

# Uma nova abordagem ao algoritmo Mergesort: divisão inteligente e conquista

*A new approach to Mergesort algorithm: Divide smart and conquer*

Daniel Nolêto Maciel Luz<sup>1</sup> e João Victor Walcacer Giani<sup>2</sup>

<sup>1</sup> Universidade Federal do Tocantins, Ciência da Computação, Palmas, Tocantins

Data de recebimento do manuscrito: 15/10/2024

Data de aceitação do manuscrito: 17/10/2024

Data de publicação: 24/10/2024

**Resumo**—Algoritmos de ordenação baseados em comparação, como Mergesort, Quicksort e Heapsort, são conhecidos por sua complexidade assintótica eficiente, com tempo de execução de  $O(n \log n)$ . Contudo, a eficiência prática é igualmente importante, especialmente com o aumento da quantidade de dados. Este estudo propõe uma variação do Mergesort, onde o array é dividido em sub-arrays que podem estar em ordem crescente ou decrescente, ajustando a função de mesclagem (*Merge*) de acordo. Como resultado, foram desenvolvidos novos algoritmos que apresentam melhorias significativas. A complexidade do pior caso permanece em  $O(n \log n)$ , enquanto no melhor caso é  $O(n)$ . Na prática, a abordagem proposta demonstrou melhorias notáveis em distribuições Gaussiana e Uniforme, superando o Mergesort tradicional com até 29% de aumento na eficiência.

**Palavras-chave**— Mergesort Algoritmos de Ordenação, Divisão e Conquista, Análise de complexidade.

**Abstract**— Comparison-based algorithms, such as Mergesort, Quicksort, and Heapsort, are considered optimal in terms of asymptotic complexity, with a runtime of  $O(n \log n)$ . However, in practice, efficiency is also a crucial factor, especially as data volumes increase. This study introduces a variation of Mergesort, where the array is not divided unconditionally. Instead, it is separated into sub-arrays that are in increasing and decreasing order, and the merge function (*Merge*) is adjusted accordingly. This results in three new algorithms, with the third being the most effective. In this algorithm, the array is divided into alternating sub-arrays of increasing and decreasing order, and the elements do not need to be consecutive. The worst-case complexity remains  $O(n \log n)$ , but in the best case, it is  $O(n)$ . In practice, the proposed algorithm demonstrated significant improvements on Gaussian and Uniform distributions, outperforming traditional Mergesort by up to 29% in efficiency.

**Keywords**— Mergesort, Sorting algorithms, Divide and conquer, Complexity analysis.

## I. INTRODUÇÕES

A ordenação e a busca são problemas clássicos na ciência da computação, sendo essenciais para a eficiência de algoritmos em diversas áreas de engenharia. O artigo em análise explora esses problemas, destacando a limitação teórica que impede algoritmos de ordenação baseados em comparação de alcançarem uma complexidade melhor que  $O(n \log n)$ . Além disso, discute a importância das constantes nos termos assintóticos para o desempenho prático desses algoritmos, ressaltando que o interesse por soluções eficientes aumentou devido à desaceleração na evolução da velocidade dos processadores em relação ao rápido crescimento do volume de dados.

O estudo foca em dois algoritmos de ordenação por comparação altamente eficientes: o *Mergesort* e o *Quicksort*. O *Mergesort*, introduzido por Jon von Neumann em 1945, adota a estratégia de divisão e conquista, mantendo uma complexidade de  $O(n \log n)$  tanto no melhor quanto no pior e no caso médio, mas requer  $O(n)$  de memória adicional. Por outro lado, o *Quicksort*, desenvolvido por Tony Hoare entre 1959 e 1960, também segue a estratégia de divisão e conquista, com complexidade de  $O(n^2)$  no pior caso e  $O(n \log n)$  no melhor e no caso médio, consumindo apenas  $O(1)$  de memória adicional.

Os experimentos realizados com dados gerados aleatoriamente, seguindo distribuições Uniforme e Gaussiana, e com dados reais, indicaram que o *Quicksort* pode ser mais eficiente em muitos casos práticos devido ao seu baixo consumo de memória e à performance média superior, especialmente quando implementado com boas estratégias de particionamento.

## II. ABORDAGENS

A principal ideia por trás do algoritmo descrito no artigo é dividir um array dado em  $n$  sub-arrays ordenados não interseccionados e combiná-los em um único array ordenado. A eficiência desse algoritmo depende de dois fatores fundamentais:

- Como a divisão será feita?
- Como os sub-arrays serão combinados?

O artigo propõe duas abordagens para a divisão, sendo que o algoritmo principal é baseado na segunda abordagem, considerada a mais eficiente. Vamos detalhar essas abordagens utilizando exemplos simples.

### a. Primeira abordagem

: Um array  $A$  com  $n$  elementos pode ser dividido em sub-arrays ascendentes de elementos consecutivos em tempo  $O(n)$ , percorrendo o array do início ao fim. Esses sub-arrays não se sobrepõem (cada termo de  $A$  pertence a exatamente um sub-array), e o número de sub-arrays  $k$  satisfaz  $1 \leq k \leq n$ .

Por exemplo, dado um array  $A$  com 12 elementos: 3, 10, 7, 4, 6, 1, 5, 8, 9, 2, 12, 11, os sub-arrays ascendentes são mostrados na Figura 1. Nesse exemplo,  $n = 12$  e  $k = 6$ . O objetivo é combinar esses sub-arrays ascendentes em um único array ordenado utilizando a função `Merge`. No melhor caso,  $k = 1$ , quando o array  $A$  já está em ordem crescente. No pior caso,  $k = n$ , quando  $A$  está em ordem inversa.

### Algorithm 1 Sort

---

```

1: Input: Unsorted array  $A[0 \dots n-1]$ 
2: Output: Sorted array  $A[0 \dots n-1]$ 
3: procedure Sort( $A, n$ )
4:    $B[0] \leftarrow -1$ 
5:    $t \leftarrow 0$ 
6:    $k \leftarrow 0$ 
7:   for  $i \leftarrow 0$  to  $n-2$  do
8:     if  $A[i+1] < A[i]$  then
9:        $t \leftarrow t+1$ 
10:       $B[t] \leftarrow i$ 
11:    end if
12:  end for
13:   $k \leftarrow t+1$ 
14:   $B[t+1] \leftarrow n-1$ 
15:  while  $k > 1$  do
16:     $t \leftarrow 0$ 
17:    for  $i \leftarrow 0$  to  $k-2$  do
18:      Merge_M( $A, i, i+1, i+2, B$ )
19:       $t \leftarrow t+1$ 
20:       $B[t] \leftarrow B[i+2]$ 
21:    end for
22:     $B[t+1] \leftarrow n-1$ 
23:     $k \leftarrow \lceil k/2 \rceil$ 
24:  end while
25:  return  $A$ 
26: procedure Merge_M( $A, p, q, r, B$ )
27:   $n1 \leftarrow B[q] - B[p]$ 
28:   $n2 \leftarrow B[r] - B[q]$ 
29:  for  $i \leftarrow 0$  to  $n1-1$  do
30:     $L[i] \leftarrow A[B[p] + i + 1]$ 
31:  end for
32:  for  $i \leftarrow 0$  to  $n2-1$  do
33:     $R[i] \leftarrow A[B[q] + i + 1]$ 
34:  end for
35:   $L[n1] \leftarrow +\infty$ 
36:   $R[n2] \leftarrow +\infty$ 
37:   $i \leftarrow 0$ 
38:   $j \leftarrow 0$ 
39:  for  $k \leftarrow 0$  to  $n1+n2-1$  do
40:    if  $L[i] \leq R[j]$  then
41:       $A[B[p] + k + 1] \leftarrow L[i]$ 
42:       $i \leftarrow i+1$ 
43:    else
44:       $A[B[p] + k + 1] \leftarrow R[j]$ 
45:       $j \leftarrow j+1$ 
46:    end if
47:  end for
48:  return

```

---

### b. Segunda abordagem

É possível combinar dois sub-arrays, um ascendente e o outro descendente, ou ambos descendentes, em um sub-array ascendente, modificando ligeiramente a função clássica `Merge`. Para isso, precisamos saber se o sub-array é descendente ou ascendente. Durante a mesclagem, podemos percorrer o sub-array descendente do final para o início.

Basta comparar os dois primeiros elementos uma vez para determinar a ordem do sub-array. Portanto, conseguimos escrever o array  $A$  como uma combinação ordenada (ascen-

dente ou descendente) de elementos consecutivos. Esses sub-arrays não se sobrepõem, e o número de sub-arrays  $k$  é mantido. Na divisão proposta, o array  $A$  seria dividido como mostrado na Figura 2.

O número  $k$  ainda é 6, mas essa abordagem oferece vantagens em relação à anterior: exceto pelo último, cada sub-array possui pelo menos dois elementos. Se  $n$  for um número par e o último sub-array tiver um elemento, então o número de termos em pelo menos um dos sub-arrays anteriores deve ser no mínimo 3. Isso implica que, nesse caso,  $k$  será no máximo igual a  $n/2$ . Se  $n$  for ímpar e o último sub-array tiver um elemento, então no pior caso temos  $k = \left(\frac{n-1}{2}\right) + 1 = \frac{n+1}{2}$ . Em outras palavras, na segunda abordagem de divisão, o valor máximo de  $k$  será  $\lceil n/2 \rceil$ . O pseudo-código para a ordenação de acordo com essa segunda abordagem é apresentado no Algoritmo 2.

---

**Algorithm 2** Sorting Algorithm according to the second approach

---

```

1: Input: Unsorted array  $A[0..n-1]$ 
2: Output: Sorted array  $A[0..n-1]$ 
3: procedure Sort( $A, n$ )
4:  $B[0] \leftarrow -1$ 
5:  $t \leftarrow 0$ 
6:  $cont \leftarrow -1$ 
7: if  $A[0] \leq A[1]$  then
8:    $cont \leftarrow 1$ 
9: end if
10: for  $i \leftarrow 0$  to  $n-3$  do
11:   if  $cont \cdot A[i+1] < cont \cdot A[i]$  then
12:      $t \leftarrow t+1$ 
13:      $B[t] \leftarrow i$ 
14:      $cont \leftarrow -1$ 
15:     if  $A[i+1] \leq A[i+2]$  then
16:        $cont \leftarrow 1$ 
17:     end if
18:   end if
19: end for
20: if  $cont \cdot A[n-1] < cont \cdot A[n-2]$  then
21:    $t \leftarrow t+1$ 
22:    $B[t] \leftarrow n-2$ 
23: end if
24:  $k \leftarrow t+1$ 
25:  $B[t+1] \leftarrow n-1$ 
26: while  $k > 1$  do
27:    $t \leftarrow 0$ 
28:   for  $i \leftarrow 0$  to  $k-2$  do
29:     Merge_M2( $A, i, i+1, i+2, B$ )
30:      $t \leftarrow t+1$ 
31:      $B[t] \leftarrow B[i+2]$ 
32:      $i \leftarrow i+1$ 
33:   end for
34:    $B[t+1] \leftarrow n-1$ 
35:    $k \leftarrow \lceil k/2 \rceil$ 
36: end while
37: return  $A$ 

```

---



---

**Algorithm 3** Continuação Algorithm 2

---

```

1: procedure Merge_M2( $A, p, q, r, B$ )
2:  $n1 \leftarrow B[q] - B[p]$ 
3:  $n2 \leftarrow B[r] - B[q]$ 
4:  $cc \leftarrow A[B[q]] - A[B[p] + 1]$ 
5:  $ct \leftarrow A[B[r]] - A[B[q] + 1]$ 
6:  $cont_L \leftarrow -1$ 
7:  $cont_R \leftarrow -1$ 
8: if  $cc > 0$  then
9:    $cont_L \leftarrow 1$ 
10: end if
11: if  $ct > 0$  then
12:    $cont_R \leftarrow 1$ 
13: end if
14: if  $cont_L = 1$  then
15:   for  $i \leftarrow 0$  to  $n1-1$  do
16:      $L[i] \leftarrow A[B[p] + i + 1]$ 
17:   end for
18: else
19:   for  $i \leftarrow 0$  to  $n1-1$  do
20:      $L[i] \leftarrow A[B[q] - i]$ 
21:   end for
22: end if
23: if  $cont_R = 1$  then
24:   for  $i \leftarrow 0$  to  $n2-1$  do
25:      $R[i] \leftarrow A[B[q] + i + 1]$ 
26:   end for
27: else
28:   for  $i \leftarrow 0$  to  $n2-1$  do
29:      $R[i] \leftarrow A[B[r] - i]$ 
30:   end for
31: end if
32:  $L[n1] \leftarrow +\infty$ 
33:  $R[n2] \leftarrow +\infty$ 
34:  $i \leftarrow 0$ 
35:  $j \leftarrow 0$ 
36: for  $k \leftarrow 0$  to  $n1+n2-1$  do
37:   if  $L[i] \leq R[j]$  then
38:      $A[B[p] + k + 1] \leftarrow L[i]$ 
39:      $i \leftarrow i+1$ 
40:   else
41:      $A[B[p] + k + 1] \leftarrow R[j]$ 
42:      $j \leftarrow j+1$ 
43:   end if
44: end for
45: return  $A$ 

```

---

### III. RESULTADOS

No artigo analisado, foram testados dois algoritmos propostos em comparação com algoritmos de ordenação comumente preferidos, em três cenários: dados gerados aleatoriamente (distribuições Gaussiana e Uniforme), dados reais e casos especiais. Os experimentos foram realizados em um computador com sistema operacional Linux (Ubuntu 16.04), equipado com um processador Intel i5 de 1,60 GHz e 8 GB de RAM DDR4. O código foi desenvolvido na linguagem C, utilizando o compilador g++ 5.4.0, e os conjuntos de dados estão disponíveis publicamente para reprodução. Em nosso caso, os experimentos foram realizados em um notebook *Nitro 5* da *Acer* com as seguintes especificações: processador *Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz*, 24GB de memória RAM, armazenamento em SSD de 480GB e uma placa de vídeo dedicada *GTX 1650* com 4GB de VRAM. E a linguagem Python, foi a utilizada para implementação.

Durante os testes, o pivô no algoritmo Quicksort foi escolhido como o termo mais próximo da média aritmética do array, garantindo que o algoritmo funcione em tempo  $O(n \log n)$  para arrays com valores distintos. Cada algoritmo foi executado em três arrays de mesmo tamanho, e a média dos tempos de execução foi utilizada para comparação.

#### a. Experimentos com Conjuntos de Dados com Distribuição Gaussiana e Uniforme

Os tempos de execução dos algoritmos foram utilizados para criar gráficos comparativos. Os experimentos demonstraram que os Algoritmos 1 e 2 superaram o Quicksort e o Mergesort em todos os testes.

#### b. Conjuntos de Dados Reais

Os resultados com dados reais mostraram que os Algoritmos 1 e 2 tiveram desempenho superior ao Mergesort e ao Quicksort. Em arrays de pequeno tamanho, como no conjunto de dados RCL, ambos se mostraram mais eficientes que outros algoritmos testados.

#### c. Casos Especiais

Os resultados para conjuntos de dados em ordem reversa indicaram que os Algoritmos 1 e 2 tiveram desempenho superior, com o Algoritmo 1 sendo particularmente eficiente ao identificar subarrays. Para dados constantes, os Algoritmos 1 e 2 foram mais rápidos na identificação de subarrays.

#### d. Elementos Repetidos nos Arrays

Nos testes com arrays que continham elementos repetidos, os Algoritmos 1 e 2 mostraram-se robustos, enquanto o Quicksort teve desempenho prejudicado. Ambos os algoritmos conseguiram manejar eficientemente a situação sem afetar o tempo de execução.

#### e. Combinações de Dois Subarrays Crescentes e Decrescentes

Os resultados para arrays compostos por combinações de subarrays mostraram que os Algoritmos 1 e 2 superaram os

demais, destacando sua eficácia em cenários de mistura de sequências.

### IV. TABELAS

Table 4  
Running times of the algorithms in seconds for arrays with various sizes (Gaussian Distribution).

Array size	Quicksort	Mergesort	Heapsort	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.000159	0.000202	0.000167	0.000182	0.000169	0.000148
10 000	0.002130	0.002559	0.002330	0.002351	0.002176	0.001972
20 000	0.004619	0.005450	0.004929	0.004876	0.004684	0.004120
50 000	0.012971	0.014587	0.014135	0.013609	0.012655	0.011804
100 000	0.027891	0.030714	0.031775	0.029373	0.027278	0.025555
250 000	0.071243	0.080616	0.085509	0.074543	0.071764	0.066011
1 000 000	0.322790	0.345641	0.401155	0.327630	0.307932	0.285997
4 000 000	1.396416	1.526262	2.298623	1.442417	1.363992	1.258511

Figura 1: Tabela 4

Table 5  
Running times of the algorithms in seconds for arrays with various sizes (Uniform Distribution).

Array size	Quicksort	Mergesort	Heapsort	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.000151	0.000196	0.000170	0.000179	0.000168	0.000149
10 000	0.001945	0.002617	0.002268	0.002229	0.002147	0.001879
20 000	0.004167	0.005144	0.005029	0.005011	0.004691	0.004303
50 000	0.011304	0.014268	0.013956	0.013613	0.012338	0.011693
100 000	0.024138	0.030326	0.029991	0.028586	0.026666	0.024379
250 000	0.063907	0.080271	0.082610	0.075881	0.071619	0.064575
1 000 000	0.284771	0.373335	0.399644	0.324742	0.308354	0.289816
4 000 000	1.339597	1.517415	2.207029	1.422698	1.342912	1.248776

Figura 2: Tabela 5

Table 6  
Running times of the algorithms in seconds for arrays with various sizes (Real data).

Data set	Column name	Quicksort	Mergesort	Heapsort	Algorithm 1	Algorithm 2	Algorithm 3
RCL	Easting_coordinate	0.001178	0.001489	0.001870	0.001521	0.001286	0.001168
	Northing_coordinate	0.227801	0.001230	0.000131	0.000034	0.000034	0.000133
	Registered_area	0.224924	0.001228	0.000129	0.000033	0.000034	0.000124
TC	LicenceNumber	0.002150	0.002337	0.002575	0.001517	0.001450	0.001443
	LicenceNumber	0.008874	0.004365	0.004721	0.002796	0.002621	0.002685
	LicenceNumber	0.006945	0.005708	0.006486	0.003565	0.003323	0.003583
NSPL	Longitude	0.135403	0.153852	0.166476	0.146932	0.136443	0.125604
CCDD	V1	0.236077	0.190593	0.208881	0.182727	0.166833	0.153685
	V2	0.236467	0.190576	0.209694	0.180450	0.167884	0.155470
	V8	0.236010	0.185069	0.214711	0.180797	0.168773	0.154672

Figura 3: Tabela 6

Table 7  
Running times of the algorithms in seconds for arrays with various sizes (Reverse Ordered Data Set).

Array size	Quicksort	Mergesort	Heapsort	Insertionsort	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.000099	0.00014	0.000141	0.001366	0.000164	0.000008	0.000015
10 000	0.00125	0.00170	0.001762	0.143933	0.001990	0.000032	0.000132
20 000	0.00259	0.00364	0.003761	0.567991	0.004224	0.000060	0.000250
50 000	0.00717	0.00971	0.010222	3.523521	0.011257	0.000131	0.000678
100 000	0.01511	0.02098	0.021520	14.073632	0.024502	0.000385	0.001335
250 000	0.04038	0.05384	0.058700	102.235113	0.062855	0.000606	0.003110
1 000 000	0.18158	0.23232	0.260769	1894.617733	0.267523	0.002455	0.012223
4 000 000	0.78468	0.99601	1.174221	30 321.395592	1.128898	0.009684	0.049067

Figura 4: Tabela 7

Table 8  
Running times of the algorithms in seconds for arrays with various sizes (Data Set of Constant Arrays).

Array size	Quicksort	Mergesort	Heapsort	Insertionsort	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.156881	0.000220	0.000014	0.000004	0.000008	0.000008	0.000016
10 000	0.365643	0.001799	0.000129	0.000034	0.000030	0.000032	0.000121
20 000	1.76159	0.003578	0.000263	0.000067	0.000054	0.000057	0.000241
50 000	13.3019	0.009879	0.000658	0.000168	0.000127	0.000136	0.000610
100 000	57.0385	0.020196	0.001372	0.000324	0.000242	0.000252	0.001179
250 000	372.040	0.052789	0.003620	0.000836	0.000565	0.000694	0.002974
1 000 000	6082.29	0.235406	0.013359	0.003300	0.002305	0.002457	0.012224
4 000 000	97 844.3	0.998917	0.054902	0.013112	0.008979	0.009611	0.048494

Figura 5: Tabela 8

Table 9  
Running times of the algorithms in seconds for arrays with various sizes (Sorted Data Set).

Array size	Quicksort	Mergesort	Heapsort	Insertionsort	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.00008	0.00014	0.000337	0.000004	0.000008	0.000008	0.000016
10 000	0.00115	0.00171	0.002116	0.000034	0.000031	0.000032	0.000121
20 000	0.00249	0.00355	0.003947	0.000065	0.000054	0.000058	0.000244
50 000	0.00675	0.00954	0.022962	0.000163	0.000126	0.000133	0.000598
100 000	0.01484	0.02031	0.026438	0.000326	0.000267	0.000287	0.001327
250 000	0.03925	0.05330	0.060521	0.000917	0.000667	0.000637	0.003747
1 000 000	0.17506	0.22959	0.269955	0.003300	0.002221	0.002355	0.012081
4 000 000	0.76437	1.00362	1.207354	0.013496	0.009012	0.009725	0.047825

Figura 6: Tabela 9

**Table 10**  
Running times of the algorithms in seconds for arrays with various sizes (Repeated elements in the array).

Array size	Quicksort	Mergesort	Heapsort	Insertionsort	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.09277	0.00016	0.000135	0.000610	0.000150	0.000146	0.000131
10 000	0.09864	0.00193	0.001739	0.058585	0.001862	0.001705	0.001557
20 000	0.40688	0.00396	0.003521	0.567991	0.004066	0.003780	0.003327
50 000	3.23725	0.01051	0.020001	1.434649	0.010541	0.010552	0.009457
100 000	14.3066	0.02203	0.026438	6.403761	0.021735	0.020772	0.019998
250 000	95.1555	0.05728	0.053421	45.995262	0.059371	0.054489	0.052760
1 000 000	1571.32	0.24689	0.236507	863.671438	0.246334	0.233350	0.216031
4 000 000	25 341.3	1.06513	1.031672	18 019.122023	1.203865	1.018913	0.939923

**Figura 7:** Tabela 10

**Table 11**  
Running times of the algorithms in seconds for arrays with various sizes (Combinations of ascending and descending two sub-arrays).

Array size	Quicksort	Mergesort	Heapsort	Insertionsort	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.000121	0.000157	0.000156	0.000705	0.000145	0.000128	0.000022
10 000	0.001463	0.001834	0.001957	0.073384	0.001799	0.001462	0.000178
20 000	0.003203	0.003868	0.004230	0.282078	0.003767	0.003195	0.000353
50 000	0.008579	0.010401	0.011437	1.750863	0.010364	0.009289	0.000889
100 000	0.018431	0.022292	0.023553	8.056585	0.022234	0.019888	0.001872
250 000	0.049227	0.057188	0.066595	58.237071	0.057170	0.052573	0.004510
1 000 000	0.210608	0.246286	0.293901	943.259505	0.245501	0.216230	0.018205
4 000 000	0.922224	1.055072	1.336828	19 343.56241	1.038832	0.963400	0.072020

**Figura 8:** Tabela 11

## V. CONCLUSÃO

Com o aumento da quantidade de dados provenientes de diversas fontes, a importância dos algoritmos de ordenação cresceu significativamente. Este estudo analisou algoritmos de ordenação, destacando que a operação de divisão adotada nos métodos atuais apresenta abordagens mais eficientes em comparação aos algoritmos clássicos.

Os resultados experimentais mostraram que os algoritmos propostos se destacaram em velocidade de execução, superando métodos tradicionais como Mergesort, Heapsort e Quicksort. No entanto, a estabilidade de alguns algoritmos de ordenação continua sendo um desafio, especialmente em condições de divisão complexas.

Futuros estudos investigarão quais arrays podem ser representados como combinações de subarrays, um crescente e outro decrescente, bem como o número médio de subarrays alternadamente crescentes e decrescentes em um array gerado aleatoriamente.

## REFERÊNCIAS

1. Sahin Emrah Amrahov, "A new approach to Mergesort algorithm: Divide smart and conquer, 2024" <https://www.sciencedirect.com/science/article/pii/S0167739X24001225>, (Acessado em 20/10/2024).
2. M. Axtmann, S. Witt, D. Ferizovic, P. Sanders, "Engineering in-place (shared memory) sorting algorithms," *ACM Trans. Parallel Comput.* 9 (1) (2022) 1–62.
3. Z. Abo-Hammour, O. AbuArqub, S. Momani, N. Shawagfeh, et al., "Optimization solution of Troesch's and Bratu's problems of ordinary type using novel continuous genetic algorithm," *Discrete Dyn. Nat. Soc.* 2014 (2014).
4. Y. Ar, Ş. Emrah Amrahov, N.A. Gasilov, S. Yigit-Sert, "A new curve fitting based rating prediction algorithm for recommender systems," *Kybernetika* 58 (3) (2022) 440–455.
5. O. A. Arqub, Z. Abo-Hammour, "Numerical solution of systems of second-order boundary value problems using continuous genetic algorithm," *Inf. Sci.* 279 (2014) 396–415.

6. N. Kartli, E. Bostanci, M.S. Guzel, "A new algorithm for optimal solution of fixed charge transportation problem," *Kybernetika* 59 (1) (2023) 45–63.
7. S. Demir, B. Tugrul, "Privacy-preserving trend surface analysis on partitioned data," *Knowl.-Based Syst.* 144 (2018) 16–20.
8. N.A. Gasilov, "On exact solutions of a class of interval boundary value problems," *Kybernetika* 58 (3) (2022) 376–399.
9. H.B. Yıldırım, K. Kullu, Ş. Emrah Amrahov, "A graph model and a three-stage algorithm to aid the physically disabled with navigation," *Universal Access Inf. Soc.* (2023) 1–11.
10. S. Sharma, N. Khodadadi, A.K. Saha, F.S. Gharehchogh, S. Mirjalili, "Non-dominated sorting advanced butterfly optimization algorithm for multi-objective problems," *J. Bionic Eng.* 20 (2) (2023) 819–843.
11. I. Saidani, A. Ouni, M. Ahasanuzzaman, S. Hassan, M.W. Mkaouer, A.E. Hassan, "Tracking bad updates in mobile apps: A search-based approach," *Empir. Softw. Eng.* 27 (4) (2022) 81.
12. H. Ferrada, "A sorting algorithm based on ordered block insertions," *J. Comput. Sci.* 64 (2022) 101866.
13. H. Zhang, B. Meng, Y. Liang, "Sort race," *Softw.-Pract. Exp.* 52 (8) (2022) 1867–1878.
14. P. Ganapathi, R. Chowdhury, "Parallel divide-and-conquer algorithms for bubble sort, selection sort and insertion sort," *Comput. J.* 65 (10) (2022) 2709–2719.
15. A. Mubarak, S. Iqbal, T. Naeem, S. Hussain, "2mm: A new technique for sorting data," *Theoret. Comput. Sci.* 910 (2022) 68–90.
16. F. Stober, A. Weiß, "On the average case of Merge Insertion," *Theory Comput. Syst.* 64 (7) (2020) 1197–1224.
17. F. Grabowski, D. Strzalka, "Dynamic behavior of simple insertion sort algorithm," *Fund. Inform.* 72 (1–3) (2006) 155–165.
18. Ş.E. Amrahov, A.S. Mohammed, F.V. Çelebi, "New and improved search algorithms and precise analysis of their average-case complexity," *Future Gener. Comput. Syst.* 95 (2019) 743–753.
19. A.S. Mohammed, Ş.E. Amrahov, F.V. Çelebi, "Interpolated binary search: An efficient hybrid search algorithm on ordered datasets," *Eng. Sci. Technol., Int. J.* 24 (2021) 1072–1079.
20. Z. Marszałek, "The analysis of energy performance in use parallel merge sort algorithms," *Inf. Technol. Control* 48 (3) (2019) 487–498.
21. D.E. Knuth, "The Art of Computer Programming," vol. 3, Pearson Education, 1997.
22. C.A. Hoare, "Quicksort," *Comput. J.* 5 (1) (1962) 10–16.

23. S. Gueron, V. Krasnov, "Fast quicksort implementation using AVX instructions," *Comput. J.* 59 (1) (2016) 83–90.
24. S. Edelkamp, A. Weiß, S. Wild, "QuickXsort: A fast sorting scheme in theory and practice," *Algorithmica* 82 (3) (2020) 509–588.
25. J. Wassenberg, M. Blacher, J. Giesen, P. Sanders, "Vectorized and performance portable quicksort," *Softw.-Pract. Exp.* 52 (12) (2022) 2684–2699.
26. M. Aumüller, M. Dietzfelbinger, "Optimal partitioning for dual-pivot quicksort," *ACM Trans. Algorithms (TALG)* 12 (2) (2015) 1–36.
27. M. Aumüller, M. Dietzfelbinger, P. Klaue, "How good is multi-pivot quicksort?" *ACM Trans. Algorithms (TALG)* 13 (1) (2016) 1–47.
28. J.W.J. Williams, "Algorithm 232: heapsort," *Commun. ACM* 7 (6) (1964) 347–348.
29. J.W.J. Williams, "Algorithm 232: heapsort," *Commun. ACM* 7 (6) (1964) 347–348.