UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI
CIÊNCIA DA COMPUTAÇÃO
AEDS II

# Trabalho Prático 2

Lucas Eduardo Bernardes de Paula,
Lucas Emanuel Pereira de Melo,
Messias Feres Curi Melo

São João Del Rei, Maio de 2023

**Alunos:**

Lucas Eduardo Bernardes de Paula
Lucas Emanuel Pereira de Melo
Messias Feres Curi Melo

Prof. Rafael Sachetto Oliveira

**Palavras-chave:** Ordenação por Seleção, Inserção, Mergesort, Quicksort, Shellsort, Heapsort / Tempo de execução / Número de comparações / Número de movimentações.

São João Del Rei, Maio de 2023

# SUMÁRIO

# INTRODUÇÃO

O objetivo do trabalho prático proposto é permitir aos alunos a familiarização com os códigos dos algoritmos de Ordenação por Seleção, Ordenação por Inserção, Shellsort, Quicksort, Heapsort e Mergesort, que foram vistos em sala de aula, além da análise de seus respectivos tempos de execução, comparações e movimentações.

Ademais, o trabalho busca mostrar e fazer uma avaliação de tais tempos de execução e comparação sob cenários distintos, como diferentes tamanhos de vetores (20, 500, 5000, 10000, 200000), e as principais diferenças envolvendo a eficiência entre eles. Assim, com este tipo de comparação, torna-se evidente os pontos positivos e negativos de cada abordagem naquela determinada situação.

# IMPLEMENTAÇÃO

Previamente, para a execução do projeto, utilizamos os códigos de ordenação disponibilizado no site “Sort Visualizer”, dessa forma, criamos uma struct chamada Registro, onde pode ser usada como o registro pequeno ou o grande, basta comentar a variável mensagem e já estará testando o registro pequeno, como observado na imagem abaixo. Podemos, também, alterar o tamanho de elementos desejado, como 20, 500, 10000, entre outros, para serem testados. Em seguida, puxamos as funções de ordenação e a integramos com a struct, colocando a no ‘registro.c’ e suas chamadas no ‘registro.h’.

```c
#define TAM 10000 // Altere o TAM para mais ou menos números de execução

typedef int chave;
typedef char mensagem;

typedef struct{
    chave chave;
    //mensagem mensagem[50][50]; // Comente ou descomente para ativar a chave grande
}Registro;
```

**Código para escolha do registro grande ou pequeno - Figura 1**

Com a struct pronta e as funções preparadas, fomos para a ‘main.c’, onde colocamos tudo em prática. Inicialmente, declaramos as variáveis a serem utilizadas ao decorrer do projeto, assim como, declaramos o chamamento da struct Registro. Logo em seguida, temos um switch onde poderá ser escolhido se queremos que o vetor gere uma ordem aleatória, uma ordem crescente ou uma ordem decrescente de dados, para assim, podermos testar o projeto em diversas abordagens.

```c
switch (3) { // Definir através do número qual ordem deseja executar
    case 1: // Ordem Aleatório
        for (int j = 0; j < TAM; j++) {
            temp[j].chave = rand() % TAM;
        }
        break;
    case 2: // Ordem Crescente
        crescente = rand() % TAM;
        for (int j = 0; j < TAM; j++, crescente++) {
            temp[j].chave = crescente;
        }
        break;
    case 3: // Ordem Decrescente
        decrescente = rand() % 10*TAM;
        for (int j = 0; j < TAM; j++, decrescente--) {
            temp[j].chave = decrescente;
        }
        break;
    default:
        printf("Números não definidos\n");
        break;
}
```

**Código para escolher a ordenação inicial do vetor - Figura 2**

Ao escolher a ordem de geração dos elementos, quantos elementos são e qual registro usar, o software irá executar 10 vezes cada função de ordenação, sendo realizado com os mesmos valores em todas as funções, para assim, não dar uma discrepância entre os testes, facilitando a análise. Após adquirir os dados, será exibido no terminal a média dos resultados obtidos, trazendo o tempo, as combinações e as movimentações realizadas, todos sendo divididos por 10, representando a média.

```c
printf("Seleção\n");
printf("Tempo médio --> %lf\n", time_total[0]/10);
printf("Média de comparações --> %lu\n", comparacao[0]/10);
printf("Média de movimentações --> %lu\n\n", movimentacao[0]/10);
```

**Código para imprimir os resultados - Figura 3**

## Complexidade dos Algoritmos

- **Melhor Caso**

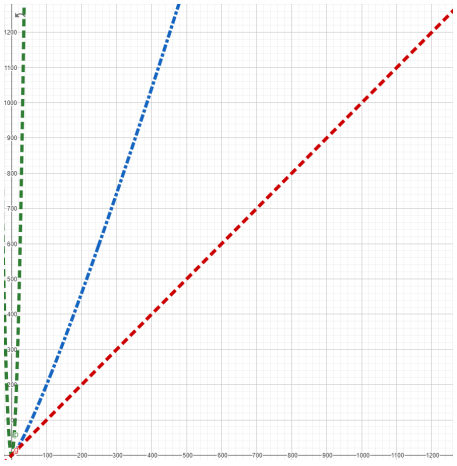

**Gráfico de Complexidade - Figura 4**

**Linha Verde**: abrange o algoritmo de Seleção que em seu melhor caso apresenta uma complexidade quadrática, ou O(n^(2)).

**Linha Azul**: abrange os algoritmos de ordenação Shellsort, Quicksort, Heapsort e Mergesort que em seu melhor caso apresentam uma complexidade log linear, ou O(n log n).

**Linha Vermelha**: Abrange o algoritmo de Inserção que em seu melhor caso apresenta uma complexidade linear, ou O(n)

- **Caso Médio**

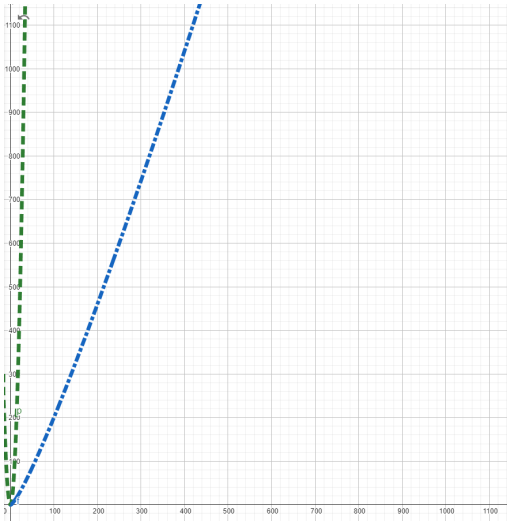

**Gráfico de Complexidade - Figura 5**

**Linha Verde**: abrange os algoritmos de Inserção e Seleção que em seu caso médio apresentam uma complexidade quadrática, ou O(n^(2)).

**Linha Azul**: abrange os algoritmos de ordenação Shellsort, Quicksort, Heapsort e Mergesort que em seu caso médio apresentam uma complexidade log linear, ou O(n log n).

- **Pior Caso**

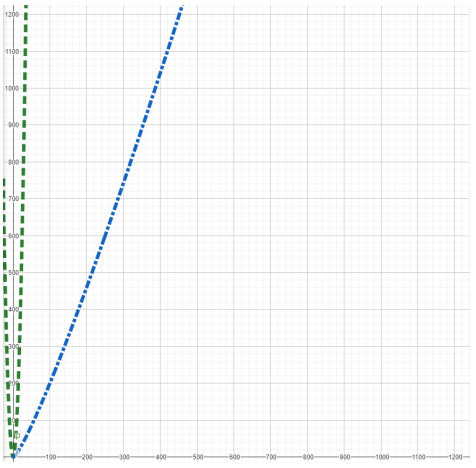

**Gráfico de complexidade - Figura 6**

**Linha Verde**: abrange os algoritmos de Inserção, Seleção, Quicksort e Shellsort que em seu pior caso apresentam complexidade quadrática, ou O(n^(2)).

**Linha Azul**: abrange os algoritmos de ordenação Heapsort e Mergesort que em seu pior caso apresentam uma complexidade log linear, ou O(n log n).

**Funções Aleatórias - Chave Pequena**

| Tempo médio por segundo | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 0.000002 | 0.000349 | 0.065053 | 0.164377 | 60.168316 |
| **Inserção** | 0.000002 | 0.000201 | 0.38471 | 0.101885 | 37.163235 |
| **Shellsort** | 0.000002 | 0.000064 | 0.002072 | 0.002968 | 0.103392 |
| **Quicksort** | 0.000002 | 0.000055 | 0.001603 | 0.002057 | 0.053462 |
| **Heapsort** | 0.000003 | 0.000097 | 0.003191 | 0.004951 | 0.099601 |
| **Mergesort** | 0.000003 | 0.000074 | 0.002733 | 0.002815 | 0.062096 |

| Média de comparações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 190 | 124750 | 12497500 | 49995000 | 19999900000 |
| **Inserção** | 104 | 63007 | 6232538 | 25000947 | 9994024442 |
| **Shellsort** | 86 | 6139 | 112401 | 261358 | 9828833 |
| **Quicksort** | 119 | 7432 | 107091 | 243712 | 6608951 |
| **Heapsort** | 149 | 9370 | 135291 | 295473 | 8876404 |
| **Mergesort** | 259 | 13335 | 183828 | 397735 | 10548424 |

| Média de movimentações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 57 | 1497 | 14997 | 29997 | 599997 |
| **Inserção** | 123 | 63506 | 6237537 | 25010946 | 9994224441 |
| **Shellsort** | 96 | 8695 | 179885 | 439245 | 20187923 |
| **Quicksort** | 91 | 5297 | 73340 | 173200 | 4539413 |
| **Heapsort** | 144 | 8077 | 114204 | 248334 | 6700096 |
| **Mergesort** | 88 | 4488 | 61808 | 133616 | 3547856 |

- **Número de elementos → 20:**
    - **Seleção**: por ser simples e fácil de implementar, a ordenação por seleção neste caso se mostra extremamente eficiente para um vetor de 20 elementos gerados aleatoriamente.

- **Inserção**: tal tipo de ordenação também é eficiente para vetores pequenos, portanto, sua análise de complexidade seria o caso médio, pois o vetor foi gerado aleatoriamente e não estava em ordem crescente (melhor caso) ou decrescente (pior caso). Possui o mesmo tempo de execução da ordenação por seleção, porém com menos comparações e mais movimentações.
- **Shellsort**: também foi extremamente eficiente para o vetor gerado aleatoriamente, com o mesmo tempo de execução dos dois anteriores; porém, com ainda menos número de comparações e movimentações.
- **Quicksort**: assim como os três primeiros, seu tempo de execução foi 0,000002 segundos, portanto, para tal vetor gerado aleatoriamente foi extremamente eficiente.
- **Heapsort**: juntamente com o Quicksort, mostrou ter uma ótima eficiência neste caso ao comparar o tempo de execução, número de movimentações e comparações.
- **Mergesort**: também foi extremamente eficiente neste caso; com uma ótima média de valores.

**Opinião do Grupo:** Após uma análise detalhada de cada um dos algoritmos e um debate entre o grupo, podemos chegar a conclusão de que por se tratar de um valor muito pequeno, não importa muito a escolha do algoritmo pois todos vão ter tempos muito rápidos e extremamente parecidos, porém, caso fosse necessário escolher um, houve um consenso sobre a escolha do shellsort em razão do menor número de troca e movimentações.

- **Número de elementos → 500:**
  - **Seleção**: a ordenação por seleção neste caso se mostra eficiente para um vetor de 500 elementos gerados aleatoriamente; porém, seu tempo de execução foi 0,000349 e realiza muito mais comparações que os outros métodos de ordenação.
  - **Inserção**: um pouco mais eficiente que por seleção; avaliando a média do tempo de execução e os números de comparações.
  - **Shellsort**: mais eficiente que por seleção por seleção e por inserção; avaliando a média tempo de execução, números de comparações e de movimentações.
  - **Quicksort**: consegue ser mais eficiente que os três métodos anteriores, com o tempo de execução de 0,000055; segundo a média do tempo de execução, números de comparações e de movimentações.
  - **Heapsort**: juntamente com o Quicksort, mostrou ter uma ótima eficiência neste caso ao comparar o tempo de execução, número de movimentações e comparações.
  - **Mergesort**: também foi extremamente eficiente neste caso, assim como os três anteriores; com uma ótima média de valores.

**Opinião do Grupo:** Após uma análise minuciosa de cada um dos algoritmos e uma discussão em grupo, chegamos à conclusão de que quatro algoritmos se destacam neste caso específico: Shellsort, Quicksort, Mergesort e Heapsort. Todos os quatro apresentaram tempos de execução muito próximos entre si. No entanto, ao optar por um

único algoritmo, o grupo escolheria o Quicksort, devido ao seu tempo de execução mais rápido em comparação aos outros algoritmos, além de apresentar um número de movimentações e comparações similar aos demais.

- **Número de elementos → 5000:**
    - **Seleção:** a ordenação por seleção, por ser mais eficiente com vetores com valores menores, neste caso de 5000 elementos já não se mostra tão eficiente quanto os outros, apesar de sua eficácia. Possui um tempo de execução de 0,065053 e faz 12497500 comparações.
    - **Inserção:** um pouco mais eficiente do que por seleção neste caso; avaliando a média do tempo de execução e os números de comparações.
    - **Shellsort:** mais eficiente que por seleção por seleção e por inserção; avaliando a média tempo de execução, números de comparações e de movimentações.
    - **Quicksort:** consegue ser mais eficiente que todos os outros métodos de ordenação neste caso, com o tempo de execução de 0,001603; segundo a média do tempo de execução, números de comparações e de movimentações.
    - **Heapsort:** junto com o Quicksort, mostrou ter uma excelente eficiência neste caso ao comparar a média de seus valores.
    - **Mergesort**: também foi extremamente eficiente neste caso; com uma ótima média de valores e um tempo de execução ainda menor que o Heapsort.

**Opinião do Grupo:** Após uma análise minuciosa de cada um dos algoritmos e uma discussão em grupo, chegamos à conclusão de que três algoritmos se destacam neste caso específico: Shellsort, Quicksort, Mergesort. No entanto, o grupo optaria pela escolha do quicksort em razão do menor tempo, além do menor número de comparações e movimentações.

- **Número de elementos → 10000:**
    - **Seleção:** novamente, por ser um número maior de elementos deste vetor e apresentar uma complexidade quadrática, o método de seleção não é o mais indicado: tempo de execução igual a 0,0164377 e 49995000 como média de comparações.
    - **Inserção:** não é muito diferente da ordenação por seleção neste caso, pois também não é o mais indicado para esta quantidade de elementos. Seu tempo de execução foi 0,101885; porém realizou menos comparações e mais movimentações.
    - **Shellsort:** mais eficiente que os métodos por seleção e inserção por funcionar muito bem em quantidades moderadas/grandes de elementos, visto a média do tempo de execução, nº de movimentações e comparações.
    - **Quicksort:** muito parecido com o método de ordenação Shellsort; porém com médias menores de nº de comparações e movimentações.
    - **Heapsort:** também com ótima eficiência e média de comparações/movimentações; porém, seu tempo de execução é um pouco maior que do Shellsort e Quicksort.

- **Mergesort:** eficiência similar com a do Quicksort; ótimas médias de valores no geral, porém com mais comparações.

**Opinião do Grupo:** Após um debate entre o grupo e uma análise fundamentada de cada um dos algoritmos foi possível notar como a entrada afeta o tempo de execução dos algoritmos de ordenação por Inserção e por Seleção, demorando mais que dez vezes o tempo de execução dos outros algoritmos, além disso, o grupo chegou a conclusão sobre a utilização do Quicksort nesse caso em específico, essa escolha ao invés do Shellsort ou do Mergesort ocorre em razão do número de trocas e do número de comparações realizadas entre eles, com o Shellsort realizando mais que o dobro das comparações do quicksort e o Mergesort 1,6 mais comparações

- **Número de elementos → 200000:**
  - **Seleção:** extremamente ineficiente para este caso em razão da sua complexidade quadrática. Basicamente 1 minuto (60s) de tempo de execução e valores gigantescos de comparações e movimentações.
  - **Inserção:** apesar de demorar 23 segundos a menos que o de seleção, também é muito ineficiente para casos com vetores com essa quantidade de elementos.
  - **Shellsort:** muito mais eficiente que os métodos de seleção e inserção, tendo seu tempo médio de execução igual a 0,1 segundos.
  - **Quicksort:** o método de ordenação mais eficiente de todos para esse caso, tendo seu tempo médio de 0,05 segundos.
  - **Heapsort:** eficiência similar ao Shellsort, com basicamente o mesmo tempo de execução; porém com menos comparações e mais movimentações.
  - **Mergesort:** juntamente com o Quicksort, possui um tempo de execução extremamente eficiente.

**Opinião do Grupo:** Nesse caso em específico, a utilização dos algoritmos de ordenação por Seleção e por Inserção já se torna praticamente inviável devido as suas complexidades quadráticas, já os algoritmos Quicksort e Mergesort apresentam comportamentos parecidos com pequena diferenças de tempo, o Quicksort acaba realizando mais movimentações e o Merge mais comparações, deixando muito similar a usabilidade de cada um deles, em razão disso, o grupo chegou à conclusão que o uso de qualquer um dos dois possuiria resultados parecidos.

**Funções Crescentes - Chave Pequena**

| Tempo médio | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 0.000002 | 0.000317 | 0.034712 | 0.137899 | 53.758827 |
| **Inserção** | 0.000001 | 0.000003 | 0.000029 | 0.000063 | 0.001320 |
| **Shellsort** | 0.000001 | 0.000015 | 0.000221 | 0.000591 | 0.013091 |
| **Quicksort** | 0.000003 | 0.000047 | 0.000768 | 0.001918 | 0.035767 |
| **Heapsort** | 0.000002 | 0.000084 | 0.001517 | 0.002512 | 0.066491 |
| **Mergesort** | 0.000003 | 0.000045 | 0.000714 | 0.001511 | 0.032579 |

| Média de comparações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 190 | 124750 | 12497500 | 49995000 | 19999900000 |
| **Inserção** | 19 | 499 | 4999 | 9999 | 199999 |
| **Shellsort** | 62 | 3506 | 55005 | 120005 | 3200006 |
| **Quicksort** | 72 | 4763 | 69772 | 151494 | 4292051 |
| **Heapsort** | 178 | 10317 | 146123 | 317541 | 8503544 |
| **Mergesort** | 243 | 11747 | 160619 | 346239 | 9083519 |

| Média de movimentações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 57 | 1497 | 14997 | 29997 | 599997 |
| **Inserção** | 38 | 998 | 9998 | 19998 | 399998 |
| **Shellsort** | 0 | 0 | 0 | 0 | 0 |
| **Quicksort** | 208 | 10204 | 131676 | 281426 | 7633816 |
| **Heapsort** | 162 | 8710 | 121866 | 263914 | 6998322 |
| **Mergesort** | 88 | 4488 | 61808 | 133616 | 3537856 |

- **Número de elementos → 20:**
    - **Seleção**: por ser simples e fácil de implementar, a ordenação por seleção neste caso é muito eficiente devido ao fato do vetor já estar ordenado crescentemente, sendo esse o seu melhor caso.

- **Inserção**: tal tipo de ordenação também é extremamente eficiente para vetores pequenos em ordem crescente pois este também é seu melhor caso, fazendo exatamente (n - 1) comparações (19).
- **Shellsort**: também foi extremamente eficiente para o vetor de ordem crescente; porém, com uma média ainda menor de comparações e movimentações (sendo esta exatamente 0).
- **Quicksort**: assim como os três primeiros, seu tempo de execução foi 0,000002 segundos, portanto, foi extremamente eficiente.
- **Heapsort e Mergesort:** assim como todos os anteriores, possuem tempo médio de execução parecidos e muito eficientes.

**Opinião do Grupo:** Após uma análise detalhada de cada um dos algoritmos e um debate entre o grupo, podemos chegar a conclusão de que por se tratar de um valor muito pequeno, não importa muito a escolha do algoritmo pois todos vão ter tempos muito rápidos e extremamente parecidos, porém, caso fosse necessário escolher um , o grupo optaria pela escolha do algoritmo de Inserção em razão de um vetor ordenado ser o seu melhor caso, onde ele apresenta complexidade O(n), uma complexidade melhor que todos os outros algoritmos de ordenação.

- **Número de elementos → 500:**
  - **Seleção**: a ordenação por seleção neste caso se mostra eficiente para um vetor de 500 elementos ordenados, porém, seus números de comparações e movimentações são relativamente grandes.
  - **Inserção**: muito mais eficiente do que por seleção por ser o seu melhor caso, com um número de comparações/movimentações extremamente menor, sendo o de comparações igual a 499.
  - **Shellsort**: também extremamente eficiente, avaliando a média tempo de execução, números de comparações e de movimentações (sendo o nº de movimentações exatamente igual a 0).
  - **Quicksort**: extremamente eficiente assim como o anterior, porém com um número de movimentações relativamente grande (10204)
  - **Heapsort e Mergesort:** médias de execução de tempo eficientes; porém com maiores números de movimentações e comparações.

**Opinião do Grupo:** Após um debate entre o grupo e uma análise minuciosa a respeito de cada um dos algoritmos, foi chegado a um consenso na utilização do algoritmo de ordenação por Inserção, em razão de ser um vetor já ordenado, o método de Inserção apresenta uma complexidade linear, fazendo com que ele seja mais rápido que os outros algoritmos, além de realizar menos comparações. Um fato que chamou a atenção do grupo durante a análise foi o fato do Shellsort não realizar movimentações, como o Shellsort é uma extensão do algoritmo de Inserção, ele também é extremamente eficiente em vetores já ordenados, fazendo com que ele seja muito eficiente nesse caso.

- **Número de elementos → 5000:**
  - **Seleção**: se mostra relativamente eficiente para um vetor de 5000 elementos ordenados, porém, seus números de comparações e movimentações são muito grandes quando comparados com outros menores.

- **Inserção**: neste caso, se mostrou o método de ordenação mais eficiente, com um número de comparações/movimentações extremamente menor, sendo o de comparações igual a 4999.
- **Shellsort**: um pouco mais eficiente do que por seleção, avaliando a média de tempo de execução, números de comparações e de movimentações (sendo o nº de movimentações exatamente igual a 0).
- **Quicksort**: extremamente eficiente com ótimo tempo de execução; porém, números de comparações e movimentações bem grandes.
- **Heapsort**: média de tempo de execução um pouco maior mas ainda eficiente, com bons números no geral.
- **Mergesort**: extremamente eficiente apesar da média de comparações e movimentações serem relativamente grandes.

**Opinião do Grupo:** Após uma análise minuciosa e um debate entre os membros do grupo, chegamos a um consenso em relação à utilização do algoritmo de ordenação por Inserção. Essa escolha se baseia no fato de que, para um vetor já ordenado, o método de Inserção apresenta uma complexidade linear, tornando-o mais rápido em comparação aos outros algoritmos e realizando menos comparações. Novamente é relevante ressaltar a eficiência do Shellsort, tendo um tempo semelhante ao Inserção e não realizando nenhuma movimentação.

- **Número de elementos → 10000:**
  - **Seleção**: seus números de comparações e movimentações são muito grandes quando comparados com os outros, portanto não é a opção mais eficiente neste caso devido ao tamanho do vetor.
  - **Inserção**: se mostrou o método de ordenação mais eficiente, com um número de comparações/movimentações extremamente menor, sendo o de comparações igual a 9999.
  - **Shellsort**: eficiência similar ao de inserção, com ótimos números de comparações e de movimentações (sendo o de movimentações exatamente igual a 0).
  - **Quicksort**: extremamente eficiente apesar de sua média de movimentações ser relativamente grande comparado ao resto.
  - **Heapsort e Mergesort:** ambas extremamente eficientes, mesmo com a média de números de comparação/movimentação relativamente grandes.

**Opinião do Grupo:** Nesse caso em específico a diferença entre o algoritmo de Inserção e o Shellsort, já é mínima, fazendo com que a usabilidade dos dois seja extremamente parecida, com uma diferença de tempo praticamente imperceptível, fazendo com que a escolha de um dos dois tenha praticamente o mesmo resultado.

- **Número de elementos → 200000:**
  - **Seleção**: extremamente ineficiente. Seus números de comparações e movimentações são muito grandes quando comparados com os outros, portanto é a opção menos eficiente neste caso devido ao tamanho do vetor.

- **Inserção**: se mostrou o método de ordenação mais eficiente em relação ao tempo de execução, com um número de comparações muito pequeno (199999) em relação aos outros.
- **Shellsort**: menos eficaz do que por inserção, mas ainda assim eficiente após a observação dos números de comparações e de movimentações (sendo o nº de movimentações exatamente igual a 0).
- **Quicksort**: extremamente eficiente apesar de sua média de movimentações ser relativamente grande comparado ao resto.
- **Heapsort e Mergesort:** ambas extremamente eficientes, mesmo com a média de números de comparação/movimentação relativamente grandes.

**Opinião do Grupo**: É observável que a partir de um determinado número de elementos a serem ordenados, o algoritmo Shellsort começa a superar o algoritmo de Inserção em termos de desempenho, em razão de poder comparar elementos mais distantes ele permite otimizar as comparações e movimentações de elementos, resultando em uma redução significativa do tempo de execução em comparação com o algoritmo de Inserção.

**Funções Decrescentes - Chave Pequena**

| Tempo médio | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 0.000002 | 0.001135 | 0.037513 | 0.134353 | 56.645866 |
| **Inserção** | 0.000002 | 0.000360 | 0.038991 | 0.149671 | 65.424454 |
| **Shellsort** | 0.000001 | 0.000025 | 0.000388 | 0.001292 | 0.024848 |
| **Quicksort** | 0.000002 | 0.000049 | 0.001245 | 0.002052 | 0.063460 |
| **Heapsort** | 0.000002 | 0.000072 | 0.001334 | 0.002917 | 0.035126 |
| **Mergesort** | 0.000002 | 0.000045 | 0.000704 | 0.001430 | 0.037087 |

| Média de comparações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 190 | 124750 | 12497500 | 49995000 | 19999900000 |
| **Inserção** | 209 | 125249 | 12502499 | 50004999 | 20000099999 |
| **Shellsort** | 80 | 5116 | 78798 | 172578 | 4789145 |
| **Quicksort** | 77 | 47782 | 72024 | 153810 | 4386311 |
| **Heapsort** | 130 | 8411 | 125297 | 275316 | 7647635 |
| **Mergesort** | 235 | 11691 | 158419 | 341839 | 3537856 |

| Média de movimentações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 57 | 1497 | 14997 | 29997 | 599997 |
| **Inserção** | 228 | 125748 | 12507498 | 50014998 | 20000299998 |
| **Shellsort** | 108 | 6300 | 86340 | 187600 | 5367360 |
| **Quicksort** | 227 | 9797 | 130526 | 293868 | 7647736 |
| **Heapsort** | 126 | 7354 | 106874 | 233394 | 6391458 |
| **Mergesort** | 88 | 4488 | 61808 | 133616 | 3537856 |

- **Número de elementos → 20:**
    - **Seleção**: por ser um vetor pequeno, simples e fácil de implementar, a ordenação por seleção neste caso é muito eficiente.
    - **Inserção**: tal tipo de ordenação também é extremamente eficiente para vetores pequenos em ordem decrescente, mesmo fazendo 209 comparações e não apenas 19 igual na ordem crescente.
    - **Shellsort**: também foi extremamente eficiente para o vetor deste tamanho de ordem decrescente; porém, com uma média ainda menor de comparações e movimentações. Possui maior eficácia.
    - **Quicksort**: também é considerada eficiente, possuindo basicamente o mesmo tempo de execução das anteriores e números médios de comparação e movimentação.
    - **Heapsort/Mergesort:** assim como os anteriores, extremamente eficientes com um ótimo tempo de execução e média de comparações/movimentações.

**Opinião do Grupo**: Assim como no vetor aleatório e no vetor crescente, a diferença de tempo de ordenação com 20 elementos é praticamente nula, fazendo com que os resultados sejam extremamente semelhantes independente da escolha, porém, caso fosse a escolha de apenas um, o grupo optaria pela escolha do Shellsort, que apresentou o menor tempo de execução entre os algoritmos testados.

- **Número de elementos → 500:**
    - **Seleção**: a ordenação por seleção neste caso continua eficiente, porém com uma média de comparações/movimentações muito maior.
    - **Inserção**: tal tipo de ordenação é semelhante ao anterior em questões de eficiência e média de comparações.
    - **Shellsort**: também foi extremamente eficiente para o vetor em ordem decrescente; porém, com uma média ainda menor de comparações e movimentações.
    - **Quicksort**: também extremamente eficaz, apesar dos maiores números de comparações e movimentações em relação ao Shellsort.

- **Heapsort/Mergesort:** extremamente eficientes com um ótimo tempo de execução; média de comparações/movimentações maior que os anteriores.

**Opinião do Grupo**: Nesse caso, com 500 elementos, já é possível notar a diferença entre os métodos de ordenação Inserção e Seleção e os outros 4 algoritmos, em razão da sua complexidade quadrática nesse caso, o tempo de execução dos dois cresce muito mais rápido. Após um debate entre o grupo, foi obtido um consenso sobre a utilização do Shellsort que obteve o menor tempo entre os seis algoritmos testados.

- **Número de elementos → 5000:**
  - **Seleção**: a ordenação por seleção neste caso continua relativamente eficiente, mesmo com uma média de comparações/movimentações muito maior.
  - **Inserção**: essa ordenação é semelhante ao anterior em questões de eficiência e média de comparações.
  - **Shellsort**: também foi extremamente eficiente para o vetor em ordem decrescente, com uma média ainda menor de comparações e movimentações. Possui maior eficácia.
  - **Quicksort**: menos eficaz que os anteriores, mas ainda assim muito eficiente, apesar da maior média de comparações e movimentações.
  - **Heapsort**/**Mergesort**: ambos extremamente eficientes; o tempo de execução do Mergesort é menor que do Heapsort.

**Opinião do Grupo**: Após um debate entre o grupo, foi obtido um consenso a respeito da utilização do Shellsort, pois embora ele tenha a mesma complexidade do Mergesort e do Heapsort nesse caso, ainda sim ele apresenta um tempo de execução extremamente menor.

- **Número de elementos → 10000:**
  - **Seleção**: a ordenação por seleção neste caso já é menos eficiente, com uma média de comparações/movimentações muito maior.
  - **Inserção**: essa ordenação é semelhante ao anterior em questões de eficiência e média de comparações.
  - **Shellsort**: também foi extremamente eficiente para o vetor em ordem decrescente; porém, com uma média ainda menor de comparações e movimentações.
  - **Quicksort**: extremamente eficiente; com valores parecidos com o Shellsort.
  - **Heapsort**/**Mergesort**: ambos também extremamente eficientes e com ótimo tempo de execução, sendo o do Mergesort um pouco menor.

**Opinião do Grupo**: Após um debate e uma análise de cada um dos algoritmos realizados pelo grupo sobre a utilização do Shellsort ou do MergeSort, concordamos que ambos apresentam tempos extremamente semelhantes e alternam na diferença entre movimentação e comparação. Um dos pontos comentados pelo grupo durante o debate foi a respeito da ineficiência dos algoritmos de Seleção e Inserção que devido a sua

complexidade quadrática nesse caso, quanto maior a entrada mais ineficiente ele vai se tornando.

- **Número de elementos → 200000:**
    - **Seleção**: a ordenação por seleção neste caso fica extremamente ineficiente, com médias exorbitantes de execução, comparações e movimentações.
    - **Inserção**: essa ordenação é semelhante ao anterior em relação à sua ineficácia, possuindo ainda maior média de tempo de execução.
    - **Shellsort**: extremamente eficiente para o vetor em ordem decrescente mesmo com este tamanho. Possui maior eficácia.
    - **Quicksort**: menos eficaz que a anterior, mas ainda assim muito eficiente, apesar dos maiores números de comparações e movimentações.
    - **Heapsort/Mergesort:** ambos extremamente eficientes; ficam atrás apenas do Shellsort no tempo de execução.

**Opinião do Grupo**: Após um debate e uma análise de cada um dos algoritmos realizados o fato que mais chamou a atenção foi a ineficiência dos algoritmos de Inserção e Seleção que com esse tamanho de entrada já é completamente inutilizável para a ordenação de um vetor em ordem decrescente, com o Inserção demorando mais de um minuto para executar. A escolha do grupo seria pelo algoritmo Shellsort que apresentou um tempo muito pequeno de execução apesar da entrada ser grande.

**Funções Aleatórias - Chave Grande**

| Tempo médio | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 0.000003 | 0.000382 | 0.070945 | 0.302698 | 345.562286 |
| **Inserção** | 0.000001 | 0.000237 | 0.042181 | 0.193548 | 236.500046 |
| **Shellsort** | 0.000002 | 0.000084 | 0.003688 | 0.007733 | 0.245581 |
| **Quicksort** | 0.000002 | 0.000069 | 0.001234 | 0.003035 | 0.084320 |
| **Heapsort** | 0.000003 | 0.000103 | 0.002488 | 0.006886 | 0.166534 |
| **Mergesort** | 0.000003 | 0.000078 | 0.001593 | 0.004548 | 0.075661 |

| Média de comparações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 190 | 124750 | 12497500 | 29995000 | 19999900000 |
| **Inserção** | 110 | 63533 | 6236645 | 24973970 | 9998904931 |
| **Shellsort** | 86 | 6192 | 113212 | 264669 | 10016314 |
| **Quicksort** | 114 | 7220 | 110669 | 244816 | 6652279 |

| | | | | | |
|---|---|---|---|---|---|
| **Heapsort** | 146 | 9340 | 135267 | 295543 | 8075950 |
| **Mergesort** | 259 | 13328 | 183864 | 397690 | 10548409 |

| **Média de movimentações** | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 57 | 1497 | 14997 | 29997 | 599997 |
| **Inserção** | 129 | 64032 | 6241644 | 24983960 | 9999104930 |
| **Shellsort** | 97 | 8847 | 182322 | 449176 | 20750391 |
| **Quicksort** | 88 | 5206 | 76175 | 169360 | 4567619 |
| **Heapsort** | 140 | 8055 | 114199 | 248367 | 6699639 |
| **Mergesort** | 88 | 4488 | 61808 | 133616 | 3537856 |

- **Número de elementos → 20:**
    - **Seleção**: por ser simples e fácil de implementar, a ordenação por seleção neste caso se mostra extremamente eficiente para um vetor de 20 elementos de chave grande gerados aleatoriamente.
    - **Inserção**: tal tipo de ordenação também é eficiente para vetores pequenos. Possui quase o mesmo tempo de execução da ordenação por seleção, porém com menos comparações e mais movimentações.
    - **Shellsort**: também foi extremamente eficiente para o vetor gerado aleatoriamente, com uma média próxima do tempo de execução dos dois anteriores; porém, com ainda menos número de comparações.
    - **Quicksort**: assim como os três primeiros, seu tempo de execução foi 0,000002 segundos, portanto, para tal vetor gerado aleatoriamente foi extremamente eficiente.
    - **Heapsort/Mergesort:** assim como as anteriores, extremamente eficientes devido ao tamanho pequeno do vetor.

**Opinião do Grupo:** Assim como nos outros testes com 20 elementos, os resultados se mostram muito rasos, visto que o tempo gasto por cada um dos algoritmos é muito parecido pois todos são eficientes com uma entrada pequena, então a escolha de cada um depende apenas dos fatores do ambiente que será utilizado.

- **Número de elementos → 500:**
    - **Seleção**: a ordenação por seleção neste caso se mostra extremamente eficiente para um vetor de 500 elementos gerados aleatoriamente com chave grande, porém realiza um número relativamente grande de comparações e movimentações.
    - **Inserção**: tal tipo de ordenação também é eficiente para vetores deste tamanho. Possui basicamente o mesmo tempo de execução da ordenação por seleção, porém com menos comparações e mais movimentações.

- **Shellsort**: foi extremamente eficiente para o vetor gerado aleatoriamente, com o mesmo tempo de execução dos dois anteriores; porém, com ainda menos número de comparações e movimentações.
- **Quicksort**: assim como os três primeiros, seu tempo de execução foi 0,000069 segundos, portanto, para tal vetor gerado aleatoriamente, foi extremamente eficiente. Demonstrou maior eficácia.
- **Heapsort/Mergesort:** também extremamente eficientes; o tempo de execução do Mergesort fica atrás apenas do Quicksort.

**Opinião do Grupo:** A partir desse valor é possível obter resultados mais conclusivos visto que já existe uma diferença de tempo entre os algoritmos, após um debate, o grupo chegou a um consenso sobre a utilização do Quicksort, Mergesort ou Shellsort, que apresentam tempos extremamente parecidos e variam na comparação de movimentações e entradas.

- **Número de elementos → 5000:**
  - **Seleção**: a ordenação por seleção neste caso se mostra extremamente relativamente eficiente; apesar de já ser mais lenta que as outras devido ao tamanho do vetor.
  - **Inserção**: é semelhante à seleção no quesito tempo de execução e média de comparações/movimentações.
  - **Shellsort**: menor tempo de execução que os dois anteriores, ou seja, bastante eficiente.
  - **Quicksort**: comparando com os três primeiros, seu tempo de execução foi o menor de todos, portanto, para tal vetor gerado aleatoriamente, foi extremamente eficiente. Demonstrou maior eficácia.
  - **Heapsort/Mergesort:** também extremamente eficientes; o tempo de execução do Mergesort fica atrás apenas do Quicksort.

**Opinião do Grupo:** Após um debate entre o grupo, foi possível concluir que o Quicksort e o método de ordenação Mergesort obtiveram os melhores resultados e são os mais indicados para a utilização nesse caso.

- **Número de elementos → 10000:**
  - **Seleção**: a ordenação por seleção neste caso se mostra eficiente para um vetor de 10000 elementos gerados aleatoriamente com chave grande, porém realiza um número relativamente grande de comparações e movimentações.
  - **Inserção**: tal tipo de ordenação também é relativamente eficiente para vetores deste tamanho. Valor próximo do tempo de execução da ordenação por seleção, porém com muito mais movimentações.
  - **Shellsort**: ainda mais eficiente para o vetor de chave grande gerado aleatoriamente e possui ainda menos número de comparações e movimentações.
  - **Quicksort**: valor próximo do Shellsort, porém com o tempo de execução ainda menor; número de comparações e movimentações próximo dos outros métodos de ordenação.

- **Heapsort/Mergesort:** extremamente eficientes comparados com os anteriores, perdendo apenas para o Quicksort (minimamente) no tempo de execução e média de comparações e movimentações.

**Opinião do Grupo:** Após uma análise dos algoritmos de ordenação testados, foi possível chegar a conclusão que os mais eficientes nesse caso foram o Quicksort e o Mergesort.

- **Número de elementos → 200000:**
  - **Seleção**: a ordenação por seleção neste caso fica extremamente ineficiente, com médias exorbitantes de execução, comparações e movimentações.
  - **Inserção**: mesmo com um menor tempo de execução, essa ordenação é semelhante ao anterior em relação à sua ineficácia, possuindo ainda maior média de tempo de execução.
  - **Shellsort**: muito eficiente para esta situação em comparação com os dois anteriores. Médias de comparações/movimentações extremamente menores.
  - **Quicksort**: com o tempo de execução ainda menor que o anterior, ou seja, muito eficiente; número de comparações e movimentações próximo dos outros métodos de ordenação.
  - **Heapsort/Mergesort:** extremamente eficientes comparados com os anteriores, com o Mergesort ganhando até do Quicksort (minimamente).

**Opinião do Grupo:** Após um debate entre os integrantes do grupo chegamos a conclusão que os algoritmos mais eficientes nesse caso foram o Quicksort e o Mergesort, os quais apresentaram tempos extremamente semelhantes em razão da sua complexidade O(N x log(N)), apesar de ter a mesma complexidade, o Heapsort teve que realizar mais movimentações fazendo com que sua eficiência diminuísse. Também é importante ressaltar o alto tempo gasto pelo Inserção e Seleção.

**Funções Crescentes - Chave Grande**

| Tempo médio | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 0.000002 | 0.000333 | 0.063741 | 0.260743 | 405.244049 |
| **Inserção** | 0.000001 | 0.000004 | 0.000116 | 0.000373 | 0.009312 |
| **Shellsort** | 0.000015 | 0.000014 | 0.000748 | 0.001489 | 0.088653 |
| **Quicksort** | 0.000003 | 0.000048 | 0.001008 | 0.002515 | 0.075507 |
| **Heapsort** | 0.000003 | 0.000088 | 0.002004 | 0.004157 | 0.096950 |
| **Mergesort** | 0.000003 | 0.000046 | 0.001016 | 0.002381 | 0.068957 |

| Média de comparações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 190 | 124750 | 12497500 | 49995000 | 19999900000 |
| **Inserção** | 19 | 499 | 4999 | 9999 | 199999 |
| **Shellsort** | 62 | 3506 | 55005 | 120005 | 3200006 |
| **Quicksort** | 70 | 4698 | 72495 | 159344 | 4240518 |
| **Heapsort** | 178 | 10317 | 146123 | 317541 | 8503544 |
| **Mergesort** | 243 | 11747 | 160619 | 346239 | 9083519 |

| Média de movimentações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 57 | 1497 | 14997 | 29997 | 599997 |
| **Inserção** | 38 | 998 | 9998 | 19998 | 399998 |
| **Shellsort** | 0 | 0 | 0 | 0 | 0 |
| **Quicksort** | 222 | 9930 | 132860 | 296868 | 7435471 |
| **Heapsort** | 162 | 8710 | 121866 | 263914 | 6998322 |
| **Mergesort** | 88 | 4488 | 61808 | 133616 | 3537856 |

- **Número de elementos → 20:**
    - **Seleção**: por ser simples e fácil de implementar, a ordenação por seleção neste caso é muito eficiente devido ao fato do vetor já estar ordenado crescentemente, sendo esse o seu melhor caso.
    - **Inserção**: tal tipo de ordenação também é extremamente eficiente para vetores de chave grande em ordem crescente pois este também é seu melhor caso, fazendo exatamente (n - 1) comparações (19).
    - **Shellsort**: também foi extremamente eficiente para o vetor de ordem crescente; porém, com uma média ainda menor de comparações e movimentações (sendo esta exatamente 0).
    - **Quicksort**: assim como os três primeiros, seu tempo de execução foi extremamente eficiente.
    - **Heapsort e Mergesort:** assim como todos os anteriores, possuem tempo médio de execução parecidos e muito eficientes.

**Opinião do Grupo:** Nesse caso em específico, em razão do vetor ser muito pequeno, os resultados são muito rasos, fazendo com que todos tenham resultados parecidos e que a utilização de cada um deles seja praticamente a mesma, dependendo apenas dos fatores do ambiente que será utilizado.

- **Número de elementos → 500:**
    - **Seleção**: a ordenação por seleção neste caso se mostra eficiente para um vetor de 500 elementos ordenados, porém, seus números de comparações e movimentações são relativamente grandes.
    - **Inserção**: muito mais eficiente do que por seleção por ser o seu melhor caso, com um número de comparações/movimentações extremamente menor, sendo o de comparações igual a 499.
    - **Shellsort**: também extremamente eficiente, avaliando a média tempo de execução, números de comparações e de movimentações (sendo o nº de movimentações exatamente igual a 0).
    - **Quicksort**: extremamente eficiente assim como o anterior, porém com um número de movimentações relativamente grande.
    - **Heapsort e Mergesort:** médias de execução de tempo eficientes; porém com maiores números de movimentações e comparações.

**Opinião do Grupo:** Por se tratar de um vetor já ordenado o método de ordenação por Inserção acaba sendo o mais eficiente em razão da sua complexidade linear, ou O(N)., e por causa disso ele seria o escolhido pelo grupo.

- **Número de elementos → 5000:**
    - **Seleção**: se mostra relativamente eficiente para um vetor de 5000 elementos ordenados, porém, seus números de comparações e movimentações são muito grandes quando comparados com outros menores.
    - **Inserção**: neste caso, se mostrou o método de ordenação mais eficiente, com um número de comparações/movimentações extremamente menor, sendo o de comparações igual a 4999. Possui maior eficácia.
    - **Shellsort**: semelhante ao de inserção, avaliando a média de tempo de execução, números de comparações e de movimentações (sendo o nº de movimentações exatamente igual a 0).
    - **Quicksort**: extremamente eficiente com ótimo tempo de execução; porém, números de comparações e movimentações bem grandes.
    - **Heapsort**: média de tempo de execução um pouco maior mas ainda eficiente, com bons números no geral.
    - **Mergesort**: extremamente eficiente apesar da média de comparações e movimentações serem relativamente grandes.

**Opinião do Grupo:** Por se tratar de um vetor já ordenado o método de ordenação por Inserção acaba sendo o mais eficiente em razão da sua complexidade linear, ou O(N), por causa disso ele seria o escolhido pelo grupo.

- **Número de elementos → 10000:**
    - **Seleção**: seus números de comparações e movimentações são muito grandes quando comparados com os outros, portanto não é a opção mais eficiente neste caso devido ao tamanho do vetor.
    - **Inserção**: se mostrou o método de ordenação mais eficiente, com um número de comparações/movimentações extremamente menor, sendo o de comparações igual a 9999.
    - **Shellsort**: eficiência similar ao de inserção, com ótimos números de comparações e de movimentações (sendo o de movimentações exatamente igual a 0).
    - **Quicksort**: extremamente eficiente apesar de sua média de movimentações ser relativamente grande comparado ao resto.
    - **Heapsort e Mergesort:** ambas extremamente eficientes, mesmo com a média de números de comparação/movimentação relativamente grandes.

**Opinião do Grupo:** Por se tratar de um vetor já ordenado o método de ordenação por Inserção acaba sendo o mais eficiente em razão da sua complexidade linear, ou O(N), por causa disso ele seria o escolhido pelo grupo.

- **Número de elementos → 200000:**
    - **Seleção**: extremamente ineficiente. Seus números de comparações e movimentações são muito grandes quando comparados com os outros, portanto é a opção menos eficiente neste caso devido ao tamanho do vetor.
    - **Inserção**: se mostrou o método de ordenação mais eficiente em relação ao tempo de execução, com um número de comparações muito pequeno (199999) em relação aos outros.
    - **Shellsort**: menos eficaz do que por inserção, mas ainda assim eficiente após a observação dos números de comparações e de movimentações (sendo o nº de movimentações exatamente igual a 0).
    - **Quicksort**: extremamente eficiente apesar de sua média de movimentações ser relativamente grande comparado ao resto.
    - **Heapsort e Mergesort:** ambas extremamente eficientes, mesmo com a média de números de comparação/movimentação relativamente grandes.

**Opinião do Grupo:** Por se tratar de um vetor já ordenado o método de ordenação por Inserção acaba sendo o mais eficiente em razão da sua complexidade linear, ou O(N), por causa disso ele seria o escolhido pelo grupo. Também é importante ressaltar a eficiência do Shellsort que também poderia ser utilizado nesse caso.

## Funções Decrescentes - Chave Grande

| Tempo médio | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 0.000002 | 0.001120 | 0.067630 | 0.324722 | 394.834778 |
| **Inserção** | 0.000002 | 0.000422 | 0.087948 | 0.426012 | 537.938416 |
| **Shellsort** | 0.000001 | 0.000030 | 0.001113 | 0.002946 | 0.141082 |
| **Quicksort** | 0.000003 | 0.000052 | 0.000926 | 0.002470 | 0.077226 |
| **Heapsort** | 0.00002 | 0.000079 | 0.001781 | 0.004972 | 0.101536 |
| **Mergesort** | 0.000002 | 0.000046 | 0.001051 | 0.003815 | 0.066125 |

| Média de comparações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 190 | 124750 | 12497500 | 49995000 | 19999900000 |
| **Inserção** | 209 | 125249 | 12502499 | 50004999 | 20000099999 |
| **Shellsort** | 80 | 5116 | 78798 | 172578 | 4789145 |
| **Quicksort** | 73 | 5026 | 69134 | 158289 | 4247244 |
| **Heapsort** | 130 | 8411 | 125297 | 275316 | 7647635 |
| **Mergesort** | 235 | 11691 | 158419 | 341839 | 9005759 |

| Média de movimentações | | | | | |
|---|---|---|---|---|---|
| **Funções** | **20** | **500** | **5000** | **10000** | **200000** |
| **Seleção** | 57 | 1497 | 14997 | 29997 | 599997 |
| **Inserção** | 228 | 125748 | 12507498 | 500014998 | 20000299998 |
| **Shellsort** | 108 | 6300 | 86340 | 187680 | 5367360 |
| **Quicksort** | 212 | 9673 | 129332 | 292891 | 7441240 |
| **Heapsort** | 1126 | 7354 | 106874 | 233394 | 6391458 |
| **Mergesort** | 88 | 4488 | 61808 | 133616 | 3537856 |

- **Número de elementos → 20:**
    - **Seleção**: por ser um vetor pequeno, simples e fácil de implementar, a ordenação por seleção neste caso é muito eficiente.
    - **Inserção**: tal tipo de ordenação também é extremamente eficiente para vetores pequenos em ordem decrescente, mesmo fazendo 209 comparações e não apenas 19 igual na ordem crescente.
    - **Shellsort**: também foi extremamente eficiente para o vetor deste tamanho de ordem decrescente; porém, com uma média ainda menor de comparações e movimentações. Possui maior eficácia.
    - **Quicksort**: também é considerada eficiente, possuindo basicamente o mesmo tempo de execução das anteriores e números médios de comparação e movimentação.
    - **Heapsort/Mergesort:** assim como os anteriores, extremamente eficientes com um ótimo tempo de execução e média de comparações/movimentações.

**Opinião do Grupo**: Assim como no vetor aleatório e no vetor crescente, a diferença de tempo de ordenação com 20 elementos é praticamente nula, fazendo com que os resultados sejam extremamente semelhantes independente da escolha, porém, caso fosse a escolha de apenas um, o grupo optaria pela escolha do Shellsort, que apresentou o menor tempo de execução entre os algoritmos testados.

- **Número de elementos → 500:**
    - **Seleção**: a ordenação por seleção neste caso continua eficiente, porém com uma média de comparações/movimentações muito maior.
    - **Inserção**: tal tipo de ordenação é semelhante ao anterior em questões de eficiência e média de comparações.
    - **Shellsort**: também foi extremamente eficiente para o vetor em ordem decrescente; porém, com uma média ainda menor de comparações e movimentações.
    - **Quicksort**: também extremamente eficaz, apesar dos maiores números de comparações e movimentações em relação ao Shellsort.
    - **Heapsort/Mergesort:** extremamente eficientes com um ótimo tempo de execução; média de comparações/movimentações maior que os anteriores.

**Opinião do Grupo**: Nesse caso, com 500 elementos, já é possível notar a diferença entre os métodos de ordenação Inserção e Seleção e os outros 4 algoritmos, em razão da sua complexidade quadrática nesse caso, o tempo de execução dos dois cresce muito mais rápido. Após um debate entre o grupo, foi obtido um consenso sobre a utilização do Shellsort que obteve o menor tempo entre os seis algoritmos testados.

- **Número de elementos → 5000:**
    - **Seleção**: a ordenação por seleção neste caso continua relativamente eficiente, mesmo com uma média de comparações/movimentações muito maior.
    - **Inserção**: essa ordenação é semelhante ao anterior em questões de eficiência e média de comparações.

- **Shellsort**: também foi extremamente eficiente para o vetor em ordem decrescente, com uma média ainda menor de comparações e movimentações que os anteriores
- **Quicksort**: ainda mais eficaz que os anteriores, ou seja, extremamente eficiente neste caso.
- **Heapsort**/**Mergesort**: ambos extremamente eficientes; o tempo de execução do Mergesort é um pouco menor que do Heapsort.

**Opinião do Grupo**: Após um debate entre o grupo, foi obtido um consenso a respeito da utilização do Quicksort, pois embora ele tenha a mesma complexidade do Mergesort e do Heapsort nesse caso, ainda sim ele apresenta um tempo de execução extremamente menor.

- **Número de elementos → 10000:**
  - **Seleção**: a ordenação por seleção neste caso já é menos eficiente, com uma média de comparações/movimentações muito maior.
  - **Inserção**: essa ordenação é semelhante ao anterior em questões de eficiência e média de comparações.
  - **Shellsort**: também foi extremamente eficiente para o vetor em ordem decrescente; porém, com uma média ainda menor de comparações e movimentações.
  - **Quicksort**: extremamente eficiente; com valores parecidos com o Shellsort.
  - **Heapsort**/**Mergesort**: ambos também extremamente eficientes e com ótimo tempo de execução, sendo o do Mergesort um pouco menor.

**Opinião do Grupo**: Após um debate e uma análise de cada um dos algoritmos realizados pelo grupo sobre a utilização do Quicksort, Shellsort ou MergeSort, concordamos que Quicksort apresenta tempo de execução menor e por isso seria o escolhido. Um dos pontos comentados pelo grupo durante o debate foi a respeito da ineficiência dos algoritmos de Seleção e Inserção que devido a sua complexidade quadrática nesse caso, quanto maior a entrada mais ineficiente ele vai se tornando.

- **Número de elementos → 200000:**
  - **Seleção**: a ordenação por seleção neste caso fica extremamente ineficiente, com médias exorbitantes de execução, comparações e movimentações.
  - **Inserção**: essa ordenação é semelhante ao anterior em relação à sua ineficácia, possuindo ainda maior média de tempo de execução.
  - **Shellsort**: extremamente eficiente para o vetor em ordem decrescente, mesmo com este tamanho.
  - **Quicksort**: um pouco ainda mais eficiente que o Shellsort.
  - **Heapsort/Mergesort:** ambos extremamente eficientes; Mergesort possui o menor tempo de execução de todos os outros métodos.

**Opinião do Grupo:** Nesse caso em específico os algoritmos mais eficientes foram o Quicksort e o Mergesort apresentando resultados extremamente parecidos, porém nesse caso o grupo optaria pelo Quicksort.

# CONCLUSÃO

Após uma análise crítica da implementação de todos os métodos de ordenação em diversos cenários distintos, com vetores de tamanhos e níveis de complexidade diferentes, torna-se claro a extrema importância do estudo da ordenação na disciplina de AEDS 2 para os estudantes de Ciência da Computação, com o objetivo de desenvolver algoritmos que lidam com diferentes números de elementos da maneira mais eficaz possível. Como tal eficácia depende de diversos fatores da natureza do algoritmo, é de extrema importância o conhecimento teórico e básico dos principais métodos de ordenação, além de toda a matéria que vimos até agora nas disciplinas de algoritmos e estruturas de dados, para que os resultados obtidos sejam sempre os mais otimizados possíveis. Portanto, após a realização de diversos testes de algoritmos, implementações e códigos, chegamos aos resultados para cada situação e os analisamos de maneira crítica, a fim de cumprir os objetivos do trabalho prático.

# REFERÊNCIAS


- **Sort Visualizer** - https://www.sortvisualizer.com. Acesso em 10 de maio de 2023.
- **Slides e material didático** disponibilizado pelo professor Rafael Sachetto Oliveira. Acesso em 15 de maio de 2023.