

Projeto Bimestral – COmparação de Algoritmos de Ordenação

Aluno: Gustavo Hammerschmidt.

Data: 18/06/2019

**Sumário**

Quicksort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Shellsort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

Heapsort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Mergesort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

Radixsort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

Introsort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

Tabela dos Dados. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

Gráficos das comparações: . . . . . . . . . . . . . . . . . . . . . . . . . 13

- Quase Ordenado . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

- Desordenado . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

- Decrescente . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

Referências Bibliográficas . . . . . . . . . . . . . . . . . . . . . . . . . . .17

**Quicksort:**

O Quicksort é um método de ordenação por comparação baseado na estratégia: dividir e conquistar (método de particionamento); ele divide um *array* em dois outros conforme um *pivot* previamente definido e os ordena – o que é bem rápido. É um algoritmo *não-estável* e seu melhor caso é de complexidade Ω(); sua complexidade no caso médio é igual à do melhor caso. Algumas implementações do Quicksort fazem com que seu pior caso seja θ(), pois escolhem o último elemento de um *array* como *pivot*. Contudo, a minha implementação do algoritmo escolheu o elemento do meio como *pivot*, o que faz com o pior caso do algoritmo seja igual aos outros dois casos: θ(). A ocupação de memória pelo algoritmo é, no melhor caso, de complexidade Ω(); e, no pior caso, é de complexidade θ(). Há algumas variações do Quicksort em que a complexidade do pior caso na ocupação de memória é igual a do melhor caso; a implementação do algoritmo utilizado não possui esta qualidade.

**Uma imagem contendo captura de tela

Descrição gerada com muito alta confiança**

No gráfico acima, vemos que o desempenho do algoritmo é bastante bom. Tem um ótimo desempenho com um *array* decrescente; levou mais tempo para ordenar um *array* quase ordenado; e o *array* desordenado teve um tempo próximo ao do quase ordenado no começo, mas, conforme o número de elementos aumentava, foi mais rapidamente ordenado. Vale constar que a diferença de tempo de ordenação do *array* quase ordenado para o decrescente sofre influência do modo como algoritmo começa a ordenar um *array*, pois, ordena da direita para a esquerda – o que é bom para um *array*  decrescente em questão de tempo; e talvez nem tanto para um *array* crescente, como visto com o *array*  quase ordenado, que é, praticamente, um *array* crescente. E o *array* desordenado teve um bom desempenho quando comparado ao quase ordenado e demorou um pouco mais quando comparado ao decrescente.

**Shellsort:**

O Shellsort é um método de ordenação bastante eficaz com pequenos conjuntos de elementos – a saber: o método utilizado é conhecido como ordenação por inserção e ele é utilizado logo após o algoritmo ter divido o conjunto de elementos em conjuntos “menores” ou melhores ordenados – ; e, dentre todos os algoritmos que possuem uma complexidade elevada no pior caso (como os de complexidade θ()), ele é o mais eficiente. Shellsort é um algoritmo *não-estável*, seu melhor caso é de complexidade Ω(); a complexidade do seu caso mediano depende do tamanho do *gap* – espaço, “pulo” que o algoritmo dá entre as comparações – escolhido pelo usuário que deve ser escolhido proporcionalmente ao tamanho do conjunto a ser analisado para melhor otimização – na minha implementação, foi selecionado o valor do gap como metade do tamanho do conjunto para todas as análises –; e a complexidade do seu pior caso também depende do valor do gap selecionado, contudo, se o valor do gap for bom para o conjunto, a complexidade no pior caso é de θ(). Já a complexidade de ocupação de memória pelo algoritmo é de, em todos casos, O().

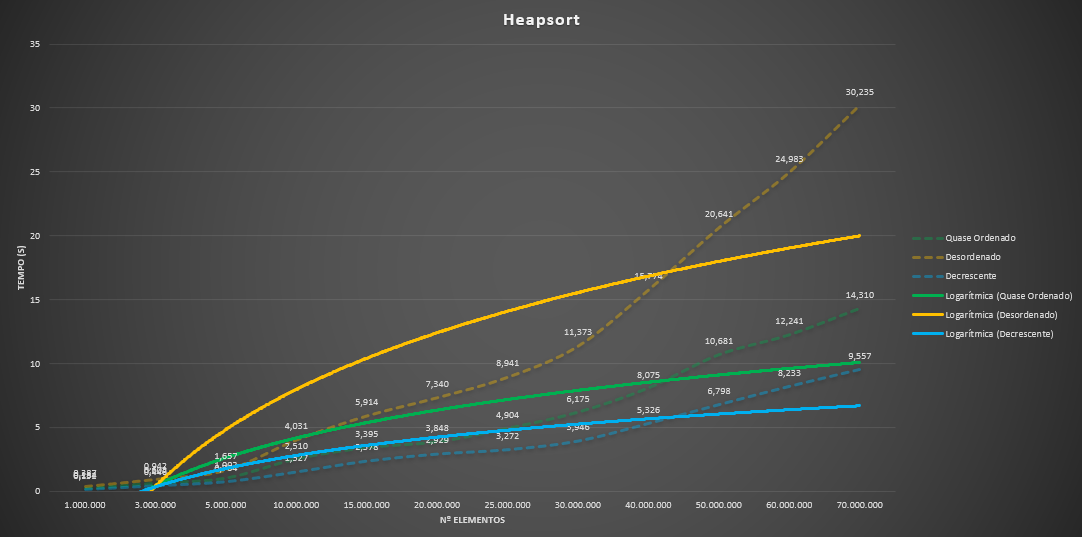
**Uma imagem contendo texto

Descrição gerada com alta confiança**

No gráfico acima, podemos observar que o algoritmo teve um bom desempenho com o *array* decrescente; e que os tempos dos *arrays* quase ordenado e desordenado sofreram várias ondulações conforme o algoritmo processava um número maior de elementos e o gap não era o ideal para a aplicação – o valor do gap permaneceu o mesmo durante o experimento todo: metade do vetor –; note que o algoritmo demorou mais para ordenar um array ordenado em vez de um array quase ordenado conforme o número de elementos do conjunto crescera. Note que o desempenho do algoritmo é quase o mesmo para os três conjuntos quando eles possuem menos de 5 milhões de elementos; o Shellsort faz comparações entre dois elementos com distância de um *gap*  entre eles, como o conjunto desordenado – em todas as comparações – realizará trocas antes de encurtar o tamanho do *gap*, ele acabará semi-ordenado enquanto o conjunto quase ordenado pode acabar com um conjunto um pouco mais bagunçado do que antes; esta diferença fica mais perceptível com conjuntos maiores, vemos que o conjunto quase ordenado se aproxima mais do conjunto desordenado no quesito tempo de execução do que do conjunto decrescente. Vale constar que a curva logarítmica aparece nos gráficos geralmente por volta dos conjuntos com 3 milhões de elementos porque, antes as esses conjuntos, os algoritmos, em geral, levam menos que um 1 segundo para ordenar e logaritmos de números menores que 1.0 tendem a serem negativos – tempo negativo, no experimento, é considerado como “instantâneo” uma vez que a diferença é de menos de alguns milissegundos (apenas para a curva logarítmica, lógico); a curva de incremento de tempo comprova que a diferença de ordenação de um *array* para o outro é consideravelmente pequena.

**Heapsort:**

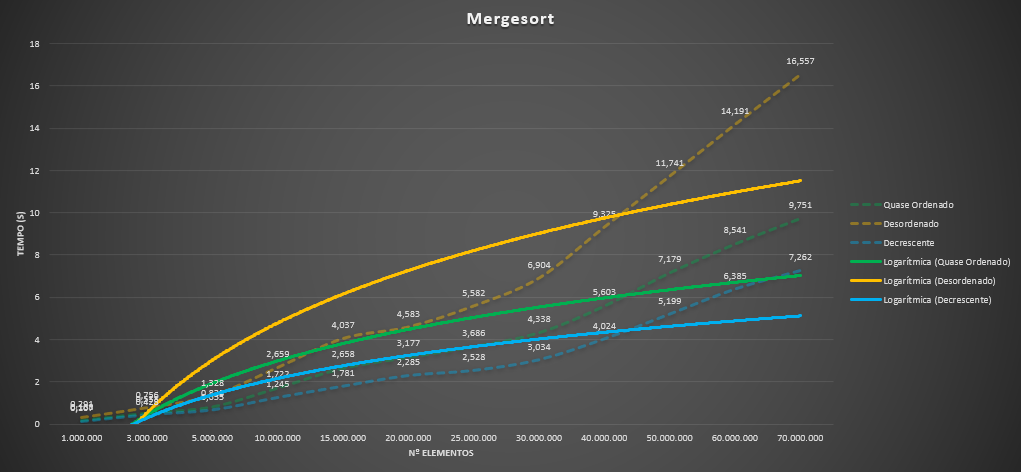
O Heapsort é um algoritmo de ordenação generalista muito bom e bem conciso – a saber: o método de ordenação utilizado é o de ordenação por seleção – ele utiliza uma estrutura de dados chamada de *heap* para ordenar os dados à medida que estes são inseridos na estrutura e, ao afinal das inserções, os elementos podem ser removidos na ordem desejada; o *heap pode* ser representado como uma árvore binária de busca ou como um vetor, e , já que queremos um *array* ordenado crescentemente, devemos construir um heap máximo – onde o maior elemento fica na raiz. O Heapsort é *não-estável*, seu desempenho é bem comportado – em todos os casos a complexidade é de O(). A complexidade de ocupação de memória pelo algoritmo é de, em todos casos, O().

****

No gráfico acima, notamos que o desempenho do algoritmo é melhor em conjuntos quase ordenados e decrescentes a partir de um número de elementos maior que 30 milhões, antes disso, o desempenho do algoritmo para um conjunto desordenado acompanha o desempenho dos outros dois conjuntos ligeiramente acima (mais demoradamente, claro) e, após 30 milhões, dispara. Vemos que o algoritmo ordena bem conjuntos quase ordenados e decrescentes; é possível notar que o conjunto quase ordenado demorou um pouco mais que o decrescente devido ao maior número de *swaps*(trocas) de posições para ordenar, afinal o *array* quase ordenado sofre várias trocas até que os elementos que estão fora de posição entrem no seu devido lugar, e o *array* decrescente realiza menos trocas pois é mais padronizado – após uma troca do elemento, é bem possível que ele já esteja na sua devida posição; novamente, essa diferença fica mais agravada conforme o tamanho dos conjuntos aumenta. Mas o desempenho dos dois conjuntos é bom quando comparados ao desordenado – que é o mais complexo de se ordenar para a implementação do algoritmo, afinal, a quantidade de trocas a serem realizadas é incerta, e, certamente, é agravada em maiores conjuntos.

**Mergesort:**

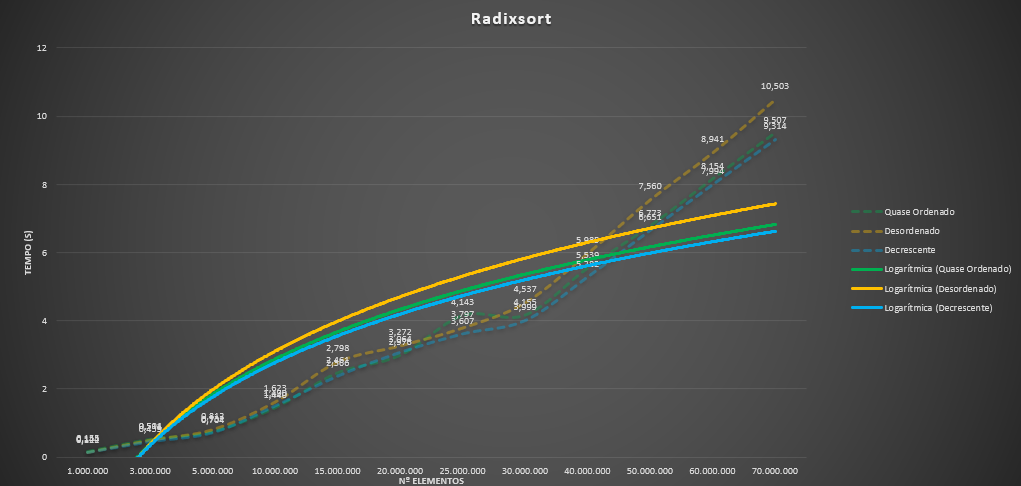
O Mergesort é um algoritmo de ordenação por mistura – realiza comparações e é do tipo dividir para conquistar (assim como o Quicksort); faz “misturas” (*merging*) . É um algoritmo *estável,* seu desempenho é de complexidade O() para todos os casos. E a complexidade de ocupação de memória do algoritmo é de O() em todos os casos – algumas implementações reduzem a complexidade a 1, não é o caso da minha implementação.

****

No gráfico acima, podemos notar que o Mergesort é um algoritmo bastante conciso com poucas “flutuações” no comportamento das curvas, e que é mais eficiente quando ordena conjuntos quase ordenados e decrescente se comparado à ordenação de um conjunto desordenado. Ele começa a demorar mais na ordenação de conjuntos desordenados com mais de 40 milhões de elementos em diante quando comparado aos outros dois conjuntos.

**Radixsort:**

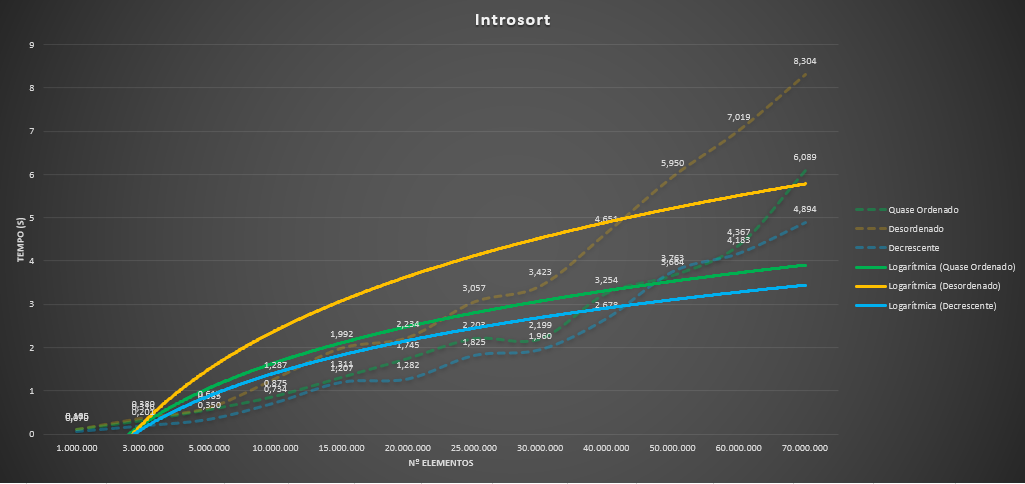
O Radixsort é um algoritmo *estável* que ordena os elementos de um conjunto com base nas chaves únicas dos elementos definidas pelo algoritmo e depois ordena as chaves. A versão do algoritmo utilizada é a LSD (*Least Significant Digit*), que ordena da esquerda para a direita, ou de ordem crescente, e é a mais utilizada para ordenar números inteiros. O Radixsort não é considerado um algoritmo de comparação pois sua complexidade foge à regra de que nenhum algoritmo tem tempo de ordenação melhor do que Ω(); sua complexidade é de O() para o caso mediano e o pior caso – onde *n* é o número de elementos e *k*  é o tamanho dos elementos a serem ordenados; para o melhor caso a complexidade é indefinida pois pode ser menor que a complexidade dos outros dois casos ou igual a eles. A complexidade de ocupação de memória pelo algoritmo é de O() para todos os casos – onde *n* é o número de elementos no conjunto e k é o tamanho dos números.

****

No gráfico acima, vemos que o Radixsort é um algoritmo bastante rápido quando comparado aos outros quatro avaliados neste relatório, levando menos de 10 segundos para conjuntos menores que 70 milhões; também é possível notar que a diferença de tempo de ordenação entre os três conjuntos é muita pequena e que há mudanças na liderança pelo pódio de mais demorado – o conjunto quase ordenado demorou mais que o desordenado com 25 milhões de elementos – mas, em geral o conjunto mais demorado de se ordenar foi o desordenado.

**Introsort:**

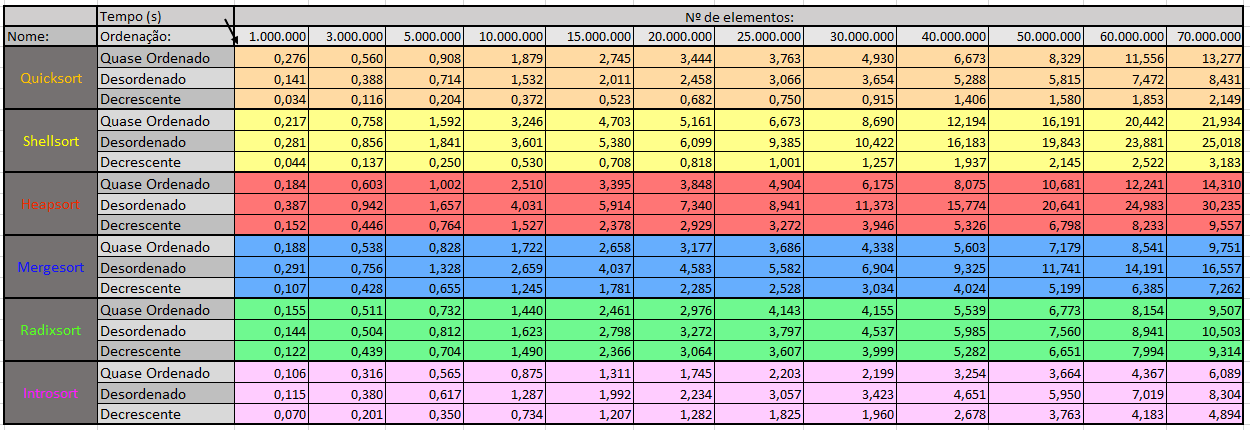
O Introsort ou *Introspective Sort*(Ordenação Introspectiva) é um algoritmo de ordenação híbrida: ele começa ordenando com o algoritmo Quicksort e, quando a quantidade de chamadas recursivas excede o logaritmo do número de elementos sendo ordenados, alterna para o Heapsort; pois, dessa forma, ele pega as melhores partes dos dois algoritmos – o que o faz bastante eficiente; a saber: o método de ordenação utilizado é de particionamento e de selecionamento. O Introsort é um algoritmo *não-estável*, sua complexidade para todos os casos é de O() e a complexidade de ocupação de memória pelo algoritmo é de O() para todos os casos.



No gráfico acima, notamos que o algoritmo demora mais para ordenar um conjunto desordenado; também percebemos que o tempo para o conjunto desordenado com 70 milhões é de aproximadamente 8,3 segundos em média – um tempo muito abaixo dos tempos dos outros algoritmos para o mesmo conjunto. Quanto aos conjuntos quase ordenado e decrescente, o algoritmo teve seus altos e baixos na hora de hibridizar a performance, sendo, em alguns momentos, tão bom na ordenação de um quase ordenado quanto um decrescente; ainda assim, obteve tempos muito bons para os dois conjuntos posto em vista as performances dos demais algoritmos.

**Tabela dos dados:**

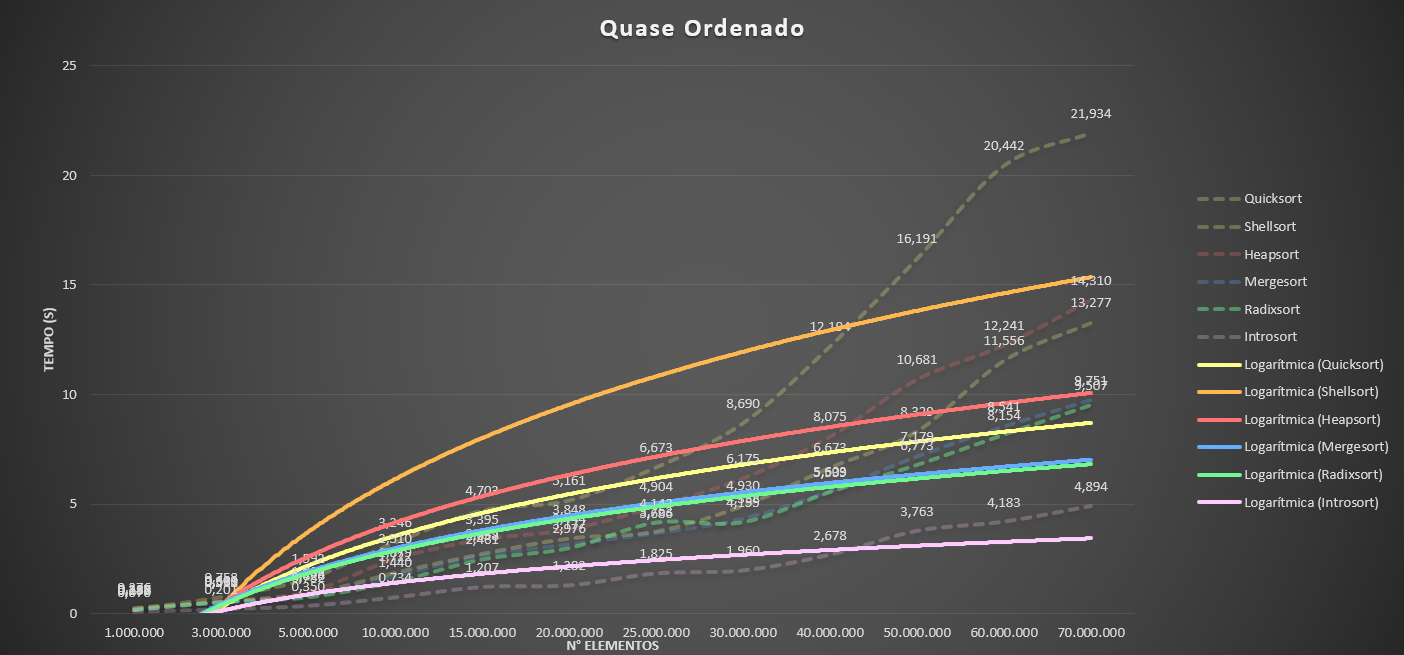
Esta é a tabela utilizada em todos os gráficos deste relatório, ela foi feita no Excel, portanto, como não tenho certeza se é necessário enviar a prova viva de que os dados dos gráficos são condizentes com a tabela abaixo, enviarei um arquivo .xlsx junto a este arquivo e o projeto em Java.



**Gráficos das Comparações:**

**- Quase\_Ordenado:**

**Uma imagem contendo texto, mapa

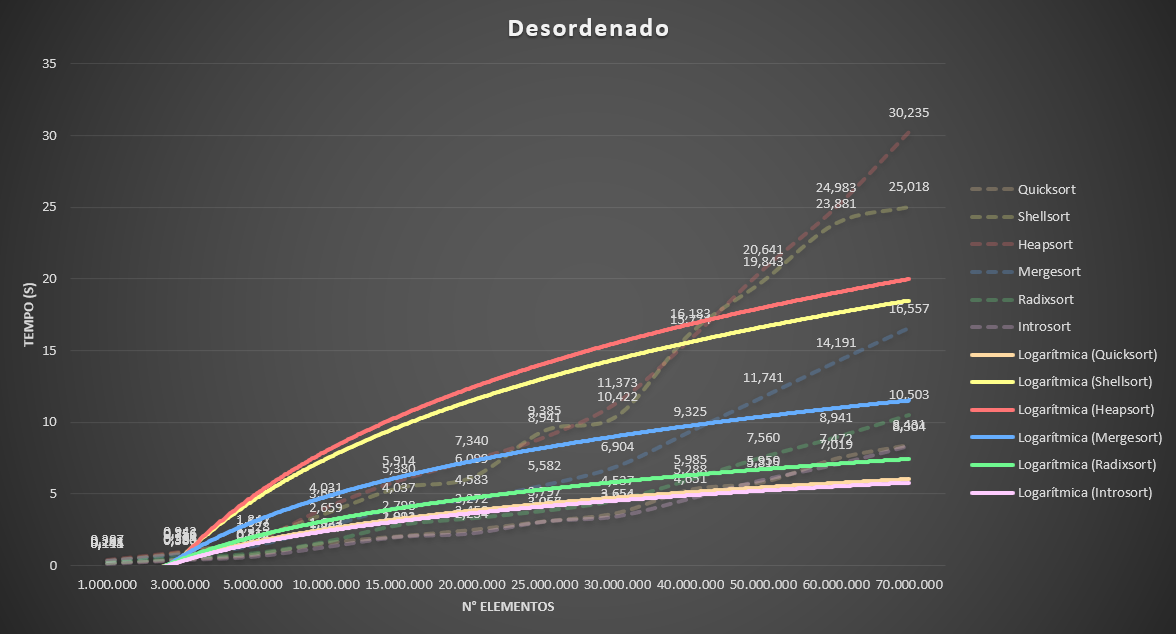
Descrição gerada com muito alta confiança**

Podemos observar que o algoritmo Shellsort teve o pior desempenho neste conjunto com um tempo bem elevado quando comparado com o segundo colocado: Heapsort, e os demais algoritmos. Note que o Quicksort consegue manter o mesmo intervalo de tempo entre ele mesmo e o Heapsort por um bom tempo, mas, conforme o tamanho do conjunto aumenta, ele se aproxima do Heapsort – a curva logarítmica de crescimento deixa bem definido esse intervalo. Também podemos observar que o Mergesort e o Radixsort andaram lado a lado enquanto a este conjunto em avaliação – como demonstrado pela curva logarítmica de crescimento. E, por fim, o melhor dos algoritmos, neste conjunto, foi o Introsort com um tempo muito satisfatório para um conjunto com muitos elementos (70 milhões) posto em vista o desempenho dos demais.

**Desordenado:**

**Uma imagem contendo texto

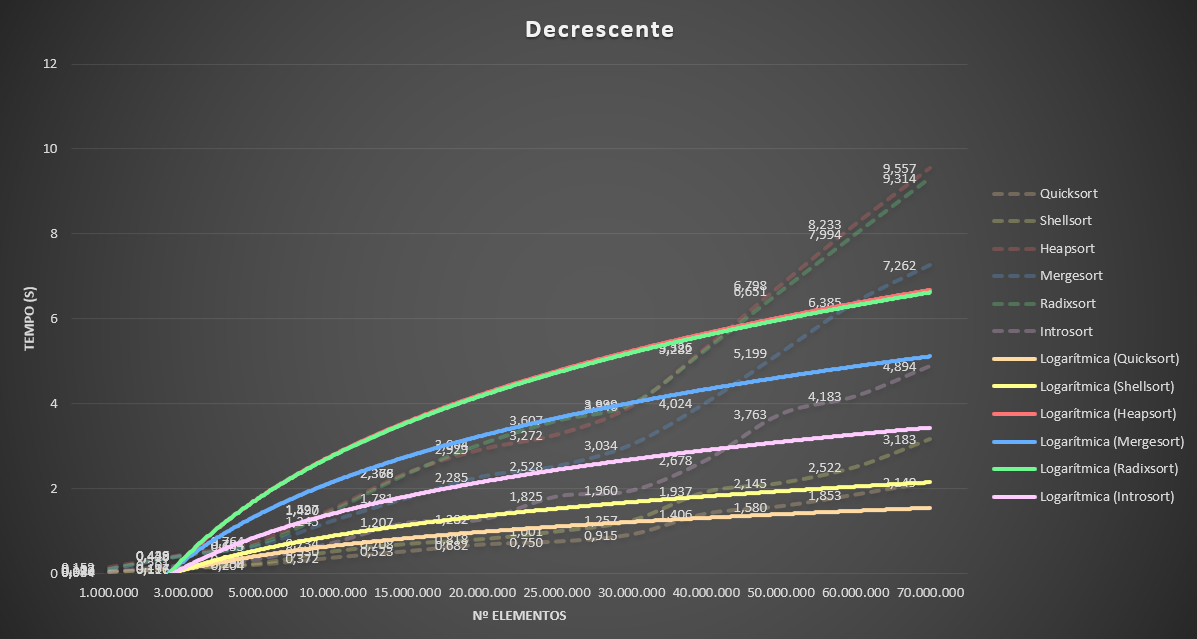
Descrição gerada com muito alta confiança**

****

É notável que os algoritmos Heapsort e Shellsort competem pelo pódio de pior desempenho neste conjunto, com tempos bem elevados quando comparados ao Mergesort, o terceiro colocado com um tempo satisfativo para todos os conjuntos a partir de 20 milhões. Vemos que o Quicksort teve seu melhor campo de atuação posto em jogo e competiu com a sua versão aprimorada: Introsort; os dois tiveram altos e baixos, mas, no fim, o Introsort manteve-se conciso por mais tempo e cumpriu o seu propósito sendo o melhor neste conjunto. Quanto ao Radixsort, não há muito que dizer teve um bom desempenho mais ficou em terceiro na corrida pelo melhor desempenho.

**Decrescente:**

**Uma imagem contendo texto, mapa

Descrição gerada com alta confiança** Neste conjunto, observamos que o Quicksort(devido a ordenação da direita para a esquerda por conta da implementação do algoritmo) foi o melhor; seguido do Shellsort e, mais atrás, o Introsort(híbrido de Quicksort e Heapsort). Enquanto os piores foram o Heapsort, o Radixsort e o Mergesort respectivamente. Porém, todos os algoritmos mantiveram-se abaixo de 10 segundos para um conjunto decrescente de 70 milhões – o que é um resultado muito bom. Contudo, vemos que os desempenhos dos algoritmos neste conjunto foram bastante concisos – como demonstrado nas curvas de incremento de tempo.

**Referências Bibliográficas:**

Quick Sort; link: <https://en.wikipedia.org/wiki/Quicksort> .

Shell Sort; link: <https://pt.wikipedia.org/wiki/Shell_sort> .

Heap Sort; link: <https://pt.wikipedia.org/wiki/Heapsort> .

Merge Sort; link: <https://pt.wikipedia.org/wiki/Merge_sort> .

Radix Sort; link: <https://pt.wikipedia.org/wiki/Radix_sort> .

Intro Sort; link: <https://en.wikipedia.org/wiki/Introsort> .

Algoritmos de Ordenação; link: <https://en.wikipedia.org/wiki/Sorting_algorithm> .

Ordenação Estável; link: <https://pt.wikipedia.org/wiki/Ordena%C3%A7%C3%A3o_est%C3%A1vel> .

Overview do Introsort; link: <https://www.techiedelight.com/introsort-algorithm/> .

Algoritmos de Ordenação; link: <https://www.geeksforgeeks.org/sorting-algorithms/> .