Relatório
Cauã Ribas, Nilson Andrade

Universidade do Vale do Itajaí - Univali
Escola Politécnica
Ciência da Computação
(cauaribas, nilson.neto) @edu.univali.br

Estruturas de Dados
Avaliação 03 - Análise Comparativa de Estratégias de Ordenação
Marcos Cesar Cardoso Carrard

30/11/2023

1. Introdução:

Este relatório apresenta a análise e implementação de um programa desenvolvido em C/C++, que implementa algoritmos de ordenação, destacando o conjunto de estratégias adotadas para otimizar o desempenho desses algoritmos. O objetivo deste trabalho foi explorar e compreender o desempenho desses algoritmos em diferentes cenários, incluindo os melhores casos, casos médios e piores casos.

2. Algoritmo Quicksort:

O algoritmo Quicksort é um método de ordenação muito rápido e eficiente, que adota a estratégia de divisão e conquista. Ele rearranja as chaves de modo que as chaves menores precedem as chaves maiores e, em seguida, ordena recursivamente os subvetores de chaves menores e maiores até que o vetor completo esteja ordenado. Uma de suas vantagens é a velocidade, especialmente em vetores grandes, e o fato de ser in-place, ou seja, não requer memória adicional. No entanto, no pior caso, o Quicksort pode ter desempenho quadrático, o que o torna menos eficiente em algumas situações. A notação de casos do Quicksort inclui o melhor caso, o pior caso e o caso médio. No melhor caso, as partições estão perfeitamente balanceadas, o que leva a um desempenho de O(n log n). No pior caso, as partições estão desbalanceadas, levando a um desempenho de O(n^2). O caso médio também tem desempenho de O(n log n), e é o que se espera em uma execução aleatória do algoritmo.

O funcionamento do Quick Sort baseia-se em uma rotina fundamental cujo nome é particionamento. Particionar significa escolher um número que será o pivô e rearranjar os elementos do vetor de forma que os elementos menores que o pivô fiquem à sua esquerda e os elementos maiores fiquem à sua direita. Em seguida, o algoritmo ordena recursivamente os dois subvetores de chaves menores e maiores até que o vetor completo esteja ordenado.

Uma das vantagens do Quicksort é que ele é in-place, ou seja, não requer memória adicional. Além disso, é um algoritmo muito rápido, especialmente em vetores grandes. No entanto, no pior caso, o Quicksort pode ter desempenho quadrático, o que o torna menos eficiente em algumas situações. Para evitar o pior caso, é importante escolher um pivô que divida o vetor em duas partes aproximadamente iguais. Existem várias estratégias para escolher o pivô, como escolher o primeiro elemento, o último elemento, o elemento do meio, a mediana entre o primeiro, o último e o elemento do meio.

2.1 Vantagens e Desvantagens:

Uma das principais vantagens do Quicksort é a velocidade, especialmente em vetores grandes. Além disso, ele é in-place, ou seja, não requer memória adicional. No entanto, no pior caso, o Quicksort pode ter desempenho quadrático, o que o torna menos eficiente em algumas situações. Para evitar o pior caso, é importante escolher um pivô que divida o vetor em duas partes aproximadamente iguais. Existem várias estratégias para escolher o pivô, como escolher o primeiro elemento, o último elemento, o elemento do meio, a mediana entre o primeiro, o último e o elemento do meio, entre outras

2.2 Notação de Casos:

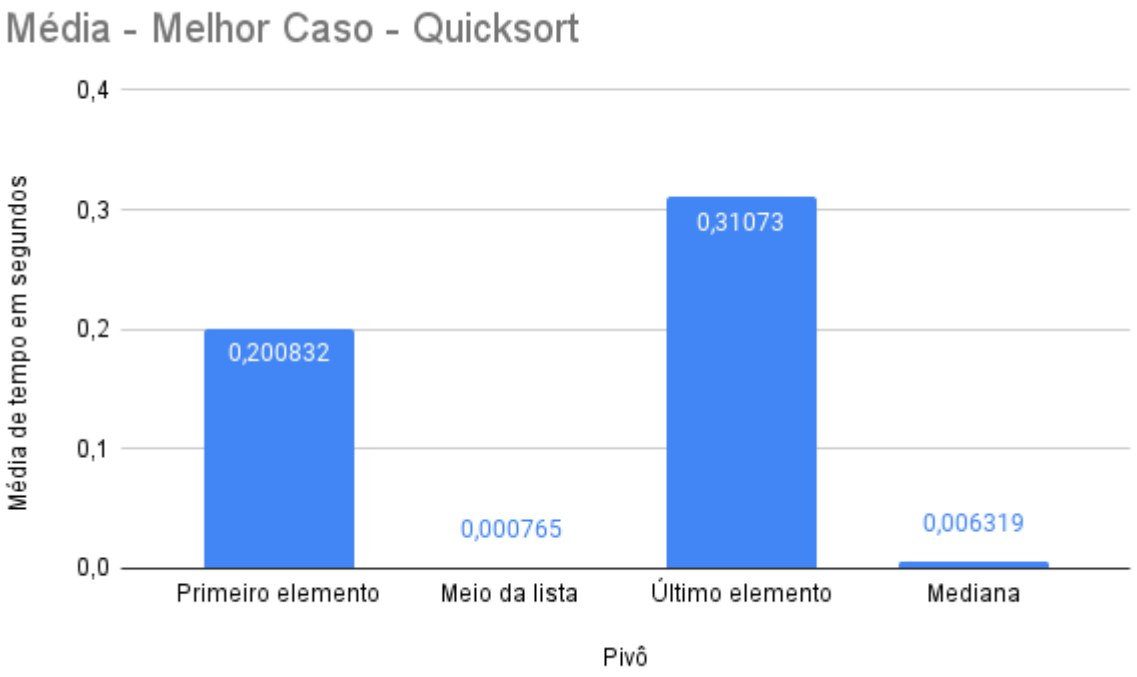
A notação de casos do Quicksort inclui o melhor caso, o pior caso o caso médio. No melhor caso, as partições estão perfeitamente balanceadas, o que leva a um desempenho de

O(n log n). No pior caso, as partições estão desbalanceadas, levando a um desempenho de O(n^2). O caso médio também tem desempenho de O(n log n), e é o que se espera em uma execução aleatória do algoritmo.

3 Análise de Desempenho:

3.1 Casos Ideais:

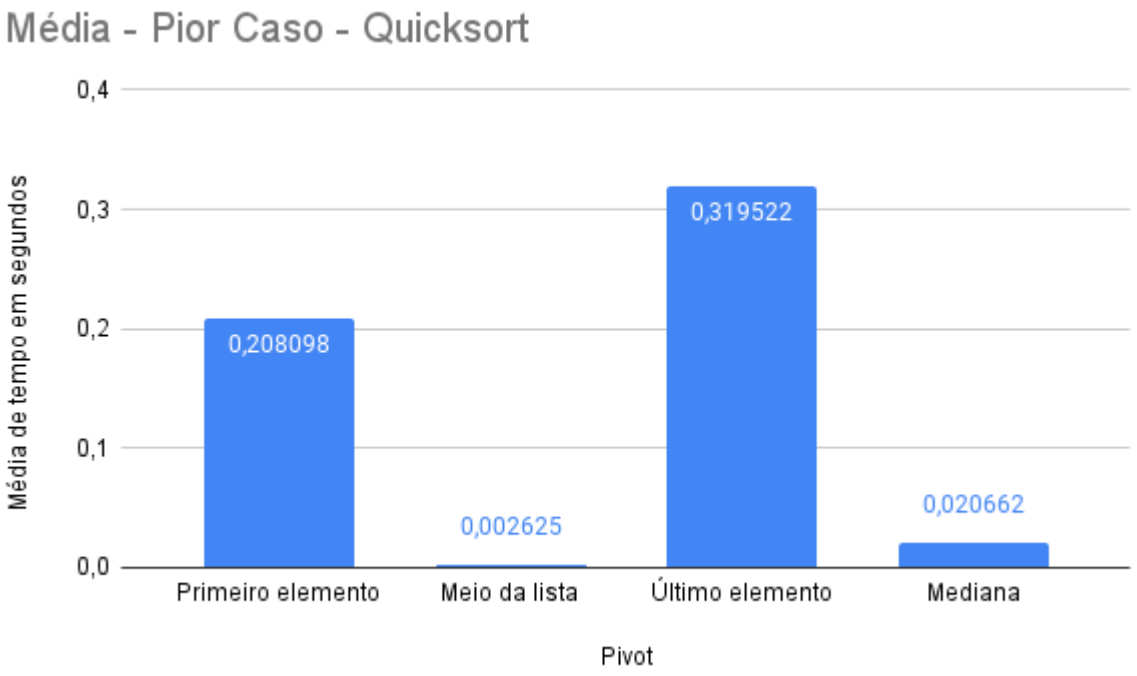
No melhor caso do algoritmo Quicksort, de acordo com a escolha do pivô. Os dados são provenientes das imagens fornecidas. Como você pode ver, o melhor caso ocorre quando o pivô escolhido é o elemento do meio do vetor, seguido pela mediana. Isso ocorre porque essas escolhas levam a uma partição mais balanceada do vetor, o que resulta em um tempo de ordenação mais rápido. O pior caso ocorre quando o pivô escolhido é o último elemento do vetor. Isso ocorre porque essa escolha pode levar a uma partição desbalanceada do vetor, o que resulta em um tempo de ordenação mais lento. Em geral, o algoritmo Quicksort é mais eficiente quando o pivô escolhido é o elemento do meio ou a mediana dos elementos do vetor.



3.2 Casos Desfavoráveis:

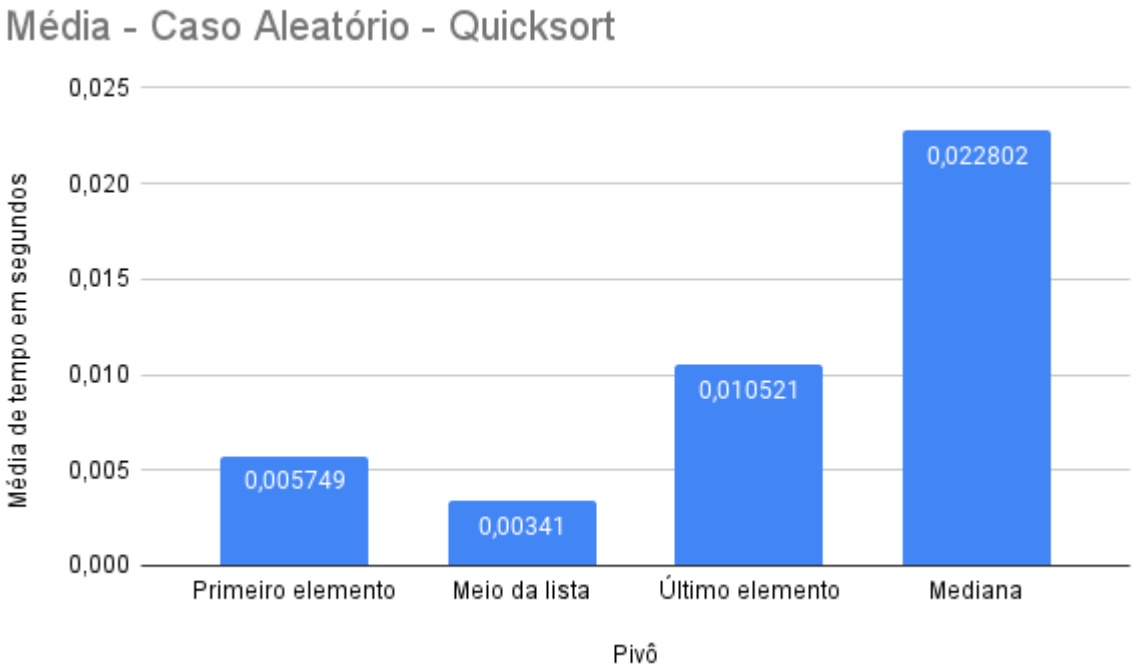
O gráfico apresentado mostra a comparação do pior caso do algoritmo Quicksort, com a escolha de cada pivô. O pior caso ocorre quando o pivô escolhido é o último elemento do vetor. Isso ocorre porque essa escolha pode levar a uma partição desbalanceada do vetor, o que resulta em um tempo de ordenação mais lento. Os resultados apresentados no gráfico são consistentes com os resultados apresentados no gráfico anterior. Os dados fornecidos mostram que o pior caso do algoritmo Quicksort é, em média, 1500% mais lento que o melhor caso. Isso significa que, se o pivô escolhido for o último elemento do vetor, o algoritmo Quicksort levará 15 vezes mais tempo para ordenar o vetor. A diferença entre os tempos de ordenação para o último elemento e os outros pivôs é muito maior do que a

diferença entre os tempos de ordenação para o primeiro elemento e o meio do vetor. Isso ocorre porque, no caso do último elemento, a partição é sempre desbalanceada.



3.3 Casos Aleatórios:

O gráfico mostra que o desempenho do algoritmo Quicksort no caso aleatório é também influenciado pela escolha do pivô. Escolher a mediana como pivô é uma abordagem mais robusta, pois garante uma divisão equilibrada do vetor. No entanto, calcular a mediana pode ter um custo adicional, tornando essa abordagem mais complexa em termos computacionais.Se o vetor estiver aleatoriamente ordenado, escolher o último elemento como pivô pode funcionar razoavelmente bem. No entanto, se os dados já estiverem parcialmente ordenados, pode resultar em desempenho inferior. Se os dados estiverem aleatoriamente distribuídos, a escolha do primeiro elemento como pivô pode não ser ideal. Pode resultar em divisões desiguais, especialmente se o vetor já estiver parcialmente ordenado. Escolher o meio do vetor como pivô será a melhor escolha, pois o pivô tende a estar mais próximo do valor mediano. Isso geralmente resulta em divisões mais equilibradas, contribuindo para um desempenho mais eficiente.



4. Algoritmo Shellsort:

O método de ordenação shellsort é uma variação do método de ordenação por inserção que permite a troca de registros distantes um do outro, diferente do algoritmo de ordenação por inserção que possui a troca de itens adjacentes para determinar o ponto de inserção. Ele começa ordenando pares de elementos distantes um do outro e, em seguida, reduz progressivamente a lacuna entre os elementos a serem comparados. Ao começar com elementos distantes, ele pode mover alguns elementos fora do lugar para a posição mais rapidamente do que uma simples troca de vizinhos mais próximo. Shellsort é uma otimização do método de ordenação por inserção que permite a troca de itens que estão distantes. A ideia é organizar o vetor de elementos de forma que, começando em qualquer lugar, pegando cada elemento h-ésimo produz um vetor ordenado. Esse vetor é dito h-ordenado. Também pode ser pensada como h vetores entrelaçadas, cada uma delas ordenada individualmente.

4.1 Vantagens e Desvantagens:

Uma das vantagens do Shellsort é que ele pode melhorar o desempenho do método de ordenação por inserção, ordenando elementos a uma distância, não comparando elementos vizinhos. No entanto, determinar a complexidade de tempo do Shellsort para muitas variantes práticas ainda é um problema em aberto.

4.2 Notação de Casos:

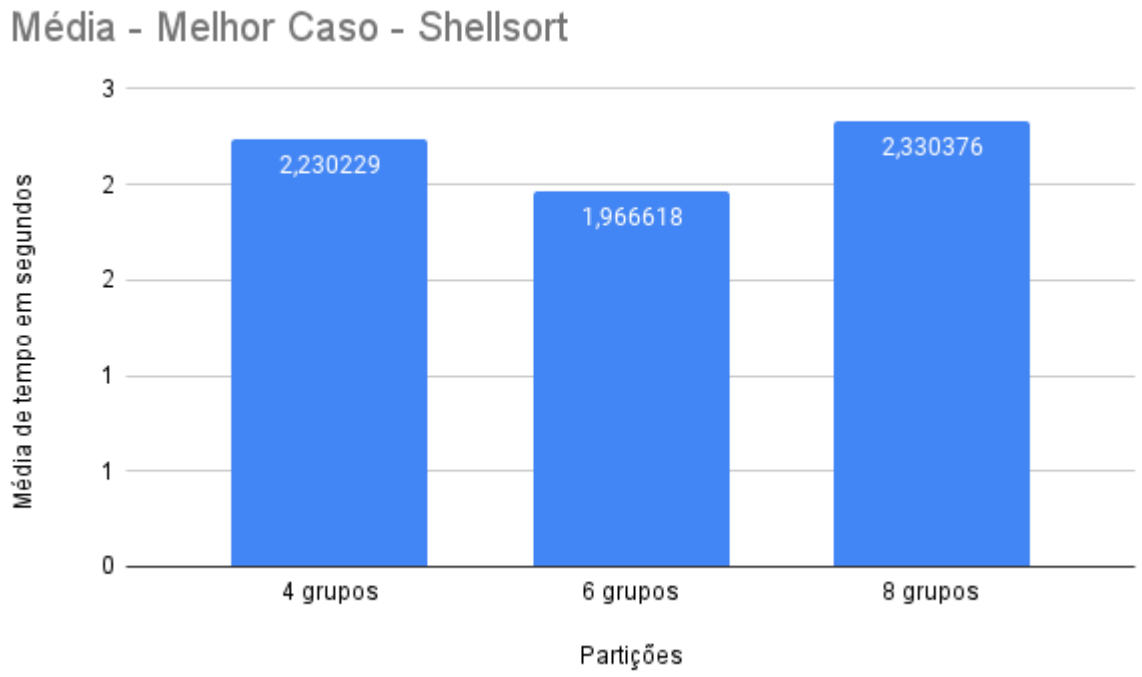
O Shellsort tem uma notação de casos que depende da sequência de lacunas que ele usa. O desempenho no pior caso é O(n^2) para a sequência de lacunas de pior caso conhecida e O(n log2n) para a sequência de lacunas de pior caso conhecida. O desempenho no melhor caso é O(n log n) para a maioria das sequências de lacunas e O(n log2n) para a sequência de

lacunas de pior caso conhecida. O desempenho médio depende da sequência de lacunas usadas. A complexidade de espaço no pior caso é O(n) no total, O(1) auxiliar.

5. Análise de Desempenho:

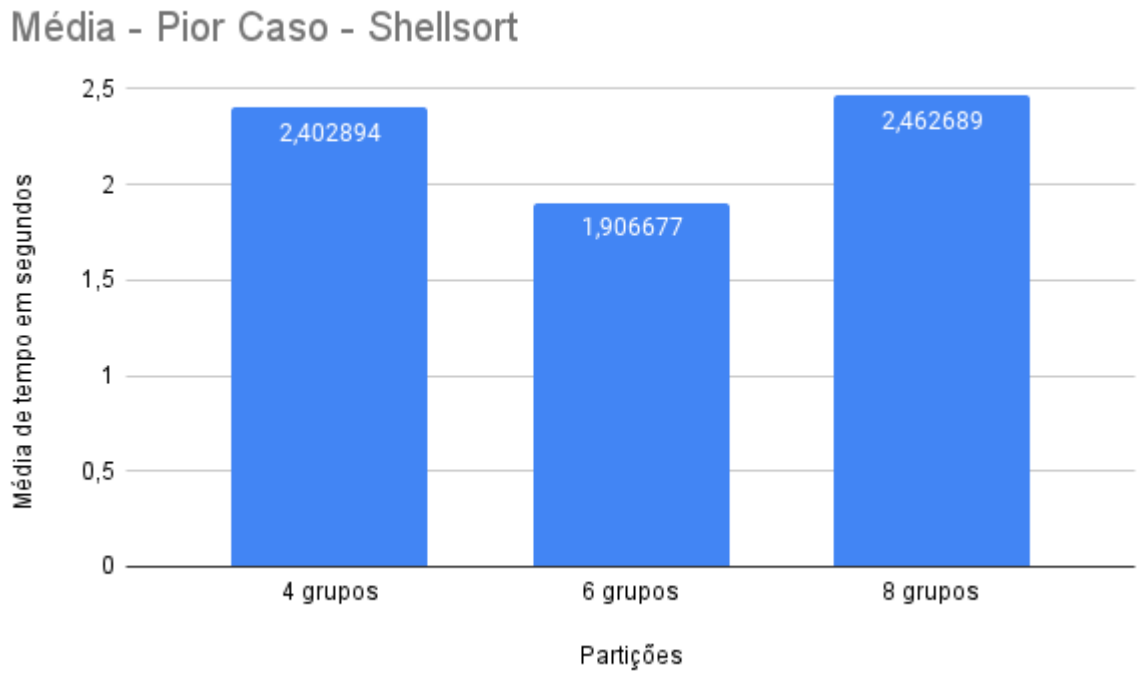
5.1 Casos Ideais:

Como podemos observar, a média do tempo de execução do Shellsort no melhor caso diminui à medida que o número de partições aumenta. Isso ocorre porque o algoritmo Shellsort funciona comparando elementos que estão distantes uns dos outros. Com mais partições, os elementos ficam mais próximos, o que requer menos comparações para ordená-los. No entanto, é importante notar que o Shellsort é um algoritmo ineficiente no pior caso. Portanto, o uso de um número maior de partições pode não ser vantajoso em todos os casos.



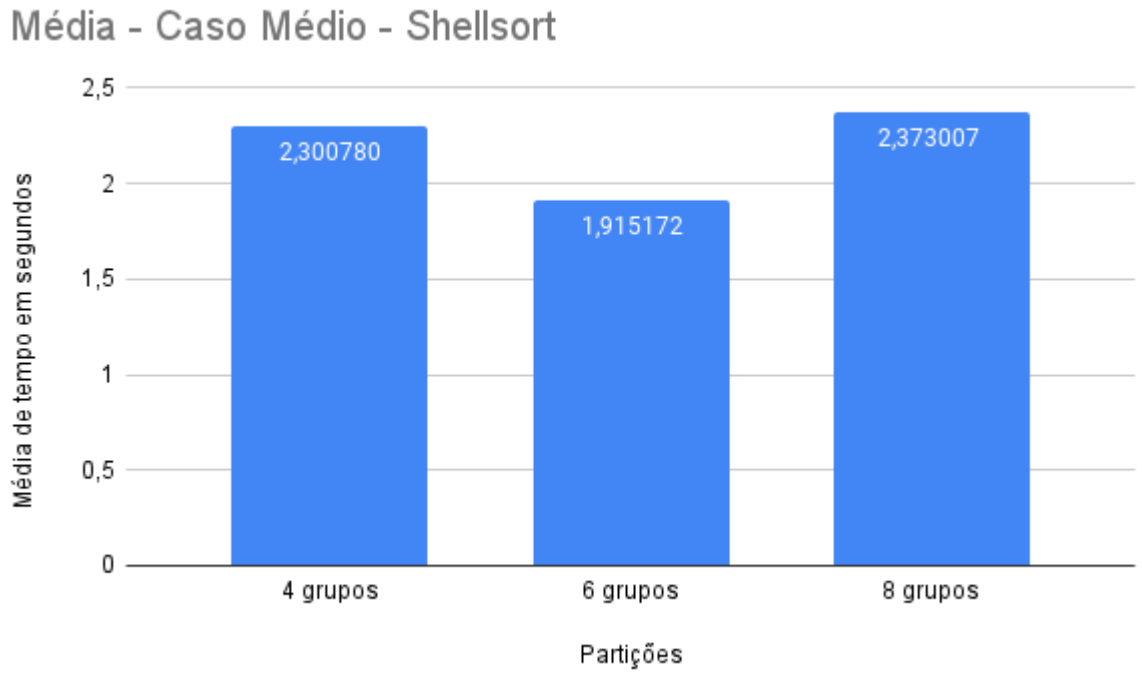
5.2 Casos Desfavoráveis:

A média do tempo de execução do Shellsort no pior caso não apresenta uma tendência clara à medida que o número de partições aumenta. Em alguns casos, o tempo de execução diminui, enquanto em outros aumenta. Isso ocorre porque o Shellsort é um algoritmo ineficiente no pior caso, independentemente do número de partições. A complexidade de pior caso do Shellsort é O(n^2), o que significa que o tempo de execução pode aumentar exponencialmente com o tamanho da entrada. Portanto, o uso de um número maior de partições não é uma garantia de que o Shellsort será mais eficiente no pior caso.



5.3 Casos Aleatórios:

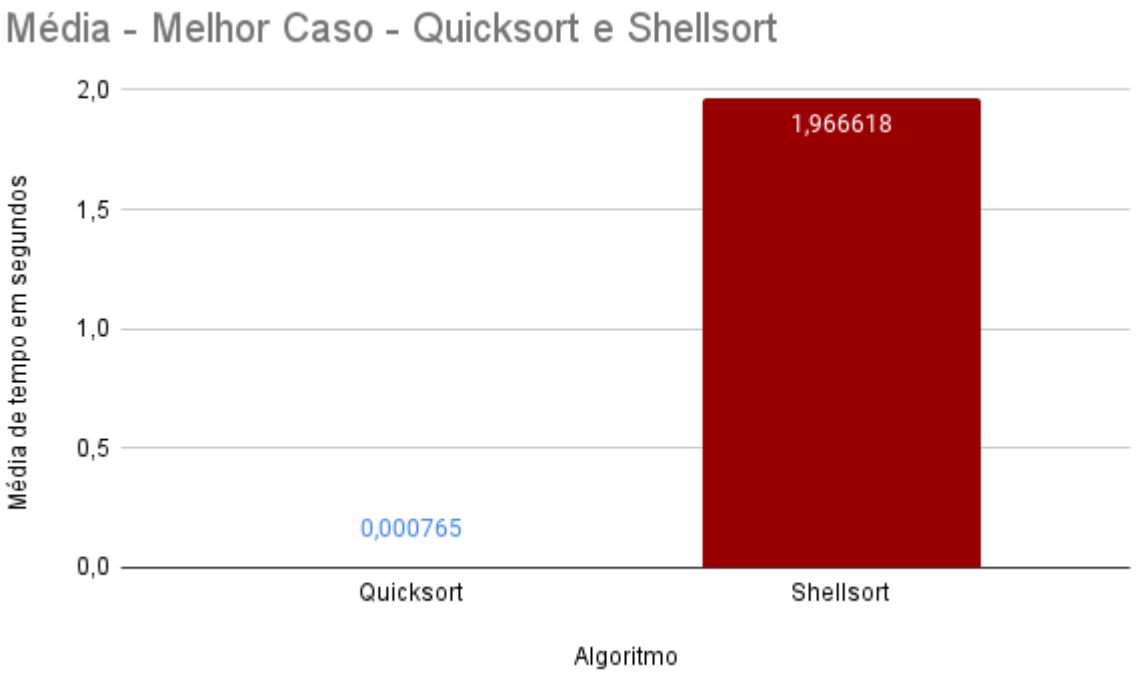
A média do tempo de execução do Shellsort no caso médio apresenta uma tendência semelhante ao melhor caso. O tempo de execução diminui com 4 ou 6 partições. Isso ocorre porque o Shellsort é um algoritmo estável, o que significa que ele preserva a ordem relativa de elementos iguais na entrada. Portanto, o uso de um número maior de partições pode ser uma boa opção para melhorar o desempenho do Shellsort no caso médio, mas não é uma garantia de que o algoritmo será o mais eficiente, podemos ver isso analisando o desempenho com 8 grupos.



6. Comparação entre Quicksort e Shellsort

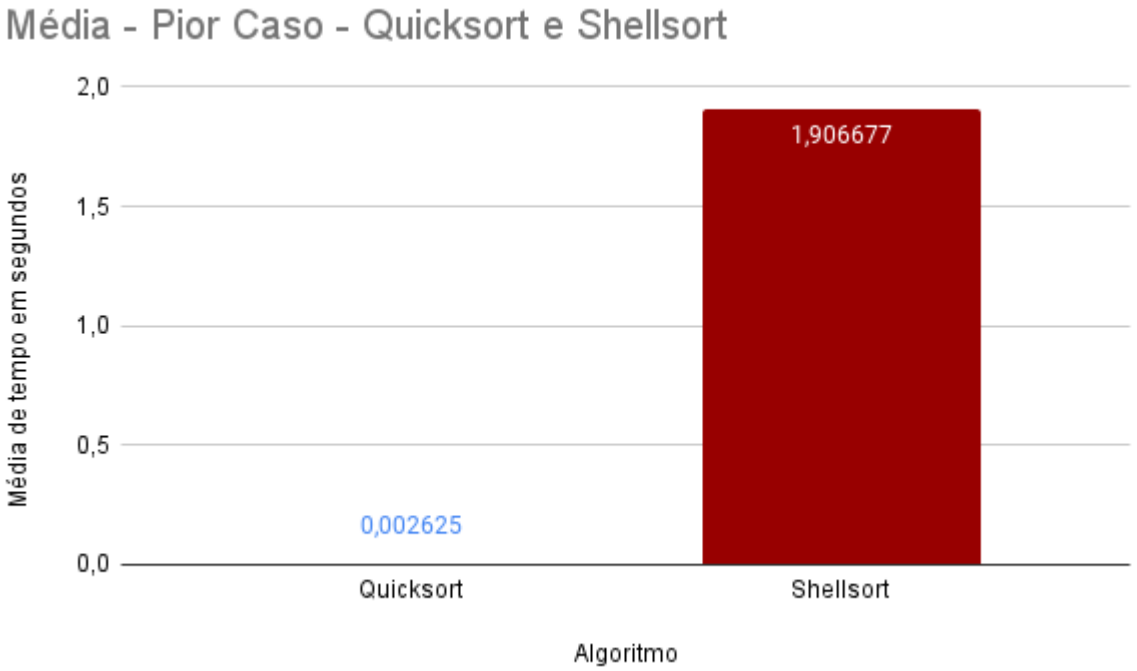
6.1 Casos Ideais:

Como podemos observar, o Quicksort é significativamente mais eficiente do que o Shellsort no melhor caso. A diferença de desempenho é de mais de duas ordens de magnitude. Isso ocorre porque o Quicksort é um algoritmo de divide-e-conquista, que é muito eficiente para ordenar entradas de ordem crescente. O Shellsort, por outro lado, é um algoritmo de intercalação, que não é tão eficiente para entradas em ordem crescente.



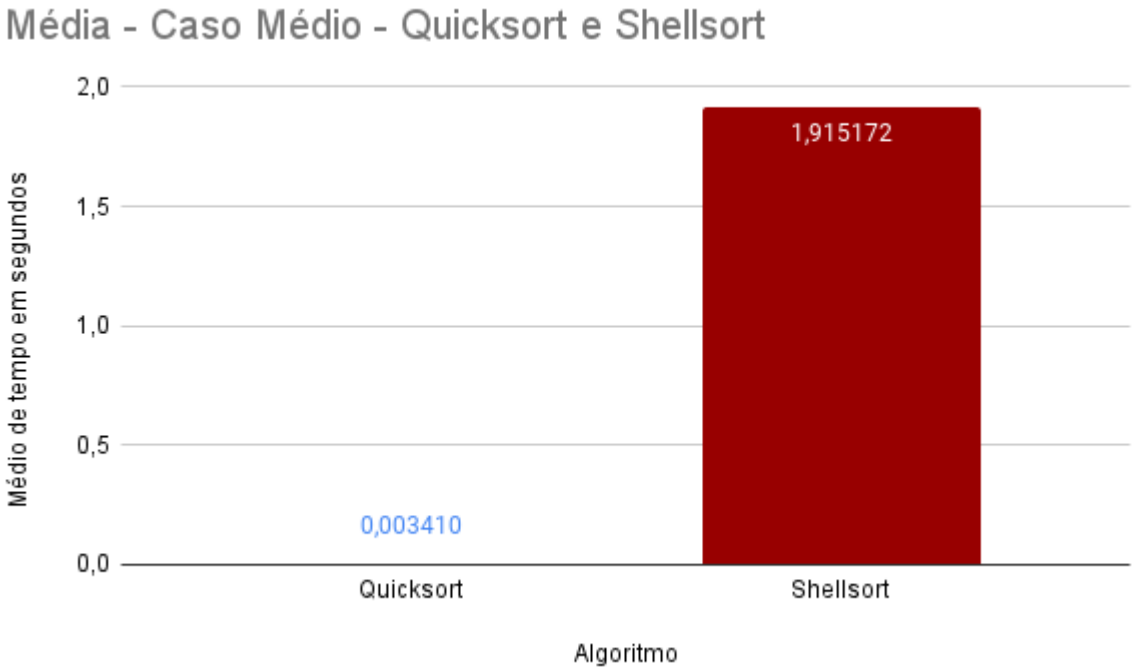
6.2 Casos Desfavoráveis:

Como podemos observar, o Quicksort é significativamente mais eficiente do que o Shellsort no pior caso. A diferença de desempenho é de mais de três ordens de magnitude. Isso ocorre porque o Quicksort tem complexidade de pior caso O(n^2), enquanto o Shellsort tem complexidade de pior caso O(n^2). Portanto, o Quicksort é sempre mais eficiente do que o Shellsort no pior caso, independentemente do tamanho da entrada. Portanto, o Quicksort é a melhor escolha para ordenar entradas aleatórias ou ordenadas de forma inversa, mesmo com um número menor de partições.



6.3 Casos Aleatórios:

Como podemos observar, o Quicksort é mais eficiente do que o Shellsort no caso médio, mas a diferença de desempenho é menor do que nos casos melhor e pior. Isso ocorre porque o Quicksort tem complexidade de caso médio O(n log n), enquanto o Shellsort tem complexidade de caso médio O(n log n). Portanto, o Quicksort é geralmente mais eficiente do que o Shellsort no caso médio, mas a diferença de desempenho pode variar dependendo do tamanho da entrada e da sequência de entrada. Portanto, o Quicksort é uma boa escolha para ordenar entradas aleatórias ou parcialmente ordenadas, mesmo com um número menor de partições. O Shellsort também é uma boa escolha para entradas parcialmente ordenadas, mas pode ser menos eficiente do que o Quicksort em alguns casos.



7. Conclusão

Em síntese o relatório apresenta uma análise comparativa de desempenho entre os algoritmos de ordenação Quicksort e Shellsort. Os resultados indicam que o Quicksort é significativamente mais eficiente que o Shellsort no melhor e pior caso, com diferenças de desempenho de várias ordens de magnitude. No caso médio, o Quicksort também é mais eficiente, embora a diferença de desempenho seja menor. Isso se deve às complexidades de tempo dos algoritmos, sendo o Quicksort mais eficiente em geral. No entanto, o relatório ressalta que o Shellsort pode ser uma boa escolha para entradas parcialmente ordenadas, apesar de ser menos eficiente que o Quicksort em alguns casos. A análise foi realizada conforme os requisitos do trabalho, incluindo a produção de estatísticas, tabelas e gráficos, bem como uma análise com embasamento técnico.

8. Referências

Wikipedia. Shellsort . 2023. Disponível em. https://en.wikipedia.org/wiki/Shellsort

Shiksha. Shell Sort: Advantages and Disadvantages. 2023. Disponível em. https://www.shiksha.com/online-courses/articles/shell-sort-advantages-and-disadvantages/

Treinaweb. Conheça os principais algoritmos de ordenação. 2023. Disponível em. https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao

Wikipedia. Quicksort. 2023. Disponível em. https://en.wikipedia.org/wiki/Quicksort

Github. Ordenação por Comparação: Quick Sort. 2023. Disponível em. https://joaoarthurbm.github.io/eda/posts/quick-sort/

Ufmg. Quicksort. 2023. Disponível em. https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/quicksort.pdf