Uma nova abordagem ao algoritmo Mergesort: divisão inteligente e conquista

A new approach to Mergesort algorithm: Divide smart and conquer

Daniel Nolêto Maciel Luz[1] e João Victor Walcacer Giani[2]

[1] *Universidade Federal do Tocantins, Ciência da Computação, Palmas, Tocantins*

Data de recebimento do manuscrito: dd/mm/aaaa
Data de aceitação do manuscrito: dd/mm/aaaa
Data de publicação: dd/mm/aaaa

Resumo— Algoritmos baseados em comparação, como Mergesort, Quicksort e Heapsort, são considerados ótimos em termos de complexidade assintótica, com tempo de execução de $O(n \log n)$. No entanto, na prática, a eficiência também é um fator crucial, especialmente com o aumento da quantidade de dados. Este estudo introduz uma variação do Mergesort, onde o array não é dividido de forma incondicional. Ao invés disso, ele é separado em sub-arrays que estão em ordem crescente e decrescente, e a função de mesclagem (*Merge*) é ajustada. Isso resulta em três novos algoritmos, dos quais o terceiro é o mais eficaz. Neste algoritmo, o array é dividido em sub-arrays alternados de ordem crescente e decrescente, e os elementos não precisam ser consecutivos. A complexidade do pior caso continua sendo $O(n \log n)$, mas no melhor caso é $O(n)$. Na prática, o algoritmo proposto apresentou melhorias significativas em distribuições Gaussiana e Uniforme, superando o Mergesort tradicional com até 29% de aumento de eficiência.

Palavras-chave— Primeira palavra ou frase-chave, segunda palavra ou frase-chave, terceira palavra ou frase-chave. (Coloque entre três e seis palavras-chave ou frases separadas por vírgula, que representam o tema do seu trabalho)

***Abstract*—**

***Keywords*—** *First word or key phrase, second word or key phrase, third word or key phrase. (Place between three and six key words or phrases separated by a comma, which represent the theme of your work)*

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, o artigo 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 nos algoritmos de ordenação por comparação mais eficientes, como o *Mergesort*, *Quicksort* e *Heapsort*. 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.

O artigo também apresenta três algoritmos alternativos ao *Mergesort* clássico. Dentre eles, o Algoritmo 3 se destaca por introduzir uma nova abordagem, que divide o array em subarrays alternadamente crescentes e decrescentes, posteriormente combinados de forma eficiente através do método de torneio. Em termos de complexidade, os três algoritmos mantêm $O(n \log n)$ no pior caso e $O(n)$ no melhor caso.

Os experimentos realizados com dados gerados aleatoriamente, seguindo distribuições Uniforme e Gaussiana, e com dados reais, indicaram que o Algoritmo 3 oferece, em média, uma melhoria de 23

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 funda-

Dados de contato: Name A. Surname, name.surname@email.com

mentais:

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

O artigo propõe três abordagens para a divisão, sendo que o algoritmo principal é baseado na terceira 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: Sorting Algorithm according to the first approach

```
Input: Unsorted array A[0...n − 1]
Output: Sorted array A[0...n − 1]
procedure Sort(A, n)
    B[0] ← −1
    t ← 0
    k ← 0
    for i ← 0 to n − 2 do
        if A[i + 1] < A[i] then
            t ← t + 1
            B[t] ← i
    end
end
k ← t + 1
B[t + 1] ← n − 1
while k > 1 do
    t ← 0
    for i ← 0 to k − 2 do
        Merge_M(A, i, i + 1, i + 2, B)
        t ← t + 1
        B[t] ← B[i + 2]
        i ← i + 1
    end
    B[t + 1] ← n − 1
    k ← ⌈k/2⌉
end
return A
```

Figura 1: Algoritmo 1

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 (ascendente 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

```
Input: Unsorted array A[0...n − 1]
Output: Sorted array A[0...n − 1]
procedure Sort(A, n)
    B[0] ← −1
    t ← 0
    cont ← −1
    if A[0] <= A[1] then
        cont ← 1
    end
    for i ← 0 to n − 3 do
        if cont ∗ A[i + 1] < cont ∗ A[i] then
            t ← t + 1
            B[t] ← i
            cont ← −1
            if A[i + 1] <= A[i + 2] then
                cont ← 1
            end
        end
    end
    if cont ∗ A[n − 1] < cont ∗ A[n − 2] then
        t ← t + 1
        B[t] ← n − 2
    end
    k ← t + 1
    B[t + 1] ← n − 1
    while k > 1 do
        t ← 0
        for i ← 0 to k − 2 do
            Merge_M2(A, i, i + 1, i + 2, B)
            t ← t + 1
            B[t] ← B[i + 2]
            i ← i + 1
        end
        B[t + 1] ← n − 1
        k ← ⌈k/2⌉
    end
    return A

    procedure Merge_M2(A, p, q, r, B)
    n1 ← B[q] − B[p]
    n2 ← B[r] − B[q]
    cc ← A[B[q]] − A[B[p] + 1]
    ct ← A[B[r]] − A[B[q] + 1]
    contL ← −1
    contR ← −1
    if cc > 0 then
        contL = 1
    end
    if ct > 0 then
        contR = 1
    end
    if contL = 1 then
        for i ← 0 to n1 − 1 do
            L[i] ← A[B[p] + i + 1]
        end
    else
        for i ← 0 to n1 − 1 do
            L[i] ← A[B[q] − i]
        end
    end
    if contR = 1 then
        for i ← 0 to n2 − 1 do
            R[i] ← A[B[q] + i + 1]
        end
    else
        for i ← 0 to n2 − 1 do
            R[i] ← A[B[r] − i]
        end
    end
    L[n1] ← +∞
    R[n2] ← +∞
    i ← 0
    j ← 0
    for k ← 0 to n1 + n2 − 1 do
        if L[i] <= R[j] then
            A[B[p] + k + 1] ← L[i]
            i ← i + 1
        else
            A[B[p] + k + 1] ← R[j]
            j ← j + 1
        end
    end
    return A
```

Figura 2: Algoritmo 2

c. Terceira Abordagem

Alguns arrays podem ser escritos como uma combinação de dois subarrays, um crescente e outro decrescente, que não se intersectam. Um exemplo simples é um array que primeiro aumenta e depois diminui ou vice-versa. Por exemplo, o array $3,5,7,9,6,5,2$ apresenta esse caso. No entanto, podemos encontrar outros tipos de exemplos. Por exemplo, o array $8,10,7,9,6,11,13,5$ é uma combinação dos subarrays $8,9,11,13$ e $10,7,6,5$. Baseando-se nesta propriedade, podemos encontrar o número como segue. Escrevemos o array dado como uma combinação de dois arrays, um crescente e o outro decrescente, até onde for possível. Depois continuamos de onde paramos e repetimos as mesmas operações.

Nessa abordagem, se considerarmos quaisquer três termos do array, esses três números serão divididos em dois subarrays, como por exemplo, $\{a,b\}$ e $\{c\}$ ou $\{a\}$ e $\{b,c\}$. No pior dos casos, teremos $\left\lceil \frac{2}{3}n \right\rceil$ subarrays na primeira etapa. Sabemos que o primeiro subarray é crescente, o segundo decrescente, o terceiro crescente, e assim por diante. Portanto, podemos combinar o primeiro com o segundo, o terceiro com o quarto, e assim por diante, em subarrays ascendentes antes de iniciar a operação de mesclagem final. Isso fará com que o número de subarrays que temos antes de entrar no loop de mesclagem seja $\left\lceil \frac{n}{3} \right\rceil$ no pior dos casos.

A ideia principal do nosso algoritmo proposto é a seguinte: Primeiro, olhamos para os primeiros $n-1$ elementos do array dado em ordem e tentamos colocar cada elemento em um subarray crescente ou decrescente. Inicialmente, atribuímos valores $-\infty$ e $+\infty$ para duas variáveis chamadas *min* e *max*, respectivamente. Se o elemento olhado do array estiver entre *min* e *max*, comparamos este elemento com o próximo elemento do array. Se o próximo elemento for maior, adicionamos o elemento olhado ao array crescente e substituímos o valor de *min* por esse elemento. Caso contrário, adicionamos o elemento olhado ao array decrescente e substituímos o valor de *max* por esse elemento.

Se o elemento olhado do array não estiver entre os valores de *min* e *max* e for maior que *min*, mudamos o valor da variável *min* por esse elemento ao colocá-lo no array crescente. Se o elemento olhado do array não estiver entre os valores de *min* e *max* e não for maior que *min* mas for menor que *max*, substituímos o valor da variável *max* por este elemento ao inseri-lo no subarray decrescente. Quando não conseguimos colocar o elemento olhado em um subarray, interrompemos a divisão e combinamos os subarrays crescentes e decrescentes encontrados em um subarray crescente. Começando pelo elemento do array que não conseguimos colocar em nenhum subarray após o processo de mesclagem, repetimos as operações acima até que os primeiros $n-1$ elementos sejam colocados em um subarray crescente ou decrescente.

Depois, tentamos colocar o último elemento do array em um dos últimos subarrays crescentes ou decrescentes que obtivemos, e combinamos esses subarrays em um único subarray ordenado. Se não conseguirmos colocar o *n*-ésimo elemento em nenhum dos últimos subarrays, tratamos esse elemento como um subarray ordenado independente. Depois dessa etapa, combinamos nossos subarrays crescentes com o método de torneio em pares até que um único array ascendente seja formado.

1. Exemplo aplicado ao array $\{3,10,7,4,6,1,5,8,9,2,12,11\}$

- Inicialmente, $-\infty$ e $+\infty$ são atribuídos às variáveis *min* e *max*, respectivamente. - O primeiro elemento é 3, e como está entre *min* e *max*, comparamos com o próximo elemento 10. - Como 10 é maior que 3, 3 é colocado em um array crescente e o valor de *min* é substituído por 3. - O próximo elemento 10 está entre $min = 3$ e $max = +\infty$. Comparamos com o próximo elemento 7. Como 10 é maior que 7, é colocado em um array decrescente, e o valor de *max* muda para 10. - O terceiro elemento 7 está entre $min = 3$ e $max = 10$ e é comparado com 4. Como 7 é maior, é colocado em um array decrescente e o valor de *max* torna-se 7. - O próximo elemento 4 está entre $min = 3$ e $max = 7$. Comparamos com 6, que é maior. Portanto, 4 é colocado no array crescente e o valor de *min* muda para 4.

Similarmente, os elementos são processados até que não possam mais ser colocados em nenhum subarray, então são mesclados como antes.

III. RESULTADOS

No artigo analisado, foram testados dois algoritmos propostos em comparação com algoritmos de ordenação comumente preferidos, em três cenários diferentes: dados gerados aleatoriamente a partir de distribuições Gaussiana e Uniforme, dados reais publicamente disponíveis 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. Os códigos-fonte e os conjuntos de dados estão disponíveis publicamente para a reprodução do estudo.

Durante os testes, o pivô no algoritmo Quicksort foi escolhido como o termo mais próximo da média aritmética

Algorithm 3: Sorting algorithm according to the third approach: Main Proposed Sorting Algorithm

```
Input: Unsorted array A[0..n-1]
Output: Sorted array A[0..n-1]
procedure Sort(A, n)
B[0] ← -1
t ← 0
inc ← -∞
dec ← +∞
k ← 0
kk ← 1
for i ← 0 to n-2 do
    if inc <= A[i] and A[i] <= dec then
        if A[i] <= A[i+1] then
            inc ← A[i]
            L[k] ← A[i]
            k ← k+1
        else
            dec ← A[i]
            R[kk] ← A[i]
            kk ← kk+1
        end
    else
        if inc <= A[i] then
            inc ← A[i]
            L[k] ← A[i]
            k ← k+1
        else
            if dec >= A[i] then
                dec ← A[i]
                R[kk] ← A[i]
                kk ← kk+1
            else
                inc ← -∞
                dec ← +∞
                t ← t+1
                B[t] ← i-1
                Merge_Inv(L, R, A, k, kk, B, t)
                k ← 0
                kk ← 1
                i ← i-1
            end
        end
    end
end
t ← t+1
B[t] ← n-2
if A[n-1] >= L[k-1] then
    L[k] ← A[n-1]
    k ← k+1
    B[t] ← n-1
else
    if A[n-1] <= R[kk-1] then
        R[kk] ← A[n-1]
        kk ← kk+1
        B[t] ← n-1
    end
end
Merge_Inv(L, R, A, k, kk, B, t)
k ← t
if B[t] == n-2 then
    B[t+1] ← n-1
    k ← t+1
end
while k > 1 do
    t ← 0
    for i ← 0 to k-2 do
        Merge_M(A, i, i+1, i+2, B)
        t ← t+1
        B[t] ← B[i+2]
        i ← i+1
    end
    B[t+1] ← n-1
    k ← ⌈k/2⌉
end
return A

procedure Merge_M(A, p, q, r, B)
n1 ← B[q] - B[p]
n2 ← B[r] - B[q]
for i ← 0 to n1-1 do
    L[i] ← A[B[p]+i+1]
end
for i ← 0 to n2-1 do
    R[i] ← A[B[q]+i+1]
end
L[n1] ← +∞
R[n2] ← +∞
i ← 0
j ← 0
for k ← 0 to n1+n2-1 do
    if L[i] <= R[j] then
        A[B[p]+k+1] ← L[i]
        i ← i+1
    else
        A[B[p]+k+1] ← R[j]
        j ← j+1
    end
end
return A

procedure Merge_Inv(L, R, A, k, kk, B, t)
L[k] ← +∞
R[0] ← +∞
il ← 0
ir ← 0
for j ← B[t-1]+1 to B[t] do
    if L[il] <= R[kk-1-ir] then
        A[j] ← L[il]
        il ← il+1
    else
        A[j] ← R[kk-1-ir]
        ir ← ir+1
    end
end
return A
```

Figura 3: Algoritmo 3

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 como o tempo de execução para arrays desse tamanho. Para a visualização dos gráficos comparativos, foram utilizados todos os conjuntos de dados descritos na Seção 4, com ênfase nos resultados para arrays com dois milhões de elementos ou mais.

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

Os tempos de execução dos algoritmos para todos os arrays nos conjuntos de dados foram utilizados para criar as Figuras 4 e 5, que correspondem às distribuições Gaussiana e Uniforme, respectivamente. Além disso, apenas alguns resultados selecionados dos tempos de execução dos arrays foram apresentados nas Tabelas 4 e 5 para comparar o algoritmo proposto com outros. Os experimentos demonstraram que o algoritmo proposto superou o Quicksort, Mergesort e Heapsort em todos os arrays de ambos os conjuntos de dados.

b. Conjuntos de Dados Reais

Os resultados dos testes com dados reais são apresentados na Tabela 6. Conforme mostrado na tabela, o algoritmo proposto (Algoritmo 3) teve um desempenho superior ao Mergesort, Quicksort e Heapsort. A razão pela qual o algoritmo Quicksort teve um desempenho inferior ao Mergesort e Heapsort no conjunto de dados CCDD foi a presença de dados repetidos. Em arrays de pequeno tamanho, como no conjunto de dados RCL, o Algoritmo 1 e o Algoritmo 2 se mostraram mais eficientes que o Algoritmo 3.

c. Casos Especiais

O algoritmo de ordenação por inserção também foi incluído nos testes realizados nesta subseção, considerando que arrays ordenados são considerados casos especiais.

1. Conjunto de Dados em Ordem Reversa

Os resultados dos testes para este conjunto de dados estão apresentados na Tabela 7. O algoritmo proposto (Algoritmo 3) superou todos os outros algoritmos de ordenação testados neste estudo. Com esse algoritmo, subarrays são encontrados e agrupados em ordem reversa, o que explica o sucesso da abordagem. Dessa forma, o número de grupos mesclados no algoritmo é menor e a complexidade do algoritmo para este caso é $O(n)$. No entanto, também pode ser observado na tabela que o Algoritmo 2 teve um desempenho melhor que o Algoritmo 3, pois o Algoritmo 2 consegue encontrar apenas um subarray decrescente ao percorrer o array do início ao fim.

d. . Conjunto de Dados de Arrays Constantes

Os resultados dos testes para este conjunto de dados são exibidos na Tabela 8. A repetição dos elementos do array não aumenta o tempo de processamento do nosso algoritmo. Para arrays constantes, o algoritmo proposto (Algoritmo 3) percebe o array como uma sequência não decrescente. Nesse conjunto de dados, o algoritmo de ordenação por inserção teve um desempenho ligeiramente melhor do que o Algoritmo 3, mas os Algoritmos 1 e 2 tiveram o melhor desempenho. Isso ocorre porque os Algoritmos 1 e 2 examinam o array do início ao fim e encontram apenas um subarray mais rapidamente do que o Algoritmo 3.

e. Conjunto de Dados Ordenados

Os resultados dos testes para este conjunto de dados são mostrados na Tabela 9. Nosso algoritmo proposto (Algoritmo 3) é muito eficiente neste caso especial. A razão por trás dos resultados é que o Algoritmo 3 encontra apenas um subarray ordenado. Nesse caso, o algoritmo de ordenação por inserção teve um desempenho ligeiramente melhor do que o Algoritmo 3, mas os Algoritmos 1 e 2 tiveram o melhor desempenho. Isso ocorre porque os Algoritmos 1 e 2 examinam o array do início ao fim e encontram apenas um subarray mais rapidamente do que o Algoritmo 3.

f. Elementos Repetidos nos Arrays

Os resultados obtidos para este conjunto de dados são mostrados na Tabela 10. Elementos repetidos no array têm um efeito significativo no desempenho dos algoritmos de ordenação; por exemplo, esses valores pioram o desempenho do algoritmo Quicksort. Por outro lado, nossos algoritmos não são afetados por essa situação. A razão é que nossos algoritmos buscam subarrays ordenados e inversamente ordenados. Além disso, o Algoritmo 3 inclui os mesmos valores, então temos menos partições, tornando a mesclagem mais rápida. No entanto, o Algoritmo 2 teve um desempenho próximo ao do Algoritmo 3. Heapsort e Mergesort também tiveram um desempenho eficiente neste caso especial. Como esperado, Quicksort e ordenação por inserção apresentaram piores valores de tempo de execução.

g. Conjunto de Dados de Arrays que são Combinações de Dois Subarrays Crescentes e Decrescentes

Os resultados dos testes para este conjunto de dados são apresentados na Tabela 11. Em tais arrays de entrada, o algoritmo proposto (Algoritmo 3) mostrou um desempenho muito superior em comparação com outros algoritmos.

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	0.410155	0.327630	0.307932	0.285997
4 000 000	1.396416	1.526262	2.298623	1.442417	1.363992	1.258511

Figura 4: 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 5: 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 6: 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	0.00009	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 7: 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	0.000122
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	0.053789	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 8: 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	0.269995	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 9: 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 10: 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	0.022392	0.025353	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 11: Tabela 11

V. CONCLUSÃO

Com o aumento da quantidade de dados provenientes de várias fontes, a importância dos algoritmos de ordenação também cresceu. Este estudo propõe um algoritmo de ordenação baseado no método divide-and-conquer, considerado uma melhoria do algoritmo Mergesort. A operação de divisão adota uma abordagem mais inteligente que a do Mergesort clássico. O algoritmo apresenta complexidade $O(n \log n)$ no melhor caso e $O(n^2)$ no pior caso, utilizando espaço adicional $O(n)$ assim como o Mergesort.

Foram produzidos diferentes conjuntos de dados com distribuições Gaussianas e Uniformes, além de dados públicos reais e conjuntos especiais criados com diferentes cenários. Os resultados experimentais mostram que o algoritmo proposto é mais rápido que os algoritmos Mergesort, Heapsort e Quicksort. No entanto, o algoritmo proposto (Algoritmo 3) não é estável. Por exemplo, ao considerar o array $2,5,4,3,4$, o algoritmo divide este array em $\{2,3,4\}$ à esquerda e $\{5,4\}$ à direita, resultando na troca de posições dos elementos 4.

O Algoritmo 1 é estável, mas do ponto de vista teórico e prático, o Algoritmo 3 é mais significativo. A estabilidade do Algoritmo 3 é dificultada pela condição de divisão não ser simples. A estabilidade dos algoritmos de ordenação, como o Quicksort, também é afetada por essas condições. Tornar o Algoritmo 3 estável não é tarefa fácil, mas é algo que esperamos conseguir em estudos futuros.

Além disso, será objeto de futuros estudos investigar quais arrays podem ser escritos como a combinação de dois sub-arrays, um crescente e outro decrescente, e encontrar o número médio de sub-arrays alternadamente crescentes e decrescentes para um array dado aleatoriamente.

REFERÊNCIAS

1. Z. Abo-Hammour, O. Alsmadi, S. Momani, O. AbuArqub, et al., "A genetic algorithm approach for prediction of linear dynamical systems," Math. Probl. Eng. 2013 (2013).

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. Gharehchopogh, 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. Marszalek, "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. A.S. Mohammed, Ş.E. Amrahov, F.V. Çelebi, "Bidirectional conditional insertion sort algorithm; an efficient progress on the classical insertion sort," Future Gener. Comput. Syst. 71 (2017) 102–112.

30. S. Goel, R. Kumar, "Brownian motus and clustered binary insertion sort methods: an efficient progress over traditional methods," Future Gener. Comput. Syst. 86 (2018) 266–280.

31. Y.M. Omar, H. Osama, A. Badr, "Double hashing sort algorithm," Comput. Sci. Eng. 19 (2) (2017) 63–69.

32. H.M. Bahig, "Complexity analysis and performance of double hashing sort algorithm," J. Egyptian Math. Soc. 27 (1) (2019) 1–12.

33. A. Zutshi, D. Goswami, "Systematic review and exploration of new avenues for sorting algorithm," Int. J. Inf. Manag. Data Insights 1 (2) (2021) 100042.

34. A. Levitin, "Introduction to the Design and Analysis of Algorithms," Pearson, 2011.

35. D.R. Musser, "Introspective sorting and selection algorithms," Softw.- Pract. Exp. 27 (8) (1997) 983–993.

36. M. Nowicki, "Comparison of sort algorithms in Hadoop and PCJ," J. Big Data 7 (1) (2020) 1–28.

37. H.M. Bahig, "A new constant-time parallel algorithm for merging," J. Supercomput. 75 (2) (2019) 968–983.

38. R. Bar Yehuda, S. Fogel, "Partitioning a sequence into few monotone subsequences," Acta Inform. 35 (5) (1998) 421–440.

39. R. Siders, "Monotone subsequences in any dimension," J. Combin. Theory Ser. A 85 (2) (1999) 243–253.

40. J.S. Myers, "The minimum number of monotone subsequences," Electron. J. Combinatorics (2002) R4.

41. B. Yang, J. Chen, E. Lu, S. Zheng, "A comparative study of efficient algorithms for partitioning a sequence into monotone subsequences," in: Theory and Applications of Models of Computation: 4th International Conference, TAMC 2007, Shanghai, China, May 22-25, 2007. Proceedings 4, Springer, 2007, pp. 46–57.

42. W. Samotij, B. Sudakov, "On the number of monotone sequences," J. Combin. Theory Ser. B 115 (2015) 132–163.

43. J. Balogh, P. Hu, B. Lidický, O. Pikhurko, B. Udvari, J. Volec, "Minimum number of monotone subsequences of length 4 in permutations," Combin. Probab. Comput. 24 (4) (2015) 658–679.

44. N. Linial, M. Simkin, "Monotone subsequences in high-dimensional permutations," Combin. Probab. Comput. 27 (1) (2018) 69–83.

45. T. Mansour, R. Rastegar, A. Roitershtein, G. Yıldırım, "The longest increasing subsequence in involutions avoiding 3412 and another pattern," Pure Math. Appl. 30 (4) (2022) 11–21.

46. "Database of registered common land in England, 2022," https://www.data.gov.uk/dataset/05c61ecc-efa9-4b7f-8fe6-9911afb44e1a/database-of-registered-common-land-in-england, (Accessed 11/24/2023).

47. "Traffic commissioners: Goods and public service vehicle operator licence records, 2014," https://www.data.gov.uk/dataset/2a67d1ee-8f1b-43a3-8bc6-e8772d162a3c/traffic-commissioners-goods-and-public-service-vehi (Accessed 11/25/2023).

48. "National statistics postcode lookup, 2022," https://geoportal.statistics.gov.uk/datasets/9ac0331178b0435e839f62f41cc61c16/about, (Accessed 11/25/2023).

49. N. Elgiriyewithana, "Credit card fraud detection dataset, 2023," https://www.kaggle.com/datasets/nelgiriyewithana/credit-card-fraud-detection-dataset-2023, (Accessed 11/26/2023).