Trabalho Prático 1 – Métodos de ordenação

Daniel Augusto Castro de Paula – 2022060550

1. Introdução

O trabalho proposto consiste em analisar diferentes métodos de ordenação estudados ao longo da disciplina de Estrutura de Dados. Por meio de análises teóricas comparativas e de complexidade experimental, o principal objetivo é contrastar essas duas análises, identificando os melhores casos de uso para cada um dos algoritmos.

2. Método

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

2.1 Arquivos

O experimento foi desenvolvido em torno de dois arquivos principais chamados tp1.c e tp1v2.c na qual no primeiro foram implementados todos os algoritmos de ordenação para comparação exclusiva de um vetor de inteiros, já no segunda houve uma pequena adaptação nesses métodos para realizar a ordenação de uma estrutura chamada item que é composta por uma chave inteira e um valor que é um conjunto de char. Além disso, foi desenvolvido um Makefile para automatizar a chamada deste programa, passando diferentes parâmetros das cargas de trabalho a serem analisadas e imprimindo todos os resultados necessários no terminal.

2.2 Funções e TADS

Além das funções para implementação dos algoritmos de ordenação foram desenvolvidas outras funções para aumentar a eficiência e modularização do experimento. Entre essas funções, destaca-se a função "swap", que recebe os endereços de memória de dois elementos e realiza a troca de seus valores. A função "reverseV" inverte a ordem dos elementos de um vetor e a função "findMax" que retorna o maior valor inteiro presente em um vetor.

Além disso, a função "generateRandomV" é responsável por gerar um vetor aleatório respeitando alguns parâmetros específicos, permitindo especificar o tamanho do vetor, se ele deve estar ordenado ou não, e o tipo de ordenação desejada (ascendente ou descendente).Para a geração desses elementos, foi utilizada uma seed fixa, garantindo a reprodutibilidade dos resultados. No arquivo tp1v2.c, há também um tad chamado "item", composto por uma chave inteira e um valor de char com 99 bytes. Por fim, a função "main" gerencia as chamadas dos métodos e controla o tempo de execução, além de receber os parâmetros das cargas de trabalho e chamar a função de ordenação escolhida.

2.3 Cargas de trabalhos

As cargas de trabalho evidenciadas durante o experimento consistem na análise da variação do tamanho do vetor que estamos ordenando, para o experimento em questão utilizou-se vetores de cinco mil a quarenta mil elementos, aumentando o range de cinco em cinco mil. Também foi analisada a variação no modo da ordenação desse vetor, incluindo três casos: onde ele pode estar ordenado decrescentemente, ordenado ascendente ou desordenado. Além disso, utilizamos uma estrutura com valor de 99 bytes para comparar o tempo dos diferentes algoritmos de ordenação, procurando evidenciar aqueles que realizam mais trocas e portanto apresentariam maior custo para esse teste.

3. Análise de Complexidade

BubbleSort: A implementação canônica do Bubble Sort, sem mecanismos de otimização significativos, consiste em um algoritmo que itera diversas vezes sobre o array, comparando elementos adjacentes e trocando-os de posição para a esquerda quando o elemento da direita é menor. Este processo se repete até que todos os elementos estejam ordenados.

O número de comparações no Bubble Sort é representado pela soma de uma série aritmética, onde n representa o número de elementos do array. Durante cada iteração, o número de comparações diminui, resultando na seguinte soma:

$$C(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$$

Em relação ao número de trocas realizadas divide-se a análise em dois casos extremos de melhor caso O(1) no qual o vetor já está ordenado e não realizamos trocas e o pior caso O(n²) no qual o vetor está inversamente ordenado e realizamos o número máximo de trocas possíveis.

Na análise de espaço dizemos que é um algoritmo O(1) pois ele não requer espaço adicional para armazenar elementos durante a ordenação.

InsertionSort: Na forma canônica compara o elemento atual com os elementos anteriores já ordenados e o insere na posição correta, repetindo o processo até ordenar todos os elementos. Para análise de complexidade de comparações pode-se analisar o melhor caso onde o vetor já está ordenado e por isso sempre vamos interromper o loop interno logo na primeira comparação realizando no total apenas n-1 comparações para percorrer todo o vetor

$$C(n) = \sum_{k=1}^{n-1} 1 = n - 1 = O(n)$$

Em contrapartida temos o pior caso onde o vetor está inversamente ordenado no qual além de passar por todos elementos do vetor, precisamos percorrer também todo o subvetor trocando os elementos que pode ser representado pela seguinte equação:

$$C(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2} = O(n^2)$$

Nesses mesmos cenários podemos analisar o número de trocas dos elementos do vetor. Para o melhor caso fazemos 2(n-1) trocas já que nunca entramos no loop interno, → O(n). E para o pior caso fazemos 2(n-1) + n(n-1)/2 pois sempre entramos no loop interno → O(n²).

Em relação a complexidade de espaço ele também é O(1) já que os únicos espaços adicionais de memória que ele usa para auxiliar na lógica tem custo constante independente do vetor.

SelectionSort: O Selection Sort é um algoritmo de ordenação que divide o vetor principal em duas partes: a parte ordenada e a parte não ordenada. O algoritmo percorre o vetor selecionando o menor elemento da parte não ordenada e o insere no final da parte ordenada, sempre diminuindo o número de elementos que vai ser necessário percorrer já que um lado do vetor já estará ordenado. Em relação ao número de comparações percebe-se que sempre teremos um valor fixo que engloba uma progressão aritmética que vai diminuindo à medida que vamos ordenando o vetor. Nesse sentido temos a seguinte equação:

$$C(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$$

O número de movimentações pode ser contado apenas no loop externo que é 3(n-1) vezes → O(n).
A complexidade de espaço assim como os algoritmos anteriores é O(1) já que o espaço não depende do tamanho do vetor de entrada.

MergeSort: A implementação do mergeSort se deu a partir de duas funções, a MergeSort e a Merge, para que de maneira recursiva dividissem os vetores pela metade até que esses subvetores tenham apenas um elemento. Após isso vamos ordenando esses subvetores e os combinando em ordem crescente.

Para um elemento temos custo T(1) = 0 já que não faremos nada. Para mais de um elemento fazemos duas chamadas do Merge Sort para ordenar cada metade ficando com o custo de 2T(n/2). Além delas temos o custo da função Merge que percorre todos os dois subvetores para inserir no vetor resultado e por isso ele tem custo linear n.

Sendo assim temos a seguinte equação: T(n) = 2T(n/2) + n que pode ser resolvida pelo teorema mestre no qual a = 2, b = 2, f(n) = n e $n^{log_2 2}$ = n. Esse é o caso 2 do teorema $\theta(n^{log_2 2})$ = θ(n), portanto T(n) = θ(n log(n)).

Em relação à complexidade de espaço do MergeSort, utiliza-se um espaço adicional na hora de mergear dois subvetores. Esse custo é linear e depende do tamanho do arranjo, podendo ser representado por O(n). As chamadas recursivas têm uma profundidade de O(n log(n)) , mas cada chamada recursiva utiliza apenas um espaço constante adicional (para manter o estado da pilha de chamadas).Portanto, a complexidade de espaço total é dominada pelo espaço necessário para os arrays temporários usados durante a fusão, resultando em uma complexidade de espaço de O(n).

QuickSort: O QuickSort foi desenvolvido na sua forma recursiva escolhendo o elemento central do vetor para tentar evitar o pior caso. Utilizou-se a função de partição que percorre todos elementos do vetor até dividir ele em dois, e a função QuickSort que coordena a condição de parada por meio dos índices da esquerda e direita do vetor.

Para analisar a complexidade de tempo do algoritmo analisaremos o pior e melhor caso. O pior caso acontece quando o pivô é o maior ou menor elemento do vetor já que criaremos partições desbalanceadas. Para esse caso podemos dizer que na função de complexidade teremos um custo linear que é o custo da partição somado com o custo para chamar o QuickSort para um vetor com apenas um elemento a menos que seria n - 1. Assim temos a seguinte função de complexidade: T(n) = T(n-1) + n na qual podemos utilizar o método da expansão de termos

T(n) = T(n-1) + n
T(n-1) = T(n-2) + n - 1
T(2) = T(1) + 2
T(1) = T(0) + 1 onde T(n) = (n(n+1))/2 portanto → O(n²)

De maneira similar pode-se analisar o melhor caso do algoritmo que é quando particionamos o vetor exatamente na metade. Nesse caso teremos o custo linear da partição somado ao custo para resolver a ordenação no vetor dividido no meio. Temos a seguinte função de complexidade:T(n) = 2T(n/2) + n que a partir do teorema mestre no qual a=2,b=2 f(n) = n calculamos $n^{log_b a}$ = n podemos concluir que se enquadra no caso 2 e portanto a complexidade é O(n log(n)).
Para análise de complexidade de espaço percebe-se que no pior caso onde a partição acontece sempre gerando um vetor com tamanho n - 1, podemos ter um chamada recursiva que pode ir até

n resultando em uma complexidade O(n). No melhor caso o vetor será dividido pela metade fazendo que a ideia de divisão e conquista seja eficiente tendo log n repartições e portanto complexidade O(log(n)).

ShellSort: O Shell Sort é uma extensão do Insertion Sort, introduzindo uma lógica de divisão do vetor principal em subvetores menores, que são ordenados primeiro, antes de ordenar o vetor completo. Esse método reduz a quantidade de deslocamentos necessários, melhorando a eficiência do algoritmo para vetores maiores e parcialmente ordenados. A análise de complexidade depende da escolha dos intervalos que irão determinar quantos subvetores iremos ordenar, para o experimento em questão escolheu-se a sequência $n/2^i$

Para análise percebe-se que o vetor principal é dividido na mesma quantidade em relação ao número do intervalo escolhido na sequência. No pior caso temos uma complexidade temporal próxima de O(n²) já que a redução dos gaps não é tão eficiente em reduzir o número de trocas necessárias para ordenar o vetor, sendo assim, os subvetores não são suficientemente pequenos para garantir uma ordenação eficiente, e o algoritmo pode se comportar de forma semelhante ao Insertion Sort em termos de quantidade de deslocamentos e comparações. Para implementação em questão no loop externo teríamos um número de iterações aproximadamente log(n), no loop intermediário teríamos n-h onde h seria o gap e no loop mais interno teríamos n/h

No melhor caso, quando o vetor está ordenado ou quase ordenado, as operações de inserção requerem poucas trocas. Com gaps maiores no início, os deslocamentos necessários diminuem resultando em um comportamento mais eficiente do que o Insertion Sort puro, reduzindo o tempo de execução para O(n log n).

Em relação à complexidade de espaço, o ShellSort é um algoritmo de ordenação in-place, o que significa que ele não requer espaço adicional significativo além do necessário para armazenar o array original. Portanto, a complexidade de espaço é O(1).

CoutingSort: Esse algoritmo consiste em criar um vetor de contadores para contar quantos elementos temos de cada valor. Após isso percorremos esse vetor para montar o vetor final já ordenado. Em relação a complexidade de tempo temos um custo linear O(n) para percorrer o vetor que recebemos para ordenar e realizar a contagem. Além dele, temos um custo O(max) onde max é o maior valor no vetor que estamos ordenando para percorrer o vetor de contagem. Assim, temos o tempo total com O(n + max). Em relação ao espaço precisamos de uma memória extra para o vetor de contagem que depende do valor max e portanto será O(max).

BucketSort: A ideia principal é dividir o vetor em baldes onde em cada um deles já vamos inserindo os elementos já na ordem correta, nesse caso usou-se o insertionSort para fazer essa ordenação dos baldes. Para análise de complexidade de tempo podemos dividir em dois casos. No melhor caso, assumimos que os elementos do array estão distribuídos uniformemente entre os baldes, resultando em uma distribuição aproximadamente uniforme de elementos em cada balde. Nesse cenário, a criação e distribuição dos elementos nos baldes tem complexidade O(n), onde n é o número de elementos no array. A ordenação de cada balde usando Insertion Sort, dado que cada balde contém aproximadamente n/k elementos (onde k é o número de baldes), é O(n^2/k^2) por balde. Como temos k baldes o total da ordenação fica O(n^2/k).

Para o pior caso, no qual temos elementos que não estão distribuídos uniformemente nos baldes teremos um ou mais baldes tendo mais elementos do que outros, tornando o bucket sort menos eficiente. A complexidade temporal aumenta ainda mais se os elementos no array estiverem presentes na ordem inversa. Se o algoritmo Insertion Sort for usado, a complexidade temporal pior caso pode chegar a O(n^2).

A complexidade espacial para o bucket sort é O(n+k), onde n = número de elementos no array, e k = número de baldes formados. O espaço ocupado por cada balde é O(k), e dentro de cada balde, temos n elementos espalhados. Portanto, a complexidade espacial se torna O(n+k).

RadixSort: Nesse algoritmo fazemos a ordenação comparando os valores do bits, utilizando-se de uma ideia similar do particionamento de vetor do QuickSort para ordenar os valores. Considerando um vetor de n elementos e com k bits em cada chave podemos dizer que a complexidade de tempo é O(nk) pois fazemos k passagens pelo vetor de n elementos para

comparação dos bits e particionamento do vetor. Para análise da complexidade de espaço, o radixSort com as chamadas recursivas terá um custo linear somado ao valor de k dos bits que irá determinar o custo dessa pilha de recursão, portanto temos a complexidade O(n+k).

4. Estratégias de robustez

No contexto do desenvolvimento do programa para testar os algoritmos de ordenação, diversas escolhas foram feitas para tornar o teste mais simples, eficiente e seguro, evitando possíveis erros. Dentre elas, destaca-se a prevenção de acesso indevido nas posições dos vetores ao utilizar funções como findMax e partition que possuem validações para garantir que a manipulação do vetor ocorra dentro dos limites corretos, prevenindo acessos fora do intervalo válido.

Outra estratégia importante é a validação na alocação e liberação de memória. Após utilizar a memória dinamicamente alocada, o programa sempre usa a função free para liberá-la, evitando vazamentos de memória. No main, validações são feitas nos parâmetros de entrada passados para a execução do programa. Isso permite ao usuário escolher o tamanho do vetor, a unicidade das chaves, a ordenação e o algoritmo selecionado, assegurando que todos os argumentos fornecidos sejam válidos antes de continuar.

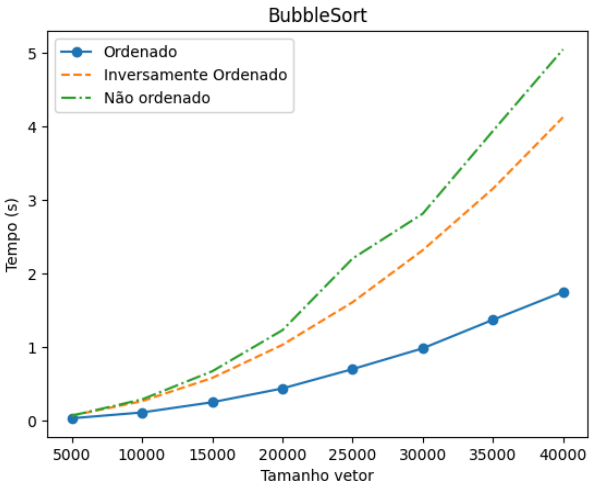
Além disso, mensagens de log foram incorporadas por meio de printf no terminal para facilitar a identificação de problemas durante a ordenação e garantir que os algoritmos estejam funcionando corretamente. Esses logs ajudam na depuração, permitindo verificar o progresso e o funcionamento dos algoritmos. Essas medidas contribuem significativamente para a robustez do programa, tornando-o mais confiável e menos suscetível a erros.

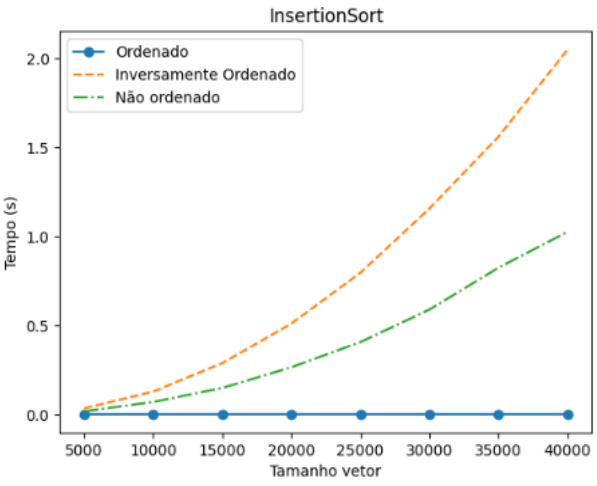
5. Análise Experimental

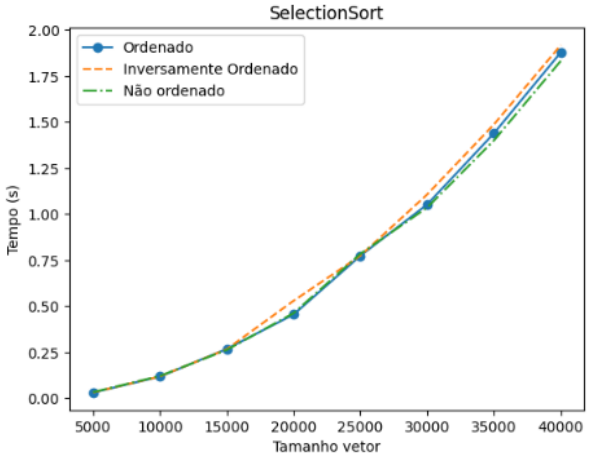
Este tópico será dividido em duas partes. Na primeira, analisaremos o custo de execução de cada algoritmo individualmente, variando o tamanho do vetor e a maneira como o vetor está ordenado previamente, permitindo uma compreensão mais detalhada das particularidades de cada algoritmo. Na segunda parte analisaremos um caso de comparação direta entre todos os algoritmos implementados para um vetor que apresenta uma estrutura que requer um espaço maior de alocação de memória.

5.1 Análises individuais

5.1.1 Resultados BubbleSort, InsertionSort, SelectionSort





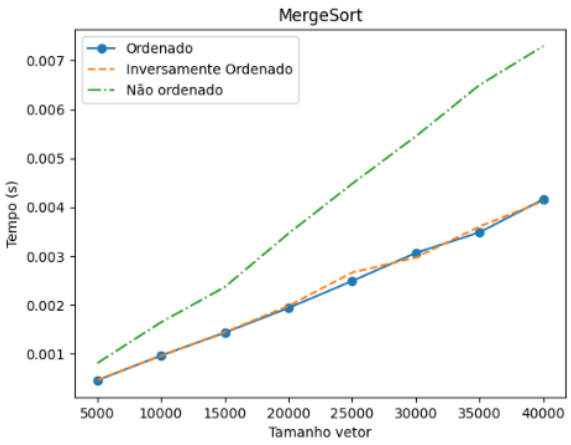


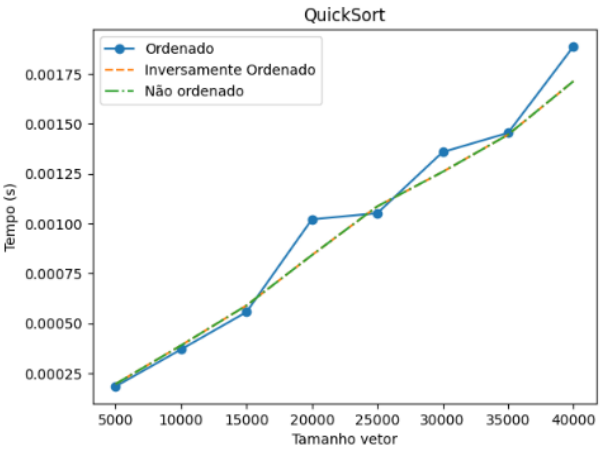
O bubbleSort apresenta o melhor tempo de execução para o vetor ordenado já que não é necessário nenhuma troca. O pior tempo é para o vetor não ordenado que requer um grande número de comparações e trocas sua função é do tipo $O(n^2)$. Para o vetor inversamente ordenado também temos uma curva com característica $O(n^2)$ já que realizamos o maior número de trocas possível, pelo fato do algoritmo implementado no experimento não apresentar nenhuma técnica de otimização conclui-se que a distribuição aleatória dos dados exigiu um custo maior do que o vetor inversamente ordenado.

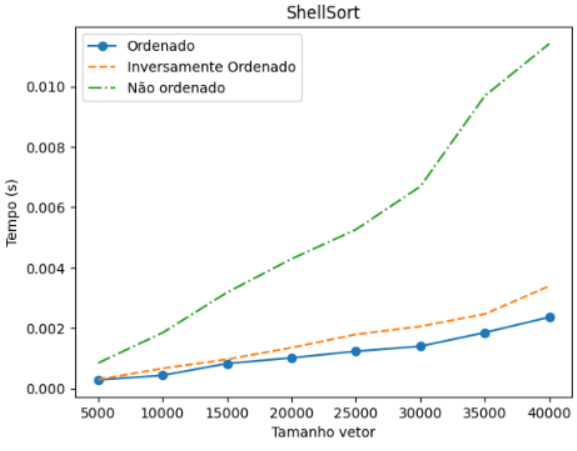
No insertionSort comprova-se a ótima eficiência teórica para vetores ordenados, no qual ele não realiza nenhuma troca, apenas comparações. Para o vetor inversamente ordenado apresenta-se o pior caso $O(n^2)$ no qual são realizadas muitas trocas e comparações. O vetor não ordenado é um pouco melhor mas também cresce de forma exponencial pois necessita de muitas inserções.

O gráfico do selectionSort comprova sua análise teórica na qual o tipo de ordenação do vetor não influencia no custo do algoritmo, sempre realizamos um número padrão de comparações. Sua vantagem está no número de trocas que é $O(n)$, portanto menor em relação aos anteriores.

5.1.2 Resultados MergeSort, QuickSort, ShellSort





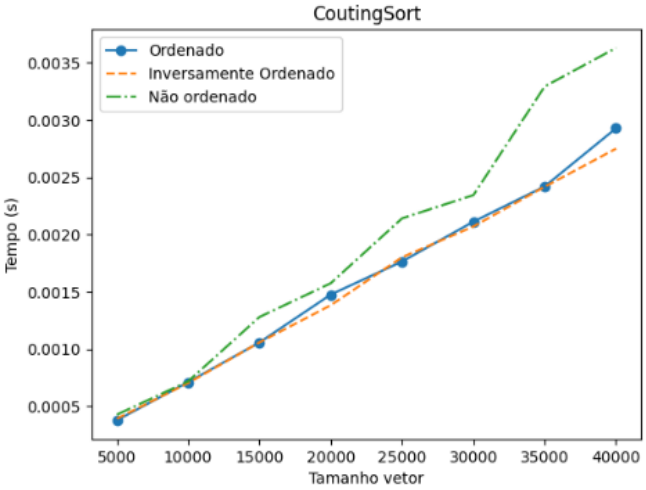


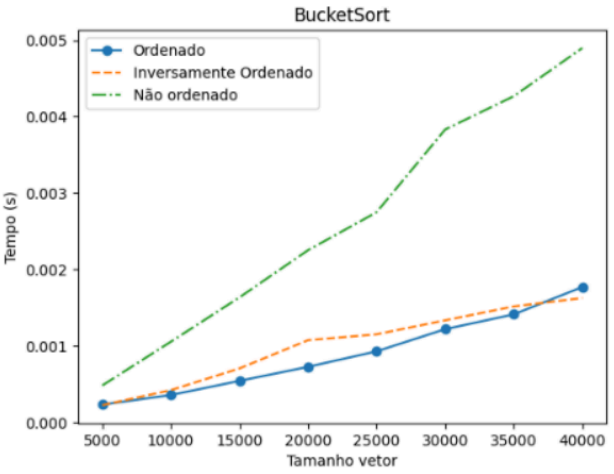
Observa-se o comportamento das curvas do mergeSort aproximando-se de uma função O(n logn). A pequena diferença no tempo de execução do MergeSort para elementos ordenados em ordem ascendente versus descendente pode ser atribuída à natureza do algoritmo, que realiza um número constante de operações de escrita.O vetor não ordenado leva mais tempo para ser processado pelo MergeSort devido à maior complexidade na previsão de ramificações e ao número maior de comparações necessárias.

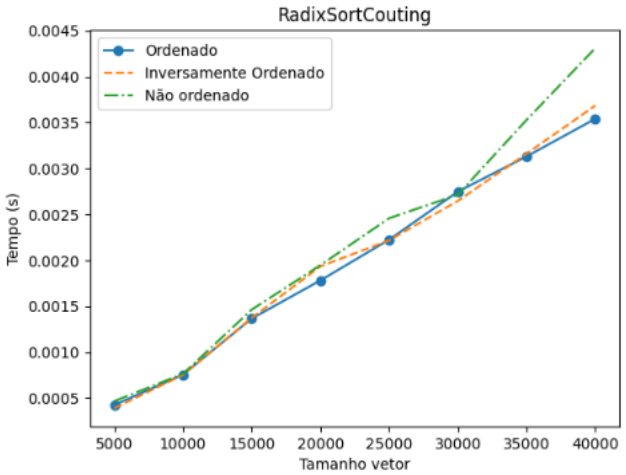
As curvas do quickSort também apresentam comportamento O(n logn), o fato de não ter uma diferença significativa entre os diferentes tipos de ordenação permite concluir que a escolha do pivô como elemento central evitou o pior caso nos testes.

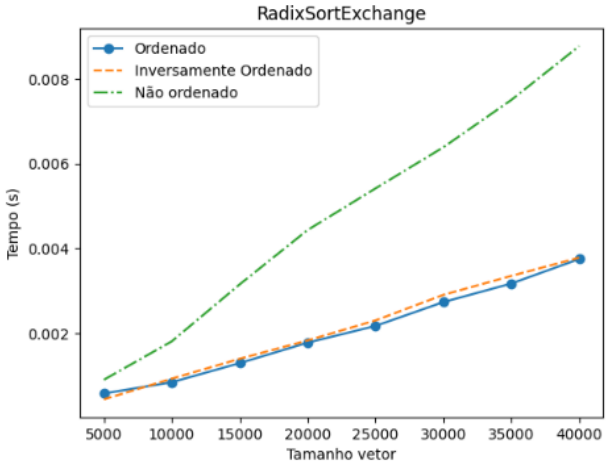
Para a implementação do shellSort com a sequência n/2^i percebe-se que para o vetor ordenado o número de comparações e troca é mínimo portanto representa o menor custo, com o vetor inversamente ordenado o número de trocas cresce porém continuamos com uma característica mais linear da curva O(n log n). O vetor não ordenado é o que requer maior custo para ordenação dentro todos apresentando comportamento com uma tendência similar ao pior caso O(n²).

5.1.2 Resultados CoutingSort, BucketSort, RadixSort (couting e exchange)









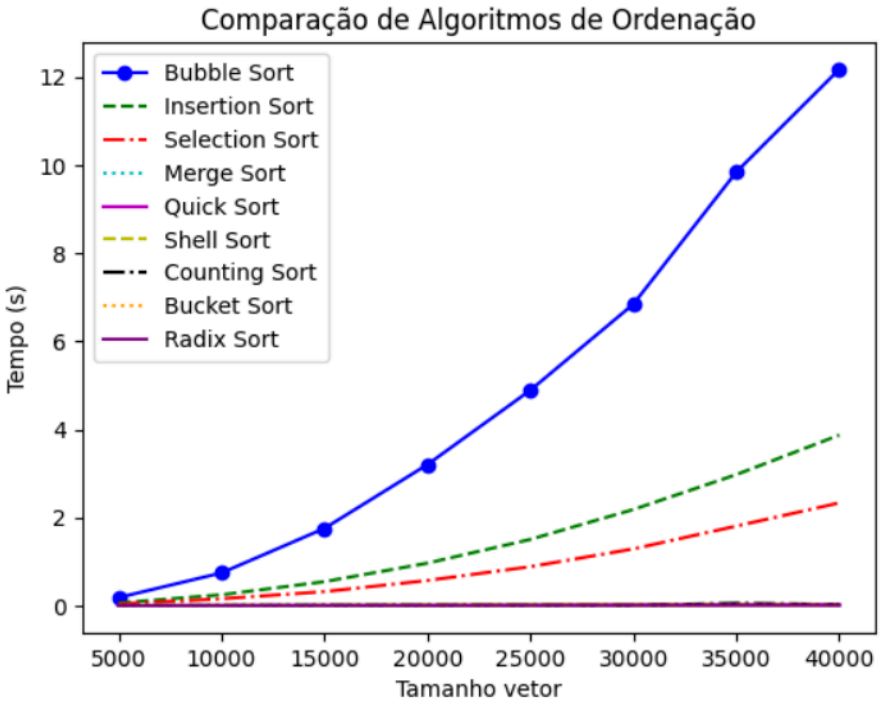
O resultado do coutingSort está de acordo com sua análise teórica de complexidade com um curva mais linear O(n+k) onde k é o valor máximo no vetor, percebe-se que seu custo não é sofre grande influência de acordo com o tipo de ordenação do vetor, nesse caso o que mais influencia é o valor máximo.

A complexidade do bucketSort é influenciada pelo número de buckets que estamos usando e pela maneira que o vetor é distribuído nesses buckets sua complexidade média é O(n^2/k) podendo ter o pior caso O(n²). Para o vetor não ordenado o tempo de execução maior pode ser explicado pela distribuição menos uniforme dos elementos nos buckets.

O RadixSortCounting realiza diversas iterações ordenando os elementos com base em um único dígito. Ele utiliza uma lógica similar ao CountingSort para garantir a ordenação estável em cada passagem. Percebe-se uma complexidade de custo linear o que está de acordo com a teoria. Para essa implementação em específico o custo não depende da ordem inicial dos elementos, portanto o tempo de execução para vetores ordenados e inversamente ordenados é semelhante ao de vetores não ordenados.

O RadixSortExchange é uma variante do RadixSort que utiliza a troca de elementos, similar ao Quicksort, mas baseando-se nos bits dos números em vez de seus dígitos. No caso dos vetores ordenados a recursão e troca são mais eficientes porque muitos elementos já estão aproximadamente na ordem correta. Cada bit de significância leva a menos trocas. Para o caso do vetor não ordenado o comportamento é mais imprevisível porque os bits são distribuídos aleatoriamente. Isso pode levar a mais trocas e chamadas recursivas, especialmente nos níveis iniciais onde os sub-vetores são maiores.

5.2 Análise Conjunta



O gráfico apresentado compara o tempo de execução de diferentes algoritmos de ordenação para ordenar uma estrutura de chave e valor, onde o valor é um caractere de 99 bytes. Este cenário é particularmente relevante porque algoritmos que realizam um grande número de trocas de elementos tendem a ser mais demorados devido ao maior tempo de manipulação e movimento de dados de maior tamanho.

Os algoritmos de ordenação que realizam um maior número de trocas, como Bubble Sort, InsertionSort, apresentam um desempenho significativamente pior, especialmente à medida que o tamanho do vetor aumenta. O selectionSort apesar de ser o algoritmo que realiza um menor número de trocas com ordem linear de n, apresentou um comportamento de acordo com a sua complexidade de tempo teórica O(n^2) para vetores grandes e por isso teve resultado pior que alguns outros métodos. Por outro lado, algoritmos como Merge Sort, Quick Sort, Counting Sort,

Bucket Sort e Radix Sort se mostraram muito eficientes, especialmente para grandes conjuntos de dados, devido ao menor número de trocas e operações de manipulação de dados.

Observando o formato das curvas dos três piores algoritmos de ordenação – Bubble Sort, Insertion Sort e Selection Sort – podemos notar que essas curvas apresentam um crescimento exponencial à medida que o tamanho do vetor aumenta. Isso se deve à complexidade temporal desses algoritmos, que é O(n^2), resultando em um aumento drástico no tempo de execução para vetores maiores. Em contraste, as curvas dos outros algoritmos de ordenação, como Merge Sort, Quick Sort, Counting Sort, Bucket Sort e Radix Sort, permanecem próximas umas das outras e quase constantes, demonstrando uma eficiência significativamente maior. Essas curvas mais próximas e de crescimento mais lento refletem as complexidades temporais mais eficientes, como O(n log n) ou até O(n), que permitem que esses algoritmos lidem com grandes conjuntos de dados de maneira mais eficaz e consistente.

Essa análise reflete a importância de escolher o algoritmo de ordenação apropriado, especialmente em contextos onde a eficiência e o tempo de execução são críticos, como em grandes estruturas de dados com diversos elementos.

6. Conclusões

Após a análise e comparação final dos experimentos com a teoria de complexidade, é possível perceber a importância da análise de complexidade de um método desenvolvido. No trabalho em questão, o objetivo final de evidenciar os melhores e piores casos de uso para os algoritmos de ordenação foi alcançado através do contraste entre a análise teórica e experimental.

Durante a realização dos testes, ficou evidente que cada algoritmo de ordenação possui características específicas que o tornam mais ou menos eficiente dependendo do contexto em que são aplicados. Por exemplo, algoritmos como QuickSort e MergeSort mostraram-se eficientes para grandes volumes de dados, devido às suas complexidades temporais médias de O(n log n). Em contrapartida, algoritmos mais simples como BubbleSort e SelectionSort, embora fáceis de implementar e entender, demonstraram ser ineficientes para conjuntos de dados maiores, devido às suas complexidades temporais de O(n²) no pior caso.

Além disso, a implementação prática dos algoritmos revelou a importância da programação defensiva e da gestão adequada de recursos, como memória. A inclusão de validações de entrada, a correta alocação e liberação de memória e o uso de logs para depuração foram fundamentais para garantir a robustez e a eficiência do programa. Essas práticas são essenciais não apenas para evitar erros e vazamentos de memória, mas também para assegurar que o programa funcione conforme esperado em diferentes cenários.

Sendo assim, o trabalho proporcionou um entendimento aprofundado dos algoritmos de ordenação, tanto do ponto de vista teórico quanto prático. A análise comparativa evidenciou que a escolha do algoritmo pode impactar significativamente o desempenho das aplicações. As lições aprendidas destacam a importância de selecionar o melhor algoritmo para determinado contexto específico, exigindo do desenvolvedor um bom entendimento dos requisitos do seu problema

7. Bibliografia

Algorithms in C robert sedgewick 1996 Chapters on Sorting Algorithms