INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO
CAMPUS BRAGANÇA PAULISTA

PAULA ANDREZZA GOMES MARINHO

ANÁLISE DE ALGORITMOS

BRAGANÇA PAULITSA
2023

SUMÁRIO

1. INTRODUÇÃO

A avaliação e análise de algoritmos de ordenação desempenham um papel fundamental na compreensão e otimização da eficiência dos algoritmos. Neste trabalho, o objetivo é avaliar o desempenho de diferentes algoritmos de ordenação, especificamente o Algoritmo da Bolha, Inserção e Seleção, estudados em sala de aula. Vamos conduzir essa análise em três cenários distintos, representados por três tamanhos de vetores: pequeno (1000 elementos), médio (10.000 elementos) e grande (100.000 elementos).

É importante destacar que, para garantir uma avaliação abrangente, evitaremos o uso de vetores já ordenados (melhor caso) e, em vez disso, trabalharemos com vetores parcialmente ordenados e invertidos (pior caso). Além disso, monitoraremos o desempenho dos algoritmos por meio da contagem de operações primitivas executadas dentro dos laços de repetição, proporcionando uma visão detalhada das complexidades computacionais envolvidas.

A metodologia consiste em aplicar cada algoritmo de ordenação em todos os tamanhos de vetores mencionados anteriormente. Registraremos o número de operações realizadas por cada algoritmo para cada cenário, permitindo-nos observar e analisar o comportamento deles em diferentes situações. Em seguida, discutiremos a função de crescimento dos algoritmos, identificando padrões e comparando seus desempenhos.

Para apresentar nossos resultados de maneira clara e informativa, criaremos gráficos matemáticos que representarão o trabalho realizado por cada algoritmo em cada cenário. Isso nos ajudará a visualizar as diferenças de desempenho de forma eficaz.

2. METODOLOGIA

- Seleção dos Algoritmos: Escolhemos avaliar os algoritmos de ordenação Bolha, Inserção e Seleção.

- Tamanhos de Vetores: pequeno (1000 elementos), médio (10.000 elementos) e grande (100.000 elementos).

- Características dos Vetores: Utilizaremos vetores parcialmente ordenados e invertidos (pior caso) para testar o desempenho dos algoritmos.

- Contagem de Operações: Registraremos o número de operações primitivas (comparações, trocas) executadas por cada algoritmo em cada cenário.

- Análise de Desempenho: Compararemos os resultados dos algoritmos em diferentes tamanhos de vetor e analisaremos a função de crescimento, identificando padrões à medida que o tamanho do vetor aumenta.

Dividiremos a metodologia em seções individuais para cada algoritmo de ordenação, aplicando os passos acima para cada um. Isso nos permitirá avaliar e comparar de forma abrangente o desempenho dos algoritmos em diversos cenários.

2.1. INFORMAÇÕES IMPORTANTES

O código foi dividido em partes para facilitar a compreensão de cada algoritmo. Para manter a consistência dos dados, um vetor aleatório é gerado apenas uma vez, com o maior tamanho de teste, que consiste em um vetor com 100.000 elementos. Sempre que é necessário realizar testes com tamanhos menores, uma cópia parcial desse vetor é criada usando a função **copyArray**. A quantidade de elementos copiados é determinada de acordo com o tamanho necessário para o teste em questão. Dessa forma, garantimos a reutilização dos mesmos valores aleatórios em diferentes tamanhos de teste.

Segue abaixo a estrutura hierárquica utilizada para organizar o código:

Cada arquivo desempenha um papel específico na aplicação e contém funções dedicadas que serão detalhadas a seguir:

1. **main.c**: Este arquivo contém a função principal do programa. Neste arquivo, são definidos e inicializados os arrays, incluindo um array de números aleatórios e um array invertido. Além disso, é responsável por chamar a função `runTests` do arquivo `tests.c`, que realiza os testes e verifica o desempenho dos algoritmos de ordenação implementados.

2. **main.exe**: Este é um arquivo executável que foi gerado a partir do código-fonte. Ele permite a execução do programa para ordenar os dados usando os algoritmos implementados.

3. **commonFunctions.c**: Este arquivo desempenha um papel fundamental ao fornecer funções comuns que são utilizadas durante a implementação dos algoritmos de ordenação. Entre essas funções estão:

- `randomArray`: Esta função gera um array de números aleatórios, o que é útil para testar o desempenho dos algoritmos de ordenação com dados não ordenados.

- `invertedArray`: Esta função cria um array com os elementos em ordem inversa, fornecendo uma entrada específica para avaliar o comportamento dos algoritmos em casos de ordenação reversa.

- `copyArray`: A função copyArray cria uma cópia do array original. Essa cópia é usada para preservar o estado do array original e garantir que ele não seja modificado durante a ordenação.

- `printArray`: A função printArray é utilizada principalmente para fins de depuração e teste. Ela permite a exibição do conteúdo do array no terminal, facilitando a verificação da corretude dos algoritmos de ordenação.

4. **tests.c**: Neste arquivo, são definidos casos de teste para os algoritmos de ordenação implementados. Ele inclui duas funções principais:

- `test`: Esta função é responsável por invocar todos os algoritmos de ordenação implementados para avaliar o desempenho de cada um.

- `runTests`: A função runTests é encarregada de chamar a função test com valores específicos de posições no vetor, permitindo testar os algoritmos com diferentes configurações de dados e fornecendo resultados para avaliação e análise.

5. **test_data.csv**: Este arquivo contém dados gerados pelos testes em formato CSV, que inclue,: Array Type, Algorithm, Dataset Size, Time Spent (seconds) e Operations

6. **bubbleSort.c**: Este arquivo contém a implementação do algoritmo de ordenação "Bubble Sort". O Bubble Sort é um algoritmo simples que percorre repetidamente a lista, compara elementos adjacentes e os troca se estiverem na ordem errada.

7. **insertion.c**: Este arquivo contém a implementação do algoritmo de ordenação "Insertion Sort". O Insertion Sort funciona construindo uma matriz ordenada um item de cada vez, movendo os itens não ordenados para a posição correta.

8. **selection.c**: Este arquivo contém a implementação do algoritmo de ordenação "Selection Sort". O Selection Sort encontra o menor elemento da matriz e o move para a primeira posição, repetindo esse processo para os elementos restantes.

2.2. BOLHA

O algoritmo de ordenação por bolha, também conhecido como "Bubble Sort," é um dos algoritmos mais simples de ordenação. Sua principal característica é percorrer a lista várias vezes, comparando pares de elementos adjacentes e trocando-os caso estejam fora de ordem. Isso é repetido até que nenhum par de elementos necessite ser trocado, indicando que a lista está ordenada.

Principais características:

- É um algoritmo fácil de entender e implementar.
- Tem uma complexidade de tempo de O(n^2) no pior caso, tornando-o ineficiente para listas grandes.
- Funciona bem em listas quase ordenadas ou com apenas alguns elementos fora de ordem.

- É um algoritmo estável, o que significa que ele preserva a ordem relativa de elementos iguais.
- É um algoritmo in-place, o que significa que ele não requer memória adicional para ordenar a lista.

2.3. INSERÇÃO

O algoritmo de ordenação por inserção, ou "Insertion Sort," é um método simples e eficaz de ordenação. Ele constrói a lista ordenada um elemento de cada vez, movendo os elementos não ordenados para a posição correta à medida que avança pela lista. Basicamente, ele divide a lista em uma parte ordenada e uma parte não ordenada, inserindo elementos da parte não ordenada na parte ordenada.

Principais características:

- É eficiente para listas pequenas ou quase ordenadas.
- Tem uma complexidade de tempo de O(n^2) no pior caso, tornando-o menos eficiente para listas grandes.
- É estável, preservando a ordem relativa de elementos iguais.
- É in-place, não requer memória adicional.
- É uma escolha sólida para listas curtas ou quando a lista já está parcialmente ordenada.

2.4.SELEÇÃO

O algoritmo de ordenação por seleção, ou "Selection Sort," é outro algoritmo simples de ordenação. Ele divide a lista em duas partes: a parte ordenada e a parte não ordenada. Em cada passo, o algoritmo seleciona o elemento mínimo da parte não ordenada e o move para a posição correta na parte ordenada. Esse processo é repetido até que toda a lista esteja ordenada.

Principais características:

- Tem uma complexidade de tempo de O(n^2) no pior caso, tornando-o ineficiente para listas grandes.

- Não é estável, o que significa que a ordem relativa de elementos iguais pode não ser preservada.
- É in-place, não requer memória adicional.
- É fácil de entender e implementar.
- Funciona bem em listas pequenas, mas não é a melhor escolha para listas maiores devido à sua complexidade de tempo quadrática.

3. RESULTADOS

Os resultados obtidos neste estudo refletem o desempenho dos algoritmos de ordenação quando aplicados a diferentes tamanhos de conjuntos de dados, considerando cenários de pior caso com vetores parcialmente ordenados e invertidos.

É importante observar que os resultados podem variar dependendo das especificações do hardware utilizado para realizar os testes. Neste estudo, os testes foram conduzidos em um sistema com as seguintes especificações:

Computador: Lenovo Ideapad Gaming 3i

Processador: Intel Core i5 1130H

Memória RAM: 8GB

O hardware do sistema pode influenciar no tempo de execução dos algoritmos. Portanto, é importante considerar que os resultados podem ser diferentes em diferentes máquinas.

Para acessar o código-fonte deste projeto e examinar os detalhes dos testes realizados, você pode visitar o seguinte repositório no GitHub: https://github.com/paulaandrezza/algorithm-analysis

3.1.Tabela BOLHA

Array Type	Algorithm	Dataset Size	Time Spent (seconds)	Operations
Random	bubbleSort	1000	0,003	254210
Random	bubbleSort	10000	0,236	24871370
Random	bubbleSort	100000	31,982	2516432883
Inverted	bubbleSort	1000	0,002	499500
Inverted	bubbleSort	10000	0,239	49995000
Inverted	bubbleSort	100000	23,997	704982704

3.2.Tabela INSERÇÃO

Array Type	Algorithm	Dataset Size	Time Spent (seconds)	Operations
Random	insertion	1000	0,000	254210
Random	insertion	10000	0,051	24871370
Random	insertion	100000	5,266	2516432883
Inverted	insertion	1000	0,001	499500
Inverted	insertion	10000	0,103	49995000
Inverted	insertion	100000	10,473	704982704

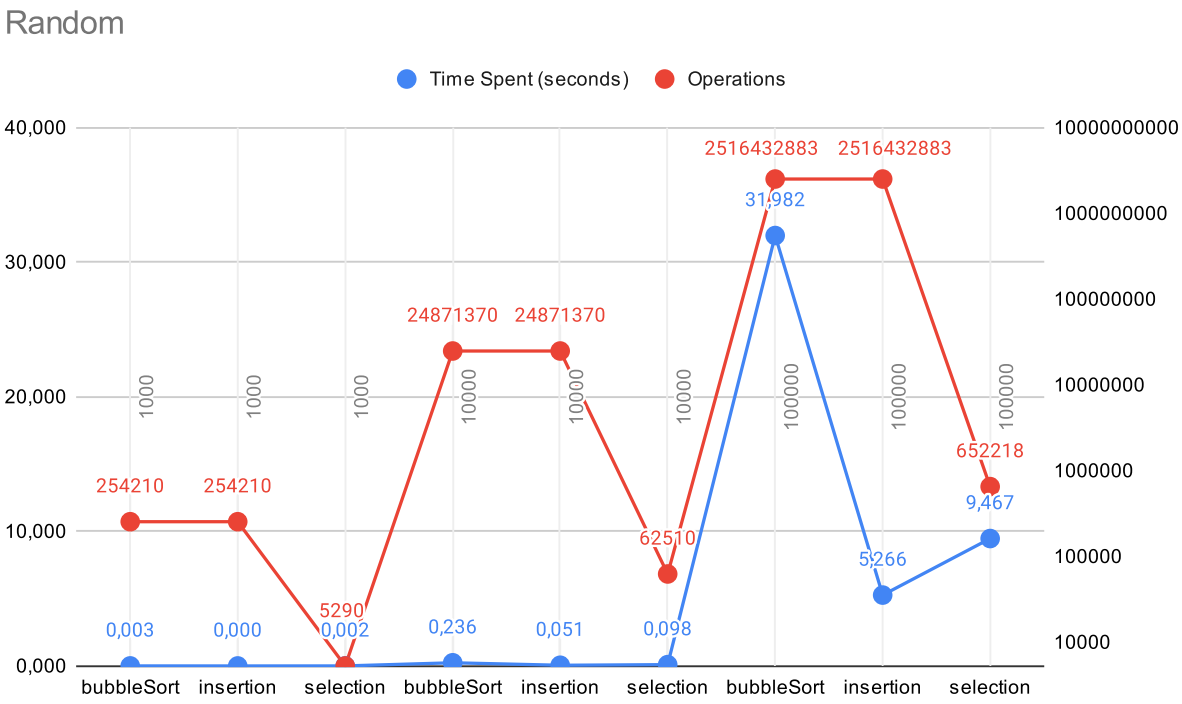
3.3.Tabela SELEÇÃO

Array Type	Algorithm	Dataset Size	Time Spent (seconds)	Operations
Random	selection	1000	0,002	5290
Random	selection	10000	0,098	62510
Random	selection	100000	9,467	652218
Inverted	selection	1000	0,001	250000
Inverted	selection	10000	0,100	25000000
Inverted	selection	100000	10,070	2500000000

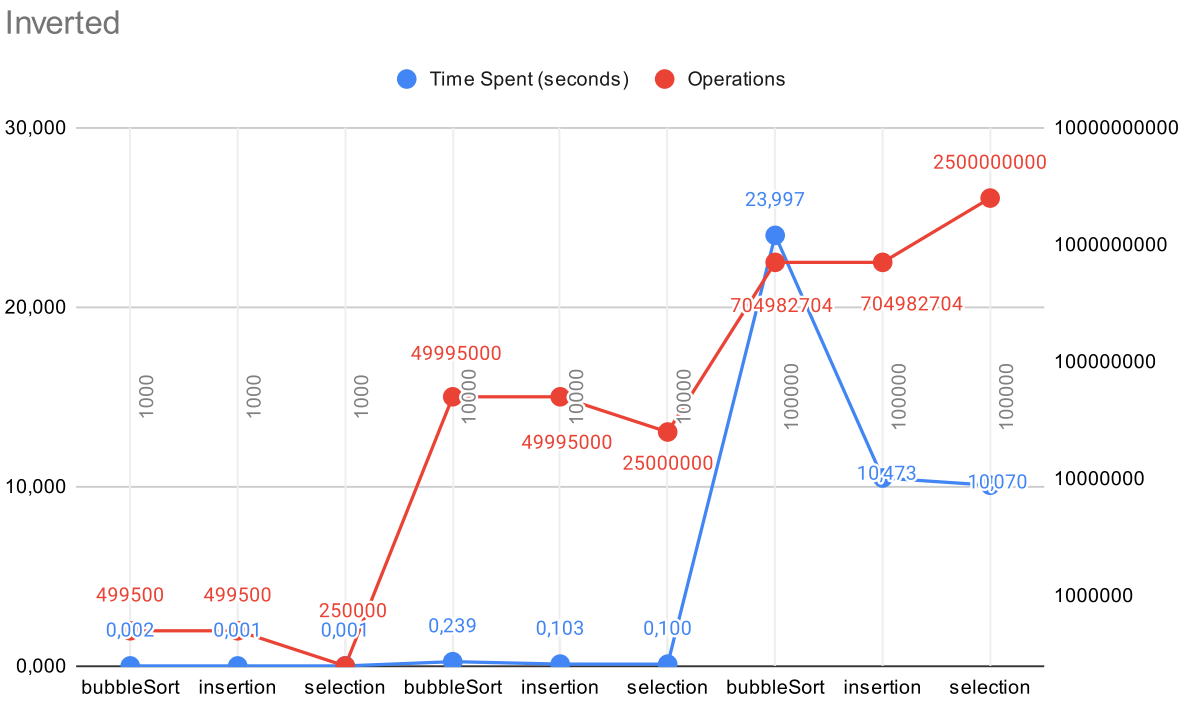
3.4.Tabela Completa com todos os casos de teste

Array Type	Algorithm	Dataset Size	Time Spent (seconds)	Operations
Random	bubbleSort	1000	0,003	254210
Random	insertion	1000	0,000	254210
Random	selection	1000	0,002	5290
Random	bubbleSort	10000	0,236	24871370
Random	insertion	10000	0,051	24871370
Random	selection	10000	0,098	62510
Random	bubbleSort	100000	31,982	2516432883
Random	insertion	100000	5,266	2516432883
Random	selection	100000	9,467	652218
Inverted	bubbleSort	1000	0,002	499500
Inverted	insertion	1000	0,001	499500
Inverted	selection	1000	0,001	250000
Inverted	bubbleSort	10000	0,239	49995000
Inverted	insertion	10000	0,103	49995000
Inverted	selection	10000	0,100	25000000
Inverted	bubbleSort	100000	23,997	704982704
Inverted	insertion	100000	10,473	704982704
Inverted	selection	100000	10,070	2500000000

3.5.Gráfico Aleatório



3.6.Gráfico Invertido



4. CONCLUSÃO

Com base nos dados do teste apresentados, podemos tirar as seguintes conclusões sobre o desempenho dos algoritmos de ordenação (Bubble Sort, Insertion Sort e Selection Sort) em diferentes cenários de conjuntos de dados (aleatórios e inversamente ordenados) e tamanhos de dados (1000, 10000 e 100000 elementos):

4.1.Para Conjuntos de Dados Aleatórios

O algoritmo de ordenação Bubble Sort tem um desempenho significativamente pior em comparação com os outros dois algoritmos, em todos os tamanhos de dados. O tempo de execução aumenta de forma substancial à medida que o tamanho do conjunto de dados aumenta, tornando-o ineficiente para grandes conjuntos de dados.

O algoritmo de ordenação Insertion Sort demonstra um desempenho melhor que o Bubble Sort, mas ainda assim é ineficiente para grandes conjuntos de dados. O tempo de execução aumenta à medida que o tamanho do conjunto de dados aumenta.

O algoritmo de ordenação Selection Sort é mais eficiente do que o Bubble Sort e o Insertion Sort para conjuntos de dados aleatórios, mesmo que seu desempenho também piore com tamanhos maiores de dados.

4.2.Para Conjuntos de Dados Inversamente Ordenados

O Bubble Sort ainda é o algoritmo menos eficiente, com um tempo de execução consideravelmente maior, especialmente em conjuntos de dados maiores.

O Insertion Sort também é menos eficiente do que o Selection Sort, especialmente em conjuntos de dados maiores.

O Selection Sort é o algoritmo mais eficiente entre os três para conjuntos de dados inversamente ordenados, embora seu desempenho ainda seja afetado pelo tamanho do conjunto de dados.

4.3.Conclusão Geral

O desempenho dos algoritmos de ordenação é altamente dependente do tipo de dados de entrada e do tamanho do conjunto de dados.

Para conjuntos de dados pequenos ou quase ordenados, o Insertion Sort e o Selection Sort podem ser opções razoáveis.

Para conjuntos de dados maiores e mais complexos, considerando tanto o tempo de execução quanto o número de operações, o Selection Sort tende a se destacar em eficiência em comparação com o Bubble Sort e o Insertion Sort.

Em resumo, a escolha do algoritmo de ordenação deve levar em consideração o tamanho do conjunto de dados e o tipo de dados de entrada, uma vez que cada algoritmo tem suas próprias vantagens e desvantagens em diferentes cenários.

5. REFERÊNCIAS BIBLIOGRAFICAS

C File Handling. **Programiz**. Disponivel em: <https://www.programiz.com/c-programming/c-file-input-output>. Acesso em: 15 set. 2023.

DA SILVA, S. F. **Ordenação I**. IFSP. Bragança Paulista. 2023.

DA SILVA, S. F. **Ordenação II**. IFSP. Bragança Paulista. 2023.

MEDINA, M.; FERTIG, C. **Algoritmos e Programação - Teoria e Prática**. [S.l.]: Novatec Editora, 2005.