# Trabalho Prático 1 - Ordenador Universal

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

**João Henrique Alves Martins**
**jalvesmartins16@ufmg.br**

## 1 . Introdução

A escolha pelo algoritmo de ordenação ideal é uma das primeiras e principais lições do curso de Estruturas de Dados. Cada vetor, dependendo da sua configuração, tem o seu algoritmo mais eficiente, e essa variação depende de diversos fatores.

O problema proposto nesse trabalho foi a implementação de um sistema de calibragem de limiares para a otimização da ordenação de vetores. Dado um vetor específico, o sistema deve fazer a sua leitura e, baseado em 6 parâmetros dados junto ao vetor, calcular valores ótimos para limiares que determinariam a escolha entre dois algoritmos de ordenação, o Quicksort e o Insertion Sort, para ordenar o vetor da forma mais eficiente, baseado nas suas características. Para resolver o problema, foi implementado um programa que lida com todo o processo de leitura e calibragem dos limiares, além de informar as estatísticas usadas nesse processo.

Essa documentação tem como objetivo esclarecer todas as características do programa que foi implementado. Além da introdução, esse documento conta com detalhes das decisões de implementação (Seção 2), análise de complexidade dos algoritmos usados (Seção 3), estratégias de robustez utilizadas no desenvolvimento (Seção 4), análise experimental (Seção 5) e conclusões finais do trabalho (Seção 6), além da bibliografia utilizada (Seção 7).

## 2 . Método

O programa foi implementado na linguagem C e compilado pelo compilador GCC (GNU Compiler Collection). Essa linguagem foi escolhida, ao invés do C++, devido às características do problema. Como o programa é apenas uma prova inicial para um outro sistema mais robusto, não existe a necessidade da implementação de classes e proteção de atributos. Foram usados o Sistema Operacional Linux (Ubuntu) e o processador o 11th Gen Intel Core i5-1135G7×8. A máquina possui 8GB de RAM.

O código fonte foi separado da seguinte forma:

- estatísticas.c (TADs que armazenam as estatísticas dos algoritmos)
- sort.c (Algoritmos de ordenação)
- ordenador.c (Funções de calibragem de limiares)
- main.c (Função principal do programa)

Os algoritmos de ordenação (Quicksort e Insertion Sort) e algumas funções auxiliares de registro de estatísticas foram baseadas nas implementações do trabalho "PA1" e nos slides da disciplina de Estrutura de Dados.

A implementação se baseia em dois TADs:

- Stat : Armazena as estatísticas da execução de um algoritmo de ordenação.
- CostStat : Armazena o custo de uma determinada configuração de execução dos algoritmos de ordenação.

A decisão de separá-los foi devido a apenas o CostStat precisar ser armazenado em forma de vetor, enquanto o Stat pode ser reutilizado em todas as execuções de ordenação, evitando um custo de espaço desnecessário.

O programa funciona da seguinte forma: Recebe em forma de arquivo alguns parâmetros sobre o vetor, como tamanho, limiar de custo, coeficientes para o cálculo do custo, a semente (Variável usada em alguns algoritmos de geração de números aleatórios para controlar os experimentos de embaralhamento de vetores) e todos os valores do vetor, que é alocado dinamicamente, visto que o tamanho do vetor é revelado apenas em tempo de execução. Após fazer a leitura desses valores, o sistema chama a função **`determinaQuebras`**, que retorna o número de quebras (v[i] > v[i + 1]) do vetor original, e imprime: O tamanho, a semente e o número de quebras.

Em seguida, o sistema começa o processo de calibragem, chamando a função **`determinaLimiarParticao,`** que retorna o tamanho máximo de vetor para qual o Insertion Sort é mais eficiente que o Quicksort na ordenação. Essa função instancia um Stat e um vetor de CostStat (Alocados estaticamente, para facilitar a gestão de memória), e começa a refinagem de valores de partição. Esse processo é baseado no cálculo do custo de 6-10 valores de partição por iteração, começando pelo intervalo [0, tamanho]. Esse cálculo é feito chamando a função **`ordenadorUniversal,`** que ordena uma cópia do vetor, utilizando o QuickSort, e armazena as estatísticas da ordenação no Stat. Em seguida, a função **`registraEstatisticas`** registra os custos referentes a cada partição em uma posição do vetor de CostStat.

O cálculo de custo é feito pela função **`calculaCusto,`** que usa os parâmetros dados na entrada como pesos para a seguinte função:

$[f = a * cmp + b * move + c * calls]$, sendo {a,b,c} o conjunto de pesos e {cmp, move, calls} as estatísticas de uma execução de algoritmo.

Os custos são impressos na tela pela função **`imprimeEstatisticas`** e, ao final de 6-10 iterações dessas (Pode variar, dependendo do tamanho da faixa), a partição que gerou o menor custo, dentro do intervalo, é impressa na tela e escolhida como referência para o cálculo da nova faixa de valores, calculada pela função **`calculaNovaFaixa.`** Esse processo se repete até que o custo de alguma das partições seja menor que a limiar de custo, dada na entrada do programa, ou que restem menos de 5 números na faixa de partições. O valor ideal de partição é retornado na função.

Após a conclusão da primeira metade do problema, é necessário ordenar o vetor, para posteriormente testá-lo com número de quebras específicos. Para isso, chamamos o **`ordenadorUniversal,`** que já ordena o vetor com a limiar de partição calculada, ou seja, já faz o uso do Insertion Sort para vetores suficientemente pequenos.

Após a ordenação, a segunda fase é iniciada com a chamada da função **determinaLimiarQuebra,** que determina o número máximo de quebras em um vetor a partir do qual é mais eficiente utilizar o Insertion Sort ao invés do QuickSort. A função faz uma varredura semelhante à usada para a parte 1, mas com alguns ajustes.

Nesse caso, é necessário criar um vetor de CostStat para cada algoritmo, visto que cada valor de quebra é testado para ambos os algoritmos. Sendo assim, para cada valor de quebra, fazemos duas cópias do vetor ordenado, sendo uma para cada algoritmo, e embaralhamos separadamente com o número de quebras que queremos, usando a função **shuffleVector** (Utilizei a mesma implementação usada na documentação do trabalho).

A reprodutibilidade do experimento é garantida ao usar a função **srand48** duas vezes com a mesma semente, para gerar o valor que nutre a função de embaralhamento.

Após obter dois vetores idênticos e embaralhados com o número alvo de quebras, o primeiro vetor é ordenado usando o Quicksort, e o segundo usando o Insertion Sort. Para cada execução, é chamada a função **registraEstatisticas,** que novamente calcula o custo, chamando a função **calculaCusto,** e registra esse valor no vetor CostStat correspondente ao algoritmo usado. Nesse caso, a impressão das estatísticas foi feita diretamente na função, visto que possui alguns detalhes que a diferenciam da função de impressão original.

Após 6-10 iterações dessas, o valor de quebra que gera a menor diferença entre as execuções dos dois algoritmos de ordenação é impresso no terminal e escolhido como referência para a próxima faixa de valores, calculada pela função `calculaNovaFaixa`. Esse processo se repete até que a diferença de custo entre as execuções de Insertion Sort dos limites da faixa seja menor que a limiar de custo, dada na entrada, ou que restem menos de 5 números na faixa. Ao final do processo, o valor ideal de limiar de quebras é retornado pela função.

Ao final dessas duas fases, o vetor é desalocado e o programa se encerra, concluindo tanto o cálculo da limiar de partição, quanto o cálculo da limiar de quebras.

É importante ressaltar que a lógica por trás das funções de varredura foi inspirada no pseudo-código contido nas instruções do trabalho. Além disso, o programa contém algumas funções auxiliares, que evitam a implementação repetitiva de processos. Dois exemplos são as funções **indiceMenorCusto** e **indiceMenorDiferenca,** que fazem a busca pelos índices de menor custo dentro dos vetores CostStat.

### 2.1 - Instruções de compilação e execução

Após extrair o conteúdo do .zip, entre no diretório extraído e execute o comando **make all.** Esse comando compilaŕa e fará a ligação de todos os arquivos.

Logo em seguida, adicione à raiz do projeto seu arquivo .txt que conterá a semente, limiar de custo, 3 coeficientes e o tamanho, seguido por tamanho * inteiros, sendo 1 por linha.

Finalmente, rode o comando: **make run FILE=arquivo.txt** ou
**./bin/tp1.out arquivo.txt**

## 3 . Análise de complexidade

**QuickSort** - Esse algoritmo, devido ao uso da mediana de três, para evitar o pior caso, tem complexidade de tempo $O(n \log n)$. Em relação ao espaço, é $O(\log n)$, devido à profundidade da recursão na pilha.

**Insertion Sort** - Esse algoritmo tem como complexidade de tempo $O(n^2)$, pois em média cada elemento inserido é comparado $n/2$ vezes, com pior caso em $n^2$. Em relação ao espaço, é $O(1)$, pois não aloca nenhuma memória extra.

**`determinaQuebras`** - Esse algoritmo tem complexidade de tempo $O(n)$, pois além de operações constantes, contém um loop que percorre uma vez o vetor de tamanho $n$. A complexidade de espaço é $O(1)$, pois não aloca memória extra.

**`registraEstatisticas`** - Esse algoritmo tem complexidade de tempo $O(1)$, pois acessa o vetor em $O(1)$ e define o valor de $2$ variáveis. Em relação à espaço, também é $O(1)$, não aloca memória.

**`imprimeEstatisticas`** - Assim como o registro, a impressão tem complexidade de tempo $O(1)$, pois apenas faz uma impressão, com acesso direto. Em relação à espaço, também é $O(1)$, não aloca memória.

**`calculaCusto`** - Essa função tem complexidade de tempo $O(1)$, pois apenas faz 3 multiplicações e retorna a soma delas. Complexidade de espaço também é $O(1)$.

**`calculaNovaFaixa`** - Essa função não contém nenhum loop e acessa os campos do vetor de forma direta, através dos índices. Além disso, chama funções auxiliares $O(1)$. Portanto, também tem complexidade de tempo $O(1)$. Não aloca memória extra, portanto é $O(1)$ em relação à espaço.

**`ordenadorUniversal -`** Esse algoritmo pode seguir duas rotas: Execução do QuickSort ou do Insertion Sort. Sendo assim, para k = número de quebras:

- Para $k < limiar\ de\ quebras$:
  Tempo = $O(k * n/2) = O(n^2)$. Espaço = $O(1)$ (Usa Insertion).

- Para $n < limiar\ de\ partição$:
  Tempo = $O(n^2)$. Espaço = $O(1)$ (Usa Insertion).

- Para n > limiar de partição:
  Tempo = $O(n \log n)$. Espaço = $O(\log n)$ (Usa QuickSort).

**`determinaLimiarParticao - Tempo:`** Essa função, além de conter várias funções auxiliares de custo constante, contém dois loops que pesam na sua complexidade assintótica. O primeiro depende da limiar de custo, que pode variar bastante, mas vamos considerar o pior caso de execução, que é quando sobram menos de 5 números na faixa. Sendo assim, no pior caso, como a cada divisão por 5 selecionamos ⅘ do vetor para a próxima iteração, a complexidade do "while' é $O(\log\_2.5\ (n))$. O segundo loop, o "for", tem algumas funções que formam sua complexidade. A primeira delas é a cópia dos vetores, que é $O(n)$. A segunda é a ordenação do vetor, feita através da função **`ordenadorUniversal,`** na configuração do QuickSort, que possui complexidade $O(n \log n)$. Portanto, como o "for" sempre vai rodar um número $k$ de vezes, sendo $k$ uma constante entre 6 e 10, a complexidade desse loop é $O(n + n \log n)$. Sendo assim, a complexidade assintótica de tempo dessa função é $O((\log\_2.5\ (n)) * (n \log n + n))$.

**`Espaço:`** A função aloca dinamicamente um vetor de cópia, de tamanho n, para cada iteração. Como esse vetor é desalocado ao fim de todas as iterações, a complexidade de espaço é $O(n)$.

**`determinaLimiarQuebra - Tempo:`** Assim como a última função, essa é formada por dois loops principais. O primeiro while segue a mesma lógica da função supracitada, portanto tem complexidade assintótica $O(\log\_2.5\ (n))$. Já o "for" também roda k vezes, sendo k uma constante entre 6 e 10, e apesar de fazer duas cópias de vetor, $O(2n) = O(n)$, então tem um começo semelhante. A diferença é que nessa função, para cada iteração, chamamos o QuickSort e o Insertion Sort, que juntas geram uma complexidade assintótica $O(n^2 + n \log n)$. Sendo assim, a complexidade assintótica de tempo dessa função é $O(\log\_2.5\ (n)) * (n^2 + n \log n + n))$.

**`Espaço:`** A função aloca dinamicamente dois vetores de cópia, de tamanho n, para cada iteração. Como esses vetores são desalocados ao fim de todas as iterações, a complexidade de espaço é $O(2n) = O(n)$.

**`shuffleVector -`** Em relação à tempo, essa função gera n números de quebras em um determinado vetor, portanto, sua complexidade é $O(n)$, sendo n o número de quebras desejado. Em relação à espaço, é $O(1)$, pois não aloca nenhuma memória extra.

**`indiceMenorCusto -`** Esse algoritmo possui um loop interno que roda de 0 até n. Portanto, sua complexidade assintótica em tempo é $O(n)$. Já a de espaço é $O(1)$, visto que ela não aloca nenhuma memória extra. Entretanto, no programa, a função é usada de forma que limita ela a complexidade de tempo $O(10) = O(1)$, que é o tamanho dos vetores passados como parâmetro.

**`indiceMenorDiferenca` -** Assim como o algoritmo acima, também possui um loop interno de 0 até n, e não aloca memória extra. Portanto é $O(n)$ para o tempo e $O(1)$ para espaço. Entretanto, no programa, a função é usada de forma que limita ela a complexidade de tempo $O(10) = O(1)$, que é o tamanho dos vetores passados como parâmetro.

**Complexidade Total -** Em geral, as funções que determinam a complexidade do programa são as que calculam as limiares, sobrepondo o resto das funções. Portanto a complexidade assintótica total do programa é:

$$
\begin{aligned}
& O(\log\_2.5\ (n)) * (n^2 + n \log n + n) + (\log\_2.5\ (n)) * (n \log n + n)) \\
= \quad & O(\log\_2.5(n) * (n^2 + 2n \log n + 2n))
\end{aligned}
$$

Em relação à espaço, é um programa eficiente no uso de memória, e tem complexidade $O(n)$, usada apenas para alocações dos vetores, além das variáveis locais e da pilha do QuickSort.

## 4 . Estratégias de Robustez

O sistema contém diversos mecanismos de segurança, contra falhas e entradas equivocadas, baseados nos conceitos da programação defensiva. Os principais gatilhos são: Verificações da entrada do vetor, que verifica se o tamanho e a limiar de custo são válidos; Verificação de existência do arquivo de entrada; Verificação do sucesso da abertura do arquivo; Verificação da alocação do vetor principal e dos vetores auxiliares; Liberação da memória de todos os vetores dinamicamente alocados.

Todas essas verificações, em caso de falha, retornam 1 (Código que indica falha na execução do programa). A implementação desses gatilhos é essencial para a identificação de possíveis falhas e para a proteção do sistema.

## 5 . Análise Experimental

Os dados utilizados na análise experimental foram obtidos utilizando a biblioteca `sys/time.h` e scripts em Python para gerar vários arquivos com configurações diferentes. Optei pelo seu uso devido à sua precisão à nível de microsegundos.

Utilizei arquivos com configurações diversas para obter os seguintes resultados, variando as diferentes propriedades do vetor, visando um resultado completo e sólido.

As medições de tempo foram feitas na execução do sistema completo, realizando a calibração das duas limiares e a impressão dos resultados.

### 5.1 - Regressão dos coeficientes

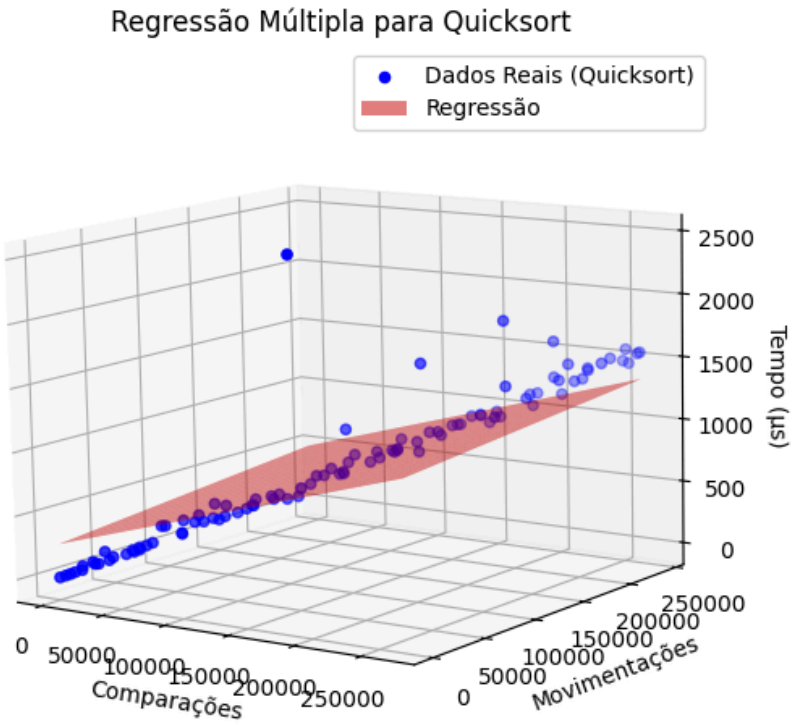


Figura 1: Relação entre Movimentações, Comparações e Tempo
(Chamadas foram fixadas para possibilitar a visualização em 3D).

Para chegar aos coeficientes ideais utilizei tamanhos diferentes de vetor como entrada (10-10000) para o QuickSort com uso de mediana de 3 e Insertion Sort para partições pequenas. Essa escolha foi feita visando equilibrar os resultados, visto que o algoritmo usa 2 métodos de ordenação. Em um arquivo .csv, fiz o registro do custo de tempo e o número de comparações, movimentações e chamadas. Após esse passo, fiz a regressão múltipla e a conclusão para o meu conjunto de dados foi a seguinte:

**a (cmp) = 0.003195;  b (move) = 0.000869; c (calls) = 0.952315;**

Diante desses valores é possível inferir que as chamadas são a operação que pesa mais no custo de tempo do QuickSort, nas configurações supracitadas.

Vale ressaltar que não utilizei nenhuma forma de *payload,* optei por manter o formato original das entradas.

## 5.2 - Relação Tempo x Tamanho do Vetor

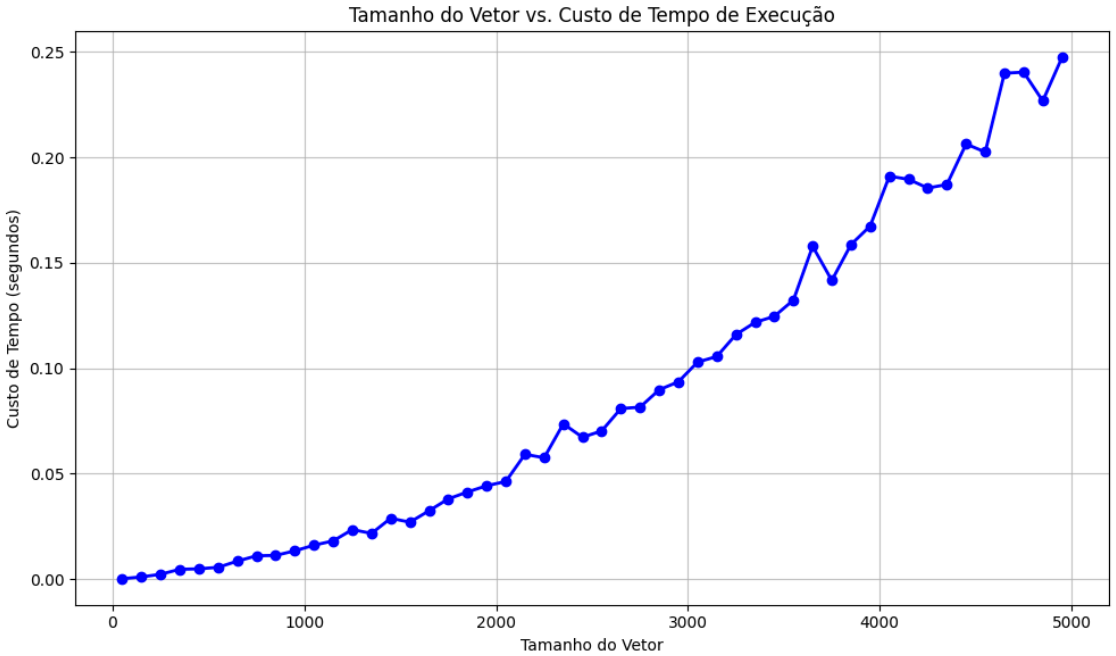


Figura 2: Relação entre Tempo de execução e Tamanho do vetor.

**Análise:**

No gráfico acima, é apresentada a relação entre o tamanho do vetor e o tempo de execução do algoritmo. A curva de crescimento da função mostra que o tempo de execução cresce de forma não linear, conforme o tamanho do vetor aumenta.

A relação não-linear entre tempo de execução e tamanho do vetor é uma das vias de demonstração da incorporação da complexidade assintótica dos algoritmos de ordenação na complexidade do sistema. Portanto, o gráfico mostra que a complexidade do sistema se aproxima de $O(\log\_2.5(n) * (n^2 + 2n \log n + 2n))$, como calculado anteriormente.

Como os algoritmos usados no sistema são otimizados, como o uso de mediana no QuickSort, e os vetores utilizados foram gerados de forma a manter uma configuração “média” de ordenação, é possível inferir que os piores casos foram evitados nas execuções das funções.

As pequenas oscilações entre algumas execuções são esperadas na execução de um conjunto de testes como esse.

### 5.3 - Relação Tempo x Configuração do Vetor

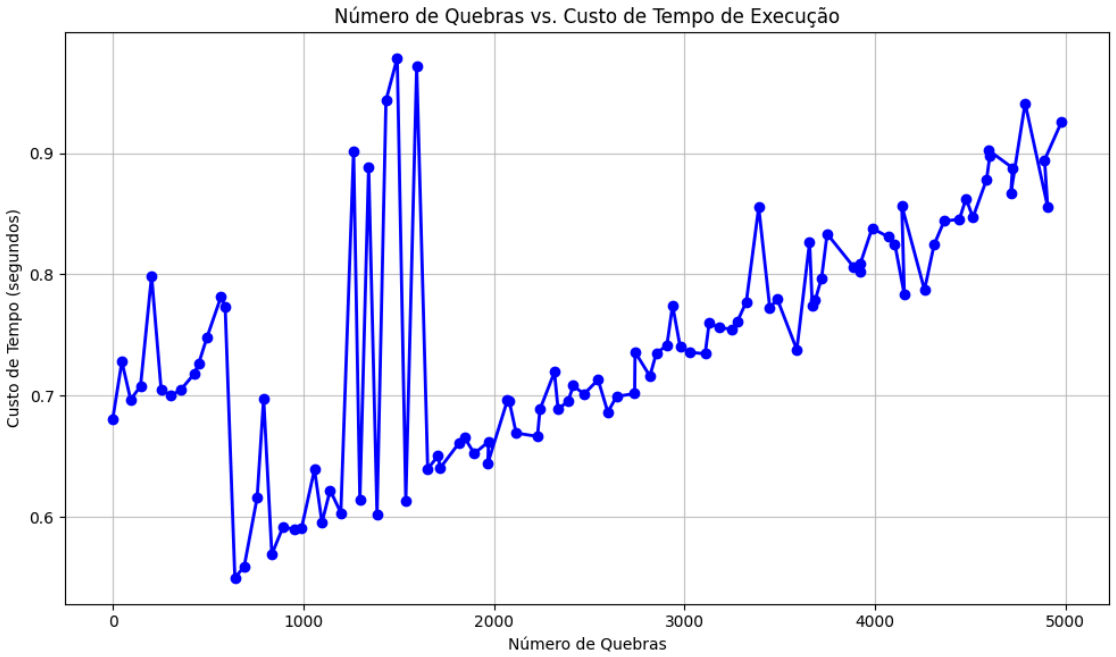


Figura 3: Relação entre o Tempo de Execução e o Número de Quebras.

**Análise:**

No gráfico acima, é apresentada a relação entre o número de quebras do vetor e o tempo de execução do algoritmo.

Os resultados foram diferentes do esperado, especialmente devido às inconsistências nos resultados para números de quebras contidos no intervalo (1000-1500). Para o experimento usei vetores de tamanho 10.000 que foram de 0 a 5000 quebras. Retirei o vetor inversamente ordenado do conjunto de dados, pois era um outlier diante desses resultados.

O crescimento do tempo de execução em função do número de quebras é linear. Um fator que pode explicar esse comportamento são as otimizações usadas nos algoritmos, que evitam os piores casos de execução.

Esse comportamento mostra que, mesmo em configurações não-ideais do vetor de entrada, o algoritmo de calibragem se comporta muito bem. Conclui-se que o gráfico demonstra o bom desempenho do sistema na execução de vetores desordenados, evidenciando a eficácia das estruturas de dados utilizadas na implementação.

## 6 . Conclusões

Nesse trabalho, utilizando os conhecimentos obtidos nas aulas de Estruturas de Dados, foi possível implementar e analisar um sistema de calibragem de limiares ideais para algoritmos de ordenação, visando otimizar a ordenação de vetores específicos.

Os resultados da análise experimental mostram a importância de sistemas que otimizem a ordenação de vetores, assim como o implementado nesse trabalho. É evidente que as peculiaridades de cada vetor afetam diretamente a forma de ordená-los, portanto, é vital que essa ordenação seja sempre a mais eficiente, especialmente se tratando de uma operação extremamente comum em diversas áreas da computação.

Nesse sentido, a eficácia do sistema implementado foi provada na análise 5.3, que demonstra o comportamento do algoritmo sob diferentes configurações de vetores no que diz respeito à ordenação.

Em conclusão, a realização do trabalho evidenciou  o impacto das decisões de implementação e do uso de diferentes Estruturas de Dados, no resultado final do projeto. Cada escolha impacta diretamente o funcionamento do produto final. Por fim, as maiores dificuldades enfrentadas foram a compreensão do enunciado e a análise experimental. Por ser a primeira vez lidando com uma documentação mais pesada, tive que ler o enunciado várias vezes até entender o trabalho e o processo da análise. Além disso, a necessidade do uso de boas práticas, especialmente em projetos maiores, foi muito evidente. Lidar com problemas em um projeto bem estruturado é infinitamente mais simples que em projetos desorganizados, sem um padrão de desenvolvimento.

## 7 . Referências

Meira, W. and Lacerda, A. (2025). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Meira, W. (2025). Instruções para a realização do Trabalho Prático 1 de Estrutura de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.