém as implementações dos quatro algoritmos de ordenação
Cada função retorna uma tupla: (array_ordenado, comparações, trocas)
Implementa contadores internos para rastreamento de operações
Quick Sort otimizado com seleção de pivô médio para evitar pior caso
leitura.py - Módulo de Leitura de Dados
Fornece função reutilizável para carregamento de arquivos CSV
Implementa validação de existência do arquivo
Verifica presença da coluna "Pontos" requerida
Trata exceções com mensagens descritivas
4.3 FUNCIONALIDADES IMPLEMENTADAS
4.3.1 CARREGAMENTO INTELIGENTE DE DADOS
A aplicação implementa carregamento robusto de dados com as seguintes características:
Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV
Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada
Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução
Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS
O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:
Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original
Medição de Tempo: Utilização do módulo time para cronometragem precisa
Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória
Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação
Carregamento e validação dos dados
Apresentação de estatísticas preliminares

Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise
4.6 TECNOLOGIAS UTILIZADAS
4.6.1 LINGUAGEM E BIBLIOTECAS
A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.
Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.
A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.
4.6.2 FORMATO DE DADOS
Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS
4.7.1 REUTILIZAÇÃO DE CÓDIGO
O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos
4.7.2 DOCUMENTAÇÃO INTERNA
O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas
Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS

Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)

4.9 CONCLUSÃO DA SOLUÇÃO

A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.

Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE

```
# ALGORITMOS

def bubble_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
            comparacoes += 1
            if arr[j] &gt; arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
    arr = arr[:]
    comparacoes = 0
```

```
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
    step = 1
    n = len(arr)
    while step &lt; n:
    for i in range(0, n, step * 2):
```

```
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
             if a[j] &lt;= pivot:
             i += 1
                if i != j:
                a[i], a[j] = a[j], a[i]
                trocas += 1
                    if i + 1 != high:
                     a[i+1], a[high] = a[high], a[i+1]
                     trocas += 1
              return i+1
    def qs(a, low, high):
          if low &lt; high:
          pi = partition(a, low, high)
         qs(a, low, pi-1)
         qs(a, pi+1, high)
          qs(arr, 0, len(arr)-1)
          return arr, comparacoes, trocas

#Leitura
import pandas as pd
```

```
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
print("="*70)
algoritmos = {
```

```
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
```

```
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em: https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Sim
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º
Merge Sort	0,1447	0,0281	8.686	10.000	2º
Selection Sort	2,2298	0,0153	499.500	992	3º

Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)

Arquivo 2:

medium.com/@deutnerg/dominando-algoritmos-de-ordena%C3%A7%C3%A3o-exemplos-pr%C3%A1ticos-em-python-para-desenvolvedores-c69659e612f1 (1997 termos)

Termos comuns: 101

Similaridade

Índice antigo (S): 1,75%

Índice novo (Si): 2,60%

Agrupamento (Sg): Moderado

O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 9065ad9co46b22t22

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados
Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.

Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.

Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.

A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do

menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal $O(n^2)$ tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos

para produzir o array completamente ordenado.
Funcionamento do Algoritmo:
O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias

progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.
Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.
3.5 ANÁLISE DE COMPLEXIDADE
3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL
A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE
Bubble Sort: Apresenta complexidade quadrática O(n^2) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).
Selection Sort: Mantém complexidade O(n^2) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.
Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.
Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n^2) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.
3.6 RESULTADOS EXPERIMENTAIS
3.6.1 METODOLOGIA DE TESTES
Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:
Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.
Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.
Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.
3.6.2 AMBIENTE DE EXECUÇÃO
Configuração do Sistema:

Linguagem: Python 3.x
Tamanho do conjunto de dados: 1.000 registros
Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas
3.6.3 RESULTADOS OBTIDOS
Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS
Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :
Observações sobre os dados coletados:
Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.
Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).
Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade O(n^2).
Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).
3.7 ANÁLISE COMPARATIVA
3.7.1 USO DE MEMÓRIA
A análise do consumo de memória revela características distintas entre os algoritmos:
Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.
Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA
Escalabilidade:
Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em

relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando 45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos $O(n^2)$
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso $O(n^2)$, os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo $O(n \log n)$ em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de

responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA

A solução foi organizada em uma arquitetura modular composta por três componentes principais:

4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS

main.py - Módulo Principal

Responsável pela orquestração do fluxo de execução completo

Realiza o carregamento dos dados via pandas

Executa sequencialmente os quatro algoritmos de ordenação

Coleta métricas de desempenho (tempo, memória, comparações, trocas)

Apresenta resultados formatados e análise comparativa

sorts.py - Módulo de Algoritmos

Contém as implementações dos quatro algoritmos de ordenação

Cada função retorna uma tupla: (array_ordenado, comparações, trocas)

Implementa contadores internos para rastreamento de operações

Quick Sort otimizado com seleção de pivô médio para evitar pior caso

leitura.py - Módulo de Leitura de Dados

Fornece função reutilizável para carregamento de arquivos CSV

Implementa validação de existência do arquivo

Verifica presença da coluna "Pontos" requerida

Trata exceções com mensagens descritivas

4.3 FUNCIONALIDADES IMPLEMENTADAS

4.3.1 CARREGAMENTO INTELIGENTE DE DADOS

A aplicação implementa carregamento robusto de dados com as seguintes características:

Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV

Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada

Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução

Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS

O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:

Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original

Medição de Tempo: Utilização do módulo time para cronometragem precisa

Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória

Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação

Carregamento e validação dos dados
Apresentação de estatísticas preliminares
Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise

4.6 TECNOLOGIAS UTILIZADAS

4.6.1 LINGUAGEM E BIBLIOTECAS

A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.

Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.

A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.

4.6.2 FORMATO DE DADOS

Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS

4.7.1 REUTILIZAÇÃO DE CÓDIGO

O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos

4.7.2 DOCUMENTAÇÃO INTERNA

O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas

Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS
Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)
4.9 CONCLUSÃO DA SOLUÇÃO
A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.
Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE
ALGORITMOS

```
def bubble_sort(arr):
  arr = arr[:]
  comparacoes = 0
  trocas = 0
  for i in range(len(arr)):
      for j in range(len(arr) - i - 1):
          comparacoes += 1
          if arr[j] &gt; arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
```

```
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
        step = 1
        n = len(arr)
```

```
    while step &lt; n:
    for i in range(0, n, step * 2):
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
            if a[j] &lt;= pivot:
            i += 1
                if i != j:
                a[i], a[j] = a[j], a[i]
                trocas += 1
                    if i + 1 != high:
                     a[i+1], a[high] = a[high], a[i+1]
                     trocas += 1
              return i+1
    def qs(a, low, high):
          if low &lt; high:
          pi = partition(a, low, high)
         qs(a, low, pi-1)
         qs(a, pi+1, high)
          qs(arr, 0, len(arr)-1)
          return arr, comparacoes, trocas
```

```
#Leitura
import pandas as pd
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
```

```
print("="*70)
algoritmos = {
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
```

```
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.
htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com
/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.
enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.
geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/
quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em:
https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://
programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/
metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/
index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Sim
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º

Merge Sort 0,1447 0,0281 8.686 10.000 2º
Selection Sort 2,2298 0,0153 499.500 992 3º
Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)
Arquivo 2: dev.to/mrpunkdasilva/entendendo-bubble-sort-304e (2949 termos)
Termos comuns: 81
Similaridade
Índice antigo (S): 1,20%
Índice novo (Si): 2,09%
Agrupamento (Sg): Moderado
O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 7331a5f2o48b44t44

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.

Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.

Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.

A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados

esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal $O(n^2)$ tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos para produzir o array completamente ordenado.
Funcionamento do Algoritmo:

O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.

Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.

3.5 ANÁLISE DE COMPLEXIDADE

3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL

A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE

Bubble Sort: Apresenta complexidade quadrática O(n^2) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).

Selection Sort: Mantém complexidade O(n^2) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.

Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.

Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n^2) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.

3.6 RESULTADOS EXPERIMENTAIS

3.6.1 METODOLOGIA DE TESTES

Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:

Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.

Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.

Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.

3.6.2 AMBIENTE DE EXECUÇÃO

Configuração do Sistema:

Linguagem: Python 3.x

Tamanho do conjunto de dados: 1.000 registros

Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas

3.6.3 RESULTADOS OBTIDOS

Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS

Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :

Observações sobre os dados coletados:

Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.

Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).

Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade $O(n^2)$.

Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).

3.7 ANÁLISE COMPARATIVA

3.7.1 USO DE MEMÓRIA

A análise do consumo de memória revela características distintas entre os algoritmos:

Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.

Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA

Escalabilidade:

Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando

45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos O(n^2)
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso O(n^2), os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo O(n log n) em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA
A solução foi organizada em uma arquitetura modular composta por três componentes principais:
4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS
main.py - Módulo Principal
Responsável pela orquestração do fluxo de execução completo
Realiza o carregamento dos dados via pandas
Executa sequencialmente os quatro algoritmos de ordenação
Coleta métricas de desempenho (tempo, memória, comparações, trocas)
Apresenta resultados formatados e análise comparativa
sorts.py - Módulo de Algoritmos
Contém as implementações dos quatro algoritmos de ordenação
Cada função retorna uma tupla: (array_ordenado, comparações, trocas)
Implementa contadores internos para rastreamento de operações
Quick Sort otimizado com seleção de pivô médio para evitar pior caso
leitura.py - Módulo de Leitura de Dados
Fornece função reutilizável para carregamento de arquivos CSV
Implementa validação de existência do arquivo
Verifica presença da coluna "Pontos" requerida
Trata exceções com mensagens descritivas
4.3 FUNCIONALIDADES IMPLEMENTADAS
4.3.1 CARREGAMENTO INTELIGENTE DE DADOS
A aplicação implementa carregamento robusto de dados com as seguintes características:
Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV
Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada
Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução
Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS
O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:
Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original
Medição de Tempo: Utilização do módulo time para cronometragem precisa
Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória
Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação
Carregamento e validação dos dados
Apresentação de estatísticas preliminares

Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise
4.6 TECNOLOGIAS UTILIZADAS
4.6.1 LINGUAGEM E BIBLIOTECAS
A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.
Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.
A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.
4.6.2 FORMATO DE DADOS
Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS
4.7.1 REUTILIZAÇÃO DE CÓDIGO
O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos
4.7.2 DOCUMENTAÇÃO INTERNA
O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas
Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS

Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)

4.9 CONCLUSÃO DA SOLUÇÃO

A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.

Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE

```
# ALGORITMOS

def bubble_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
            comparacoes += 1
            if arr[j] &gt; arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
    arr = arr[:]
    comparacoes = 0
```

```
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
        step = 1
        n = len(arr)
    while step &lt; n:
    for i in range(0, n, step * 2):
```

```
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
             if a[j] &lt;= pivot:
             i += 1
                if i != j:
                a[i], a[j] = a[j], a[i]
                trocas += 1
                    if i + 1 != high:
                     a[i+1], a[high] = a[high], a[i+1]
                     trocas += 1
              return i+1
    def qs(a, low, high):
          if low &lt; high:
          pi = partition(a, low, high)
         qs(a, low, pi-1)
         qs(a, pi+1, high)
          qs(arr, 0, len(arr)-1)
          return arr, comparacoes, trocas

#Leitura
import pandas as pd
```

```
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
print("="*70)
algoritmos = {
```

```
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
```

```
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em: https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Sim
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º
Merge Sort	0,1447	0,0281	8.686	10.000	2º
Selection Sort	2,2298	0,0153	499.500	992	3º

Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)
Arquivo 2: guievbs.github.io/sorting-algorithms/quick_sort (789 termos)
Termos comuns: 72
Similaridade
Índice antigo (S): 1,55%
Índice novo (Si): 1,85%
Agrupamento (Sg): Moderado
O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 5e61a300o42b24t24

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.

Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.

Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.

A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados

esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal $O(n^2)$ tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos para produzir o array completamente ordenado.
Funcionamento do Algoritmo:

O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.

3.4 Quick Sort

O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:

O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.

Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.

3.5 ANÁLISE DE COMPLEXIDADE

3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL

A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE

Bubble Sort: Apresenta complexidade quadrática O(n²) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).

Selection Sort: Mantém complexidade O(n²) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.

Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.

Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n²) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.

3.6 RESULTADOS EXPERIMENTAIS

3.6.1 METODOLOGIA DE TESTES

Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:

Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.

Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.

Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.

3.6.2 AMBIENTE DE EXECUÇÃO

Configuração do Sistema:

Linguagem: Python 3.x

Tamanho do conjunto de dados: 1.000 registros

Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas

3.6.3 RESULTADOS OBTIDOS

Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS

Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :

Observações sobre os dados coletados:

Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.

Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).

Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade $O(n^2)$.

Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).

3.7 ANÁLISE COMPARATIVA

3.7.1 USO DE MEMÓRIA

A análise do consumo de memória revela características distintas entre os algoritmos:

Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.

Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA

Escalabilidade:

Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando

45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos O(n^2)
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso O(n^2), os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo O(n log n) em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA
A solução foi organizada em uma arquitetura modular composta por três componentes principais:
4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS
main.py - Módulo Principal
Responsável pela orquestração do fluxo de execução completo
Realiza o carregamento dos dados via pandas
Executa sequencialmente os quatro algoritmos de ordenação
Coleta métricas de desempenho (tempo, memória, comparações, trocas)
Apresenta resultados formatados e análise comparativa
sorts.py - Módulo de Algoritmos
Contém as implementações dos quatro algoritmos de ordenação
Cada função retorna uma tupla: (array_ordenado, comparações, trocas)
Implementa contadores internos para rastreamento de operações
Quick Sort otimizado com seleção de pivô médio para evitar pior caso
leitura.py - Módulo de Leitura de Dados
Fornece função reutilizável para carregamento de arquivos CSV
Implementa validação de existência do arquivo
Verifica presença da coluna "Pontos" requerida
Trata exceções com mensagens descritivas
4.3 FUNCIONALIDADES IMPLEMENTADAS
4.3.1 CARREGAMENTO INTELIGENTE DE DADOS
A aplicação implementa carregamento robusto de dados com as seguintes características:
Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV
Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada
Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução
Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS
O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:
Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original
Medição de Tempo: Utilização do módulo time para cronometragem precisa
Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória
Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação
Carregamento e validação dos dados
Apresentação de estatísticas preliminares

Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise
4.6 TECNOLOGIAS UTILIZADAS
4.6.1 LINGUAGEM E BIBLIOTECAS
A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.
Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.
A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.
4.6.2 FORMATO DE DADOS
Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS
4.7.1 REUTILIZAÇÃO DE CÓDIGO
O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos
4.7.2 DOCUMENTAÇÃO INTERNA
O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas
Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS

Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)

4.9 CONCLUSÃO DA SOLUÇÃO

A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.

Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE

```
# ALGORITMOS

def bubble_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
            comparacoes += 1
            if arr[j] &gt; arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
    arr = arr[:]
    comparacoes = 0
```

```
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
    step = 1
    n = len(arr)
    while step &lt; n:
    for i in range(0, n, step * 2):
```

```
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
            if a[j] &lt;= pivot:
            i += 1
                if i != j:
                a[i], a[j] = a[j], a[i]
                trocas += 1
                    if i + 1 != high:
                     a[i+1], a[high] = a[high], a[i+1]
                     trocas += 1
              return i+1
    def qs(a, low, high):
           if low &lt; high:
           pi = partition(a, low, high)
          qs(a, low, pi-1)
          qs(a, pi+1, high)
           qs(arr, 0, len(arr)-1)
           return arr, comparacoes, trocas

#Leitura
import pandas as pd
```

```
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
print("="*70)
algoritmos = {
```

```
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
```

```
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em: https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Sim
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º
Merge Sort	0,1447	0,0281	8.686	10.000	2º
Selection Sort	2,2298	0,0153	499.500	992	3º

Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)
Arquivo 2: pt.wikipedia.org/wiki/Merge_sort (1652 termos)
Termos comuns: 47
Similaridade
Índice antigo (S): 0,85%
Índice novo (Si): 1,21%
Agrupamento (Sg): Moderado
O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 25900242o44b23t23

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.
Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.
Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.
A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados

esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal $O(n^2)$ tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos para produzir o array completamente ordenado.
Funcionamento do Algoritmo:

O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.

Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.

3.5 ANÁLISE DE COMPLEXIDADE

3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL

A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE

Bubble Sort: Apresenta complexidade quadrática O(n²) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).

Selection Sort: Mantém complexidade O(n²) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.

Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.

Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n²) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.

3.6 RESULTADOS EXPERIMENTAIS

3.6.1 METODOLOGIA DE TESTES

Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:

Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.

Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.

Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.

3.6.2 AMBIENTE DE EXECUÇÃO

Configuração do Sistema:

Linguagem: Python 3.x

Tamanho do conjunto de dados: 1.000 registros

Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas

3.6.3 RESULTADOS OBTIDOS

Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS

Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :

Observações sobre os dados coletados:

Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.

Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).

Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade $O(n^2)$.

Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).

3.7 ANÁLISE COMPARATIVA

3.7.1 USO DE MEMÓRIA

A análise do consumo de memória revela características distintas entre os algoritmos:

Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.

Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA

Escalabilidade:

Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando

45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos $O(n^2)$
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso $O(n^2)$, os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo $O(n \log n)$ em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA

4.1 VISÃO GERAL DA APLICAÇÃO

A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA
A solução foi organizada em uma arquitetura modular composta por três componentes principais:
4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS
main.py - Módulo Principal
Responsável pela orquestração do fluxo de execução completo
Realiza o carregamento dos dados via pandas
Executa sequencialmente os quatro algoritmos de ordenação
Coleta métricas de desempenho (tempo, memória, comparações, trocas)
Apresenta resultados formatados e análise comparativa
sorts.py - Módulo de Algoritmos
Contém as implementações dos quatro algoritmos de ordenação
Cada função retorna uma tupla: (array_ordenado, comparações, trocas)
Implementa contadores internos para rastreamento de operações
Quick Sort otimizado com seleção de pivô médio para evitar pior caso
leitura.py - Módulo de Leitura de Dados
Fornece função reutilizável para carregamento de arquivos CSV
Implementa validação de existência do arquivo
Verifica presença da coluna "Pontos" requerida
Trata exceções com mensagens descritivas
4.3 FUNCIONALIDADES IMPLEMENTADAS
4.3.1 CARREGAMENTO INTELIGENTE DE DADOS
A aplicação implementa carregamento robusto de dados com as seguintes características:
Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV
Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada
Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução
Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS
O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:
Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original
Medição de Tempo: Utilização do módulo time para cronometragem precisa
Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória
Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação
Carregamento e validação dos dados
Apresentação de estatísticas preliminares

Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise
4.6 TECNOLOGIAS UTILIZADAS
4.6.1 LINGUAGEM E BIBLIOTECAS
A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.
Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.
A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.
4.6.2 FORMATO DE DADOS
Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS
4.7.1 REUTILIZAÇÃO DE CÓDIGO
O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos
4.7.2 DOCUMENTAÇÃO INTERNA
O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas
Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS

Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)

4.9 CONCLUSÃO DA SOLUÇÃO

A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.

Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE

```
# ALGORITMOS

def bubble_sort(arr):
  arr = arr[:]
  comparacoes = 0
  trocas = 0
  for i in range(len(arr)):
      for j in range(len(arr) - i - 1):
          comparacoes += 1
          if arr[j] &gt; arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
  arr = arr[:]
  comparacoes = 0
```

```
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
        step = 1
        n = len(arr)
    while step &lt; n:
    for i in range(0, n, step * 2):
```

```
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
             if a[j] &lt;= pivot:
             i += 1
                 if i != j:
                 a[i], a[j] = a[j], a[i]
                 trocas += 1
                      if i + 1 != high:
                       a[i+1], a[high] = a[high], a[i+1]
                       trocas += 1
              return i+1
    def qs(a, low, high):
           if low &lt; high:
           pi = partition(a, low, high)
          qs(a, low, pi-1)
          qs(a, pi+1, high)
           qs(arr, 0, len(arr)-1)
           return arr, comparacoes, trocas

#Leitura
import pandas as pd
```

```
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
print("="*70)
algoritmos = {
```

```
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
```

```
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em: https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Sim
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º
Merge Sort	0,1447	0,0281	8.686	10.000	2º
Selection Sort	2,2298	0,0153	499.500	992	3º

Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)
Arquivo 2: pt.wikipedia.org/wiki/Melhor_caso%2C_pior_caso_e_caso_m%C3%A9dio (1526 termos)
Termos comuns: 35
Similaridade
Índice antigo (S): 0,65%
Índice novo (Si): 0,90%
Agrupamento (Sg): Moderado
O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: bd1642b3o32b31t31

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.
Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.
Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.
A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados

esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal $O(n^2)$ tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos para produzir o array completamente ordenado.
Funcionamento do Algoritmo:

O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.

Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.

3.5 ANÁLISE DE COMPLEXIDADE

3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL

A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE

Bubble Sort: Apresenta complexidade quadrática $O(n^2)$ tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso $O(n)$ ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço $O(1)$.

Selection Sort: Mantém complexidade $O(n^2)$ em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.

Merge Sort: Garante complexidade $O(n \log n)$ em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar $O(n)$ para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.

Quick Sort: Apresenta complexidade $O(n \log n)$ no caso médio, mas pode degradar para $O(n^2)$ no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar $O(\log n)$ para a pilha de recursão.

3.6 RESULTADOS EXPERIMENTAIS

3.6.1 METODOLOGIA DE TESTES

Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:

Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.

Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.

Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.

3.6.2 AMBIENTE DE EXECUÇÃO

Configuração do Sistema:

Linguagem: Python 3.x

Tamanho do conjunto de dados: 1.000 registros

Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas

3.6.3 RESULTADOS OBTIDOS

Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS

Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :

Observações sobre os dados coletados:

Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.

Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).

Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade $O(n^2)$.

Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).

3.7 ANÁLISE COMPARATIVA

3.7.1 USO DE MEMÓRIA

A análise do consumo de memória revela características distintas entre os algoritmos:

Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.

Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA

Escalabilidade:

Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando

45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos O(n^2)
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso O(n^2), os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo O(n log n) em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA
A solução foi organizada em uma arquitetura modular composta por três componentes principais:
4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS
main.py - Módulo Principal
Responsável pela orquestração do fluxo de execução completo
Realiza o carregamento dos dados via pandas
Executa sequencialmente os quatro algoritmos de ordenação
Coleta métricas de desempenho (tempo, memória, comparações, trocas)
Apresenta resultados formatados e análise comparativa
sorts.py - Módulo de Algoritmos
Contém as implementações dos quatro algoritmos de ordenação
Cada função retorna uma tupla: (array_ordenado, comparações, trocas)
Implementa contadores internos para rastreamento de operações
Quick Sort otimizado com seleção de pivô médio para evitar pior caso
leitura.py - Módulo de Leitura de Dados
Fornece função reutilizável para carregamento de arquivos CSV
Implementa validação de existência do arquivo
Verifica presença da coluna "Pontos" requerida
Trata exceções com mensagens descritivas
4.3 FUNCIONALIDADES IMPLEMENTADAS
4.3.1 CARREGAMENTO INTELIGENTE DE DADOS
A aplicação implementa carregamento robusto de dados com as seguintes características:
Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV
Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada
Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução
Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS
O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:
Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original
Medição de Tempo: Utilização do módulo time para cronometragem precisa
Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória
Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação
Carregamento e validação dos dados
Apresentação de estatísticas preliminares

Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise
4.6 TECNOLOGIAS UTILIZADAS
4.6.1 LINGUAGEM E BIBLIOTECAS
A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.
Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.
A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.
4.6.2 FORMATO DE DADOS
Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS
4.7.1 REUTILIZAÇÃO DE CÓDIGO
O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos
4.7.2 DOCUMENTAÇÃO INTERNA
O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas
Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS

Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)

4.9 CONCLUSÃO DA SOLUÇÃO

A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.

Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE

```
# ALGORITMOS

def bubble_sort(arr):
  arr = arr[:]
  comparacoes = 0
  trocas = 0
  for i in range(len(arr)):
      for j in range(len(arr) - i - 1):
          comparacoes += 1
          if arr[j] &gt; arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
  arr = arr[:]
  comparacoes = 0
```

```
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
        step = 1
        n = len(arr)
    while step &lt; n:
    for i in range(0, n, step * 2):
```

```
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
             if a[j] &lt;= pivot:
             i += 1
                 if i != j:
                 a[i], a[j] = a[j], a[i]
                 trocas += 1
                      if i + 1 != high:
                       a[i+1], a[high] = a[high], a[i+1]
                       trocas += 1
              return i+1
    def qs(a, low, high):
           if low &lt; high:
           pi = partition(a, low, high)
          qs(a, low, pi-1)
          qs(a, pi+1, high)
           qs(arr, 0, len(arr)-1)
           return arr, comparacoes, trocas

#Leitura
import pandas as pd
```

```
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
print("="*70)
algoritmos = {
```

```
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
```

```
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em: https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Sim
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º
Merge Sort	0,1447	0,0281	8.686	10.000	2º
Selection Sort	2,2298	0,0153	499.500	992	3º

Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)
Arquivo 2: www.dio.me/articles/trabalho-academico-algoritmo-de-ordenacao-selection-sort-92aef5fc1119 (8267 termos)
Termos comuns: 261
Similaridade
Índice antigo (S): 2,19%
Índice novo (Si): 6,73%
Agrupamento (Sg): Baixo
O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 55daee74o33b0t0

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e

Classificação de Dados
Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.

Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.

Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.

A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro

elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal $O(n^2)$ tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos para produzir o array completamente ordenado.

Funcionamento do Algoritmo:
O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho

mínimo.
Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.

3.5 ANÁLISE DE COMPLEXIDADE

3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL

A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE

Bubble Sort: Apresenta complexidade quadrática O(n^2) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).
Selection Sort: Mantém complexidade O(n^2) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.
Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.
Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n^2) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.

3.6 RESULTADOS EXPERIMENTAIS

3.6.1 METODOLOGIA DE TESTES

Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:
Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.
Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.
Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.

3.6.2 AMBIENTE DE EXECUÇÃO

Configuração do Sistema:
Linguagem: Python 3.x

Tamanho do conjunto de dados: 1.000 registros
Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas

3.6.3 RESULTADOS OBTIDOS

Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS

Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :

Observações sobre os dados coletados:

Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.

Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).

Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade $O(n^2)$.

Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).

3.7 ANÁLISE COMPARATIVA

3.7.1 USO DE MEMÓRIA

A análise do consumo de memória revela características distintas entre os algoritmos:

Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.

Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA

Escalabilidade:
Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários. No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.

2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando 45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos $O(n^2)$
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso $O(n^2)$, os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo $O(n \log n)$ em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do

código.

4.2 ARQUITETURA DO SISTEMA

A solução foi organizada em uma arquitetura modular composta por três componentes principais:

4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS

main.py - Módulo Principal

Responsável pela orquestração do fluxo de execução completo

Realiza o carregamento dos dados via pandas

Executa sequencialmente os quatro algoritmos de ordenação

Coleta métricas de desempenho (tempo, memória, comparações, trocas)

Apresenta resultados formatados e análise comparativa

sorts.py - Módulo de Algoritmos

Contém as implementações dos quatro algoritmos de ordenação

Cada função retorna uma tupla: (array_ordenado, comparações, trocas)

Implementa contadores internos para rastreamento de operações

Quick Sort otimizado com seleção de pivô médio para evitar pior caso

leitura.py - Módulo de Leitura de Dados

Fornece função reutilizável para carregamento de arquivos CSV

Implementa validação de existência do arquivo

Verifica presença da coluna "Pontos" requerida

Trata exceções com mensagens descritivas

4.3 FUNCIONALIDADES IMPLEMENTADAS

4.3.1 CARREGAMENTO INTELIGENTE DE DADOS

A aplicação implementa carregamento robusto de dados com as seguintes características:

Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV

Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada

Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução

Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS

O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:

Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original

Medição de Tempo: Utilização do módulo time para cronometragem precisa

Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória

Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação
Carregamento e validação dos dados

Apresentação de estatísticas preliminares
Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise
4.6 TECNOLOGIAS UTILIZADAS
4.6.1 LINGUAGEM E BIBLIOTECAS
A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.
Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.
A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.
4.6.2 FORMATO DE DADOS
Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS
4.7.1 REUTILIZAÇÃO DE CÓDIGO
O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos
4.7.2 DOCUMENTAÇÃO INTERNA
O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas
Separadores visuais para organização do código

4.8 LIMITAÇÕES E TRABALHOS FUTUROS
Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)
4.9 CONCLUSÃO DA SOLUÇÃO
A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.
Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE

```
# ALGORITMOS

def bubble_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
            comparacoes += 1
            if arr[j] &gt; arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
    arr = arr[:]
```

```
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
        step = 1
        n = len(arr)
    while step &lt; n:
```

```
    for i in range(0, n, step * 2):
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
             if a[j] &lt;= pivot:
             i += 1
                if i != j:
                a[i], a[j] = a[j], a[i]
                trocas += 1
                    if i + 1 != high:
                     a[i+1], a[high] = a[high], a[i+1]
                     trocas += 1
              return i+1
    def qs(a, low, high):
           if low &lt; high:
           pi = partition(a, low, high)
          qs(a, low, pi-1)
          qs(a, pi+1, high)
           qs(arr, 0, len(arr)-1)
           return arr, comparacoes, trocas

#Leitura
```

```
import pandas as pd
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
print("="*70)
```

```
algoritmos = {
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
```

```
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.
htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com
/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.
enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.
geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/
quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em:
https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://
programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/
metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/
index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Sim
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º
Merge Sort	0,1447	0,0281	8.686	10.000	2º

Selection Sort 2,2298 0,0153 499.500 992 3º
Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)
Arquivo 2: www.kufunda.net/publicdocs/Estruturas de dados e seus algoritmos (3a. ed.). (Jayme Luiz Szwarcfiter).pdf (66737 termos)
Termos comuns: 236
Similaridade
Índice antigo (S): 0,33%
Índice novo (Si): 6,09%
Agrupamento (Sg): Baixo
O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 9b36b68co32b0t0

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e

Classificação de Dados
Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.

Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.

Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.

A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro

elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo O(n^2) tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal O(n^2) tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos para produzir o array completamente ordenado.

Funcionamento do Algoritmo:
O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho

mínimo.
Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.
3.5 ANÁLISE DE COMPLEXIDADE
3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL
A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE
Bubble Sort: Apresenta complexidade quadrática O(n^2) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).
Selection Sort: Mantém complexidade O(n^2) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.
Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.
Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n^2) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.
3.6 RESULTADOS EXPERIMENTAIS
3.6.1 METODOLOGIA DE TESTES
Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:
Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.
Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.
Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.
3.6.2 AMBIENTE DE EXECUÇÃO
Configuração do Sistema:
Linguagem: Python 3.x

Tamanho do conjunto de dados: 1.000 registros
Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas
3.6.3 RESULTADOS OBTIDOS
Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS
Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :
Observações sobre os dados coletados:
Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.
Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).
Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade $O(n^2)$.
Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).
3.7 ANÁLISE COMPARATIVA
3.7.1 USO DE MEMÓRIA
A análise do consumo de memória revela características distintas entre os algoritmos:
Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou $O(\log n)$, consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.
Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA

Escalabilidade:
Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.

2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando 45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos $O(n^2)$
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso $O(n^2)$, os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo $O(n \log n)$ em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do

código.

4.2 ARQUITETURA DO SISTEMA

A solução foi organizada em uma arquitetura modular composta por três componentes principais:

4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS

main.py - Módulo Principal

Responsável pela orquestração do fluxo de execução completo

Realiza o carregamento dos dados via pandas

Executa sequencialmente os quatro algoritmos de ordenação

Coleta métricas de desempenho (tempo, memória, comparações, trocas)

Apresenta resultados formatados e análise comparativa

sorts.py - Módulo de Algoritmos

Contém as implementações dos quatro algoritmos de ordenação

Cada função retorna uma tupla: (array_ordenado, comparações, trocas)

Implementa contadores internos para rastreamento de operações

Quick Sort otimizado com seleção de pivô médio para evitar pior caso

leitura.py - Módulo de Leitura de Dados

Fornece função reutilizável para carregamento de arquivos CSV

Implementa validação de existência do arquivo

Verifica presença da coluna "Pontos" requerida

Trata exceções com mensagens descritivas

4.3 FUNCIONALIDADES IMPLEMENTADAS

4.3.1 CARREGAMENTO INTELIGENTE DE DADOS

A aplicação implementa carregamento robusto de dados com as seguintes características:

Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV

Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada

Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução

Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS

O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:

Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original

Medição de Tempo: Utilização do módulo time para cronometragem precisa

Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória

Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação
Carregamento e validação dos dados

Apresentação de estatísticas preliminares
Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise
4.6 TECNOLOGIAS UTILIZADAS
4.6.1 LINGUAGEM E BIBLIOTECAS
A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.
Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.
A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.
4.6.2 FORMATO DE DADOS
Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS
4.7.1 REUTILIZAÇÃO DE CÓDIGO
O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos
4.7.2 DOCUMENTAÇÃO INTERNA
O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas
Separadores visuais para organização do código

4.8 LIMITAÇÕES E TRABALHOS FUTUROS
Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)
4.9 CONCLUSÃO DA SOLUÇÃO
A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.
Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE
```
# ALGORITMOS

def bubble_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
            comparacoes += 1
            if arr[j] &gt; arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
    arr = arr[:]
```

```
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
        step = 1
        n = len(arr)
    while step &lt; n:
```

```
    for i in range(0, n, step * 2):
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
             if a[j] &lt;= pivot:
             i += 1
                if i != j:
                a[i], a[j] = a[j], a[i]
                trocas += 1
                    if i + 1 != high:
                     a[i+1], a[high] = a[high], a[i+1]
                     trocas += 1
              return i+1
    def qs(a, low, high):
           if low &lt; high:
           pi = partition(a, low, high)
          qs(a, low, pi-1)
          qs(a, pi+1, high)
           qs(arr, 0, len(arr)-1)
           return arr, comparacoes, trocas

#Leitura
```

```
import pandas as pd
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
print("="*70)
```

```
algoritmos = {
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
```

```
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.
htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com
/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.
enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.
geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/
quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em:
https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://
programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/
metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/
index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Sim
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º
Merge Sort	0,1447	0,0281	8.686	10.000	2º

Selection Sort 2,2298 0,0153 499.500 992 3º
Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)

Arquivo 2:

www.dio.me/articles/quick-sort-origem-funcionamento-e-aplicacoes-praticas-artigo-b0757c97fc62 (5276 termos)

Termos comuns: 207

Similaridade

Índice antigo (S): 2,31%

Índice novo (Si): 5,34%

Agrupamento (Sg): Baixo

O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 9f407636o36b0t0

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados
Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO
A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.
Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.
Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.
A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO
3.1 SELECTION SORT
O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do

menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo O(n^2) tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal O(n^2) tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos

para produzir o array completamente ordenado.
Funcionamento do Algoritmo:
O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias

progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.
Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.
3.5 ANÁLISE DE COMPLEXIDADE
3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL
A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE
Bubble Sort: Apresenta complexidade quadrática O(n²) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).
Selection Sort: Mantém complexidade O(n²) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.
Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.
Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n²) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.
3.6 RESULTADOS EXPERIMENTAIS
3.6.1 METODOLOGIA DE TESTES
Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:
Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.
Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.
Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.
3.6.2 AMBIENTE DE EXECUÇÃO
Configuração do Sistema:

Linguagem: Python 3.x
Tamanho do conjunto de dados: 1.000 registros
Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas
3.6.3 RESULTADOS OBTIDOS
Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS
Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :
Observações sobre os dados coletados:
Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.
Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).
Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade O(n^2).
Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).
3.7 ANÁLISE COMPARATIVA
3.7.1 USO DE MEMÓRIA
A análise do consumo de memória revela características distintas entre os algoritmos:
Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.
Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA
Escalabilidade:
Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em

relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando 45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos $O(n^2)$
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso $O(n^2)$, os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo $O(n \log n)$ em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de

responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA

A solução foi organizada em uma arquitetura modular composta por três componentes principais:

4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS

main.py - Módulo Principal

Responsável pela orquestração do fluxo de execução completo

Realiza o carregamento dos dados via pandas

Executa sequencialmente os quatro algoritmos de ordenação

Coleta métricas de desempenho (tempo, memória, comparações, trocas)

Apresenta resultados formatados e análise comparativa

sorts.py - Módulo de Algoritmos

Contém as implementações dos quatro algoritmos de ordenação

Cada função retorna uma tupla: (array_ordenado, comparações, trocas)

Implementa contadores internos para rastreamento de operações

Quick Sort otimizado com seleção de pivô médio para evitar pior caso

leitura.py - Módulo de Leitura de Dados

Fornece função reutilizável para carregamento de arquivos CSV

Implementa validação de existência do arquivo

Verifica presença da coluna "Pontos" requerida

Trata exceções com mensagens descritivas

4.3 FUNCIONALIDADES IMPLEMENTADAS

4.3.1 CARREGAMENTO INTELIGENTE DE DADOS

A aplicação implementa carregamento robusto de dados com as seguintes características:

Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV

Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada

Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução

Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS

O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:

Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original

Medição de Tempo: Utilização do módulo time para cronometragem precisa

Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória

Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação

Carregamento e validação dos dados
Apresentação de estatísticas preliminares
Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise

4.6 TECNOLOGIAS UTILIZADAS

4.6.1 LINGUAGEM E BIBLIOTECAS

A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.

Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.

A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.

4.6.2 FORMATO DE DADOS

Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS

4.7.1 REUTILIZAÇÃO DE CÓDIGO

O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos

4.7.2 DOCUMENTAÇÃO INTERNA

O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas

Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS
Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)
4.9 CONCLUSÃO DA SOLUÇÃO
A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.
Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE

```
# ALGORITMOS

def bubble_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
            comparacoes += 1
            if arr[j] &gt; arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
```

```
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
        step = 1
        n = len(arr)
```

```
    while step &lt; n:
    for i in range(0, n, step * 2):
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
             if a[j] &lt;= pivot:
             i += 1
                 if i != j:
                 a[i], a[j] = a[j], a[i]
                 trocas += 1
                      if i + 1 != high:
                       a[i+1], a[high] = a[high], a[i+1]
                       trocas += 1
              return i+1
    def qs(a, low, high):
           if low &lt; high:
           pi = partition(a, low, high)
          qs(a, low, pi-1)
          qs(a, pi+1, high)
           qs(arr, 0, len(arr)-1)
           return arr, comparacoes, trocas
```

```
#Leitura
import pandas as pd
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
```

```
print("="*70)
algoritmos = {
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
```

```
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em: https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Sim
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º

Merge Sort 0,1447 0,0281 8.686 10.000 2º
Selection Sort 2,2298 0,0153 499.500 992 3º
Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º

==========

Arquivo 1: APS 2o. Relatório ABNT.docx (3874 termos)

Arquivo 2: www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c (5719 termos)

Termos comuns: 196

Similaridade

Índice antigo (S): 2,08%

Índice novo (Si): 5,05%

Agrupamento (Sg): Baixo

O texto abaixo é o conteúdo do documento **Arquivo 1**. Os termos em vermelho foram encontrados no documento **Arquivo 2**. Id: 7635b59ao53b19t47

==========

FACULDADE VANGUARDA

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS
Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados

São José dos Campos
2025

Luiz Gustavo Francisco de Souza

ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

Desenvolvimento de Aplicação de Ordenação e
Classificação de Dados
Pontuações de Jogadores

Atividades Práticas Supervisionadas do curso de ENGENHARIA DA COMPUTAÇÃO da FACULDADE VANGUARDA, sob orientação de:
Prof. MSc. Fernando Mauro de Souza ? Prof. Responsável
Prof. MSc. André Yoshimi Kusumoto ? Coordenador

São José dos Campos
2025

INTRODUÇÃO

A ordenação e classificação de dados representam fundamentos essenciais na Engenharia da Computação, permeando desde aplicações cotidianas até sistemas complexos de larga escala. Em um contexto onde o volume de dados cresce exponencialmente, a capacidade de organizar informações de forma eficiente torna-se não apenas essencial, mas imprescindível para o desenvolvimento de soluções computacionais robustas e performáticas.

Este trabalho, desenvolvido no âmbito da disciplina de Estrutura de Dados e Programação, tem como objetivo explorar e analisar diferentes algoritmos de ordenação aplicados a diferentes conjuntos de dados, no meu caso este conjunto é Pontuações de jogadores em um campeonato global. Uma boa escolha de algoritmo de ordenação adequado pode significar a diferença entre um sistema eficiente e um que apresenta queda de performance em cenários reais de uso.

Neste relatório, irei apresentar quatro algoritmos clássicos de ordenação: Bubble Sort, Selection Sort, Merge Sort e Quick Sort. Cada um desses algoritmos será analisado sob diferentes perspectivas, incluindo sua complexidade temporal e espacial, comportamento em diferentes cenários de dados e aplicabilidade prática ao conjunto de dados trabalhado.

A metodologia adotada combina fundamentação teórica com experimentação prática, implementando cada algoritmo na linguagem Python e coletando métricas de desempenho que possibilitam uma análise comparativa objetiva. Os resultados experimentais obtidos permitirão identificar qual algoritmo apresenta melhor adequação ao contexto específico do problema proposto, considerando fatores como tamanho do conjunto de dados, distribuição dos valores e recursos computacionais disponíveis.

ALGORITMOS DE ORDENAÇÃO E CLASSIFICAÇÃO

3.1 SELECTION SORT

O Selection Sort é um algoritmo de ordenação por comparação que opera através da seleção iterativa do

menor (ou maior) elemento da porção não ordenada do array, realizando sua permuta com o primeiro elemento não ordenado. Este processo é repetido sistematicamente até que todo o conjunto de dados esteja completamente ordenado.
Funcionamento do Algoritmo:
O algoritmo inicia sua execução localizando o menor elemento do array e permutando-o com o elemento na primeira posição, garantindo que o menor valor ocupe sua posição definitiva. Subsequentemente, o processo é replicado para os elementos remanescentes, identificando o segundo menor elemento e posicionando-o na segunda posição do array.

Este procedimento continua de forma iterativa, processando os elementos restantes até que todos estejam dispostos em ordem crescente. A cada iteração, um elemento adicional é colocado em sua posição final, reduzindo progressivamente o tamanho da porção não ordenada do array.
A principal característica deste algoritmo é sua previsibilidade: independentemente da disposição inicial dos dados, ele sempre realizará o mesmo número de comparações, apresentando complexidade de tempo O(n^2) tanto no melhor quanto no pior caso.
3.2 BUBBLE SORT
O Bubble Sort é considerado o algoritmo de ordenação mais elementar e intuitivo, operando através da comparação e permuta repetida de elementos adjacentes que estejam em ordem incorreta. Apesar de sua simplicidade conceitual e facilidade de implementação, este algoritmo não é recomendado para grandes conjuntos de dados devido à sua elevada complexidade temporal O(n^2) tanto no caso médio quanto no pior caso.
Funcionamento do Algoritmo:
A ordenação é executada através de múltiplas varreduras (passagens) pelo array. Na primeira varredura, o maior elemento é deslocado para a última posição, alcançando sua posição definitiva. Na segunda varredura, o segundo maior elemento é movido para a penúltima posição, e assim sucessivamente.

Em cada varredura, o algoritmo processa exclusivamente os elementos que ainda não foram posicionados corretamente. Após k varreduras completas, os k maiores elementos já terão sido movidos para as últimas k posições do array, encontrando-se em suas posições finais.
Durante cada varredura, são comparados todos os pares de elementos adjacentes na porção não ordenada. Quando um elemento de maior valor precede um elemento de menor valor, suas posições são permutadas. Através deste processo característico de "flutuação" ou "borbulhamento" dos valores maiores em direção ao final do array, o maior elemento entre os não ordenados alcança sua posição definitiva ao final de cada passagem.
Uma otimização comum consiste em interromper o algoritmo quando nenhuma troca é realizada em uma varredura completa, indicando que o array já está ordenado.
3.3 MERGE SORT
O Merge Sort é um algoritmo de ordenação amplamente reconhecido por sua eficiência consistente e estabilidade, fundamentado no paradigma algorítmico de Dividir e Conquistar. Seu funcionamento baseia-se na divisão recursiva do array de entrada em subarrays progressivamente menores, na ordenação individual destes subarrays e, finalmente, na combinação (merge) ordenada dos mesmos

para produzir o array completamente ordenado.
Funcionamento do Algoritmo:
O processo de ordenação do Merge Sort pode ser decomposto em três fases distintas:Divisão (Divide): O array é recursivamente dividido em duas metades aproximadamente iguais até que cada subarray contenha apenas um elemento único. Um subarray unitário é considerado trivialmente ordenado por definição, constituindo o caso base da recursão.
Conquista (Ordena): Cada subarray é ordenado individualmente através da aplicação recursiva do próprio algoritmo Merge Sort, subdividindo o problema em instâncias progressivamente menores até alcançar os casos base.
Combinação (Merge): Os subarrays ordenados são mesclados de forma ordenada através de um processo de intercalação. Este procedimento compara os elementos iniciais de cada subarray, selecionando o menor para compor o array resultante, mantendo a propriedade de ordenação. O processo de combinação continua recursivamente, intercalando os subarrays até que todos tenham sido unidos, resultando no array completamente ordenado.
O Merge Sort apresenta complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), garantindo desempenho previsível e eficiente mesmo para grandes volumes de dados. No entanto, requer espaço auxiliar O(n) para armazenar os subarrays temporários durante o processo de intercalação.
3.4 Quick Sort
O Quick Sort é um algoritmo de ordenação altamente eficiente fundamentado no paradigma de Dividir e Conquistar, caracterizado pela seleção estratégica de um elemento denominado pivô e pelo subsequente particionamento do array em torno deste elemento, posicionando-o em sua localização definitiva na sequência ordenada.

Funcionamento do Algoritmo:
O algoritmo opera através de quatro etapas principais que se repetem recursivamente:
Escolha do Pivô: Um elemento do array é selecionado como pivô, servindo como referência para o particionamento. A estratégia de seleção pode variar significativamente, incluindo opções como o primeiro elemento, o último elemento, um elemento aleatório, o elemento central ou a mediana de três valores. A escolha da estratégia de seleção do pivô pode impactar significativamente o desempenho do algoritmo.
Particionamento: O array é reorganizado através de um processo de particionamento, de forma que todos os elementos com valores menores que o pivô sejam posicionados à sua esquerda, enquanto todos os elementos com valores maiores sejam colocados à sua direita. Elementos iguais ao pivô podem ser posicionados em qualquer um dos lados, dependendo da implementação. Ao final desta etapa, o pivô encontra-se em sua posição definitiva no array ordenado, e seu índice é retornado para orientar as chamadas recursivas subsequentes.
Chamadas Recursivas: O algoritmo é aplicado recursivamente aos dois subarrays resultantes do particionamento (elementos à esquerda e à direita do pivô), subdividindo o problema em instâncias

progressivamente menores. Este processo recursivo continua até que os subarrays alcancem tamanho mínimo.
Caso Base: A recursão é interrompida quando um subarray contém zero ou um elemento, visto que um array vazio ou unitário já se encontra ordenado por definição, não necessitando de processamento adicional.
3.5 ANÁLISE DE COMPLEXIDADE
3.5.1 COMPLEXIDADE TEMPORAL E ESPACIAL
A análise de complexidade algorítmica permite compreender o comportamento dos algoritmos de ordenação em diferentes cenários, fornecendo métricas teóricas que orientam a escolha da solução mais adequada para cada contexto específico.

Tabela 1 - Complexidade Temporal e Espacial dos Algoritmos

3.5.2 ANÁLISE COMPARATIVA DE COMPLEXIDADE
Bubble Sort: Apresenta complexidade quadrática O(n^2) tanto no caso médio quanto no pior caso, realizando n-1 passagens pelo array com comparações adjacentes. O melhor caso O(n) ocorre quando o array já está ordenado e uma otimização com flag é implementada. Opera in-place com complexidade de espaço O(1).
Selection Sort: Mantém complexidade O(n^2) em todos os cenários, pois sempre realiza o mesmo número de comparações independentemente da disposição inicial dos dados. Realiza apenas n-1 trocas no total, sendo mais eficiente que o Bubble Sort em termos de movimentações de elementos.
Merge Sort: Garante complexidade O(n log n) em todos os casos devido à sua natureza recursiva de divisão e conquista. Requer espaço auxiliar O(n) para armazenar os subarrays durante o processo de intercalação, sendo sua principal limitação.
Quick Sort: Apresenta complexidade O(n log n) no caso médio, mas pode degradar para O(n^2) no pior caso quando o pivô escolhido é consistentemente o menor ou maior elemento. Utiliza espaço auxiliar O(log n) para a pilha de recursão.
3.6 RESULTADOS EXPERIMENTAIS
3.6.1 METODOLOGIA DE TESTES
Os experimentos foram realizados com um conjunto de dados contendo 1.000 pontuações de jogadores, representando um cenário típico de sistemas de ranking em campeonatos. Para garantir a robustez estatística dos resultados, foram executadas 10 rodadas de testes para cada algoritmo em três cenários distintos:
Cenário Aleatório: Pontuações distribuídas aleatoriamente, simulando entrada de dados não ordenada típica de sistemas reais.
Cenário Ordenado: Pontuações já ordenadas em ordem crescente, representando o melhor caso para a maioria dos algoritmos.
Cenário Pior Caso: Pontuações ordenadas em ordem decrescente, representando o pior caso operacional.
3.6.2 AMBIENTE DE EXECUÇÃO
Configuração do Sistema:

Linguagem: Python 3.x
Tamanho do conjunto de dados: 1.000 registros
Métricas coletadas: Tempo de execução (segundos), uso de memória (MB), número de comparações e número de trocas
3.6.3 RESULTADOS OBTIDOS
Tabela 2 - Médias de Desempenho - Cenário Aleatório (10 rodadas com 1.000 registros)

Tabela 3 - Médias de Desempenho - Cenário Ordenado (10 rodadas com 1.000 registros)

Tabela 4 - Médias de Desempenho - Cenário Pior Caso (10 rodadas com 1.000 registros)

3.6.4 ANÁLISE GRÁFICA DOS RESULTADOS
Os gráficos a seguir ilustram comparativamente o desempenho dos algoritmos nos três cenários testados :
Observações sobre os dados coletados:
Tempo de Execução: O Quick Sort apresentou consistentemente os menores tempos de execução em todos os cenários, seguido pelo Merge Sort. Os algoritmos quadráticos (Bubble Sort e Selection Sort) demonstraram tempos significativamente superiores.
Uso de Memória: O Merge Sort apresentou maior consumo de memória (0,0281 MB) devido à necessidade de arrays auxiliares, enquanto os demais algoritmos operaram com consumo mínimo (0,0153-0,0154 MB).
Número de Comparações: O Merge Sort realizou consistentemente menos comparações (4.932-8.686), enquanto Bubble Sort e Selection Sort executaram 499.500 comparações em todos os cenários, confirmando sua complexidade O(n^2).
Número de Trocas: O Selection Sort minimizou o número de trocas (0-992), seguido pelo Quick Sort. O Bubble Sort apresentou o maior número de trocas no pior caso (499.329).
3.7 ANÁLISE COMPARATIVA
3.7.1 USO DE MEMÓRIA
A análise do consumo de memória revela características distintas entre os algoritmos:
Algoritmos In-Place (Bubble Sort, Selection Sort, Quick Sort): Operam com complexidade espacial O(1) ou O(log n), consumindo entre 0,0153-0,0154 MB. Estes algoritmos realizam a ordenação diretamente no array original, sendo ideais para sistemas com restrições de memória.
Merge Sort: Requer espaço auxiliar O(n), consumindo 0,0281 MB (aproximadamente 83% mais memória que os algoritmos in-place). Esta sobrecarga é necessária para armazenar os subarrays temporários durante o processo de intercalação. Para o contexto de 1.000 pontuações de jogadores, este overhead é aceitável, mas pode tornar-se problemático em sistemas com milhões de registros.

3.7.2 COMPORTAMENTO POR TAMANHO DE ENTRADA
Escalabilidade:
Quick Sort: Demonstrou excelente escalabilidade, mantendo desempenho superior em todos os cenários . No cenário aleatório, processou 1.000 registros em média de 0,1029s, sendo 36,97 vezes mais rápido que o Bubble Sort.
Merge Sort: Apresentou desempenho consistente e previsível em todos os cenários, com variação mínima entre melhor (0,0738s) e pior caso (0,0975s), característica valiosa para sistemas que exigem garantias de tempo de resposta.
Selection Sort: Manteve desempenho constante independentemente da distribuição inicial dos dados, sempre realizando 499.500 comparações. Tempo médio de 1,4312s (ordenado) a 2,2298s (aleatório).
Bubble Sort: Apresentou a maior variação de desempenho entre cenários: 1,7484s (ordenado) a 4,5242s (pior caso), demonstrando sensibilidade à distribuição inicial dos dados.
3.7.3 ESTABILIDADE E CARACTERÍSTICAS DE ORDENAÇÃO
Estabilidade: Refere-se à capacidade do algoritmo de manter a ordem relativa de elementos com chaves iguais.
Algoritmos Estáveis (Bubble Sort e Merge Sort): Preservam a ordem original de pontuações idênticas, característica essencial para sistemas de ranking onde critérios secundários (como data de registro) devem ser mantidos.
Algoritmos Não-Estáveis (Selection Sort e Quick Sort): Podem alterar a ordem relativa de elementos com valores iguais. Para o contexto de pontuações de jogadores, se dois jogadores possuem a mesma pontuação, a ordem entre eles pode ser alterada.
Adaptabilidade:
Quick Sort: Apresentou excelente adaptabilidade, com tempo reduzido em 55,9% no cenário ordenado (0,0454s) comparado ao aleatório (0,1029s).
Merge Sort: Demonstrou baixa adaptabilidade, com variação de apenas 24,2% entre melhor e pior caso, característica esperada devido à sua complexidade consistente O(n log n).
Bubble Sort e Selection Sort: O Bubble Sort mostrou alguma adaptabilidade (tempo 54% menor no cenário ordenado), enquanto o Selection Sort manteve comportamento uniforme.
3.8 ALGORITMO MAIS ADEQUADO
3.8.1 RECOMENDAÇÃO
Para o contexto específico de ordenação de pontuações de jogadores em campeonatos, recomenda-se a utilização do Quick Sort como algoritmo primário de ordenação.
3.8.2 JUSTIFICATIVA
A recomendação do Quick Sort fundamenta-se nos seguintes aspectos técnicos evidenciados pelos resultados experimentais:
1. Desempenho Superior Consistente: O Quick Sort apresentou os menores tempos de execução em todos os cenários testados:
Cenário Aleatório: 0,1029s (média)
Cenário Ordenado: 0,0454s (melhor desempenho)
Cenário Pior Caso: 0,0766s (ainda superior aos concorrentes)
Este desempenho representa uma melhoria de 36,97 vezes em relação ao Bubble Sort e 21,66 vezes em

relação ao Selection Sort no cenário aleatório, que é o mais representativo de situações reais.
2. Eficiência de Memória: Com consumo de apenas 0,0154 MB, o Quick Sort opera in-place, utilizando 45,2% menos memória que o Merge Sort (0,0281 MB). Para sistemas de ranking que podem processar milhares ou milhões de jogadores, esta economia é significativa.

3. Número Otimizado de Operações:
Comparações: 10.956 (cenário aleatório) - aproximadamente 45,6 vezes menos que os algoritmos $O(n^2)$
Trocas: 4.902 (cenário aleatório) - significativamente inferior ao Bubble Sort (245.957)
4. Complexidade Prática vs. Teórica: Embora o Quick Sort possua complexidade de pior caso $O(n^2)$, os resultados experimentais demonstraram que, mesmo no cenário de pior caso (dados invertidos), o algoritmo manteve desempenho superior (0,0766s), 59 vezes mais rápido que o Bubble Sort (4,5242s) no mesmo cenário.
5. Adequação ao Contexto de Aplicação: Para sistemas de ranking de jogadores:
Atualizações Frequentes: O Quick Sort processa rapidamente novos conjuntos de pontuações após cada partida
Escalabilidade: Mantém eficiência mesmo com aumento significativo do número de jogadores
Recursos Limitados: Opera eficientemente sem overhead significativo de memória
Considerações sobre o Merge Sort: O Merge Sort seria uma alternativa viável quando:
A estabilidade da ordenação for requisito obrigatório (manter ordem de empates por critério secundário)
Houver necessidade de garantias absolutas de tempo $O(n \log n)$ em todos os cenários
O overhead de memória de 83% for aceitável para o sistema
Limitações dos Algoritmos Quadráticos: Os resultados confirmam que Bubble Sort e Selection Sort não são adequados para aplicações de produção com conjuntos de dados superiores a algumas centenas de elementos, apresentando degradação de desempenho inaceitável para sistemas modernos de ranking.
Conclusão: A análise quantitativa dos dados experimentais, combinada com os requisitos específicos de sistemas de pontuação de jogadores (rapidez, eficiência de memória e escalabilidade), estabelece o Quick Sort como a solução mais adequada para o contexto proposto. Sua implementação proporcionará resposta rápida aos usuários, consumo otimizado de recursos e capacidade de escalar para campeonatos com milhares de participantes. Entretanto, em ambientes de produção global, onde os dados passam por pipelines, particionamento (sharding) e processamento externo, o Quick Sort pode não ser a única escolha ideal, requisitos como estabilidade, entrada em disco, resistência a padrões adversariais ou previsibilidade de latência podem demandar alternativas como Timsort, Introsort ou abordagens distribuídas.

SOLUÇÃO DESENVOLVIDA
4.1 VISÃO GERAL DA APLICAÇÃO
A solução desenvolvida consiste em uma aplicação Python voltada para análise comparativa de algoritmos de ordenação aplicados a pontuações de jogadores em campeonatos. O sistema foi projetado com foco em desempenho, modularidade e facilidade de análise, permitindo a execução automatizada de testes e a coleta sistemática de métricas de desempenho.
A aplicação foi estruturada seguindo princípios de engenharia de software, com separação clara de

responsabilidades através de módulos independentes, facilitando manutenção, teste e extensibilidade do código.

4.2 ARQUITETURA DO SISTEMA

A solução foi organizada em uma arquitetura modular composta por três componentes principais:

4.2.1 ESTRUTURA DE ARQUIVOS

4.2.2 DESCRIÇÃO DOS MÓDULOS

main.py - Módulo Principal

Responsável pela orquestração do fluxo de execução completo

Realiza o carregamento dos dados via pandas

Executa sequencialmente os quatro algoritmos de ordenação

Coleta métricas de desempenho (tempo, memória, comparações, trocas)

Apresenta resultados formatados e análise comparativa

sorts.py - Módulo de Algoritmos

Contém as implementações dos quatro algoritmos de ordenação

Cada função retorna uma tupla: (array_ordenado, comparações, trocas)

Implementa contadores internos para rastreamento de operações

Quick Sort otimizado com seleção de pivô médio para evitar pior caso

leitura.py - Módulo de Leitura de Dados

Fornece função reutilizável para carregamento de arquivos CSV

Implementa validação de existência do arquivo

Verifica presença da coluna "Pontos" requerida

Trata exceções com mensagens descritivas

4.3 FUNCIONALIDADES IMPLEMENTADAS

4.3.1 CARREGAMENTO INTELIGENTE DE DADOS

A aplicação implementa carregamento robusto de dados com as seguintes características:

Leitura via Pandas: Utilização da biblioteca pandas para processamento eficiente de arquivos CSV

Validação de Integridade: Verificação automática da existência do arquivo e estrutura esperada

Análise Preliminar: Exibição de estatísticas básicas (mínimo, máximo, média) antes da execução

Conversão para Lista: Transformação dos dados em estrutura Python nativa para processamento

4.3.2 EXECUÇÃO AUTOMATIZADA DE ALGORITMOS

O sistema executa automaticamente os quatro algoritmos de ordenação configurados, realizando:

Isolamento de Dados: Cada algoritmo recebe uma cópia independente do array original

Medição de Tempo: Utilização do módulo time para cronometragem precisa

Rastreamento de Memória: Emprego do módulo tracemalloc para medição de consumo de memória

Contagem de Operações: Registro automático de comparações e trocas durante a execução

4.3.3 COLETA DE MÉTRICAS DE DESEMPENHO
Para cada algoritmo executado, a aplicação coleta e armazena dados em uma tabela.
4.3.4 APRESENTAÇÃO DE RESULTADOS
A aplicação gera uma saída formatada e estruturada contendo:
Tabela Comparativa:
Ranking de Desempenho:
Classificação ordenada por tempo de execução
Identificação automática do algoritmo mais eficiente
Cálculo de speedup (ganho de velocidade relativo ao Bubble Sort)

Análise Comparativa:
4.4 DETALHES DE IMPLEMENTAÇÃO
4.4.1 OTIMIZAÇÕES IMPLEMENTADAS
Quick Sort com Pivô Médio: Uma otimização crítica foi implementada no algoritmo Quick Sort para evitar degradação de desempenho em dados ordenados:
Esta modificação garante que mesmo em cenários de dados já ordenados ou inversamente ordenados, o Quick Sort mantenha complexidade próxima a O(n log n).
Preservação de Dados Originais: Todos os algoritmos recebem cópias independentes do array original (arr[:]), garantindo que:
O conjunto de dados original permanece intacto
Cada algoritmo opera sobre dados idênticos
Comparações são justas e reproduzíveis
4.4.2 CONTADORES DE OPERAÇÕES
Cada algoritmo implementa contadores internos para rastreamento preciso de operações:
Estes contadores permitem análise detalhada além do tempo de execução, revelando o comportamento algorítmico interno.
4.5 INTERFACE E EXPERIÊNCIA DO USUÁRIO
4.5.1 INTERFACE DE LINHA DE COMANDO (CLI)
A aplicação utiliza interface de linha de comando (CLI) com formatação visual para melhor legibilidade:
Características da Interface:
Separadores visuais com linhas de 70 caracteres
Títulos centralizados para cada seção
Formatação de números com separadores de milhares
Precisão de 6 casas decimais para medições de tempo
Feedback em tempo real durante execução
4.5.2 FLUXO DE EXECUÇÃO
Inicialização:
Exibição do título da aplicação

Carregamento e validação dos dados
Apresentação de estatísticas preliminares
Processamento:
Execução sequencial dos algoritmos
Feedback visual de progresso para cada algoritmo
Confirmação de conclusão com tempo decorrido
Finalização:
Exibição da tabela comparativa completa
Apresentação do ranking de desempenho
Cálculo e exibição de speedups
Mensagem de conclusão da análise

4.6 TECNOLOGIAS UTILIZADAS

4.6.1 LINGUAGEM E BIBLIOTECAS

A aplicação foi desenvolvida em Python 3.x, escolhida por sua sintaxe clara e ampla disponibilidade de bibliotecas para análise de dados. A biblioteca Pandas foi utilizada para leitura e manipulação eficiente de arquivos CSV, permitindo carregamento rápido e conversão dos dados para estruturas nativas do Python.

Para medição de desempenho, foram empregados os módulos nativos time e tracemalloc, responsáveis respectivamente pela cronometragem precisa do tempo de execução e pelo rastreamento do consumo de memória durante a ordenação. O módulo pathlib, também nativo, foi utilizado para manipulação de caminhos de arquivos e validação de existência de recursos.

A escolha de bibliotecas predominantemente nativas reduz dependências externas, simplifica a instalação e garante maior compatibilidade entre diferentes ambientes de execução.

4.6.2 FORMATO DE DADOS

Arquivo CSV de Entrada:
Formato: CSV (Comma-Separated Values)
Estrutura: 1.000 registros de jogadores
Colunas: ID, Username, País, Pontos, Vitórias, Derrotas
Coluna utilizada: "Pontos" (valores numéricos inteiros)
Faixa de valores: 1.001 a 3.999 pontos

4.7 RECURSOS ADICIONAIS IMPLEMENTADOS

4.7.1 REUTILIZAÇÃO DE CÓDIGO

O módulo leitura.py implementa função genérica reutilizável:
Aceita caminho de arquivo como parâmetro
Retorna lista de valores prontos para processamento
Pode ser importada em outros projetos

4.7.2 DOCUMENTAÇÃO INTERNA

O código inclui:
Docstrings em funções principais
Comentários explicativos em seções críticas

Separadores visuais para organização do código
4.8 LIMITAÇÕES E TRABALHOS FUTUROS
Interface: Limitada a linha de comando, sem interface gráfica
Persistência: Resultados não são salvos automaticamente
Visualização: Ausência de gráficos gerados automaticamente
Entrada: Caminho do CSV fixo no código (requer alteração manual)
4.9 CONCLUSÃO DA SOLUÇÃO
A solução desenvolvida atende integralmente aos requisitos estabelecidos na APS, implementando com sucesso os quatro algoritmos de ordenação solicitados e fornecendo análise comparativa detalhada baseada em métricas objetivas. A arquitetura modular facilita manutenção e extensões futuras, enquanto a interface CLI oferece feedback claro e organizado sobre o desempenho de cada algoritmo.
Os resultados experimentais confirmam as previsões teóricas de complexidade, validando a implementação e demonstrando o impacto prático da escolha adequada de algoritmos de ordenação em contextos reais de desenvolvimento de software.

CÓDIGO-FONTE
ALGORITMOS

```
def bubble_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
            comparacoes += 1
            if arr[j] &gt; arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocas += 1
return arr, comparacoes, trocas

def selection_sort(arr):
```

```
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            comparacoes += 1
            if arr[j] &lt; arr[min_index]:
                min_index = j
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]
        trocas += 1
    return arr, comparacoes, trocas
def merge_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def merge(left, right):
        nonlocal comparacoes, trocas
        res = []
        i = j = 0
        while i &lt; len(left) and j &lt; len(right):
            comparacoes += 1
            if left[i] &lt; right[j]:
                res.append(left[i])
                i += 1
            else:
                res.append(right[j])
                j += 1
            trocas += 1
        while i &lt; len(left):
            res.append(left[i])
            i += 1
        trocas += 1
        while j &lt; len(right):
            res.append(right[j])
            j += 1
            trocas += 1
        return res
    step = 1
    n = len(arr)
```

```
    while step &lt; n:
    for i in range(0, n, step * 2):
        left = arr[i:i+step]
    right = arr[i+step:i+2*step]
merged = merge(left, right)
for j, val in enumerate(merged):
arr[i+j] = val
step *= 2
return arr, comparacoes, trocas

"""precisei fazer um ajuste no pivot, agora usa
pivot do meio (evita pior caso com dados ordenados)"""
def quick_sort(arr):
    arr = arr[:]
    comparacoes = 0
    trocas = 0
    def partition(a, low, high):
        nonlocal comparacoes, trocas
        mid = (low + high) // 2
       a[mid], a[high] = a[high], a[mid]
       pivot = a[high]
       i = low ? 1
          for j in range(low, high):
          comparacoes += 1
            if a[j] &lt;= pivot:
            i += 1
                if i != j:
                a[i], a[j] = a[j], a[i]
                trocas += 1
                    if i + 1 != high:
                     a[i+1], a[high] = a[high], a[i+1]
                     trocas += 1
              return i+1
    def qs(a, low, high):
          if low &lt; high:
          pi = partition(a, low, high)
         qs(a, low, pi-1)
         qs(a, pi+1, high)
          qs(arr, 0, len(arr)-1)
          return arr, comparacoes, trocas
```

```
#Leitura
import pandas as pd
from pathlib import Path
def carregar_pontos(caminho_arquivo: str):
"""Lê um arquivo CSV e devolve a lista de pontos (coluna 'Pontos').
caminho_arquivo: caminho para o arquivo CSV.
retorna: lista de inteiros ou floats com os pontos.
"""
caminho = Path(caminho_arquivo)
if not caminho.exists():
raise FileNotFoundError(f"Arquivo não encontrado: {caminho_arquivo}")
dados = pd.read_csv(caminho)
if "Pontos" not in dados.columns:
raise ValueError("A coluna 'Pontos' não existe no arquivo CSV.")
pontos = dados["Pontos"].tolist()
return pontos



"""
APS - ANÁLISE DE ALGORITMOS DE ORDENAÇÃO
Versão simples: carrega dados ? executa ? mostra resultados
"""
import pandas as pd
import time
import tracemalloc
from algoritmos import sorts
#CARREGAR DADOS
print("="*70)
print("ANÁLISE DE ALGORITMOS DE ORDENAÇÃO".center(70))
print("="*70)
# Caminho do CSV
CSV_PATH = "projeto_ordenacao\data\Jogadores.csv"
print(f"\nCarregando: {CSV_PATH}")
dados = pd.read_csv(CSV_PATH)
pontos = dados["Pontos"].tolist()
print(f"{len(pontos)} jogadores carregados")
print(f" Menor: {min(pontos)} | Maior: {max(pontos)} | Média: {sum(pontos)/len(pontos):.0f}")
# EXECUTA ALGORITMOS
print("\n" + "="*70)
print("EXECUTANDO ANÁLISE".center(70))
```

```
print("="*70)
algoritmos = {
"Bubble Sort": sorts.bubble_sort,
"Selection Sort": sorts.selection_sort,
"Merge Sort": sorts.merge_sort,
"Quick Sort": sorts.quick_sort
}
resultados = {}
for nome, funcao in algoritmos.items():
print(f"\nExecutando {nome}...")
# Medir memória
tracemalloc.start()
# Medir tempo
inicio = time.time()
resultado, comp, troc = funcao(pontos[:])
fim = time.time()
# Capturar memória
_, mem_pico = tracemalloc.get_traced_memory()
tracemalloc.stop()
# Salvar
resultados[nome] = {
'tempo': fim - inicio,
'memoria': mem_pico / (1024 * 1024),
'comparacoes': comp,
'trocas': troc
}
print(f"Concluído em {resultados[nome]['tempo']:.6f}s")
#MOSTRA RESULTADOS
print("\n" + "="*70)
print("RESULTADOS".center(70))
print("="*70)
print(f"\n{'Algoritmo':&lt;18} {'Tempo (s)':&lt;12} {'Memória (MB)':&lt;14} {'Comparações':&lt;15} {'Trocas'}")
print("-"*70)
for nome, dados in resultados.items():
print(f"{nome:&lt;18} {dados['tempo']:&lt;12.6f} {dados['memoria']:&lt;14.4f} "
f"{dados['comparacoes']:&lt;15,} {dados['trocas']:,}")
print("-"*70)
#RANKING
print("\nRANKING POR VELOCIDADE:")
ranking = sorted(resultados.items(), key=lambda x: x[1]['tempo'])
for i, (nome, dados) in enumerate(ranking, 1):
```

```
print(f" {i}º - {nome}: {dados['tempo']:.6f}s")
print(f"\nMELHOR ALGORITMO: {ranking[0][0]}")
print(f" ({ranking[0][1]['tempo']:.6f} segundos)")
# Speedup
print(f"\nCOMPARAÇÃO DE VELOCIDADE:")
tempo_base = resultados['Bubble Sort']['tempo']
for nome in ['Selection Sort', 'Merge Sort', 'Quick Sort']:
velocidade = tempo_base / resultados[nome]['tempo']
print(f" {nome} é {velocidade:.2f}x mais rápido que Bubble Sort")
print("\n" + "="*70)
print("ANÁLISE CONCLUÍDA".center(70))
print("="*70)
```

REFERÊNCIAS

BLOG CYBERINI. Bubble Sort. Disponível em:https://www.blogcyberini.com/2018/02/bubble-sort.htmlSISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ Acesso em: 18 out. 2025.

ENJOYALGORITHMS. Comparison of Sorting Algorithms. Disponível em: https://www.enjoyalgorithms.com/blog/comparison-of-sorting-algorithms/ . Acesso em: 25 out. 2025.

GEEKSFORGEEKS. Time and Space Complexity Analysis of Bubble Sort. Disponível em: https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/ . Acesso em: 13 out. 2025.

GEEKSFORGEEKS. Quick Sort vs Merge Sort. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/ . Acesso em: 25 out. 2025.

NIKOO28. Selection Sort ? Explanation with illustration. Study Algorithms, 3 jan. 2014. Disponível em: https://studyalgorithms.com/array/selection-sort/ . Acesso em: 10 out. 2025.

PROGRAMIZ PRO. Comparing Quick Sort, Merge Sort, and Insertion Sort. Disponível em: https://programiz.pro/resources/dsa-merge-quick-insertion-comparision/ . Acesso em: 25 out. 2025.

SISTEVOLUTION. Método MergeSort. Disponível em: https://systevolution.wordpress.com/2011/10/26/metodo-mergesort/ . Acesso em: 2 out. 2025.

W3SCHOOLS. Data Structures and Algorithms (DSA). Disponível em: https://www.w3schools.com/dsa/index.php

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Sim
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	Não

Algoritmo	Tempo Médio (s)	Memória (MB)	Comparações	Trocas	Ranking
Quick Sort	0,1029	0,0154	10.956	4.902	1º

Merge Sort 0,1447 0,0281 8.686 10.000 2º
Selection Sort 2,2298 0,0153 499.500 992 3º
Bubble Sort 3,8051 0,0153 499.500 245.957 4º

Algoritmo Tempo Médio(s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0454 0,0154 8.046 565 1º
Merge Sort 0,0738 0,0281 5.132 10.000 2º
Selection Sort 1,4312 0,0153 499.500 0 3º
Bubble Sort 1,7484 0,0153 499.500 0 4º

Algoritmo Tempo Médio (s) Memória (MB) Comparações Trocas Ranking
Quick Sort 0,0766 0,0154 8.466 4.047 1º
Merge Sort 0,0975 0,0281 4.932 10.000 2º
Selection Sort 1,7824 0,0153 499.500 552 3º
Bubble Sort 4,5242 0,0153 499.500 499.329 4º