Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Davi Bernardes Oliveira Ribeiro - 8183

ANÁLISE DA COMPLEXIDADE DE ALGORTIMOS

Rio Paranaíba - MG

2023

Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Davi Bernardes Oliveira Ribeiro - 8183

ANÁLISE DA COMPLEXIDADE DE ALGORTIMOS

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmo da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

Rio Paranaíba - MG

2023

1 RESUMO

Os Algortimos estão presentes cada vez mais presentes e tendo grandes impactos na vida dos seres humanos, podemos observar o seu uso em mídias sociais, navegações por gps, pesquisas na internet e também e vários aspectos, por isto devemos ter um olhar mais atento para eles. [CORMEN,2012]

Especificadamente, os algoritmos de ordenação apresentam grande importância no estudo teórico de algoritmos, bem como em aplicações rotineiras e práticas do cotidiano. Diante dessa grande importância ao longo dos anos surgem alguns algoritmos de ordenação na literatura, onde são analisados casos em que cada algoritmo se enquadra. Sendo assim se torna necessário um estudo para analisar o comportamento desses algoritmos, bem como suas complexidades para indicar qual é o mais adequado para cada tipo de problema. [FACOM,2018]

...

Sumário

2 INTRODUÇÃO

A ordenação de elementos em computação é uma tarefa fundamental, desempenhando um papel importante na otimização de algoritmos e na melhoria do desempenho de sistemas. Diversos algoritmos de ordenação foram desenvolvidos ao longo dos anos, cada um com suas próprias características e complexidades. Este estudo explora e testa alguns dos algoritmos de ordenação mais comuns, como Insertion, Bubble, Shell, Merge Sort, Quick, Heap e outros, com o objetivo de analisar sua complexidade e tempo de execução em diferentes casos. Através de testes e experimentos, examinaremos as complexidades de cada algoritmo, considerando os casos de melhor e pior desempenho, e apresentaremos os resultados obtidos.

A eficiência dos algoritmos de ordenação é nescessárias para a seleção do algoritmo mais adequado para uma aplicação. Investigaremos a complexidade de tempo e espaço de cada algoritmo, questionando como eles se comportam sob diferentes testes. Além disso, a metodologia de melhor e pior caso será aplicada para fornecer resultados valiosos sobre o desempenho esperado em diferentes cenários. Ao final, os resultados obtidos nos testes de desempenho serão apresentados, destacando as vantagens e desvantagens de cada algoritmo. Este estudo tem como objetivo aprofundar nossa compreensão dos algoritmos de ordenação.

...

3 ALGORITMOS

3.1 *Insertion Sort*

O algortimo Insetion Sort é uma técnica e método bastente conhecido, com uma simples e eficiente ordenação, ele organiza elementos em uma lista ou array. Ele se se fomenta do princípio de construir uma lista ordenada com um elemento de cada vez, iserindo assim respectivamente, elementos não classificados na posição correta dentro da lista já classificada, tendo suas vantagens e desvantegens como seus casos possíveuis, como o Melhor caso (Hipostse), Pior caso e também o médio Caso, para sua utilização a depender de cada situação demonstrada.

Suas vantagens são bastantes proveitosas em sua utilização como:

1. **Pequenos Conjuntos de Dados:** Quando a quantidade de dados é pequena, ele pode ser mais rápido que vários algortimos mais complexos.

2. **Dados Quase ordenados:** Quando a maioria dos elementos que estão no vetor ou array, estão em ordem, requerindo assim poucas comparações a serem feitas.

3. **Aprendizado inicial de Algortimos:** O Insertion Sort é bastante utilizado para ensinar sobre algortimos de ordenação, devido a sua alta fácilidade em aprendizado.

Análise Matemática:

Analisando a forma matemática do Insertion Sort, podemos observar que o algoritmo tem um custo matemático para sua aplicação, defindo assim, com T(n) com seu custo total.

O Algoritmo pode ser representado seu custo abaixo, observe a figura 1 a seguir:

$$T(n) = c_1 + c_2 + nc_3 + (n-1)c_4 + (n-1)c_5 + c_6(n-1) + c_7\sum_{j=1}^{n-1} t_j + c_8\sum_{j=1}^{n-1}(t_j - 1) + c_9\sum_{j=1}^{n-1}(t_j - 1) + c_{10}(n-1)$$

Depende-se de Tj de interações para cada valor de j.

Análise Pseudo Código:

A inserção dos elementos é divido em duas fases, a primeira etapa, procura a posição de inserção do elemento. Na segunda etapa, desloca-se os elementos da esquerda para a direita para que assim a posição de inserção fique disponível para que o elemento seja colocado.

Podemos observar o trecho do pseudocódigo do insertion short para entender melhor o seu funcionamento e sua lógica por trás do algoritmo: [CORMEN,2012]

```
//algoritmo adaptado o livro "Algoritmos: teoria e prática" (Cormen et al)
01. insertionSort(A[0...n-1])
02. |   para i ← 1 até n - 1
03. |   |   elemento ← A[i]
04. |   |   para(j ← i - 1; j ≥ 0 E A[j] > elemento; j ← j - 1)
05. |   |   |   A[j + 1] ← A[j] //deslocamento para a direita
06. |   |   fim_para
07. |   |   A[j + 1] = elemento //insere o elemento na parte ordenada
08. |   fim_para
09. fim_insertionSort
```

Neste trecho, O algoritmo utiliza o laço enquanto (while) por para (for).

Analisando a Figura 1.2, pode-se ver que teoricamente, o algoritmo do Insertion Sort é simples comparado a demais algoritmos, porém tem casos de aplicação e cada caso impacta diretamente no desenvolvimento do algoritmo.

3.2 *Bubble Sort*

O algortimo Bubble Sort é um algortimo simples, que funciona trocando os elementos se eles estiverem na ordem errada. O algoritmo não é adequado para grandes conjuntos de dados, pois sua complexidade de pior caso é bastante alta. O algortimo é de tipo por bolha, onde ele percorre na esquerda e compara os elementos, o elemento mais alto é colocado na direita, fazendo assim a ordenação, o processo continua para encontrar o segundo maior posteriomente e assim por diante até o final, tendo suas vantagens e desvantegens como seus casos possíveis de existir,

Suas vantagens são bastantes proveitosas em sua utilização como:

1. **Algortimo Estável:** Quando os dados tem o mesmo valor de chave, mantêm sua ordem relativa na saída classificada.

2. **Fácil Aprendizado:** O Bubble Sort é bastante utilizado para ensinar sobre algortimos de ordenação, devido também como o Insertion Sort, tendo ambos sua alta fácilidade em aprendizado.

Análise Matemática:

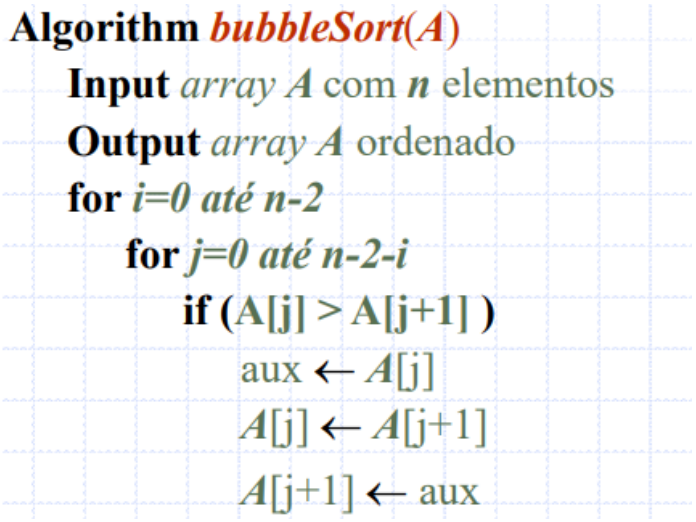
Analisando a forma matemática do Bubble Sort, considerando um vetor de n elementos, temos comparações em qualquer caso sempre executadas sendo feitas, com complexidade representada por O(n2). [URFPE, 2020]
Sua eficiência é representada assintoticamente por O(n2):

$$\sum_{i=1}^{n} C_i = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Análise Pseudo Código:

Observando a estrutura do Bubble Sort, O método da bolha resolve problemas através de várias passagens sobre a sequência no código, fazendo uma "varredura"por completo. Na primeira passagem, uma vez encontrado o maior elemento, este terá sua colocação trocada até atingir a última posição. Na segunda passagem, uma vez encontrado o segundo maior elemento, este terá sua colocação trocada até atingir a penúltima posição, E assim por diante.

Podemos observar o trecho do pseudocódigo do Bubble sort na figura 2.2 para entender melhor o seu funcionamento e sua lógica por trás do algoritmo: [IFRN,2013]



Analisando a Figura 2.2, pode-se ver que teoricamente, o algoritmo do Bubble Sort é simples na construção com dois laços for de repetição, tendo sempre independente do resultado do algortimo, a sua busca feita de maneira sucetivamente realizada até sair dos dois laços de repetição, tendo um custo pior para vetores com altos índices de verificações.

3.3 *Selection Sort*

O algortimo Selection Sort é um algortimo que tem sua ordenação por seleção, onde funciona comparando os elementos, por seleção ou direta, a principal ideia é ordenadar o vetor selecionando em cada iteração os menores (ou maiores) elementos os selecionando e assim, ordenando da esquerda para a direita como em caso de ordem crescente do algoritmo, onde quando o menor elemento da lista é localizado na 1a posição a iteração ocorre para localizar o segundo menor elemento da lista na 2a posição e assim sucessivamente até que a lista se ordene em ordem crescente como exemplo, tendo vantagens e desvantagens na sua utilização a depender do caso ou quantidade de números ou elementos. [URFPE, 2020]

O Selection Sort tem vantagens, que são bastantes proveitosas em sua utilização nos casos nescessários, como:

1. **Fácil Implementação:** O Selection Sort é um dos algoritmos de ordenação mais simples de entender e de ter a sua implementação, devido a simplicidade do código. podendo ser útil para fins educativos. [CORMEN, 2009]

2. **Mínimo uso de memória:** O Selection Sort é otimizado pois, classifica a lista in-place, ou seja, não requer memória adicional além daquela usada para armazenar a lista original. [CORMEN, 2009]

3. **Estabilidade:** O algortimo, possuí uma estabilidade em seu uso, tendo a ordenação estável, o que significa que ele não altera a ordem relativa de elementos com chaves iguais, podendo ser bastante últil em algumas aplicações. [CORMEN, 2009]

Análise Matemática:

Analisando a forma matemática do Selection Sort, podemos considerar um vetor de n elementos, temos comparações em qualquer caso, com complexidade representada por O(n2) como na fígura 3.1 abaixo, pode-se ver que tem o peso bem parecido com o *Bubble Sort*. [URFPE, 2020]
Sua eficiência é representada assintoticamente por O(n2):

$$\sum_{i=1}^{n} C_i = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Análise Pseudo Código:

Observando toda a construção do Selection Sort, O método seleção e comparação feitas, resolve problemas pela várias repetições de comparações com os menores (ou maiores) elementos para sua posição de ordenação, quando terminado, temos uma sequência totamente ordenada pela sua configuração.

O algortimo procura o valor mínimo para começar por toda a lista, logo após ele troca o valor selecionado para seu lugar respectivo já ordenado, repetindo após este passo até toda a lista estar ordenada do modo desejado para sua ordenação.

Podemos observar o trecho do pseudocódigo do Selection Sort na figura 3.2 para entender melhor o seu funcionamento e sua lógica por trás do algoritmo: [UFG,2017]

SELECTION-SORT(A, n)		Tempo
1	**para** $i \leftarrow 1$ **até** $n-1$ **faça**	$\Theta(n)$
2	$min \leftarrow i$	$\Theta(n)$
3	**para** $j \leftarrow i+1$ **até** n **faça**	$\Theta(n^2)$
4	**se** $A[j] < A[min]$ **então** $min \leftarrow j$	$\Theta(n^2)$
5	$A[i] \leftrightarrow A[min]$	$\Theta(n)$

Analisando a Figura 3.2, observa-se que, o algoritmo do Selection Sort, é simples na construção com laços de repetições, com a sua busca feita de maneira ininterrupta sendo feita até terminar as repetições e seleções, tendo um custo pior para listas ordenadas de forma decrescente ou inversa, pelo alto gasto de tempo para a execução.

3.4 *Shell Sort*

O Shell Sort, é um algortimo proposto por *Donas Shell* em 1959, sendo uma extensão do algoritmo de ordenação por inserção. Por exemplo, a ordenação por inserção só troca itens adjacentes, para determinar o ponto de inserção São efetuadas n-1 comparações e movimentações quando o menor item está na última posição. O algortimo com o método de Shell, contorna este problema, permitindo trocas mais distantes. Tendo vantagens na sua utilização a depender do caso ou quantidade de números ou elementos. [UFMG, 2012]

O Shell Sort possuí patricidades em sua utilização, promovendo benefícios para seus usuários, podemos observar tais vantagens nos seguintes casos indicados abaixo:

1. **Melhoria na eficiência em relação ao Insertion Sort:** O Shell Sort é pode ter uma melhoria na eficiência em relação ao Insertion Sort. Ele divide a lista, em subgrupos e ordena esses subgrupos usando o Insertion Sort, trazendo uma aceleraração significativamente rápida no processo de ordenação. [CORMEN, 2009]

2. **Eficiente para listas de tamanho médio:** Pode-se observar a classificação Shell que funciona bem para listas de tamanho médio. Ele tem uma melhora a classificação por inserção, que é mais eficiente para listas pequenas, mas se torna menos eficaz à medida que o tamanho da lista aumenta. Usando o conceito de incremento decrescente, a classificação Shell pode ter um melhor desempenho do que a classificação por inserção para listas de tamanho médio. [SCOLARHAT, 2023]

Análise Matemática:

Analisando de forma matemática do Shell Sort, podemos considerar que até hoje não foi possível definir uma fórmula fechada extamente, pelo algortimo depender das lacunas em sua ordenação. [SCIELO, 2019]

Pode-se observar a complexidade de tempo do pior caso do Shell Sort com uma quantidade de lacunas do Shell original é O (n2), onde n é definido como número de elementos a serem classificados. Porem, tem muitas outras sequências de lacunas que podem alcançar melhor desempenho. Uma sequência de lacunas comumente usada é a sequência de Knuth, onde

a complexidade defendida pelo pior caso t(n) para qualquer sequência pode ser definida na fígura 4.1 abaixo: [SCIELO, 2019]

$$t(n) = O(n^2 \log k) = O(n^2 \log(\log^c n)) = O(n^2 \log \log n)$$

Análise Pseudo Código:

Analisando o método do Shell Sort, podemos perceber que o algortimo é eficaz porque move elementos maiores para o final da sequência mais rapidamente do que o algoritmo de inserção tradicional, reduzindo o número total de comparações e movimentações necessárias. O algortimo primeiramente define as lacunas (gaps) que são utilizadas para dividir o array, por uma forma já pré-definida, após começa-se com a maior lacuna e aplica-se o algoritmo de inserção a cada sub-sequência formada pela lacuna atual, para que elementos distintos possam assim ser comparados, continuando reduz a lacuna para a próxima menor e repita assim o processo de ordenação das sub-sequências. continua-se reduzindo a lacuna e ordenando até que a lacuna seja igual a 1, até que então, aplica-se o algoritmo de inserção uma última vez, onde com uma lacuna de 1 para garantir que todos os elementos sejam ordenados.

Pode-se observar o trecho do pseudocódigo do Shell Sort na figura 4.2 abaixo para entender melhor o seu funcionamento e sua lógica por trás do algoritmo: [SCIELO, 2019]

Dados: $V[1..n]$ *de inteiros, sequência de passos* $p_1, \ldots, p_k$
Resultado: $V[i] \leq V[i+1]$, *para todo* $1 \leq i < n$
para $j \leftarrow k$ **até** 1 **incremento** -1 **faça**
    para $i \leftarrow 1$ **até** p_j **faça**
        InsertionSort($S_{i,j}$), sendo $S_{i,j}$ o vetor que consiste dos elementos de V presentes nos índices $i, i+p_j, i+2p_j, \ldots, i+\left\lfloor \frac{n-i}{p_j} \right\rfloor p_j$

Analisando a Figura 4.2, pode-se observar toda a construção e o funcionamento base do Shell Sort, com uma pré definição de suas lacunas (gaps).

3.5 *Merge Sort*

O algortimo *Merge Sort* também conhecido por ordenação utilizando a mistura, idealizado por John von Neumann por volta de 1940, é um exemplo prático de algoritmo de ordenação por comparação do tipo dividir-para-conquistar. Onde em suma, consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). Podemos ver que o algortimo utiliza a recurção para fazer este processo, tendo ao final de todo desenvolvimento, seus pontos positivos e negativos para sua utilização. [CORMEN, 2012]

Por utilizar o método de Divisão e Conquista, percebos algumas vantens em sua utilização:

1. **Estabilidade em seu uso:** O Merge Sort é um algoritmo de ordenação estável, onde ele mantém a ordem relativa dos elementos com chaves iguais.

2. **Complexidade de tempo consistente:** O algoritmo possui uma complexidade de tempo de O(n log n) no pior caso, no melhor caso e no caso médio. Isso significa que ele é eficiente para uma ampla gama de tamanhos de entrada e tipos de dados. [CORMEN, 2009]

3. **Fácil de se implementar:** pode-se observar que ele é relativamente simples de entender e implementar. Sua estrutura recursiva e o processo de trocas fazem ser adequado para programação.

Análise Matemática:

Analisando a forma matemática do Merge Sort, podemos observar que o algoritmo tem um custo matemático para sua aplicação.

O Algoritmo pode ser representado com a etapa de divisão e conquista como:

1. Dividir: Calcula o ponto médio do sub-arranjo, o que demora um tempo constante.

2. Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho n/2, o que contribui com 2T(n/2) para o tempo de execução.

3. Combinar: Unir os sub-arranjos em um único conjunto ordenado, que leva o tempo.

Resultando estes processos chegamos a:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1 \\ 2T(n/2) + \Theta(n), & \text{se } n > 1 \end{cases}$$

Análise Pseudo Código:

O Merge Sort começa com a função MergeSort, que recebe como entrada um array arr, bem como os índices esquerdo l e direito r. Inicialmente, verifica se l é menor que r, o que é a condição de parada da recursão. Se verdadeiro, o algoritmo calcula o ponto médio m entre l e r. Em seguida, a função MergeSort é chamada recursivamente nas metades esquerda e direita do array (arr, l, m e arr, m + 1, r). O passo final é a chamada da função Merge para combinar as duas metades ordenadas em um único array ordenado, garantindo que os elementos permaneçam em ordem crescente durante o processo de fusão. A função Merge compara os elementos das duas metades e os combina de forma ordenada em um array temporário, que é posteriormente copiado de volta para o array original, resultando em um array totalmente ordenado. Esse processo é repetido até que todo o array esteja ordenado de maneira crescente. [CORMEN, 2009]

Podemos observar o trecho do pseudocódigo do Merge sort para entender melhor o seu funcionamento e sua lógica por trás do algoritmo:

Pseudo Código do Merge e Merge Sort:

```
01. merge(A[0...n - 1], inicio, meio, fim)
02. |   tamEsq ← meio - inicio + 1 //tamanho do subvetor esquerdo
03. |   tamDir ← fim - meio //tamanho do subvetor direito
04. |   inicializar vetor Esq[0...tamEsq - 1]
05. |   inicializar vetor Dir[0...tamDir - 1]
06. |   para i ← 0 até tamEsq - 1
07. |   |   Esq[i] ← A[inicio + i] //elementos do subvetor esquerdo
08. |   fim_para
09. |   para j ← 0 até tamDir - 1
10. |   |   Dir[j] ← A[meio + 1 + j] //elementos do subvetor direito
11. |   fim_para
12. |   idxEsq ← 0 //índice do subvetor auxiliar esquerdo
13. |   idxDir ← 0 //índice do subvetor auxiliar direito
14. |   para k ← inicio até fim
15. |   |   se(idxEsq < tamEsq)
16. |   |   |   se(idxDir < tamDir)
17. |   |   |   |   se(Esq[idxEsq] < Dir[idxDir])
18. |   |   |   |   |   A[k] ← Esq[idxEsq]
19. |   |   |   |   |   idxEsq ← idxEsq + 1
20. |   |   |   |   senão
21. |   |   |   |   |   A[k] ← Dir[idxDir]
22. |   |   |   |   |   idxDir ← idxDir + 1
23. |   |   |   |   fim_se
24. |   |   |   senão
25. |   |   |   |   A[k] ← Esq[idxEsq]
26. |   |   |   |   idxEsq ← idxEsq + 1
27. |   |   |   fim_se
28. |   |   senão
29. |   |   |   A[k] ← Dir[idxDir]
30. |   |   |   idxDir ← idxDir + 1
31. |   |   fim_se
32. |   fim_para
33. fim_merge
```

```
01. mergesort(A[0...n - 1], inicio, fim)
02. |   se(inicio < fim)
03. |   |   meio ← (inicio + fim) / 2 //calcula o meio
04. |   |   mergesort(A, inicio, meio) //ordena o subvetor esquerdo
05. |   |   mergesort(A, meio + 1, fim) //ordena o subvetor direito
06. |   |   merge(A, inicio, meio, fim) //funde os subvetores esquerdo e direito
07. |   fim_se
08. fim_mergesort
```

3.6 *Quick Sort*

O algortimo *Quick Sort*, idealizado por C.A.R. Hoare por volta de 1960, uma curiosidade do algoritmo é que, ele foi criado ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido. Foi publicado em 1962 após uma série de refinamentos, o algoritmo é classíficado como um algoritmo de ordenação não estável. [AZEREDO, 1996]

O algoritmo trabalha desta maneira: segundo um arranjo A contendo n números, ele escolhe um elemento como pivô (tradicionamelmente o elemento do meio) e divide o arranjo em duas partições, uma contendo elementos menores ou iguais ao pivô e a outra contendo elementos maiores que o pivô. Logo, o algoritmo é recursivamente aplicado a ambas as partições até que o arranjo esteja ordenado por um todo.

Podemos citar alguns exemplos de vantagens do Quick Sort:

1. **Eficiência média:** O Quick Sort é conhecido por sua eficiência média, uma vez que geralmente apresenta um desempenho muito bom na ordenação de grandes conjuntos de dados. Sua complexidade média de tempo é O(n log n), o que o torna mais rápido do que muitos outros algoritmos de ordenação, como o Bubble Sort ou o Selection Sort.

2. **Uso da memória:** O Quick Sort é um algoritmo de ordenação in-place, o que significa que ele não requer memória adicional significativa para armazenar dados intermediários durante o processo de ordenação. Isso o torna mais eficiente em termos de uso de memória em comparação com algoritmos que exigem espaço adicional para armazenar

cópias temporárias dos dados.

3. **Adaptação a conjuntos de dados parcialmente ordenados:** O Quick Sort é eficaz mesmo quando o conjunto de dados já está parcialmente ordenado. Isso ocorre porque o algoritmo divide o conjunto de dados em subconjuntos menores e os ordena independentemente. Isso evita a necessidade de realizar muitas operações de comparação quando os dados já estão parcialmente ordenados, tornando-o adequado para uma ampla gama de cenários.

Análise do Quick Sort:

Observando o Quick Sort, podemos observar que, ele tem algumas outras diferenças em relação ao merge sort. O quicksort funciona a partir da posição e seu tempo de execução no pior caso é tão ruim quanto o selection sort e o insertion sort:

$$\Theta(n^2)$$

Mas o seu tempo de execução médio é tão bom quanto o do merge sort:

$$\Theta(n \log_2 n)$$

Então, porque escolher as vezes o quicksort quando o merge sort é no mínimo tão bom quanto ele? Isso se deve porque o fator constante oculto na notação big-Θ para o quicksort é muito bom. pode-se falar que, o quicksort supera o merge sort e supera significativamente o selection sort e o insertion sort.

Análise Pseudo Código:

O processo começa com a função QuickSort, que recebe como entrada um array arr, juntamente com os índices primeiro (p) e último (u) que denotam a faixa de elementos a serem ordenados. A recursão é essencial para o funcionamento do Quick Sort e segue esta lógica: [CORMEN, 2009]

1. Verificação da Condição de Parada: Inicialmente, o algoritmo verifica se p é menor que u. Esta é a condição de parada da recursão. Se for verdadeira, o algoritmo continua.

2. Escolha do Pivô: O algoritmo escolhe um elemento do array como pivô.

3. Particionamento do Array: A função Partition é chamada para rearranjar os elementos do array de forma que os elementos menores que o pivô fiquem à esquerda, e os elementos maiores fiquem à direita. O índice do pivô, após a partição, é retornado.

4. Recursão nas Partições: O algoritmo chama recursivamente a função QuickSort para as duas partições resultantes. Uma partição contém os elementos menores que o pivô (arr, p, Partição - 1), e a outra contém os elementos maiores (arr, Partição + 1, u).

Pseudo Código da função Parciona com pivo no início e do Quick Sort:

```
01. particao(A[0...n - 1], inicio, fim) //partição de Hoare
02. |   pivo ← A[inicio] //o pivô é o primeiro elemento
03. |   i ← inicio + 1 //índice i faz a varredura da esquerda para direita
04. |   j ← fim índice j faz a varredura da direita para esquerda
05. |   enquanto(i ≤ j) //enquanto os índice não se ultrapassarem
06. |   |   enquanto(i ≤ j E A[i] ≤ pivo) //enquanto A[i] não for maior que o pivô
07. |   |   |   i ← i + 1
08. |   |   fim_enquanto
09. |   |   enquanto(i ≤ j E A[j] > pivo) //enquanto A[j] não for menor ou igual ao pivô
10. |   |   |   j ← j - 1
11. |   |   fim_enquanto
12. |   |   se(i < j)
13. |   |   |   trocar(A, i, j) //se os índices não se ultrapassarem, troque os elementos
14. |   |   fim_se
15. |   fim_enquanto
16. |   trocar(A, inicio, j) //coloca o pivô na posição de ordenação
17. |   retorne j;
28. fim_particao
```

```
01. quicksort(A[0...n - 1], inicio, fim) //Quicksort de Hoare
02. |   se(inicio < fim)
03. |   |   q = particao(A, inicio, fim) //realiza a partição
04. |   |   quicksort(A, inicio, q - 1) /ordena a partição esquerda
05. |   |   quicksort(A, q + 1, fim) //ordena a partição direita
06. |   fim_se
07. fim_quicksort
```

3.7 *Heap Sort*

O *Heap Sort*, possui o mesmo princípio de funcionamento da ordenação por seleção, sendo desenvolvido e criado por Robert W. Floyd e J.W.J Williams em 1964.
O algoritmo utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, quando chega ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap. ao observar o "Heap"vemos que ele pode ser representada como uma árvore (uma árvore binária com suas caracteristicas) ou como um vetor. Para uma ordenação decrescente, deve ser construída uma heap mínima (o menor elemento fica na raiz). Para uma ordenação crescente, deve ser construído uma heap máximo (o maior elemento fica na raiz). [CORMEN, 2012]

Pela sua Volatilidade, podemos observar que o Heap Sort possui algumas vantagens em sua utilização:

1. **Eficiência de Espaço:** Pode-se observar que ele tem uma eficiência de espaço O(1) no local. Isso significa que a quantidade de memória adicional necessária para executar o algoritmo é constante, independentemente do tamanho da entrada. Em comparação com o Merge Sort, que requer espaço adicional para armazenar temporariamente subconjuntos, o Heap Sort pode ser mais eficiente em termos de espaço. [CORMEN, 2009]

2. **Desempenho em Situações de Pior Caso:** O Heap Sort possui uma complexidade de tempo O(n log n) no pior caso, que é igual à complexidade do Merge Sort. No

entanto, em alguns casos práticos, o Heap Sort pode ter um desempenho melhor em situações de pior caso em comparação com outros algoritmos de ordenação, como Quick Sort. Isso pode ser benéfico em cenários onde a garantia de um tempo de execução consistente é crucial.

3. **Melhor Utilização de Memória:** O Heap Sort é mais eficiente em termos de cache de memória devido ao seu acesso sequencial e estrutura de dados baseada em árvore (heap). Em comparação com o Merge Sort, que envolve acessos não sequenciais a arrays, o Heap Sort pode resultar em menos falhas de cache e, portanto, ter um desempenho mais rápido em certas situações.

Análise Matemática e Pseudo Código:

Ao olhar pela complexidade de tempo do *Heap Sort* onde se manifesta que é O(n log n) no melhor, médio e pior caso, sendo eficiente até mesmo para grandes volumes de dados a ser tratados. No entanto, onde podemos ressaltar que a construção do Heap pode ser diferente dos outros algortimos, pela maneira onde o usuário trata o Heap em sua construção.
Dentro do algoritmo *Heap Sort* pode ter algumas funções heap, além das funções já usadas na dentro do algoritmo original, como o *buildHeap* e *Heapify*.

Observando a Construção do código do Heap Sort, temos que:

```
// Func. (HEAP_SORT).
double heap_sort(int* a, int n) {
    clock_t tempI, tempF;
    tempI = clock();

    build_min_heap(a, n);
    for (int i = n - 1; i > 0; i--) {
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        min_heapify(a, i, 0);
    }

    tempF = clock();
    double tempo = (tempF - tempI) / (double)CLOCKS_PER_SEC;
    cout << endl << "Tempo Heap Sort: " << tempo << endl;

    return tempo;
}
```

Figura 1: Código *Heap Sort*

A função *Heap Minimum*, retorna o elemento de menor ou maior valor no heap. Em um heap mínimo, o menor elemento está sempre na raiz do heap. :

```
void heap_minimum(int* a, int tamanho) {
    cout << "Vetor original:\n";
    imprimir_array(a, tamanho);
    cout << "Construindo min heap:\n";
    build_min_heap(a, tamanho);
    imprimir_array(a, tamanho);
    cout << "Elemento minimo: " << a[0] << "\n\n";
}
```

Figura 2: Código *Heap Minimum*

A função *Heap Extract Min*, é responsável por extrair o elemento mínimo de um heap mínimo e ajustar a estrutura do heap para mantê-lo válido:

```
void heap_extract_min(int* a, int& tamanho) {
    if (tamanho < 1) {
        cout << "Heap vazio. Não e possivel extrair o minimo.\n";
        return;
    }

    cout << "Vetor original:\n";
    imprimir_array(a, tamanho);

    cout << "Removendo o minimo...\n";
    int min = a[0];
    a[0] = a[tamanho - 1];
    tamanho--;
    min_heapify(a, tamanho, 0);

    cout << "Elemento minimo removido: " << min << "\n";
    cout << "Heap apos a extracao:\n";
    imprimir_array(a, tamanho);
    cout << "\n";
}
```

Figura 3: Código *Heap Extract Min*

A função *Heap Increase Key*, é usada para aumentar o valor de uma chave em um heap mínimo e, em seguida, ajustar a estrutura do heap para garantir que a propriedade de heap mínimo seja mantida:

```
void heap_extract_min(int* a, int& tamanho) {
    if (tamanho < 1) {
        cout << "Heap vazio. Não e possivel extrair o minimo.\n";
        return;
    }

    cout << "Vetor original:\n";
    imprimir_array(a, tamanho);

    cout << "Removendo o minimo...\n";
    int min = a[0];
    a[0] = a[tamanho - 1];
    tamanho--;
    min_heapify(a, tamanho, 0);

    cout << "Elemento minimo removido: " << min << "\n";
    cout << "Heap apos a extracao:\n";
    imprimir_array(a, tamanho);
    cout << "\n";
}
```

Figura 4: Código *Heap Increase Key*

4 ANÁLISE DE COMPLEXIDADE

4.1 *Insertion Sort*

Ao observamos o algoritmo Insertion Sort, podemos ver que temos casos que implicam diretamente em seu funcionamento, temos o Melhor caso, Pior caso e médio caso, todos estes casos tem uma maneira de relação com o algoritmo devido ao seu desenvolvimento como visto no pseudocódigo, devido a forma como os números ou itens são organizados, podemos classífica-los em cada tipo de caso existente.

O Insertion Sort tem a fórmula geral, onde a fórmula é:

$$c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{2}^{n} t_j + c_6 \sum_{2}^{n} (t_j - 1) + c_7 \sum_{2}^{n} (t_j - 1) + c_8(n-1).$$

Podemos definir cada caso como:

1. Melhor Caso: Ocorre quando o vetor está ordenado em sua estrutura, pois, ao contrário do pior caso, a condição A[j] ¿ elemento sempre será falsa. então, o código do laço interno nunca será executado. Assim, teremos apenas as n - 1 iterações do laço externo, portanto a complexidade no tempo é O(n). [CORMEN,2012]

   O melhor caso de complexidade do algoritmo é onde a entrada do algoritmo já está

totalmente ordenado de forma crescente.

$$t(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1).$$

A fórmula de melhor caso é simplificada para:

$$t(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (-c_2 - c_4 - c_5 - c_8).$$

Onde no final chega-se a:

$$t(n) = An + B.$$

2. Pior Caso: Ocorre quando os elementos do vetor estão em ordem decrescente, pois a condição A[j] ¿ elemento sempre será verdadeira. então, o laço interno realizará a quantidade máxima de iterações. Nesse caso, o Insertion Sort terá complexidade no tempo com O(n2). [CORMEN,2012]

$$\sum_{2}^{n} Tj = \frac{n(n+1)}{2} - 1 \tag{1}$$

$$\sum_{2}^{n} (Tj - 1) = \frac{n(n-1)}{2} \tag{2}$$

A fórmula geral do pior caso, se da por:

$$t(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

Onde após simplificada chega-se a:

$$t(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8).$$

Onde notamos que a complexidade do pior caso se dá por uma função quadrática, como citada no íncio:

$$T(n) = An^2 + Bn + C$$

3. Médio Caso: Ocorre quando temos uma média entre todos os casos, com valores aleatórios, sendo assim podemos observar que o médio caso tem valor respectivo, O(n^2). [CORMEN,2012]

   A fórmula do médio caso é:

$$t(n) = \left(\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{4} - \frac{c_6}{4} + \frac{c_7}{4} + c_8\right) n - (c_2 + c_4 - \frac{c_5}{2} - \frac{c_6}{2} + c_7).$$

Em suma, o Insertion Sort é um algortimo de ordenação que tem pontos altos e baixos em sua utilização, a depender dos itens ou números que serão listados para ordenação, se for uma lista já ordenada, teremos um melhor tempo de resposta, caso esteja com os elementos em ordem decrescente, teremos o pior caso, onde demandará mais tempo de execução e memória, e em geral, temos o médio caso, que é uma média entre o melhor e pior caso em seu tempo de execução e também de desenvolvimento para ser realizado, tendo elementos aleatórios para fazer a ordenação nescessária.

4.2 *Bubble Sort*

Ao observamos o algoritmo Bubble Sort, podemos observar que temos casos que implicam diretamente em seu funcionamento, temos o Melhor caso, Pior caso e Médio caso, todos estes casos possuem características que afetam ou beneficiam o desenvolvimento como visto no pseudocódigo, analisando a forma como o vetor ou array é organizado, podemos classífica-los em cada tipo de caso existente.

A fórmula geral inicial é:

$$c_1(n-1) + c_2 \sum_{2}^{n} t_j + c_3 \sum_{2}^{n} (t_j - 1) + c_4 \sum_{2}^{n} (t_j - 1)$$

Onde no final, podemos chegar que o termo geral de complexidade se define pelo custo:

$$t(n) = \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right) n^2 + \left(c_1 + \frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}+\right) n - (c_1 + \frac{c_2}{2}).$$

Podemos definir cada caso como:

1. Melhor Caso: O melhor caso ocorre quando a lista de entrada já está ordenada. Nesse cenário, o Bubble Sort fará apenas uma passagem pela lista para verificar se não há trocas possíveis e, em seguida, para. Portanto, o número de comparações é mínimo, e o número de trocas é zero. [DONALD E. KNUTH, 1998]

   Temos que com seu custo no melhor caso chega a:

$$t(n) = \left(\frac{c_2}{2} + \frac{c_3}{2}\right) n^2 + \left(c_1 + \frac{c_2}{2} + \frac{c_3}{2}\right) n - (c_1 + \frac{c_2}{2}).$$

2. Pior Caso: Pode-se observar o pior caso quando a lista de entrada está totalmente invertida, ou seja, em ordem decrescente, Nesse cenário, o Bubble Sort, fará com que o número máximo de comparações e trocadas em cada passagem pela lista, tornando-o caso mais ineficiente. [ROBERT L. KRUSE, 1997]

   Sua complexidade tem valor de O(n^2). Onde chega ao custo de:

$$t(n) = \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right) n^2 + \left(c_1 + \frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}+\right) n - (c_1 + \frac{c_2}{2}).$$

3. Médio Caso: O caso ocorre quando a lista de entrada está desordenada, e o Bubble Sort realizará várias passagens pela lista, trocando elementos até que a lista esteja completamente ordenada. [CORMEN,2009]

   O médio caso pode ser representado por:

$$t(n) = \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{4}\right) n^2 + \left(c_1 + \frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{4}+\right) n - (c_1 + \frac{c_2}{2}).$$

Podemos resaltar, que o Bubble Sort é um algortimo em tese simples, e com limitações

para altos números de trocas e verificações a serem feitas, como observado no pseudo código, observando que sua complexidade quadrática resulta em desempenho insatisfatório em comparação com outros algortimos de ordanação, que apresentam assim uma maior eficiência para realizar altos índices de buscas.

4.3 *Selection Sort*

O algoritmo Selection Sort, tem três casos ou situações que intereferem em seu tempo de execução e consumo de memória na sua utilização, os três casos mais influêntes são: *(Melhor Caso, Médio Caso e Pior Caso)*, podemos observar que os casos irão ter uma forte influência em seu desenvolvimento pois possuem características que afetam ou beneficiam o desenvolvimento como visto no pseudocódigo, analisando a forma como o vetor ou array é organizado.

O Termo Geral pode se dar por:

$$c_1 n + c_2(n-1) + c_3 \sum_{i=1}^{n-1} t_j + c_4 \sum_{i=1}^{n-1} (t_j - 1) + c_5 \sum_{i=1}^{n-1} (t_j - 1) + c_6(n-1) + c_7(n-1) + c_8(n-1).$$

Podemos definir cada caso como:

1. Melhor Caso: O melhor caso ocorre quando a lista de entrada já está completamente ordenada. Nesse caso, o Selection Sort ainda precisará percorrer todos os elementos da lista para verificar se eles estão no lugar certo, mas não precisará fazer nenhuma seleção ou troca, pois a lista já está ordenada. [CORMEN,2009]

   Podemos observar o cálculo a seguir:

$$\sum_{2}^{n} j = \frac{n(n+1)}{2} - 1 \tag{3}$$

$$\sum_{2}^{n} (j-1) = \frac{n(n-1)}{2} \tag{4}$$

   Colocando no termo geral temos que:

$$c_1 n + c_2(n-1) + c_3 \frac{n(n+1)}{2} - 1 + c_4 \frac{n(n-1)}{2} + c_5 \frac{n(n-1)}{2} + c_6(n-1)$$

   Chegando ao final, temos que: T(n) = an^2 + bn + c

2. Pior Caso: O pior caso do Selection Sort ocorre quando a lista de entrada está completamente ordenada de forma inversa, ou seja, em ordem decrescente. Nesse caso, o Selection Sort terá que fazer o máximo de comparações e seleções. [CORMEN,2009].

   O cálculo do pior caso tem uma semelhança com o termo geral, porem vemos modificação no c3:

$$t(n) = \left(\frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2}\right) n - (c_2 + c_3 + c_6).$$

   Teremos a mesma equação do melhor caso no final:

   T(n) = an^2 + bn + c

3. Médio Caso: O caso médio do Selection Sort é geralmente semelhante ao seu pior caso. Isso ocorre porque, independentemente da entrada, o algoritmo deve percorrer toda a lista em busca do elemento mínimo e fazer as trocas. [URFPE, 2020].

   Podemos ver que temos na complexidade:

$$t(n) = (c_3)\, n^2 + \left(c_1 + c_2 + \frac{c_3}{2} + \frac{c_4}{2} + c_5\right) n - (c_2 + \frac{1}{2}c_3 + c_4 + c_5 + c_6)$$

   E mais uma vez chegaremos a equação que se resulta em:

   T(n) = an^2 + bn + c

Em suma, o Selection Sort é um algortimo em com um conceito simples, com limitações para altos números de seleções e verificações (Pior caso) a serem feitas, como observado no pseudo código, sendo indicado para introdução no estudo de algortimos e também desenvolvimento inicial, não sendo recomendado para altos índices de uso de dados.

4.4 *Shell Sort*

O algoritmo Shell Sort, tem três casos ou situações que intereferem em seu tempo de execução e consumo de memória na sua utilização, os três casos mais influêntes são: *(Melhor Caso, Médio Caso e Pior Caso)*, podemos observar que os casos irão ter uma forte influência em seu desenvolvimento pois possuem características que afetam ou beneficiam o desenvolvimento como visto no pseudocódigo na fígura 4.2, analisando a forma como é organizado.

O *Shell Sort* não possue uma complexidade definida, pois seu cálculo é bastante complexo, porém, podemos analisar cada caso como:

1. Melhor Caso: O melhor caso ocorre quando a lista de entrada já está parcialmente ordenada, sendo nescessários poucas trocas. Nesse cenário, o Shell Sort pode ter um desempenho próximo de O(n), onde "n"é o tamanho da lista. Isso ocorre porque as lacunas (gaps) utilizadas no algoritmo permitem que os elementos se movam para suas posições corretas de forma mais eficiente do que em algoritmos de ordenação quadráticos, como o Bubble Sort ou o Selection Sort. [CORMEN,2009]

2. Pior Caso: O pior caso do Shell Sort ocorre quando a lista de entrada está totalmente desordenada. No entanto, mesmo no pior caso, o Shell Sort tende a superar os algoritmos de ordenação quadráticos, dependendo das lacunas escolhidas. Portanto, em cenários adversos, o Shell Sort ainda é uma escolha melhor do que algoritmos de ordenação quadráticos em termos de desempenho. [R. SEDGEWICK,2011]

3. Médio Caso: O caso médio do Shell Sort é mais difícil de analisar matematicamente pois, à dependência das lacunas escolhidas e à falta de uma fórmula geral para calcular sua complexidade exata. Porém, na prática, o Shell Sort tende a ter um desempenho melhor do que os algoritmos de ordenação quadráticos, tornando-o uma opção eficaz para grandes volume de dados.

Em suma, o Shell Sort é um algortimo que, mesmo no pior caso, oferece desempenho superior aos algoritmos de ordenação quadráticos. Seu desempenho pode ser ainda melhor em cenários de dados parcialmente ordenados.

4.5 *Merge Sort*

O algoritmo Merge Sort, tem três casos ou situações que intereferem em seu tempo de execução e consumo de memória na sua utilização, os três casos mais influêntes são: *(Melhor Caso, Médio Caso e Pior Caso)*, onde podemos observar que os casos irão ter uma forte influência e impacto em seu desenvolvimento pois possuem características que afetam ou beneficiam o como visto no pseudocódigo, analisando a forma como o método de divisão e conquista é implementado.

Podemos definir a base como:

Complexidade de tempo: $\Theta(n \log_2 n)$.
Complexidade de espaço: $\Theta(n)$.

O Tempo do Algortimo se define por:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1 \\ 2T(n/2) + \Theta(n), & \text{se } n > 1 \end{cases}$$

A complexidade do tempo é:

$$T(n) = 2^1 . T\left(\frac{n}{2^1}\right) + 1.n$$
$$T(n) = 2^1 . \left(2.T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n = 2^2 . T\left(\frac{n}{2^2}\right) + 2.n$$
$$T(n) = 2^2 . \left(2.T\left(\frac{n}{2^3}\right) + \frac{n}{4}\right) + 2.n = 2^3 . T\left(\frac{n}{2^3}\right) + 3.n$$
$$\vdots$$
$$T(n) = 2^{h-1} . \left(2.T\left(\frac{n}{2^h}\right) + \frac{n}{2^{h-1}}\right) + h.n = 2^h . T\left(\frac{n}{2^h}\right) + h.n$$

Para que $T(1) = T\left(\frac{n}{n}\right) = \Theta(1)$ teremos que fazer com que $2^h = n \Rightarrow \lg 2^h = \lg n \Rightarrow h = \lg n$

Então:

$$T(n) = 2^h . T\left(\frac{n}{2^h}\right) + n.h = n.T\left(\frac{n}{n}\right) + n.\lg n = n.T(1) + n.\lg n = n.\Theta(1) + n.\lg n = \Theta(n.\lg n)$$

Após analisarmos a complexidade do tempo, podemos definir cada caso como:

1. Melhor Caso: O melhor caso ocorre quando as duas listas de entrada já estão ordenadas. Nesse cenário, o merge é muito eficiente, pois pode simplesmente mesclar as duas listas sem a necessidade de reordenar os elementos. A complexidade de tempo é O(n), onde "n"é o tamanho total das listas de entrada. [CORMEN,2009]

2. Pior Caso: O pior caso ocorre quando as duas listas de entrada estão em ordem decrescente. Nesse cenário, o merge deve comparar e reorganizar todos os elementos, tornando-o menos eficiente. A complexidade de tempo no pior caso também é O(n log n), mas com um fator de constante maior do que o caso médio.

3. Médio Caso: O caso médio é o que normalmente ocorre na prática. As duas listas de entrada não estão completamente ordenadas, mas há alguma ordem parcial. O merge sort divide repetidamente a lista em duas partes e mescla cada parte, o que resulta em uma complexidade de tempo médio de O(n log n).

Podemos cloncluir que, o desempenho do Merge Sort depende fortemente do estado das listas de entrada. O merge é especialmente eficiente quando as listas já estão ordenadas, como no melhor caso. No entanto, o merge sort, no geral, é um algoritmo de ordenação eficaz e estável, com complexidade média de O(n log n).

4.6 *Quick Sort*

O Quick Sort, possuí três casos ou situações que intereferem em seu tempo de execução e apontam diferentes impactos também no consumo de memória na sua execução, os três casos mais influêntes são: *(Melhor Caso, Médio Caso e Pior Caso)*, podemos observar que os casos irão ter pequenas ou grandes impactos comparando se um por um para sua utilização observando principalmente do ponto de vista matemático.

Desenvolvendo os três casos, obtemos:

1. Melhor Caso: O melhor caso ocorre quando a entrada já está completamente ordenado. Nesse caso, o algoritmo escolhe pivôs que dividem o arranjo em duas partes aproximadamente iguais, tendo uma ampla eficiênia em seu uso, sobre a análise de complexidade é, definida pelo tempo gasto com os pivôs e também nas chamadas recursivas. T(n) = 2T($\frac{n}{2}$)+$\Theta(n)$

   Após resolver está recorrência, chegamos a uma complexidade de tempo: $\mathcal{O}(n \log n)$

2. Pior Caso: O pior caso ocorre quando a lista de entrada já está ordenado de forma decrescente(inversa) onde assim resulta a escolhas pessímas de pivôs.

   Neste caso, podemos ver que o algortimo pode-se resultar na complexidade de: T(n)=T(n-1)+$\Theta(n)$.

   Resolvendo a recorrência, chegamos a complexidade de: $\mathcal{O}(n^2)$.

3. Médio Caso: O caso médio do Quick Sort é mais difícil para defirmos, pois o caso médio depende totalmente da escolha dos pivôs do algortimo, mas em suma, podemos observar no caso médio, a escolha do pivô na metade do arranjo ou lista, onde então teremos as comparações dos lados esquerdo e direito do pivô em partes iguais em tese.

   Podemos observar o custo seguinte de tempo na complexidade: $\mathcal{O}(n \log n)$

O Quick Sort com sua complexidade de tempo médio O(n log n) em cenários típicos, tornando-o uma escolha sólida para grande parte das aplicações.

4.7 *Heap Sort*

Ao Observamos o *Heap Sort*, pode-se ter uma analise de sua complexidade, onde o algortimo tem complexidade de $\mathcal{O}(n \log n)$ para os casos médio, pior caso e melhor caso, A complexidade O(n log n) torna o Heap Sort eficiente para grandes conjuntos de dados, sendo bom para vetores ou ordem de números acima de 100.000 itens ou mais.

Ao observar a estrutura do Heap Sort, temos que:

`Heap Sort(A)`	Custo	Vezes
1 — **constroi**(A)	C1	$\mathcal{O}(n \log n)$
2 — **For** $i \leftarrow$ comprimento[A] **down to** 2	C2	$\mathcal{O}(n)$
3 — **do** trocar $A[1] \leftrightarrow A[i]$	C3	$\mathcal{O}(n)$
4 — tamanho-do-heap[A] $\leftarrow$ tamanho-do-heap[A] $- 1$	C4	$\mathcal{O}(n)$
5 — **VerificaMin**(A, i)	C5	$\mathcal{O}(\log n)$

O algortimo possui a função *BuildHeap*, nesta função temos que o custo equivale a:

`Constroi(A)`	Custo	Vezes
1 — tamanho-do-heap[A] $\leftarrow$ comprimento[A]	C1	$\mathcal{O}(1)$
2 — **For** $i \leftarrow$ comprimento[$A/2$] **down to** 1	C2	$\mathcal{O}(n)$
3 — **do** VerificaMin(A, i)	C3	$\mathcal{O}(\log n)$

Analisando a todo o custo, chegamos ao final, totalizando a seguinte recorrência:

$$T(n) = T\left(\frac{2}{3}n\right) + O(1) \tag{5}$$

Pode-se definir nesta recorrência, utilizando o Teorema Mestre, para chegar a tal resolução:

$Start = a = 2,\ b = \frac{2}{3},\ n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = 0(1)$
$Question = \frac{f(n)}{n^{\log_b a}} = \frac{1}{1} = 1$

Chegando ao final como:
$f(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(1\right)$

Resolvendo-se em:
$T(n) = \Theta\left(n^{\log_b a} * \log n\right) = \Theta\left(\log n\right)$

5 TABELA E GRÁFICO

5.1 *Insertion Sort*

INSERTION SORT

PROJETO DE ALGORITMOS

TIPOS	10	100	1000	10000	100000	1000000	SEGUNDOS
CRESCENTE	0	0	0	0	0.001	0.002	
DECRESCENTE	0	0	0.001	0.102	10.334	1046.39	
ALEATÓRIO	0	0	0	0.051	5.163	520.315	

Figura 5: Tabela de tempo por segundo do algoritmo Insertion Sort

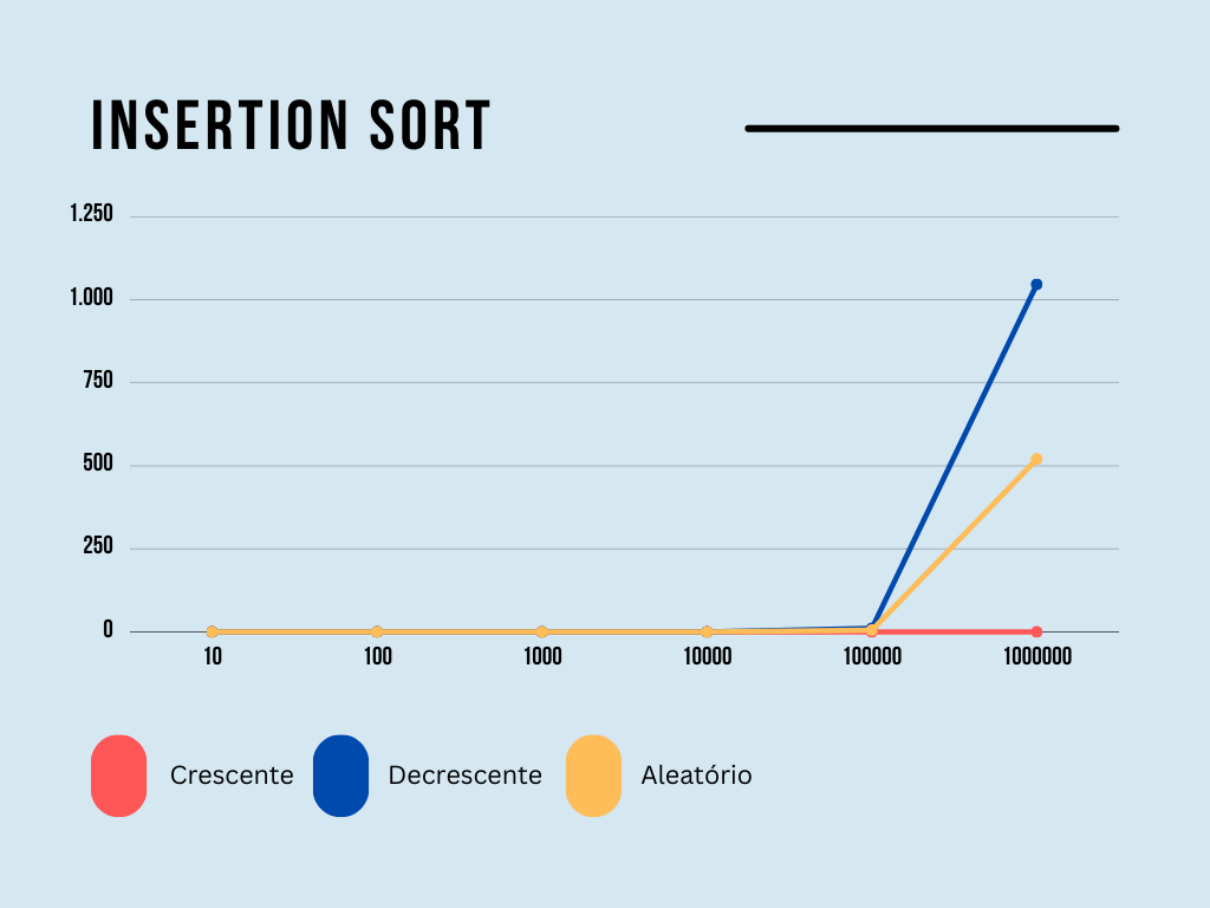


Figura 6: Gráfico de tempo por segundo do algoritmo Insertion Sort

Ao observar a tabela e gráfico do algoritmo Insertion Sort, pode-se perceber claramente a evidência dos três casos: melhor, pior e médio caso. Resalta-se que o tempo de execução tem forte influência devido as configurações de hardware e peças do computador utilizado para o cálculo de exercício.

No Melhor caso, o tempo de execução sai de 0 segundos somente com 100.000 números, enquanto no pior caso já com 1000 números para ordenação já percebe-se a alteração no tempo de execução, provando-se também que o médio caso é um caso médio entre o melhor e pior caso respectivamente, pois teve seu valor alterado somente com 10.000 números e com uma pequena diferença no tempo de execução do algortimo exposto.

5.2 *Bubble Sort*

BUBBLE SORT

PROJETO DE ALGORITMOS

TIPOS	10	100	1000	10000	100000	1000000	SEGUNDOS
CRESCENTE	0	0	0.001	0.087	8.833	884.23	
DECRESCENTE	0	0	0.002	0.172	17.307	1734.13	
ALEATÓRIO	0	0	0.001	0.215	23.863	2414.33	

Figura 7: Tabela de tempo por segundo do algoritmo Bubble Sort

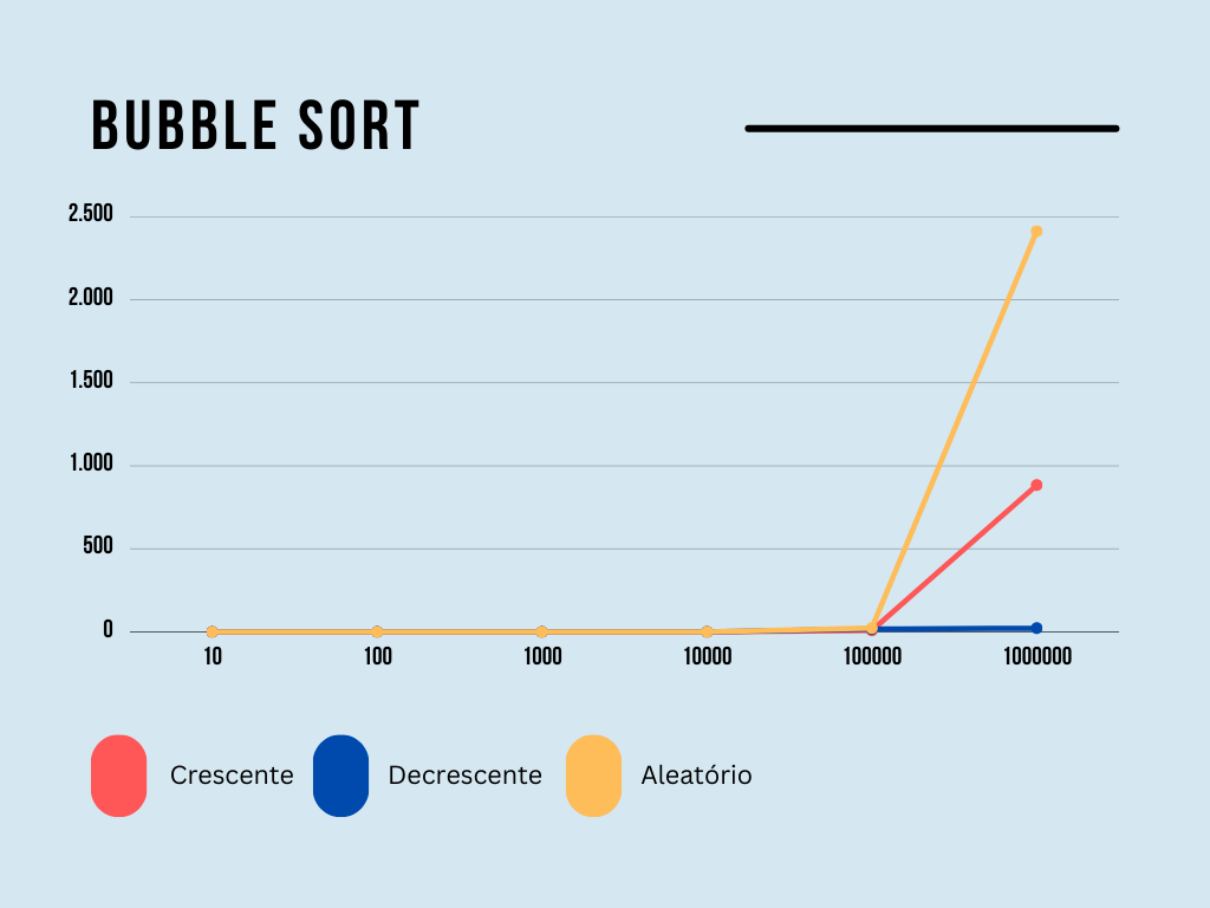


Figura 8: Gráfico de tempo por segundo do algoritmo Bubble Sort

Ao observar a tabela e gráfico do algoritmo Bubble Sort (Método da Bolha), pode-se perceber claramente a evidência dos três casos: melhor, pior e médio caso. cada um com suas particularidades, resalta-se que o tempo de execução tem forte influência devido as configurações de hardware e peças do computador utilizado para o cálculo de exercício.

No Melhor caso, o tempo de execução sai de 0 segundos somente com 1.000 números, enquanto no pior caso também com 1.000 números, provando a téoria do método de bolha do algortimo Bubble Sort, onde como visto em seu pseudo código. Para a ordenação de 1.000.000 de elementos, já se percebe-se a alteração, onde com o tempo de execução, o médio caso (aleatório) passa a ter um maior custo de tempo, e o menor custo de tempo com o melhor caso (crescente).

5.3 *Selection Sort*

SELECTION SORT

PROJETO DE ALGORITMOS

TIPOS	10	100	1000	10000	100000	1000000	SEGUNDOS
CRESCENTE	0	0	0.001	0.088	8.723	877.239	
DECRESCENTE	0	0.001	0.001	0.086	8.828	883.146	
ALEATÓRIO	0	0	0.001	0.087	8.725	867.139	

Figura 9: Tabela de tempo por segundo do algoritmo Selection Sort

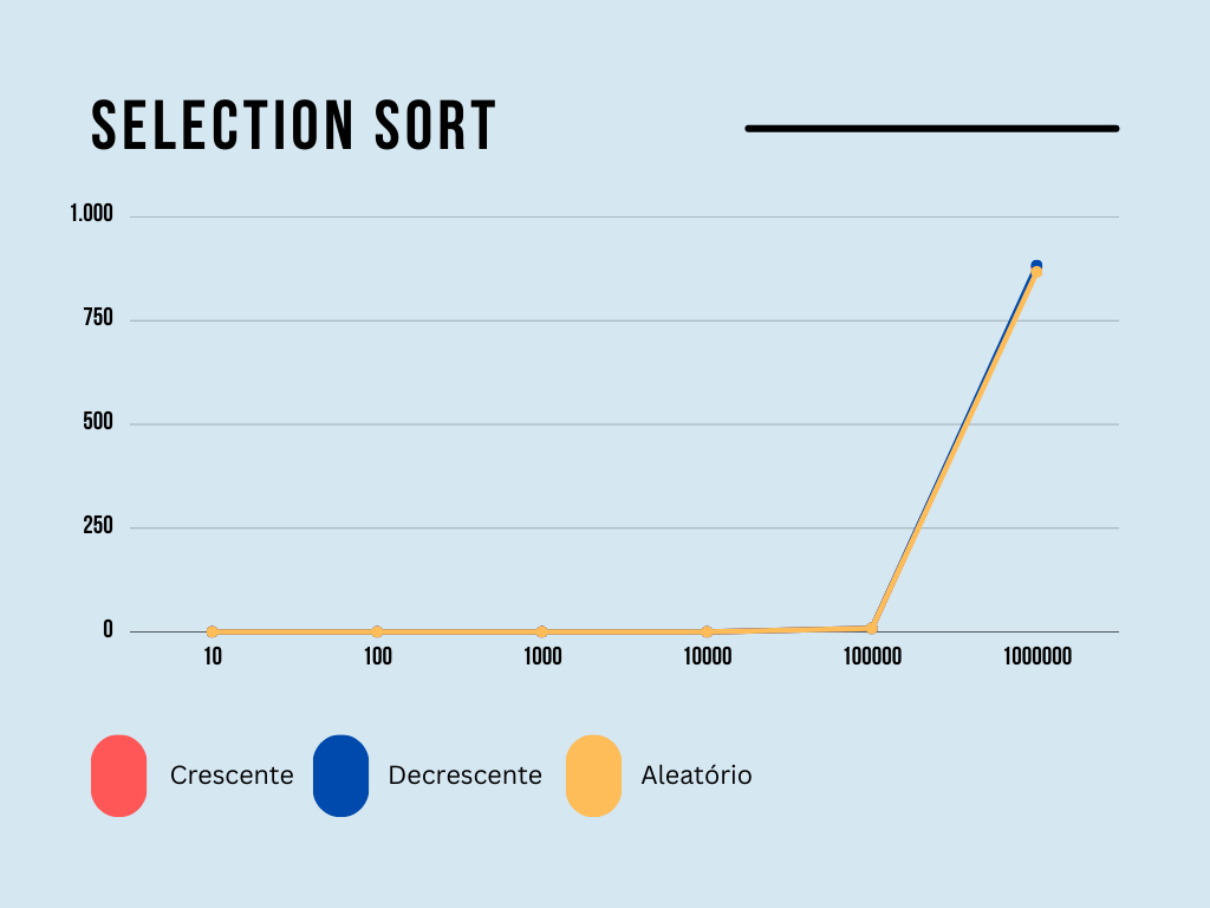


Figura 10: Gráfico de tempo por segundo do algoritmo Selection Sort

Ao observar a tabela e gráfico do algoritmo Selection Sort, pode-se perceber claramente a evidência dos três casos: melhor, pior e médio caso. Resalta-se que o tempo de execução tem forte influência devido as configurações de hardware e peças do computador utilizado para o cálculo de exercício.

No Melhor caso, o tempo de execução sai de 0 segundos somente com 1.000 números, enquanto no pior caso já com 100 números para ordenação já percebe-se a alteração no tempo de execução, todos três casos possuem tempo de custo parecido com 1.000.000 de elementos nos três casos, mas ainda assim, no pior caso (decrescente) obtem-se o pior tempo de execução do algortimo.

5.4 *Shell Sort*

SHELL SORT

PROJETO DE ALGORITMOS

TIPOS	10	100	1000	10000	100000	1000000	SEGUNDOS
CRESCENTE	0	0	0	0	0.005	0.06	
DECRESCENTE	0	0	0	0	0.008	0.083	
ALEATÓRIO	0	0	0	0.001	0.021	0.299	

Figura 11: Tabela de tempo por segundo do algoritmo Shell Sort

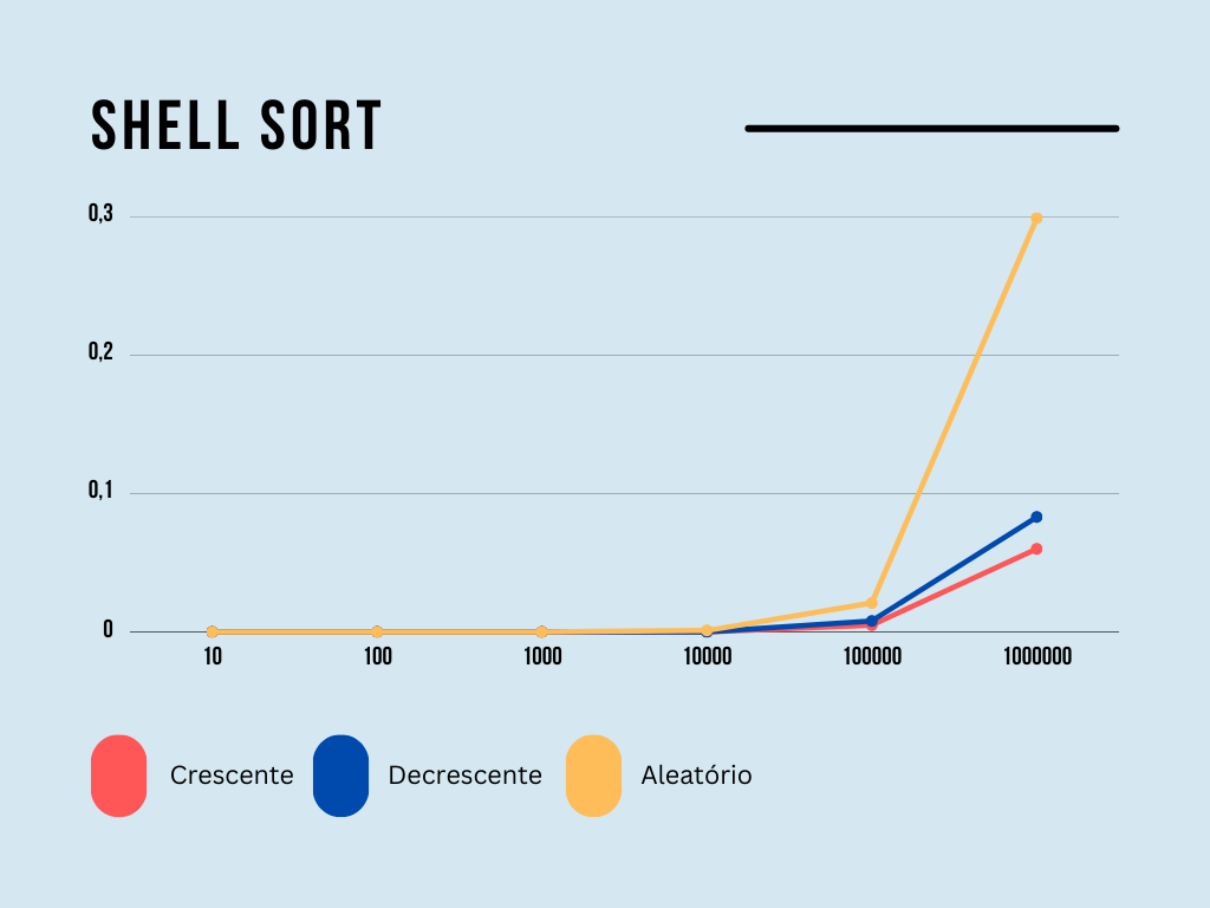


Figura 12: Gráfico de tempo por segundo do algoritmo Shell Sort

Ao observar a tabela e gráfico do algoritmo Shell Sort, pode-se perceber claramente a vantagem no seu custo de ordenação nos três casos: melhor, pior e médio caso. Resalta-se que o tempo de execução sofre forte influência devido as configurações de hardware e peças do computador utilizado para o cálculo de exercício.

No Melhor caso, o tempo de execução sai de 0 segundos somente com 100.000 números, também no pior caso somente com 100.00 elementos percebe-se a alteração no tempo de execução, e o médio caso, tem-se alteração já com 10.000, sendo os elementos aleatórios os mais demorados e com maior consumo de dados entre os três casos, observa-se também que comparado aos demais algortimos, podemos ver que o Shell Sort, possuí um melhor desempenho total em todos os casos em comparação, devido ao seu modo de execução demonstrado com seu pseudo código, tendo principamente ótimos resultados com altas sequências númericas como a de 1.000.000 de elementos.

5.5 *Merge Sort*

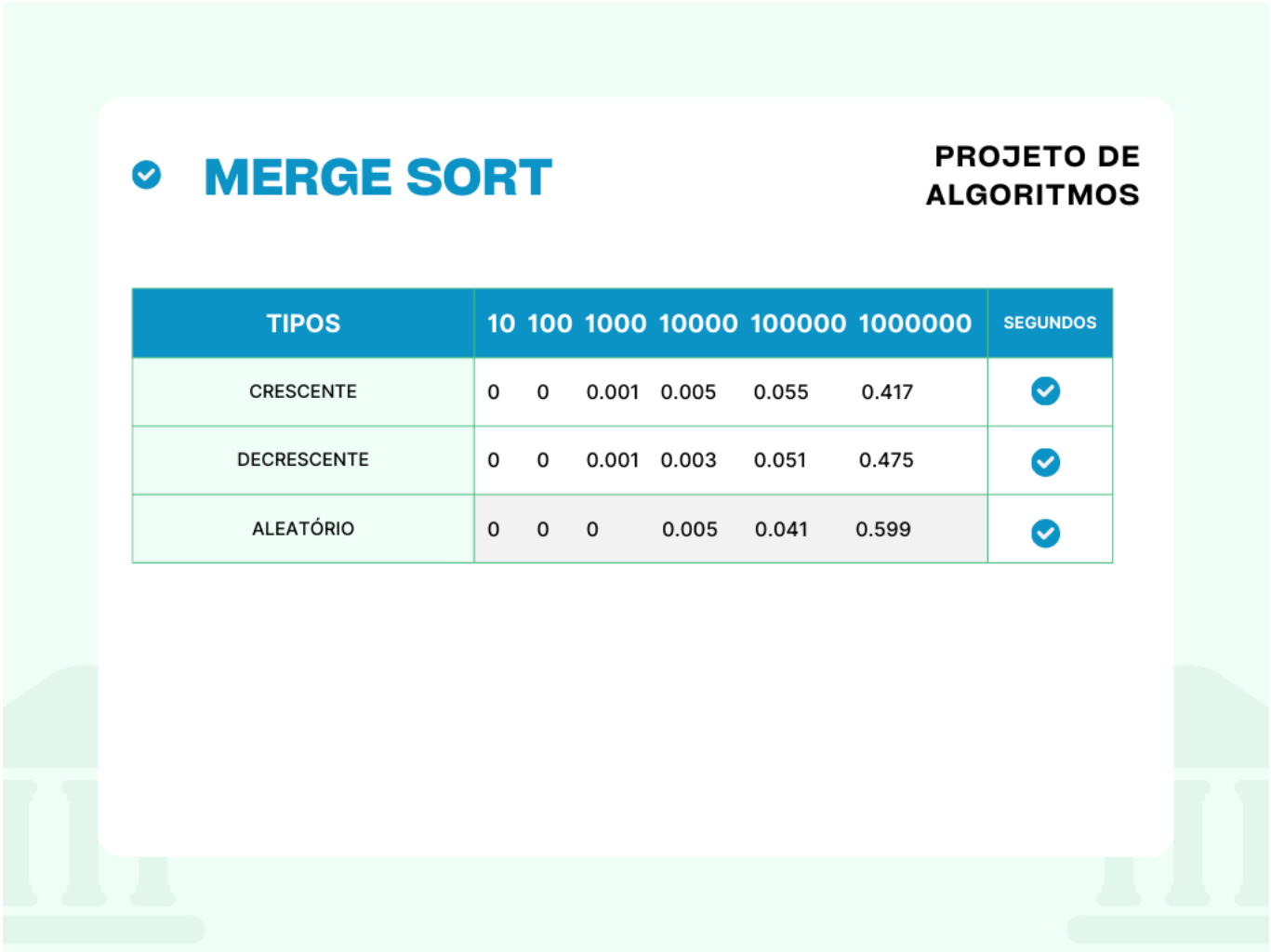


Figura 13: Tabela de tempo por segundo do algoritmo Merge Sort

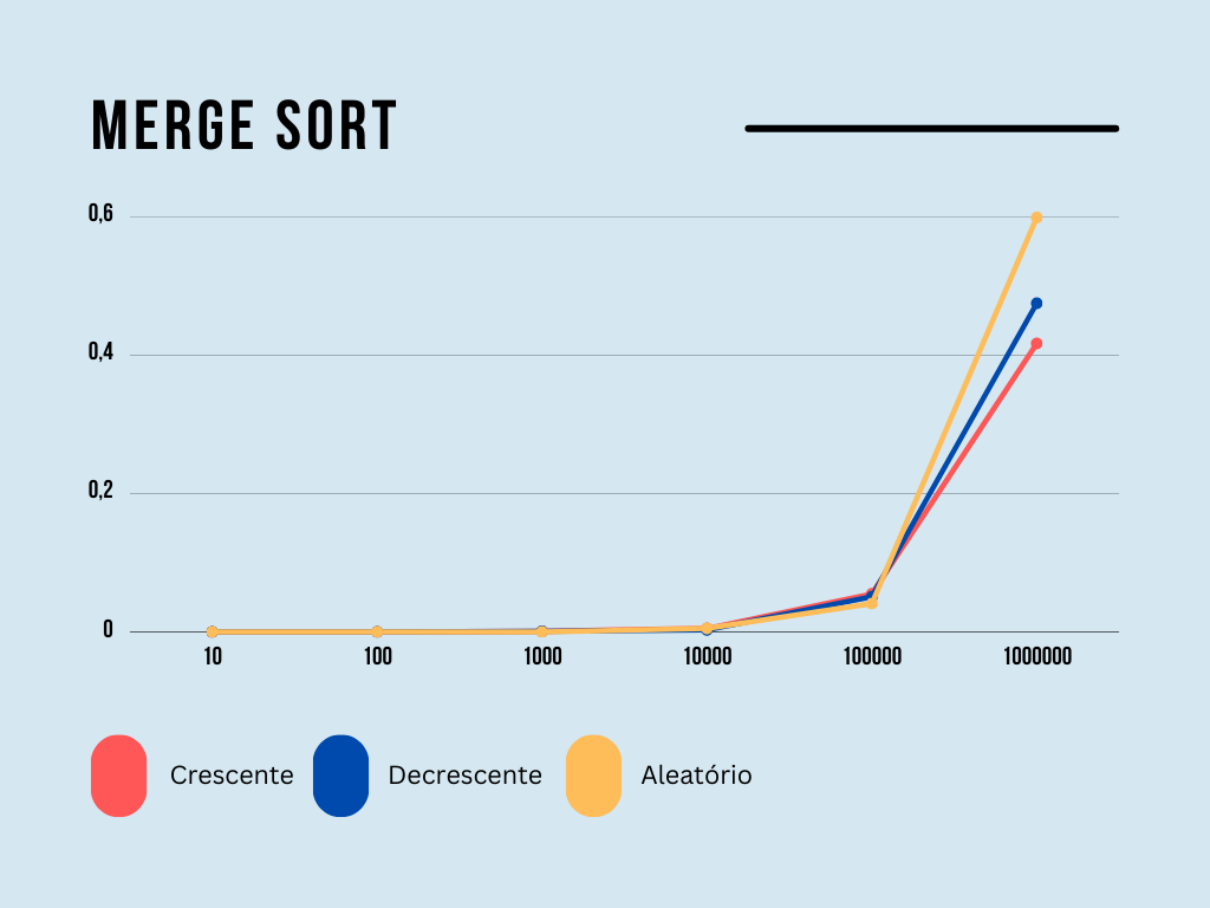


Figura 14: Gráfico de tempo por segundo do algoritmo Merge Sort

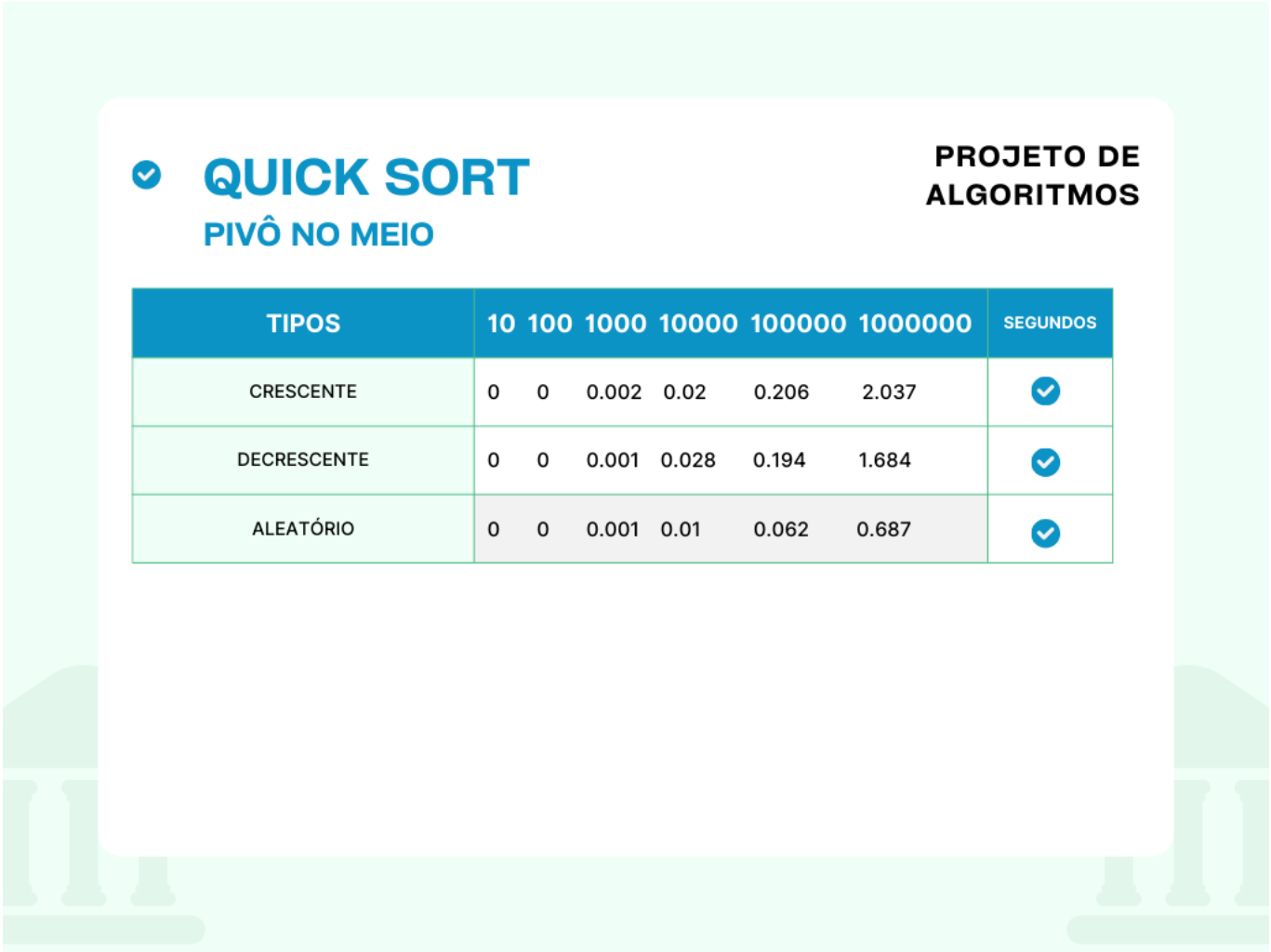
Ao verificar o Merge Sort, podemos ver resultados ótimos em seu tempo de execução, especialmente ao considerar 1000 elementos. No caso aleatório, a ordenação é tão eficiente que ocorre em 0 segundos, observando a capacidade do Merge Sort em resolver o problema de modo rápido comparado a outros algortimos de ordenação que vimos como o Insertion, Selection, Bubble e outros mais.

Ao verificar a ordenação para 1 milhão de elementos no cenário aleatório, o Merge Sort continua a sua eficácia, com um tempo de 0.599 segundos. Esse desempenho se mostra em diferentes tamanhos de conjuntos destaca a estabilidade do algoritmo, sem olhar a complexidade que podemos encontrar.

Além disso, ao colocar o Merge Sort em conjuntos já ordenados, seja de forma crescente ou decrescente, observamos que o algoritmo mantém uma eficiência notável. Com 1 milhão de elementos, a ordenação crescente ocorre em 0.417 segundos, enquanto a ordenação decrescente leva 0.475 segundos. Esses resultados mostram a força do Merge Sort a diferentes tamanhos e tipos de dados, garantindo um desempenho consistente em várias situações.

Em resumo, o Merge Sort destaca-se pela eficiência em situações diversas, proporcionando resultados bons, especialmente em cenários aleatórios e em conjuntos extensos de 1 milhão de elementos para ordenar. Sua capacidade de lidar com dados ordenados de modo crescente e decrescente tem uma eficácia forte e rapída do algortimo.

5.6 *Quick Sort*



TIPOS	10	100	1000	10000	100000	1000000	SEGUNDOS
CRESCENTE	0	0	0.002	0.02	0.206	2.037	
DECRESCENTE	0	0	0.001	0.028	0.194	1.684	
ALEATÓRIO	0	0	0.001	0.01	0.062	0.687	

Figura 15: Tabela de tempo por segundo do algoritmo Quick Sort

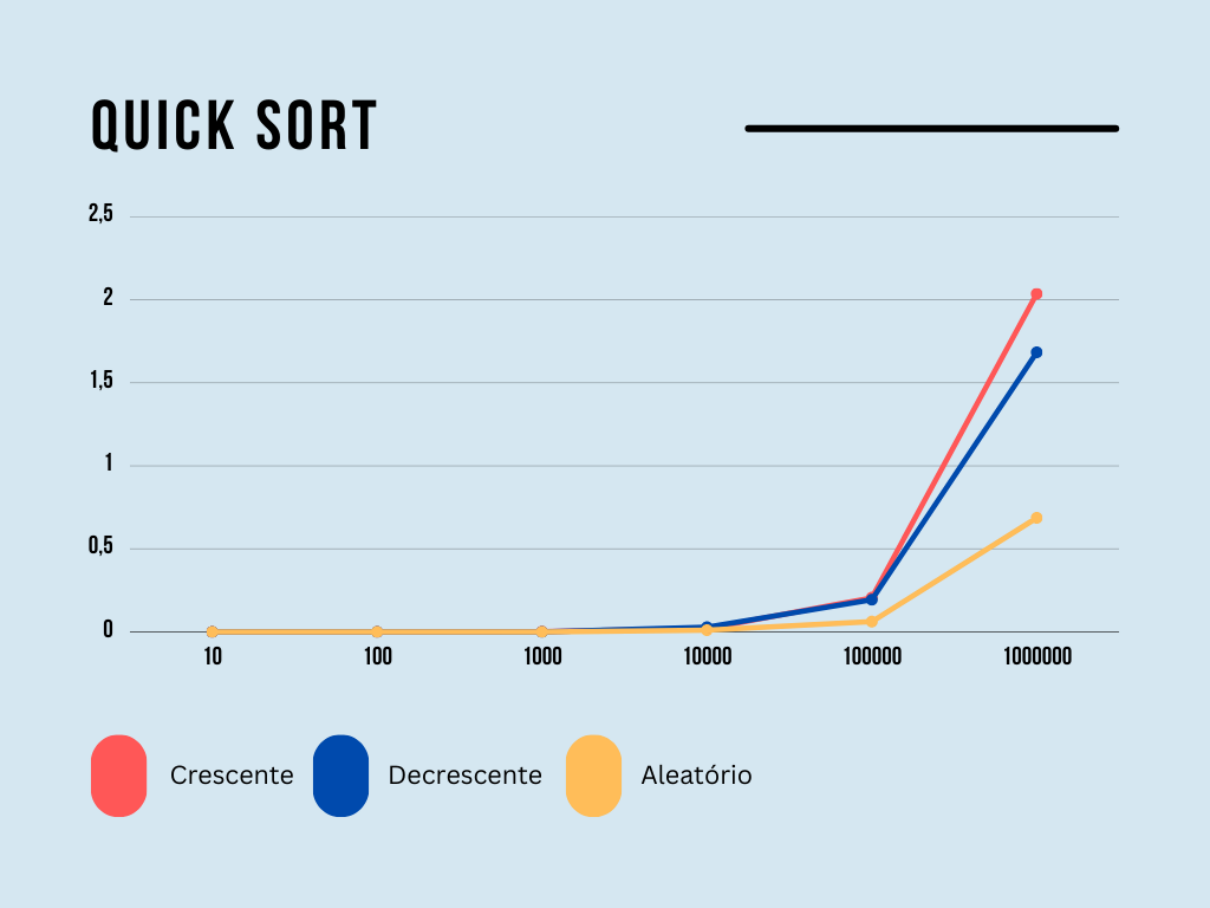


Figura 16: Gráfico de tempo por segundo do algoritmo Quick Sort

Ao analisar o algoritmo Quick Sort com a escolha do pivô no meio, vemos a influência de três cenários fundamentais: o melhor, o pior e o caso médio. É importante destacar que o desempenho está intimamente ligado às características do hardware e às configurações específicas do computador empregado nos testes.

No melhor caso, o Quick Sort com pivô no meio demonstra eficácia ao apresentar tempos de execução muito rápido, mesmo diante de um volume grande de dados, como 100.000 números. Essa seleção do pivô contribui para uma redução do tempo de execução, em condições otimizadas.

No pior caso, onde já está ordenado, vemos e observamos uma variação pequena no tempo de execução ao trabalhar com um conjunto relativamente pequeno de dados, como 1000 números para ordenação. Contudo, a escolha do pivô no meio contribui para ajudar algumas das variações extremas ao ordenar no Quick Sort.

O caso médio, mostra como um ponto de equilíbrio entre o melhor e o pior caso. O tempo de execução começa a subir um pouco apenas quando a quantidade de dados atinge a ordem de 10.000 números, mas ainda assim apresenta um ótimo tempo de ordenação. Nesse caso, vemos uma diferença mínima no tempo de execução do algoritmo, confirmando assim que o caso médio é uma situação intermediária.

É relevante mostrar que a escolha do pivô no meio no Quick Sort, exerce uma influência positiva na eficiência do algoritmo em diversos casos, tendo uma distribuição mais regular dos dados analisados.

5.7 *Heap Sort*

HEAP SORT

PROJETO DE ALGORITMOS

TIPOS	10	100	1000	10000	100000	1000000	SEGUNDOS
CRESCENTE	0	0	0	0.002	0.026	0.261	
DECRESCENTE	0	0	0	0.002	0.018	0.282	
ALEATÓRIO	0	0	0	0.002	0.028	0.376	

Figura 17: Tabela de tempo por segundo do algoritmo Heap Sort

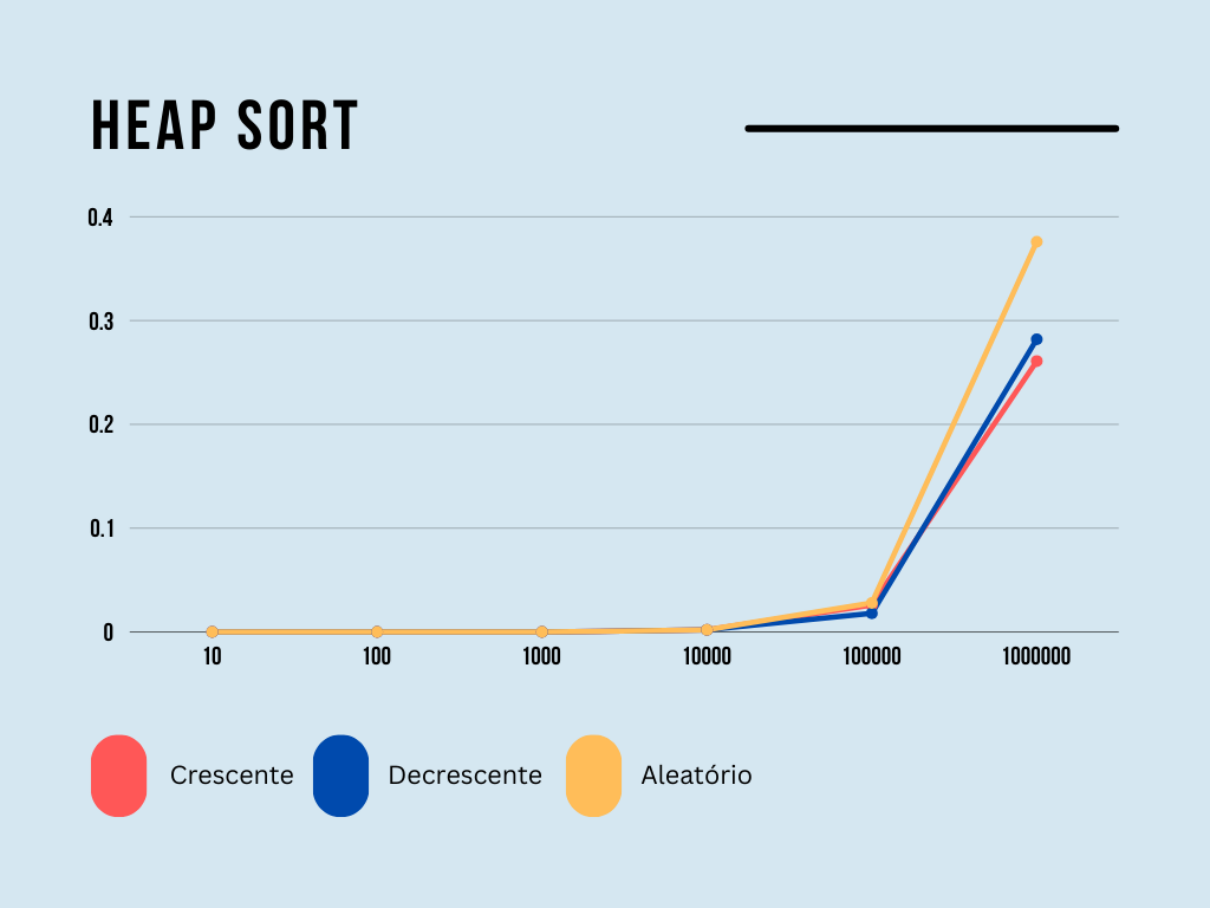


Figura 18: Gráfico de tempo por segundo do algoritmo Heap Sort

O *Heap Sort* é um algoritmo de ordenação eficiente que utiliza a estrutura de dados conhecida como heap. Sua notável performance em diversos cenários torna-o uma escolha robusta para a ordenação de conjuntos de dados, especialmente em situações onde a eficiência é crucial.

O Heap Sort destaca-se pela rapidez e pela capacidade de manter uma ordenação consistente. Ao observamos o gráfico 18, podemos ver que o algoritmo se demonstra estável em seu tempo comparado entre os três modos, (crescente, decrescente e aleatôrio) devido a sua complexidade matemática ser a mesma para os três, como por exemplo com 10.000 números ou elementos, pode-se perceber uma diferença ao chegar-se a 1.000.000 de elementos, porém ainda assim, não se tem uma uma grande diferença entre os três modos, provando assim, que o Heap Sort é um ótima opção para se trabalhar com grande quantidade de dados e números a serem tratados.

5.8 *Análise Geral*

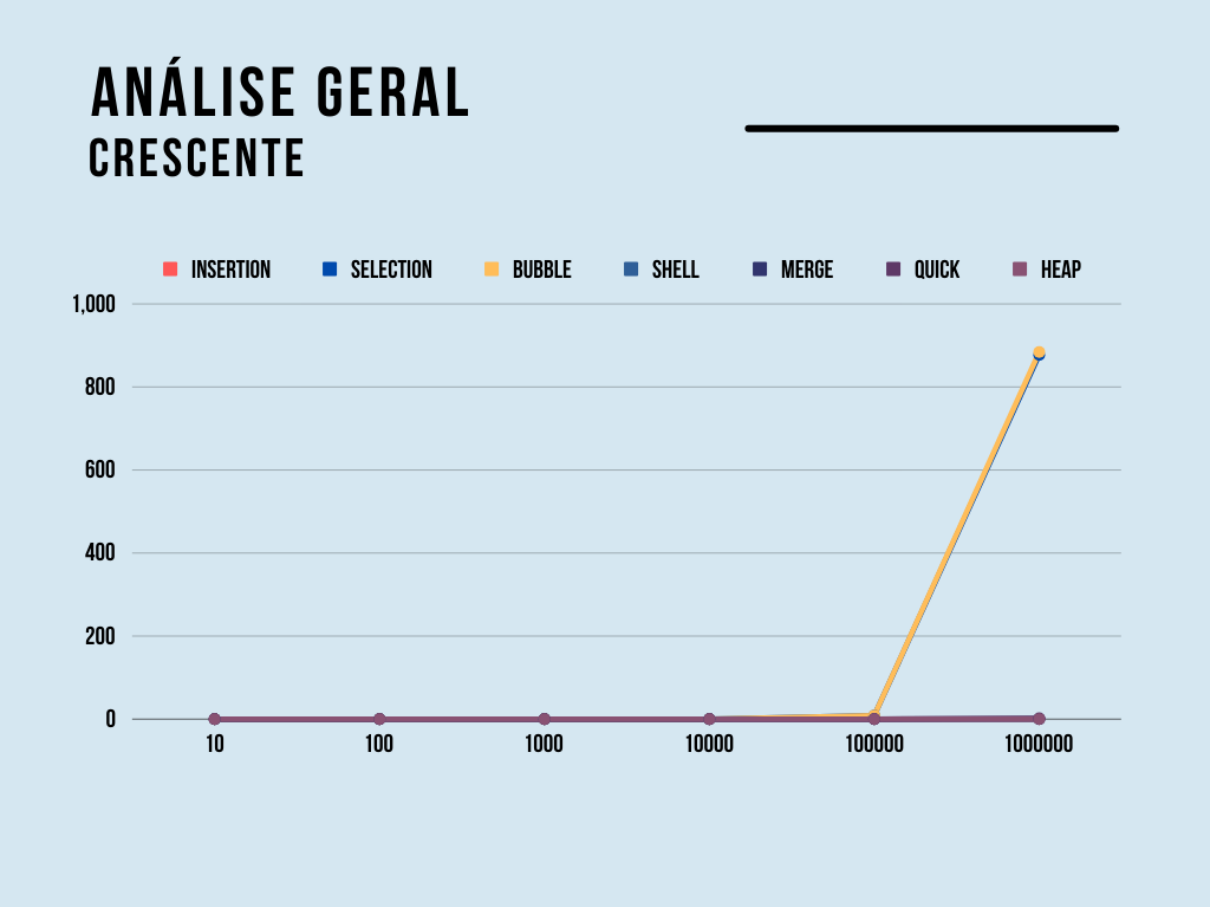


Figura 19: Gráfico de todos algoritmos no modo Crescente

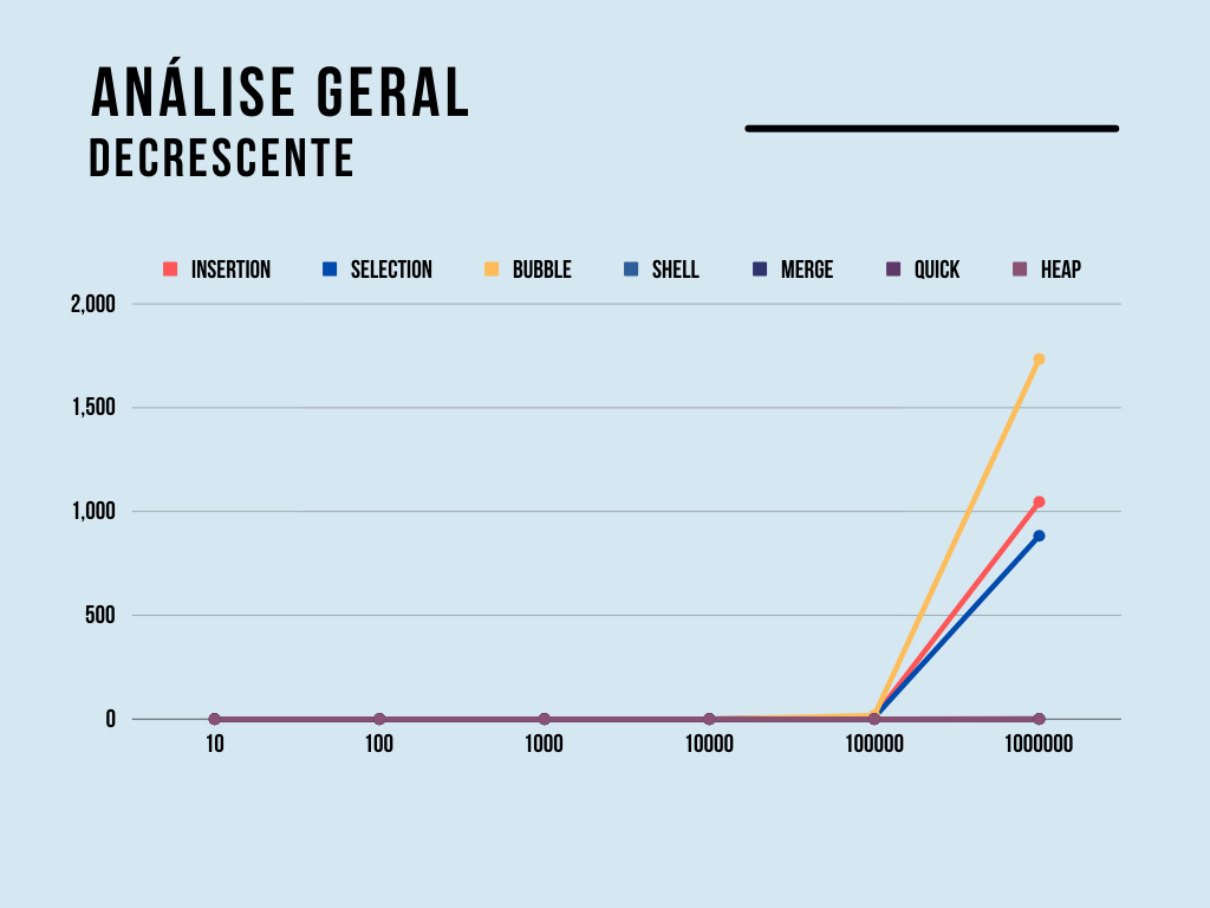


Figura 20: Gráfico de todos algoritmos no modo Decrescente

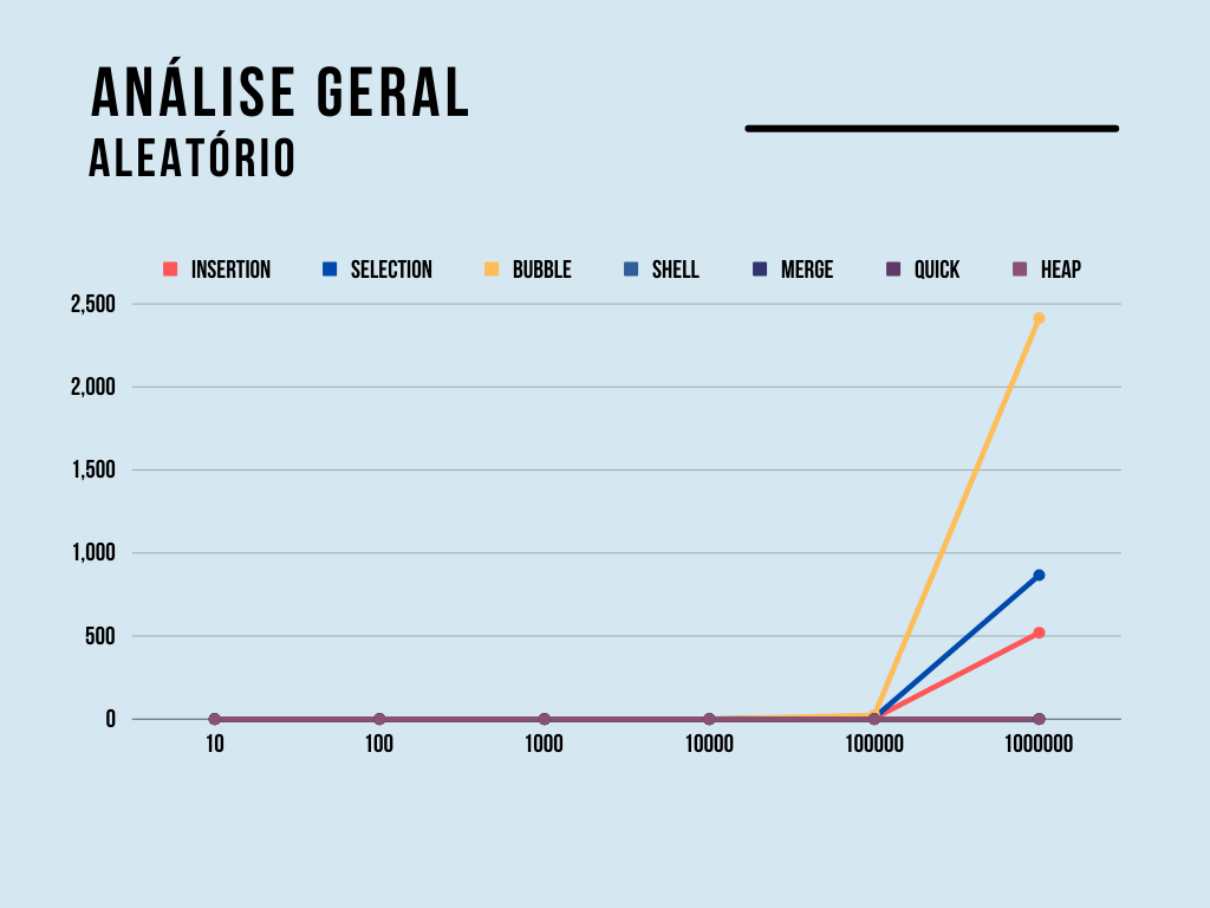


Figura 21: Gráfico de todos algoritmos no modo Aleatório

Fazendo uma análise de todos os algortimos testados e analísados em sua complexidade, podemos observar de acordo com o 3 gráficos acima *(Modo Crescente, Decrescente e Aleatório)* que os pontos postivos e negativos influenciam diretamente no desenvolvimento dos algortimos expostos neste estudo.

Os Algortimos iníciais como o Insertion, Selection e Bubble, tem facilidades em ordenações com pequenos conjuntos de dados, porém, ao chegar em indicíes mais elevados de ordenação com cem mil ou 1 milhão de elementos, vemos que estes algortimos possuem difículdade em entregar um ordenação rápida de tal aranjo, isto se deve a complexidade destes algortimos e também em sua construção.

Já os algortimos como Heap, Shell, Merge e Quick (em todas as suas versões de seleção de pivô), vemos que tem maior fácilidade em ordenação em uma carga de alto indíce de elementos a serem ordenados, pois utilizam técnicas mais complexas em seu desenvolvimento que possibilita assim ter um melhor tempo de resposta e também de otimização em suma sobre a tarefa designada.

Todos os algortimos de ordenação tem suas vantagens e desvantagens, e sempre possuem ensinamentos para todos aqueles que estão avaliando e estudando suas complexidades e códigos, ter um conhecimento deles é suma importância para poder aprofundar os conhecimentos em algortimos de ordenação.

6 CONCLUSÃO

Analisando a complexidade dos algoritmos de ordenação, vemos que estes algoritmos desempenham e tem um papel fundamental na compreensão da ordenação e fornecem também uma base sólida para a estudos de algortimos mais complexos.

O *Insertion Sort* é um algoritmo simples e intuitivo, adequado para pequenos conjuntos de dados. Sua complexidade é O(n2) no pior caso, tornando-o ineficiente para grandes conjuntos de dados desordenados. No entanto, ele brilha em conjuntos de dados quase ordenados, onde seu desempenho se aproxima de O(n).

O *Selection Sort* é um algoritmo simples que tem uma complexidade de O(n2) no pior caso, independentemente da distribuição dos dados. Apesar de sua simplicidade, é menos eficiente em comparação com outros algoritmos de ordenação e é raramente usado em cenários de produção para grandes conjuntos de dados.

O *Bubble Sort*, embora seja um dos algoritmos mais simples de entender e implementar, possui uma complexidade de O(n2) no pior caso e é geralmente menos eficiente do que o Insertion Sort e o Selection Sort. Assim como o Selection Sort, é raramente utilizado em cenários de produção para ordenação de grandes volumes de dados.

O *Shell Sort*, por outro lado, representa uma melhoria significativa em relação aos algoritmos de ordenação anteriores. Com a escolha adequada das lacunas (gaps), o Shell Sort pode alcançar uma complexidade média melhor do que O(n2), tornando-o uma escolha viável para ordenação em larga escala. Embora sua análise matemática seja complexa, sua eficácia prática é evidente em muitas implementações de software.

O *Merge Sort*, reconhecido pela sua eficiência, revela resultados notáveis em diferentes cenários. Com uma complexidade de tempo estável, especialmente para conjuntos de dados extensos, destaca-se em situações aleatórias e mantém um desempenho consistente ao lidar com conjuntos já ordenados.

O *Quick Sort* é conhecido por sua eficiência ao escolher pivôs estrategicamente, proporcionando rapidez em cenários diversos de ordenação. Enquanto isso, o Merge Sort destaca-se pela estabilidade e consistência, sendo eficaz em conjuntos de dados de diferentes padrões. Ambos os algoritmos são amplamente utilizados devido às suas características específicas e

desempenho notável.

O *Heap Sort*, utiliza Heaps para fazer a ordenação de forma eficiente, principalmente para altos volumes de dados, tendo uma resposta melhor e mais rápida nestes casos, podemos observar que, o Heap Sort, tem uma complexidade maior em sua construção e desenvolvimento com seus códigos, sendo assim, mais complexo que algortimos aqui desenvolvidos como o Insertion Sort, Bubble Sort e outros.

A escolha do algoritmo de ordenação depende das características da utilização dos requisitos específicos do problema proposto. Os Algoritmos mais simples, como Insertion Sort, Selection Sort e Bubble Sort, podem ser úteis para pequenas sequências de dados ou quando o desempenho não é uma prioridade. No entanto, para ordenação eficiente de grandes volumes de dados, o Heap Sort, Merge Sort e Quick Sort tem como uma opção mais eficiente, sendo algoritmos mais fortes para um valor mais expressivo de dados.

...

7 REFERÊNCIAS BIBLIOGRÁFICAS

https://www.blogcyberini.com/2018/06/insertion-sort.html:

https://www.facom.ufu.br/albertini/1sem2018/ada/aulas/02analiseAssintotica.pdf

https://www.geeksforgeeks.org/bubble-sort/

https://docente.ifrn.edu.br/robinsonalves/disciplinas/estruturas-de-dados/ordenacao.pdf

https://docente.ifrn.edu.br/robinsonalves/disciplinas/estruturas-de-dados/ordenacao.pdf

https://sca.profmat-sbm.org.br/profmattcc.php?id1=5511id2=171053345

https://homepages.dcc.ufmg.br/ cunha/teaching/20121/aeds2/shellsort.pdf

https://www.scholarhat.com/tutorial/datastructures/shell-sort-in-data-structures

https://www.scielo.br/j/tema/a/G7Ybrdy6w4K6gFYRvSj4tft/?lang=pt

https://www.blogcyberini.com/2018/07/merge-sort.html

https://www.ime.usp.br/ pf/algoritmos/aulas/hpsrt.html

CORMEN, T. H. et al. Algoritmos: teoria e prática. 3 ed. Rio de Janeiro: Elsevier, 2012.

Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 5.2.1: Sorting by Insertion, pp. 80–105.

Michael T Goodrich, Algorithm Desing: Foundations, Analysis and Internet Examples, Second Edition. 2002, ISBN 0-471-38365-1. Section 4.4.6; Comparison of Sorting Algorithms, pp. 244-245.

Robert Sedgewick, Philippe Flajolet, An Introduction to the Analysis of Algorithms, Second Edition, Pearson Education, 2013. ISBN-13: 978-0-321-90575-8. Section 7.6: Inversions and Insertion Sorts, pp. 384-388.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein, "Introduction to Algorithms", 3ª Edição, MIT Press, 2009.

R. Souza, F. Oliveira e P. Pinto. Um Limite Superior para a Complexidade do ShellSort. In "III Encontro de Teoria da Computação". SBC, Natal, Brasil (2018), pp. 49–52.

N. Tokuda. An improved Shellsort. Anais do 12º IFIP World Computer Congress on Algorithms, Software, Architecture-Information Processing, 1 (1992), 449–457.

AZEREDO, Paulo A. (1996). Métodos de Classificação de Dados e Análise de suas Complexidades. Rio de Janeiro: Campus. ISBN 85-352-0004-5