Trabalho Prático 1

Ordenador Universal

Thiago Henrique Silva de Almeida

2024014180

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

thiagohenriquesilva@ufmg.br

1. Introdução

A ordenação de dados é uma das tarefas mais fundamentais e recorrentes na Ciência da Computação, sendo crucial para o desempenho de diversos sistemas e algoritmos. Diante da variedade de algoritmos de ordenação existentes e da diversidade de características dos dados de entrada, torna-se evidente que nenhuma única abordagem é universalmente ótima. Nesse contexto, surge a proposta de desenvolvimento de um Tipo Abstrato de Dados (TAD) denominado Ordenador Universal, idealizado pela empresa Zambs, com o objetivo de selecionar dinamicamente o algoritmo de ordenação mais eficiente para um dado vetor.

Este trabalho consiste na implementação de uma prova de valor do TAD Ordenador Universal, capaz de decidir, de forma autônoma, entre os algoritmos de Inserção e Quicksort, com base em critérios empíricos relacionados ao vetor a ser ordenado. A seleção leva em consideração dois aspectos principais: (i) o grau de ordenação prévio do vetor, medido pelo número de quebras (posições em que um elemento é menor que seu antecessor) e (ii) o tamanho da partição mínima a partir da qual o uso de Quicksort se justifica.

A lógica de escolha do algoritmo está fundamentada em dados experimentais. Para isso, são medidas as estatísticas de execução dos algoritmos, como o número de comparações, movimentações e chamadas recursivas. Esses dados são então utilizados em uma função de custo parametrizada pelos coeficientes a, b e c, que refletem o impacto relativo de cada uma das estatísticas na plataforma de execução.

A determinação dos limiares ideais (número máximo de quebras e tamanho mínimo de partição) é feita de forma iterativa, por meio de estratégias de varredura e refinamento de faixa. Esses procedimentos buscam minimizar o custo total de ordenação previsto pela função empírica mencionada.

Este trabalho, portanto, além de implementar os algoritmos e mecanismos de análise, também propõe uma abordagem prática para ajuste fino dos parâmetros de decisão do Ordenador Universal, contribuindo para a criação de um TAD inteligente, adaptável a diferentes cenários e dados de entrada.

2. Método

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

A implementação do sistema foi organizada de forma modular, explorando o paradigma de Tipos Abstratos de Dados (TADs) para representar os principais componentes do sistema. A seguir, descrevem-se as principais estruturas de dados, classes e funções implementadas.

2.1 Estruturas

2.1.1 Estrutura item_t

Todos os algoritmos de ordenação trabalham com vetores do tipo item_t, definido como uma estrutura com 2 atributos, que são a key (Campo utilizado como chave de ordenação) e a payload (Vetor de caracteres que simula o peso do conteúdo referenciado pela chave). Essa estrutura permite representar os dados de forma genérica e compatível com diferentes algoritmos.

2.1.2 Estrutura estatisticas

Estrutura responsável por armazenar o número de comparações, movimentações e chamadas de função durante todas as várias ordenações realizadas pelo algoritmo.

2.2 Classes e TADs

2.2.1 Classe OrdenadorUniversal

A classe OrdenadorUniversal foi projetada para realizar uma ordenação adaptativa, ou seja, sua principal finalidade é decidir qual algoritmo de ordenação empregar, entre Insertion Sort e QuickSort, com base em características do vetor de entrada e no custo estimado de cada operação. Para isso, ela combina estratégias de análise do vetor e uma abordagem empírica de testes para calcular limiares ideais que norteiam suas decisões.

Ao ser instanciada, a classe recebe o vetor original, seu tamanho, os pesos para as métricas de custo (comparações, movimentações e chamadas recursivas) e uma semente aleatória que será usada para controle de embaralhamento em testes posteriores. Ela armazena essas informações e computa, imediatamente, o número de quebras no vetor. A ideia de "quebra" refere-se a transições de elementos fora de ordem, o que serve como uma métrica simples da desordem do vetor.

A partir dessas informações, o comportamento central da classe gira em torno de dois processos distintos, mas complementares: a determinação do limiar de partição e a determinação do limiar de quebra.

No processo de cálculo do limiar de partição (método determinaLimiarParticao), o objetivo é descobrir qual valor mínimo de partição torna o QuickSort mais eficiente. Isso é importante porque, em vetores menores ou altamente fragmentados, o custo recursivo do QuickSort pode

ser maior que o benefício de sua velocidade assintótica. Para encontrar esse valor ideal, a classe executa múltiplas ordenações variando o tamanho da partição mínima. Em cada uma dessas execuções, o vetor é copiado, o algoritmo é aplicado e são coletadas estatísticas de desempenho. Com base nos pesos definidos para comparações, movimentações e chamadas recursivas, o custo total da execução é calculado. Após isso, a faixa de partições é ajustada (método calculaNovaFaixa), restringindo-se à região mais próxima de onde foi observado o menor custo, e o processo se repete até que a diferença entre os melhores custos seja suficientemente pequena. Esse refinamento progressivo busca encontrar o ponto ótimo de desempenho do QuickSort para aquele vetor específico.

Já na determinação do limiar de quebra (método determinaLimiarQuebra), o foco muda. Agora, busca-se entender em que ponto o número de quebras em um vetor torna vantajoso abandonar o Insertion Sort e preferir o QuickSort. Para isso, a classe gera artificialmente diferentes quantidades de quebras no vetor original (método adicionaQuebras), utilizando a semente aleatória para manter reprodutibilidade, e os submete aos dois algoritmos de ordenação. Os custos são novamente registrados e comparados. A análise não busca o menor custo absoluto, mas sim o ponto em que a diferença de desempenho entre os dois algoritmos é a menor possível. Isso indica uma "zona de transição" em que qualquer um dos dois poderia ser eficiente, e a partir desse ponto é seguro assumir que, ao aumentar a desordem, o QuickSort se tornará sempre mais vantajoso.

Ambos os limiares calculados, de partição e de quebra, são fundamentais para que o método escolheAlgoritmo tome sua decisão. Este método verifica se o número de quebras no vetor é inferior ao limiar calculado e, nesse caso, opta pelo Insertion Sort, considerando que o vetor está suficientemente ordenado. Caso contrário, verifica o tamanho do vetor e se ele for maior que o limiar de partição, o QuickSort é utilizado, ou caso contrário, volta-se novamente ao Insertion Sort.

Dessa forma, a OrdenadorUniversal se adapta dinamicamente ao conteúdo do vetor, utilizando tanto heurísticas estáticas (como o número de quebras) quanto uma análise experimental baseada em medições reais de desempenho. Com isso, ela consegue entregar uma estratégia de ordenação robusta e otimizada, adaptada às características do conjunto de dados fornecido, minimizando o custo de execução de acordo com os critérios definidos.

2.3 Módulos

2.3.1 Módulo AlgoritmosOrdenacao

Este módulo reúne as implementações dos algoritmos de ordenação que serão utilizados pela classe OrdenadorUniversal, além de funções auxiliares fundamentais para o acompanhamento estatístico das operações e manipulação do vetor. Ele se apoia fortemente em uma estrutura de contadores de operações, a fim de registrar o custo computacional de cada algoritmo de forma precisa e comparável.

No início, são definidas três funções que manipulam a estrutura estatisticas, responsável por armazenar o número de comparações (cmp), movimentações (move) e chamadas recursivas (calls). A função resetcounter zera todos esses contadores, preparando o ambiente para uma

nova execução limpa. Já as funções inccmp, incmove e inccalls permitem incrementar individualmente cada um dos contadores durante a execução dos algoritmos.

A função mediana serve como critério de escolha do pivô no QuickSort. Ela recebe três elementos e retorna aquele que estiver no meio em termos de ordenação, o que ajuda a evitar casos patológicos de desempenho ao selecionar pivôs extremos. Para realizar a comparação entre elementos do tipo item_t, esta função depende da sobrecarga do operador <=, implementada externamente.

A função insercao implementa o algoritmo de Insertion Sort no intervalo do vetor que vai de l até r. Para cada elemento a partir da posição l+1, ela busca sua posição correta na parte já ordenada do vetor, deslocando os elementos maiores à direita e inserindo o valor atual no local apropriado. Cada movimentação e comparação é contabilizada na estrutura estatisticas, e uma chamada é registrada no início da função.

Já a função particao executa o procedimento de particionamento necessário ao QuickSort. Ela calcula a mediana entre os extremos e o centro do vetor para escolher o pivô, e então avança os ponteiros i e j a partir das extremidades até que elementos fora de lugar sejam encontrados e trocados. Durante o processo, todas as comparações com o pivô e as trocas entre elementos são registradas nas estatísticas.

A função quickSort realiza a ordenação propriamente dita. Trata-se de uma implementação clássica do QuickSort com uma modificação estratégica: quando o tamanho de uma subpartição cai abaixo de um limiar (menorTamanhoParticao), a ordenação deixa de usar recursão e passa a utilizar o Insertion Sort. Isso é feito porque o Insertion Sort pode ser mais eficiente em vetores pequenos ou quase ordenados. A função também registra chamadas recursivas e atualiza as estatísticas conforme as chamadas a particao e insercao ocorrem.

Complementando o módulo, há a função embaralharVetor, que introduz aleatoriedade no vetor através de um número fixo de trocas aleatórias entre pares de elementos. Esse procedimento é importante para gerar vetores com diferentes graus de desordem e, portanto, diferentes números de quebras. A função é usada nos testes experimentais que avaliam o desempenho dos algoritmos sob diferentes condições.

Por fim, a função contaQuebras percorre o vetor do início até a penúltima posição e conta quantas vezes um elemento é maior que seu sucessor. O número de quebras assim identificado serve como uma métrica de desordem, fundamental para as decisões da OrdenadorUniversal.

Em conjunto, este módulo oferece todos os componentes de base necessários para a execução, avaliação e comparação dos algoritmos de ordenação usados no sistema. Ele integra lógica de ordenação, controle estatístico e ferramentas de preparação dos dados, permitindo análises detalhadas de desempenho em diferentes cenários.

3. Análise de Complexidade

A classe OrdenadorUniversal concentra a lógica de escolha adaptativa do algoritmo de ordenação com base em características do vetor, como o número de quebras e o tamanho das partições. Cada um de seus métodos contribui com uma etapa dessa análise, e suas complexidades podem ser avaliadas da seguinte forma:

3.1 escolheAlgoritmo

O método escolheAlgoritmo é o núcleo da funcionalidade adaptativa. Ele começa computando o número de quebras no vetor através da função contaQuebras, que tem complexidade O(n), onde n é o tamanho do vetor. Em seguida, avalia se o número de quebras no vetor ultrapassar o limiar de quebras passado como argumento, se não entrar nesse nó, alavia se o tamanho do vetor é menor que o menor tamanho de partição, também recebido como argumento. A complexidade de tempo do método escolheAlgoritmo é, portanto, dominada pela complexidade do algoritmo de ordenação selecionado para operar sobre a cópia do vetor. No pior caso, tanto o Insertion Sort quanto o Quicksort (mesmo com as otimizações de mediana de três e corte para Inserção) podem apresentar complexidade $O(n^2)$. No caso médio, o Quicksort híbrido oferece O(n log n), enquanto o Insertion Sort para vetores quase ordenados pode se aproximar de O(n). O espaço adicional utilizado é O(n) para a cópia do vetor, mais O(log n) para a pilha de recursão do Quicksort, se este for o escolhido (ou O(1) de espaço auxiliar para o Insertion Sort puro, desconsiderando a cópia).

3.2 determinaLimiarParticao

Na função determinaLimiarParticao, dentro do laço, para cada tamanho de partição testado, ela chama escolheAlgoritmo, que executa um Quicksort híbrido, ou seja, um Quicksort que, quando as partições ficam pequenas o suficiente, usa Insertion Sort para ordenar essas pequenas partições.

Sabemos que o Quicksort padrão tem complexidade média O(n log n) e, com o uso de Insertion Sort para partições pequenas, o desempenho prático melhora, mas a complexidade assintótica permanece O(n log n) na média. No pior caso, o Quicksort pode chegar a $O(n^2)$, mas isso é mitigado pelo uso da mediana de três como pivô e pelo corte para Insertion Sort em partições pequenas.

No cenário da função, em cada iteração do laço do-while, aproximadamente seis valores diferentes de tamanho de partição são testados (o laço varia tamanhoParticao de menorTamanhoParticao até maiorTamanhoParticao em passos de aproximadamente (maiorTamanhoParticao−menorTamanhoParticao)/5.

Cada chamada a escolheAlgoritmo executa o Quicksort híbrido sobre o vetor todo, com complexidade O(n log n). Como a função pode repetir o laço várias vezes (até o critério de convergência ser satisfeito), o tempo total pode ser visto como O(k * m * n log n), onde k é o número de iterações do do-while (normalmente pequeno, pois o intervalo é reduzido a cada iteração), $m \approx 6$ é o número de testes por iteração, n é o tamanho do vetor. Como k e m são constantes pequenas, a complexidade assintótica domina-se por O(n log n).

3.3 determinaLimiarQuebra

A função determinaLimiarQuebra realiza iterações onde testa diversos valores de limiar de quebra, compara os desempenhos de ambos os algoritmos e ajusta o intervalo de busca conforme os resultados. Em cada iteração do do-while, são feitos cerca de seis testes. Em cada teste, um vetor com quebras artificiais é gerado usando adicionaQuebras, que copia o vetor original com interrupções. Esse vetor é ordenado duas vezes: uma com Quicksort híbrido e outra com Insertion Sort. Como ambas as ordenações são executadas sobre o vetor completo de tamanho n, cada uma custa, no pior caso, $O(n^2)$, mas no caso médio, Quicksort híbrido mantém O(n log n) e Insertion $O(n^2)$. Como estamos comparando os dois algoritmos em diferentes limiares de quebra, o tempo da função depende da soma dessas execuções. Assumindo que o número de iterações e testes por iteração são limitados, a complexidade de tempo da função fica aproximadamente entre O(n log n) e $O(n^2)$, dependendo do comportamento real das quebras.

Quanto ao espaço, além de variáveis auxiliares e da pilha de recursão do Quicksort (que consome O(log n), a função aloca vetores temporários com adicionaQuebras, cuja complexidade de espaço é O(n). Portanto, a complexidade de espaço total da função é O(n).

Dessa forma, todos os métodos da classe OrdenadorUniversal que fazem análise empírica por meio de repetidas ordenações são custosos em tempo, mas ainda controlados, pois o número de testes é limitado e o tamanho do vetor é fixo. O uso de memória permanece modesto, já que não há estruturas auxiliares complexas além da pilha de chamadas recursivas nos algoritmos de ordenação. A inteligência do sistema está na capacidade de prever o comportamento dos algoritmos com base em métricas internas do vetor, o que permite adaptar a escolha do algoritmo de forma eficiente mesmo com um custo inicial de análise.

4. Estratégias de Robustez

As funções implementadas demonstram algumas estratégias de robustez importantes para garantir a confiabilidade e estabilidade do sistema. Uma das estratégias de robustez mais evidentes é a forma como os loops de calibração nas funções determinaLimiarParticao e determinaLimiarQuebra procedem. A condição de continuidade do laço do-while, que inclui (indexadorTeste >= 5) (ou numMPS > 5 na especificação original do problema), assegura que o processo de refinamento da faixa de busca só prossiga se um número mínimo de pontos de teste distintos (tipicamente 5 ou 6) puderam ser avaliados na iteração corrente. Isso garante que a decisão de continuar a busca iterativa (caso a diferencaCusto ainda esteja acima do limiarCusto) seja baseada em uma quantidade razoável de dados experimentais, reduzindo a chance de o algoritmo convergir prematuramente ou de forma instável devido a ruídos em poucas medições. Se a faixa se tornar tão pequena que não seja possível gerar pelo menos 5 pontos de teste distintos, o laço é interrompido, evitando iterações improdutivas e loops infinitos. Essa abordagem contribui para a estabilidade e a confiabilidade do processo de calibração dos limiares.

Na função determinaLimiarQuebra, observamos o uso de alocação dinâmica por meio da função adicionaQuebras, que retorna um vetor alocado com new item_t. O uso de delete para liberar corretamente essa memória após o uso foi identificado como uma necessidade crítica e constitui um bom exemplo de gerenciamento de memória. O uso adequado de new e delete evita vazamentos de memória, especialmente importante em loops onde múltiplas alocações ocorrem sucessivamente.

O módulo de validação de arquivos implementa uma série de verificações rigorosas quanto ao formato dos dados de entrada. Com o uso da função lerIntEstrito e leitura sequencial controlada com std::istringstream, o programa garante que os dados estejam exatamente no formato esperado: Verifica se há o número exato de inteiros e doubles necessários, garante que os dados estejam estritamente formatados e sem sobras no fluxo, verifica se o vetor possui exatamente o número de elementos esperado e detecta dados extras ao final do arquivo.

Essas medidas evitam que o sistema funcione com configurações incorretas ou dados inconsistentes, reforçando a robustez e a confiabilidade da execução.

5. Análise Experimental

A análise experimental foi conduzida inicialmente fixando-se os parâmetros KEYSZ em 5 e PLSZ em 10, o que define o tamanho da chave e da carga útil de cada item do vetor a ser ordenado. Com essa configuração, vetores foram gerados com diferentes valores de VETSZ, variando de 10 até 10 milhões de elementos. Todos os vetores foram preenchidos de forma aleatória utilizando uma semente fixa no gerador de números pseudoaleatórios, garantindo a reprodutibilidade dos testes.

Para cada tamanho de vetor, foi executado o algoritmo QuickSort com um limiar fixo para a substituição por InsertionSort, e foram registradas as métricas de desempenho, sendo o número de chamadas recursivas (calls), o número de comparações (comp), o número de movimentações de elementos (moves) e o tempo real de execução. Esses dados foram reunidos em uma tabela com sete instâncias representando diferentes escalas do problema.

Com esses resultados, aplicou-se uma regressão linear múltipla tomando os valores de chamadas, comparações e movimentações como variáveis independentes e o tempo de execução como variável dependente. O modelo resultante forneceu três coeficientes, denominados a, b e c que representam o peso relativo de cada tipo de operação no custo computacional total. O coeficiente a correspondeu ao impacto das comparações, b ao impacto das movimentações e c ao impacto das chamadas.

Os coeficientes obtidos pela regressão foram então utilizados para calibrar o TAD Ordenador Universal, servindo como base para o cálculo dos dois parâmetros centrais de sua lógica adaptativa: o limiarParticao e o limiarQuebra. Assim, a função de custo derivada da regressão não é apenas utilizada para avaliar o desempenho, mas para guiar dinamicamente o comportamento do algoritmo durante a ordenação.

Com os limiares ajustados automaticamente a partir da função de custo calibrada, foi possível realizar a segunda fase da análise experimental, que comparou diretamente três abordagens, o

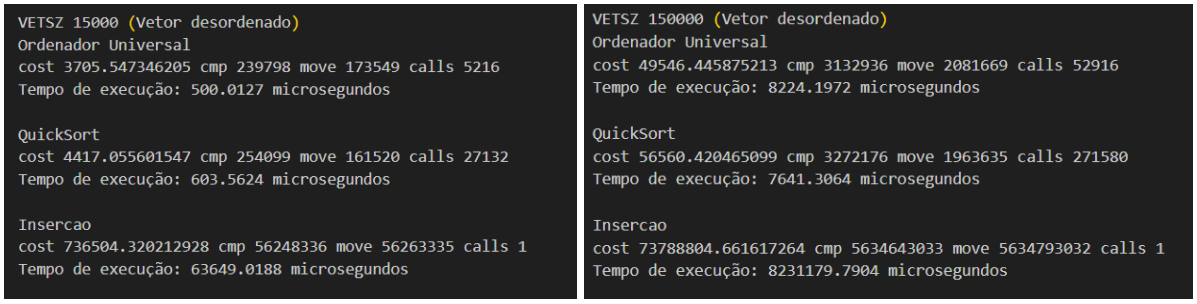
Ordenador Universal, o QuickSort puro e o InsertionSort puro. Essa comparação foi feita em três condições distintas do vetor de entrada: desordenado aleatoriamente, completamente ordenado e ordenado em ordem inversa. Para cada cenário e para cada tamanho de vetor, foram novamente medidas as mesmas métricas de custo e tempo de execução, permitindo avaliar o desempenho relativo de cada algoritmo.

Por fim, todo o processo foi repetido com novos parâmetros para os itens: KEYSZ foi aumentado para 10 e PLSZ para 100, o que elevou o custo de movimentação de dados e permitiu observar como o desempenho dos algoritmos se comportava sob maior carga de memória. Esse segundo conjunto de experimentos reforçou os resultados obtidos anteriormente e demonstrou a robustez da estratégia adaptativa do Ordenador Universal.

5.1 Vetores Desordenados

Para vetores aleatoriamente desordenados, o Ordenador Universal apresenta desempenho consistentemente melhor que o InsertionSort e ligeiramente superior ao QuickSort puro em estatísticas. Isso se dá porque ele identifica que o cenário favorece algoritmos mais eficientes em grandes volumes, como o QuickSort (com Inserção em partições pequenas), e o aplica seletivamente, como se pôde observar em:

Testes com KESZ = 10 e PLSZ = 100



Figuras 1 e 2 – Desempenho dos algoritmos em vetores desordenados (Imagem gerada pelo autor)

5.2 Vetores Ordenados

Neste caso, o InsertionSort torna-se altamente eficiente, pois ele realiza pouquíssimas comparações em vetores já ordenados. O Ordenador Universal, ao detectar o padrão, também opta pelo InsertionSort nesses casos, o que o torna tão eficiente quanto o melhor algoritmo possível, dentre os analisados, para esse cenário.

As estatísticas de execução do Ordenador Universal são iguais às do InsertionSort e muito inferiores às QuickSort, que realiza partições desnecessárias e mais comparações, o número de calls, cmps e moves comprova que o Ordenador Universal teve a mesma eficiência do Insertion Sort para vetores ordenados, que espera um total de tamanho-1 comparações, e 2*tamanho - 2 movimentações, devido a essa implementação específica do Insertion Sort, que faz 1 move ao copiar v[i] para aux e 1 move ao copiar aux de volta para v[j+1] (que é o mesmo lugar onde já estava), como se pôde observar em:

Testes com KEYSZ = 5 e PLSZ = 10

```
VETSZ 1500 (Vetor ordenado)
Ordenador Universal
cost 70.687173882 cmp 1499 move 2998 calls 1
Tempo de execução: 1.9996 microsegundos

QuickSort
cost 687.171278481 cmp 14940 move 2964 calls 1976
Tempo de execução: 7.7888 microsegundos

Insercao
cost 70.687173882 cmp 1499 move 2998 calls 1
Tempo de execução: 2.1108 microsegundos
```

```
VETSZ 15000 (Vetor ordenado)
Ordenador Universal
cost 707.306633080 cmp 14999 move 29998 calls 1
Tempo de execução: 93.6608 microsegundos

QuickSort
cost 9067.878397655 cmp 197086 move 26694 calls 17680
Tempo de execução: 156.2929 microsegundos

Insercao
cost 707.306633080 cmp 14999 move 29998 calls 1
Tempo de execução: 71.9659 microsegundos
```

Figuras 3 e 4 – Desempenho dos algoritmos em vetores ordenados (Imagem gerada pelo autor)

5.3 Vetores Inversamente Ordenados

Esse é um cenário desfavorável ao InsertionSort, pois ele exige o maior número possível de movimentações e comparações. Isso é refletido no custo astronômico do InsertionSort nesses casos. O Ordenador Universal, mais uma vez, detecta a ineficiência e evita o uso do InsertionSort. Ele usa QuickSort com ajustes, resultando em desempenho melhor do que QuickSort puro, com menos comparações e custo geral menor. As estatísticas do Ordenador Universal são melhores que as do InsertionSort (várias ordens de magnitude) e ligeiramente melhores que do QuickSort, como se pôde observar em:

Testes com KEYSZ = 10 e PLSZ = 100

```
VETSZ 150000 (Vetor inversamente ordenado)
Ordenador Universal
cost 42325.178184269 cmp 2217260 move 508616 calls 49150
Tempo de execução: 3231.6006 microsegundos

QuickSort
cost 49251.900862037 cmp 2456823 move 478395 calls 168932
Tempo de execução: 4301.7387 microsegundos

Insercao
cost 147327127.279572666 cmp 11250074997 move 11250224996 calls 1
Tempo de execução: 17806809.9280 microsegundos
```

```
VETSZ 15000 (Vetor inversamente ordenado)
Ordenador Universal
cost 2999.311943313 cmp 162972 move 51474 calls 3070
Tempo de execução: 334.7153 microsegundos

QuickSort
cost 3887.734347392 cmp 195024 move 47070 calls 16382
Tempo de execução: 267.8479 microsegundos

Insercao
cost 1473260.549374524 cmp 112507499 move 112522498 calls 1
Tempo de execução: 131375.8113 microsegundos
```

Figuras 5 e 6 – Desempenho dos algoritmos em vetores inversamente ordenados (Imagem gerada pelo autor)

Também foi possível observar a diferença de performance em vetores de mesmo tamanho, com mesmo número de quebras, mas com KEYSZ e PLSZ diferentes:

Testes com KESZ = 5 e PLSZ = 10

```
VETSZ 150000 (Vetor ordenado)
Ordenador Universal
cost 7073.501225056 cmp 149999 move 299998 calls 1
Tempo de execução: 807.6289 microsegundos

QuickSort
cost 117185.114751284 cmp 2546085 move 418131 calls 228928
Tempo de execução: 1469.5511 microsegundos

Insercao
cost 7073.501225056 cmp 149999 move 299998 calls 1
Tempo de execução: 256.0331 microsegundos
```

Testes com KESZ = 10 e PLSZ = 100

```
VETSZ 150000 (Vetor ordenado)
Ordenador Universal
cost 862.936157822 cmp 149999 move 299998 calls 1
Tempo de execução: 1758.6198 microsegundos

QuickSort
cost 50903.677920772 cmp 2456806 move 253395 calls 168930
Tempo de execução: 4314.3586 microsegundos

Insercao
cost 862.936157822 cmp 149999 move 299998 calls 1
Tempo de execução: 1843.4003 microsegundos
```

Figuras 7 e 8 – Desempenho dos algoritmos em vetores idênticos, mas dados de pesos diferentes (Imagem gerada pelo autor)

O que evidencia ainda mais o poder de algoritmos eficientes, já que o tempo de execução cresce muito de acordo com o conteúdo que se deve movimentar e com as chaves que se deve comparar.

6. Conclusão

A realização deste trabalho proporcionou uma compreensão muito mais concreta e tangível sobre o impacto das escolhas algorítmicas na prática. Ao partir de uma análise teórica e evoluir para uma implementação experimental minuciosa, foi possível observar diretamente como diferentes algoritmos de ordenação respondem a variações no tamanho, ordenação e estrutura dos dados. Mais do que números, os experimentos proporcionaram a experiência de visualizar um algoritmo levando microsegundos para ordenar vetores, enquanto outro, ao ser aplicado aos mesmos dados, exigia milhões de comparações e minutos de execução, o que foi essencial para consolidar o entendimento sobre como e por que algoritmos se comportam de maneiras tão distintas. Essa diferença, muitas vezes diluída em representações assintóticas como $O(n \log n)$ ou $O(n^2)$, torna-se concreta ao ver a máquina responder de forma tão discrepante. Uma vivência que vai além da abstração matemática da complexidade e revela, de fato, o custo computacional das escolhas de abordagem.

Nesse contexto, o Ordenador Universal se mostrou especialmente relevante. Sua capacidade de adaptar-se automaticamente às características do vetor, ajustando dinamicamente os parâmetros de limiar de partição e limiar de quebras com base em uma função de custo calibrada experimentalmente, permitiu conciliar os pontos fortes do QuickSort e do InsertionSort. Em cenários onde um algoritmo puro seria ineficiente em termos de desempenho, o Ordenador Universal manteve sua eficiência ao reconhecer os limites e transitar entre as abordagens. Essa flexibilidade é o que o torna verdadeiramente poderoso, ele não apenas executa uma ordenação, mas decide inteligentemente como fazê-la da forma mais eficaz possível, o que reforça a importância de se aliar teoria, prática e adaptabilidade na construção de soluções tecnológicas robustas.

Bibliografia

Cunha, Ferreira, Lacerda e Meira Jr. (2025). Slides virtuais da disciplina de Estruturas de Dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.