Comparativo do Desempenho entre Algoritmos de Ordenção Clássicos

João dos Santos Neto

***Abstract*— No mundo da computação, a frequência em que a quantidade de dados se encontra é altíssima, próxima a escala de milhões, isso ocorre pelo fato do mundo estar atualmente bastante globalizado. Em certos momentos, é preciso que haja uma ordenação.**

Essa necessidade é praticamente impossível para o ser humano realizar manualmente, logo existem algoritmos clássicos de ordenação que partem do menos complexos como o Bubble-Sort, Insertion-Sort e Selection-Sort para os mais complexos como o Quick-Sort e Merge-Sort.

Para uma abordagem mais complexa para esse problema visa-se obter uma comparação entre os algoritmos de ordenação clássicos, observando o desempenho em tempo de execução utilizando a média, suas características e Análise Assintótica para diferentes cenários.

I. INTRODUÇÃO

Atualmente, no ano de 2023, é bastante comum as empresas ou organizações terem informações em quantidades significativas em seus bancos de dados. Esses dados têm uma finalidade de uso dentro delas, sendo obrigadas a seguir normas, regulações e leis que protegem seus clientes [1].

Da mesma forma, a área de Big Data também precisa lidar com inúmeras quantidades de dados, nos quais, em muitos casos, há a necessidade de estarem ordenados, o que ajuda a ter um melhor controle e visualização para coletar informações de interesse [2].

Devido a essa enorme quantidade de dados a serem manipulados, é necessário de uma organização, seja ela ordenada de forma crescente ou decrescente. Isso é um fator importante para que as informações sejam analisadas de forma mais fácil, com uma boa visualização e rapidez [3].

Portanto, como solução para esse problema, foram utilizados os seguintes algoritmos: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort e Quick Sort [4]. Serão aplicados em diferentes cenários de entrada para verificação do tempo de execução, onde tais entradas estarão organizadas de maneira crescente, decrescente e desordenada.

II. METODOLOGIA

Para conseguir obter resultados que faça a comparação dos algoritmos utiliza-se os seguintes tópicos abaixo:

A. Ambiante de Teste

TABLE I
CONFIGURAÇÕES DO AMBIENTE

Notebook	Processador	Memória RAM
Lenovo ideapad 3	AMD Ryzen 5 5500U	6 GB

B. Algortimos de Ordenação

1) Bubble Sort: O Bubble Sort é um algoritmo de ordenação dos mais simples. Seu entendimento é percorrer o conjunto de números várias vezes e, a cada passagem, realizar diversas trocas para levar o maior elemento da sequência para o topo.

O processo recebe o nome de ”Bubble Sort” devido a maneira como os elementos ”flutuam” para suas posições corretas, assim como as bolhas sobem na água. A cada iteração, o maior elemento ”sobe” para o topo da lista, enquanto os elementos menores ”descem”[5].

Algorithm 1 Bubble Sort

0: **procedure** BUBBLESORT(A, n) {A é o array a ser ordenado, n é o tamanho do array} **for** $i \leftarrow 0$ *to* $n-1$ **do**

end

$j \leftarrow 0$ to $n-1-i$ {Reduz o número de comparações a cada iteração} **if** $A[j] > A[j+1]$ **then**

0:

end

Trocar $A[j]$ e $A[j+1]$

0:

0:

0:

0: =0

```
int *bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++){
            if (arr[j] > arr[j+1]) {
                int temp = arr[j],;
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    return arr;
}
```

2) Insertion Sort: O Insertion Sort é um algoritmo simples e eficiente, especialmente adequado para conjuntos de dados pequenos. Sua abordagem envolve mover um elemento por vez através da lista, comparando cada elemento com os demais e colocando-o na posição correta. Essa técnica de ”inserir” elementos na posição apropriada contribui para a ordenação gradual da lista.

O Insertion Sort, apesar de sua simplicidade e facilidade de implementação, possui uma complexidade de tempo

de O(n**2), o que significa que seu desempenho piora quadráticamente com o aumento do tamanho dos conjuntos de dados. Essa característica limita sua eficácia para ordenar grandes conjuntos de dados[6].

Algorithm 2 Insertion Sort

procedure INSERTIONSORT(A, n) {A é o array a ser ordenado, n é o tamanho do array} **for** $i \leftarrow 1$ *to* $n-1$ **do**
end
$chave \leftarrow A[i]$
$j \leftarrow i-1$ **while** $j \geq 0$ *e* $A[j] > chave$ **do**
end
$A[j+1] \leftarrow A[j]$
$j \leftarrow j-1$
$A[j+1] \leftarrow chave$
=0

```
int *insertionSort(int arr[], int n){
    int i, key, j;
    for (i = 1; i < n; i++){
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
    return arr;
}
```

3) Selection Sort: Selection Sort basea-se em repor sempre o menor valor para a primeira posição, depois o de segundo menor valor para a segunda posição, e assim sucessivamente com o restante, até os últimos dois elementos.

A ideia central do Selection Sort é dividir a lista em duas partes. A parte ordenada começa vazia e cresce, enquanto a parte não ordenada contém os elementos restantesua complexidade de tempo é da ordem de O(n**2), o que significa que seu desempenho pode ser limitado para conjuntos de dados grandes.[7].

```
int *selectionSort(int arr[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++){
            if (arr[j] < arr[min_idx]){
                min_idx = j;
            }
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
    return arr;
}
```

Algorithm 3 Selection Sort

procedure SELECTIONSORT(A, n) {A é o array a ser ordenado, n é o tamanho do array} **for** $i \leftarrow 0$ *to* $n-1$ **do**
end
$indiceMenor \leftarrow i$ **for** $j \leftarrow i+1$ *to* $n-1$ **do**
end
$A[j] < A[indiceMenor]$
$indiceMenor \leftarrow j$
Trocar $A[i]$ e $A[indiceMenor]$
=0

4) Merge Sort: Merge sort é um algoritmo baseado na concepção de divisão e conquista, onde divide continuamente o conjunto ao meio até que ela não possa ser mais dividido para que seja possivel interlar e ordenar desde o elemento isolado até os subconjuntos acima que foram divididos.

Essa técnica de divisão e conquista é a chave para a eficiência do Merge Sort. A complexidade de tempo do Merge Sort é O(n log n), o que o torna mais eficiente em comparação com algoritmos de ordenação quadrática, como o Bubble Sort ou o Insertion Sort, especialmente para conjuntos de dados grandes[8].

```
void mergeSort(int arr[], int l, int r){
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
void merge(int arr[],int l,int m,int r){
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2){
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
```

```
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1){
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2){
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Algorithm 4 Merge Sort

procedure MERGESORT(A, n) {A é o array a ser ordenado, n é o tamanho do array} **if** $n > 1$ **then**
end
$meio \leftarrow \lfloor n/2 \rfloor$
  $esquerda \leftarrow$ subarray$(A, 0, meio)$
  $direita \leftarrow$ subarray$(A, meio, n)$
  MERGESORT($esquerda, meio$)
  MERGESORT($direita, n - meio$)
  MERGE($A, esquerda, meio, direita, n - meio$)

  procedure MERGE($A, esquerda, nl, direita, nd$)
    $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$
while $i < nl$ *e* $j < nd$ **do**
end
esquerda[i] $\leq$ direita[j]
    $A[k] \leftarrow$ esquerda[i]
    $i \leftarrow i + 1$ **else**
end
$A[k] \leftarrow$ direita[j]
    $j \leftarrow j + 1$

    $k \leftarrow k + 1$
    while $i < nl$ **do**
end
$A[k] \leftarrow$ esquerda[i]
  $i \leftarrow i + 1$, $k \leftarrow k + 1$
  while $j < nd$ **do**
end
$A[k] \leftarrow$ direita[j]
$j \leftarrow j + 1$, $k \leftarrow k + 1$
end procedure
=0

5) Quick Sort: Quick Sort baseia-se no mesmo principio do Merge Sort , utiliza a técnica particionamento, que significa escolher um número qualquer, chamado de pivot, e colocá-lo em uma posição tal que os demais à esquerda sejam menores ou iguais e à direita são maiores.

Quick Sort é conhecido por sua eficiência média e é amplamente utilizado na ordenação de conjuntos muitos grandes, destacando-se pela rapidez na ordenção dos elementos com base na escolha estratégica do pivot[9].

Algorithm 5 QuickSort

procedure QUICKSORT($A, inicio, fim$) {A é o array a ser ordenado, $inicio$ e fim definem a subárea a ser ordenada} **if** $inicio < fim$ **then**
end
$p \leftarrow$ PARTICIONAR($A, inicio, fim$)
  QUICKSORT($A, inicio, p - 1$)
  QUICKSORT($A, p + 1, fim$)

  procedure PARTICIONAR($A, inicio, fim$)
    $pivo \leftarrow A[fim]$
    $i \leftarrow inicio - 1$
for $j \leftarrow inicio$ *to* $fim - 1$ **do**
end
$A[j] \leq pivo$
    Trocar $A[i + 1]$ e $A[j]$
    $i \leftarrow i + 1$

    Trocar $A[i + 1]$ e $A[fim]$
    return $i + 1$
  end procedure=0

```
void swap_quick(int* a,int* b){
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[],int low,int high){
    int mid = low + (high - low) / 2;
    int pivot = arr[mid];
    swap_quick(&arr[mid], &arr[high]);
    int i = (low - 1);
    for (int j = low; j <= high-1;j++){
        if (arr[j] <= pivot) {
            i++;
            swap_quick(&arr[i], &arr[j]);
        }
    }
    swap_quick(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quick(int arr[],int low,int high){
    if (low < high) {
```

```
    int pi = partition(arr, low, high);
    quick(arr, low, pi - 1);
    quick(arr, pi + 1, high);
  }
```

C. In-Place

Um algoritmo ”in-place” é aquele que opera diretamente sobre a estrutura de dados de entrada, modificando-a sem a necessidade de mais espaço[10]. Ele é mais eficiente em termos de uso de memória, pois não requer alocação de espaço extra proporcional ao tamanho dos dados de entrada.

D. Não In-place

Um algoritmo ”não in-place” é aquele que, ao operar sobre dados, precisa de mais espaço para armazenar informações temporárias, além da estrutura de dados original. Isso pode resultar em maior uso de memória, pois o espaço necessário pode ser proporcional ao tamanho dos dados de entrada.

E. Análise Asssintótica

Análise assintótica leva em consideração grandes entradas para tornar relevante apenas a ordem de crescimento das funções de tempo de execução, ou seja é feita uma avaliação do algoritmo quando o valor de N tende a infinito[11].

Para este experimento será utilizada a Notação Big O, ideal para expressar limites assintóticos superiores, uma vez que ela limita o crescimento do tempo de execução superior para valores sufientemente grandes de entrada.

F. Linguagem de Programação

Será desfrutado da Linguagem C[12], que foi desenvolvida por Dennis Ritchie na decade de 70, por apresentar aspectos bastante fortes no seu controle de memoria, em sua sintaxe simples e eficiência, esses aspectos ajudarão a proporcionar resultados mais precisos.

G. Ambiente de Desenvolvimento Integrado

Um software que oferece ferramentas para desenvolvedores criarem, editarem, depurarem e gerenciarem código-fonte de maneira eficiente[13]. Exemplos incluem Visual Studio Code, Eclipse e PyCharm. Nesta pesquisa será utilizado o Visual Studio Code como ambiente.

H. Entradas

Para este experimento, serão utilizadas 3 conjuntos com total de 500.00 mil números cada, organizados de maneira crescente, decrescente e desordenados. Com isso, será possível obter uma ánalise comparativa através do Método de Validação.

Essa quantidade foi escolhida pelo fato de que o Ambiente de Teste não conseguir oferecer uma quantidade maior de memória RAM para realização dos experimentos com entradas em escala de milhões.

I. Extração da Média

Será desenvolvido um código de acordo com a linguagem de programação escolhida para realizar o cálculo do tempo de execução. Os algoritmos serão submetidos a uma etapa que consiste em 30 repetições, com o objetivo de extrair a média. A Figura 1 demonstra o passo a passo realizado do respectivo código.

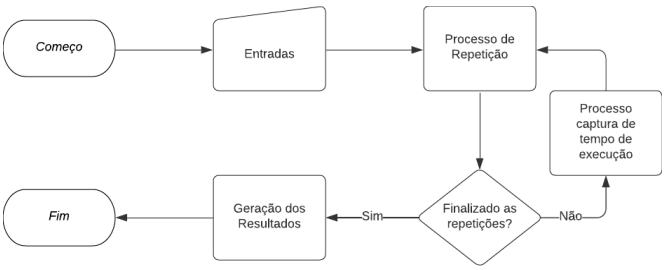


Fig. 1. Fluxograma da Extração da Média

III. RESULTADOS

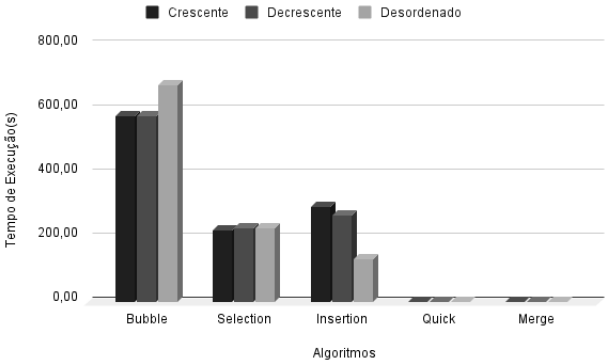
A Tabela II apresenta os resultados obtidos pela quantidade de números citadas e pela extração da média proposta.

TABLE II
MÉDIAS DO TEMPO DE EXECUÇÃO(S)

Algortimos X Entradas	Crescente	Decrescente	Desordenado
Bubble	580,67	579,76	675,76
Selection	225,93	230,72	231,88
Insertion	298,46	270,66	133,49
Quick	0,30	0,30	0,11
Merge	0,04	0,04	0,04

A Figura 2 mostra um melhor parâmetro de visualização da comparação feita dos resultados obtidos pelas médias.

Fig. 2. Tempo de Execução(s) X Algoritmos



A Tabela III apresenta as notações obtidas pela avaliação da Análise de Assintótica[14] de cada algoritmo apresentado.

TABLE III
NOTAÇÃO ASSINTÓTICA

Algortimos X Complexidade de Tempo	Melhor Caso	Caso Médio	Pior Caso
Bubble	O(n)	O(n2)	O(n2)
Selection	O(n2)	O(n2)	O(n2)
Insertion	O(n)	O(n2)	O(n2)
Quick	O(n log n)	O(n log n)	O(n2)
Merge	O(n log n)	O(n log n)	O(n log n)

Com base nas ilustrações e dados apresentados, torna-se evidente que algoritmos mais simples, como Bubble, Insertion e Selection, não são viáveis para grandes quantidades de dados. No entanto, para casos em que a quantidade de dados se situa dentro do intervalo de 100 a 1000, sua aplicação é recomendada devido à simplicidade de implementação.

Por outro lado, o uso de Merge e Quick é essencial em quantidades elevadas; no entanto, sua implementação é mais complexa. Vale ressaltar que, para que o Quick demonstre total desempenho, é necessário lidar com conjuntos desordenados. Enquanto isso, o Merge apresenta características consistentes em quaisquer conjuntos, tornando-se a melhor opção para resolução.

Apesar de o uso do Merge ser a melhor opção neste experimento, este é considerado um algoritmo não in-place, ou seja, irá consumir mais memória para realizar a ordenação dos elementos. Por outro lado, o Quick demonstra um ótimo desempenho em cenários de completa desordenação, além de ser um algoritmo in-place.

Portanto, em cenarios de baixas quantidades de dados, onde não há uma alta escabilidade, e com implementações menos complexa usa-se o Bubble, Insertion ou Selection. Para ambientes de mais escabilidade recomenda-se fortemente os algortimos Merge e Quick apesar de serem mais complexos que os demais, todavida usa-se o Merge com cautela pelo fato de usar mais espaço de memória para ordenar.

REFERENCES

[1] Walter Aranha Capanema. A responsabilidade civil na lei geral de proteção de dados. *Cadernos Jurídicos: Direito Digital e proteção de dados pessoais, ano*, 21:163–170, 2020.

[2] Xin Luna Dong and Divesh Srivastava. Big data integration. In *2013 IEEE 29th international conference on data engineering (ICDE)*, pages 1245–1248. IEEE, 2013.

[3] Andrelina Maria Machado de Moraes, Ana Cristina da Silva Barros, Francisco Igor dos Reis, Girlane Caroline Pereira Santos Gonçalves, Georgia Thamyres Leite Ramos, Janete Brasil, Layla Dhierissa dos Santos Lima Torres, Lorena Gabrielle Ribeiro de Azevedo, Marcelina Gomes Brandão, Michele Pereira Araújo, et al. A importância da atuação do enfermeiro como gestor na organização e administração da atenção básica: uma revisão integrativa. *Revista Eletrônica Acervo Saúde/Electronic Journal Collection Health ISSN*, 2178:2091, 2018.

[4] Victor Augusto Fernandes da Silva[1]. Revisão dos algoritmos de ordenação.

[5] Owen Astrachan. Bubble sort: an archaeological algorithmic analysis. *ACM Sigcse Bulletin*, 35(1):1–5, 2003.

[6] Michael A Bender, Martin Farach-Colton, and Miguel A Mosteiro. Insertion sort is o (n log n). *Theory of Computing systems*, 39:391–397, 2006.

[7] Sultanullah Jadoon, Salman Faiz Solehria, and Mubashir Qayum. Optimized selection sort algorithm is faster than insertion sort algorithm: a comparative study. *International Journal of Electrical & Computer Sciences IJECS-IJENS*, 11(02):19–24, 2011.

[8] Joella Lobo and Sonia Kuwelkar. Performance analysis of merge sort algorithms. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 110–115. IEEE, 2020.

[9] Wang Xiang. Analysis of the time complexity of quick sort algorithm. In *2011 international conference on information management, innovation management and industrial engineering*, volume 1, pages 408–410. Ieee, 2011.

[10] Yan Gu, Omar Obeya, and Julian Shun. Parallel in-place algorithms: Theory and practice. In *Symposium on algorithmic principles of computer systems (APOCS)*, pages 114–128. SIAM, 2021.

[11] Moacir A Ponti. Notação assintótica e complexidade.

[12] Jorge Matheus Gomes Viegas. Análise de desempenho de linguagens de programação e apis quânticas. 2021.

[13] Mik Kersten and Gail C Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, 2005.

[14] Marco Antonio de Castro Barbosa, Laira Vieira Toscani, and Leila Ribeiro. Anac–uma ferramenta para análise automática da complexidade de algoritmos. *Revista do CCEI*, 5(8):57, 2001.