Algoritmos de ordenação: um estudo comparativo

Gabriel de Oliveira Guedes Nogueira
ICMC, Universidade de São Paulo

15 de Setembro, 2019

1 Introdução

Algoritmos de ordenação são métodos usados para se colocar elementos de uma lista em uma certa ordem. Desde os primórdios da computação, essa classe de algoritmos atraiu bastante pesquisas, que buscam tornar os métodos de ordenação cada vez mais eficientes. Nos dias atuais, com as volumosas quantidades de dados que caracterizam o fenômeno do *Big Data*, a importância desses algoritmos passa a ser ainda maior. Nesse contexto, o presente trabalho visa fazer uma análise comparativa entre os algoritmos de ordenação mais clássicos: o *bubble sort* (normal e com sentinela), o *selection sort*, o *insertion sort*, o *shellsort*, o *heapsort*, o *quicksort* e o *merge sort*.

Para tanto, serão usados dados obtidos pela execução de cada um desses métodos em diferentes cenários através de um programa escrito na linguagem C. Cada algoritmo será executado cinco vezes em vetores aleatórios, quase ordenados (cada elemento distante de sua posição correta por não mais do que 10 posições), ordenados e inversamente ordenados de tamanho variando entre 10^2 e 10^6. Cada algoritmo terá o seu tempo de execução, número de comparações feitas e número de trocas feitas contabilizados em cada um desses casos e uma média da performance do algoritmo será obtida para cada cenário.

Os valores obtidos serão dispostos em tabelas e plotados usando a linguagem R, a fim de facilitar a sua comparação. Ao final do trabalho, esperamos ter uma noção clara sobre quais são os melhores algoritmos para cada tipo de situação.

2 Fundamentação Teórica

Serão oito os algoritmos de ordenação aqui estudados, cada um com as suas vantagens e desvantagens. Antes de iniciarmos, contudo, devemos ter alguns conceitos e termos básicos em mente. Um *arquivo* de tamanho n é uma sequência de n itens $r[0], r[1]...r[n-1]$. Uma *chave* $k[i]$ é associada à cada registro $r[i]$ e geralmente é um dos campos desse registro. Quando classificamos os registros de um arquivo a partir de seus campos chaves, estamos realizando uma *ordenação pela chave*. Ademais, um algoritmo de ordenação é dito *estável* caso ele preserve a ordem relativa de registros de chaves iguais, isto é, se registros com chaves iguais aparecem na sequência ordenada na mesma ordem em que estavam na sequência original.

Analisaremos, inicialmente, os algoritmos de ordenação mais intuitivos. O talvez mais famoso deles é o *bubble sort*, também chamado de *ordenação por flutuação*. A

sua estratégia consiste em percorrer o arquivo continuamente trocando a posição de registros adjacentes caso estejam fora de ordem. A versão otimizada desse método, chamada de *bubble sort com sentinela*, faz uso de uma variável *sentinela* (ou *flag*) para parar o algoritmo caso, em uma dada iteração, não tenham havido trocas de registros, significando que o arquivo já está ordenado. Essa característica faz com que esse algoritmo se saia bastante bem quando recebe um arquivo já ordenado ou quase ordenado, apresentando, nesse caso, complexidade de tempo $O(n)$. Para os demais casos, contudo, trata-se de um dos métodos de ordenação menos eficientes, com complexidade de tempo $O(n^2)$. Tanto o *bubble sort* quanto a sua versão otimizada são estáveis.

O *selection sort*, também chamado de *ordenação por seleção*, apresenta uma estratégia ainda mais intuitiva. Enquanto o arquivo não estiver ordenado, o algoritmo irá buscar pelo registro com menor chave na parte não ordenada do arquivo e irá colocá-lo em sua devida posição. Apesar de pouco eficiente, apresentando uma complexidade de tempo $O(n^2)$ em todos os casos, o *selection sort* é um dos métodos de ordenação que realiza o menor número de trocas, apresentando, nesse quesito, complexidade $O(n)$. Não é estável.

Finalmente, o último e mais eficiente dos algoritmos de ordenação intuitivos é o *insertion sort*. A sua estratégia baseia-se na inserção de um registro em um arquivo já ordenado (considere que um arquivo de um único registro já está ordenado!), lembrando bastante a ordenação que costuma-se fazer em um jogo de cartas tradicional. A cada iteração, o algoritmo escolhe um registro (geralmente o primeiro) da parte não ordenada do arquivo e o compara com os registros da parte já ordenada, que contém, inicialmente, apenas o primeiro elemento do arquivo. Esses registros, caso necessário, são então deslocados para a direita até que a posição do registro a ser inserido seja encontrada. Nos piores casos e nos casos médios, o *insertion sort* apresenta complexidade de tempo $O(n^2)$, enquanto que em seu melhor caso, quando recebe um arquivo já ordenado ou quase ordenado, apresenta complexidade de tempo $O(n)$. Trata-se de um algoritmo de ordenação estável.

O primeiro algoritmo avançado que analisaremos é o *shellsort*, uma versão melhorada do *insertion sort*. Enquanto que o seu progenitor realiza a troca apenas de registros adjacentes, o método de Shell também realiza a troca entre registros distantes um do outro por um intervalo chamado de *gap*. A estratégia desse algoritmo é aplicar uma série de *insertion sorts* com *gaps* decrescentes, até que o arquivo esteja ordenado. É essencial que o último *gap* a ser aplicado seja de tamanho 1, visto que, nesse caso, a ordenação resultante é equivalente ao *insertion sort* tradicional, garantindo, assim, a ordenação do arquivo. O *shellsort* é instável e a sua complexidade é extremamente

dependente do método que se usa para gerar a sequência de *gaps*. Nesse trabalho, usaremos o método de Ciura, que faz uso de uma sequência obtida empiricamente e acarreta uma complexidade indetermina ao *shellsort*.

O *heapsort*, por sua vez, é o algoritmo de ordenação clássico que talvez apresente a estratégia mais confusa. Esse método consiste em transformar o arquivo em uma estrutura de dados chamada *heap*, uma implementação de uma árvore binária na qual a chave de cada nó é maior ou igual à chave de seus filhos. Note que, para que essa propriedade seja satisfeita, a raiz da heap (o seu primeiro elemento) deve ser o seu registro de maior chave. O *heapsort* aproveita-se dessa propriedade para obter a ordem correta dos registros, transformando o arquivo em uma *heap* a cada vez que uma troca é realizada. Esse método de ordenação apresenta complexidade de tempo $O(nlogn)$, é indiferente ao fato de o arquivo recebido estar ordenado ou não e não é estável.

Um dos algoritmos de ordenação mais rápidos e mais usados é o *quicksort*. Sua estratégia, baseada no conceito de divisão e conquista, consiste nos seguintes passos: escolher um registro, chamado *pivot*, do arquivo; reordenar o arquivo de forma que elementos menores ou iguais ao *pivot* venham antes dele, enquanto que elementos maiores do que o *pivot* venham depois dele; aplicar, de forma recursiva, os passos acima para o sub-arquivo com elementos menores ou iguais ao *pivot* e para o sub-arquivo com elementos maiores do que o *pivot*. O processo de escolha do *pivot* e reordenação do arquivo é chamado de *particionamento* e é a etapa mais importante do algoritmo. Em média, o *quicksort* é extremamente rápido, realizando $O(nlogn)$ comparações e apresentando excelentes resultados quando comparado com outros algoritmos clássicos. Em seu pior caso, contudo, esse algoritmo realizará $O(n^2)$ comparações, uma complexidade tão ruim quanto à do *bubble sort*. Esse caso ocorre quando o *pivot* escolhido é o maior ou menor elemento do arquivo. As chances de que o pior caso ocorra podem ser reduzidas drasticamente caso usemos um bom método para a escolha do *pivot*. Nesse trabalho, o método adotado foi o de escolha aleatória do *pivot*. Note que o *quicksort* não é estável.

Finalmente, o último algoritmo de ordenação aqui analisado é o *merge sort*. Assim como o *quicksort*, o *merge sort* faz uso do conceito de divisão e conquista. Sua estratégia consiste em dividir o arquivo não-ordenado em n sub-arquivos com 1 registro (um arquivo com um único registro já está ordenado!) e, então, repetidamente combinar (*merge*) os sub-arquivos para produzir novos arquivos ordenados. Ao final do processo, o arquivo resultante estará ordenado. Note que, ao contrário do *quicksort*, cuja parte mais importante é o particionamento do arquivo, a parte mais importante do *merge sort* é a combinação dos sub-arquivos. Esse método de ordenação é estável

e realiza $O(nlogn)$ comparações em todos os seus casos. Apesar do bom desempenho, a maior desvantagem do *merge sort* é precisar de $O(n)$ de espaço auxiliar. Todos os algoritmos vistos até aqui, com exceção do *quicksort*, que usa $O(logn)$ de espaço auxiliar, fazem uso apenas de $O(1)$ de espaço auxiliar.

3 Resultados e Discussão

VETOR ALEATÓRIO	10^2	10^3	10^4	10^5	10^6
Bubble Sort	$6.3 * 10^{-5}$ s $5 * 10^3$ comp. $2.5 * 10^3$ trocas	$2.9 * 10^{-3}$ s $5 * 10^5$ comp. $2.5 * 10^5$ trocas	0.3s $5 * 10^7$ comp. $2.5 * 10^7$ trocas	33.8s $5 * 10^9$ comp. $2.5 * 10^9$ trocas	3366s $5 * 10^{11}$ comp. $2.5 * 10^{11}$ trocas
Bubble Sort c/ sentinela	$6.4 * 10^{-5}$ s $5 * 10^3$ comp. $2.5 * 10^3$ trocas	$2.7 * 10^{-3}$ s $5 * 10^5$ comp. $2.5 * 10^5$ trocas	0.3s $5 * 10^7$ comp. $2.5 * 10^7$ trocas	34.2s $5 * 10^9$ comp. $2.5 * 10^9$ trocas	3422s $5 * 10^{11}$ comp. $2.5 * 10^{11}$ trocas
Selection Sort	$3.4 * 10^{-5}$ s $5 * 10^3$ comp. 10^2 trocas	$1.3 * 10^{-3}$ s $5 * 10^5$ comp. 10^3 trocas	0.12s $5 * 10^7$ comp. 10^4 trocas	11.3s $5 * 10^9$ comp. 10^5 trocas	1126s $5 * 10^{11}$ comp. 10^6 trocas
Insertion Sort	$3.8 * 10^{-5}$ s $2.5 * 10^3$ comp. $2.5 * 10^3$ trocas	$1.8 * 10^{-3}$ s $2.5 * 10^5$ comp. $2.5 * 10^5$ trocas	0.16s $2.5 * 10^7$ comp. $2.5 * 10^7$ trocas	15.7s $2.5 * 10^9$ comp. $2.5 * 10^9$ trocas	1561s $2.5 * 10^{11}$ comp. $2.5 * 10^{11}$ trocas
Heapsort	$2.9 * 10^{-5}$ s $1.5 * 10^3$ comp. $4.1 * 10^2$ trocas	$2.1 * 10^{-4}$ s $2.5 * 10^4$ comp. $7.3 * 10^3$ trocas	$2.3 * 10^{-3}$ s $3.5 * 10^5$ comp. $1.1 * 10^5$ trocas	0.03s $4.5 * 10^6$ comp. $1.4 * 10^6$ trocas	0.36s $5.5 * 10^7$ comp. $1.7 * 10^7$ trocas
Shellsort	$1.8 * 10^{-5}$ s $7 * 10^2$ comp. $3.3 * 10^2$ trocas	$1.7 * 10^{-4}$ s $1.3 * 10^4$ comp. $6.4 * 10^3$ trocas	$1.9 * 10^{-3}$ s $1.9 * 10^5$ comp. $9.8 * 10^4$ trocas	0.026s $2.6 * 10^6$ comp. $1.3 * 10^6$ trocas	0.36s $4 * 10^7$ comp. $2.6 * 10^7$ trocas
Quicksort	$1.8 * 10^{-5}$ s $9.7 * 10^2$ comp. $2.3 * 10^2$ trocas	$1.4 * 10^{-4}$ s $1.4 * 10^4$ comp. $3.1 * 10^3$ trocas	$1.3 * 10^{-3}$ s $1.9 * 10^5$ comp. $3.8 * 10^4$ trocas	0.016s $2.3 * 10^6$ comp. $4.6 * 10^5$ trocas	0.18s $2.8 * 10^7$ comp. $5.4 * 10^6$ trocas
Merge Sort	$2.5 * 10^{-5}$ s $5.4 * 10^2$ comp. $6.7 * 10^2$ trocas	$1.8 * 10^{-4}$ s $8.7 * 10^3$ comp. 10^4 trocas	$1.9 * 10^{-3}$ s $1.2 * 10^5$ comp. $1.3 * 10^5$ trocas	0.023s $1.5 * 10^6$ comp. $1.7 * 10^6$ trocas	0.25s $1.9 * 10^7$ comp. $2 * 10^7$ trocas

Figure 1: Desempenho dos algoritmos em vetores aleatórios.

Para testar a performance dos algoritmos, foram usados vetores totalmente aleatórios, vetores quase ordenados, vetores inversamente ordenados e vetores ordenados. O vetor quase ordenado é tal que cada elemento está a uma distância não maior do que dez posições da sua posição correta. Há cinco vetores de cada tipo, com tamanhos

variando de 10^2 à 10^6. Cada algoritmo foi executado cinco vezes em vetores de todos os tipos com tamanho variando dentro do limite especificado. Um dado vetor de um determinado tipo e tamanho será alimentado, sem sofrer modificações, à todos os algoritmos. As médias do tempo de execução, número de comparações e número de trocas realizadas por cada algoritmo foram calculados para cada tamanho e tipo de vetor. Os dados foram dispostos em tabelas e gráficos plotados com a linguagem R. Nas tabelas, as melhoras performances para cada cenário foram sublinhadas com cores diferentes para cada uma das categorias: azul para o melhor tempo de execução, verde para o melhor número de comparações e amarelo para o melhor número de trocas.

VETOR QUASE ORDENADO	10^2	10^3	10^4	10^5	10^6
Bubble Sort	$3.2 * 10^{-5}$ s $5 * 10^3$ comp. 130 trocas	$1.8 * 10^{-3}$ s $5 * 10^5$ comp. $1.3 * 10^3$ trocas	0.12s $5 * 10^7$ comp. $1.3 * 10^4$ trocas	11.6s $5 * 10^9$ comp. $1.3 * 10^5$ trocas	1132s $5 * 10^{11}$ comp. $1.3 * 10^6$ trocas
Bubble Sort c/ sentinela	$9.2 * 10^{-6}$ s $7 * 10^2$ comp. 130 trocas	$5.7 * 10^{-5}$ s $8* 10^3$ comp. $1.3 * 10^3$ trocas	$4.6 * 10^{-4}$ s $8 * 10^4$ comp. $1.3 * 10^4$ trocas	$4.2 * 10^{-3}$ s 10^6 comp. $1.3 * 10^5$ trocas	0.042 s 10^7 comp. $1.3 * 10^6$ trocas
Selection Sort	$3.1 * 10^{-5}$ s $5 * 10^3$ comp. 66 trocas	$1.6 * 10^{-3}$ s $5 * 10^5$ comp. 673 trocas	0.12s $5 * 10^7$ comp. $6.8 * 10^3$ trocas	11.5s $5 * 10^9$ comp. $6.8 * 10^4$ trocas	1115s $5 * 10^{11}$ comp. $6.8 * 10^5$ trocas
Insertion Sort	$7.2 * 10^{-6}$ s 234 comp. 136 trocas	$3.4 * 10^{-5}$ s $2.3 * 10^3$ comp. $1.3 * 10^3$ trocas	$3 * 10^{-4}$ s $2.3 * 10^4$ comp. $1.3 * 10^4$ trocas	$2.6 * 10^{-3}$ s $2.3 * 10^5$ comp. $1.3 * 10^5$ trocas	0.023 s $2.3 * 10^6$ comp. $1.3 * 10^6$ trocas
Heapsort	$2.9 * 10^{-5}$ s $1.6 * 10^3$ comp. 431 trocas	$2 * 10^{-4}$ s $2.6 * 10^4$ comp. $7.6 * 10^3$ trocas	$2 * 10^{-3}$ s $3.6 * 10^5$ comp. $1.1 * 10^5$ trocas	0.025s $4.6 * 10^6$ comp. $1.4 * 10^6$ trocas	0.27s $5.6 * 10^7$ comp. $1.8 * 10^7$ trocas
Shellsort	$9.2 * 10^{-6}$ s $5 * 10^2$ comp. 10^2 trocas	$5.6 * 10^{-5}$ s $7.8 * 10^3$ comp. 10^3 trocas	$5.3 * 10^{-4}$ s $1.1 * 10^5$ comp. 10^4 trocas	$6.4 * 10^{-3}$ s $1.4 * 10^6$ comp. 10^5 trocas	0.06s $1.5 * 10^7$ comp. 10^6 trocas
Quicksort	$1.6 * 10^{-5}$ s $9.3 * 10^2$ comp. $1.7 * 10^2$ trocas	$9.3 * 10^{-5}$ s $1.4 * 10^4$ comp. $1.7 * 10^3$ trocas	$9.4 * 10^{-4}$ s $1.8 * 10^5$ comp. $1.7 * 10^4$ trocas	0.011s $2.3 * 10^6$ comp. $1.7 * 10^5$ trocas	0.11s $2.8 * 10^7$ comp. $1.7 * 10^6$ trocas
Merge Sort	$2.2 * 10^{-5}$ s 408 comp. $6.7 * 10^2$ trocas	$1.3 * 10^{-4}$ s $5.7 * 10^3$ comp. 10^4 trocas	$1.3 * 10^{-3}$ s $7.5 * 10^4$ comp. $1.3 * 10^5$ trocas	0.015s $9.1 * 10^5$ comp. $1.7 * 10^6$ trocas	0.15s $1.1 * 10^7$ comp. $2 * 10^7$ trocas

Figure 2: Desempenho dos algoritmos em vetores quase ordenados.

Para vetores gerados aleatoriamente, o *bubble sort*, mesmo em sua versão otimizada, foi o algoritmo com o pior desempenho, levando quase 1h para ordenar arquivos com

10^6 registros. O *quicksort*, ao contrário, mostrou-se extremamente rápido para todos os tamanhos de vetores, ressaltando a sua eficiência de tempo de execução em casos médios. O *merge sort*, apesar de ser o método de ordenação que mais requere espaço auxiliar, realizou o menor número de comparações. O *selection sort*, por sua vez, apesar de seu desempenho geral ruim, destacou-se como o algoritmo que menos realiza trocas de registros, tanto para vetores gerados aleatoriamente quanto para os demais tipos de vetores.

VETOR INVERSAMENTE ORDENADO	10^2	10^3	10^4	10^5	10^6
Bubble Sort	$4.8 * 10^{-5}$ s $5 * 10^3$ comp. $5 * 10^3$ trocas	$3.1 * 10^{-3}$ s $5 * 10^5$ comp. $5 * 10^5$ trocas	0.3s $5 * 10^7$ comp. $5 * 10^7$ trocas	30.7s $5 * 10^9$ comp. $5 * 10^9$ trocas	2952s $5 * 10^{11}$ comp. $5 * 10^{11}$ trocas
Bubble Sort c/ sentinela	$5.3 * 10^{-5}$ s $5 * 10^3$ comp. $5 * 10^3$ trocas	$3.1 * 10^{-3}$ s $5 * 10^5$ comp. $5 * 10^5$ trocas	0.3s $5 * 10^7$ comp. $5 * 10^7$ trocas	30.7s $5 * 10^9$ comp. $5 * 10^9$ trocas	2967s $5 * 10^{11}$ comp. $5 * 10^{11}$ trocas
Selection Sort	$2.5 * 10^{-5}$ s $5 * 10^3$ comp. 50 trocas	$1.3 * 10^{-3}$ s $5 * 10^5$ comp. $5 * 10^2$ trocas	0.12s $5 * 10^7$ comp. $5 * 10^3$ trocas	12.5s $5 * 10^9$ comp. $5 * 10^4$ trocas	1189s $5 * 10^{11}$ comp. $5 * 10^5$ trocas
Insertion Sort	$4.6 * 10^{-5}$ s $5 * 10^3$ comp. $5 * 10^3$ trocas	$3.4 * 10^{-3}$ s $5 * 10^5$ comp. $5 * 10^5$ trocas	0.32s $5 * 10^7$ comp. $5 * 10^7$ trocas	32.3s $5 * 10^9$ comp. $5 * 10^9$ trocas	3165s $5 * 10^{11}$ comp. $5 * 10^{11}$ trocas
Heapsort	$2.5 * 10^{-5}$ s $1.5 * 10^3$ comp. $4.1 * 10^2$ trocas	$2.2 * 10^{-4}$ s $2.5 * 10^4$ comp. $7.3 * 10^3$ trocas	$1.9 * 10^{-3}$ s $3.5 * 10^5$ comp. $1.1 * 10^5$ trocas	0.022s $4.5 * 10^6$ comp. $1.4 * 10^6$ trocas	0.25s $5.5 * 10^7$ comp. $1.7 * 10^7$ trocas
Shellsort	$4.8 * 10^{-6}$ s $5.8 * 10^2$ comp. $2.5 * 10^2$ trocas	$1.16 * 10^{-4}$ s $9.3 * 10^3$ comp. $3 * 10^3$ trocas	$7 * 10^{-4}$ s $1.3 * 10^5$ comp. $3.8 * 10^4$ trocas	0.0088s $1.8 * 10^6$ comp. $1.3 * 10^6$ trocas	0.22s $4 * 10^7$ comp. $2.6 * 10^7$ trocas
Quicksort	$1.1 * 10^{-5}$ s $9.3 * 10^2$ comp. 183 trocas	$1.25 * 10^{-4}$ s $1.3 * 10^4$ comp. $1.8 * 10^3$ trocas	$8.4 * 10^{-4}$ s $1.8 * 10^5$ comp. $1.8 * 10^4$ trocas	0.009s $2.3 * 10^6$ comp. $1.8 * 10^5$ trocas	0.1s $2.8 * 10^7$ comp. $1.8 * 10^6$ trocas
Merge Sort	$1.2 * 10^{-5}$ s 316 comp. $6.7 * 10^2$ trocas	$1.5 * 10^{-4}$ s $4.9 * 10^3$ comp. 10^4 trocas	$1.1 * 10^{-3}$ s $6.4 * 10^4$ comp. $1.3 * 10^5$ trocas	0.012s $8.1 * 10^5$ comp. $1.7 * 10^6$ trocas	0.14s 10^7 comp. $2 * 10^7$ trocas

Figure 3: Desempenho dos algoritmos em vetores inversamente ordenados.

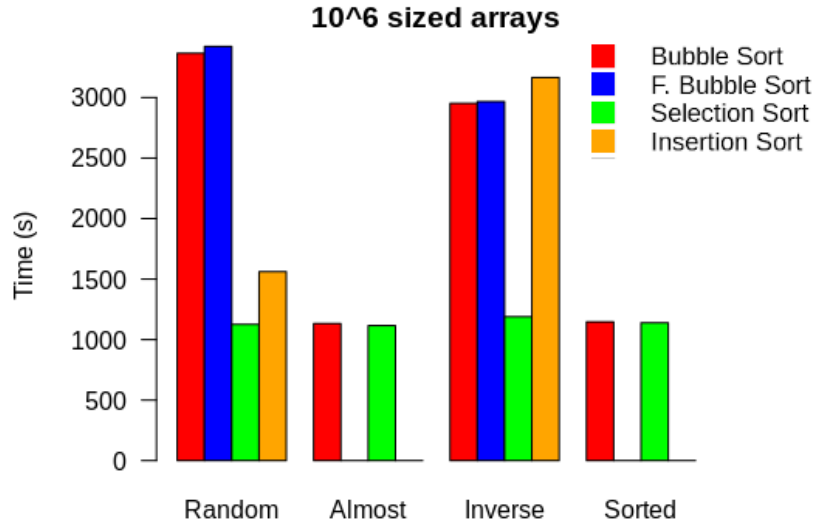
Para vetores quase ordenados, os algoritmos mais simples apresentaram o melhor desempenho. O *insertion sort* apresentou o tempo de execução mais rápido e o menor número de comparações, seguido de perto pelo *bubble sort com sentinela*. O *shellsort*, apesar de seu excelente desempenho nesse cenário, não foi capaz de ultrapassar o seu

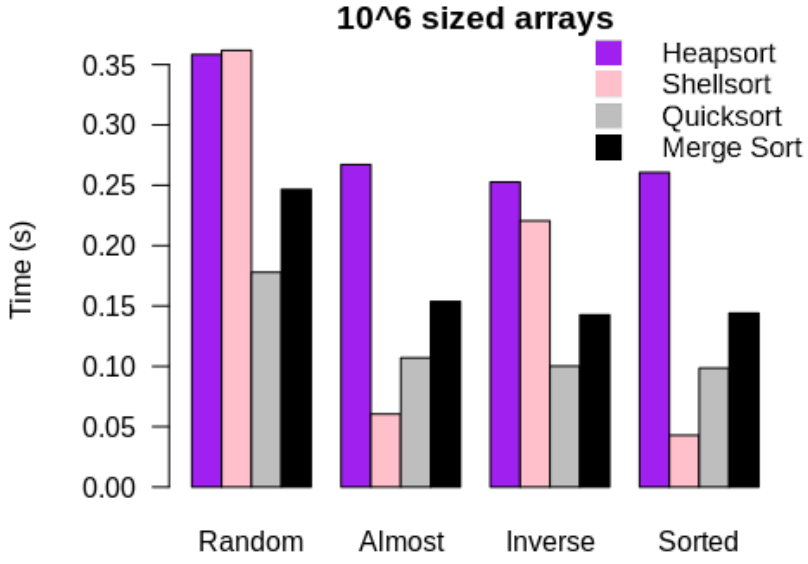
progenitor (*o insertion sort*), muito provavelmente devido ao *overhead* acrescentado ao algoritmo pelo uso de *gaps*, que não se mostraram tão úteis para situações em que os vetores estão quase ordenados.

VETOR ORDENADO	10^2	10^3	10^4	10^5	10^6
Bubble Sort	$2.5 * 10^{-5}$ s $5 * 10^3$ comp. 0 trocas	$1.2 * 10^{-3}$ s $5 * 10^5$ comp. 0 trocas	0.11s $5 * 10^7$ comp. 0 trocas	11.4s $5 * 10^9$ comp. 0 trocas	1146s $5 * 10^{11}$ comp. 0 trocas
Bubble Sort c/ sentinela	10^{-6} s 100 comp. 0 trocas	$2.8 * 10^{-6}$ s 10^3 comp. 0 trocas	$5.6 * 10^{-5}$ s 10^4 comp. 0 trocas	$2.5 * 10^{-4}$ s 10^5 comp. 0 trocas	0.002 s 10^6 comp. 0 trocas
Selection Sort	$2.5 * 10^{-5}$ s $5 * 10^3$ comp. 0 trocas	$1.2 * 10^{-3}$ s $5 * 10^5$ comp. 0 trocas	0.12s $5 * 10^7$ comp. 0 trocas	11.4s $5 * 10^9$ comp. 0 trocas	1139s $5 * 10^{11}$ comp. 0 trocas
Insertion Sort	10^{-6} s 100 comp. 0 trocas	$1.1 * 10^{-5}$ s 10^3 comp. 0 trocas	$7 * 10^{-5}$ s 10^4 comp. 0 trocas	$3.3 * 10^{-4}$ s 10^5 comp. 0 trocas	0.003 s 10^6 comp. 0 trocas
Heapsort	$2.7 * 10^{-5}$ s $1.6 * 10^3$ comp. 445 trocas	$2 * 10^{-4}$ s $2.6 * 10^4$ comp. $7.7 * 10^3$ trocas	$2 * 10^{-3}$ s $3.6 * 10^5$ comp. $1.1 * 10^5$ trocas	0.022s $4.6 * 10^6$ comp. $1.4 * 10^6$ trocas	0.26s $5.6 * 10^7$ comp. $1.8 * 10^7$ trocas
Shellsort	$6 * 10^{-6}$ s $4 * 10^2$ comp. 0 trocas	$6 * 10^{-5}$ s $6.8 * 10^3$ comp. 0 trocas	$3.5 * 10^{-4}$ s $9.6 * 10^4$ comp. 0 trocas	$4.3 * 10^{-3}$ s $1.3 * 10^6$ comp. 0 trocas	0.043s $1.4 * 10^7$ comp. 0 trocas
Quicksort	$1.3 * 10^{-5}$ s $9.3 * 10^2$ comp. $1.3 * 10^2$ trocas	$1.2 * 10^{-4}$ s $1.4 * 10^4$ comp. $1.3 * 10^3$ trocas	$8.3 * 10^{-4}$ s $1.8 * 10^5$ comp. $1.3 * 10^4$ trocas	0.009s $2.3 * 10^6$ comp. $1.3 * 10^5$ trocas	0.1s $2.8 * 10^7$ comp. $1.3 * 10^6$ trocas
Merge Sort	$1.2 * 10^{-5}$ s 356 comp. $6.7 * 10^2$ trocas	$1.6 * 10^{-4}$ s $5 * 10^3$ comp. 10^4 trocas	$1.2 * 10^{-3}$ s $7 * 10^4$ comp. $1.3 * 10^5$ trocas	0.013s $8.5 * 10^5$ comp. $1.7 * 10^6$ trocas	0.144s 10^7 comp. $2 * 10^7$ trocas

Figure 4: Desempenho dos algoritmos em vetores ordenados.

Para vetores inversamente ordenados, não houve uma variação grande no tempo de execução dos algoritmos. O *quicksort* mostrou-se o mais rápido para vetores pequenos e grans, enquanto que o *shellsort* obteve um melhor desempenho para vetores de tamanho médio. O *merge sort* apresentou, novamente, a menor quantidade de comparações dentre os algoritmos analisados. O número de trocas realizadas pelo *bubble sort* e pelo *insertion sort* foi significativamente maior do que para os outros casos. Já para o *selection sort*, o número de trocas manteve-se inalterado em relação aos cenários anteriores.





Para vetores já ordenados, o campeão absoluto em todas as categorias foi o *bubble sort com sentinela*. Trabalhando em $O(n)$ neste cenário, o algoritmo mostrou velocidade de execução bastante superior aos demais, sendo seguido de perto somente pelo *insertion sort*. Por não possuir uma *flag* para indicar quando o vetor já está ordenado, o *bubble sort* normal apresentou novamente um resultado ruim, apesar de haver melhoras

em relação aos cenários anteriores. Nota-se aqui que mesmo recebendo um vetor já ordenado o *heapsort* irá transformá-lo em uma *heap*, de forma que o seu desempenho nesse caso apresenta pouca diferença em relação aos demais.

NOTA: a maior parte dos gráficos plotados em R a partir dos dados obtidos não foi colocada neste artigo, a fim de mantê-lo organizado. Caso o leitor tenha interesse, pode obtê-los executando o código em R que acompanha este trabalho.

4 Considerações Finais

Observa-se, a partir do exposto, que não há algoritmo de ordenação perfeito. Nenhum dos métodos de ordenação aqui analisados foi capaz de apresentar o melhor desempenho em todas as categorias. Nesse contexto, a melhor escolha sempre depende das características do problema em questão.

Nota-se, no entanto, que as aplicações para certos algoritmos são bastante restritas, quando não inexistentes. O *bubble sort* regular, por exemplo, não se destacou em nenhum dos cenários. Sua versão otimizada mostrou bom desempenho apenas em casos de vetores ordenados ou quase ordenados. O *selection sort*, por sua vez, apesar de apresentar o melhor número de trocas para todos os casos, tem aplicação bastante restrita, visto que situações em que um bom número de trocas e mais importante do que um bom tempo de execução e um bom número de comparações são bastante raras.

Por outro lado, o *shellsort* e principalmente o *quicksort* demonstraram um bom desempenho geral, sobretudo nos casos médios (vetores aleatórios). Isso faz com que esses algoritmos sejam a escolha ideal para situações em que não sabemos o que esperar do problema.

5 Referências

CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, CLIFFORD. ***Introduction to Algorithms****. 2ed. The MIT Press, 2001.*

CORMEN, T et al. **Algorithms**. Disponível em: <`https://www.khanacademy.org/computing/computer-science/algorithms`>. Acesso em: 15 de Setembro, 2019.

Contribuidores da Wikipedia. **Sorting algorithm**. Disponível em: <`https://en.wikipedia.org/wiki/Sorting_algorithm`>. Acesso em: 15 de Setembro, 2019.

FEOFILOFF, P. **Projeto de Algoritmos em C**. Disponível em: <`https://www.ime.usp.br/~pf/algoritmos/`>. Acesso em: 15 de Setembro, 2019.