*Relatório da Atividade : complexidade de algoritmos*

**Francisco Carlos Silva Pimentel**

1Pro-Reitoria de Pesquisa e Pós Graduação (PECS) – Universidade Estadual do Maranhão (UEMA)  
CEP: 65.064-390 – São Luís – MA – Brasil

{[fcalpimentel@hotmail.com](mailto:cacaupimentel@hotmail.com)}

# INTRODUÇÃO

O presente trabalho refere-se ao relatório de implementação de análise de complexidade sobre os algoritmos de ordenação. Para demonstração, um executável foi escrito em linguagem C++, no qual codificou-se 3 (três) algoritmos de ordenação com o objetivo de metrificar e comparar o desempenho e o consumo de memória de cada um deles.

O executável produzido foi denominado de *“sort.exe”* e organizou-se este relatório em cinco capítulos, contando com esta introdução. No segundo abordou-se a escolha dos algoritmos de classificação e seu cálculo de complexidade. No terceiro, contemplou-se os testes de desempenho, o arquivo de saída, os trechos dos códigos e os resultados de tempo e memória. No quarto, foram analisados os resultados dos algoritmos com o comparativo dos gráficos. No quinto, as considerações finais.

# Código fonte e executável

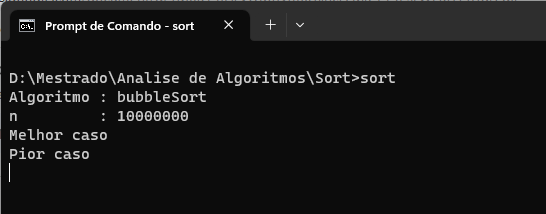
O arquivo executável é produto de um código escrito em linguagem *“C++”* e compilado com o *Cygwin*, cujo código fonte se encontra no arquivo original com a nomeclatura *“****main.cpp”***. Durante o processo de compilação foi atribuído ao executável o nome de *“****sort.exe”***.

Ainda na mesma pasta do executável e do código fonte encontram-se mais alguns arquivos:

* dois arquivos gerenciados pela execução do programa que são: *“****entrada.txt”*** e **“*resultado.csv”***; e
* um arquivo chamado de *“c.bat”* que é responsável pela compilação e execução do arquivo *“****sort.exe”***.

Todos os arquivos (código fonte, executável e demais auxiliares) podem ser obtidos por meio do acesso público ao site do *github*, [https://github.com/calpimentel/AnaliseAlgoritmos-Complexidade](https://github.com/cacaupimentel/AnaliseSorter).

Para disparar a execução do programa basta baixá-lo e digitar na linha de *prompt* a palavra ***sort***,seguido da tecla *Enter*. Não há necessidade de escrever a extensão *.EXE*. isto pode ser observado conforme a Figura 1.



**Figura 1 - Disparo do executável**

# ALGORITMOS DE CLASSIFICAÇÃO

Como delimitação do escopo deste trabalho os seguintes algoritmos foram selecionados: *Quick sort, Merge sort e Bubble sort*. Conforme os critérios pré-estabelecidos do trabalho de projeto de complexidade dos algoritmos de ordenação foram observados para escolha de cada algoritmo, o que segue:

* Usem técnicas de comparação;
* Complexidades diferentes;
* Consumo de memória extra diferente;
* Melhor, Pior e caso médio;
* Pelo menos uma das variáveis tenha que ter ordem significativamente diferente.

**Quadro 1 - Performance dos algoritmos de ordenação com técnicas de comparação**

| Algoritmo | Melhor | Média | Pior | Memória | Estável | Método |
| --- | --- | --- | --- | --- | --- | --- |
| *Quick sort* |  |  |  |  | Não | Particionamento |
| *Merge sort* |  |  |  |  | Sim | Junção |
| *Bubble sort* |  |  |  | 1 | Sim | Troca |

**Fonte: *Wikipedia* (2023).**

A escolha dos algoritmos, listados no Quadro 1, justificam-se pelo fato de possuírem diferentes complexidades para o melhor e pior caso, bem como possuírem métodos de classificação diferentes. Tal escolha se embasou na lista de desempenho de algoritmos do site *Wikipedia (2023)* que usam técnicas de comparação.

As diferenças entre os algoritmos também podem ocorrer nos quesitos técnicas de classificação e consumo de memória. Nos requisitos, em que, possivelmente, possam haver empate, desempatam-se com outros.

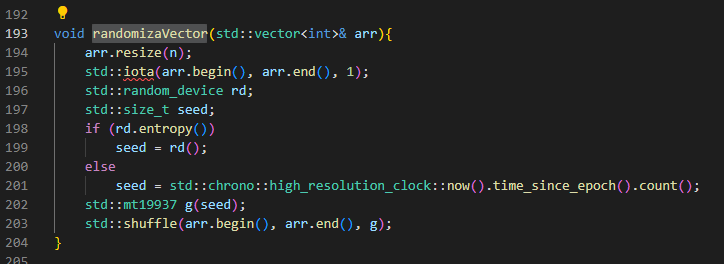
# Identificação dos algoritmos de classificação

De acordo com as demandas e os pré-requisitos do proposto trabalho, deverá existir previamente configurado um arquivo texto como o nome **“*entrada.txt”*.** Neste arquivo só podem existir duas linhas. A primeira linha conterá um número inteiro que se refere a instrução da quantidade de registros sequenciais em um *array*, os quais serão submetidos ao algoritmo de classificação. E a segunda conterá caracteres alfanuméricos com o nome do algoritmos que vai ordenar esta quantidade sequencial descrita na primeira linha. O nome do algoritmo deve ser idêntico ao nome da função que se encontra no código. Os nomes, no código, para disparar os algoritmos foram padronizados como:

* ***buubleSort*** para *Bubble Sort*;
* ***mergeSort*** para *Merge Sort*;
* ***quickSort*** para *Quick Sort*.

# Cálculo da complexidade

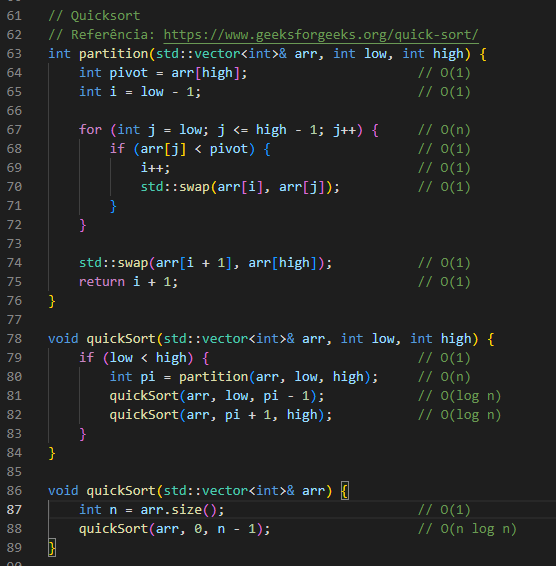
Este tópico será dedicado à demonstração do cálculo de complexidade dos algoritmos Quick Sort, Merge Sort e Bubble Sort em seus melhores, piores e médios casos. Porém, no caso médio, há uma configuração diferente: cinco listas com a mesma quantidade sequencial de números serão distribuídas de forma aleatória. Para tanto, criou-se uma função no código ***“ramdomizaVector()”***(Figura 2), para evitar que a distribuição aleatória da sequência de números nos cinco vetores não se repita, isto ocorre a partir da linha 197 a 201.



**Figura 2 - Função de distribuição randômica da sequência do array**

# *Quick Sort* - complexidade

A implementação do código em *C++* do algoritmo *Quick Sort* encontra-se disponível em *GeeksforGeeks* (2023), conforme ilustração da análise de complexidade da Figura 3.



**Figura 3 - Função de distribuição randômica da sequência do array**

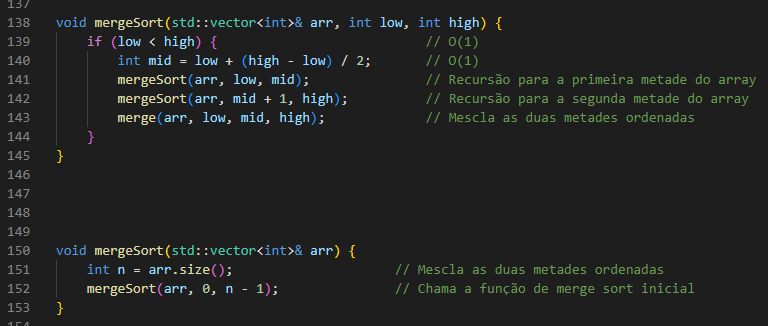
Para o pior caso, segundo Carmen e outros autores (2012, p.154), o *Quick sort “(…) tempo de execução* ***Q(n2)*** *ocorre quando o arranjo de entrada já está completamente ordenado — uma situação comum na qual a ordenação por inserção é executada no tempo* ***O(n)****”*. Já Press [2000, p. 450], relata que o pior caso está em um desbalanceamento das subsequências, em que *“(...) uma sequência tem todos os elementos restantes e a outra nenhum.”* Segundo os autores, o pior caso seria **n-1**. elementos e um com 0 elementos, como descrito na seguinte equação:

De acordo com os pesquisadores (Cormen et al., 2012; Preiss, 2000), que concordam no melhor caso, este algoritmo necessita dividir o problema em outros menores, posicionando o pivô na metade da lista, em outras palavras, o tamanho **n/2**. A expressão abaixo expressa o resultado:

Ainda seguindo o raciocínio dos mesmos autores em relação ao particionamento balanceado, ou o caso médio, a criação de uma divisão de 9 para 1 proporcionalmente, resulta em uma compensação entre os casos pior e bom. Isto pode ser observado pela seguinte expressão (Cormen, et al., 2012; Preiss, 2000):

# *Merge Sort* - complexidade

A função *Merge Sort* utilizada no código fonte foi uma implementação em C++ disponível em GeeksforGeeks (2023). A Figura 4 exibe o código do algoritmo em três funções: a principal, “*mergeSort()”,* ganhou o mesmo nome com construtores distintos para receberem diferentes parâmetros; uma outra que dividiu o array em duas partes e ativa a recursão de chamada; e, a maior função, merge (Figura 5), que foi responsável por unir as duas listas ordenadas.



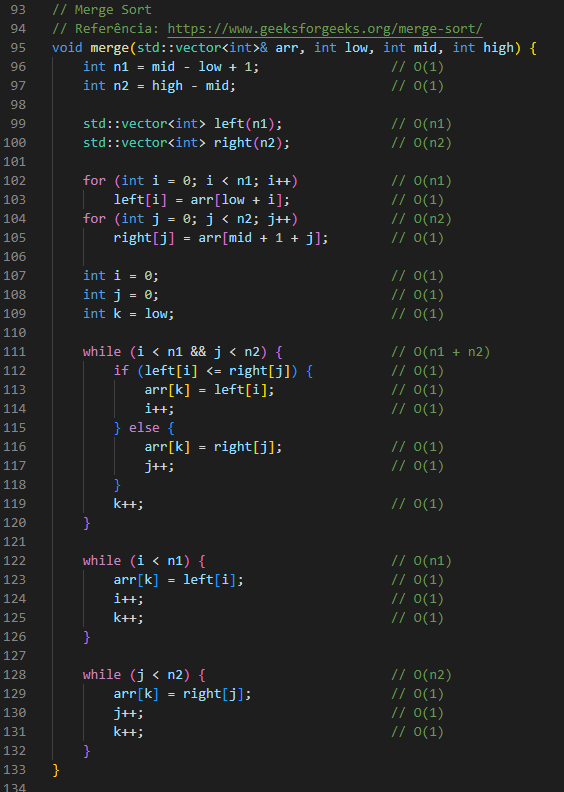
**Figura 4 - Código *Merge sort* função mergeSort recursiva e retorno**

Conforme Cormen e outros autores (2012, p. 38-39), os algoritmos de divisão e conquista, como o caso do *Merge Sort*, por seu código efetuar as divisões sempre ao centro, tendem a gerar uma constância entre os casos (melhor, pior e médio), tendo sua equação definida como segue:

Segundo ChatGPT (2023), a análise de complexidade assintótica do *Merge Sort possui*:

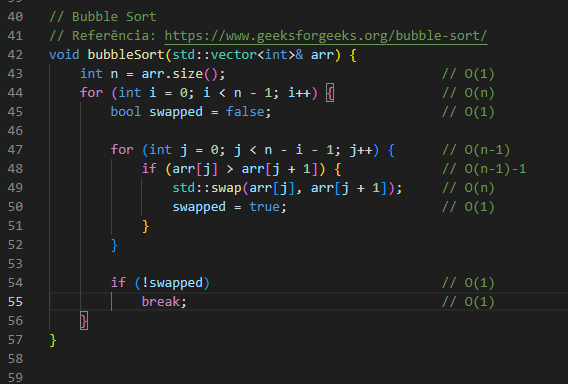
* *“a função merge com um loop que executa no máximo* ***n1 + n2*** *vezes, onde n1 e n2 são os tamanhos das metades do array a serem mescladas, portanto, a complexidade dessa função é O(n1 + n2);*
* *a função mergeSort, recursiva, que divide o array ao meio em cada chamada recursiva, logo, o número total de chamadas recursivas é O(log n), onde n é o tamanho do array.”*

Deste modo, resume-se que a complexidade do Merge Sort, no melhor caso será O(n log n), ocorrendo quando o array estiver completamente desordenado ou já estando ordenado.



**Figura 5 - Código *Merge sort* função merge**

# *Bubble Sort* - complexidade

O algoritmo *bubble sort*, em termos de codificação, é um algoritmo bem simples (ver Figura 6) e muito conhecido, baseado em troca de valores, conforme Preiss (2000). Apesar de sua simplicidade, ele pode ter seu desempenho alterado pela quantidade de amostras na classificação (Cormen, 2012). Para o código fonte deste trabalho optou-se por uma implementação em C++ cuja fonte encontra-se disponível em GeeksforGeeks (2023). 

**Figura 6- Código e complexidade do algoritmo *bubble sort***

Conforme pode-se observar na Figura 6, a função equivalente ao algoritmo bubble sort recebe o mesmo nome *“bubbleSort()”* , tendo um único array como parâmetro correspondente à sequência de números para ordenação. Todo o algoritmo se baseia em dois loops, cuja ideia básica foi varrer a lista no primeiro loop, linha 44, em sequência. O segundo loop, linha 47, inicia a partir da posição atual do loop externo e continua com varreduras até o final da lista. Na linha 48 fez-se os testes dos valores, com a finalidade de verificar se o valor da posição do loop externo é menor que o valor do loop interno, ocorrendo uma troca de valores na linha 49.

A complexidade do algoritmo bubble sort pode ser analisado da seguinte forma:

* No melhor caso, supondo que o array está ordenado, o algoritmo ainda executará ***n-1*** iterações no loop externo, mas as iterações internas serão interrompidas antes da primeira troca, pois o array já está ordenado. Portanto, a complexidade no melhor caso é ***O(n)***;
* No pior caso, o algoritmo precisará executar n-1 iterações no ***loop 'i'*** e para cada iteração externa, precisará executar ***n-i-1*** iterações internas (o *loop* 'j'). Portanto, o número total de iterações será ***(n-1) + (n-2) + ... + 2 + 1***, que é uma soma de uma progressão aritmética, resultando em ***(n-1)\*(n-2)/2***. Assim, a complexidade assintótica do *Bubble Sort* no pior caso é **O(n^2)**.
* No caso médio, o *Bubble Sort* executa ***(n-1)\*(n-2)/2*** iterações em média, resultando em uma complexidade assintótica de ***O(n^2)***.

# DESEMPENHO

Para cumprir as demandas deste trabalho, este tópico foi dedicado à análise de desempenho dos três algoritmos do experimento. E para atingir a meta dos testes de desempenho, fez-se necessário a aplicação de algumas tecnologias, que neste caso, usou as seguintes plataformas e equipamentos:

* **Linguagem de Programação**: Por ser uma linguagem de maior desempenho optou-se pelo uso da linguagem C++ para rodar algoritmos de classificação nas casas de milhões de registros.
* **Hardware**: Para executar os algoritmos, usou-se um notebook *Acer Nitro* 5 *Windows* 11, processador Intel *Core* i5, 8 gBytesgigabytes de RAM, HD 512 gigabytes NvMa SSD e placa de vídeo *NVidia GForce* 4 gigabytes.
* **Plataforma**: Utilizou-se a plataforma de desenvolvimento Anaconda, especificamente o prompt anaconda. Também fez necessário a instalação das bibliotecas Cuda e CDNN bem toda a configuração necessária para integrar o processamento da placa de vídeo.

# Arquivo de saída

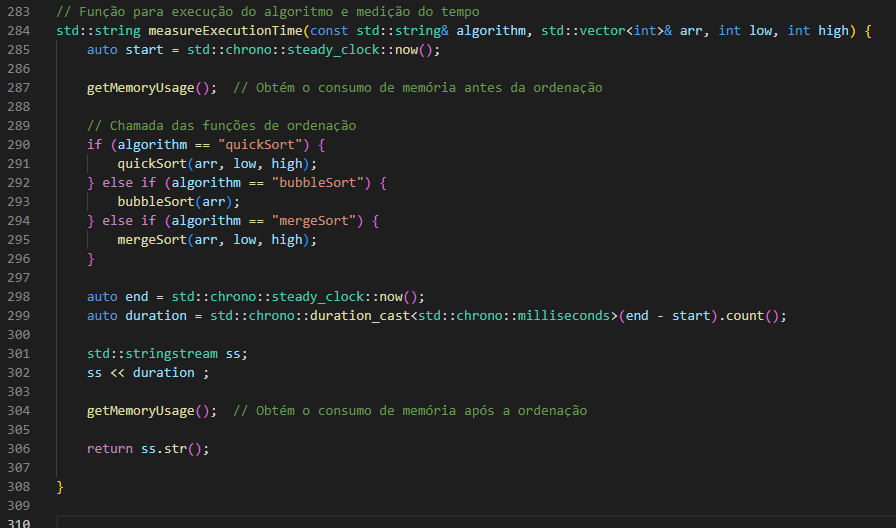
A proposta para criação do arquivo de saída foi um arquivo no formato “csv” com a estrutura:

* Data no formato “AAAA-MM-DD HH mm ss;
* Algoritmo, conforme padronização do código;
* Quantidade “N” de registros definidos no arquivo de entrada;
* Tipo do caso de desempenho (melhor, pior e médio);
* Tempo medido em milissegundos;
* Memória utilizada em megabytes.

# Codificação e métricas de desempenho

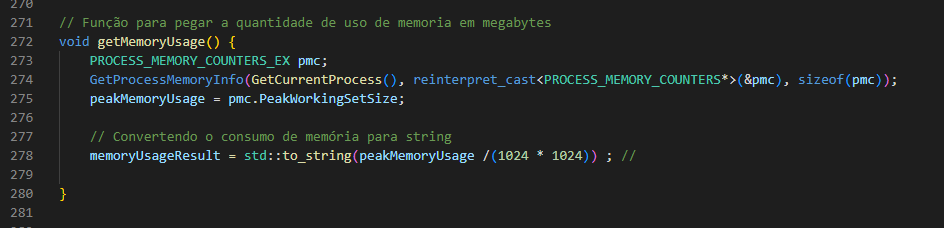
No desenvolvimento do código, algumas funções foram criadas com o objetivo de coletar as métricas em tempo de execução em cada algoritmo. A Figura 7 mostra a primeira função, “*measureExecutionTime****()”,*** que é responsável pela métrica de consumo de tempo, Todas variáveis que guardam as métricas são globais, isto foi um artifício que tornou mais fácil a coleta. Basicamente a função recebe três parâmetros, que são:

* Nome do algoritmo: Receber o nome do algoritmo facilita muito na questão de redução de código, pode-se observar, a partir da linha 290 a 295, que há um teste a respeito do nome do algoritmo passado no arquivo de entrada, este mesmo nome dispara a execução do algoritmo.
* Vetor: Este parâmetro corresponde ao *array* gerado com a sequência de números que serão ordenados pelo algoritmo.
* *Low e High*: Já estes dois parâmetros são apenas utilizados nos algoritmos *Quick* e *Merge Sort.*



**Figura 7- Código e complexidade do algoritmo *bubble sort***

Na própria função de tempo guarda-se os valores iniciais do tempo e consumo de memória. A ideia é colher novamente os mesmos valores e calcular a diferença. Inicialmente o tempo foi colhido em segundo, mas por muitos erros em tempo de execução alterou-se o código para colher tudo em milissegundos e posteriormente converter nos gráficos.



**Figura 8- Função de métrica de consumo de memória**

A função de consumo de memória *getMemoryUsage()* foi usada para pegar a quantidade de memória utilizada na execução do algoritmo. Na linha 274, pode-se observar como a função consegue obter a métrica de consumo de memória, usa-se uma API do windows para obter este consumo no gerenciamento dos processos de execução.

# Resultados de tempo

Com o arquivo de resultados dos 3 algoritmos, os testes do tempo, medido em milissegundos, foram resumidos no Quadro 2, no qual se observou que no caso médio o melhor tempo foi do *quickSort*, no melhor caso o *bubbleSort* teve os menores tempos, já o pior caso quem ganhou foi o *mergeSort*. Durante a execução dos algoritmos, especialmente, com a quantidade menor que dez mil, o trabalho demandou a medida em segundo, o que ocasionou bastante erro, em tempo de execução, devido a velocidade do hardware. Assim, optou-se por utilizar a medida em milissegundo para mitigar os valores de tempo zerado, que ainda continuaram.

**Quadro 2 - Desempenho do Tempo medido em milissegundos**

| **Qnt. (n)** | **100** | **1.000** | **10.000** | **100.000** | **1.000.000** | **10.000.000** |
| --- | --- | --- | --- | --- | --- | --- |
| **médio** |  |  |  |  |  |  |
| bubbleSort | 0 | 3 | 356 | 38097 | 3932556 |  |
| mergeSort | 0 | 0 | 5 | 59 | 640 | 11083 |
| quickSort | 0 | 0 | 1 | 15 | 199 |  |
| **melhor** |  |  |  |  |  |  |
| bubbleSort | 0 | 0 | 0 | 0 | 2 | 26 |
| mergeSort | 0 | 0 | 5 | 50 | 534 | 7255 |
| quickSort | 0 | 3 | 371 | 36845 | 5132846 |  |
| **pior** |  |  |  |  |  |  |
| bubbleSort | 0 | 4 | 404 | 40780 | 4214605 | 436655353 |
| mergeSort | 0 | 0 | 5 | 53 | 540 | 9754 |
| quickSort | 0 | 2 | 225 | 22541 | 2299824 |  |

Os testes no *Bubble Sort* e *Quick Sort* não terminaram até a data de entrega deste relatório, por isso os valores não estão preenchidos. Durante a execução destes algoritmos ocorreram estouros dos limites da pilha de memória, à medida que os vetores de ordenação ultrapassaram cinco dígitos, sendo ajustados de forma empírica, reiniciando a compilação várias vezes, até atingir 80 megas de limite para a pilha de memória.

# Resultados de memória

No Quadro 3 foram listados o consumo de memória, medidos em megabytes. Observou-se que com até seis dígitos na quantidade de “n” houve um certo equilíbrio no consumo de memória entre os algoritmos. Acima deste limite o quickSort se eleva bastante em relação aos demais.

**Quadro 3 - Desempenho de memória medido em megabytes**

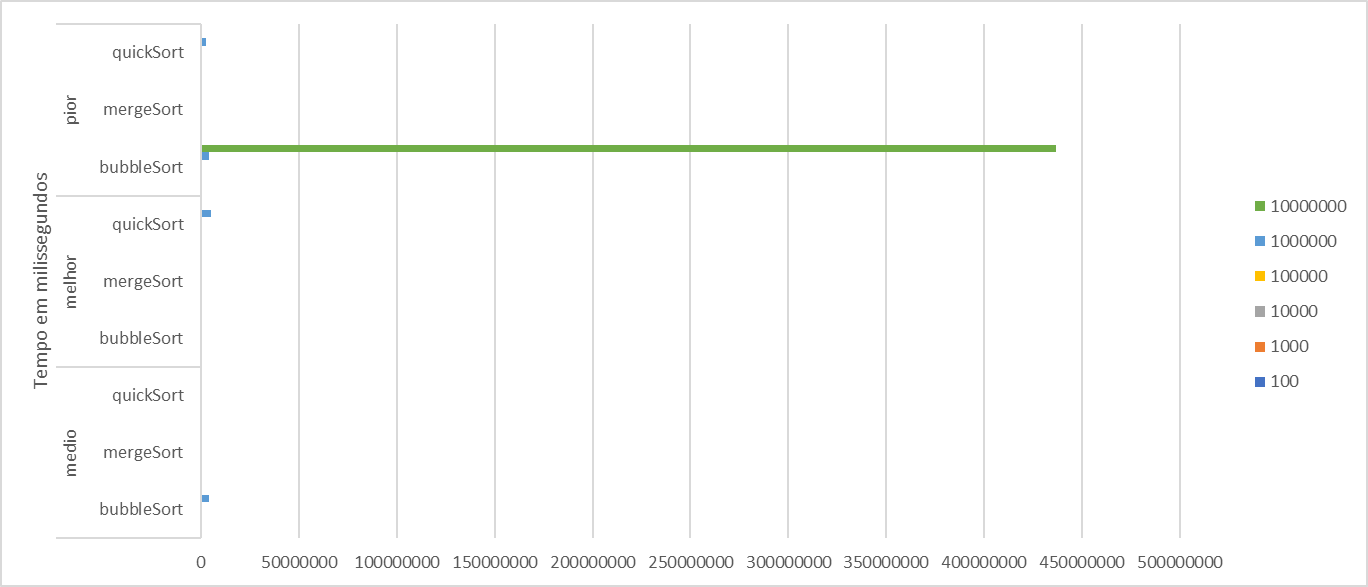
| **Qnt. (n)** | **100** | **1.000** | **10.000** | **100.000** | **1.000.000** | **10.000.000** |
| --- | --- | --- | --- | --- | --- | --- |
| **médio** |  |  |  |  |  |  |
| bubbleSort | 8 | 8 | 8 | 11 | 35 |  |
| mergeSort | 7 | 7 | 8 | 11 | 38 | 313 |
| quickSort | 7 | 7 | 8 | 17 | 83 |  |
| **melhor** |  |  |  |  |  |  |
| bubbleSort | 8 | 8 | 8 | 11 | 35 | 275 |
| mergeSort | 7 | 7 | 7 | 10 | 38 | 313 |
| quickSort | 7 | 7 | 8 | 17 | 71 |  |
| **pior** |  |  |  |  |  |  |
| bubbleSort | 8 | 8 | 8 | 11 | 35 | 275 |
| mergeSort | 7 | 7 | 8 | 11 | 38 | 313 |
| quickSort | 7 | 7 | 8 | 17 | 74 |  |

Registrando que ocorreu a mesma situação de incluir apenas os resultados dos testes concluídos.

# ANALISANDO OS RESULTADOS

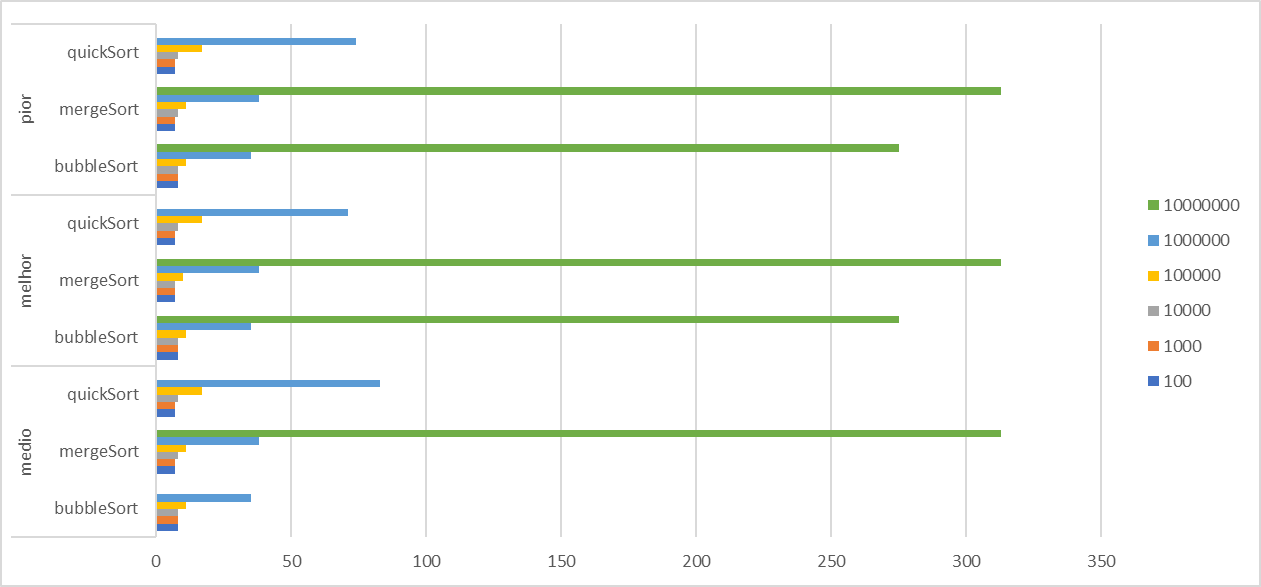
Quando se analisam os dados obtidos no arquivo de resultados, em relação a métrica do tempo de execução, percebe-se que no melhor caso o *bubbleSort* se manteve estável com os melhores tempos, porém nos demais casos ele não performa como os demais, conforme o Gráfico 1.

A maior surpresa foi o desempenho do *mergeSort,* no pior caso com tempo de execução menor que o *quickSort*, o qual performou muito bem no caso médio.



**Gráfico 1 - Desempenho dos algoritmos por tempo de execução**

Em se tratando do consumo de memória, até dez mil na quantidade de “n”, os algoritmos se equiparam. Todavia, acima dos cem mil, o *bubbleSort* destaca-se pelo menor consumo de memória e sua estabilidade em todos os casos, ilustrado no Gráfico 2.



**Gráfico 2 - Comparativo do uso de memória do algoritmos**

# CONSIDERAÇÕES FINAIS

Por fim, entende-se que o relatório cumpriu com as solicitações propostas para a tarefa de complexidade, quando demonstrou que o objetivo foi atendido.

# Referências

ChatGPT, [ChatGPT (openai.com)](https://chat.openai.com/), Acessado 9 jun, 2023.

Cormen, Thomas H, et al. **Algoritmos:** teoria e prática. Tradução Arlete Simille Marques de: Introduction to algorithms, 3rd ed. Rio de Janeiro: Elsevier, 2012. il. ISBN 978-85-352-3699-6

GeeksforGeeks. Bubble Sort – Data Structure and Algorithm Tutorials. 26 mai, 2023. Disponível em <https://www.geeksforgeeks.org/merge-sort/>. Acessado em 08 jun. 2023.

Preiss, Bruno R. **Estruturas de dados e algoritmos:** padrões de projetos orientados a objeto com Java. Tradução: Elizabeth Ferreira. Rio de Janeiro: Campos, 2000. ISBN: 85-7110-0693-0.

Wikipedia. *Sorting algorithm, 11 abr*, *2023.*

Disponível em: <https://en.wikipedia.org/wiki/Sorting_algorithm>. Acessado em 9 jun, 2023.