Algoritmos de ordenação - Analise

Essa documentação dará uma breve análise dos algoritmos de ordenação estudados na disciplina de Algoritmos e estruturas de Dados II, na universidade Estadual do mato grosso do Sul. O trabalho consiste em uma breve explicação dos algoritmos, seguindo de uma análise do tempo de coletado no laboratório de informática.

Introdução

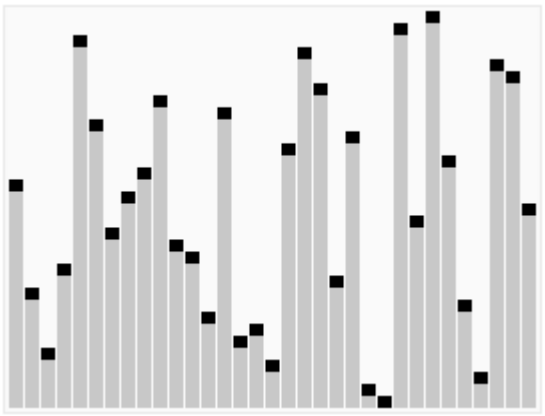
Sabemos que um bom algoritmo de ordenação tem uma complexidade entre `n` e `n*log(n)`, já um algoritmo ingênuo, péssimo, se faz em n^2. Documentaresmos aqui uma sucinta analise desses diversos casos, será apresentado a complexidade do algoritmo, o tempo de execução implementado na linguagem `C`.

BubleSort

Funcionamento:

A ideia é percorrer o vetor diversas vezes, e a cada passagem fazer flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

Um gif com o funciomaneto do algoritmo:



Estabilidade:

O BubleSort é um algoritmo de **ordenação estável**.

Complexidade:

$$Pior\ \ caso : O(n^2)$$
$$Caso\ médio : O(n^2)$$
$$Melhor\ caso : O(n^2)$$

Logo a baixo, temos uma tabela de valores com tempos execução em `milisegundos` para entradas de 10 elementos a 1 milhão. Os tempos de execução do algoritmo bubleSort foram coletados em um computador onde os unicos processos eram o do algoritmo e do sistema operacional para que a analise fosse a mais precisa.

Informações Técnicas da maquina onde o algoritmo foi radado:

```
function setup() {
    const mySetup = {
        processor: {
          product: "Intel(R) Core(TM) i3-7100 CPU @ 3.90GHZ"
          size: "987MHz"
          capacity: "3900MHz"
          width: 64
        }
        memory: 8,
        HD: "Sata - 500GB",
        SO: "Ubuntu - 64Bits"
    }
}
```

Primeira Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,004	0,003	0,002
100	0,182	0,184	0,093
1.000	14,586	18,876	9,237
10.000	394,649	392,764	219,612
100.000	40.914 - ***Segundos***	36.327 - ***Segundos***	18.950 - ***Segundos***
500.000	17.080 - ***Minutos***	15.139 - ***Minutos***	7.903 - ***Minutos***

Segunda Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,003	0,004	0,001
100	0,182	0,182	0,117
1.000	14,906	17,720	5,076
10.000	397,340	366,845	222,245
100.000	40.90 - ***Segundos***	36.318 - ***Segundos***	18.949 - ***Segundos***
500.000	17.078 - ***Minutos***	15.149 - ***Minutos***	7.909 - ***Minutos***

Terceira Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,003	0,003	0,001
100	0,181	0,183	0,093
1.000	15,001	17,887	9,421
10.000	393,298	364,374	190,147
100.000	40.910 - ***Segundos***	36.337 - ***Segundos***	18,949 - ***Segundos***
500.000	17.065 - ***Minutos***	15.141 - ***Minutos***	7.925 - ***Minutos***

Você também pode vizualizar um gráfico de comparação entre todos os algoritmos de ordenação analisados aqui, na seção - Gráficos.

Agora, vamos analisar o tempo de execução do algoritmo bubleSort com entradas muito maiores, vamos tratar aqui nessa sucitan analise valores na casa dos 1, 10, 100 Milhões de elementos.

Primeiro vamos fazer uma conta simples com base nos dados que já posuimos para aproximarmos o tempo de execução real do algoritmo sendo implementado em alguma linguagem de programação.

Temos que a complexidade do algoritmo em todos os casos é:

$$Complexidadade : O(n^2)$$

Agora, calcularemos quantas operações *(troca)* o mesmo fará com uma entrada de 1.000.000 de elementos no pior caso:

$$\begin{aligned} 500.000^2 &= 250000000000 \\ 1.000.000^2 &= 1e+12 \end{aligned}$$

$$\begin{aligned} 250000000000 &----> 1693.881 \\ 1e+12 &----> x \end{aligned}$$

$$x \approx 6775.524$$

Após feita uma simulação real em um computador com valores randomicos o tempo marcado foi de: `6665.009` ou 1.85 Horas, muito próximo do esperado, agora veremos uma média para os valores de `10.000.000` e `100.000.000`

Os procedimentos seguidos para calcular o tempo de execução para as entradas seguinte é o memso usado para a entrada de `1.000.000`

Entrada de 10.000.000 para o BubleSort é de:

$$n = 10.000.000$$
$$Tempo = 188.209 Horas$$

Ou Aproximadamente **8** Dias

Entrada de 100.000.000 para o BubleSort é de:

$$n = 100.000.000$$
$$Tempo = 313 Horas$$

Ou Aproximadamente **13** Dias

Considerações finais: O bubleSort não é uma das melhores opções para se escolher, independente do caso a complexidade sempre se manterá `O(n²)`. E Como visto a maneira que a quantidade de elementos aumenta, aumenta junto o tempo e processamento do computador.

BubleSort - Melhorado

Esse algoritmo é o mesmo que o anterior mas com algumas melhorias, ele é menos ingenuo e um pouco mais inteligente, ele consegue perceber quando não há trocas no vetor e assim logo encerrar a sua execução, visto que o mesmo já está ordenado, ele também não percorre o mesmo tamanho de vetor, a cada execução ele altera o `n` variavel em função da ultima troca realizada, aqui o algoritmo para melhor entendimento:

Estabilidade:

O BubleSort Melhorado é um algoritmo de **ordenação estável**.

```
void enhance_bubbleSort(int32_t *array, int32_t n) {
    int32_t boolean = 1,
        nFlag = n,
        guard = n,
        j;

    while(boolean) {
```

```
        j = 0;
        boolean = 0;

        while(j < nFlag - 1) {
            if(array[j] > array[j + 1]) {
                Swap(array[j], array[j + 1]);
                boolean = 1;
                guard = j;
            }
            j++;
        }
        nFlag = guard + 1;
    }
}
```

Você também pode encontra-lo implementado na parte de código.

Primeira Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,003	0,002	0,000
100	0,137	0,160	0,001
1.000	11,463	14,362	0,013
10.000	307,479	286,898	0,093
100.000	31.672 - ***Segundos***	28.697 - ***Segundos***	0.206 - ***Milisegundos***
500.000	13.292 - ***Minutos***	11.957 - ***Minutos***	4.037 - ***Milisegundos***

Segunda Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,003	0,002	0,000
100	0,134	0,143	0,001
1.000	10,776	14,823	0,012
10.000	305,760	291,615	0,093
100.000	31,711 - ***Segundos***	28,665 - ***Segundos***	0,964 - ***Milisegundos***
500.000	13.280 - ***Minutos***	11.955 - ***Minutos***	4.703 - ***Milisegundos***

Terceira Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,003	0,002	0,000
100	0,137	0,143	0,001
1.000	10,328	16,571	0,010
10.000	307,196	288,789	0,062
100.000	31,671 - ***Segundos***	28,679 - ***Segundos***	1,133 - ***Milisegundos***
500.000	13.271 - ***Minutos***	11.959 - ***Minutos***	4.846 - ***Milisegundos***

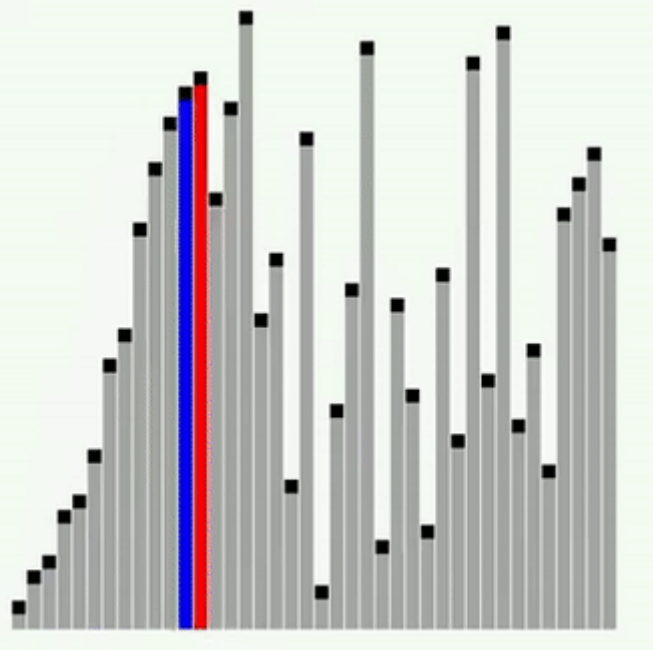
Você também pode vizualizar um gráfico de comparação entre todos os algoritmos de ordenação analisados aqui, na seção - Gráficos.

Aproximações de tempo para entradas de `1.000.000`, `10.000.000` e `100.000.000`

1.000.0000	Horas	Dias
10.000.000	141	7
100.000.000	258	15

InsertionSort

Insertion Sort, ou *ordenação por inserção*, é o algoritmo de ordenação que, dado uma estrutura (array, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.



Estabilidade:

O InsertionSort é um algoritmo de **ordenação estável**.

Complexidade:

$$Pior\ \ caso : O(n^2)$$
$$Caso\ médio : O(n^2)$$
$$Melhor\ caso : O(n)$$

Primeira Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,001	0,001	0,000
100	0,037	0,059	0,002
1.000	2,820	5,703	0,015
10.000	94,605	143,877	0,031
100.000	5.613 - ***Segundos***	11.169 - ***Segundos***	1.515 - ***Milisegundos***
500.000	2.326 - ***Minutos***	4.648 - ***Minutos***	7.498 - ***Milisegundos***

Segunda Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,000	0,001	0,000
100	0,016	0,059	0,003
1.000	2,807	5,508	0,016
10.000	86,267	116,950	0,036
100.000	5,626 - ***Segundos***	11,161 - ***Segundos***	0.505 - ***Milisegundos***
500.000	2.328 - ***Minutos***	4.671 - ***Minutos***	7.771 - ***Milisegundos***

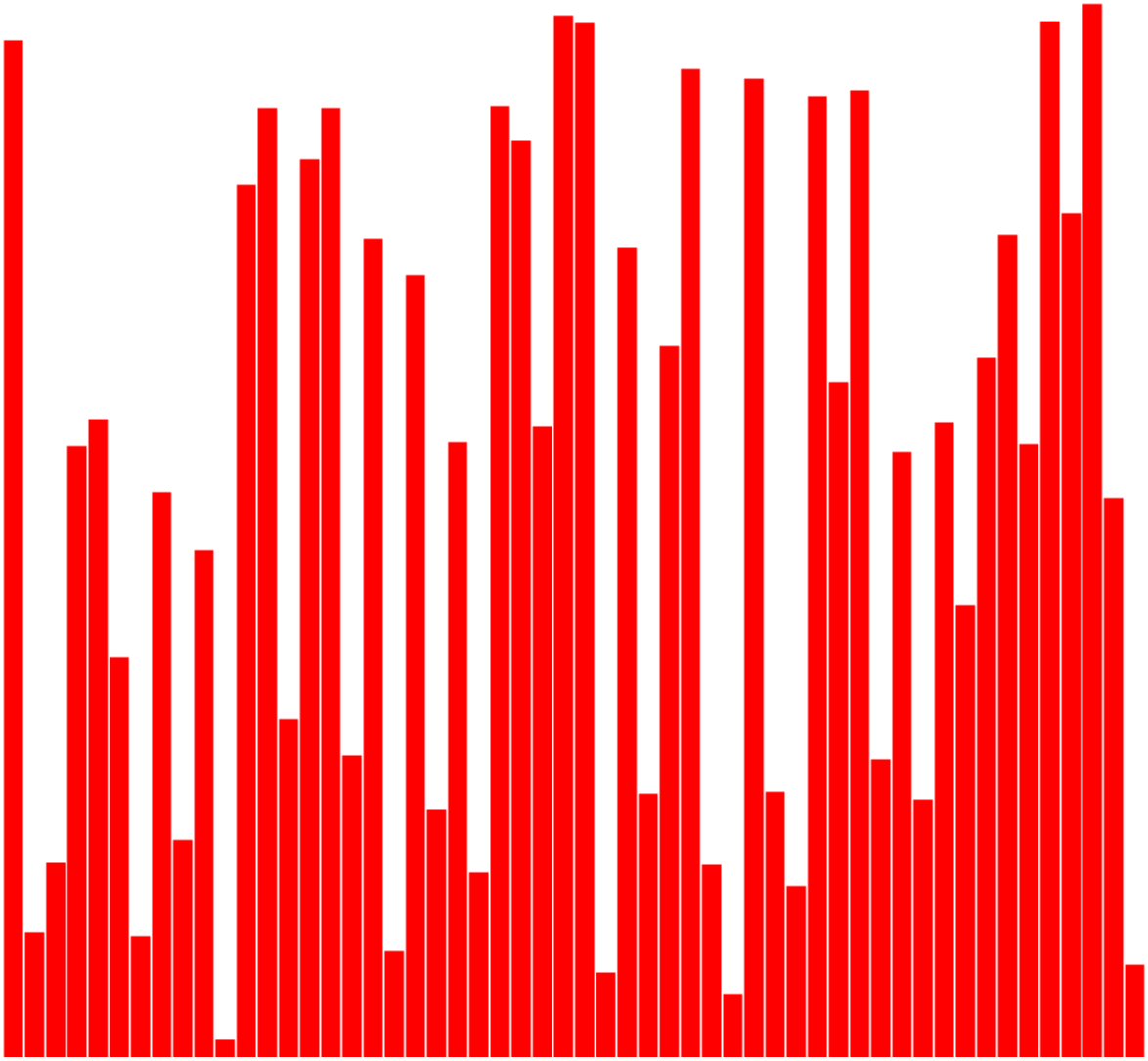
Terceira Execução - Os tempos da tabela se encontram em Milissegundos:

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0,001	0,001	0,001
100	0,037	0,059	0,002
1.000	2,794	6,818	0,016
10.000	93,317	137,413	0,148
100.000	5,621 - ***Segundos***	11,162 - ***Segundos***	0,394 - ***Milisegundos***
500.000	2.326 - ***Minutos***	4.649 - ***Minutos***	7,895 - ***Milisegundos***

Você também pode vizualizar um gráfico de comparação entre todos os algoritmos de ordenação analisados aqui, na seção - Gráficos.

MergeSort

No MergeSort sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas).



Estabilidade:

O mergeSort é um algoritmo de **ordenação estável**, e é possível adaptar a estrutura a ser ordenada de forma a tornar a ordenação não estável.

Complexidade:

$$Piorcaso\ \ caso : O(n * logn)$$
$$Caso\ médio : O(n * logn)$$
$$Melhor\ caso : O(n * logn)$$

Tabela de valores da **média mergeSort** - milisegundos:

NOTA: TODOS OS TEMPOS DE EXECUÇÃO, ESTÃO NO ARQUIVO DENTRO DA PASTA DO TRABALHO....

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0.004	0.001	0.001
100	0.014	0.008	0.005
1.000	0.326	0.062	0.075
10.000	1.783	1.101	1.389
100.000	47.342	36.944	35.237
500.000	142.978	98.390	87.473
1.000.000	174.377	99.008	108.729
10.000.000	2,018 - ***Segundos***	962.411	967.106
100.000.000	21,862 - ***Segundos***	10,531- ***Segundos***	10,757 - ***Segundos***

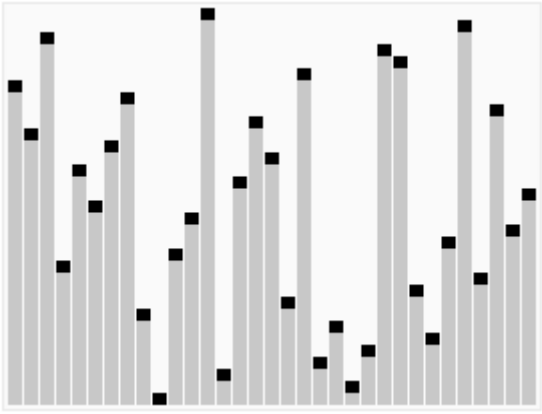
HeapSort

O heapsort utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap.

Um heap pode ser representada como uma árvore, observe, somente **represetando** para melhor vizualização.

6 5 3 1 8 7 2 4

Abaixo um gif que demonstra o algoritmo em execução:



Estabilidade:

O heapsort não é um algoritmo de ordenação estável. Porém, é possível adaptar a estrutura a ser ordenada de forma a tornar a ordenação estável.

Complexidade:

$$Piorcaso\ \ caso:\ O(n^2)$$
$$Caso\ médio: O(n * logn)$$
$$Melhor\ caso: O(n * logn)$$

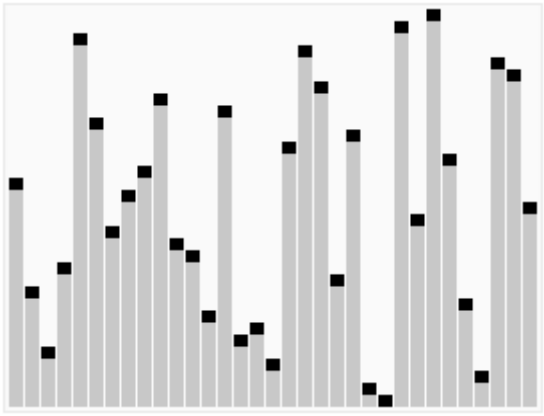
Tabela de valores da **média HeapSort** - milisegundos:

NOTA: TODOS OS TEMPOS DE EXECUÇÃO, ESTÃO NO ARQUIVO DENTRO DA PASTA DO TRABALHO....

Valores	Aleatório(Tempo)	Decrescente(Tempo)	Crescente (Tempo)
10	0.002	0.001	0.002
100	0.023	0.024	0.037
1.000	0.192	0.386	0.274
10.000	3.324	6.904	4.791
100.000	65.074	22.842	24.090
500.000	165.324	98.390	87.473
1.000.000	295.120	194.478	190.590
10.000.000	4,382 - ***Segundos***	2,054 - ***Segundos***	2,072 - ***Segundos***
100.000.000	62,490 - ***Segundos***	23,161- ***Segundos***	23,095 - ***Segundos***

QuickSort

Como o mergeSort , o quicksort dividi e conquista, e é um algoritmo recursivo. A maneira que o quicksort usa a divizão e conquista é um pouco diferente de como o mergeSort o faz. Na mesclagem, a etapa de divisão não faz praticamente nada e todo o trabalho real acontece na etapa de combinação. O Quicksort é o oposto: todo o trabalho real acontece na etapa de divisão.



O algoritm quicksort é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 19601, quando visitou a Universidade de Moscovo como estudante. Naquela época, Hoare trabalhou em um projeto de tradução de máquina para o National Physical Laboratory. Ele criou o quicksort ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido. Foi publicado em 1962 após uma série de refinamentos.

Estabilidade:

O quicksort é um algoritmo de ordenação por comparação não-estável

Complexidade:

$$Piorcaso\ \ caso:\ O(n^2)$$
$$Caso\ médio: O(n * logn)$$
$$Melhor\ caso: O(n * logn)$$

Os gráficos do quickSort estão melhores desenvolvidos na parte de gráficos..

Os valores de várias execuções do quickSort também se encontram nos arquivos, lá se encontram 3 execuções de cada uma das variações do quickSort

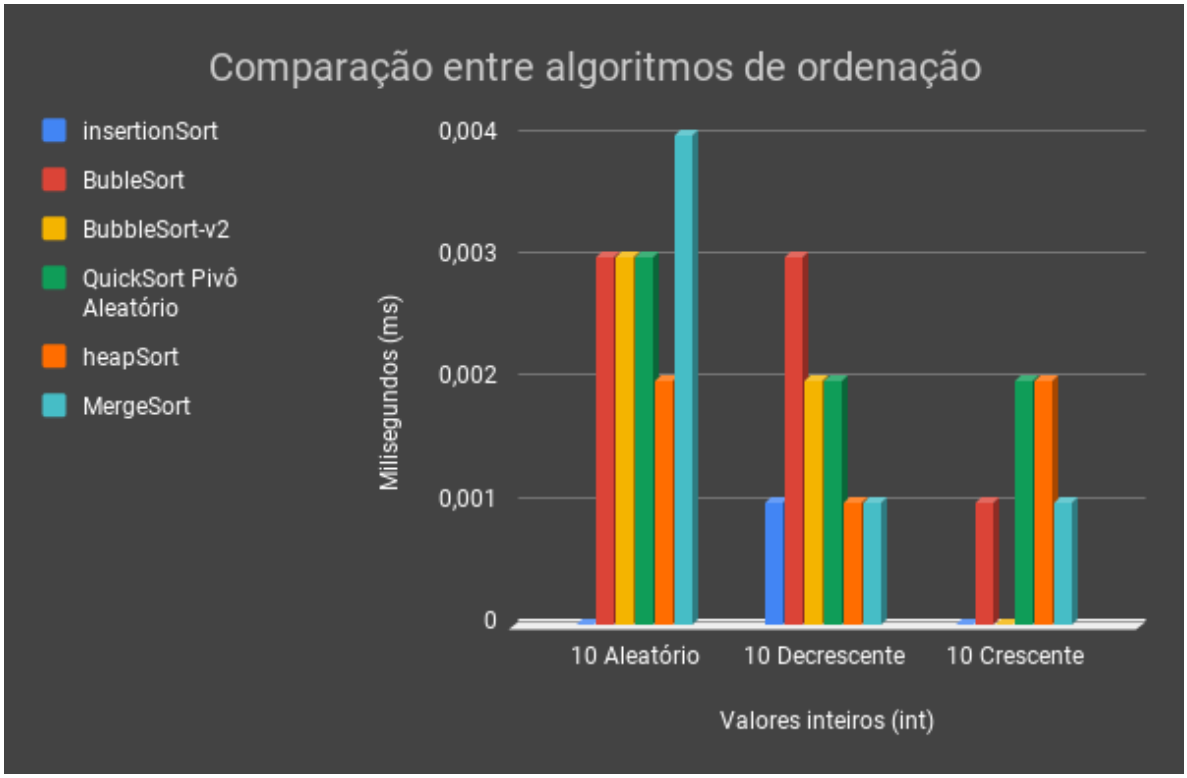
Considerações: Por mais que o o QuickSort possua o pior caso em n^2 ele é o mais utilizado na prática, isso se da pelo fato de que os casos médios das ordenações ficam em $O(n*logn)$, O quickSort também é um dos algoritmos mais conhecidos no mundo, e ele é customizado para subrir a necessidade do caso.

Gráficos

Nesta seção, você poderá encontrar vários gráficos contendo comparações entre os algoritmos analisados aqui nessa documentação. Atente-se para a medição do tempo, alguns gráficos mostram medições de valores em *milisegundos*, e outros por sua vez, apresentam os tempos em *segundos*.

Os dados foram obtidos no laboratório de computação 4, da UEMS, foram coletadas 3 execuções de cada algoritmo e feito uma media somando os tempos de execução e divido por 3. Os gráficos abaixo apresentam algumas médias dos dados coletados, todos os dados obtidos podem ser visto no arquivo no diretório desse trabalho, também existe um arquivo chamado média dos valores onde contem as médias de cada algoritmo.

10 ELEMENTOS

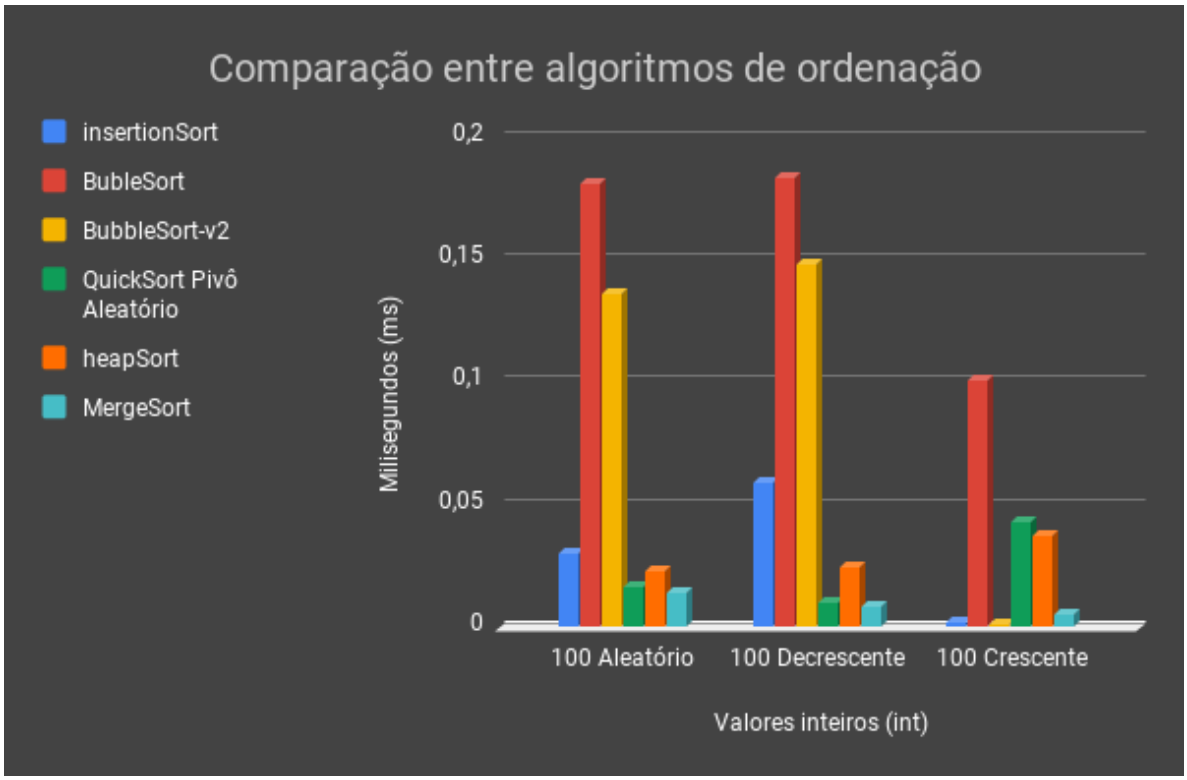


10 Aleatório
- insertionSort: **0**
- BubleSort: **0,003**
- BubbleSort-v2: **0,003**
- QuickSort Pivô Aleatório: **0,003**
- heapSort: **0,002**
- MergeSort: **0,004**

10 Crescente
- insertionSort: **0**
- BubleSort: **0,001**
- BubbleSort-v2: **0**
- QuickSort Pivô Aleatório: **0,002**
- heapSort: **0,002**
- MergeSort: **0,001**

10 Decrescente
- insertionSort: **0,001**
- BubleSort: **0,003**
- BubbleSort-v2: **0,002**
- QuickSort Pivô Aleatório: **0,002**
- heapSort: **0,001**
- MergeSort: **0,001**

100 ELEMENTOS

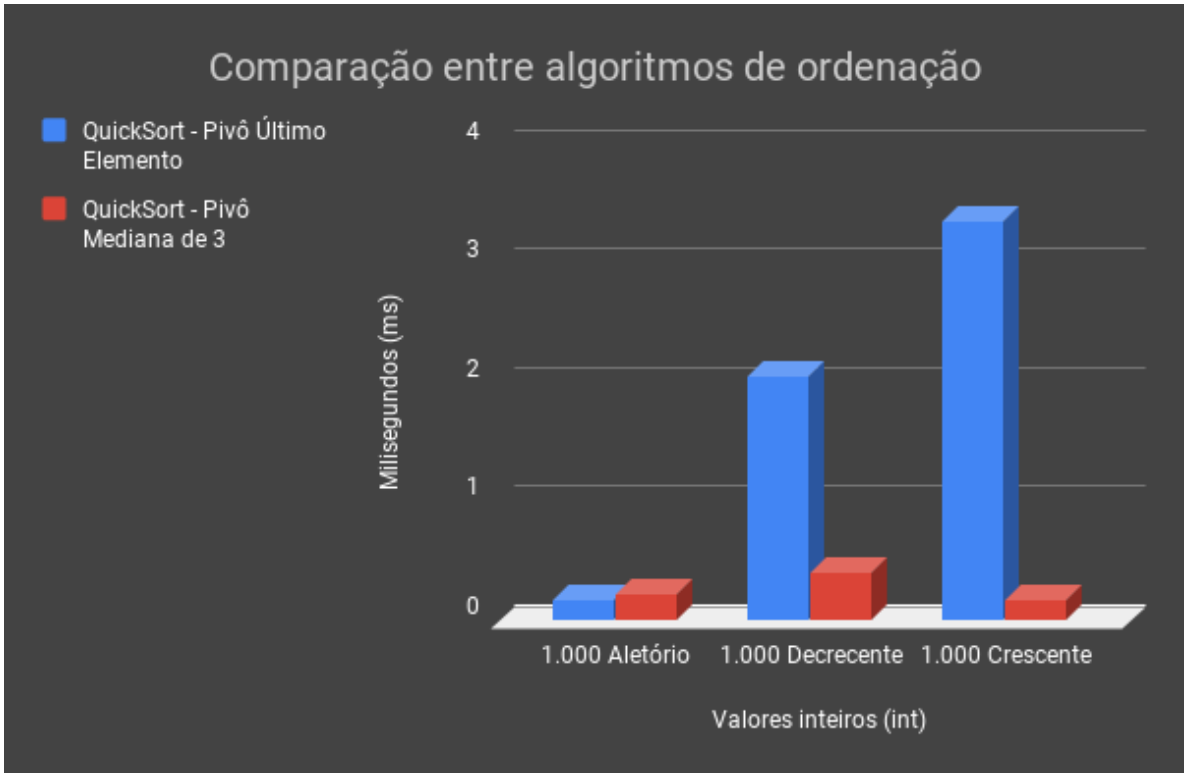


100 Aleatório
- insertionSort: **0,03**
- BubleSort: **0,181**
- BubbleSort-v2: **0,136**
- QuickSort Pivô Aleatório: **0,016**
- heapSort: **0,023**
- MergeSort: **0,014**

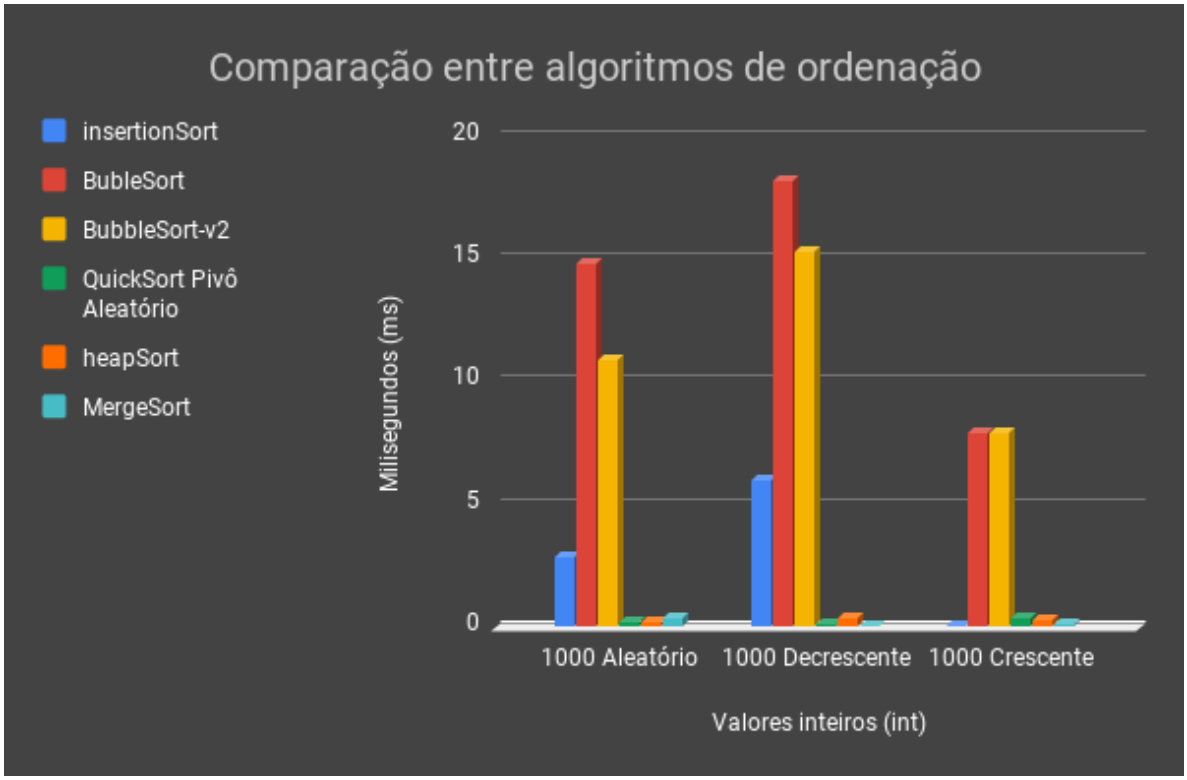
100 Crescente
- insertionSort: **0,002**
- BubleSort: **0,101**
- BubbleSort-v2: **0,001**
- QuickSort Pivô Aleatório: **0,043**
- heapSort: **0,037**
- MergeSort: **0,005**

100 Decrescente
- insertionSort: **0,059**
- BubleSort: **0,183**
- BubbleSort-v2: **0,148**
- QuickSort Pivô Aleatório: **0,01**
- heapSort: **0,024**
- MergeSort: **0,008**

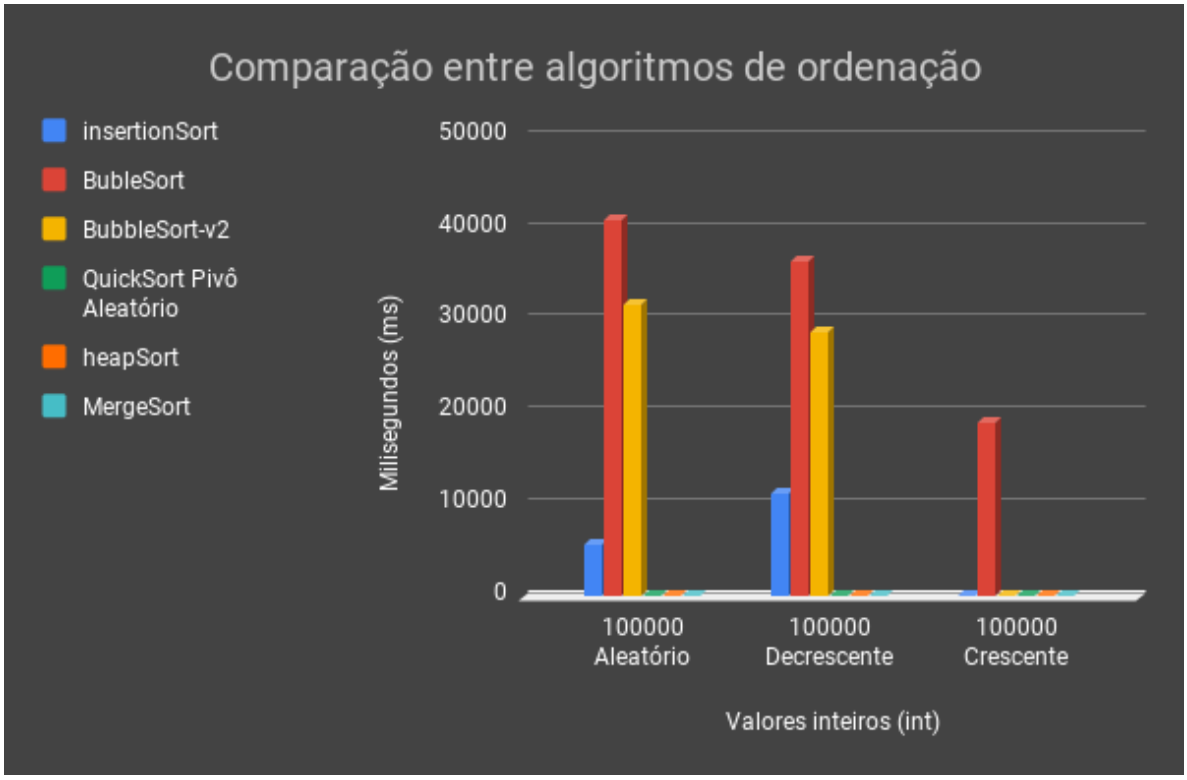
1000 ELEMENTOS



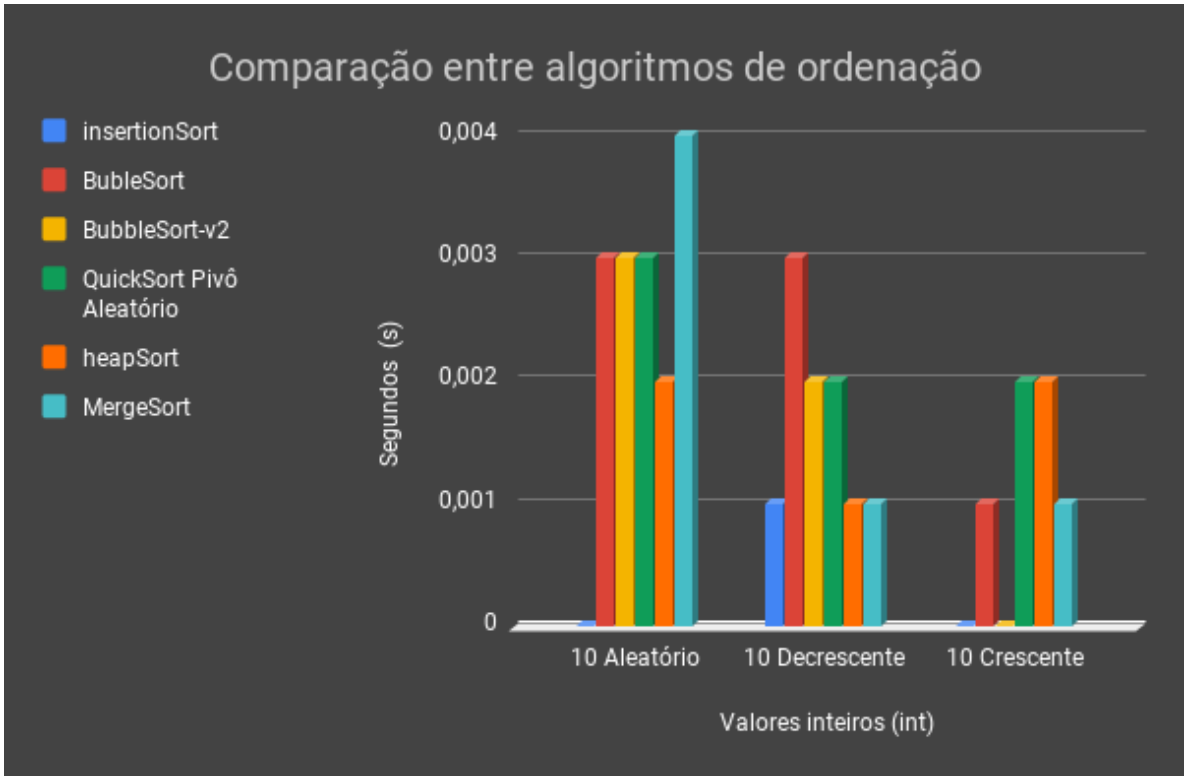
10.000 ELEMENTOS



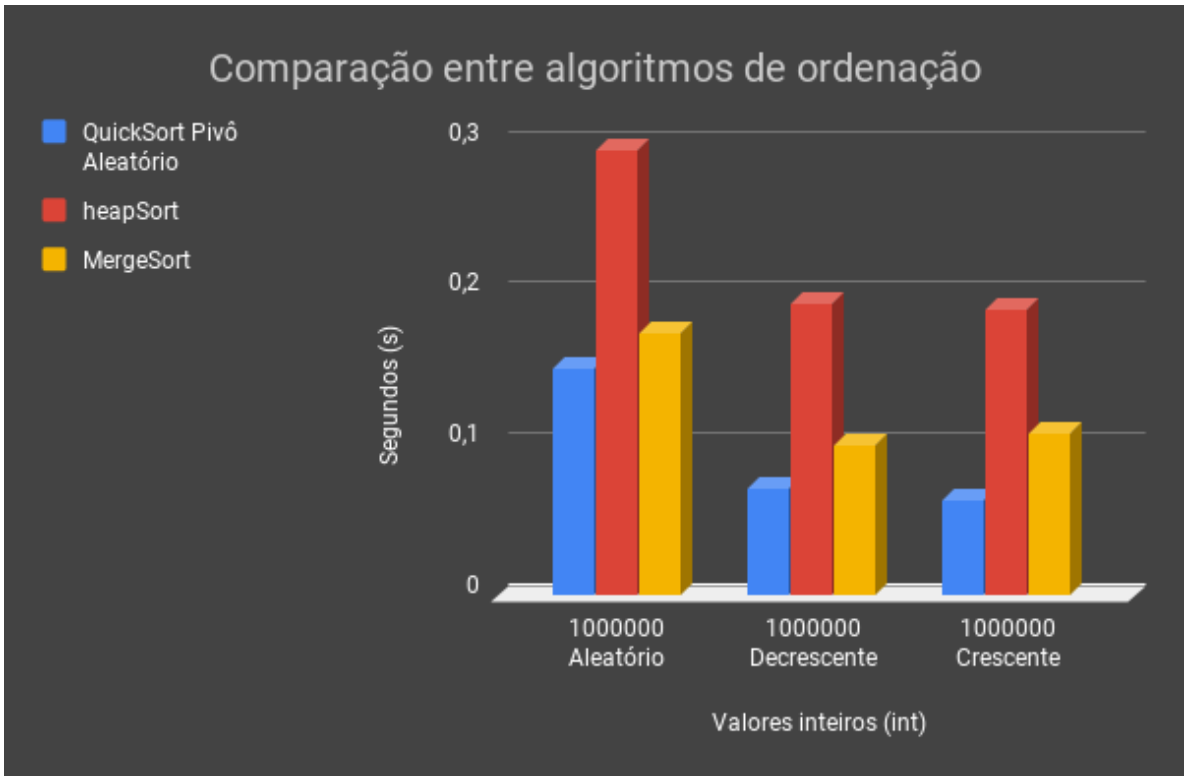
100.000 ELEMENTOS



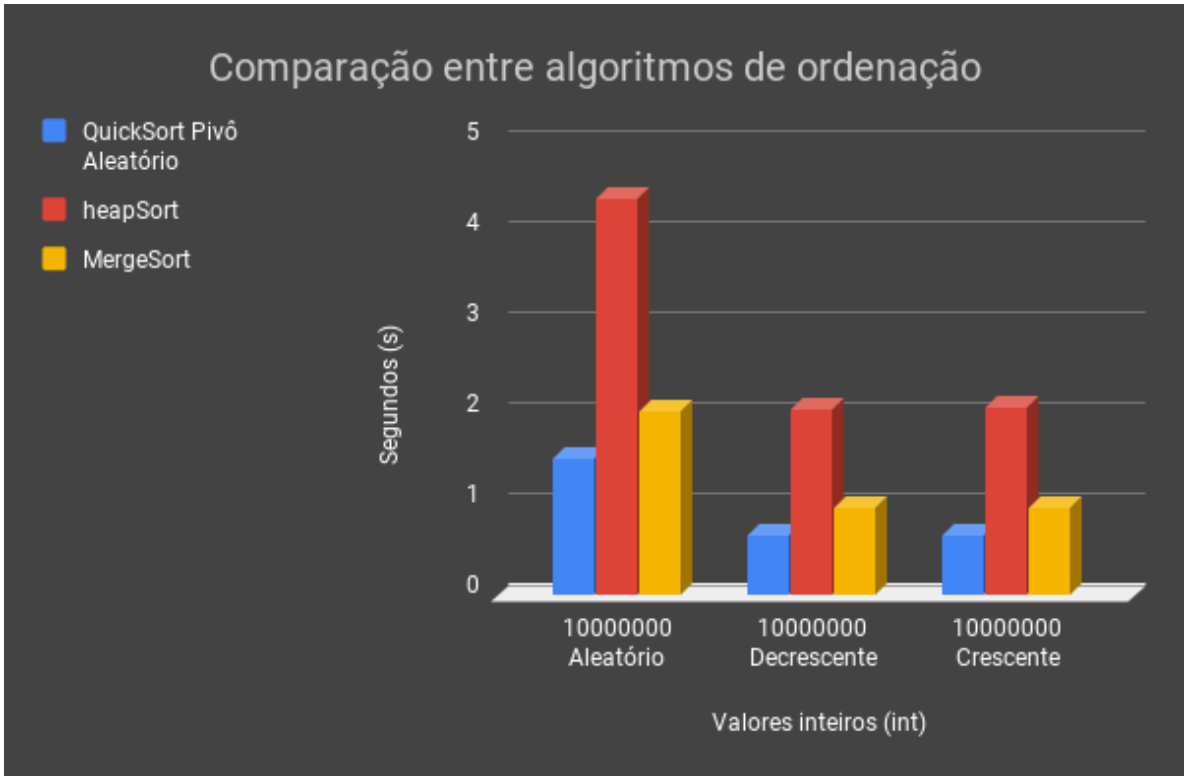
500.000 ELEMENTOS



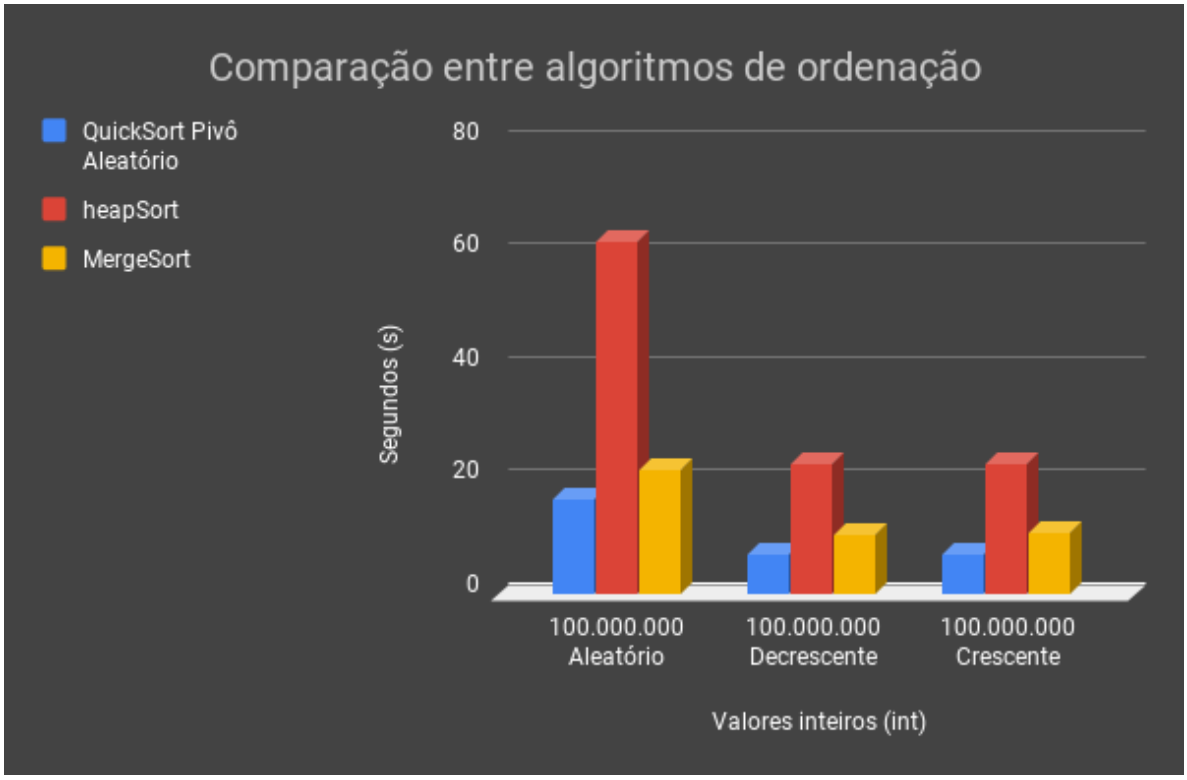
1.000.000 ELEMENTOS



10.000.000 ELEMENTOS

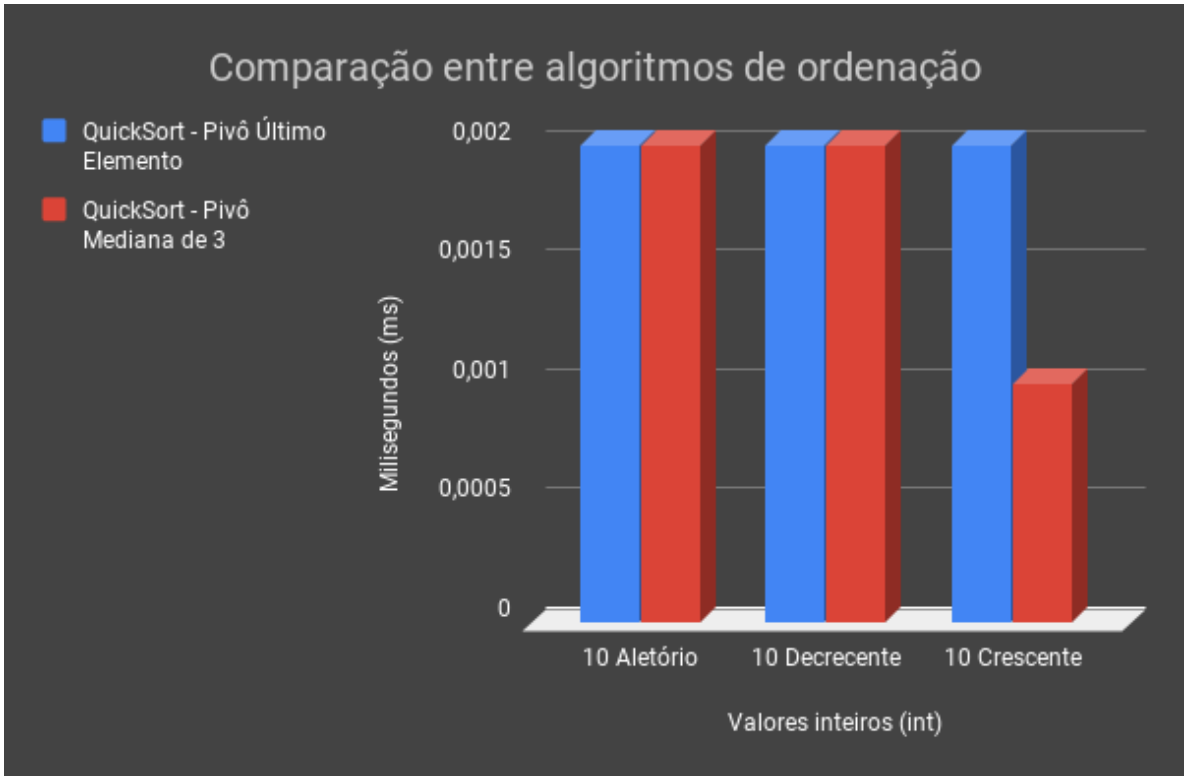


100.000.000 ELEMENTOS

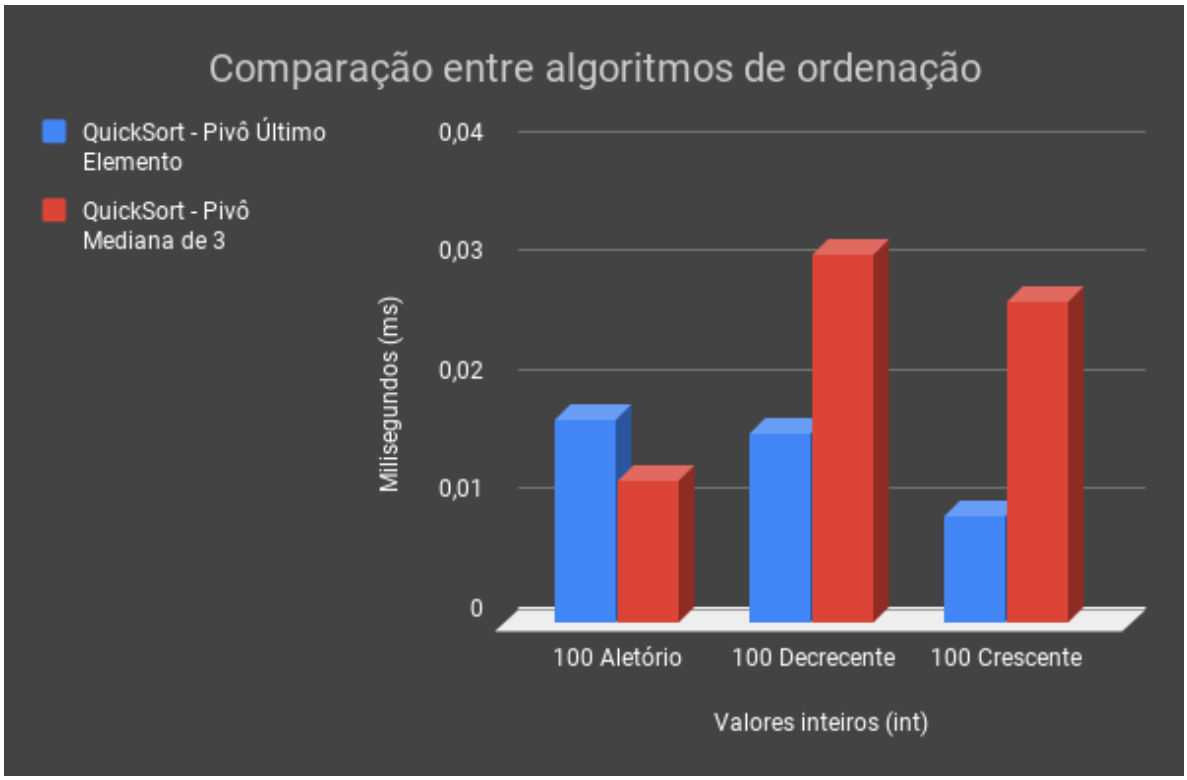


Comparação entre variações do quickSort

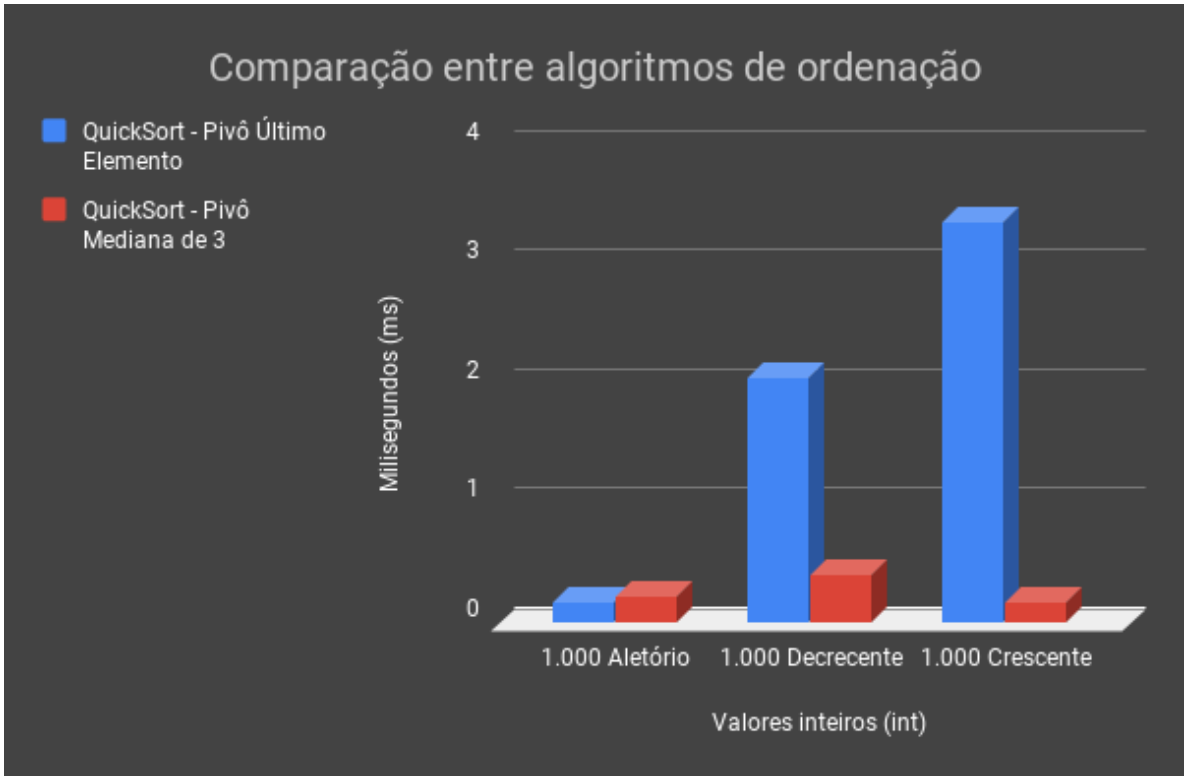
10 ELEMENTOS



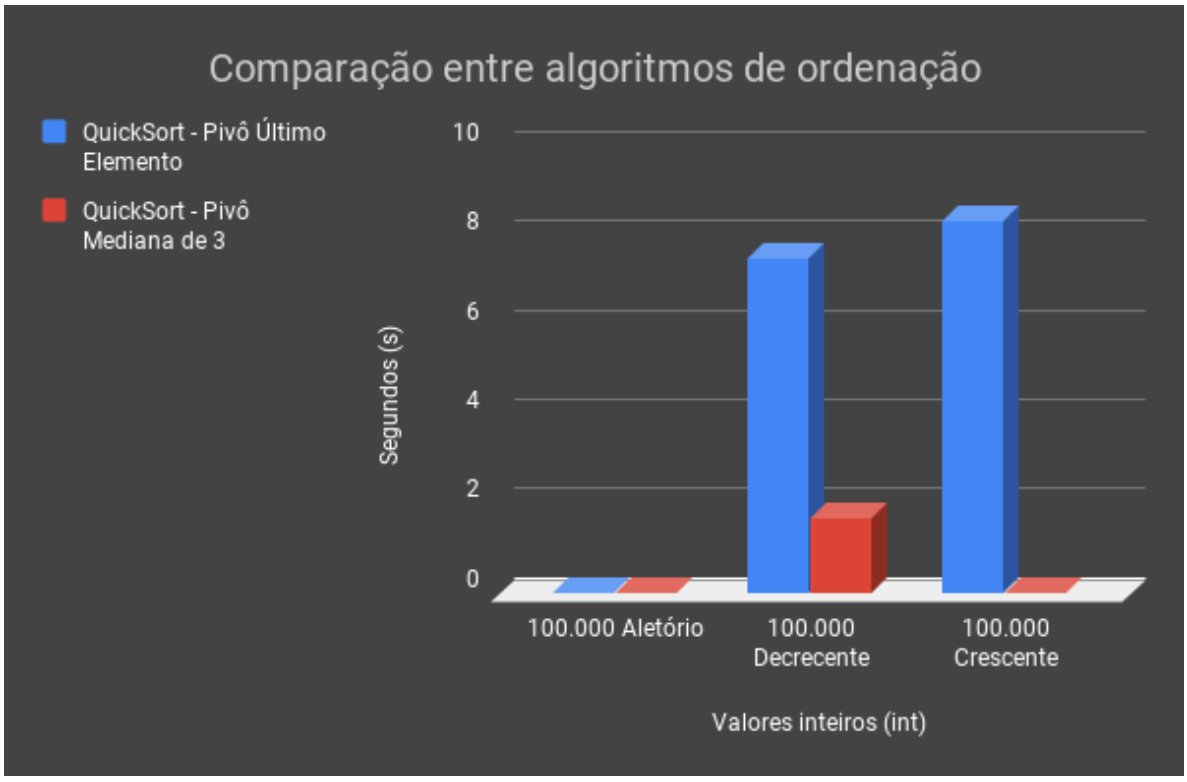
100 ELEMENTOS



1.000 ELEMENTOS



100.000 ELEMENTOS



> Nome: Fernando Lima
>
> Rgm: 36716