Comparação de algoritmos de ordenação em comparações, trocas e tempo

Daniel de Lima Franceschetti[1]

Rodrigo Tamae[2]

RESUMO

Esta pesquisa compara o desempenho dos algoritmos de ordenação Bubble sort, Insertion sort, Quicksort, Mergesort e Heapsort, em termos de comparações, trocas e tempo de execução. Utilizando vetores pré-ordenados (em ordem crescente, decrescente e aleatória) e de tamanhos variados (2000, 16000 e 128000 inteiros), os resultados demonstram a ineficiência dos algoritmos O(n^2) com cargas maiores, destacando a superioridade dos algoritmos O($n$log$n$), especialmente o Quicksort, em termos de tempo de execução; o que confirma os resultados já documentados na literatura e observados na prática sobre o desempenho dos algoritmos de ordenação. As conclusões destacam a importância da escolha do algoritmo de acordo com o contexto do problema e sugerem áreas para pesquisas futuras, como análises mais aprofundadas dos algoritmos.

Palavras-chave: Algoritmos de ordenação, estudo comparativo, tempo de execução, Bubble sort, Insertion sort, Quick sort, Merge sort, Heap sort, linguagem de programação C.

ABSTRACT

This survey compares the performance of sorting algorithms Bubble sort, Insertion sort, Quicksort, Mergesort, and Heapsort in terms of comparisons, swaps, and execution time. By employing pre-sorted arrays (in ascending, descending, and random order) and varying sizes (2000, 16000, and 128000 integers), the findings demonstrate the inefficiency of O(n^2) algorithms with larger workloads, while highlighting the superiority of O($n$log$n$) algorithms, particularly Quicksort, in terms of execution time. This reaffirms previously documented literature and practical observations regarding sorting algorithm performance. The conclusions underscore the importance of algorithm selection based on problem context and suggest areas for future research, such as more in-depth analysis of algorithms.

Keywords: Sorting algorithms, comparative study, execution time, Bubble sort, Insertion sort, Quick sort, Merge sort, Heap sort, C programming language.

[1] Discente do curso de Bacharelado em Ciência da Computação, da Universidade Federal do Vale do São Francisco. E-mail: daniel.f@discente.univasf.edu.br

[2] Docente do curso de Bacharelado em Ciência da Computação, da Universidade Federal do Vale do São Francisco. E-mail: rodrigo.tamae@univasf.edu.br

Introdução

Algoritmos de ordenação são métodos computacionais que organizam elementos em uma determinada sequência de acordo com um critério predefinido. Esses algoritmos desempenham um papel crucial na otimização de processos computacionais, influenciando diretamente a eficiência e o desempenho de sistemas e algoritmos em uma ampla gama de contextos.

Para guiar esta pesquisa, foi elaborada a seguinte questão norteadora: será possível confirmar os resultados já documentados na literatura e observados na prática sobre o desempenho dos algoritmos de ordenação?

Dessa forma, esta pesquisa visa comparar via experimentação o desempenho dos algoritmos de ordenação Bubble sort, Insertion sort, Quicksort, Mergesort e Heapsort; levando em consideração a quantidade de comparações e trocas entre elementos, bem como o tempo de execução, a fim de confirmar os resultados já alcançados por trabalhos usados como referência.

Esta pesquisa empírica tem uma natureza básica com uma abordagem quantitativa, comparando o desempenho dos algoritmos de ordenação através de um método descritivo. Os algoritmos escolhidos foram considerados por sua relevância (embora o Bubble sort tenha o seu uso recomendado apenas para fins educacionais). Com exceção do Bubble sort, esses algoritmos foram investigados em (Sedgewick; Wayne, 2011).

Os objetivos específicos são: testar os algoritmos de ordenação escolhidos seguindo o método de pesquisa; comparar os resultados com trabalhos correlatos; e identificar as principais causas das diferenças ou semelhanças de desempenho.

Esta pesquisa, assim como os trabalhos correlatos, concentra-se na ordenação interna, sequencial (em contraposição à paralela), de dados do tipo inteiro. A ordenação externa é necessária quando a sequência de elementos não cabe na memória principal. Assume-se que na experimentação, ao falar de ordenação, está-se referindo a ordenação de inteiros em um arranjo unidimensional (ou vetor) na ordem crescente.

Com o computador cada vez mais integrado ao funcionamento da sociedade, é necessário lidar com uma quantidade de dados cada vez maior. O primeiro passo para

organizar esses dados muitas vezes é a pela ordenação (Sedgewick; Wayne, 2011, p. 243).

Este documento está organizado nas seguintes seções: Fundamentação teórica; Trabalhos correlatos; Experimento; Resultados e discussão; Considerações finais; e Referências.

1. Fundamentação teórica

Esta seção apresenta os elementos conceituais que são pertinentes para o experimento com os algoritmos de ordenação.

Nesse contexto, a ordenação é o processo de organizar uma sequência de elementos em alguma ordem lógica, como a numérica crescente ou alfabética. Por exemplo, os contatos telefônicos em uma lista de contatos podem estar ordenados alfabeticamente pelo nome de cada contato. Quando os computadores eram menos poderosos e os métodos de armazenamento de dados eram mais limitados, era estimado que até 30% dos ciclos de CPU eram gastos em ordenação. Se essa porcentagem é menor hoje, uma possível razão é que os algoritmos são relativamente eficientes, não que a ordenação diminuiu sua importância (Sedgewick; Wayne, 2011, p. 243).

A principal razão pela qual a ordenação é tão útil é que é muito mais fácil procurar um elemento em uma sequência ordenada, p. ex.: procurar um item no índice ordenado de um livro; remover duplicatas de uma longa lista, como uma lista de e-mails ou uma lista de sites; realizar cálculos estatísticos, como remover valores discrepantes, encontrar a mediana ou calcular percentis. A ordenação também surge como um subproblema crítico em muitas aplicações que parecem não ter nada a ver com a ordenação. Compressão de dados, computação gráfica, biologia computacional, gestão da cadeia de abastecimento, otimização combinatória, escolha social e votação são apenas alguns dos muitos exemplos (Sedgewick; Wayne, 2011, p. 336).

O Bubble sort é um dos algoritmos de ordenação mais simples. Ele percorre repetidamente a sequência, compara os elementos adjacentes e os troca se estiverem na ordem errada. Essas passagens pela sequência são repetidas até que nenhuma troca precise ser realizada durante uma passagem. Por conta disso, em média, a ordem de crescimento desse algoritmo, isto é, como o tempo de execução aumenta à medida que o tamanho da sequência aumenta, é n^2. Isso significa que quando a sequência de entrada (n) aumenta, o tempo de execução cresce aproximadamente quadraticamente. Por conta disso, é um algoritmo com uma aplicação muito limitada, sendo eficiente apenas com cargas de elementos pequeníssimas.

Listagem 1 - Bubble sort em Java

```
public class Bubble {
 public static void sort(Comparable[] a) {
  int n = a.length;
  for (int i = 0; i < n; i++) {
   int exchanges = 0;
   for (int j = n-1; j > i; j--) {
    if (less(a[j], a[j-1])) {
     exch(a, j, j-1);
     exchanges++;
    }
   }
   if (exchanges == 0) break;
  }
 } // See page 245 for less(), exch(), isSorted(), and main().
}
```

Fonte: Sedgewick; Wayne, 2022, https://algs4.cs.princeton.edu/21elementary/Bubble.java.html (Editado pelo autor, 2024).

O Insertion sort funciona abrindo espaço para inserir um item movendo os itens maiores uma posição para a direita, antes de inserir o item atual na posição vaga. Assim como o Bubble, o Insertion sort tem uma ordem de crescimento n² (em média), mas na prática esse último pode funcionar bem com certos tipos de sequências não aleatórias, mesmo que sejam um pouco maiores (Sedgewick; Wayne, 2011, p. 250).

Listagem 2 - Insertion sort em Java

```
public class Insertion {
 public static void sort(Comparable[] a) {
  // Sort a[] into increasing order.
  int N = a.length;
  for (int i = 1; i < N; i++) {
   // Insert a[i] among a[i-1], a[i-2], a[i-3]... ..
   for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
    exch(a, j, j-1);
   }
  }
 } // See page 245 for less(), exch(), isSorted(), and main().
}
```

Fonte: Sedgewick; Wayne, 2011, p. 251 (Editado pelo autor, 2024).

O Quicksort é provavelmente o algoritmo tradicional mais usado; sendo popular por não ser difícil de implementar, funciona em uma variedade de casos e é substancialmente mais rápido que os outros métodos típicos. Ele usa o método “dividir para conquistar”, particionando um arranjo em dois sub arranjos, e ordenando os sub arranjos independentemente. Ele é um complemento do Mergesort: para o Mergesort, o arranjo é dividido entre dois sub arranjos que são ordenados e depois combinados para fazer um único arranjo ordenado; para o Quicksort, o arranjo é ordenado de forma que, quando os dois sub arranjos são ordenados, o arranjo inteiro é ordenado. Ambos têm crescimento de n log n (em média) (Sedgewick; Wayne, 2011, p. 288).

Listagem 3 - Quicksort em Java

```
public class Quick {
 public static void sort(Comparable[] a) {
  StdRandom.shuffle(a);         // Eliminate dependence on input.
  sort(a, 0, a.length - 1);
 }
 private static void sort(Comparable[] a, int lo, int hi) {
  if (hi <= lo) return;
  int j = partition(a, lo, hi); // Partition (see page 291).
  sort(a, lo, j-1);             // Sort left part a[lo .. j-1].
  sort(a, j+1, hi);             // Sort right part a[j+1 .. hi].
 }
}
```

Fonte: Sedgewick; Wayne, 2011, p. 289 (Editado pelo autor, 2024).

O Mergesort também usa o método “dividir para conquistar”, dividindo um arranjo em duas metades, ordenando essas duas metades (recursivamente) e depois combinando os resultados. Uma das vantagens do Mergesort é que ele garante a ordenação de *n* elementos em tempo proporcional a *n* log *n*. Sua principal desvantagem é que ele usa espaço extra proporcional a *n* (Sedgewick; Wayne, 2011, p. 270).

Listagem 4 - Mergesort em Java

```
public class Merge {
 private static Comparable[] aux;      // auxiliary array for merges
 public static void sort(Comparable[] a) {
  aux = new Comparable[a.length];      // Allocate space just once.
  sort(a, 0, a.length - 1);
 }
 private static void sort(Comparable[] a, int lo, int hi) {
  // Sort a[lo..hi].
  if (hi <= lo) return;
  int mid = lo + (hi - lo)/2;
  sort(a, lo, mid);       // Sort left half.
  sort(a, mid+1, hi);     // Sort right half.
  merge(a, lo, mid, hi);  // Merge results
 }
 public static void merge(Comparable[] a, int lo, int mid, int hi) {
  // Merge a[lo..mid] with a[mid+1..hi].
  int i = lo, j = mid+1;
  for (int k = lo; k <= hi; k++) // Copy a[lo..hi] to aux[lo..hi].
   aux[k] = a[k];
  for (int k = lo; k <= hi; k++) // Merge back to a[lo..hi].
   if      (i > mid)              a[k] = aux[j++];
   else if (j > hi )              a[k] = aux[i++];
   else if (less(aux[j], aux[i])) a[k] = aux[j++];
   else                           a[k] = aux[i++];
 }
}
```

Fonte: Sedgewick; Wayne, 2011, p. 271 e 273 (Editado pelo autor, 2024).

O Heapsort envolve a construção de um *heap* a partir da sequência de entrada por meio de um processo chamado *heapify*, seguido pela remoção repetida do elemento raiz (que é o máximo ou mínimo dependendo se é um *max-heap* ou *min-heap*) e a restauração da propriedade do *heap* até que todos os elementos são ordenados. Ele pode ser interessante quando o espaço é muito limitado (por exemplo, em um sistema embarcado ou em um dispositivo móvel de baixo custo). Ele é popular porque pode ser implementado com apenas algumas dezenas de linhas (mesmo em

código de máquina), proporcionando ao mesmo tempo um desempenho ideal. No entanto, mesmo tendo em média a ordem de crescimento *n* log *n,* raramente é usado em aplicações típicas em sistemas modernos porque tem baixo desempenho de cache: itens do arranjo raramente são comparados com itens próximos, então o número de perdas de cache é muito maior do que para Quicksort, Mergesort e até mesmo Shellsort, onde a maioria das comparações são com itens próximos (Sedgewick; Wayne, 2011, p. 323-327).

Listagem 5 - Heapsort em Java

```
// ...
public static void sort(Comparable[] a) {
 int N = a.length;
 for (int k = N/2; k >= 1; k--)
  sink(a, k, N);
 while (N > 1) {
  exch(a, 1, N--);
  sink(a, 1, N);
 }
}
private void sink(int k) {
 while (2*k <= N) {
  int j = 2*k;
  if (j < N && less(j, j+1)) j++;
  if (!less(k, j)) break;
  exch(k, j);
  k = j;
 }
}
// ...
```

Fonte: Sedgewick; Wayne, 2011, p. 316 e 324 (Editado pelo autor, 2024).

2. Trabalhos correlatos

Nesta seção, são apresentados e discutidos os trabalhos correlatos escolhidos, e em quais aspectos são diferentes desta pesquisa. A escolha desses artigos como correlatos é baseada em sua relevância para o tema, bem como na variedade de abordagens que oferecem. Cada um dos artigos selecionados contribui para a compreensão dos algoritmos de ordenação de diferentes maneiras, desde a observação teórica até a implementação prática e a comparação de desempenho.

Em Yang et al. (2011), foram selecionados os algoritmos Bubble sort, Selection sort, Insertion sort, Quicksort e Mergesort para um experimento que comparou o tempo de execução dos algoritmos com sequências de entrada ordenadas de forma aleatória, crescente e decrescente. Levou-se em consideração a complexidade de tempo e espaço. Concluiu-se que os algoritmos se encaixam em uma dessas categorias de complexidade de tempo[3]: $O(n^2)$, sendo os algoritmos Bubble, Selection e Insertion sort; e $O(n\log n)$, sendo o Quicksort e Mergesort. Para uma aplicação, o algoritmo deve ser selecionado de acordo com os atributos da sequência de entrada. Se a carga for pequena (p. ex. menos de 2000 elementos), Insertion e Selection sort são aceitáveis. Se alguns padrões ou regras puderem ser encontrados na sequência (p. ex. elementos parcialmente ordenados), Insertion e Bubble sort são aceitáveis. Mas quando a carga é grande (p. ex. mais de 2000 elementos), Mergesort e Quicksort são essencialmente a escolha necessária.

Em Sepahyar et al. (2019), foram selecionados os algoritmos Insertion sort, Selection sort, Quicksort e Shellsort para um experimento que comparou o tempo de execução dos algoritmos com sequências de entrada ordenadas de forma aleatória e decrescente. Concluiu-se que a performance de cada algoritmo depende da sequência de entrada. O Shellsort obteve resultado satisfatório e mostrou que as melhores sequências de gap são Fibonacci e números primos com quantidade limitada, significando que essas reduzem a complexidade de tempo, isto é, são mais otimizadas. O Quicksort performou melhor com a mediana da sequência sendo o pivô. Os outros algoritmos obtiveram resultados regulares.

Em Souza et al. (2017), foram selecionados os algoritmos Bubble sort, Insertion sort, Selection sort, Quicksort, Mergesort e Shellsort para um experimento que comparou a quantidade de comparações e trocas entre elementos e o tempo de execução; tendo sequências de entrada que variaram em tamanho e ordem inicial (crescente, decrescente e aleatória). Concluiu-se que: o Bubble sort é insatisfatório em praticamente em qualquer caso; o Insertion sort é útil para sequências pequenas (entre 8 a 20 elementos); o Selection sort é uma melhoria do Insertion sort, tendo como

[3] A notação *Big O* (p. ex. $O(n^2)$ ou $O(n\log n)$) é uma das formas de descrever uma ordem de crescimento, comumente usada para indicar o desempenho de uma função quando seu argumento tende ao infinito (Sedgewick; Wayne, 2011, p. 206 e 207).

vantagem um número de trocas muito inferior ao número de comparações, sendo útil ao trabalhar com componentes que quanto mais se reescreve, mais se desgasta, que consequentemente perde sua eficiência (p. ex. memórias EEPROM e Flash); o Mergesort foi considerado mediano, sendo desvantajoso em estruturas pequenas ou já ordenadas, porém tendo melhores resultados com sequências ordenadas aleatoriamente; o Quicksort se apresenta ser excelente na maior parte do tempo, mesmo que esse prejudique nos recursos computacionais em algumas situações; o Shellsort teve o resultado mais satisfatório, principalmente em estruturas maiores e desordenadas, além disso, por ser uma melhoria do Selection sort, carrega as mesmas vantagens do seu predecessor.

Em Karunanithi (2014), a tese teve como objetivo principal fazer uma pesquisa sobre alguns algoritmos de ordenação com base na literatura e suas possíveis implementações, essencialmente discutindo e comparando as diferenças entre teoria e prática. Os algoritmos foram definidos e suas eficiências e outras propriedades discutidas e comparadas detalhadamente. Por meio da literatura foi encontrado diferentes “sabores” (ou variantes) que são melhorias dos algoritmos básicos. Também foi coletado dados de tempo de execução a fim de comparar suas variantes de implementação. As quantidades de interesse incluíram o número de operações de diferentes tipos executadas, o tempo de execução e o consumo de memória. Os algoritmos considerados foram: Bubble sort, Insertion sort, Selection sort, Quicksort, Mergesort, Heapsort, Radix sort, Bucket sort e Counting sort. Concluiu-se que os três fatores tempo de execução, número de trocas e memória usada, são críticos para a eficiência dos algoritmos. Foi entendido que os resultados do experimento mostraram que o comportamento teórico do desempenho é relevante para o desempenho prático de cada algoritmo. No entanto, houve uma pequena diferença no desempenho relativo dos algoritmos em alguns casos de teste. Além disso, o comportamento do algoritmo muda de acordo com a carga da sequência de entrada a ser ordenada e com o tipo de elementos. Os algoritmos O($n$logn) superam os O(n^2) em grandes sequências de dados. Mesmo assim, foi notado que cada algoritmo de ordenação tem suas próprias vantagens e desvantagens. Assim, a escolha de um algoritmo eficiente depende do tipo de problema.

Em Pandey (2008), a tese discutiu vários algoritmos de ordenação comparando uns com os outros. Também foi discutido sobre os algoritmos fundamentais e os algoritmos avançados[4], sendo estes modificações dos algoritmos fundamentais. Geralmente, o algoritmo de ordenação avançado é específico do problema e mostra seu melhor comportamento para o problema apropriado. Os algoritmos foram comparados com base em vários fatores importantes, como complexidade, memória, método, etc. Alguns algoritmos de ordenação são orientados a problemas e seu desempenho e eficiência dependem do problema; existem vantagens e desvantagens em qualquer algoritmo de ordenação. Após este estudo, concluiu-se que não existe um algoritmo de ordenação que funcione com base na prioridade, ou seja, um algoritmo que ordene dados específicos antes de outros dados gerais. Por conta disso, foi proposto um algoritmo de ordenação que funciona com base na prioridade.

Portanto, a inclusão desses artigos como correlatos proporciona uma base para compreender e contextualizar os resultados obtidos. Embora os detalhes específicos sobre a eficiência, complexidade e comportamento de cada algoritmo possam variar ligeiramente entre os estudos, o consenso geral é que esses algoritmos podem ser classificados em termos de desempenho, com alguns sendo mais eficientes do que outros em diferentes cenários e conjuntos de dados.

Por exemplo, algoritmos como Quicksort e Mergesort são frequentemente reconhecidos por sua eficiência em cenários médios e grandes, enquanto Bubble sort e Insertion sort tendem a ser menos eficientes em grandes conjuntos de dados devido à sua complexidade quadrática.

Por isso, em convergência com esses estudos, esta investigação se concentra em experimentar, seguindo o método presente na seção a seguir, os algoritmos de ordenação com a hipótese de confirmar os resultados já documentados na literatura e observados na prática.

[4] Exemplo desses são: Library sort, Cocktail sort, Comb sort, Gnome sort, Patience sort, Pigeonhole sort, Smooth sort, Strand sort, Topological sort, Tally sort, Postman sort (Pandey, 2008).

A Tabela 1 sumariza a proposta, método e resultado dos trabalhos correlatos.

Tabela 1 - Trabalhos correlatos sumarizados

Artigo	Proposta	Método	Resultado
Experimental Study on the Five Sort Algorithms (Yang et al., 2011)	Comparar Bubble sort, Selection sort, Insertion sort, Quicksort e Mergesort em complexidade de tempo e espaço.	Os algoritmos foram testados com sequências de entrada que variaram em tamanho e ordem inicial (aleatória, crescente e decrescente).	O Bubble sort, Selection sort e Insertion sort são algoritmos O(n^2), e portanto performam mal com sequências maiores. Enquanto isso, Mergesort e Quicksort são O($n$log$n$), e por isso escalam muito melhor com sequências maiores.
Comparing Four Important Sorting Algorithms Based on Their Time Complexity (Sepahyar et al., 2019)	Comparar Insertion sort, Selection sort, Quicksort e Shellsort em complexidade de tempo.	Os algoritmos foram testados com sequências de entrada que variaram em tamanho e ordem inicial (aleatória e decrescente).	O Shellsort e Quicksort obtiveram resultados satisfatórios enquanto que os outros obtiveram resultados regulares, ou seja, já esperados de algoritmos O(n^2).
Algoritmos de Ordenação: Um Estudo Comparativo (Souza et al., 2017)	Comparar Bubble sort, Insertion sort, Selection sort, Quicksort, Mergesort e Shellsort em quantidade de comparações e trocas entre elementos e o tempo de execução.	Os algoritmos foram testados com sequências de entrada que variaram em tamanho e ordem inicial (crescente, decrescente e aleatória).	Bubble sort, Selection sort, Insertion performaram mal e Quicksort, Mergesort e Shellsort performaram bem. O Shellsort teve o resultado mais satisfatório.
A Survey, Discussion and Comparison of Sorting Algorithms (Karunanithi, 2014)	A tese discutiu e comparou as diferenças entre teoria e prática sobre os algoritmos de ordenação.	Os algoritmos foram discutidos e comparados detalhadamente. Também foi coletado dados de tempo de execução a fim de comparar suas variantes de implementação.	Os fatores de tempo de execução, número de trocas e memória usada, são críticos para a eficiência dos algoritmos. Os algoritmos O($n$log$n$) superam os O(n^2) em grandes sequências de dados.
Study and Comparison of Various Sorting Algorithms (Pandey, 2008)	A tese discutiu vários algoritmos de ordenação comparando uns com os outros.	Foi discutido sobre os algoritmos fundamentais e os algoritmos avançados.	Foi entendido que existem vantagens e desvantagens em qualquer algoritmo de ordenação. Como concluiu-se que não existia um algoritmo de ordenação que funcionasse com base na prioridade, este foi proposto na tese.

Fonte: autor (2024).

3. Experimento

Neste experimento é utilizado a linguagem de programação C implementada pelo GNU Compiler Collection (GCC) (GNU, 2024). Para cada um dos algoritmos escolhidos, é testado vetores já ordenados nestas formas: inteiros em ordem crescente (ou seja, já ordenados); inteiros em ordem decrescente; e inteiros ordem aleatória (inteiros com valores inicializados usando a função *rand*). Para cada uma dessas ordenações iniciais são usadas as cargas de 2000, 16000 e 128000 inteiros. Com isso, 5 algoritmos sendo testados em vetores com 3 ordenações iniciais diferentes e com 3 cargas diferentes, o total de testes é 45, mas para uma maior precisão, cada teste individual é rodado 3 vezes, considerando a média aritmética como valor final. Será medido a quantidade de comparações, trocas e o tempo total de ordenação em milissegundos (usando a função *clock* da biblioteca padrão *time.h*).

Esta abordagem é semelhante à de Souza et al. (2017), que também utilizou cargas de dados de diferentes tamanhos e com pré-ordenações diferentes, o que permite ver como cada algoritmo se comporta com pequenas, médias e grandes quantidades de dados em diferentes situações em termos de comparações, trocas e tempo.

O computador usado como plataforma de experimentação tem os seguintes componentes: processador AMD Ryzen™ 5 5600G @ Clock base de 3.9GHz; memória RAM 16 GiB DDR4 3200MHz (KF3200C16D4/8GX 2x8 GiB em *dual channel*); sistema operacional Linux x86_64 (distribuição Ubuntu 22.04); compilador C GCC (Ubuntu 11.4.0-1ubuntu1~22.04). Os algoritmos usados são inspirados no que é possível encontrar no *website* Rosetta Code (Mol, 2020). A implementação dos testes está disponível em Franceschetti (2024).

4. Resultados e discussão

O resultado dos testes usando a carga de 2000 inteiros em vetores inicialmente ordenados em ordem crescente, decrescente e aleatória estão na Tabela 2.

Tabela 2 - Resultado dos testes com a carga 1

	2000 inteiros								
	Ordem crescente			Ordem decrescente			Ordem aleatória		
Algoritmo	**Comp.**	**Trocas**	**Tempo (ms)**	**Comp.**	**Trocas**	**Tempo (ms)**	**Comp.**	**Trocas**	**Tempo (ms)**
Bubble sort	1999	0	0.004	1999000	1999000	5.899	1997347	998451	4.377
Insertion sort	1999	1999	0.005	2000999	2000999	3.614	1000450	1000450	1.937
Quicksort	23951	0	0.032	23952	1000	0.033	31525	5405	0.113
Mergesort	10864	21728	0.043	11088	32816	0.056	19429	34771	0.131
Heapsort	28850	21301	0.206	23784	18709	0.218	26349	20152	0.245

Fonte: autor (2024).

Por meio da tabela é possível ver que o Bubble e Insertion sort mostram-se serem os melhores no caso em que o vetor já está ordenado (em ordem crescente). Por conta de serem algoritmos mais simples que os demais, esses obtiveram um número menor de comparações, trocas e tempo no caso com o vetor ordenado; porém, as limitações desses algoritmos ficam claras ao ver a quantidade consideravelmente maior de comparações, trocas e tempo nos outros casos. Na ordem decrescente, a média de comparações e trocas do Bubble e Insertion sort comparada com os demais é cerca de 100 vezes maior; e em tempo, cerca de 40 vezes mais tempo. Na ordem aleatória, a média de comparações e trocas do Bubble e Insertion sort comparada com os demais é cerca de 50 vezes maior; e em tempo, cerca de 20 vezes mais tempo. Por isso, com apenas 2000 inteiros o Bubble e Insertion sort já se apresentam como ineficientes.

Como também visto nos trabalhos correlatos, o Bubble e Insertion sort fazem parte de uma classe de algoritmos com uma ordem de crescimento de n^2, ou mais comumente $O(n^2)$, enquanto que os demais são $O(n \log n)$. Mesmo que os algoritmos Quicksort, Mergesort e Heapsort sejam da mesma classe, as implementações do Quicksort tem as ordenações que são predominantemente as mais rápidas e eficientes

dentro desses algoritmos tradicionais e de propósito geral. Desde a sua invenção, inúmeras implementações desse algoritmo sustentaram essa afirmação. Geralmente, a razão do Quicksort ser o mais rápido é que ele usa menos instruções no seu laço interno, fazendo com que seu tempo de execução em média seja aproximadamente O(*n*log*n*) vezes *c*, sendo esse *c* uma constante menor que as constantes correspondentes dos outros algoritmos de ordenação log-lineares (Sedgewick; Wayne, 2011, p. 343).

O resultado dos testes usando a carga de 16000 inteiros estão na Tabela 3.

Tabela 3 - Resultado dos testes com a carga 2

	16000 inteiros								
	Ordem crescente			**Ordem decrescente**			**Ordem aleatória**		
Algoritmo	**Comp.**	**Trocas**	**Tempo (ms)**	**Comp.**	**Trocas**	**Tempo (ms)**	**Comp.**	**Trocas**	**Tempo (ms)**
Bubble sort	15999	0	0.025	127992000	127992000	398.788	127976600	64127771	286.771
Insertion sort	15999	15999	0.038	128007999	128007999	244.011	64143770	64143770	122.442
Quicksort	239615	0	0.279	239616	8000	0.299	317107	54901	1.278
Mergesort	110912	221824	0.422	112704	334528	0.559	203271	370031	1.525
Heapsort	305648	220381	2.053	262052	197877	1.913	282865	209254	2.507

Fonte: autor (2024).

O mesmo que foi falado no resultado de teste com a primeira carga também pode ser falado para essa segunda, que acentua ainda mais a inaplicabilidade dos algoritmos O(n²) com cargas maiores que 2000, por exemplo. Apenas considerando as métricas utilizadas nos testes, o Quicksort continua a melhor escolha, tendo sempre o menor tempo e apenas perdendo para o Mergesort ou Heapsort em quantidade de comparações ou trocas em alguns casos.

O resultado dos testes usando a carga de 128000 inteiros estão na Tabela 4.

Tabela 4 - Resultado dos testes com a carga 3

	128000 inteiros								
	Ordem crescente			Ordem decrescente			Ordem aleatória		
Algoritmo	Comp.	Trocas	Tempo (ms)	Comp.	Trocas	Tempo (ms)	Comp.	Trocas	Tempo (ms)
Bubble sort	127999	0	0.196	8191936000	8191936000	25082.570	8191640704	4086586013	31244.614
Insertion sort	127999	127999	0.323	8192063999	8192063999	17402.164	4086714012	4086714012	8538.078
Quicksort	2300927	0	2.581	2300928	64000	2.731	3099035	527382	11.679
Mergesort	1079296	2158592	4.164	1093632	3252224	5.495	2010514	3667038	14.760
Heapsort	3021686	2151607	18.737	2663346	1963511	18.304	2838998	2058239	23.054

Fonte: autor (2024).

O mesmo que foi falado nos últimos resultados também é observado nesse. Por exemplo, enquanto que o Bubble sort demora cerca de **31 segundos** para ordenar 128000 inteiros em ordem aleatória, o Quicksort, cerca de **11 milissegundos**.

Considerações finais

Conforme o que já foi visto extensivamente na literatura e na prática, os algoritmos O(n^2) (p. ex. Bubble e Insertion sort) não são práticos com sequências cada vez maiores, já que existem alternativas O($n\log n$) (p. ex. Quicksort, Mergesort e Heapsort) muito mais eficientes. Dentro desses, o Quicksort, quando se trata de tempo de execução, é o melhor em geral. Entretanto, como também foi notado pelos trabalhos correlatos, a escolha de um algoritmo depende do tipo de problema, sendo o Quicksort na maior parte do tempo a melhor escolha dentre esses algoritmos tradicionais. Uma exceção notável é o uso do Insertion sort na ordenação de uma sequência de tamanho menor que 64 no Python (CPython, 2016). A experimentação realizada tem suas limitações: foram escolhidos apenas 5 algoritmos dentre vários; a ordenação de strings, números decimais, dados maiores ou estruturas de dados mais complexas não foram levadas em consideração; não houve teste em sequências parcialmente ordenadas; uso de memória não foi considerada.

Para trabalhos futuros, a recomendação seria: fazer uma análise[5] de algoritmos propriamente dita, levando em conta a complexidade computacional de forma mais profunda; testar conjuntos de dados diversificados, incluindo sequências parcialmente ordenadas e com outros tipos de dados; usar técnicas estatísticas que fornecem um resultado mais concreto; explorar algoritmos mais sofisticados, como os que não são baseados em comparação ou os que são paralelos e distribuídos; avaliar o impacto de diferentes linguagens de programação e ambientes de desenvolvimento; investigar os algoritmos de ordenação externa (com conjuntos de dados que não cabem na memória principal).

Referências

CPYTHON: *commit* de repositório arquivado. 2016. Disponível em: https://web.archive.org/web/20160128232837/https://hg.python.org/cpython/file/tip/Objects/listsort.txt. Acesso em: 11 maio 2024.

FRANCESCHETTI, Daniel de Lima. **Sorting Algorithms Comparison**. GitHub, 2024. Disponível em: https://github.com/danielsource/univasf-sorting-algorithms-comparison. Acesso em: 11 mai. 2024.

GNU Project. **GNU Compiler Collection**. 11.4.0. Free Software Foundation, 2024. Disponível em: https://gcc.gnu.org/. Acesso em: 11 mai. 2024.

MOL, Michael. **Rosetta Code**: Sorting Algorithms. 2020. Disponível em: https://rosettacode.org/wiki/Category:Sorting_Algorithms. Acesso em: 11 mai. 2024.

KARUNANITHI, Ashok Kumar. **A Survey, Discussion and Comparison of Sorting Algorithms**. Tese de mestrado, Department of Computing Science Umeå University, 2014.

PANDEY, Ramesh Chand. **Study and Comparison of Various Sorting Algorithms**. Tese de mestrado, Computer Science And Engineering Department Thapar University, 2008.

SEDGEWICK, Robert; WAYNE, Kevin. **Algorithms**. 4. ed. Westford, MA, USA, Addison-Wesley, 2011.

[5] Esta análise poderia classificar os algoritmos em: complexidade computacional, uso de memória, recursividade, estabilidade, pelo uso da comparação (algoritmos podem ou não examinar os dados usando o operador de comparação), etc.

SEPAHYAR, Soheil et al. Comparing Four Important Sorting Algorithms Based on Their Time Complexity. **Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence**, New York, NY, USA, 2019. p. 320-327.

SOUZA, Jackson E. G et al. Algoritmos de Ordenação: Um Estudo Comparativo. **Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA (ISSN 2526-7574)**, v. 1, n. 1, 2017. Disponível em: https://periodicos.ufersa.edu.br/ecop/article/view/7082. Acesso em: 7 mai 2024.

YANG, You et al. Experimental study on the five sort algorithms. **2011 Second International Conference on Mechanic Automation and Control Engineering**, Inner Mongolia, China, 2011. p. 1314-1317.