

UNIVERSIDADE FEDERAL DE MINAS GERAIS
CIÊNCIA DA COMPUTAÇÃO

DCC205: Estruturas de Dados

Autor: Luisa Lopes Carvalhaes
Matrícula: 2023028064

Trabalho Prático I

“Métodos de Ordenação”

June 24, 2024

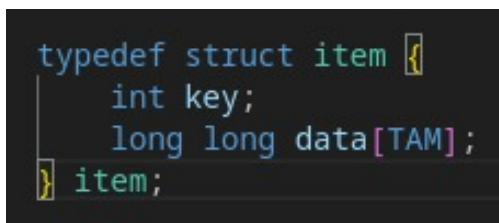
1 Introdução

O objetivo principal deste trabalho é analisar experimentalmente os métodos de ordenação e contrastar os resultados dessa análise com a análise de complexidade teórica de cada método estudada na matéria de Estrutura de Dados. Os algoritmos a serem analisados incluem o Método da Bolha, Método da Inserção, Método da Seleção, Merge Sort, Quick Sort, Shell Sort, Counting Sort, Bucket Sort e Radix Sort. A fim de cumprir o objetivo de contrastar a análise experimental dos métodos de ordenação com sua análise de complexidade teórica, três cargas de trabalho foram propostas para compreender melhor o desempenho dos algoritmos em diferentes cenários. Dessa maneira será possível entender os compromissos envolvidos ao escolher algum desses métodos de ordenação ao lidar com determinado contexto.

2 Método

Neste trabalho prático, os algoritmos de ordenação mencionados foram implementados em linguagem C, seguindo práticas e técnicas discutidas em sala de aula para garantir eficiência e precisão. Cada algoritmo foi implementado como uma função separada, conforme as instruções do enunciado. Os algoritmos estão organizados em um arquivo `main.c`, onde cada um é chamado para operar sobre um vetor de itens, representado por um Tipo Abstrato de Dados (TAD) `item`, permitindo a criação de itens de tamanhos variados `TAM`, e vetores de tamanho variado, além de outras cargas de trabalho.

O tamanho do vetor de itens é alocado estaticamente, enquanto o tamanho do vetor em si é alocado dinamicamente. Um script python foi desenvolvido para executar o código C, gerando as cargas de trabalho definidas para avaliar cada algoritmo. Esse script também captura o tempo de execução exato de cada algoritmo durante a execução desse no código em C e reúne esses dados em gráficos que serão apresentados posteriormente para ilustrar os resultados de cada método de ordenação.



```
typedef struct item {
    int key;
    long long data[TAM];
} item;
```

Figure 1: Struct Item

Algoritmos Implementados

Foram implementados os seguintes algoritmos: Método da Bolha, Método da Inserção, Método da Seleção, Merge Sort, Quick Sort, Shell Sort, Counting Sort, Bucket Sort, Radix Sort.

Carga de Trabalho

Foram propostas diferentes configurações de teste para a análise experimental, abrangendo diversas cargas de trabalho que exploram as características dos algoritmos de ordenação. As cargas de trabalho consideradas são definidas pelas seguintes dimensões:

- Tamanho do vetor: Variação de 1 a 3000 elementos;
- Tamanho do item do vetor: Variação de 1 a 1000 bytes;
- Configuração inicial do vetor: ordenado e inversamente ordenado.;
- Outras variações de carga, específicas para certos algoritmos.

Essas configurações proporcionarão uma análise abrangente do desempenho dos algoritmos de ordenação em uma variedade de cenários. Cada carga de trabalho terá o propósito de explorar as características individuais dos algoritmos.

3 Análise de Complexidade

Nesta seção, os algoritmos serão agrupados em classes de acordo com sua complexidade. Será apresentada de maneira sucinta a complexidade teórica de cada algoritmo para comparação posterior com análises experimentais.

Complexidade de Tempo Quadrático ($O(n^2)$)

Nesta classe, a complexidade dos algoritmos aumenta quadraticamente com o tamanho da entrada, resultando em um crescimento rápido do tempo de execução conforme o problema se torna maior. Na categoria de tempo quadrático, temos:

- **Método da Bolha:** O Bubble Sort compara sucessivamente pares adjacentes de elementos e os troca se estiverem na ordem errada. Este processo se repete até que a lista esteja ordenada. Tempo : $O(n^2)$, Espaço : $O(1)$
- **Método da Inserção:** O Insertion Sort constrói a lista ordenada um elemento por vez, movendo os elementos maiores para a direita e inserindo o elemento atual na posição correta. Tempo : $O(n^2)$, Espaço : $O(1)$
- **Método da Seleção:** O Selection Sort seleciona repetidamente o menor elemento da lista não ordenada e o coloca na posição correta. Tempo : $O(n^2)$, Espaço : $O(1)$

Complexidade de Tempo Quase Linear ($O(n \log n)$)

Algoritmos nesta classe têm um tempo de execução quase linear com um fator logarítmico adicional, sendo eficientes para uma ampla gama de tamanhos de entrada. Na categoria de tempo quase linear, destacam-se:

- **Merge Sort:** O Merge Sort divide a lista em sublistas até que cada sublista contenha um único elemento e, em seguida, faz o "merge" dessas sublistas em ordem. Tempo : $O(n \log n)$, Espaço : $O(n)$
- **Quick Sort:** O Quick Sort seleciona um pivô e particiona a lista em elementos menores e maiores que o pivô, ordenando as sublistas recursivamente. No pior caso, ocorre quando o pivô é sempre o menor ou maior elemento. Tempo : $O(n^2)$ no pior caso, $O(n \log n)$ no caso médio, Espaço : $O(\log n)$
- **Shell Sort:** O Shell Sort é uma generalização do Insertion Sort que permite a troca de elementos a uma distância maior, em intervalos. A complexidade varia com a escolha desses intervalos. Tempo : $O(n^2)$ no pior caso, $O(n \log n)$ em média, Espaço : $O(1)$

Complexidade de Tempo Linear ($O(n)$)

Algoritmos com complexidade linear têm um tempo de execução que cresce proporcionalmente ao tamanho da entrada, sendo altamente eficientes para grandes conjuntos de dados. Nesta categoria, encontramos:

- **Counting Sort:** O Counting Sort conta o número de ocorrências de cada valor e utiliza essa contagem para posicionar os elementos na lista ordenada. Tempo : $O(n + k)$, onde k é o intervalo dos números, Espaço : $O(k)$

- **Bucket Sort:** O Bucket Sort distribui os elementos em vários baldes (buckets) e então ordena cada balde individualmente. O pior caso ocorre quando os elementos não são distribuídos uniformemente. Tempo : $O(n^2)$ no pior caso, $O(n + k)$ em média, Espaço : $O(n + k)$
- **Radix Sort:** O Radix Sort ordena os elementos considerando a representação binária de cada dígito, um de cada vez, Tempo : $O(n \cdot k)$, onde k é o número de dígitos (em bits) das chaves, Espaço : $O(n + k)$

4 Estratégias de Robustez

Durante a implementação dos algoritmos, foram adotados cuidados para garantir sua robustez. Inicialmente, foi desenvolvida uma estrutura de dados `item`, já explicada anteriormente, que é simples, projetada para facilitar a manipulação de diferentes tipos e tamanhos de dados de forma prática. Além disso, o código está bem organizado, com funções modularizadas e de fácil chamada. A alocação de memória, tanto dinâmica quanto estática, foi tratada com atenção, incluindo a desalocação adequada quando necessário. Outro ponto relevante é que o código não requer interação direta com o usuário, pois será executado por meio de um script, simplificando o processo e reduzindo a possibilidade de erros relacionados à entrada de dados. Essas práticas tornam o código mais previsível e menos suscetível a falhas, operando de maneira eficiente em conjuntos de dados predefinidos. Em suma, o código adota uma abordagem simples e genérica, porém cuidadosa, visando robustez e eficiência do programa.

5 Análise Experimental

Serão apresentados os resultados dos experimentos realizados em termos de desempenho computacional, bem como as análises destes. Os algoritmos foram agrupados em suas respectivas complexidades teóricas anteriormente discutidas. Os experimentos utilizaram diferentes cargas de trabalho mencionadas, variando o tamanho do vetor e o tamanho dos itens no vetor. Em seguida, para melhor entendimento e visualização dos resultados, foram feitos gráficos que terão suas interpretações para cada classe de algoritmos. Para simplificar, algoritmos da mesma classe compartilharão um mesmo gráfico, e cada carga de trabalho terá seu próprio gráfico.

Complexidade $O(n^2)$

Método da Bolha, Método da Inserção, Método da Seleção

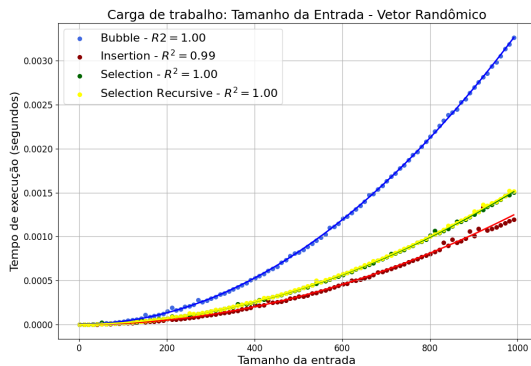


Figure 2: Algs. $O(n^2)$ - Tamanho do vetor

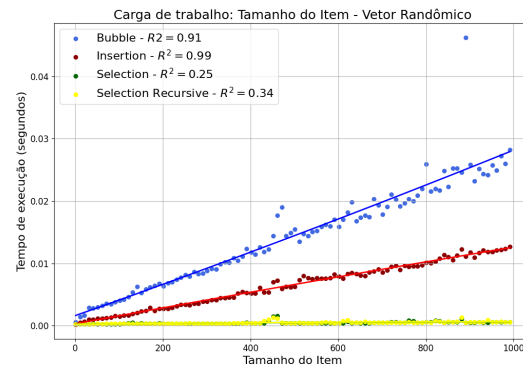


Figure 3: Algs. $O(n^2)$ - Tamanho do item

. Os gráficos mostram que a análise computacional está de acordo com a análise teórica, no que tange à forma da curva que os valores assumem à medida que o tamanho da entrada cresce. É visivelmente o formato

quadrático que as curvas assumem, o que é evidenciado pela aproximação da curva com duas variáveis e seu R^2 . Além disso, é notável a diferença de desempenho à medida que o tamanho do vetor e o tamanho do item crescem. Observamos que o Selection Sort se mostra mais eficiente no aumento da quantidade de itens, pois dentre esses três algoritmos, é o que realiza menos movimentações, cerca de $O(n)$, enquanto os outros dois realizam cerca de $O(n^2)$ movimentações. Logo, ele é menos afetado pelo tamanho do item que precisa ser movido durante as trocas. Por fim, os gráficos, de maneira geral, estão conforme o esperado, com o Bubble Sort, que é menos eficiente entre eles por conta do alto número de movimentações e comparações, crescendo em tempo mais rapidamente, seguido de Insertion e Selection, que são bastante próximos em termos de eficiência como foi visto em aula.

Complexidade $O(n \log n)$

Merge Sort, Quick Sort, Shell Sort

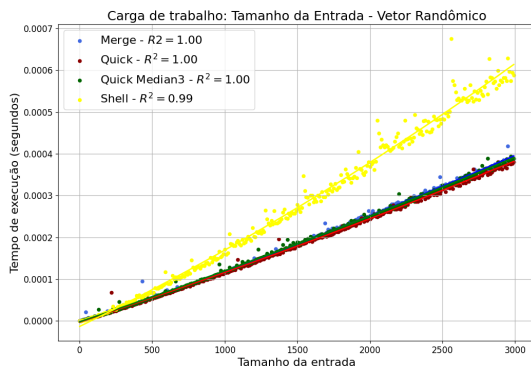


Figure 4: Algs. $O(n \log n)$ - Tamanho do vetor

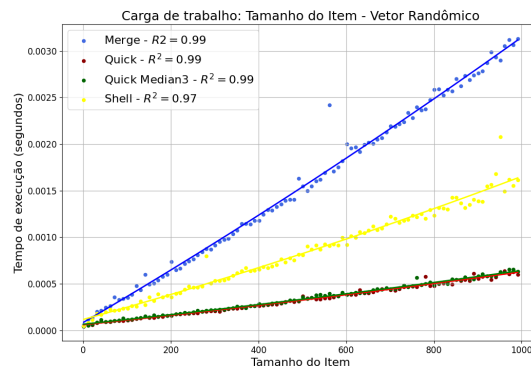


Figure 5: Algs. $O(n \log n)$ - Tamanho do item

Os gráficos também mostram que a análise computacional está de acordo com a análise teórica, com valores que crescem formando uma curva que segue uma distribuição $n \log n$. Podemos observar que o Merge Sort e os Quicksorts estão bem próximos à medida que o tamanho do vetor de entrada cresce, o que ocorre porque seus casos médios têm complexidade $n \log n$. Já o Shell Sort, que se apresenta como o menos eficiente no gráfico, o que faz sentido, pois o algoritmo é uma extensão do método de ordenação por inserção e sua complexidade é melhor que $O(n^2)$, porém não tão boa quanto a dos outros dois algoritmos.

No que tange ao aumento do tamanho do item do vetor, o Merge Sort é o mais afetado, devido ao grande número de comparações e trocas, além da necessidade de espaço de memória adicional de ordem linear para a etapa de conquista do algoritmo. Já o Shell Sort, ao ordenar subsequências periódicas, reduz o número de comparações e movimentações, assim como o Quick Sort, que, por sua vez, se sai melhor nesse cenário por realizar menos comparações e trocas.

Complexidade $O(n)$

Counting Sort $O(n + k)$, Bucket Sort $O(n + k)$, Radix Sort $O(k \times n)$

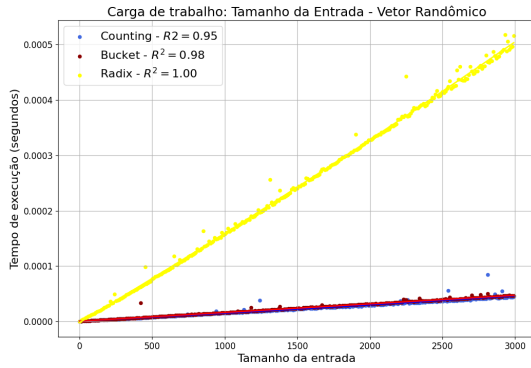


Figure 6: Algs. $O(n)$ - Tamanho do vetor

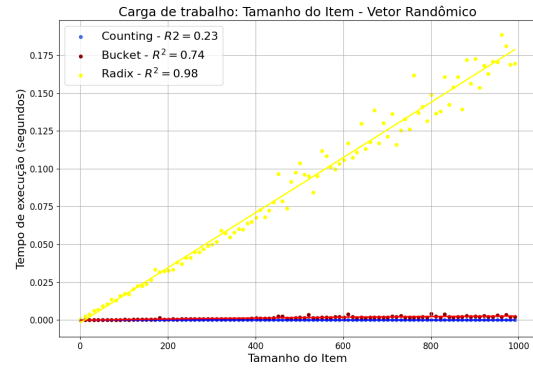


Figure 7: Algs. $O(n)$ - Tamanho do item

Os gráficos evidenciam que a análise computacional está alinhada com a teórica, o que será ainda mais claro na próxima seção, onde todos os algoritmos serão comparados diretamente, e será visível a diferença da curva dos algoritmos $O(n^2)$ e os $O(n)$. Notamos uma discrepância marcante entre o Radix Sort e os outros algoritmos. Isso ocorre porque o Radix Sort processa a representação binária das chaves para ordenar o vetor. Portanto, é plausível que o Radix Sort seja mais lento à medida que o tamanho do vetor aumenta, pois nessa configuração de dados usei chaves de valor variável em bits; um detalhamento dessa característica será explorado também no próximo tópico. Além disso, quanto ao aumento do tamanho do item, observamos que os gráficos permanecem bastante semelhantes. Isso acontece porque esses métodos são não comparativos, ou seja, não realizam comparações diretas entre elementos, mas sim entre suas chaves. Outros atributos, como a escala dos valores das chaves nos elementos da lista, têm maior impacto. Por exemplo, o Counting Sort é dominado pela magnitude do maior elemento presente na lista, podendo se tornar extremamente ineficiente com chaves muito grandes. O Bucket Sort, por sua vez, depende da distribuição dos dados a serem ordenados. Se os valores das chaves estiverem concentrados em um pequeno intervalo, pode resultar em uma inserção custosa e um tempo de execução próximo da ordem quadrática, pois os elementos serão majoritariamente incluídos nos mesmos "buckets".

6 Análise Comparativa

Ao combinar o entendimento da análise de complexidade teórica com os resultados da análise experimental, é possível determinar quais tipos de algoritmos serão mais adequados e eficientes dependendo do cenário em que estão inseridos. Esta seção visa comparar os algoritmos previamente definidos entre si, levando em consideração a configuração inicial do vetor. Além disso, serão examinados os casos mais marcantes e interessantes de comparação entre algoritmos, destacando como diferentes cargas de trabalho podem influenciar a visualização dos dados. Isso é crucial, pois diferentes representações gráficas podem revelar perspectivas distintas e mais abrangentes sobre o problema em questão.

Complexidade $O(n^2)$

Método da Bolha, Método da Inserção, Método da Seleção

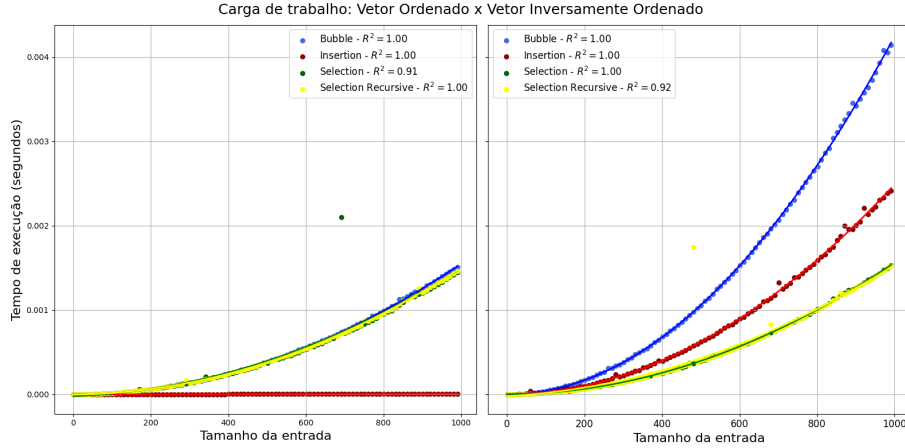


Figure 8: Algs. $O(n^2)$ - Ordenado x Inversamente ordenado

Comparando agora as configurações iniciais dos vetores, vemos que os gráficos mostram que a análise computacional retifica a análise teórica, o que se demonstra principalmente ao observar a curva do Insertion Sort no seu melhor caso (vetor ordenado), onde, de forma teórica, a complexidade do algoritmo cai para $O(n)$, fato que se reflete no gráfico. De maneira geral, o vetor ordenado representa a configuração que mais favorece os 3 algoritmos. No vetor inversamente ordenado, o padrão também é o esperado, seguindo a complexidade de $O(n^2)$ e gerando uma curva com forma quadrática, que se assemelha ao caso médio (vetor randômico). No entanto, o Insertion Sort demorou mais que o Selection Sort, pois seu pior caso é o vetor inversamente ordenado, e por fazer várias movimentações, performa mal, assim como o Bubble Sort, que já era o mais ineficiente.

Complexidade $O(n \log n)$

Merge Sort, Quick Sort, Shell Sort

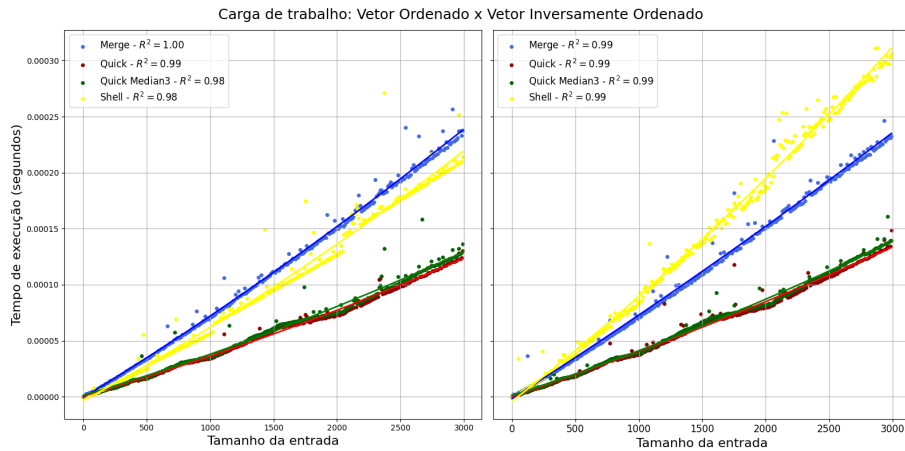


Figure 9: Algs. $O(n \log n)$ - Ordenado x Inversamente ordenado

Os gráficos também mostram que a análise computacional está de acordo com a análise teórica, com valores que crescem se aproximando do limite assintótico de $n \log n$. Podemos observar que os gráficos são similares, com variações entre MergeSort e Shell Sort como aqueles que demoram mais tempo. Isso ocorre porque o vetor ordenado ou inversamente ordenado não é necessariamente o melhor ou pior caso para nenhum dos algoritmos apresentados nesta classe. Por exemplo, o pior caso do QuickSort ocorre quando o

pivô é escolhido recorrentemente em uma das extremidades; a cada iteração, o algoritmo produz apenas uma partição com tamanho semelhante ao do vetor original. Sendo assim, o método é incapaz de dividir de forma eficiente sua ordenação, incorrendo em um custo computacionalmente mais elevado. Porém, esse não seria o caso de nenhum dos Quicksorts implementados, pois para tal, seria necessário o uso de uma configuração muito específica de vetor, ou uma implementação que permitesse essa escolha desfavorável de pivôs. Os três algoritmos escolhidos são indiferentes a esse tipo de configuração inicial do vetor., por isso tem gráficos semelhantes.

Complexidade $O(n)$

Counting Sort $O(n + k)$, Bucket Sort $O(n + k)$, Radix Sort $O(k \times n)$

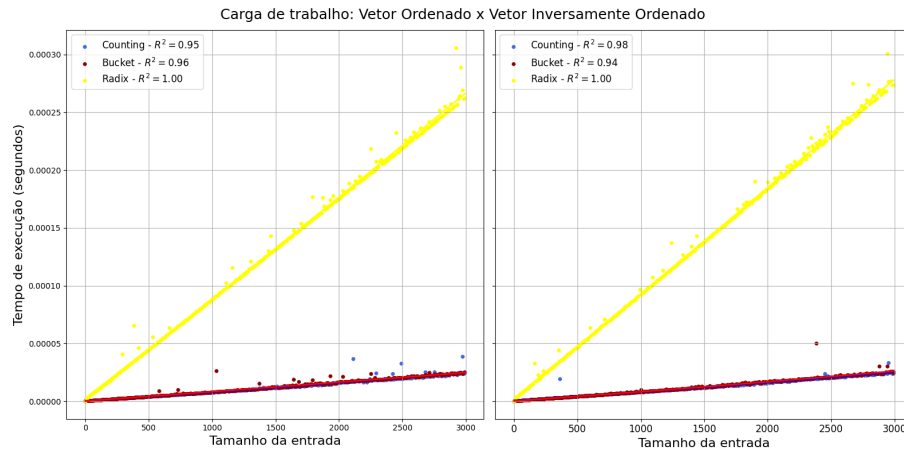


Figure 10: Algs. $O(n)$ - Ordenado x Inversamente ordenado

Neste caso comparativo, podemos observar que ambos os gráficos são praticamente idênticos. Isso ocorre porque esses métodos não dependem da configuração inicial do vetor, já que estar ordenado ou inversamente ordenado não corresponde ao melhor ou pior caso para nenhum deles. Isso se deve ao fato de que são algoritmos não comparativos em geral. Por exemplo, no Bucket Sort, a ordenação prévia do vetor pode influenciar o número de elementos em cada balde (bucket), mas isso apenas se essa ordenação contar com chaves distribuídas de forma desbalanceada e, conseqüentemente, impactar seu desempenho. No entanto, o impacto não é tão direto nessa escolha de carga de trabalho específica. No tópico a seguir, exploraremos um pouco dessas configurações que impactariam mais os métodos não comparativos.

Outras comparações e visualização de dados

Alguns gráficos interessantes para refletir sobre:

Um outro ponto importante da interpretação de dados deste trabalho é compreender a visualização dos dados coletados com a experimentação de cada algoritmo. Muitas vezes, os gráficos nos apresentam apenas parte da informação, então certas representações podem parecer confusas à primeira vista. Porém, com um olhar atento e uma boa análise, é possível perceber esses detalhes. Por exemplo, abaixo temos a comparação de todos os algoritmos:

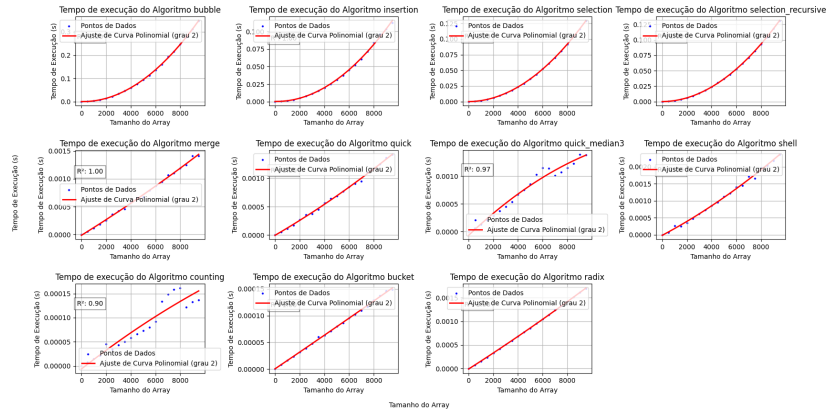


Figure 11: Todos algoritmos: eixo y independente

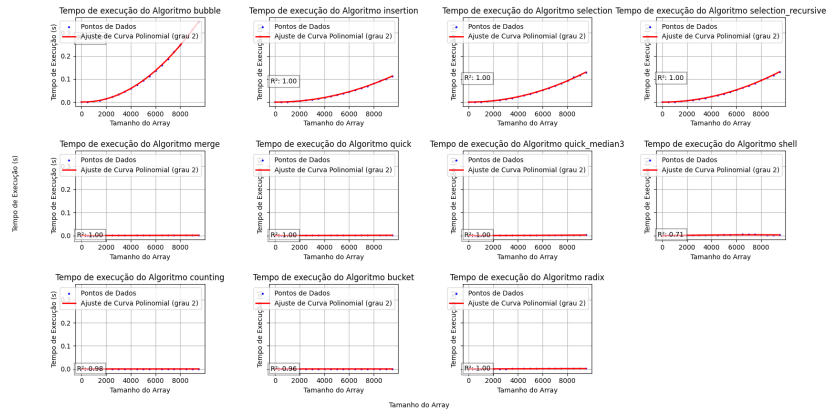


Figure 12: Todos algoritmos: eixo y compartilhado

Na figura 11, observamos que cada gráfico possui seu próprio eixo y, indicando que cada um está em uma escala particular, dependendo apenas dos resultados dos seus dados. No segundo gráfico, todos os algoritmos compartilham a mesma escala no eixo y, evidenciando assim as diferenças entre eles. É claro no segundo gráfico a distinção entre os algoritmos $O(n^2)$ e os demais, enquanto no primeiro gráfico essa distinção não era tão evidente. Mesmo entre algoritmos da mesma classe, isso pode ocorrer, e agrupá-los no mesmo gráfico pode causar uma perda desse detalhamento, como por exemplo:

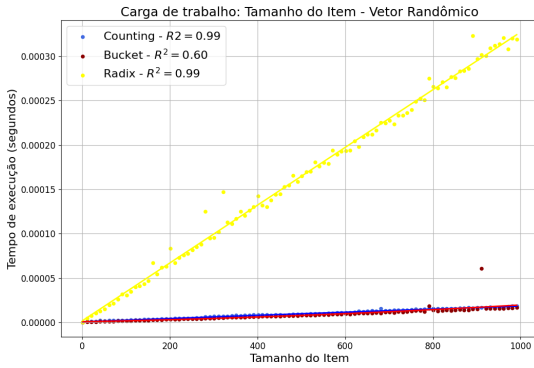


Figure 13: Algs. $O(n)$ - Todos

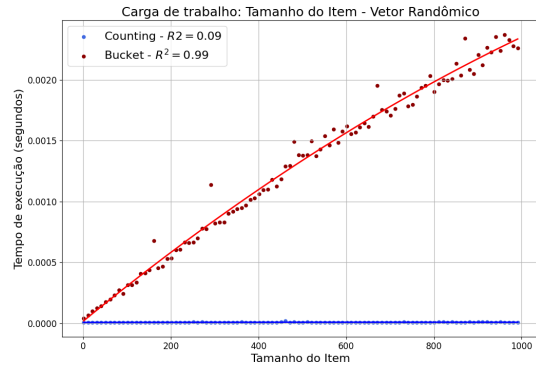


Figure 14: Algs. $O(n)$ - BucketSort x CountingSort

Na figura onde os três algoritmos estão juntos, o counting sort e o bucket sort acabam parecendo muito próximos devido ao mau desempenho do radix (isso ocorre pois minhas chaves não beneficiavam o caso do radix, como já explicado anteriormente). Excluindo o radix do gráfico, conseguimos ver a diferença entre o counting sort e o bucket sort. Esses exemplos servem apenas para dar um maior panorama para essa análise, e para que seja possível entender com mais detalhes as implementações. Quando juntas é possível compará-las mais facilmente e ter uma visão geral do problema que estamos buscando entender.

Outras análises comparativas - Bucketsort, counting sort e radix sort

Devido a esses métodos não comparativos não terem seus melhores e piores casos explícitos nas análises anteriores realizadas, decidi explorar outros aspectos dos algoritmos para entender melhor suas características. Com dados randômicos em cada vetor, e vetores com chaves de até três dígitos de tamanho no máximo, temos a seguinte correlação entre os algoritmos, que servirá como uma base para as próximas comparações.

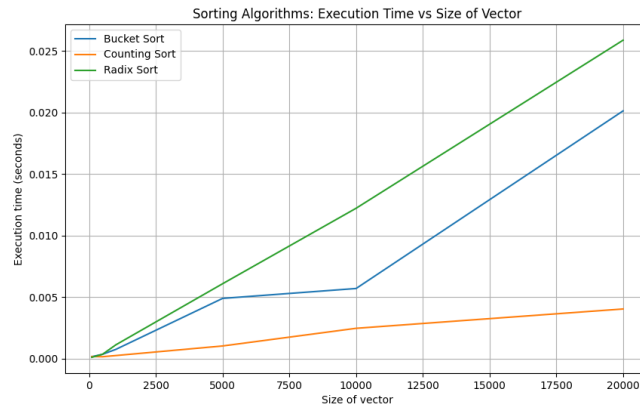


Figure 15: Algoritmos $O(n)$

Com esses dados bem definidos, vamos explorar os melhores e piores casos de cada um dos algoritmos. Por serem algoritmos não comparativos, para explorar suas especificidades, teremos que analisar aspectos como o tamanho das chaves e a distribuição de chaves nos vetores.

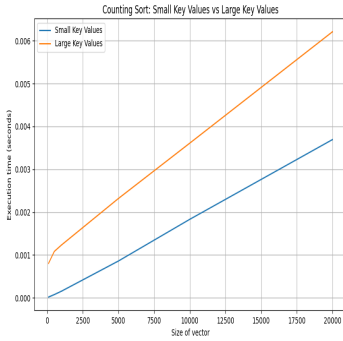


Figure 16: CountingSort - Tamanho das chaves

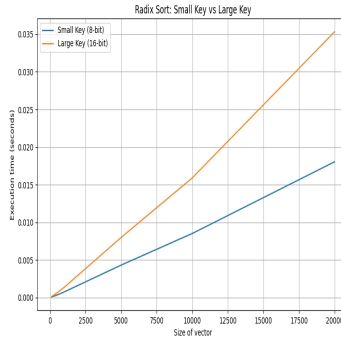


Figure 17: RadixSort - Tamanho binário das chaves

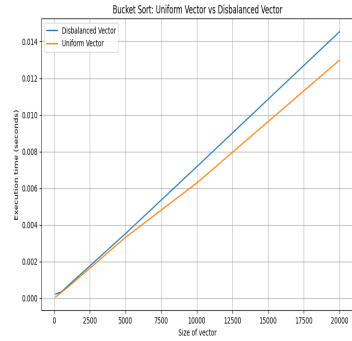


Figure 18: BucketSort - Distribuição das chaves

Nas figuras acima, vemos que, ao explorar outros aspectos da configuração inicial do vetor, obtemos resultados muito diferentes. Vamos entender cada um dos gráficos. A figura 16 refere-se ao algoritmo do CountingSort. Como sua complexidade é $O(n + k)$, onde k é o intervalo dos números, ele depende do maior valor das chaves presentes no vetor. Assim, à medida que o tamanho da chave aumenta, o algoritmo se torna menos eficiente. O "melhor caso" seria para chaves de até 10 dígitos, o que já não é extremamente eficiente, mas comparado à chaves de até 10000, o "pior caso" mostra uma diferença considerável. Logo, ao enfrentar um problema, devemos estar atentos a essa natureza do algoritmo. A figura 17 refere-se ao RadixSort, que também é influenciado pelo tamanho das chaves, especificamente pela representação binária delas. O "melhor caso" seria chaves representadas por 8 bits, enquanto o "pior caso" seria com representação de 16 bits, mostrando uma diferença significativa, similar ao CountingSort. Por fim, no BucketSort, temos casos de melhor e pior muito claros: o pior caso ocorre quando os valores das chaves estão concentrados em um pequeno intervalo, fazendo com que todos os valores se agrupem em um mesmo "bucket" aumentando a complexidade; o melhor caso ocorre quando os valores estão uniformemente distribuídos. No gráfico, vemos apenas uma pequena diferença, pois o vetor desbalanceado é apenas ligeiramente desbalanceado e não um caso extremo de desbalanceamento dos dados.

7 Conclusões

A comparação entre a teoria dos métodos de ordenação e os resultados práticos obtidos nos experimentos revela "insights" cruciais para a escolha e otimização de algoritmos na prática. Os testes destacaram que não existe um método universalmente superior; cada um possui suas vantagens e limitações específicas. Isso nos faz refletir sobre a complexidade das decisões que enfrentamos ao implementar soluções computacionais. Por isso, um conhecimento aprofundado sobre o funcionamento dos algoritmos se torna tão importante, com ele é possível distinguir o algoritmo que se adapta melhor ao cenário em que trabalhamos.

Os experimentos nos proporcionaram uma visão mais profunda das implicações práticas na escolha dos métodos de ordenação, desde entender a complexidade teórica até lidar com os desafios reais de desempenho. A falta de consenso sobre qual método é "o melhor" reforça a importância de uma análise cuidadosa dos compromissos envolvidos: eficiência, estabilidade e viabilidade prática de cada algoritmo.

Ademais, ficou claro que a adaptação do método de ordenação às características específicas do problema, como o volume e o tipo de dados, é crucial. Este estudo é um pontapé inicial para a compreensão dos algoritmos e Estruturas de Dados para nós, futuros profissionais, ao enfrentar decisões de implementação complexas e ao buscar soluções eficientes e escaláveis para diversos contextos computacionais.

References

- [1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.

- [2] N. Ziviani, *Projeto de Algoritmos com Implementações em Pascal e C*, 3^a edição, Cengage Learning, 2011.