X Semana de Ciência e Tecnologia IFMG - *Campus* Bambuí

X Jornada Científica

# Estudo comparativo de métodos de ordenação utilizando bases de dados de nomes

**Júlio César Machado Álvares [1]; Laerte Mateus Rodrigues[2];**

[1]Estudante de Engenharia de Computação. Instituto Federal Minas Gerais (IFMG) *campus* Bambuí. Rod. Bambuí/Medeiros km 5. CEP: 38900-000. Bambuí-MG. [2]Professor Orientador – IFMG.

**RESUMO –** Os algoritmos de ordenação são amplamente utilizados na computação uma vez que o conjunto a ser ordenado pode ter características que tornam um algoritmo mais eficiente que o outro. Com o objetivo de construir um cenário prático para este tipo de análise foi utilizado uma base de dados com 19.997 nomes e aplicado os algoritmos *Heap sort, Quick sort, Shell sort, Merge sort e Insertion sort*.

**Palavras-chave:** Ordenação lexicográfica, Análise de Algoritmos, Algoritmos de ordenação.

## INTRODUÇÃO

Das diversas áreas da Ciência da Computação, o estudo da complexidade de algoritmos tem grande peso, por determinar quais algoritmos são melhores para determinadas situações. Um dos problemas pouco triviais clássicos é a ordenação de elementos. No decorrer da evolução da computação, vários métodos foram desenvolvidos afim de reduzir o custo computacional e consequentemente sendo mais eficientes, tendo grandes destaques, como por exemplo, o *Quick sort*.

Uma aplicação de métodos de ordenação é encontrado, por exemplo, em bancos de dados, onde não há garantia de ordenação física dos seus registros, sendo assim, quando necessário o mesmo tem de ordenar os registros segundo os critérios informados na consulta SQL.

O trabalho tem como objetivo fazer uma comparação geral entre cinco métodos de ordenação, sendo eles *Insertion sort, Heap sort, Quick sort, Merge sort e Shell sort*, e desenvolver um programa para uso educacional em aulas de complexidade de algoritmos.

## METODOLOGIA

Afim de simular um ambiente mais próximo da realidade, foi extraído da base de dados "Adventure Works Relational Dataset repository" (MOTL, 2014) 19.997 nomes da tabela *Person* unindo as colunas *FirstName, MiddleName* e *LastName.*

Cada cenário foi feito afim de fazer com que o algoritmo trabalhasse o máximo que o cenário esboçasse. O caso médio, que foi tomado como base, fez com que todos os algoritmos trabalhassem na base de dados *in natura*. O melhor e o pior caso dependeram de um tratamento inicial na base de dados, sendo este, para o melhor caso, ordenar a base antes de submeter o algoritmo e o pior caso, análogamente mas ordenada de forma inversa a primeira. Ordenar a base de dados inicialmente faz com que os algoritmos de ordenação executem a menor quantidade possível de passos, sendo que não há movimentos a serem feitos, pois a base já se encontra ordenada, com exceção do *Quick sort* que apresenta uma degenaração no melhor caso. Ordenar inversamente os dados cria um cenário onde o algoritmo não poderá aplicar a *priori*, uma estratégia que reduza seu esforço para a tarefa proposta.

Todos os algoritmos têm características únicas e todas elas foram preservadas, apenas a forma como é comparado os valores foi modificado. O *insertion sort* tem como caracteristica principal a criação de uma matriz inserindo uma única vez cada elemento e um elemento por vez, garantindo que todos já estão no seu devido lugar com apenas uma inserção (BIERNACKI, 2013). Diferente do *insertion*, o *heap sort* usa uma abstração de árvore para ordenar os elementos, onde os maiores elementos são inseridos em direção à raiz, ou seja, todos os filhos são menores que os pais (CORMEN, 2008). O *quick* e o *merge sort* apresentam idéias muito semelhantes, ambos usando estratégias de divisão e conquista para ordenar os elementos, a diferença entre eles é que o *quick* divide o vetor em sub-vetores e define um pivô, jogando todos os elementos maiores para a direita e os menores para a esquerda em relação ao pivô e após isso reconstrói o vetor, o *merge sort* divide o vetor em várias partes, resolve o sub-vetor e depois intercala os elementos no vetor original, garantindo que estão nos seus lugares (CORMEN, 2008). O *shell sort* implementa o insertion sort de forma otimizada, sendo que ele divide os maiores grupos em menores, arranjando-os na matriz e na sua última passada por ela, é puramente *insertion sort* (KNUTH, 1998).

Para comparar os nomes, foi usado o método lexicográfico, que consiste em uma ordem natural do produto cartesiano, ou seja, a palavra que tem uma letra menor que outra é considerada menor e também é levado em consideração o tamanho da palavra, sendo que, se uma cadeia de caractéres é o início de outra, então a que possui a menor quantidade de caractéres é a menor (HORSTMANN, 2009). Afim de fazer essa comparação, foi implementado o seguinte código em C++ apresentado no Algoritmo 1.

```
1 bool max_string(string s1, string s2){
2     int i = 0;
3     while(i < s1.length() && i < s2.length()){
4           if(tolower(s1[i]) < tolower(s2[i])) return true;
5           else if(tolower(s1[i])>tolower(s2[i])) return false;
6           i++;
7     }
8     return i >= s1.length();
9 }
```

Algoritmo 1: Função de comparação lexicográfica.

O funcionamento do código consiste em retornar verdadeiro se a primeira palavra for menor e falso se não. Para determinar qual, o laço de repetição executa enquanto a variável auxiliar for menor que o comprimento de ambas as palavras, comparando letra com letra de cada uma, se as letras forem iguais a variável auxiliar é incrementada. A função *tolower* retorna o número correspondente a posição da letra minúscula na tabela ASCII, sendo que maiúsculas e minúsculas diferem na tabelas ASCII, mas não diferem na ordenação lexicográfica. O laço termina quando a variável auxiliar alcança o tamanho de uma das palavras. Se as palavras forem completamente idênticas, o resultado tanto faz, e é retornado falso. A complexidade dessa função é da ordem de O(h), sendo "h" o tamanho da menor palavra.

Para computar o tempo de execução de cada algoritmo, foi usado os tipos de dados *clock_t*, da biblioteca *ctime*, nativa do C++. Seu funcionamento consiste em marcar quantos *clocks* o processador teve entre o início e o fim da execução do algoritmo, usando a função *clock* e após isso aplicar a seguinte fórmula: $time = (end - begin) / CLOCKSPERSEC$

*Time* é o tempo de execução, *end* e *begin* são os marcos de início e fim dos *clocks* e *CLOCKSPERSEC* é uma constante da biblioteca que define quantos *clocks* são feitos em um segundo (C TIME, 2017).

## RESULTADOS E DISCUSSÃO

A Figura 1 apresenta a relação do tempo gasto por cada algoritmo nos 3 cenários apresentados anteriormente.

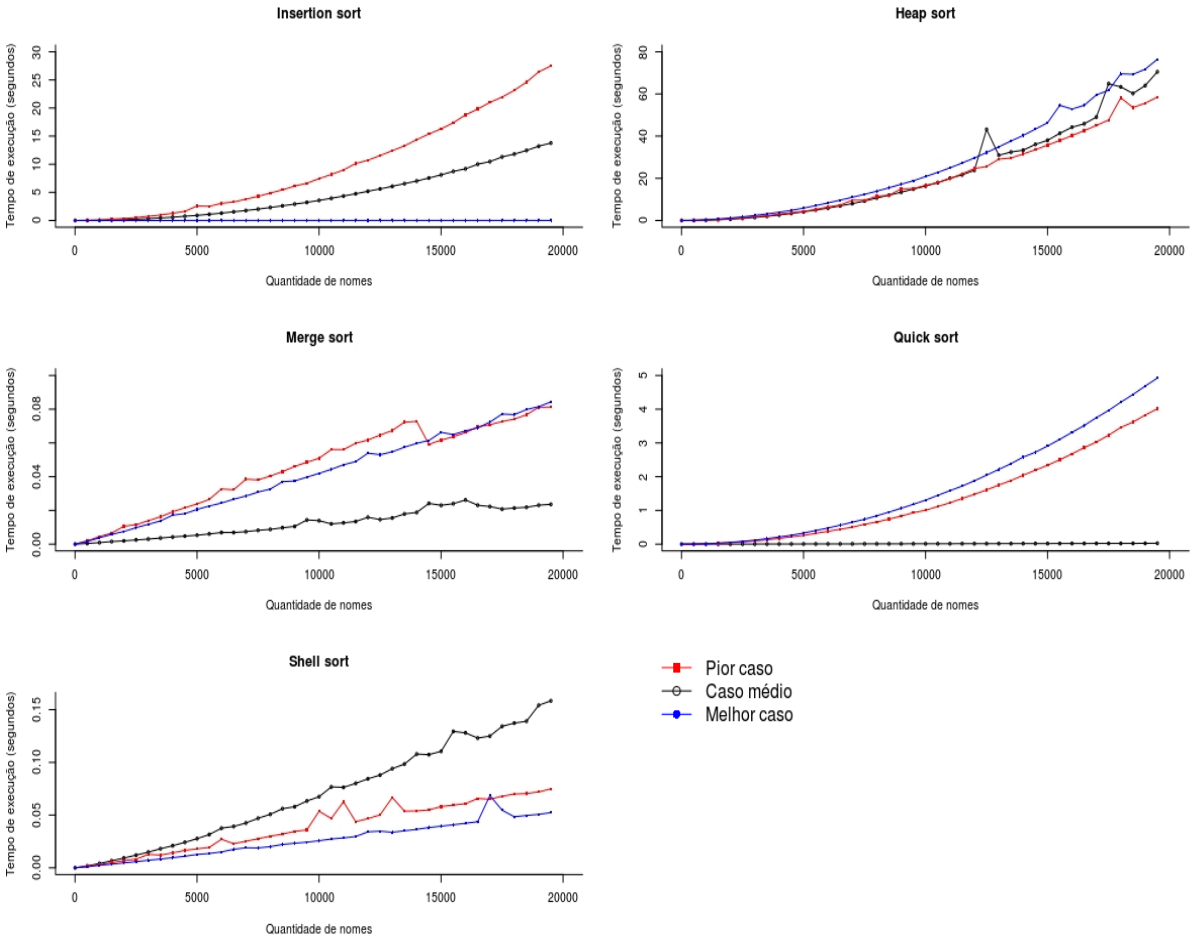


*Figura 1: Relação do tempo de execução de cada algoritmo de ordenação nos 3 cenários apresentados anteriormente para um conjunto de 19.997 nomes.*

Os algoritmos *Shell sort, Merge sort e Heap sort* não uma taxa de crescimento no seu tempo de execução um comportamento regular devido ao custo de comparação dos nomes. Uma vez que ao comparar chaves inteiras o custo é constante, mas ao fazer comparação lexicográfica o custo dela também influencia na execução do algoritmo.

Para o algoritmo *Merge sort*, o custo de itercalação se apresentou constante à partir de aproximadamente 14.500 elementos devido a uma possível semelhança da quantidade de comparações, devido a isso, o custo do pior e do melhor caso foram próximos. Os algoritmos H*eap sort* e S*hell sort* também apresentam picos devido ao mesmo fator, ou seja, o custo da comparação nestes casos isolados afetou o tempo total.

O algoritmo *Quick sort* para o caso médio apresenta custo próximo de O(n log n), enquanto para o pior caso e o melhor caso, sua complexidade tende a apresentar comportamento

quadrático. Tal comportamento deve-se à posição do pivô na implementação do algoritmo, sendo que o pivô é deslocado do início ao fim do sub-vetor, estando ordenado ou inversamente ordenado.

## CONCLUSÕES

É perceptível que com o avanço da computação e a otimização de agoritmos, tarefas como ordenação de conjuntos não necessariamente seguem um comportamento como apresentado pela literatura. Fatores externos à ordenação influenciam em seu resultado, como mostrado que o custo da comparação lexicográfica afeta o tempo de execução fazendo com que alguns algoritmos de ordenação não apresentem uma taxa de crescimento segundo seu custo assintótico.

Para determinar um melhor algoritmo é necessário analisar o cenário no qual ele será submetido, por exemplo, se a quantidade de elementos for razoável e o problema não depender diretamente do tempo de execução da ordenação, o algoritmo *Insertion sort* atende perfeitamente as necessidades, mesmo apresentando a maior complexidade, sendo que para até cerca de 5000 elementos, seu custo comparado ao *Quick sort* não diverge de mais de 1 segundo.

## REFERÊNCIAS BIBLIOGRÁFICAS

KNUTH, D. E. **The Art of Computer Programming**, volume 3: Sorting and Searching. Addison. 1998.

CORMEN, Thomas H. **Introduction to algorithms**. MIT press, 2009.

MOTL, Jon. **The CTU Prague Relational Learning Repository**, 2014. Disponível em: <https://relational.fit.cvut.cz/dataset/AdventureWorks>, Acesso em: 10 de setembro de 2017.

**C Time Library,** The C++ Resources Network. Disponível em: <http://www.cplusplus.com/reference/ctime/>, Acesso em: 10 de setembro de 2017.

**Tolower Function**, The C++ Resources Network. Disponível em: <http://www.cplusplus.com/reference/cctype/tolower/>, Acesso em: 10 de setembro de 2017.

BIERNACKI, Christophe; JACQUES, Julien. A generative model for rank data based on insertion sort algorithm. **Computational Statistics & Data Analysis**, v. 58, p. 162-176, 2013.

HORSTMANN, Cay. **Conceitos de computação com o essencial de C++**. Bookman Editora, 2009.